

Effective

UNIT TESTING

A guide for Java developers



 MANNING

LASSE KOSKELA

Effective Unit Testing

Effective Unit Testing

A GUIDE FOR JAVA DEVELOPERS

LASSE KOSKELA



MANNING
SHELTER ISLAND

For online information and ordering of this and other Manning books, please visit www.manning.com. The publisher offers discounts on this book when ordered in quantity. For more information, please contact


Special Sales Department
Manning Publications Co.
20 Baldwin Road
PO Box 261
Shelter Island, NY 11964
Email: orders@manning.com

©2013 by Manning Publications Co. All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by means electronic, mechanical, photocopying, or otherwise, without prior written permission of the publisher.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in the book, and Manning Publications was aware of a trademark claim, the designations have been printed in initial caps or all caps.

© Recognizing the importance of preserving what has been written, it is Manning's policy to have the books we publish printed on acid-free paper, and we exert our best efforts to that end. Recognizing also our responsibility to conserve the resources of our planet, Manning books are printed on paper that is at least 15 percent recycled and processed without the use of elemental chlorine.

 Manning Publications Co.
20 Baldwin Road
PO Box 261
Shelter Island, NY 11964

Development editor: Frank Pohlman
Copyeditor: Benjamin Berg
Technical proofreader: Phil Hanna
Proofreader: Elizabeth Martin
Typesetter: Dottie Marsico
Cover designer: Marija Tudor

ISBN 9781935182573
Printed in the United States of America
2 3 4 5 6 7 8 9 10 – DP – 18

*Few sights are as captivating as the pure joy
of learning new things.*

brief contents

PART 1	FOUNDATIONS	1
1	■ The promise of good tests	3
2	■ In search of good	15
3	■ Test doubles	27
PART 2	CATALOG.....	45
4	■ Readability	47
5	■ Maintainability	78
6	■ Trustworthiness	115
PART 3	DIVERSIONS	137
7	■ Testable design	139
8	■ Writing tests in other JVM languages	156
9	■ Speeding up test execution	170

contents

preface xv
acknowledgments xvii
about this book xix
about the cover illustration xxiv

PART 1 FOUNDATIONS.....1

1 *The promise of good tests* 3

- 1.1 State of the union: writing better tests 4
- 1.2 The value of having tests 5
 - Factors of productivity* 8 ■ *The curve of design potential* 10
- 1.3 Tests as a design tool 10
 - Test-driven development* 11 ■ *Behavior-driven development* 13
- 1.4 Summary 14

2 *In search of good* 15

- 2.1 Readable code is maintainable code 16
- 2.2 Structure helps make sense of things 18
- 2.3 It's not good if it's testing the wrong things 20
- 2.4 Independent tests run easily in solitude 21

- 2.5 Reliable tests are reliable 23
- 2.6 Every trade has its tools and tests are no exception 25
- 2.7 Summary 26

3 *Test doubles* 27

- 3.1 The power of a test double 28
 - Isolating the code under test* 28 ■ *Speeding up test execution* 30
 - Making execution deterministic* 31 ■ *Simulating special conditions* 32 ■ *Exposing hidden information* 32
- 3.2 Types of test doubles 33
 - Test stubs are unusually short things* 34 ■ *Fake objects do it without side effects* 35 ■ *Test spies steal your secrets* 36
 - Mock objects object to surprises* 38
- 3.3 Guidelines for using test doubles 39
 - Pick the right double for the test* 40 ■ *Arrange, act, assert* 40
 - Check for behavior, not implementation* 42 ■ *Choose your tools* 42 ■ *Inject your dependencies* 43
- 3.4 Summary 44

PART 2 CATALOG45

4 *Readability* 47

- 4.1 Primitive assertions 48
 - Example* 48 ■ *What to do about it?* 49 ■ *Summary* 51
- 4.2 Hyperassertions 51
 - Example* 51 ■ *What to do about it?* 54 ■ *Summary* 55
- 4.3 Bitwise assertions 56
 - Example* 56 ■ *What to do about it?* 56 ■ *Summary* 57
- 4.4 Incidental details 57
 - Example* 57 ■ *What to do about it?* 58 ■ *Summary* 60
- 4.5 Split personality 60
 - Example* 61 ■ *What to do about it?* 61 ■ *Summary* 64
- 4.6 Split logic 64
 - Example* 65 ■ *What to do about it?* 66 ■ *Summary* 69
- 4.7 Magic numbers 70
 - Example* 70 ■ *What to do about it?* 71 ■ *Summary* 71

4.8	Setup sermon	72
	<i>Example</i>	72 ■ <i>What to do about it?</i> 73 ■ <i>Summary</i> 74
4.9	Overprotective tests	75
	<i>Example</i>	75 ■ <i>What to do about it?</i> 76 ■ <i>Summary</i> 76
4.10	Summary	76

5 Maintainability 78

5.1	Duplication	79
	<i>Example</i>	79 ■ <i>What to do about it?</i> 80 ■ <i>Summary</i> 82
5.2	Conditional logic	82
	<i>Example</i>	83 ■ <i>What to do about it?</i> 83 ■ <i>Summary</i> 84
5.3	Flaky test	85
	<i>Example</i>	85 ■ <i>What to do about it?</i> 86 ■ <i>Summary</i> 87
5.4	Crippling file path	88
	<i>Example</i>	88 ■ <i>What to do about it?</i> 89 ■ <i>Summary</i> 90
5.5	Persistent temp files	91
	<i>Example</i>	91 ■ <i>What to do about it?</i> 92 ■ <i>Summary</i> 93
5.6	Sleeping snail	94
	<i>Example</i>	94 ■ <i>What to do about it?</i> 95 ■ <i>Summary</i> 96
5.7	Pixel perfection	97
	<i>Example</i>	97 ■ <i>What to do about it?</i> 98 ■ <i>Summary</i> 101
5.8	Parameterized mess	102
	<i>Example</i>	103 ■ <i>What to do about it?</i> 106 ■ <i>Summary</i> 108
5.9	Lack of cohesion in methods	108
	<i>Example</i>	109 ■ <i>What to do about it?</i> 110 ■ <i>Summary</i> 112
5.10	Summary	113

6 Trustworthiness 115

6.1	Commented-out tests	116
	<i>Example</i>	116 ■ <i>What to do about it?</i> 117 ■ <i>Summary</i> 118
6.2	Misleading comments	118
	<i>Example</i>	118 ■ <i>What to do about it?</i> 119 ■ <i>Summary</i> 120
6.3	Never-failing tests	121
	<i>Example</i>	121 ■ <i>What to do about it?</i> 122 ■ <i>Summary</i> 123

- 6.4 Shallow promises 123
 - Example(s)* 123 ▪ *What to do about it?* 125 ▪ *Summary* 126
- 6.5 Lowered expectations 127
 - Example* 127 ▪ *What to do about it?* 128 ▪ *Summary* 129
- 6.6 Platform prejudice 129
 - Example* 129 ▪ *What to do about it?* 130 ▪ *Summary* 132
- 6.7 Conditional tests 133
 - Example* 133 ▪ *What to do about it?* 134 ▪ *Summary* 135
- 6.8 Summary 135

PART 3 DIVERSIONS..... 137

7 Testable design 139

- 7.1 What's testable design? 140
 - Modular design* 140 ▪ *SOLID design principles* 141
 - Modular design in context* 143 ▪ *Test-driving toward modular design* 143
- 7.2 Testability issues 143
 - Can't instantiate a class* 144 ▪ *Can't invoke a method* 144
 - Can't observe the outcome* 145 ▪ *Can't substitute a collaborator* 145 ▪ *Can't override a method* 145
- 7.3 Guidelines for testable design 146
 - Avoid complex private methods* 146 ▪ *Avoid final methods* 147
 - Avoid static methods* 148 ▪ *Use new with care* 148 ▪ *Avoid logic in constructors* 149 ▪ *Avoid the Singleton* 150 ▪ *Favor composition over inheritance* 151 ▪ *Wrap external libraries* 152
 - Avoid service lookups* 152
- 7.4 Summary 154

8 Writing tests in other JVM languages 156

- 8.1 The premise of mixing JVM languages 157
 - General benefits* 157 ▪ *Writing tests* 159
- 8.2 Writing unit tests with Groovy 160
 - Simplified setup for tests* 161 ▪ *Groovier JUnit 4 tests* 163

8.3 Expressive power with BDD tools 163

Groovy specs with easyb 164 ▪ *Spock Framework: steroids for writing more expressive tests* 165 ▪ *Spock Framework's test doubles are on steroids, too* 167

8.4 Summary 168

9 *Speeding up test execution* 170

9.1 Looking for a speed-up 171

The need for speed 171 ▪ *Approaching the situation* 172
Profiling a build 172 ▪ *Profiling tests* 175

9.2 Speeding up test code 178

Don't sleep unless you're tired 178 ▪ *Beware the bloated base class* 179 ▪ *Watch out for redundant setup and teardown* 181
Be picky about who you invite to your test 182 ▪ *Stay local, stay fast* 183 ▪ *Resist the temptation to hit the database* 184
There's no slower I/O than file I/O 186

9.3 Speeding up the build 187

Faster I/O with a RAM disk 188 ▪ *Parallelizing the build* 189
Offload to a higher-powered CPU 194 ▪ *Distribute the build* 196

9.4 Summary 200

appendix A JUnit primer 202

appendix B Extending JUnit 209
index 217

preface

On the night of June 10, 2009, I found an email in my inbox from Christina Rudloff from Manning, asking me if I knew anyone who might be a good candidate to write a Java edition of Roy Osherove's book, *The Art of Unit Testing in .NET*. I told her I'd do it.

That was a long time ago and what you're looking at right now has very little in common with Roy's book. Let me explain.

The project started as a straight translation from .NET to Java, only rewriting where necessary to match the changing technology platform, its tooling, and its audience. I finished the first chapter, the second chapter, the third chapter, and suddenly I found myself rewriting not just short passages but entire chapters. The tone of voice wasn't mine; sometimes I would disagree or have preferences incompatible with Roy's, and sometimes I simply felt strongly about saying something, setting things straight, and putting a stake into the ground.

Eventually, I decided to start over.

It was clear that we were not in a translation project. This was a brand new title of its own—a book that helps a Java programmer improve his tests, gaining a deeper insight into what makes a test good and what kind of pitfalls to look out for. You can still see Roy's thinking in many ways in this book. For instance, the chapter titles of the catalog in part 2 I've blatantly borrowed from Roy and chapter 7 was written largely thanks to Roy's counterpart in *The Art of Unit Testing in .NET*.

This is a book for the Java programmer. Yet, I didn't want to straitjacket the ideas in this book artificially, so I tried to steer away from being too language-specific even though all of the code examples in the pattern catalog, for example, are Java. Writing good tests is a language-agnostic problem and I heartily recommend you read this

book thoughtfully, even if you happen to spend most of your office hours hacking on another programming language.

Along those same lines, I didn't want to give you a tutorial on JUnit or my favorite mock object library. Aside from such technology being a constantly changing landscape and bound to become stale information within months of publication, I wanted to write the kind of book that I would want to read. I like focused books that don't force me to lug around dead weight about a testing framework I already know by heart or a mock object library I don't use. For these reasons, I've tried to minimize the amount of technology-specific advice. There is some but I want you to know that I've done my best to keep it to a minimum—just enough to have meaningful conversations about the underlying concepts that I find essential in writing, running, maintaining, and improving tests.

I tried to write the book I would've wanted to read. I hope you will enjoy it and, most importantly, integrate some of these ideas into your own practice.

acknowledgments

When I signed up to write this book I thought it'd be a short project. Everything was supposed to be straightforward with no wildcards in sight. I should've known better. My wishful thinking was shattered as weeks turned to months and months turned to years. Without the help of many, many people this book would definitely not be in your hands and most likely it'd still be a work-in-progress.

From the moment this project was initiated in a casual email exchange with Manning Publication's Christina Rudloff, a massive amount of help has poured in and all of it has been very much appreciated—and needed.

I'd like to thank the team at Manning for their support and persistence. In no specific order, Michael Stephens, Elle Suzuki, Steven Hong, Nick Chase, Karen Tegtmeyer, Sebastian Stirling, Candace M. Gillhoolley, Maureen Spencer, Ozren Harlovic, Frank Pohlmann, Benjamin Berg, Elizabeth Martin, Dottie Marsico, Janet Vail, and Mary Piergies.

A special thanks goes to the fine individuals that served as domain experts or reviewers and contributed their time to put their specific experience and expertise to improving this book. Again, in no specific order, I'd like to extend my most sincere gratitude to Jeremy Anderson, Christopher Bartling, Jedidja Bourgeois, Kevin Conaway, Roger Cornejo, Frank Crow, Chad Davis, Gordon Dickens, Martyn Fletcher, Paul Holser, Andy Kirsch, Antti Koivisto, Paul Kronquist, Teppo Kurki, Franco Lombardo, Saicharan Manga, Dave Nicolette, Gabor Paller, J. B. Rainsberger, Joonas Reynders, Adam Taft, Federico Tomassetti, Jacob Tomaw, Bas Vodde, Deepak Vohra, Rick Wagner, Doug Warren, James Warren, Robert Wenner, Michael Williams, and Scott Sauyet.

Special thanks to Phil Hanna for his technical review of the manuscript just before it went into production.

And last, but definitely not least, I'd like to thank my family for their continued support. I imagine it has at times felt like a never-ending endeavor to get this book to print. Thank you for understanding all of those late nights with a computer on my lap and for carrying me through the rough spots.

about this book

Developer testing has been increasing its mindshare significantly among Java developers over the past 10 years or so. Today, no computer science student graduates without having at least read about automated unit tests and their importance in software development. The idea is simple—to ensure that our code works and keeps working—but the skill takes significant effort to learn.

Some of that effort goes to simply writing tests and learning the technology such as a test framework like JUnit. Some of that effort (and quite possibly most of it) that's required for truly mastering the practice of writing automated unit tests goes to reading test code and improving it. This constant refactoring of tests—trying out different ways of expressing your intent, structuring tests for various aspects of behavior, or building the various objects used by those tests—is our pragmatic way of teaching ourselves and developing our sense for unit tests.

That sense is as much about what good unit tests are like as it is about what not-so-good unit tests are like. There may be some absolute truths involved (such as that a code comment repeating exactly what the code says is redundant and should be removed) but the vast majority of the collective body of knowledge about unit tests is highly context-sensitive. What is generally considered good might be a terrible idea in a specific situation. Similarly, what is generally a bad idea and should be avoided can sometimes be just the right thing to do.

It turns out that often the best way to find your way to a good solution is to try one approach that seems viable, identify the issues with that approach, and change the approach to remove the icky parts. By repeating this process of constantly evaluating and evolving what you have, eventually you reach a solution that works and doesn't smell all that bad. You might even say that it's a pretty good approach!

With this complexity in mind, we've adopted a style and structure for this book where we don't just tell you what to do and how to write unit tests. Instead, we aim to give you a solid foundation on what kind of properties we want our tests to exhibit (and why) and then give you as many concrete examples as we can to help you develop your sense for test smells—to help you notice when something about your test seems to be out of place.

Audience

This book is aimed at Java programmers of all experience levels who are looking to improve the quality of the unit tests they write. While we do provide appendices that teach you about a test framework (JUnit), our primary goal is to help Java programmers who already know how to write unit tests with their test framework of choice to write better unit tests. Regardless of how many unit tests you've written so far, we're certain that you can still get better at it, and reading a book like this might be just what you need to stimulate a line of thought that you've struggled to put into words.

Roadmap

Effective Unit Testing takes on a multifaceted challenge that calls for a structure that supports each of those facets. In our wisdom (gained through several iterations of failed attempts) we've decided to divide this book into three parts.

Part 1 begins our journey toward better tests by introducing what we're trying to achieve and why those goals should be considered desirable in the first place. These three chapters present the fundamental tools and simple guidelines for writing a good test.

Chapter 1 starts off with the value proposition of automated unit tests. We establish the value by considering the many things that factor into our productivity as programmers and how well-written automated unit tests contribute to that productivity or prevent things from dragging us down.

Chapter 2 sets the bar high and attempts to define what makes a test good. The properties and considerations in this chapter serve as the core foundation for part 2, touching on how we want our tests to be readable, maintainable, and reliable.

Chapter 3 steps out of the line for a moment to introduce test doubles as an essential tool for writing good tests. It's not really using test doubles that we're after but rather using them well and with consideration. (They're not a silver bullet in case you were wondering.)

Part 2 turns the tables and offers a stark contrast to part 1 in its approach, presenting a catalog of test smells you should watch for. Along with describing a suspect pattern in test code we'll suggest solutions to try when you encounter such a smell. The chapters in this part are divided into three themes: smells that suggest degraded readability, smells that indicate a potential maintenance nightmare, and smells that reek of trust issues. Many of the smells in part 2 could be featured in any of these three chapters, but we've tried to arrange them according to their primary impact.

Chapter 4 focuses on test smells that are primarily related to the intent or implementation of a test being unnecessarily opaque. We touch on things like illegible assertions, inappropriate levels of abstraction, and information scatter within our test code.

Chapter 5 walks through test smells that might lead to late nights at the office, because it takes forever to update one mess of a unit test related to a small change or because making that small change means we need to change a hundred tests. We take on code duplication and logic in our test code and we expound on the horrors of touching the filesystem. And it's not like we're giving a free pass to slow tests either because time is money.

Chapter 6 concludes our catalog of test smells with a sequence of gotchas around assumptions. Some of these assumptions are made because there's an inconvenient comment in our test code and some are the unfortunate products of a failure to express ourselves unambiguously.

Part 3 could have been called "advanced topics." It's not, however, because the topics covered here don't necessarily build on parts 1 or 2. Rather, these are topics that a Java programmer might stumble onto at any point on his or her test-writing journey. After all, almost everything about "good" unit tests is context-sensitive so it's not surprising that a pressing topic for one programmer is a minor consideration for another, whether it's about inheritance between unit test classes, about the programming language we use for writing tests, or about the way our build infrastructure executes the tests we've written.

Chapter 7 picks up on where chapter 2 left off, exploring what constitutes testable design. After a brief overview of useful principles and clarifying how we are essentially looking for modular designs, we study the fundamental testability issues that untestable designs throw our way. The chapter concludes with a set of simple guidelines to keep us on the righteous path of testable design.

Chapter 8 throws a curveball by posing the question, what if we'd write our unit tests in a programming language other than Java? The Java Virtual Machine allows the modern programmer to apply a number of alternative programming languages and integrate it all with plain Java code.

Chapter 9 returns to common reality by taking on the challenge of dealing with increasingly slow build times and delayed test results. We look for solutions both within our test code, considering ways of speeding up the code that's being run as part of our build, and in our infrastructure, pondering whether we could get that extra bit of oomph from faster hardware or from a different way of allocating work to the existing hardware.

Despite JUnit's popularity and status as the de facto unit test framework within the Java community, not all Java programmers are familiar with this great little open source library. We've included two appendices to help those individuals and programmers who haven't squeezed all the horsepower out of JUnit's more advanced features.

Appendix A offers a brief introduction to writing tests with JUnit and how JUnit pokes and prods those tests when you tell it to run them. After skimming through this

appendix you'll be more than capable of writing tests and making assertions with JUnit's API.

Appendix B takes a deeper dive into the JUnit API with the goal of extending its built-in functionality. While not trying to cover everything about JUnit to the last bit of detail, we've chosen to give you a brief overview of the two common ways of extending JUnit—rules and runners—and devote this appendix to showcasing some of the built-in rules that are not only useful, but also give you an idea of what you can do with your custom extensions.

Code conventions

The code examples presented in this book consist of Java source code as well as a host of markup languages and output listings. We present the longer pieces of code as listings with their own headers. Smaller bits of code are run inline with the text. In all cases, we present the code using a monospaced font like this, to differentiate it from the rest of the text. We frequently refer to elements in code listings taken from the text. Such references are also presented using a monospaced font, to make them stand out. Many longer listings have numbered annotations that we refer to in the text.

Code downloads

The Manning website page for this book at www.manning.com/EffectiveUnitTesting offers a source code package you can download to your computer. This includes selected parts of the source code shown in the book, should you want to take things further from where we left off.

The download includes an Apache Maven 2 POM file and instructions for installing and using Maven (<http://maven.apache.org>) to compile and run the examples. Note that the download doesn't include the various dependencies, and you need to have an internet connection when running the Maven build for the first time—Maven will then download all the required dependencies from the internet. After that, you're free to disconnect and play with the examples offline.

The code examples were written against Java 6, so you'll need to have that installed in order to compile and run the examples. You can download a suitable Java environment from www.oracle.com. (To compile the code, you'll need to download the JDK, not the JRE.)

We recommend installing a proper IDE as well. You may want to download and install the latest and greatest version of Eclipse (www.eclipse.org) or another mainstream tool like IntelliJ IDEA (www.jetbrains.com) or NetBeans (www.netbeans.org). All of these should work fine as long as you're familiar with the tool.

What's next?

This book should give you enough insight to start visibly improving your unit tests. It's going to be a long journey, and there's bound to be a question or two that we haven't

managed to predict or answer in full. Fortunately, you'll have plenty of peers on this journey and many of them will be more than happy to share and discuss the nuances of test code online.

Manning has set up an online forum where you can talk to the authors of Manning titles. That includes the book you're reading right now so head over to the Author Online forum for this book at www.manning.com/EffectiveUnitTesting.

There's also an active community of test-infected programmers over at the *testdrivendevelopment* and *extremeprogramming* Yahoo! Groups. While these forums aren't exclusively for discussions about unit tests, they are excellent places for holding those discussions. Besides, maybe you'll manage to pick up some new ideas outside of test code, too.

If you're looking for a more focused forum for having discussions about developer testing, head over to the CodeRanch at <http://www.coderanch.com> and the excellent Testing forum. Again, a lovely group of very helpful people over there.

Most importantly, however, I suggest that you actively talk about your test code with your peers at work. Some of the best insights I've had about my code have been through having someone else look at it on the spot.

Author Online

Purchase of *Effective Unit Testing* includes free access to a private web forum run by Manning Publications, where you can make comments about the book, ask technical questions, and receive help from the author and from other users. To access the forum and subscribe to it, point your web browser to <http://www.manning.com/EffectiveUnitTesting>. This page provides information on how to get on the forum once you are registered, what kind of help is available, and the rules of conduct on the forum.

Manning's commitment to our readers is to provide a venue where a meaningful dialogue between individual readers and between readers and the author can take place. It is not a commitment to any specific amount of participation on the part of the author, whose contribution to the book's forum remains voluntary (and unpaid). We suggest you try asking the author some challenging questions, lest his interest stray!

The Author Online forum and the archives of previous discussions will be accessible from the publisher's website as long as the book is in print.

about the cover illustration

The figure on the cover of *Effective Unit Testing* is captioned “A Man from Drnis, Dalmatia, Croatia.” The illustration is taken from a reproduction of an album of Croatian traditional costumes from the mid-nineteenth century by Nikola Arsenovic, published by the Ethnographic Museum in Split, Croatia, in 2003. The illustrations were obtained from a helpful librarian at the Ethnographic Museum in Split, itself situated in the Roman core of the medieval center of the town: the ruins of Emperor Diocletian’s retirement palace from around AD 304. The book includes finely colored illustrations of figures from different regions of Croatia, accompanied by descriptions of the costumes and of everyday life.

Drnis is a small town in inland Dalmatia, built on the ruins of an old medieval fortress. The figure on the cover is wearing blue woolen trousers and, over a white linen shirt, a blue woolen vest which is richly trimmed with the colorful embroidery typical for this region. He is carrying a long pipe and a burgundy jacket is slung over his shoulder. A red cap and leather moccasins complete the outfit.

Dress codes and lifestyles have changed over the last 200 years, and the diversity by region, so rich at the time, has faded away. It is now hard to tell apart the inhabitants of different continents, let alone of different hamlets or towns separated by only a few miles. Perhaps we have traded cultural diversity for a more varied personal life—certainly for a more varied and fast-paced technological life.

Manning celebrates the inventiveness and initiative of the computer business with book covers based on the rich diversity of regional life of two centuries ago, brought back to life by illustrations from old books and collections like this one.

Part 1

Foundations

This first part of the book aims to give you, the reader, and me, the author, a shared context to build upon throughout the chapters. With the ultimate purpose of this book being to help you improve your ability to write good tests, chapter 1 begins with an overview of what kind of value we can extract from writing tests in the first place. Once we've discussed the dynamics of programmer productivity and the kind of impact that our tests—and the quality of our tests—can have on it, we'll conclude the chapter with a brief introduction to two methods that are closely related to automated tests: *test-driven development* (TDD) and *behavior-driven development* (BDD).

Chapter 2 takes on the challenge of defining what makes for a good test. In short, we'd like to write tests that are readable, maintainable, and reliable. Part 2 will go deeper into this rabbit hole by turning the question around and reviewing a collection of examples of what we *don't* want to get.

Part 1 concludes with chapter 3, which tackles one of the most essential tools in the modern programmer's tool belt—test doubles. We'll establish the legitimate uses of test doubles, such as isolating code so that it can be tested properly, and we'll make a distinction between the types of test doubles we might reach out to. Finally, we'll throw in guidelines for making good use of test doubles to help you get the benefits without stumbling on common pitfalls.

After reading these first three chapters you should have a good idea of what kind of tests you want to write and why. You should also have a clear understanding of test doubles as a useful vehicle for getting there. The rest of this book will then build on this foundation and give you more ammunition for your journey in the real world.

The promise of good tests

In this chapter

- The value of having unit tests
- How tests contribute to a programmer's productivity
- Using tests as a design tool

When I started getting paid for programming, the world looked a lot different. This was more than 10 years ago and people used simple text editors like Vim and Emacs instead of today's integrated development environments like Eclipse, NetBeans, and IDEA. I vividly remember a senior colleague wielding his Emacs macros to generate tons of calls to `System.out.println` as he was debugging our software. Even more vivid are my memories of deciphering the logs those printouts ended up in after a major customer had reported that their orders weren't going through like they should.

That was a time when “testing” for most programmers meant one of two things—the stuff that somebody else does after I'm done coding, or the way you run and poke your code before you say you're done coding. And when a bug did slip through, you'd find yourself poking and prodding your code again—this time adding a few more logging statements to see if you could figure out where things went wrong.

Automation was a state-of-the-art concept for us. We had makefiles to compile and package our code in a repeatable manner, but running automated tests as part of the build wasn't quite in place. We did have various shell scripts that launched one or two "test classes"—small applications that operated our production code and printed what was happening and what our code was returning for what kind of input. We were far from the kind of standard testing frameworks and self-verifying tests that report all failures in our assertions.

We've come a long way since those days.

1.1 *State of the union: writing better tests*

Today, it's widely recommended that developers write automated tests that fail the build when there are regressions. Furthermore, an increasing number of professionals is leaning on a test-first style of programming, using automated tests not for protecting against regression but for aiding them in design, specifying the behavior they expect from code before writing that code, thereby validating a design before verifying its implementation.

Being a consultant, I get to see a lot of teams, organizations, products, and code bases. Looking at where we are today, it's clear that automated tests have made their way into the mainstream. This is good because without such automated tests, most software projects would be in a far worse situation than they are today. Automated tests improve your productivity and enable you to gain and sustain development speed.

Help! I'm new to unit testing

If you aren't that familiar with writing automated tests, this would be a good time to get acquainted with that practice. Manning has released several books on JUnit, the de facto standard library for writing unit tests for Java, and the second edition of *JUnit in Action* (written by Petar Tahchiev, et al. and published in July 2010) is a good primer for writing tests for all kinds of Java code, from plain old Java objects to Enterprise JavaBeans.

In case you're at home with writing unit tests but you're new to Java or JUnit, perhaps all you need to get the most out of this book is to first check out appendix A, so that you won't have trouble reading the examples.

Automated tests being mainstream doesn't mean that our test coverage is as high as it should be or that our productivity couldn't improve. In fact, a significant part of my work in the last five years or so has revolved around helping people write tests, write tests before code, and especially write *better* tests.

Why is it so important to write better tests? What'll happen if we don't pay attention to the quality of our tests? Let's talk about what kind of value tests give us and why the quality of those tests matters.

1.2 The value of having tests

Meet Marcus. Marcus is a prominent programming wiz who graduated two years ago and joined the IT group of a local investment bank, where he's been developing the bank's online self-service web application. Being the youngest member of the team, he kept a low profile at first and focused his energy on learning about the banking domain and getting to know "the way things are done here."

A few months on, Marcus started noticing that a lot of the team's work was *rework*: fixing programmer mistakes.¹ He started paying attention to the types of mistakes the team was fixing and noticed that most of them would've been caught fairly easily by unit tests. Marcus started writing unit tests here and there when he felt that the code was particularly prone to having mistakes in it.

Tests help us catch mistakes.

Time went on and the rest of the team was writing unit tests, too, here and there. Marcus had become test-infected and rarely left a piece of code that he touched without fairly good coverage from automated tests.² They weren't spending any more time fixing errors than they had before, but for the first time, their total number of open defects was on the decline. The tests started having a clearly visible impact on the quality of the team's work.

Almost a year had passed since Marcus wrote the first test in their code base. On his way to the company Christmas party, Marcus realized how time had flown and started thinking back to the changes they'd seen. The team's test coverage had grown quickly and had started stabilizing in the recent weeks and months, peaking at 98% branch coverage.

For a while Marcus had thought that they should push that number all the way to 100%. But in the last couple of weeks, he'd more or less made up his mind—those missing tests wouldn't give them much more value, and putting any more effort into writing tests wouldn't yield additional benefit. A lot of the code that wasn't covered with tests was there only because the APIs they used mandated the implementation of certain interfaces that Marcus's team wasn't using, so why test those empty method stubs?

100% CODE COVERAGE ISN'T THE GOAL 100% sure sounds better than, say, 95%, but the difference in how valuable those tests are to you may be negligible. It really depends on what kind of code isn't covered by tests and whether the tests you have are able to highlight programming mistakes. Having 100% coverage doesn't guarantee the lack of defects—it only guarantees that all of your code was executed, regardless of whether your application behaves like it should. So rather than obsessing about code coverage, focus your attention on making sure that the tests you do write are meaningful.

¹ For some reason, people referred to them as errors, defects, bugs, or issues.

² The term *test-infected* was coined by Erich Gamma and Kent Beck in their 1998 Java Report article, "Test-Infected: Programmers Love Writing Tests."

The team had reached the plateau—the flat part of the curve where additional investments have diminishing returns. In the local Java user group meetings, many teams had reported similar experiences and drawn sketches like figure 1.1.

What changed Marcus's thinking was Sebastian, a senior software architect who'd previously consulted for the investment bank. Sebastian joined the self-service team and quickly became a mentor for the more junior team members, including Marcus. Sebastian was an old fox who seemed to have worked with almost every major programming language suitable for developing web applications. The effect Sebastian had on Marcus was the way he wrote unit tests.

Marcus had formed a habit of back-filling unit tests for any code he'd written before checking it into the team's version control system. But Sebastian's style focused on starting with a test that would fail (obviously), writing enough code to make that test pass, and then writing another failing test. He'd work in this loop until he was done with the task.

Working with Sebastian, Marcus had noticed how his own style of programming was starting to evolve. He'd structure his objects differently and his code looked somehow different, simply because he started looking at the design and behavior of his code from the caller's perspective.

Tests help us shape our design to actual use.

Pointing out what he'd noticed, Marcus felt as if something clicked. As he tried to put into words how his programming style had changed and what kind of effects it had on the outcome, it dawned on Marcus that the tests he'd written weren't merely a quality assurance tool and protection from mistakes and regression. The tests served as a *way of designing code* such that it served a concrete purpose. The code Marcus wrote to pass those failing tests tended to be much simpler than what he used to write, and yet it did everything it needed to.

Tests help us avoid gold-plating by being explicit about what the required behavior is.

What Marcus, our protagonist in this story, experienced is not unlike what many test-infected programmers go through as their awareness and understanding about tests develops. We often start writing automated unit tests because we believe they'll help us avoid embarrassing, time-consuming mistakes. Later on, we may learn that having the tests as a safety net is just part of the equation and another—perhaps bigger—

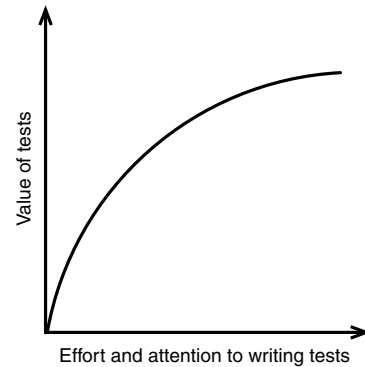


Figure 1.1 The more tests we have, the less value an additional test is likely to yield. The first tests we write most likely target some of the most crucial areas of the code base and as such tend to be high-value and high-risk. When we've written tests for almost everything, most likely the bits that are still lacking test coverage are the least crucial and least likely to break.

part of the benefit is the thinking process we go through as we formulate those tests into code.

Most developers readily understand the benefits of writing automated tests. How many and what kind of tests to write are questions that many still struggle to answer. In my consulting work I've noticed that most programmers agree they should write *some* tests. Similarly, I've noticed that few programmers hold the opinion that they should aim for a 100% code coverage—meaning that their tests exercise every possible execution path through the production code.

So what can we make of this? Most of us agree that writing *some* tests is a no-brainer. However, the closer we get to full code coverage, the fewer of us would agree that aiming for such a volume of tests is a good idea. This is what some folks call a *diminishing return* and it's often visualized with a curve similar to figure 1.1.

In other words, adding a hundred carefully chosen automated tests to a code base is a clear improvement, but when we already have 30,000 tests, those additional hundred tests aren't as likely to make much difference. But as Marcus, our young programmer in an investment bank, learned while working with his mentor, it's not as simple as that. The preceding reasoning fails to acknowledge what I call the *Law of the Two Plateaus*:

*The biggest value of writing a test
lies not in the resulting test
but in what we learn from writing it.*

The Law of the Two Plateaus refers to the two levels of thinking about tests that Marcus's story highlighted. In order to realize the full potential of tests, we need to climb not one but *two* hills, shown in figure 1.2.

As long as we perceive tests merely as a quality assurance tool, our potential is bounded by the lower arc in figure 1.2. Changing our frame of mind toward tests as a programming tool—a design tool—is the catalyst that allows us to move past the *quality plateau* toward the second plateau, the *test-as-design-tool plateau*.

Unfortunately, most code bases seem to stagnate somewhere along the first curve, with the developers not taking that catalyst step in using their tests to drive design. Correspondingly, the developers don't focus enough of their attention on the cost of maintaining the constantly growing test suite.

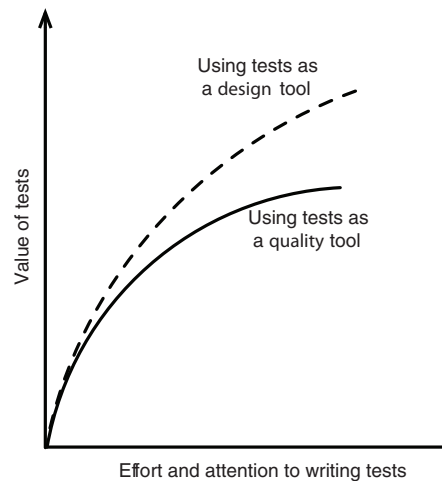


Figure 1.2 The first plateau lies where writing more tests or better tests no longer yields additional value. The second plateau is further up the hill, where we uncover further payoffs from changing our mindset—thinking about tests and testing as a resource richer than a mere tool for verification.

From this position, the big question is what are the factors influencing a programmer's productivity, and what role do unit tests have in that dynamic?

1.2.1 Factors of productivity

The fastest way to write tests is to type them in as quickly as you can without paying attention to the health of an important part of your code base—the test code. But as we'll soon discuss, there are good reasons why you should take the time to nurture your test code, refactor away duplication, and generally pay attention to its structure, clarity, and maintainability.

Test code naturally tends to be simpler than production code.³ Nevertheless, if you cut corners and take on this technical debt of code quality within your tests, it's going to slow you down. In essence, test code ridden with duplication and superfluous complexity lowers your productivity, eating away some of the positive impact of having those tests in place.

It's not just about your tests' readability, but also their reliability and general trustworthiness as well as the time it takes to run those tests. Figure 1.3 illustrates the system of forces around tests that influences our productivity.

Note how the two direct influences for productivity I've identified in this figure are the *length of the feedback cycle* and *debugging*. In my experience these are the two main culprits that consume a programmer's time while on the keyboard.⁴ Time spent debugging is rework that could've largely been avoided if you'd learned about the mistake soon after it happened—and the longer your feedback cycle is, the longer it takes to even begin debugging.

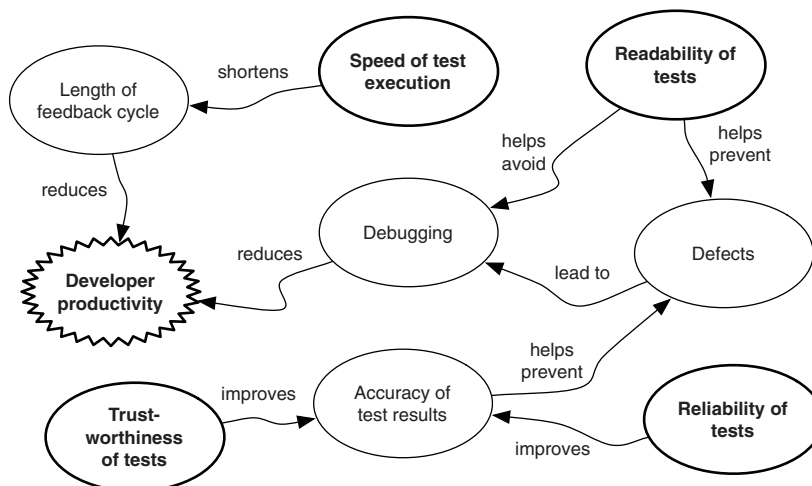


Figure 1.3 Multiple forces of influence related to tests directly or indirectly affect productivity.

³ For instance, few tests make use of conditional statements, loops, and so on.

⁴ There's a lot to be said about meetings as the ultimate killer of productivity, but that's another book.

The rising cost of fixing a bug

What do you think fixing a bug would cost on average? At the XP Day 2009 conference in London, Google's Mark Striebeck reported on Google's estimates around the cost of delay in fixing defects.

Google had estimated that it cost \$5 to fix a bug immediately after a programmer had introduced it. Fixing the same defect would cost \$50 if it escaped the programmer's eyes and was found only after running a full build of the project. The cost surged to \$500 if it was found during an integration test, and to \$5000 if it managed to find its way to a system test.

Considering these numbers, surely it's better to find out about such issues as soon as you possibly can!

Waiting for validation and verification of a change you've made is largely bound to the *speed of test execution*, which is one of the root causes highlighted with bold in the diagram. The other three root causes all contribute to the amount of *debugging* a programmer does.

Lack of *readability* naturally makes you slower in your analysis and encourages you to pick up a debugger because reading the test code doesn't make you any wiser—it's so hard to understand what's going on. Such a lack of readability also induces more *defects*, as it's difficult to *see* a mistake you're making, and defects lead to more debugging sessions.

Another factor that tends to increase the number of defects slipping through is the *accuracy of your test results*. This accuracy is a fundamental requirement for you to be able to rely on your test suite in identifying when you've made a mistake and introduced a defect. The two remaining test-related root causes that influence productivity are direct contributors to the tests' accuracy, namely their *trustworthiness* and *reliability*. In order to rely on your tests being accurate in their reports, the tests need to assert what they promise to and they have to provide that assertion in a reliable, repeatable manner.

It's by paying attention to these influences to your productivity that you can scale the quality plateau. These few root causes we've identified are the key for a programmer to become more productive. The prerequisite to being productive is to know your tools well enough that they are not a constant distraction. Once you know your programming language, you can navigate its core APIs. When you are familiar with the problem domain, focus on the root causes—the readability, reliability, trustworthiness, and speed of your tests.

It is by paying attention to these influences to your productivity that you can scale the quality plateau.

This is also the stuff that most of the rest of this book will revolve around—helping you improve your awareness of and your sensibility toward the test code's readability, trustworthiness, and reliability, and making sure that you can sustain your way of working with the test code by ensuring its maintainability.

Before we go there, let's address that second curve from figure 1.2.

1.2.2 *The curve of design potential*

Let's assume you've written the most crucial tests first—for the most common and essential scenarios and for the most critical hotspots in the software's architecture. Your tests are of high quality, and you've mercilessly refactored away duplication, keeping your tests lean and maintainable. Now imagine you've accumulated such high coverage with your tests that the only ones that you haven't written are those that directly test trivial behaviors such as field accessors. It's fair to say that there's little value to be added by writing those tests. Hence, the earlier gesture toward diminishing returns—you've reached an upper limit for the value you can extract from "just" writing tests.

This is the quality plateau, caused by what we *don't* do. I say that because to go beyond this point and continue your journey toward a higher level of productivity, you need to stop thinking about tests the way you're thinking about them. In order to recover that lost potential I'm talking about, you need to harness a whole different kind of value from the tests you write—creative, design-oriented value rather than the kind of protective, verification-oriented value of tests protecting you from regression and defects.

To summarize, in order to get across both of these two plateaus and reach the full potential of your tests, you need to:

- 1 Treat your test code like you treat production code—refactoring it mercilessly, creating and maintaining high quality tests that you can trust and rely on.
- 2 Start using your tests as a design tool that informs and guides the design of your code toward its actual purpose and use.

As we established earlier, it's the former of these two that most programmers stumble over, struggling to produce the kind of high quality that we're looking for from test code while minimizing the operational cost of writing, maintaining, and running those tests. That's also the focus of this book—having good tests.

That means that we're not going to devote much time and space to discussing the aspect of using tests as a design tool.⁵ With that said, I do want to give you an overall understanding of the kind of dynamics and a way of working that I'm referring to, so let's elaborate on that topic before moving on.

1.3 *Tests as a design tool*

Automated tests written by programmers have traditionally been considered a quality assurance effort targeted at the dual purposes of verifying the correctness of an implementation at the time of its writing and verifying its correctness in the future as the code base evolves. That's using tests as a verification tool—you envision a design, write code to implement the design, and write tests to verify that you've implemented it correctly.

⁵ It so happens that I've written an excellent book on that topic—*Test Driven* (Manning Publications, 2007). Another great pick would be *Growing Object-Oriented Software, Guided by Tests* by Steve Freeman and Nat Pryce (Addison-Wesley, 2009).

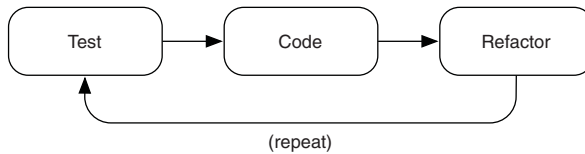


Figure 1.4 The test-driven development process starts by writing a failing test, writing code to pass the test, and refactoring the code base to improve its design.

Using automated tests as a design tool turns everything upside down. When you use tests to design your code, you transform the canonical sequence of “design, code, test” into “test, code, design.” Yes, that’s right. Test comes before code, and you conclude with a retroactive design activity. That concluding design activity is better known as *refactoring*, making the sequence into *test-code-refactor*, visualized as figure 1.4.

If this sounds familiar, you’ve probably heard of a practice called *test-first programming* or *test-driven development (TDD)*.⁶ That’s exactly what we’re talking about here so let’s call it what it is.

1.3.1 Test-driven development

TDD, illustrated in figure 1.5, is a disciplined programming technique built around a simple idea: *no production code is written before there’s a failing test that justifies the existence of that code*. That’s why it’s sometimes referred to as test-first programming.

It’s more than that; by writing a test first, you drive your production code toward a design called for by the test. This has the following desirable consequences:

- *Code becomes usable*—The design and API of your code are suitable for the kind of scenarios you’re going to use it for.
- *Code becomes lean*—The production code only implements what’s required by the scenarios it’s used for.

First, regardless of where in the system’s blueprints you’re currently working and what other components, classes, or interfaces exist or don’t exist, you always have a concrete scenario for which you design a solution. You encode that scenario into an executable example in the shape of an automated unit test. Running that test and seeing it fail gives you a clear goal of making that test pass by writing just enough production code—and no more.

Sketching that scenario into compiling executable test code is a design activity. The test code becomes a client for the production code you’re developing and lets you validate your design before you’ve written a single line. Expressing your needs in the form of concrete examples is a powerful validation tool—value that we can only extract by using tests as a design tool.

Second, by ruthlessly following the rule of only writing enough code to make the tests pass, you can keep your design simple and fit for a purpose. There’s no sign of gold-plating anywhere because no line of code exists without a test—a scenario—that

⁶ Or you actually read the story about Marcus and his mentor, Sebastian, a few pages earlier...

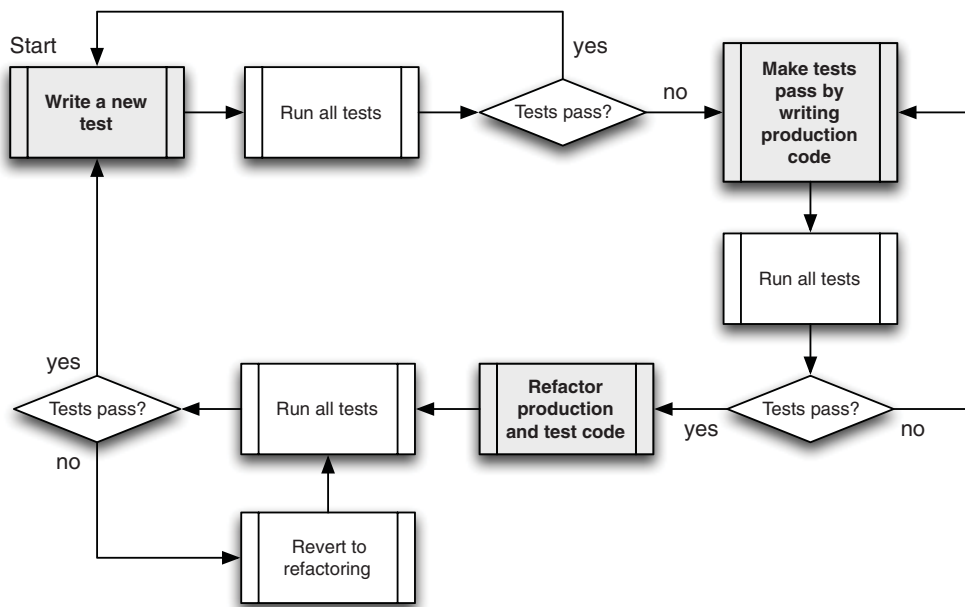


Figure 1.5 Test-driven development is a cyclic process of writing a failing test, making it pass, and refactoring the code to amplify clarity of intent while minimizing duplication. Every step along the way, you're constantly running tests to make sure that you know where you stand.

warrants it. One of the worst offenders of code quality and a major contributor to developers' productivity crawling to a halt is *accidental complexity*.

Accidental complexity is unnecessary complexity. It's complexity that could be avoided by substituting a simpler design that still meets the requirements. Sometimes we programmers like to demonstrate our mental capacity by producing such complex designs that we have trouble understanding them ourselves. I will bet you recognize that primal instinct in yourself, too. The problem with complex designs is that complexity kills productivity and unnecessary complexity is, well, unnecessary and counterproductive.

The combination of a test specifying a flaw or missing functionality in the code, the rule of writing only enough code to make the tests pass, and the ruthless refactoring toward a simple design⁷ are enormously powerful at keeping such accidental complexity at bay. It's not a magic potion, and ultimately what you end up with is influenced by the programmer's design sense and experience.

There's much more to TDD than this—so much more that entire books have been written about the method, including my previous book, *Test Driven* (Manning, 2007), and the more recent *Growing Object-Oriented Software, Guided by Tests* by Steve Freeman and Nat Pryce (Addison-Wesley, 2009). If you'd prefer a more compact introductory

⁷ Simple design isn't the same as *simplistic* design.

title on TDD, I recommend picking up a copy of Kent Beck's *Test Driven Development: By Example* (Addison-Wesley, 2002).

With such widely available and thorough descriptions of TDD around, we're going to keep it short and let you refer to those titles for a more in-depth exploration. But I do want to discuss one aspect of TDD because it has close ties to our main topic of good tests—a style of programming called *behavior-driven development*.

1.3.2 Behavior-driven development

You may have heard of something called BDD, which stands for behavior-driven development. Though people have programmed in some kind of a test-first manner for several decades, the method we know as test-driven development was described and given a name during the 1990s.

Some 10 years later, a London-based consultant named Dan North stepped up to champion the next evolution of test-first programming, after realizing that the vocabulary and mindset of TDD speaking about “tests” was leading people along the wrong path. Dan gave this style of TDD the name behavior-driven development and, in his introductory article published in *Better Software* in 2006, he described BDD like this:

It suddenly occurred to me that people's misunderstandings about TDD almost always came back to the word “test.”

That's not to say that testing isn't intrinsic to TDD—the resulting set of methods is an effective way of ensuring your code works. However, if the methods do not comprehensively describe the behaviour of your system, then they are lulling you into a false sense of security.

I started using the word “behaviour” in place of “test” in my dealings with TDD and found that not only did it seem to fit but also that a whole category of coaching questions magically dissolved. I now had answers to some of those TDD questions. What to call your test is easy—it's a sentence describing the next behaviour in which you are interested. How much to test becomes moot—you can only describe so much behaviour in a single sentence. When a test fails, simply work through the process described above—either you introduced a bug, the behaviour moved, or the test is no longer relevant.

I found the shift from thinking in tests to thinking in behaviour so profound that I started to refer to TDD as BDD, or behaviour-driven development.

After Dan started writing and talking about BDD, others in the global community of software developers stepped forward to further develop the ideas of example-driven, specification-of-behavior oriented thinking that was embodied in Dan North's BDD. As a result, people today refer to BDD in contexts and domains wider than that of code—most notably integrating BDD with business analysis and specification activities at the requirements level.

A particular concept that BDD practitioners vouch for is *outside-in* development with acceptance tests. This is how Matt Wynne and Aslak Hellestøy, authors of *The Cucumber Book: Behaviour-Driven Development for Testers and Developers*, describe it:

Behaviour-Driven Development builds upon Test-Driven Development by formalizing the good habits of the best TDD practitioners. The best TDD practi-

tioners work from the outside-in, starting with a failing customer acceptance test that describes the behavior of the system from the customer's point of view. As BDD practitioners, we take care to write the acceptance tests as examples that anyone on the team can read. We make use of the process of writing those examples to get feedback from the business stakeholders about whether we're setting out to build the right thing before we get started.

As testament to this development, a crop of BDD tools and frameworks has popped up seeking to embed the underlying ideas, practices, and conventions into the software developer's tool chain. We'll see some of those tools in chapter 8.

When we speak of "good tests" in the remainder of this book, keep in mind the realization Dan made and pay attention to words. Vocabulary matters.

1.4 Summary

We've come a long way from the days of "cowboy coding" with programmers hacking away at code without any tests to catch them when they fall from the saddle. Developer testing and automated tests are commonplace or, if not exactly standard practice, are definitely a topic of active discussion. The value of having a thorough suite of automated tests for your code is undeniable.

In this chapter you familiarized yourself with the Law of the Two Plateaus. The first plateau is where the value programmers are getting out of their tests is limited because they already have full test coverage. But you can get further. Programmers who pay attention to the quality of their tests zoom right past the first plateau, aiming for the top. It's one thing to have tests, and it's a whole nother thing to have good tests. That's why you're reading this book and that's the challenge we're going to take up in the remaining chapters.

After all, you get much more out of the climb if you go all the way to the top rather than stopping halfway.

In search of good

In this chapter

- What makes a test “good”?
- Testing relevant behavior
- The importance of reliable tests

We’re on a journey of learning about good tests. We want to learn to identify good tests, write good tests, and improve not-so-good tests so they become good tests—or at least closer to being good tests. The question is, What makes a test “good”? What are the magic ingredients? There are several aspects to consider, including:

- The test code’s readability and maintainability
- How the code is organized both within the project and within a given source file
- What kinds of things a test is checking for
- How reliable and repeatable a test is
- How a test makes use of test doubles

We’ll be taking a closer look at all of these aspects in this chapter.

The preceding list is far from being comprehensive. The range of factors that may tip your test-quality scale either way is endless. Similarly, some of the factors

don't matter that much in all contexts. For some tests, their execution speed may be crucial, whereas for other tests, being extremely focused is key.

Furthermore, some of the quality of test code is in the eye of the beholder. As is the case with code in general, personal preference has a role in defining what “good” is—I'm not going to pretend that it wouldn't. I'm also not going to pretend that I can avoid my bias and preference from coming through in this book. Though I've tried to steer away from pure matter-of-taste questions, you'll find numerous sections where my opinions clearly show through. I think that's fine. After all, the best I can offer is my honest (and opinionated) view of things based on my experience, shaped by the wonderful individuals and software professionals from whom I've learned about code and, specifically, about test code.

With that disclaimer out of the way, let's discuss some of those aspects of test quality and establish what about them makes them relevant to our interests.

2.1 *Readable code is maintainable code*

Yesterday I popped into our office on my way back from a consulting gig and struck up a conversation with a colleague about an upcoming 1K competition that my colleague was going to attend. Such competitions are an age-old tradition at demo parties—a type of geek gathering where hackers gather at a massive arena for a long weekend with their computers, sleeping bags, and energy drinks. Starting from the first gatherings, these hackers have faced off, wielding their mad skills at producing 3D animations with what most people would today consider antiquated hardware.

A typical constraint for such animations has been size. In the case of the competition my colleague was preparing for, the name *1K* refers to the maximum size of the code compiled into a binary executable, which must be less than 1,024 bytes. Yes, that's right—1,024 bytes. In order to squeeze a meaningful program into such a tiny space, the competitors need to resort to all kinds of tricks. For example, one common trick to pack your code more tightly is to use the same name for many variables—because the resulting code compresses a bit better like that. It's crazy.

What's also crazy is the resulting code. When they're done squeezing the code down to 1,024 bytes, the source code is undecipherable. You can barely recognize which programming language they've used! It's essentially a write-only code base—once you start squeezing and compressing, you can't make functional changes because you wouldn't be able to tell what to edit where and how.

To give you a vivid taste of what such code might look like, here's an actual submission from a recent JS1k competition where the language of choice is JavaScript and it needs to fit into 1,024 bytes:

```
<script>with(document.body.style){margin="0px";overflow="hidden";}
var w=window.innerWidth;var h=window.innerHeight;var ca=document.
getElementById("c");ca.width=w;ca.height=h;var c=ca.getContext("2d");
m=Math;fs=m.sin;fc=m.cos;fm=m.max;setInterval(d,30);function p(x,y,z){
return{x:x,y:y,z:z};}function s(a,z){r=w/10;R=w/3;b=-20*fc(a*5+t);
return p(w/2+(R*fc(a)+r*fs(z+2*t))/z+fc(a)*b,h/2+(R*fs(a))/z+fs(a)*b);
}function q(a,da,z,dz){var v=[s(a,z),s(a+da,z),s(a+da,z+dz),s(a,z+dz)]
```

```
;c.beginPath();c.moveTo(v[0].x,v[0].y);for(i in v)c.lineTo(v[i].x,v[i].y);c.fill();}var Z=-0.20;var t=0;function d(){t+=1/30.0;c.fillStyle="#000";c.fillRect(0,0,w,h);c.fillStyle="#f00";var n=30;var a=0;var da=2*Math.PI/n;var dz=0.25;for(var z=Z+8;z>Z;z-=dz){for(var i=0;i<n;i++){fog=1/(fm((z+0.7)-3,1));if(z<=2){fog=fm(0,z/2*z/2);}var k=(205*(fog*Math.abs(fs(i/n*2*3.14+t))))>>0;k*=(0.55+0.45*fc((i/n+0.25)*Math.PI*5));k=k>>0;c.fillStyle="rgb("+k+", "+k+", "+k+)" ";q(a,da,z,dz);if(i%3==0){c.fillStyle="#000";q(a,da/10,z,dz);}a+=da;}}Z-=0.05;if(Z<=dz)Z+=dz;}</script>
```

Granted, that is a couple of magnitudes more extreme a situation than what you'd find at a typical software company. But we've all seen code at work that makes our brains hurt. Sometimes we call that kind of code *legacy* because we've inherited it from someone else and now we're supposed to maintain it—except that it's so difficult that our brains hurt every time we try to make sense of it. Maintaining such unreadable code is hard work because we expend so much energy understanding what we're looking at. It's not just that. Studies have shown that poor readability correlates strongly with defect density.¹

Automated tests are a useful protection against defects. Unfortunately, automated tests are also code and are also vulnerable to bad readability. Code that's difficult to read tends to be difficult to test, too, which leads to fewer tests being written. Furthermore, the tests we do write often turn out to be far from what we consider good tests because we need to kludge our way around the awkwardly structured, difficult-to-understand code with APIs and structures that aren't exactly test-friendly.

We've established (almost to the point of a rant) that code readability has a dire impact on the code's maintainability. Now what about the readability of test code? How is that different, or is it any different? Let's take a look at a not-so-far-fetched example of unreadable test code, shown in this listing:

Listing 2.1 Code doesn't have to be complex to lack readability

```
@Test
public void flatten() throws Exception {
    Env e = Env.getInstance();
    Structure k = e.newStructure();
    Structure v = e.newStructure();
    //int n = 10;
    int n = 10000;
    for (int i = 0; i < n; ++i) {
        k.append(e.newFixnum(i));
        v.append(e.newFixnum(i));
    }
    Structure t = (Structure) k.zip(e.getCurrentContext(),
        new IOObject[] {v}, Block.NULL_BLOCK);
    v = (Structure) t.flatten(e.getCurrentContext());
    assertNotNull(v);
}
```

¹ Raymond P.L. Buse, Westley R. Weimer. "Learning a Metric for Code Readability." *IEEE Transactions on Software Engineering*, 09 Nov. 2009. IEEE computer Society Digital Library. IEEE Computer Society, <http://doi.ieeecomputersociety.org/10.1109/TSE.2009.70>

What is this test checking? Would you say that it's easy to decipher what's going on here? Imagine yourself being the new guy on this team—how long would it take you to figure out the test's intent? What kind of code forensics would you have to go through to grok the situation if this test suddenly starts failing? Based on how I feel about that hideous snippet of code, I'll wager that you immediately identified a handful of things that could be improved about the poor little test—and that readability is a common theme with those improvements.

2.2 *Structure helps make sense of things*

I've had the pleasure and horror of seeing numerous code bases that weren't full of beautiful strides of genius flowing from one source file to the next. Some of them never jumped to another source file because it was all there—all of the code and logic triggered by, say, the submission of a web form would reside in a single source file. I've had a text editor crash due to the size of a source file I foolishly tried to open. I've seen a web application vomit an error because a JavaServer Pages file had grown so big that the resulting byte code violated the Java class file specification. It's not just that structure would be useful—the lack of structure can be damaging.

What's common among most of these instances of never-ending source listings is that nobody wanted to touch them. Even the simplest conceptual changes would be too difficult to map onto the source code in front of you. There was no structure your brain could rely on. Divide and conquer wasn't an option—you had to juggle the whole thing in your head or be prepared for a lot of contact between your forehead and the proverbial brick wall.

As illustrated by figure 2.1 you don't want just *any* structure to help make sense of things. You need structure that makes sense as such—one that's aligned with the way your brain and your mental models are prepared to slice and dice the world. Blindly externalizing snippets of code into separate source files, classes, or methods does reduce the amount of code you're looking at a given point in time, thereby alleviating the problem of overloading your brain. But it doesn't get you much closer to isolating and understanding the one aspect of the program's logic that we're interested in right now. For that you need a structure that makes sense.

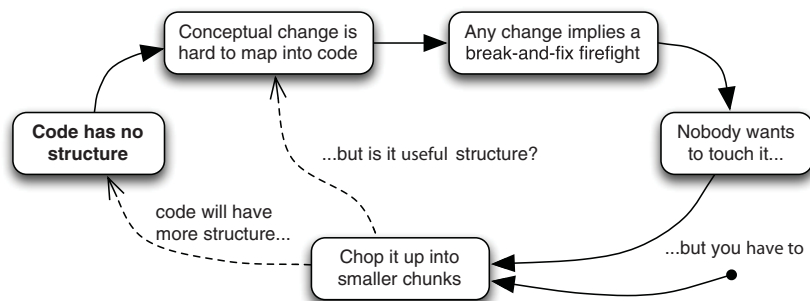


Figure 2.1 It's not just about having structure—it needs to be a useful structure.

When faced with monolithic, never-ending source listings, the obvious solution is to chop them up into smaller pieces, extracting blocks of code into methods. You might go from a huge 500-line method in one class to dozens of methods on 10 classes with the average length dropping below 10 lines per method. That would introduce more structure into the code—at least if you ask a compiler. You’d also be able to see whole methods on your screen at once instead of scrolling back and forth.

But if the boundaries for splitting up that monolith don’t make sense—if they don’t map to the domain and your abstractions—we might be doing more harm than good because the concepts might now be physically scattered farther from each other than before, increasing the time you spend going back and forth between source files. It’s simple. What matters is whether the structure of your code helps you locate the implementation of higher-level concepts quickly and reliably.

Test code is an excellent example of this phenomenon. Let’s say you have an application that’s fairly well covered by an automated test—one automated test. Imagine that this test exercises all of the application’s business logic and behavior through a single monolithic test method that takes half an hour to execute. Now say that the test eventually fails, as illustrated in figure 2.2, because you’re making a change in how mailing addresses are represented internally in the application and you mess up something while you’re at it. There’s a bug. What happens next?

I’d imagine it’s going to take a while to pinpoint the exact place in the test code where your programming error manifests itself. There’s no structure in the test code to help you see what affects what, where a certain object is instantiated, what the value of a given variable is at the point where things fall apart, and so forth. Eventually, when you’ve managed to identify and correct your mistake, you have no choice but to run the whole test—all 30 minutes of it—to make sure that you really did fix the problem and that you didn’t break something else in the process.

Continuing this thought experiment, fast-forwarding an hour or so, you’re about to make another change. This time, having learned from your previous mistake, you’re careful to make sure you’ve understood the current implementation in order to ensure that you’ll make the right kind of change. How would you do that? By reading the code, and especially perusing test code that shows you in concrete terms how the production code is expected to behave. Except that you can’t find the relevant parts of the test code because there’s no structure in place.

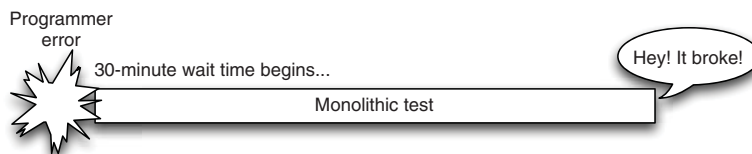


Figure 2.2 Long delay in feedback is a real productivity-killer

What you need are focused tests that are readable, accessible, and comprehensible so that you can:

- Find test classes that are relevant for the task at hand
- Identify the appropriate test methods from those classes
- Understand the lifecycle of objects in those test methods

These are all things that you can get by paying attention to your tests' structure and making sure that it's useful. Having a useful structure is hardly enough, of course.

2.3 *It's not good if it's testing the wrong things*

More than once I've concluded a journey of reading and debugging code to find the cause for an undesirable system behavior at almost the same place I started looking. A particularly annoying detail overlooked in such a bug-hunt is the contents of a test. The first thing I tend to do when digging into code is to run all the tests to tell me what's working and what's not. Sometimes I make the mistake of trusting what the tests' names tell me they're testing. Sometimes it turns out that those tests are testing something completely different.

This is related to having a good structure—if a test's name misrepresents what it tests, it's akin to driving with all the road signs turned the wrong way. You should be able to trust your tests.

A couple of years ago I was carrying out a code audit for a product that had been under development for more than a decade. It was a big codebase and I could tell from its structure that some parts were clearly newer than others. One of the things that distinguished more recent code from the older was the presence of automated tests. But I quickly found out that I couldn't tell from the tests' names what they were supposed to verify and, looking closer, it turned out that the tests weren't actually testing what they promised to test. It wasn't a Java codebase but I've taken the freedom to translate a representative example to Java:

```
public class TestBmap {
    @Test
    public void mask() {
        Bmap bmap = new Bmap();
        bmap.addParameter(IPSEC_CERT_NAME);
        bmap.addParameter(IPSEC_ACTION_START_DAYS, 0);
        bmap.addParameter(IPSEC_ACTION_START_HOURS, 23);
        assertTrue(bmap.validate());
    }
}
```

Looking at this code you'll immediately notice that the test's name is less than perfect. But on a closer look it turns out that whatever "mask" means for a "Bmap," the test is only checking that certain parameters are a valid combination. Whether the validation works is somewhat irrelevant if the actual behavior isn't correct even when the input is valid.

There's a lot to be said about testing the right things, but it's also crucial to test those right things the right way. Of particular importance from a maintainability point of view is that your tests are checking for the intended behavior and not a specific implementation. That's a topic we'll touch on in the next chapter, so let's leave it at that for now.

2.4 Independent tests run easily in solitude

There's a lot to be said about tests, what they should or shouldn't contain, what they should or shouldn't specify, and how they should be structured in the name of readability. What goes on *around* tests sometimes plays an equally vital role.

Human beings—our brains to be more exact—are enormously powerful information processors. We can make seemingly instant evaluations of what's going on in our physical surroundings and react in a blink. We dodge that incoming snowball before we even realize what we're reacting to. These reactions are in our DNA. They're behavioral recipes that instruct our body to move when our senses observe a familiar pattern. Over time our cookbook of these recipes grows in sophistication, and we're soon lugging around a complex network of interconnected patterns and behaviors.

This happens at work, too. Exploring a foreign code base for the first time, we'll have formed a clear picture of the most prevalent conventions, patterns, *code smells*, and pitfalls within 15 minutes. What makes this possible is our ability to recognize a familiar pattern and be able to tell what else we're likely to see nearby.

WHAT'S A CODE SMELL? A *smell* in code is a hint that something *might* be wrong with the code. To quote the Portland Pattern Repository's Wiki, "If something smells, it definitely needs to be checked out, but it may not actually need fixing or might have to just be tolerated."

For example, one of the first things I pay attention to when introducing myself to a new codebase is the size of methods. If methods are long I *know* that there are a bunch of other issues to deal with in those particular modules, components, or source files. Another signal I'm tuning to is how descriptive the names of the variables, methods and classes are.

Specifically in terms of test code, I pay attention to the tests' *level of independence*, especially near architectural boundaries. I do this because I've found so many code smells by taking a closer look at what's going on in those boundaries, and I've learned to be extra careful when I see dependencies to:

- Time
- Randomness
- Concurrency
- Infrastructure
- Pre-existing data
- Persistence
- Networking

What these things have in common is that they tend to complicate what I consider the most basic litmus test for a project's test infrastructure: can I check out a fresh copy from version control to a brand new computer I just unboxed, run a single command, lean back, and watch a full suite of automated tests run and pass?

Isolation and independence are important because without them it's much harder to run and maintain tests. Everything a developer needs to do to their system in order to run unit tests makes it that much more burdensome.

Whether you need to create an empty directory in a specific location in your filesystem, make sure that you have a specific version of MySQL running at a specific port number, add a database record for the user that the tests use for login, or set a bunch of environment variables—these are all things that a developer shouldn't need to do. All of these small things add up to increased effort and weird test failures.²

A characteristic of this type of dependency is that things like the system clock at the time of test execution or the next value to come out from a random number generator are *not in your control*. As a rule of thumb, you want to avoid erratic test failures caused by such dependencies. You want to put your code into a bench vise and control everything by passing it test doubles and otherwise isolating the code to an environment that behaves exactly like you need it to.

Don't rely on test order within a test class

The general context for the advice of not letting tests depend on each other is that you should not let tests in one class depend on the execution or outcome of tests in *another* class. But it really applies to dependencies within a single test class, too.

The canonical example of this mistake is when a programmer sets up the system in a starting state in a `@BeforeClass` method and writes, say, three consecutive `@Test` methods, each modifying the system's state, trusting that the previous test has done its part. Now, when the first test fails, all of the subsequent tests fail, but that's not the biggest issue here—at least you're alerted to something being wrong, right?

The real issue is when some of those tests fail for the *wrong* reason. For instance, say that the test framework decides to invoke the test methods in a different order. False alarm. The JVM vendor decides to change the order in which methods are returned through the Reflection API. False alarm. The test framework authors decide to run tests in alphabetical order. False alarm again.³

You don't want false alarms. You don't want your tests failing when the behavior they're checking isn't broken. That's why you shouldn't intentionally make your tests brittle by having them depend on each other's execution.

² If you can't find a way to avoid such manual configuration, at least make sure developers need to do it only once.

³ If this sounds far-fetched, you should know that JUnit doesn't promise to run test methods in any particular order. In fact, several tests in the NetBeans project started breaking when Java 7 changed the order in which declared methods are returned through `Class.getDeclaredMethods()`. Oops. I guess they didn't have independent tests...

One of the most unusual examples of a surprising test failure is a test that passes as part of the whole test suite but fails miserably when it's run alone (or vice versa). Those symptoms reek of interdependent tests. They assume that another test is run before they are, and that the other test leaves the system in a particular state. When that assumption kicks you in the ankle, you have one hellish debugging session ahead.

To summarize, you should be extra careful when writing tests for code that deals with time, randomness, concurrency, infrastructure, persistence, or networking. As a rule of thumb, you should avoid these dependencies as much as you can and localize them into small, isolated units so that most of your test code doesn't need to suffer from the complications and you don't have to be on your toes all the time—just in those few places where you tackle the tricky stuff.

So how would that look in practice? What exactly should you do? For example, you could see if you can find a way to:

- Substitute test doubles for third-party library dependencies, wrapping them with your own adapters where necessary. The tricky stuff is then encapsulated inside those adapters that you can test separately from the rest of application logic.
- Keep test code and the resources they use together, perhaps in the same package.
- Let test code produce the resources it needs rather than keeping them separate from the source code.
- Have your tests set up the context they need. Don't rely on any other tests being run before the one you're writing.
- Use an in-memory database for integration tests that require persistence, as it greatly simplifies the problem of starting tests with a clean data set. Plus, they're generally superfast to boot up.
- Split threaded code into asynchronous and synchronous halves, with all application logic in a regular, synchronous unit of code that you can easily test without complications, leaving the tricky concurrency stuff to a small, dedicated group of tests.

Achieving test isolation can be difficult when working with legacy code that wasn't designed for testability and therefore doesn't have the kind of modularity you'd like to have. But even then, the gain is worth the effort of breaking those nasty dependencies and making your tests independent from their environment and from each other. After all, you need to be able to rely on your tests.

2.5 *Reliable tests are reliable*

In the previous section I said that sometimes a test is testing something completely different than what you thought it tests. What's even more distracting is that sometimes they don't test a damn thing.

A colleague used to call such tests *happy tests*, referring to a test happily executing a piece of production code—possibly all of its execution paths—without a single assertion being made. Yes, your test coverage reports look awesome as the tests are thoroughly executing every bit of code you’ve written. The problem is that such tests can only fail if the production code being invoked throws an exception. You can hardly rely on such tests to watch your back, can you? Especially if the programmers have had a convention to encapsulate all of the test methods’ bodies into a try-catch.⁴ This listing shows an example of one such bugger.

Listing 2.2 Can you spot the flaw in this test?

```
@Test
public void shouldRefuseNegativeEntries() {
    int total = record.total();
    try {
        record.add(-1);
    } catch (IllegalArgumentException expected) {
        assertEquals(total, record.total());
    }
}
```

Some tests are less likely to fail than others, and the previous listing is a prime example of the other extreme, where the test is likely to never fail (and likely never has in the past either). If you look carefully, you’ll notice that the test won’t fail even if `add(-1)` doesn’t throw an exception as it’s supposed to.

Tests that can hardly ever fail are next to useless. With that said, a test that passes or fails intermittently is an equally blatant violation toward fellow programmers; see figure 2.3.

Some years ago I was consulting on a project and spent most of my days pair programming with the client’s technical staff and other consultants. One morning I took on a new task with my pair and ran the related test set as the first thing to do as usual. My pair was intimately familiar with the codebase, having written a significant part of it, and was familiar with its quirks, too. I noticed this when a handful of the tests failed on the first run before we’d touched anything. What tipped me off was how my pair responded to the test failure—he routinely started rerunning the tests again and again until after four or five times all of the tests had passed at least once. I’m not 100% sure, but I don’t think those particular tests all passed at the same time even once.

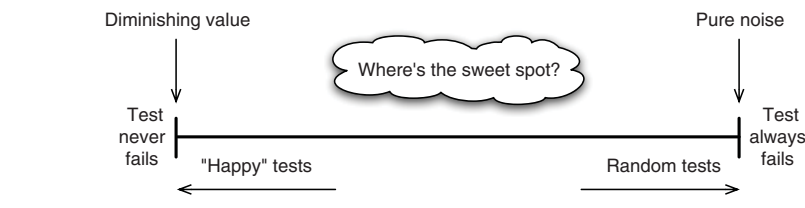


Figure 2.3 Tests have little value if they’ve never failed or if they’re failing all the time.

⁴ True story. I spent an hour removing them and the rest of the day fixing or deleting the failed tests that were uncovered by my excavation.

As flabbergasted as I was, I realized that I was looking at a bunch of tests that represented a whole different cast of unreliable tests. These particular tests turned out to be failing randomly because the code under test incorporated nondeterministic logic that screwed up the tests some 50% of the time. In addition to the use of pseudo-random generators in the code being tested, a common cause for such intermittently failing behavior is the use of time-related APIs. My favorite is a call to `System.currentTimeMillis()`, but a close second is a ubiquitous `Thread.sleep(1000)` sprinkled throughout in an attempt to test asynchronous logic.

In order to rely on your tests, they need to be repeatable. If I run a test twice, it must give me the same result. Otherwise, I'll have to resort to manual arbitration after every build I make because there's no way to tell whether 1250/2492 tests means that everything's all right or that all hell's broken loose with that last edit. There's no way to tell.

If your logic incorporates bits that are asynchronous or dependent on current time, be sure to isolate those bits behind an interface you can use for substituting a "test double" and make the test repeatable—a key ingredient of a test being reliable.

2.6 Every trade has its tools and tests are no exception

What's this test double I speak of? If you don't have test doubles in your programmer's toolkit, you're missing out on a lot of testing power. *Test double* is an umbrella term for what some programmers have come to know as *stubs*, *fakes*, or *mocks* (which is short for mock object). Essentially they're objects that you substitute for the real implementation for testing purposes. See figure 2.4.

You could say that test doubles are a test-infected programmer's best friend. That's because they facilitate many improvements and provide many new tools for our disposal, such as:

- Speeding up test execution by simplifying the code that would otherwise be executed
- Simulating exceptional conditions that would otherwise be difficult to create
- Observing state and interaction that would otherwise be invisible to your test code

There's a lot more to test doubles than this and we'll come back to this topic in more detail in the next chapter. But test doubles aren't the only tool of the trade for writing good automated tests.

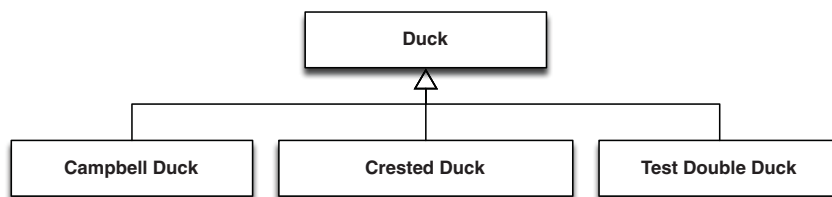


Figure 2.4 A test double for a duck looks just like a duck and quacks *almost* like a duck—but certainly isn't a real duck.

Perhaps the most essential tool of the trade is a test framework such as JUnit. I still remember some of my first attempts at getting a piece of code to work like I wanted it to. When I messed up and got stuck, I'd uncomment half a dozen statements that would print to the console and relaunch the program so I could analyze the console output and figure out what I'd broken and how.

It wasn't more than a couple of months into my professional career when I first bumped into this same practice being upheld by commercial software developers. I trust that I don't need to point out how incredibly unprofessional and wasteful such a practice is compared to writing automated, repeatable tests with tools like JUnit.

In addition to a proper test framework and test doubles, my top three tools of the trade for software developers writing automated tests include one more tool in the chain—the build tool. Whatever your build process looks like, whichever tool or technology your build scripts use under the hood, there's no good excuse for not integrating your automated tests as part of that build.

2.7 **Summary**

In this chapter we've established several coarse properties for what a good test is. We noted that these things are dependent on context and that there are few absolute truths when it comes to what makes a test “good.” We did identify a number of issues that generally have a major impact on how good or appropriate—how fit for its purpose—an automated test is.

We began by noting that one of the essential virtues for a test is its readability, because lacking the ability to be read and understood, test code becomes a maintenance problem that solves itself very soon—by getting deleted because it's too costly to maintain.

We then pointed out how test code's structure makes it usable, allowing the programmer to quickly find their way to the right pieces and helping the programmer understand what's going on—a direct continuation on readability.

Next we shed light on how tests sometimes test the wrong things and how that can create problems by leading you down the wrong path or by muddying the waters, hiding the test's actual logic and making the test itself unreadable.

To conclude the underlying theme of tests sometimes being unreliable, we identified some of the common reasons for such unreliability and how important it is for tests to be repeatable.

Lastly, we identified three essential tools of the trade for writing automated tests—a test framework, an automated build that runs tests written with that framework, and test doubles for improving your tests and ability to test. This third topic is important enough that we've dedicated the next chapter to discussing the use of test doubles for writing good tests.

Test doubles

In this chapter

- What can we do with test doubles?
- What kind of test doubles do we have at our disposal?
- Guidelines for using test doubles

The concept of *stubs* or *dummies* has been around about as long as we've had the ability to structure software code into classes and methods. Once the primary reason for creating such utilities was to serve as a placeholder until the real thing became available—to allow you to compile and execute one piece of code before its surrounding pieces were in place.

These objects have much more diverse purposes in the context of modern developer testing. Instead of merely allowing the compilation and execution of code without certain dependencies present, the test-infected programmer creates a variety of such “for testing only” facilities in order to isolate the code under test, speed up test execution, make random behavior deterministic, simulate special conditions, and to give tests access to otherwise hidden information.

These purposes are served by similar and yet different types of objects that we collectively refer to as *test doubles*.¹

¹ Though the term *test double* was first introduced to me by fellow Manning author J. B. Rainsberger. I credit Gerard Meszaros and his book, *xUnit Test Patterns: Refactoring Test Code* (Addison Wesley, 2007), for popularizing the term and the related taxonomy within the software development community.

We'll begin exploring test doubles by first establishing why a developer might turn to test doubles. Understanding the potential benefits of using test doubles, we'll look at the different types at our disposal. We'll conclude this chapter with a handful of simple guidelines for using test doubles.

But now, let's ask ourselves, *What's in it for me?*

3.1 The power of a test double

Mahatma Gandhi said, "Be the change you want to see in the world." Test doubles fulfill Gandhi's call to action by becoming the change you want to see in your code. Far-fetched? Let me elaborate.

Code is a mass. It's a network of code that refers to other code. Each of these chunks has a defined behavior—behavior you, the programmer, have defined for them. Some of that behavior is atomic and contained within a single class or method. Some of that behavior implies interaction between different chunks of code.

Every now and then, in order to verify that the behavior of one chunk of code is what you want it to be, your best option is to replace the code around it so that you gain full control of the environment in which you're testing your code. You're effectively *isolating the code under test from its collaborators* to test it, as illustrated in figure 3.1.

This is the most fundamental of the reasons for employing a test double—to isolate the code you want to test from its surroundings. There are many other reasons, too, as implied in the opening of this chapter, where we suggested the need for "for testing only" facilities in order to:

- Isolate the code under test
- Speed up test execution
- Make execution deterministic
- Simulate special conditions
- Gain access to hidden information

There are multiple types of test doubles that you could use for achieving these effects. Most of them can be gained with any one type of a test double, whereas some are a better fit for a particular type of test double. We'll come back to these questions in section 3.2. Right now, I'd like to create a shared understanding of the reasons listed here—the reasons we have test doubles in the first place and the purposes we use them for.

3.1.1 Isolating the code under test

When we talk about isolating code under test in the context of an object-oriented programming language such as Java, our world consists of two kinds of things:

- The code under test
- Code that the code under test interacts with

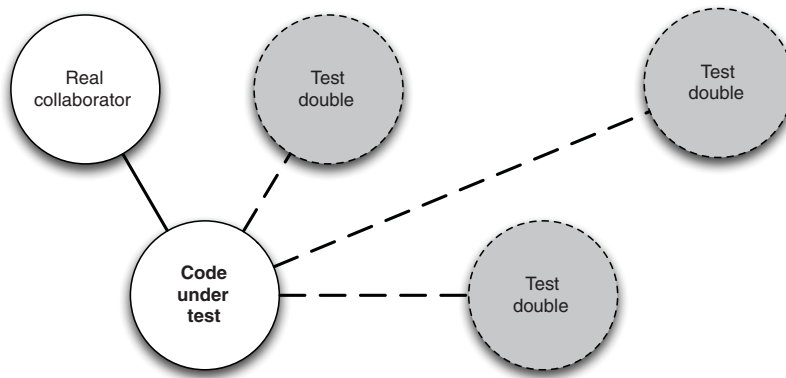


Figure 3.1 Test doubles help you isolate the code under test so that you can test all facets of its behavior. In this example we’ve replaced three of the four collaborators with test doubles, deciding to scope the test to the behavior and collaboration of the code under test and one particular collaborator.

When we say we want to “isolate the code under test” we mean we’re isolating the code we want to test from all other code. By doing this we not only make our tests more focused and approachable, but also easier to set up for. In practice, “all other code” includes code that’s invoked from the code we want to test. The following listing illustrates this through a simple example.

Listing 3.1 Code under test (Car) and its collaborators (Engine and Route)

```

public class Car {
    private Engine engine;

    public Car(Engine engine) {
        this.engine = engine;
    }

    public void start() {
        engine.start();
    }

    public void drive(Route route) {
        for (Directions directions : route.directions()) {
            directions.follow();
        }
    }

    public void stop() {
        engine.stop();
    }
}

```

As you can see, this example consists of a Car, the car’s Engine, and Route, a sequence of Directions to follow. Let’s say we want to test the Car. We have a total of four classes, one of which is the code under test (Car) and two of which are collaborators

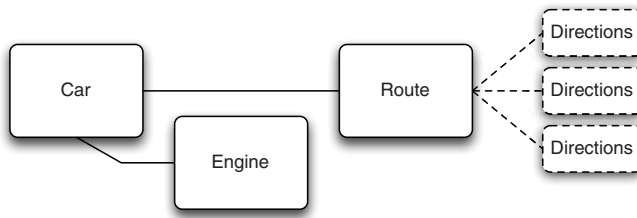


Figure 3.2 A Car uses its Engine and its Routes directly and the Route's Directions only indirectly.

(Engine and Route). Why isn't Directions also a collaborator? Well, it is in the sense that Car references and invokes methods on Directions. But there's another way to look at this scenario. Let's see if figure 3.2 helps clarify this notion.

If we raise our attention up one level of abstraction from the classes that are referenced from Car's methods and place ourselves in the shoes of the Car, what we see—as illustrated in figure 3.2—is that Directions are accessed *through* Route. Because Directions are acquired through Route, isolating the Car from all of its collaborators can be done by replacing Engine and Route with test doubles. As we substitute a faux implementation of Route, we're in total control of the kind of Directions it provides to the Car.

Now that you understand the underlying principle of how you can gain control by substituting a couple of test doubles, let's see what kinds of nice things you can accomplish with them.

3.1.2 Speeding up test execution

One of the pleasant side effects of replacing real collaborators with test doubles is that a test double's implementation is often faster to execute than that of the real thing. Sometimes, a test double's speed isn't just a side effect but rather the primary reason for using a test double.

Consider the driving example from figure 3.2. Let's say that initializing a Route involves a weighted-graph search algorithm for finding the shortest path between the Car's current location and its desired destination. With the complexity of today's street and highway networks, this calculation takes some time. Though the algorithm might churn through once relatively fast, even small delays in test execution add up. If every one of the tests initializes a Route, you might be wasting seconds or even minutes worth of CPU cycles in this algorithm—and minutes is *forever* when talking about a developer getting immediate feedback by running automated tests.

With a test double in place that always returns a canned example, a precalculated route to the destination, the unnecessary wait is avoided and tests run that much faster. Which is nice. You'll want to test those slow Route algorithms somewhere—with separate focused tests—but you don't want to be running those slow algorithms *everywhere*.

Though speed is almost always a good thing, it's not always the most important thing. After all, a faster car isn't that useful if you're heading in the wrong direction.

3.1.3 Making execution deterministic

I once listened to Tony Robbins, the well-known motivational speaker, talk about surprises and how, even though we all say we like surprises, we only like the surprises that we wanted. That's so true and it applies to software as well, especially when talking about testing code.

Testing is about specifying behavior and verifying conformance to that specification. This is simple and straightforward as long as our code is totally deterministic and its logic doesn't contain even a hint of randomness. In essence, for code (and tests) to be deterministic, you need to be able to run your tests repeatedly against the same code and always get the same result.

It's not uncommon to find a situation where your production code needs to incorporate an element of randomness or otherwise compromise your demand for identical results for repeated executions. For example, if you were developing a craps game, you'd better make it so that the way the dice will fall can't be predicted—so it's random.²

Perhaps the most typical case where an element of nondeterministic behavior is called for is time-dependent behavior. Coming back to the example of a Car asking a Route for Directions, imagine that the algorithm used to calculate the route considers the time of day among other things, like traffic, speed limits, and so on, as shown next.

Listing 3.2 Sometimes code's behavior is inherently nondeterministic

```
public class Route {
    private Clock clock = new Clock();
    private ShortestPath algorithm = new ShortestPath();

    public Collection<Directions> directions() {
        if (clock.isRushHour()) {
            return algorithm.avoidBusyIntersections();
        }
        return algorithm.calculateRouteBetween(...);
    }
}
```

Calculated route
different during
rush hour!

If this is the case, how could you check that the algorithm calculates the correct route regardless of when the test is executed? After all, the algorithm surely acquires the time from some kind of a clock and, though the algorithm might suggest taking the highway at 3:40 p.m. and 3:45 p.m., the best result might suddenly be to take the interstate when it's now 3:50 p.m. and the afternoon traffic starts choking up the highway.

Again, test doubles can lend a hand with these kinds of sources for nondeterministic behavior. For example, a particular instance of a game of craps is suddenly much easier to simulate when your dice are test doubles rigged to yield a known sequence of faces. Similarly, it's much easier to describe the expected output for a log file if

² Craps is a dice game in which players bet on the outcome of one or more rolls of a pair of dice. That's what the gangsters are playing in the movies as they crouch on a street corner throwing dice over stacks of paper money.

you’ve just replaced the system clock with a test double that always yields the same moment in time.

Making execution deterministic is at the core of being in control of your collaborators and being able to eliminate all variables while setting up the exact scenario and setup you want to test. Speaking of scenarios, test doubles also enable you to simulate conditions that normally shouldn’t happen.

3.1.4 Simulating special conditions

Most of the software we write tends to be simple and straightforward—at least in the sense that most of the code is deterministic. As a result of this we can reconstruct almost any situation we want by instantiating a suitable object graph to pass as arguments to the code we’re testing. We *could* test that the `Car` from listing 3.1 ends up in the right place when we start from “1 Infinite Loop, Cupertino, CA,” set the destination to “1600 Amphitheatre Parkway, Mountain View, CA,” and say `drive()`.

Some conditions we can’t create using just the APIs and features of our production code. Say that our `Route` gets its directions over the internet from Google Maps. How would we test that `Route` behaves well even in the unfortunate scenario that our internet connection dies while requesting directions to the destination?

Short of having your test disable the computer’s network interfaces for the duration of the test, there’s not much you can do to fabricate such a network-connection error but substitute a test double somewhere that throws an exception when asked to connect.³

3.1.5 Exposing hidden information

Our last (but not least important) *raison d’être* for test doubles is to give our tests access to information that’s otherwise inaccessible. Especially in the context of Java, the first association with “exposing information” might point toward giving tests access to read, or write to, other objects’ private fields. Though you might sometimes decide to go that way,⁴ the kind of information I’m thinking of is about the interactions between the code under test and its collaborators.

Let’s use our trusty `Car` example one more time to help you grasp this dynamic. Here’s a snippet of code inside the `Car` class, reproduced from listing 3.1:

```
public class Car {
    private Engine engine;

    public void start() {
        engine.start();
    }

    // rest omitted for clarity
}
```

³ Though programming a Lego Mindstorms robot to physically unplug your network cable would be infinitely more awesome.

⁴ It’s generally not a good idea to probe objects’ internals from outside. Whenever I feel the need to do something like that, it turns out that a missing abstraction was hiding in my design.

As you see from the listing, a Car starts its Engine when someone starts the Car. How would you test that this actually happens? You'd expose the private field to your test code and add a method to Engine for determining whether the engine is running. But what if you don't want to do that? What if you don't want to litter your production code with methods that are only there for testing purposes?

By now you've probably guessed that the answer is a test double. By substituting a test double for the Car's Engine, you can add that for-testing-only method to *test code* and avoid littering your production code with an `isRunning()` method that will never be used in production. In test code, that approach might look like the following:

Listing 3.3 A test double can deliver the inside scoop

```
public class CarTest {
    @Test
    public void engineIsStartedWhenCarStarts() {
        TestEngine engine = new TestEngine();
        new Car(engine).start();
        assertTrue(engine.isRunning());
    }
}

public class TestEngine extends Engine {
    private boolean isRunning;

    public void start() {
        isRunning = true;
    }

    public boolean isRunning() {
        return isRunning;
    }
}
```

As you can see in the listing, our sample test configures the Car with a test double **1**,⁵ starts the Car, and uses the test double's knowledge to verify that the engine was started **2** as expected. Let me emphasize that `isRunning()` is *not* a method on the Engine—it's something we've added to TestEngine to reveal information that a regular Engine doesn't.⁶

All right. You now understand the most common reasons for wanting to reach for a test double. That means it's time to look at the different types of test doubles and the kinds of advantages each has to offer.

3.2 Types of test doubles

You've seen a number of reasons to use test doubles, and we've hinted at there being multiple types of test doubles to choose from. It's time we take a closer look at those types. Figure 3.3 illustrates four types of objects that fall under this umbrella term.

⁵ This style of passing dependencies through a constructor is called *constructor injection*.

⁶ Speaking of test doubles extending the real object's interface, there's no reason why you couldn't add a method like `assertIsRunning()` on TestEngine!

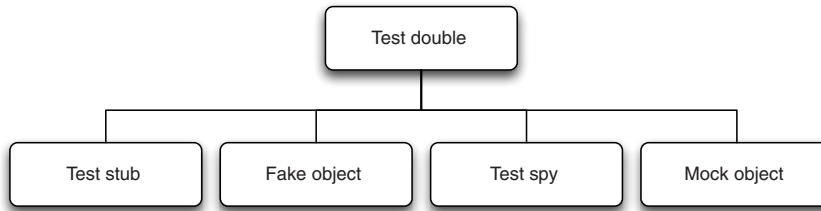


Figure 3.3 There are four types of objects we clump under the umbrella term *test double*.

Now that we’ve laid out the taxonomy around test doubles, let’s establish what they are, how they differ, and for which of the aforementioned purposes they’re typically applied. Let’s start with the simplest of them all.

3.2.1 Test stubs are unusually short things

My dictionary offers the following definition for a *stub*:

stub: (*noun*) a truncated or unusually short thing.

That turns out to be an accurate definition for what test stubs are. A test stub’s (or just *stub* for short) purpose is to stand in for the real implementation with the simplest possible implementation. The most basic example of such an implementation would be an object with all of its methods being one-liners that return an appropriate default value.

Say you’re working on code that’s supposed to generate an audit log of its operations by writing onto a remote log server, accessible through an interface named `Logger`. Assuming the `Logger` interface only defined one method for producing such logs, our test stub for the `Logger` interface might look like this:

```
public class LoggerStub implements Logger {
    public void log(LogLevel level, String message) { }
}
```

Notice how the `log()` method doesn’t do anything? That’s a typical example of what a stub does—nothing. After all, you’re stubbing out the real `Logger` implementation precisely because you don’t care about the log when testing something completely different, so why log anything? But sometimes doing nothing isn’t enough. For example, if the `Logger` interface also defines a method for determining the currently set `LogLevel`, your stub implementation might end up looking like this:

```
public class LoggerStub implements Logger {
    public void log(LogLevel level, String message) {
        // still a no-op
    }

    public LogLevel getLogLevel() {
        return LogLevel.WARN; // hard-coded return value
    }
}
```

Whoa! We're hard-coding the `getLogLevel()` method on this class to always return `LogLevel.WARN`. Isn't that *wrong*? In most cases that's perfectly OK. After all, we're using a test stub as a stand-in for the real `Logger` for three excellent reasons:

- 1 Our test isn't interested in what the code under test is logging.
- 2 We don't have a log server running so it would fail miserably anyway.
- 3 We don't want our test suite to generate gazillion bytes of output on the console either (let alone writing all of that data to a file).

In short, a stub implementation of `Logger` is a perfect match for what we need.

There are times when a simple hard-coded return statement and a bunch of empty void methods won't do the trick. Sometimes you need to fill in at least some behavior, and sometimes you need your test double to behave differently depending on the kind of messages it receives. In those situations you might turn to a *fake object* instead.

3.2.2 Fake objects do it without side effects

Compared to test stubs, a fake object is a more elaborate variation of a test double. Whereas a test stub can return hard-coded return values and each test may instantiate its own variation to return different values to simulate a different scenario, a fake object is more like an optimized, thinned-down version of the real thing that replicates the behavior of the real thing, but without the side effects and other consequences of using the real thing.

The canonical example of a case where you might pull out a fake object is persistence. Say your application architecture is such that persistence services are provided by a number of repository objects that know how to store and look up certain types of objects. Here's an example of the kind of API such a repository might provide:

```
public interface UserRepository {  
    void save(User user);  
    User findById(long id);  
    User findByUsername(String username);  
}
```

Without some kind of a test double, your tests for application code that uses a repository object would all try to hit a real database somewhere. You might be well off stubbing out this `UserRepository` interface to return exactly what your test wants. But simulating more complicated scenarios might prove to be complicated. On the other hand, the `UserRepository` interface is simple enough so that you could implement a stupid-simple, in-memory database on top of basic data types. The next listing serves as an example:

Listing 3.4 Fake objects aren't necessarily difficult to implement

```
public class FakeUserRepository implements UserRepository {  
    private Collection<User> users = new ArrayList<User>();  
  
    public void save(User user) {  
        if (findById(user.getId()) == null) {
```

```

        users.add(user);
    }
}

public User findById(long id) {
    for (User user : users) {
        if (user.getId() == id) return user;
    }
    return null;
}

public User findByUsername(String username) {
    for (User user : users) {
        if (user.getUsername().equals(username)) {
            return user;
        }
    }
    return null;
}
}

```

The good thing about this kind of an alternative implementation as a substitute for the real thing is that it quacks and wobbles like a duck, but it wobbles much faster than a real duck—even if you’d loop through a 50-item list every time someone wants to look up a `User`.

Test stubs and fake objects are often a lifesaver, as you can throw them in to replace the real thing that’s slow and depends on all kinds of things you don’t have while running tests. These two basic test doubles aren’t always sufficient, though. Every now and then you find yourself facing a brick wall you wish you could see through—in order to verify that your code behaves like it should. In those cases you might reach for a *test spy* instead.

3.2.3 *Test spies steal your secrets*

How would you test the following method?

```
public String concat(String first, String second) { ... }
```

Most people would say, pass in this and that and check that the return value is such and such. And that would likely make a lot of sense. After all, the return value being correct is what you’re most interested in about this method. Now, how about the following method—how would you test it?

```
public void filter(List<?> list, Predicate<?> predicate) { ... }
```

There’s no return value here to assert against. The only thing this method does is take in a list and a predicate and filter out from the given list all the items that don’t fulfill the given predicate. In other words, the only way to verify that this method works as it should is by examining the list afterward. It’s like a cop going undercover and reporting afterward what he observed.⁷ Often you can do this without a test double. In this example you can ask the `List` object whether it contains what you expect it to contain.

⁷ I admit—I’ve watched too much television—but test spies *are* just like that.

Where a test double—a test spy, since that’s what we’re discussing now—comes in handy is when none of the objects passed in as arguments can reveal through their API what you want to know. The following listing shows an example where this is the case.

Listing 3.5 Case for a test spy—the arguments don’t provide enough intel for a test

```
public class DLog {
    private final DLogTarget[] targets;

    public DLog(DLogTarget... targets) {
        this.targets = targets;
    }

    public void write(Level level, String message) {
        for (DLogTarget each : targets) {
            each.write(level, message);
        }
    }
}

public interface DLogTarget {
    void write(Level level, String message);
}
```

1 DLog is given a number of DLogTargets

2 Each target receives the same message

3 DLogTarget only defines the write() method

Let’s walk through the scenario in the previous listing. The object under test is a distributed log object, `DLog`, which represents a group of `DLogTargets` 1. When the `DLog` is written to, it should write that same message 2 to all of its `DLogTargets`. From the test’s point of view, things are awkward, as the `DLogTarget` interface only defines one method—`write()` 3—and that none of the real implementations of `DLogTarget`, `ConsoleTarget`, and `RemoteTarget` provides a method that would reveal whether a given message has been written to it.

Enter the test spy. The next listing illustrates how a smart programmer whips up his own secret agent and puts her to work.

Listing 3.6 Test spies are often simple to implement

```
public class DLogTest {
    @Test
    public void writesEachMessageToAllTargets() throws Exception {
        SpyTarget spy1 = new SpyTarget();
        SpyTarget spy2 = new SpyTarget();
        DLog log = new DLog(spy1, spy2);
        log.write(Level.INFO, "message");
        assertTrue(spy1.received(Level.INFO, "message"));
        assertTrue(spy2.received(Level.INFO, "message"));
    }
}

private class SpyTarget implements DLogTarget {
    private List<String> log = new ArrayList<String>();

    @Override
    public void write(Level level, String message) {
        log.add(concatenated(level, message));
    }
}
```

1 Spies sneak in

3 Spies report back

2 Make write() do some bookkeeping

```

boolean received(Level level, String message) {
    return log.contains(concatenated(level, message));
}

private String concatenated(Level level, String message) {
    return level.getName() + ": " + message;
}
}

```

← Let test ask for what it wants to know

That's all there is to test spies. You ❶ pass them in like any other test double. Then you make the spy ❷ keep track of which messages it was sent and ❸ let the test debrief the spy afterward, asking whether a given messages was received. Neat!

In short, test spies are test doubles built to record what's happening with them so that the test will know afterward what happened. Sometimes we take this concept farther and our test spies become full-blown *mock objects*. If test spies are like undercover cops, then mock objects are like having a remote-controlled cyborg infiltrate the mob. This might need some explanation...

3.2.4 Mock objects object to surprises

A mock object is a special kind of test spy. It's an object that's configured to behave in a certain way under certain circumstances. For example, a mock object for the `UserRepository` interface might be told to return `null` when `findById()` is invoked with the parameter `123` and to return a given instance of `User` when `findById()` is invoked with `124`. At this point we're basically talking about stubbing certain method calls, considering their parameters.

But mock objects can be much more precise by failing the test as soon as something unexpected happens. For instance, assuming we've told the mock object how it should behave when `findById()` is called with `123` or `124`, it'll do exactly as told when those calls come in. For any other invocation—whether it's to a different method or to `findById()` with a parameter other than what we've told the mock object about—the mock will throw an exception, effectively failing the test. Similarly, the mock will complain if `findById()` is called too many times—unless we've told it to allow any number of invocations—and it will complain if an expected interaction never took place.

Mock object libraries such as JMock, Mockito, and EasyMock are mature tools that a test-infected programmer can realize a lot of power from. Each of these libraries has its own style of doing things, but mostly you can do all the same stuff with any of these libraries.

This isn't intended to be a comprehensive tutorial on mock object libraries *du jour*, but let's take a quick look at the example in the next listing, which illustrates more concretely how you might make use of one such library. We're using JMock here because I happened to have an active project ongoing where JMock was used.

Listing 3.7 JMock lets you configure mock objects at runtime

```

public class TestTranslator {
    protected Mockery context;

    @Before
    public void createMockery() throws Exception {
        context = new JUnit4Mockery();
    }

    @Test
    public void usesInternetForTranslation() throws Exception {
        final Internet internet = context.mock(Internet.class);
        context.checking(new Expectations() {{
            one(internet).get(with(containsString("langpair=en%7Cfi")));
            will(returnValue("{\\"translatedText\\":\\"kukka\\"}"));
        }});
        Translator t = new Translator(internet);
        String translation = t.translate("flower", ENGLISH, FINNISH);
        assertEquals("kukka", translation);
    }

    ...
}

```

For such a short snippet of test code, this sample features a lot of typical constructs for a test that uses a mock object library. To start, we're telling the library to create a mock object for a given interface.

That somewhat awkward block inside `context.checking()` is where the test is teaching the mock `Internet` what kind of interactions it should expect and what to do when those interactions take place. In this case, we're saying we expect one call to its `get()` method with a parameter that contains the string `"langpair=en%7Cfi"` and that the mock should respond to that invocation by returning the provided string.

Finally, we pass the mock object on to the `Translator` object we're testing, exercise the `Translator`, and assert that the `Translator` provides the correct translation for our scenario.

That's not all the asserting we're doing here, though. As we established earlier, mock objects can be strict about the expected interactions actually taking place. In our case of the mock `Internet`, the mock is effectively asserting that it received exactly one call to the `get()` method with a parameter that did contain the defined substring.

3.3 Guidelines for using test doubles

Test doubles are the programmer's tools, just like hammers and nails are for a carpenter. There are appropriate ways to bang a nail with a hammer and there are less appropriate ways—and it's good to be able to identify which is which.

Let's start with what I consider perhaps the most important guideline to keep in mind when you're about to reach for a test double—picking up the right tool from your toolbox.

3.3.1 Pick the right double for the test

There are many kinds of test doubles to choose from and they all seem to be slightly different. What are their sweet spots, and which one should we choose?

There's no hard and fast rule for much of this stuff, but in general you should probably mix and match a little. What I mean by that is that there are certain cases where you only want "an object that returns 5," and other cases where you're also keen on knowing that a certain method was invoked. Sometimes, both of these interests are combined within a single test, leading you to use both a stub or a fake and a mock object in the same test.

Having said that, there's no clear-cut principle here beyond picking the option that results in the most readable test. I can't resist the temptation to try to impose some logic and heuristics on this problem of when to use which:

- If you care about a certain interaction taking place in the form of method calls between two objects, the odds are you'll want to go with a mock object.
- If you decided to go with a mock object but your test code ends up looking less pretty than you'd like, see whether a simple, hand-made test spy would do the trick.
- If you only care about the collaborator objects being there and feeding the object under test with responses your test is in control of, go with a stub.⁸
- If you want to run a complex scenario that relies on a service or component that's unavailable or unusable for your test's purposes, and your quick attempt at stubbing all of those interactions grinds to a halt or results in messy test code that's going to be a pain to maintain, consider implementing a fake instead.
- If none of the above sufficiently describes your particular situation at hand, toss a coin—heads is a mock, tails is a stub, and if the coin is standing on its side, you have my permission to whip up a fake.

If that list is too much to memorize, don't fret. J.B. Rainsberger, author of *JUnit Recipes* (Manning, 2004), has an easy to remember rule for picking the right type of test double:

Stub queries; mock actions.

There! Now that we're on a roll with heuristics for when to use which kind of test double, let's continue with guidelines for *how* to use them.

3.3.2 Arrange, act, assert

There's something to be said about coding conventions. The problem with standards is that there are so many of them. Luckily, when it comes to structuring your unit tests, there's a fairly established practice that most programmers consider sensible. It's called *arrange-act-assert* and it basically means that you want to organize your tests such

⁸ If you decided to go with a stub but your test also has a mock object in it, consider using your chosen mock object library to create that stub—it can do that, too, and the resulting test code might be easier for the eye.

that you first arrange the objects used in the test, then trigger the action and make assertions about the outcome last.

Let's take another look at the test in listing 3.7, reproduced in the next listing, and see how it complies with this convention of organizing a test method.

Listing 3.8 Arrange-act-assert makes for a clear structure

```
@Test
public void usesInternetForTranslation() throws Exception {
    final Internet internet = context.mock(Internet.class);
    context.checking(new Expectations() {{
        one(internet).get(with(containsString("langpair=en%7Cfi")));
        will(returnValue("{\"translatedText\":\"kukka\"}"));
    }});
    Translator t = new Translator(internet);

    String translation = t.translate("flower", ENGLISH, FINNISH);
    assertEquals("kukka", translation);
}
```

① Arrange

② Act

③ Assert

Note how I've added whitespace between the three blocks of code. This is to underline the role of the three blocks.⁹

The first five lines in this test are all about ① arranging the collaborator objects to be used in the test. In this case we're only dealing with one mock object for the `Internet` interface, but it's not uncommon to set up more than one collaborator in the beginning of a test. Then there's the object under test, `Translator`—its initialization is also part of arranging objects for the test.¹⁰

In the next block of code we invoke the translation ② (the behavior under test) and eventually make assertions ③ about the expected outcome, whether it's direct output or a bunch of side effects.

GIVEN, WHEN, THEN The *behavior-driven development* movement has popularized the use of a vocabulary and structure similar to arrange-act-assert: Given (a context), When (something happens), Then (we expect certain outcome). The idea is to specify desired behavior in a more intuitive language and, though arrange-act-assert does make for an easy mnemonic, the given-when-then alternative flows better and tends to direct one's thinking more naturally toward behavior (rather than implementation details).

This structure is fairly common and helps keep our tests focused. If one of the three sections feels "big," it's a sign that the test might be trying to do too much and needs more focus. While we're on the topic, let's talk about what tests should focus on.

⁹ Though whitespace can definitely be overused, a consistent use of whitespace can help programmers see the structure of tests more easily.

¹⁰ Technically, setting an expectation on a mock object is an act of verification—an assertion rather than mere arrangement. But in this case we're using the mock object as a stub. Some might call heresy on this—and they'd be right.

3.3.3 Check for behavior, not implementation

People make mistakes. One mistake that programmers new to mock object libraries tend to make often is overly detailed expectations for their mock objects. I'm talking about those tests where every possible object involved in the test is a mock and every single method call between those objects is strictly specified.

Yes, testing gives us certainty in the sense that it'll break and sound the alarm as soon as anything changes. And therein lies the problem—even the smallest change that may be irrelevant from the perspective of what the test is supposed to verify will break the test. It's like hammering so many nails into that two-by-four, nailing it down so tight that the poor thing looks like Swiss cheese squashed into a bed of nails.

The fundamental problem with such tests is created by lack of focus. A test should test just one thing and test it well while communicating its intent clearly. Looking at an object you're testing, you need to ask yourself what is the desired behavior you want to verify, and what is an implementation detail that isn't necessary to nail down in our test?

The desired, intended behavior is something you should configure your mock objects to expect. The implementation detail is something you should seek to provide by stubs or non-strict mock objects that don't mind if an interaction never happens or if it happens more than once.

Check for behavior, not implementation. That should be always in your mind when you pull out your favorite mock object library. Speaking of which...

3.3.4 Choose your tools

Java programmers are at an advantage when it comes to mock object libraries—there are plenty to choose from. As I said earlier, you can do more or less the same stuff with any of the leading libraries, but they have slight differences in APIs and may have some distinct features that tip the scale in a particular direction for your particular needs.

Perhaps the most distinctive of such features is Mockito's separation of *stubbing* and *verification*. This requires elaboration, so let's look at an example where I've used Mockito to write the same test we saw implemented with JMock:

```
@Test
public void usesInternetForTranslation() throws Exception {
    final Internet internet = mock(Internet.class);
    when(internet.get(containsString("langpair=en%7Cfi")))
        .thenReturn("{\"translatedText\":\"kukka\"}");
    Translator translator = new Translator(internet);
    String result = translator.translate("flower", ENGLISH, FINNISH);
    assertEquals("kukka", result);
}
```

Mockito's API is more compact than JMock's. Other than that, looks more or less the same, right? Yes, except that this test written with Mockito is just stubbing the `get()` method—it would pass with flying colors even if that interaction never happened. If we truly were interested in verifying that the `Translator` uses `Internet`, we'd add another call to Mockito's API for checking that:

```
verify(internet).get(argThat(containsString("langpair=en%7Cfi")));
```

Or, if we're not feeling that precise:

```
verify(internet).get(anyString());
```

When it comes to mock object library APIs it's often a matter of preference. But Mockito has a clear advantage for a style of testing that relies mostly on stubbing—and that may or may not be an advantage in your particular context. This is key—test code is just as important to keep readable and concise, maintainable today and tomorrow. It's worth stopping for a moment to weigh the alternatives on the table and make an informed choice of tooling.

Let me borrow from J.B. again to crystallize the difference between JMock and Mockito's approach and sweet spot:¹¹

When I want to rescue legacy code, I reach for Mockito. When I want to design for new features, I reach for JMock.

Different central assumptions of JMock and Mockito make each one better at its respective task. By default, JMock assumes that a test double (a “mock”) expects clients not to invoke anything at any time. If you want to relax that assumption, then you have to add a stub. On the other hand, Mockito assumes that a test double (sadly, also a “mock”) allows clients to invoke anything at any time. If you want to strengthen that assumption, then you have to verify a method invocation. This makes all the difference.

Regardless of which library you decide to go with today, our third and final guideline for using test doubles is as relevant as ever.

3.3.5 Inject your dependencies

In order to be able to use test doubles, you need a way to substitute them for the real thing. When it comes to dependencies—collaborator objects that you'd like to replace for the purposes of testing—our guideline says that they shouldn't be instantiated in the same place they're used. In practice, that means storing such objects as private members or acquiring them, for example, through factory methods.

Once you've isolated the dependency, you need access to it. You could break encapsulation with visibility modifiers—making private things public or package private—or use the Reflection API to assign a test double to a private field. That could turn ugly real soon. A better option would be to employ *dependency injection* and pass dependencies into the object from outside, typically using *constructor injection* like we did in the Translator example.¹²

We've now talked about test doubles for a long time and I'm itching to move on to part 2, so let's wrap this chapter up with a recap of what we've learned about test doubles.

¹¹ J.B. Rainsberger blog, “JMock v. Mockito, but Not to the Death,” October 5, 2010, <http://mng.bz/yW2m>.

¹² Note that dependency injection doesn't necessarily mean that you must use a dependency injection *framework*. Simply passing in dependencies through constructors already goes a long way!

3.4 Summary

We began exploring the topic of test doubles by looking at the reasons why you might want to reach for a test double. The common denominator behind these reasons is the need to isolate the code under test so that you can simulate all scenarios and test all behaviors the code should exhibit.

Sometimes the reason for using a test double stems from the need to make your tests run faster. Test doubles, with their straightforward implementation, often run a magnitude or two faster than the implementation they're substituting for. Sometimes, the code you're testing depends on random or otherwise nondeterministic behavior, such as the time of day. In these cases, test doubles ease our test-infected lives by turning the unpredictable into predictable.

You also learned that test doubles may be the only feasible way to simulate certain special situations and to verify the desired behavior of objects without altering the design of your production code to expose details and information merely for the sake of testability.

Having established the benefits test doubles can offer, we proceeded to study the differences between four types of test doubles: test stubs, fake objects, test spies, and mock objects. Test stubs in their extreme simplicity are perhaps best suited for cutting off irrelevant collaborators. Fake objects offer a lightning-fast alternative for situations where the real thing is difficult or cumbersome to use. Test spies can be used to access information and data that's otherwise hidden, and mock objects are like test spies on steroids, adding the ability to dynamically configure their behavior and to verify that the interactions you expect to happen do in fact take place.

We concluded the chapter with a tour of some basic guidelines for using test doubles, starting from when and in what kind of situations a given type of test double would likely be a good match. Especially when using mock objects, it's important to avoid nailing down the implementation too tightly. Your mocks should verify that the desired behavior is there and try to be as forgiving as possible as to how exactly that behavior was achieved.

As usual, tools can be of great help, and it pays to evaluate which libraries are best suited for your particular use and style of using mock objects. Finally, it can make a world of difference as far as testability goes to inject your dependencies from outside rather than hard-wiring them inside the code you're trying to test.

This chapter on test doubles concludes part 1. We've now explored the foundations of writing good tests. From understanding the benefits of having tests, to knowing the kind of properties you should seek from the tests you write, to handling one of the most essential testing tools in a programmer's toolbox—test doubles—you have now mastered enough basics and you're ready to start honing your skills further. Especially, you're ready to start training your sense for the many ways a test can fail, from being good to being a real head-puzzler, a smelly sock on the floor, or even a downright maintenance burden. It's time to turn to part 2 and focus our attention on smells.

Test smells, that is.

Part 2

Catalog

The goal of part 2 is to help us better identify and fix problems with our test code, and we’re taking a “catalog”-style approach to accomplishing that. This catalog is one of *test smells*—traits and attributes we often encounter in test code that create or aggravate problems in terms of maintainability, readability, and trustworthiness.

We are wallowing in this puddle of anti-patterns instead of walking through a list of good practices because I’ve found it more effective to teach programmers to identify problems with their tests and to rectify them. After all, if we only know what a good test looks like, we’re still blind to the many flaws our code may have.

My good friend Bas Vodde likes to say that if you remove all the test smells you can identify, what’s left is likely to be a pretty good test. Besides, the design principles and guidelines for good code are just as valid for test code as they are for production code—which means we already have a body of knowledge to refer to in determining what kind of properties our test code should exhibit.

Our catalog of test smells is divided into three chapters, each collecting a group of test smells around a theme. Chapter 4 presents smells that suggest issues with the readability of our tests. Chapter 5 continues with advice on keeping our tests from becoming a maintenance issue. Chapter 6 concludes this part with a set of test smells that often correlate with flaky or otherwise unreliable tests.

This division of test smells into the three chapters is based on what each smell *mostly* or most strongly impacts—some could easily be featured in multiple chapters! Many of the smells in this catalog are also very much interconnected and often appear in code bases hand in hand. We’ll be sure to point out these

connections where appropriate. Some of these smells are the opposite extremes of an axis—extremes are rarely optimal, after all. We'll be sure to point out those connections, too.

All right. Time to roll up our sleeves and dig into the catalog.

4

Readability

In this chapter

- Test smells around assertions
- Test smells around information scatter within the code base
- Test smells around excess or irrelevant detail

Tests are the programmer's way of expressing and validating assumptions and intended behavior of the code. Reading the tests for a given mass of code should provide the programmer with an understanding of what the code *should* do. Running those tests should tell the programmer what the code actually *does*.

Assertions play a key role in deciphering the behavior of code. Though all of the code that makes up a test tells a story about how the code under test is expected to behave, it's the assertion statements that contribute the punchlines. It is the assertions that we study to be able to tell whether or not the code passes expectations. In a similar vein, the test framework uses those same assertions to check our assumptions during test execution.

Consequently many of the issues we discuss in this chapter revolve around assertions, but equally important for readability is all that other code in your tests—the code that arranges the fixture and the code that invokes or triggers actions on the code under test.

To summarize, in this chapter we're focusing on a wide range of smells and problems with test code exhibiting undesirable properties. Those problems add cognitive load on the programmer, making it harder to read and understand the test's intent and what it's testing.

After all, reading test code shouldn't be hard work. You shouldn't have to strain your brain cells and stretch the limits of your short-term memory to grasp what's going on when you walk through a test. Yet every now and then we bump into that nasty bugger we've seen oh-so-many times before. The nasty bugger that's only 10 lines long but looks like a mixture of Swahili, Hebrew, and ancient scripture.

There are plenty of test smells like that, but a particularly common one is plagued with what I call the *primitive assertion*. That's a great place to start our catalog of test smells, so let's begin with that.

4.1 Primitive assertions

Assertions are supposed to express an assumption or intent. They're supposed to be statements about the behavior of the code. The problem with primitive assertions is that they don't seem to make sense because the rationale and intent of the assertion is hidden somewhere behind seemingly meaningless words and numbers, making them difficult to understand—and to validate the assertion's correctness.

In other words, a primitive assertion is one that uses more primitive elements than the behavior it's checking. Let's look at an example that exhibits this smell.

4.1.1 Example

Our example, shown in listing 4.1, presents a test for a *grepping* utility. Grepping essentially means processing some kind of textual input line by line and including or excluding lines that contain a certain substring or pattern. Tests are supposed to be concrete examples of the functionality your objects provide, so let's look at the test and see if that clears up what our grep utility should do. If only this test didn't suffer from primitive assertions!

Listing 4.1 Primitive assertions are unnecessary additions to your cognitive load

```
@Test
public void outputHasLineNumbers() {
    String content = "1st match on #1\nand\n2nd match on #3";
    String out = grep.grep("match", "test.txt", content);
    assertTrue(out.indexOf("test.txt:1 1st match") != -1);
    assertTrue(out.indexOf("test.txt:3 2nd match") != -1);
}
```

So, what's going on in the test? First, we define a literal string `content` to represent a piece of input, then we invoke the `grep` utility, and then we assert something about the `grep`'s output. That *something* is the issue here. It's not clear what we're asserting because the assertion is too primitive—it doesn't speak the same language as the rest of the test.

In more concrete terms, what we’re doing in this assertion is calculating the index of another literal string from the output and comparing it to -1; if the index is -1, the test fails. The test’s name is `outputHasLineNumbers` and apparently, for the specific invocation of `grep()` in listing 4.1, the output should include the filename being grepped on every line in the output, concatenated with the line number.

Unfortunately we have to go through this whole thinking process in order to understand why we’re calculating an index, why we’re looking for `test.txt:1` and not something else, why we’re comparing it to -1, and whether the test is failing when the index is -1 or when it’s not -1? It’s not rocket science to figure this out, but it’s cognitive work that our brains shouldn’t need to do.

4.1.2 What to do about it?

The essence of a primitive assertion is that the level of abstraction is too low when it comes to asserting stuff. With this in mind, we can find at least a couple of potential improvements to the preceding example.

The first improvement relates to getting rid of the complicated logic around the magic number, -1. Instead of combining the work horse that is `assertTrue` with a `!=` comparison, let’s make the desired logic clear by reformulating it using the `assertThat` syntax introduced in JUnit 4.4, as illustrated next.

Listing 4.2 JUnit 4’s `assertThat` syntax helps in making assertions human-readable

```
@Test
public void outputHasLineNumbers() {
    String content = "1st match on #1\nand\n2nd match on #3";
    String out = grep.grep("match", "test.txt", content);
    assertThat(out.indexOf("test.txt:1 1st match"), is(not(-1)));
    assertThat(out.indexOf("test.txt:3 2nd match"), is(not(-1)));
}
```

We use the `assertThat` syntax and the accompanying Hamcrest matcher utilities, `is` and `not`. This small change relieves our brains from the burden of figuring out whether we expect the index to be -1. The intent is now clearly expressed in plain English.¹

The second potential improvement relates to the abstraction level of the standard Java API we’re using. More specifically, we should reconsider how we determine whether a given substring is found within a string. In the case of identifying that a given substring is found in the `grep`’s output, the concept of an “index” and the magic number -1 are concepts from the wrong abstraction level. Following through with this change, our test would probably evolve to something like the next listing.

¹ Hamcrest (<https://github.com/hamcrest/JavaHamcrest>) is an API and a collection of *matcher* implementations that other libraries such as JUnit can use to match objects against a variety of expectations.

Listing 4.3 Expressing expectations at the right level of abstraction

```
@Test
public void outputHasLineNumbers() {
    String content = "1st match on #1\nand\n2nd match on #3";
    String out = grep.grep("match", "test.txt", content);
    assertTrue(out.contains("test.txt:1 1st match"));
    assertTrue(out.contains("test.txt:3 2nd match"));
}
```

Here we're using `contains` from `java.lang.String` instead of the more down-to-the-metal way of `indexOf`. Again, there's less cognitive load for the programmer reading this code.

Finally, we could consider a combination of these improvements, using the more literate approach of `assertThat`, as well as preferring `String#contains` over `String#indexOf`. The following listing shows what that would look like.

Listing 4.4 Combining the use of `assertThat` with the right level of abstraction

```
@Test
public void outputHasLineNumbers() {
    String content = "1st match on #1\nand\n2nd match on #3";
    String out = grep.grep("match", "test.txt", content);
    assertThat(out.contains("test.txt:1 1st match"), equals(true));
    assertThat(out.contains("test.txt:3 2nd match"), equals(true));
}
```

Comparing this version with listing 4.3, you can see that our last improvement is more verbose than our second improvement alone, but it's definitely more concise and expressive than the first improvement (listing 4.2) alone. When weighing alternatives to expressing intent in test code, you should keep in mind that the nature and purpose of tests puts a higher value on readability and clarity than, say, code duplication or performance. Often you'll find that a little extra verbosity can go a long way toward expressiveness, readability, and maintainability.

Knowing your tools well helps in finding a good way to express your intent. In this example, knowing that JUnit includes a small collection of convenient Hamcrest matchers, including `org.junit.JUnitMatchers#containsString()`, might lead you to evolve your test toward this listing.

Listing 4.5 One more slight improvement by making proper use of Hamcrest matchers

```
@Test
public void outputHasLineNumbers() {
    String content = "1st match on #1\nand\n2nd match on #3";
    String out = grep.grep("match", "test.txt", content);
    assertThat(out, containsString("test.txt:1 1st match"));
    assertThat(out, containsString("test.txt:3 2nd match"));
}
```

Which of these styles matches your sense of aesthetic best? That's probably the approach you'll want to take. But regardless of whether you end up using `assertThat`, `assertTrue`, or `assertFalse` (or even `assertEquals`), you'll want to express your assertion with the language and vocabulary of the functionality you're testing.

4.1.3 Summary

Whenever you see an assertion that involves comparisons such as `!=` or `==`, especially to magic numbers like `-1` or `0`, ask yourself whether the level of abstraction is right. If an assertion doesn't make sense immediately, it's likely a primitive assertion and calls for refactoring.

This test smell is essentially the identical twin of the general *primitive obsession* code smell, which refers to the use of primitive types to represent higher-level concepts. Think of a phone number represented as a `String`, a mobile phone's enabled features encoded into a `byte`, or the mobile broadband subscription described as a pair of `Date` objects. When writing tests, you should focus on expressing these concepts at their level, not in terms of their implementation.

Sometimes your tests aren't obsessed with using language primitives but instead are obsessed with detail. I'm talking about the unit test equivalent of the office micro-manager. Tests that suffer from that kind of obsession often sport overly detailed assertions—so detailed that you might call them hyperassertions.

4.2 Hyperassertions

A *hyperassertion* is one that's so scrupulous in nailing down every detail about the behavior it's checking that it becomes brittle and hides its intent under its overwhelming breadth and depth. When you encounter a hyperassertion, it's hard to say what exactly it's supposed to check, and when you step back to observe, that test is probably breaking far more frequently than the average. It's so picky that any change whatsoever will cause a difference in the expected output.

Let's make this discussion more concrete by looking at an example test that suffers from this condition.

4.2.1 Example

The following example is my own doing. I wrote it some years back as part of a sales-presentation tracking system for a medical corporation. The corporation wanted to gather data on how presentations were carried out by the sales fleet that visited doctors to push their products. Essentially, they wanted a log of which salesman showed which slide of which presentation for how many seconds before moving on.

The solution involved a number of components. There was a plug-in in the actual presentation file, triggering events when starting a new slide show, entering a slide, and so forth—each with a timestamp to signify when that event happened. Those events were pushed to a background application that appended them into a log file. Before synchronizing that log file with the centralized server, we transformed the log

file into another format, preprocessing it to make it easier for the centralized server to chomp the log file and dump the numbers into a central database. Essentially, we calculated the slide durations from timestamps.

The object responsible for this transformation was called a `LogFileTransformer` and, being test-infected as I was, I'd written some tests for it. The following listing presents one of those tests—the one that suffered from hyperassertion—along with the relevant setup. Have a look and see if you can detect the culprit.

Listing 4.6 Hyperassertion makes a test brittle and opaque

```
public class LogFileTransformerTest {

    private String expectedOutput;
    private String logFile;

    @Before
    public void setUpBuildLogFile() {
        StringBuilder lines = new StringBuilder();
        appendTo(lines, "[2005-05-23 21:20:33] LAUNCHED");
        appendTo(lines, "[2005-05-23 21:20:33] session-id###SID");
        appendTo(lines, "[2005-05-23 21:20:33] user-id###UID");
        appendTo(lines, "[2005-05-23 21:20:33] presentation-id###PID");
        appendTo(lines, "[2005-05-23 21:20:35] screen1");
        appendTo(lines, "[2005-05-23 21:20:36] screen2");
        appendTo(lines, "[2005-05-23 21:21:36] screen3");
        appendTo(lines, "[2005-05-23 21:21:36] screen4");
        appendTo(lines, "[2005-05-23 21:22:00] screen5");
        appendTo(lines, "[2005-05-23 21:22:48] STOPPED");
        logFile = lines.toString();
    }

    @Before
    public void setUpBuildTransformedFile() {
        StringBuilder file = new StringBuilder();
        appendTo(file, "session-id###SID");
        appendTo(file, "presentation-id###PID");
        appendTo(file, "user-id###UID");
        appendTo(file, "started###2005-05-23 21:20:33");
        appendTo(file, "screen1###1");
        appendTo(file, "screen2###60");
        appendTo(file, "screen3###0");
        appendTo(file, "screen4###24");
        appendTo(file, "screen5###48");
        appendTo(file, "finished###2005-05-23 21:22:48");
        expectedOutput = file.toString();
    }

    @Test
    public void transformationGeneratesRightStuffIntoTheRightFile()
        throws Exception {
        TempFile input = TempFile.withSuffix(".src.log").append(logFile);
        TempFile output = TempFile.withSuffix(".dest.log");
        new LogFileTransformer().transform(input.file(), output.file());
        assertTrue("Destination file was not created", output.exists());
    }
}
```

```

    assertEquals(expectedOutput, output.content());
}

// rest omitted for clarity
}

```

Did you see the hyperassertion? You probably did—there are only two assertions in there—but which is the culprit here and what makes it too hyper?

The first assertion checks that the destination file was created. The second assertion checks that the destination file’s content is what’s expected. Now, the value of the first assertion is questionable and it should probably be deleted. But the second assertion is our main concern—the hyperassertion:

```
assertEquals(expectedOutput, output.content());
```

This is a relevant assertion in the sense that it verifies exactly what the name of the test implies—that the right stuff ended up in the right file. The problem is that the test is too broad, resulting in the assertion being a wholesale comparison of the whole log file. It’s a thick safety net, that’s for sure, as even the tiniest of changes in the output will fail the assertion. And therein lies the problem.

A test that has never failed is of little value—it’s probably not testing anything. On the other end of the spectrum, a test that always fails is a nuisance. What we’re looking for is a test that has failed in the past—proving that it’s able to catch a deviation from the desired behavior of the code it’s testing—and that will break again if we make such a change to the code it’s testing.

The test in this example fails to fulfill this criterion by failing too easily, making it brittle and fragile. But that’s only a symptom of a more fundamental issue—the problem of being a hyperassertion. The small changes in the log file’s format or content that would break this test are valid reasons to fail it. There’s nothing intrinsically wrong about the assertion. The problem lies in the test violating a fundamental guiding principle for what constitutes a good test.

A test should have only one reason to fail.

If that principle seems familiar, it’s a variation of a well-known object-oriented design principle, the Single Responsibility Principle, which says, “A class should have one, and only one, reason to change.”² Now let’s clarify why this principle is of such importance.

Catching many kinds of changes in the generated output is good. But when the test does fail, we want to know why. In this example it’s difficult to tell what happened if this test, `transformationGeneratesRightStuffIntoTheRightFile`, suddenly breaks. In practice, we’ll always have to look at the details to figure out what had changed and broken the test. If the assertion is too broad, many of those details that break the test are irrelevant.

How should we go about improving this test?

² Robert C. Martin. *Agile Software Development: Principles, Patterns, and Practices*, (Prentice Hall, 2002).

4.2.2 What to do about it?

The first order of action when encountering an unduly detailed hyperassertion is to identify irrelevant details and remove them from the test. In this example, you might look at the log file being transformed and try to reduce the number of lines.

We want it to represent a valid log file and be elaborate enough for the purposes of the test. For example, the log file has timings for five screens. Maybe two or three would be enough? Could we get by with just one?

This question brings us to the next improvement to consider: splitting the test. Asking yourself how few lines in the log file you could get by with quickly leads to concerns about the test no longer testing x, y, or z. Here, x, y, and z are prime candidates for separate tests in their own right. The following listing presents one possible solution where each aspect of the log file and its transformation are extracted into separate tests.

Listing 4.7 Assertions focused on just one aspect are readable, less brittle

```
public class LogFileTransformerTest {

    private static final String END = "2005-05-23 21:21:37";
    private static final String START = "2005-05-23 21:20:33";
    private LogFile logFile;

    @Before
    public void prepareLogFile() {
        logFile = new LogFile(START, END);
    }

    @Test
    public void overallFileStructureIsCorrect()
        throws Exception {
        StringBuilder expected = new StringBuilder();
        appendTo(expected, "session-id###SID");
        appendTo(expected, "presentation-id###PID");
        appendTo(expected, "user-id###UID");
        appendTo(expected, "started###2005-05-23 21:20:33");
        appendTo(expected, "finished###2005-05-23 21:21:37");
        assertEquals(expected.toString(), transform(logFile.toString()));
    }

    @Test
    public void screenDurationsGoBetweenStartedAndFinished()
        throws Exception {
        logFile.addContent("[2005-05-23 21:20:35] screen1");
        String out = transform(logFile.toString());
        assertTrue(out.indexOf("started") < out.indexOf("screen1"));
        assertTrue(out.indexOf("screen1") < out.indexOf("finished"));
    }

    @Test
    public void screenDurationsAreRenderedInSeconds()
        throws Exception {
        logFile.addContent("[2005-05-23 21:20:35] screen1");
        logFile.addContent("[2005-05-23 21:20:35] screen2");
        logFile.addContent("[2005-05-23 21:21:36] screen3");
    }
}
```

Check that common headers are placed correctly

Check screen durations' place in the log

Check screen duration calculations

```

    String output = transform(logFile.toString());
    assertTrue(output.contains("screen1###0"));
    assertTrue(output.contains("screen2###61"));
    assertTrue(output.contains("screen3###1"));
}

// rest omitted for brevity

private String transform(String log) { ... }
private void appendTo(StringBuilder buffer, String string) { ... }
private class LogFile { ... }
}

```

This solution introduces a test helper class, `LogFile`, that establishes the standard *envelope*—the header and footer—for the log file being transformed, based on the given starting and ending timestamps. This allows the second and third test, `screenDurationsGoBetweenStartedAndFinished` and `screenDurationsAreRenderedInSeconds`, to append just the screen durations to the log, making the test more focused and easier to grasp. In other words, we delegate some of the responsibility for constructing the complete log file to `LogFile`. In order to ensure that this responsibility receives due diligence, the overall file structure is verified by the first test, `overallFileStructureIsCorrect`, in the context of the simplest possible scenario: an otherwise empty log file.

This refactoring has given us more focus by hiding details from each test that are irrelevant for that particular test. That’s also the downside of this approach—some of the details are hidden. In applying this technique, you must ask yourself what you value more: being able to see the whole thing in one place, or being able to see the essence of a test quickly.

I suggest that, most of the time when speaking of unit tests, the latter is more desirable, as the fine-grained, focused tests point you quickly to the root of the problem in case of a test failure. With all tests making assertions against the whole transformed log file, for example, a small change in the file syntax could easily break all of your tests, making it more difficult to figure out what exactly broke.

4.2.3 Summary

It’s good to be interested in the whole. But you can shoot yourself in the foot by making assertions too *hyper*. A hyperassertion cuts out too large a chunk of output and side effects for bit-to-bit comparison, which makes it harmful due to the resulting brittleness. The assertion fails if any small detail changes, regardless of whether that change is relevant to the interests of this particular test.

A hyperassertion also makes it difficult for the programmer to identify the intent and essence of the test. When you see a test that seems to bite off a lot, ask yourself what exactly you want to verify? Then, try to formulate your assertion in those terms.

The terms and words we use are crucial in conveying intent, but we’re talking about automated tests and, therefore, the programming language structures we use also deserve thought. As our next test smell points out, not all language structures are created equal.

4.3 Bitwise assertions

In the beginning of this chapter I described a primitive assertion as one that's so cryptic because of its low abstraction level that its intent and meaning remain a mystery to the reader. A *bitwise assertion* is a special case of primitive assertion that I feel warrants special mention because its elements are close to home for so many programmers and far from home for equally many programmers.

Perhaps an example will help concretize this hazard.

4.3.1 Example

Let's take a look at a concrete example that exhibits this particular smell. Consider the test in this listing.

Listing 4.8 Bitwise assertion, also known as “what did this operator do again?”

```
public class PlatformTest {  
    @Test  
    public void platformBitLength() {  
        assertTrue(Platform.IS_32_BIT ^ Platform.IS_64_BIT);  
    }  
}
```

What are we checking here? The assertion includes a description and it's asserting that the result of that `^` bit operation between two boolean values is true. What does that bit operation do? When would this assertion fail?

If you happen to work in a relatively low-level domain where bit operations are business as usual, you were probably quick to deduce that we're performing an XOR operation, which means that the assertion will fail if both sides of the bit operator have the same boolean value.

Now that we're all clear on what this test is doing, what's wrong with it?

The bit operator is what's wrong with this test. Bit operators are a powerful language feature for operating on bits and bytes. But in the context of this test, we're not in the domain of bits and bytes. We're in the domain of “we should be running on either a 32-bit or a 64-bit architecture” and that essence gets hidden behind the foreign bit operator.

That particular bit operator may be the perfect match for performing a concise assertion efficiently, but we're not in the business of optimizing test assertions. We're in the business of making sure that we're building the right thing right—that our code does what we expect it to do and that what we're expecting makes sense. For that purpose, there's bound to be a better way to express our intent than a bit operator.

4.3.2 What to do about it?

This time, the solution is simple. Replace the bit operator with one or more Boolean operators, expressing the expectations clearly one by one. The next listing presents one way to do it for our example test:

Listing 4.9 Prefer Boolean operators over bit operators

```
public class PlatformTest {  
    @Test  
    public void platformBitLength() {  
        assertTrue("Not 32 or 64-bit platform?",  
            Platform.IS_32_BIT || Platform.IS_64_BIT);  
        assertFalse("Can't be 32 and 64-bit at the same time.",  
            Platform.IS_32_BIT && Platform.IS_64_BIT);  
    }  
}
```

Yes, it's more verbose, but it's also more explicit about what we're stating with our assertions. The plain English messages are enormously helpful in this situation, and replacing the single bit operator with two simple explicit Boolean operators makes the logic more accessible to programmers who don't wear a binary watch.³

4.3.3 Summary

Bitwise assertions exhibit the same problems as other types of primitive assertions: their meaning is hard to pick up and our poor brains have to work overtime to figure out what's going on. Bit operators are a powerful language construct, and an essential one at that for many application domains. But when bit operators are used as a shorthand for expressing logic that isn't inherently about bits, we've stepped on a downward spiral. Leave bit operators for bit operations and strive to express higher-level concepts with language that belongs to that abstraction level.

4.4 Incidental details

What makes code readable is that it reveals its intent, purpose, and meaning promptly and squarely to the reader. When programmers scan a block of code, they're looking for the beef, and any sauce that gets in the way is unwanted. Sometimes our test code has too much sauce. We call that smell *incidental details*.

Let's look at an example again. This is a longer one. Take your time to figure out what this test is trying to verify.

4.4.1 Example

The following is another example from the JRuby project. JRuby has a module called `ObjectSpace`. `ObjectSpace` gives your program access to all the living objects in the runtime. For example, you can iterate through all objects of a given type. The code in the next listing presents a test for `ObjectSpace` that ensures this lookup by type works as it should. Let's take a look.

³ Wearing a binary watch does have a nerdy vibe to it, but I'm having a difficult enough time already dealing with time zones and setting the alarm clock while on the move.

Listing 4.10 This test tries to hide its beef. How quickly can you tell what it does?

```

public class TestObjectSpace {
    private Ruby runtime;
    private ObjectSpace objectSpace;

    @Before
    public void setUp() throws Exception {
        runtime = Ruby.newInstance();
        objectSpace = new ObjectSpace();
    }

    @Test
    public void testObjectSpace() {
        IRubyObject o1 = runtime.newFixnum(10);
        IRubyObject o2 = runtime.newFixnum(20);
        IRubyObject o3 = runtime.newFixnum(30);
        IRubyObject o4 = runtime.newString("hello");

        objectSpace.add(o1);
        objectSpace.add(o2);
        objectSpace.add(o3);
        objectSpace.add(o4);

        List storedFixnums = new ArrayList(3);
        storedFixnums.add(o1);
        storedFixnums.add(o2);
        storedFixnums.add(o3);

        Iterator strings = objectSpace.iterator(runtime.getString());
        assertSame(o4, strings.next());
        assertNull(strings.next());

        Iterator numerics = objectSpace.iterator(runtime.getNumeric());
        for (int i = 0; i < 3; i++) {
            Object item = numerics.next();
            assertTrue(storedFixnums.contains(item));
        }
        assertNull(numerics.next());
    }
}

```

So we're testing the behavior of an `ObjectSpace`, and we do that by first “adding” objects of certain types to it and then checking that we get those same objects back from the `ObjectSpace` when we ask for an `Iterator` of that type.

Was it easy to figure this out? It's not rocket science, but that's a long-winded way to express our intent. A human mind can juggle only so many concepts at once, and keeping track of all those lists and iterators is too much. Besides, the assertions themselves reek of primitive assertion.

4.4.2 *What to do about it?*

When the beef is hidden, you need to uncover it. The essence of a test should be front and center, and the way to accomplish this boils down to a few simple guidelines:

- 1 Extract the nonessential setup into private methods or the setup.
- 2 Give things appropriate, descriptive names.
- 3 Strive for a single level of abstraction in a method.

Let's see how these guidelines could be applied to our example.

First, let's move the creation of those four objects into the setup method, turning the local variables into fields on the test class. After all, they're just props for our test. Second, since our test makes a distinction between the types of those objects—whether they're “Fixnums” or “Strings”—we should name them such that the type is obvious. This listing illustrates one way to do that.

Listing 4.11 Naming objects used in a test makes the fixture easier to understand

```
public class TestObjectSpace {
    ...
    private IRubyObject string;
    private List<IRubyObject> fixnums;

    @Before
    public void setUp() throws Exception {
        ...
        string = runtime.newString("hello");
        fixnums = new ArrayList<IRubyObject>() {{
            add(runtime.newFixnum(10));
            add(runtime.newFixnum(20));
            add(runtime.newFixnum(30));
        }};
    }
    ...
}
```

Now we can turn our attention to the test method itself and the third guideline: striving for a single level of abstraction in a method. In terms of abstraction, what we care about in this test is that once we've added certain objects into the ObjectSpace, those exact objects are found through their respective iterators. The following listing shows the clear improvement in readability that going down to a single level of abstraction can provide.

Listing 4.12 Our test method now has a single level of abstraction

```
public class TestObjectSpace {
    private Ruby runtime;
    private ObjectSpace space;
    private IRubyObject string;
    private List<IRubyObject> fixnums;

    @Before
    public void setUp() throws Exception {
        runtime = Ruby.newInstance();
        space = new ObjectSpace();

        string = runtime.newString("hello");
        fixnums = new ArrayList<IRubyObject>() {{
            add(runtime.newFixnum(10));
```

test setupCreate
fixture objects in setup

```

        add(runtime.newFixnum(20));
        add(runtime.newFixnum(30));
    });
}

@Test
public void testObjectSpace() {
    addTo(space, string);
    addTo(space, fixnums);

    Iterator strings = space.iterator(runtime.getString());
    assertContainsExactly(strings, string);

    Iterator numerics = space.iterator(runtime.getNumeric());
    assertContainsExactly(numerics, fixnums);
}

private void addTo(ObjectSpace space, Object... values) { }
private void addTo(ObjectSpace space, List values) { }

private void assertContainsExactly(Iterator i, Object... values) { }
private void assertContainsExactly(Iterator i, List values) { }
}

```

↑
test setupCreate
fixture objects in setup

Populate
ObjectSpace

Check
contents of
ObjectSpace

That's already much better; the beef is visible now that all that setup and detail was moved outside of the test method itself. Alternatively, you could've tried giving the fixture objects descriptive names without moving them into a setup method, to see if that would've been a sufficient fix for the smell of incidental details. After all, only one test uses those fixture objects.

In any case, once you've recognized the incidental details, you still need to deal with the remaining smells of split personality (see section 4.5) and having a crappy test name. We'll leave these as an exercise for the reader.

4.4.3 Summary

When you're done writing a test, take a step back and ask yourself, "Is it perfectly clear and obvious what this test is all about?" If you hesitate to say yes, you're not done yet.

The beef is there, but it's crushed under the sheer volume of code in the test. Naming classes, methods, fields, and variables with descriptive names helps highlight the test's meaning, but it's hardly enough. To properly remedy such a situation, you should seek to extract nonessential bits of code and information into private helpers and setup methods. An especially useful idea is to try to keep a single level of abstraction in your test methods, as that tends to lead to more expressive names, a glaringly obvious flow of the test, and just enough sauce on the beef.

4.5 Split personality

One of the easiest ways to improve your tests is to look for a condition I call *split personality*. When a test exhibits split personality, it thinks it embodies multiple tests in itself. That's not right. A test should only check one thing and check it well.⁴

Time for an example again.

⁴ Note that this is different from the "one assertion per test" guideline that you may have bumped into—you might use multiple assertions to check that one thing!

4.5.1 Example

The test class in the next listing is for a piece of command-line interface. Essentially, it's testing the behavior of a Configuration object with different command-line arguments. Let's take a look.

Listing 4.13 Test for parsing command-line arguments

```
public class TestConfiguration {
    @Test
    public void testParsingCommandLineArguments() {
        String[] args = { "-f", "hello.txt", "-v", "--version" };
        Configuration c = new Configuration();
        c.processArguments(args);
        assertEquals("hello.txt", c.getFileName());
        assertFalse(c.isDebugEnabled());
        assertFalse(c.isWarningsEnabled());
        assertTrue(c.isVerbose());
        assertTrue(c.shouldShowVersion());

        c = new Configuration();
        try {
            c.processArguments(new String[] { "-f" });
            fail("Should've failed");
        } catch (InvalidArgumentException expected) {
            // this is okay and expected
        }
    }
}
```

The multiple personalities this test is supporting include something about a file name, something about debugging, enabling warnings, verbosity, displaying a version number, and dealing with an empty list of command-line arguments. Note that this test doesn't follow the arrange-act-assert structure you learned about in section 3.3.11.

It's obvious that we're asserting many things here. Though they're all related to parsing command-line arguments, they're also relevant in isolation from each other.

4.5.2 What to do about it?

The primary issue with this test is that it's trying to do too much, so we'll start by fixing that. There's also some duplication, but we'll deal with that, too. Let's begin by getting rid of that distraction so we can more clearly see the main issue.

The test is currently instantiating a Configuration with the default constructor in multiple places. We could make `c` a private field and instantiate it in a `@Before` method, like so:

```
public class TestConfiguration {
    private Configuration c;

    @Before
    public void instantiateDefaultConfiguration() {
        c = new Configuration();
    }

    ...
}
```

With the duplication out of our way we're left with two different calls to `processArguments()` and six different assertions (including the try-catch-fail pattern). That would suggest at least two different scenarios—two different tests. Let's extract those two segments into their own test methods:

```
public class TestConfiguration {
    ...

    @Test
    public void validArgumentsProvided() {
        String[] args = { "-f", "hello.txt", "-v", "--version" };
        c.processArguments(args);
        assertEquals("hello.txt", c.getFileName());
        assertFalse(c.isDebugEnabled());
        assertFalse(c.isWarningsEnabled());
        assertTrue(c.isVerbose());
        assertTrue(c.shouldShowVersion());
    }

    @Test(expected = InvalidArgumentException.class)
    public void missingArgument() {
        c.processArguments(new String[] { "-f" });
    }
}
```

With these coarse scenarios separated, it's also much easier to come up with a descriptive name for the test methods. That's always a good sign if you ask me.

We're still only halfway there. Looking at the assertions in the first test, you can see that some of the conditions being checked are explicit results of the provided command-line arguments, whereas some of them are implicit, default values. To make things better in that aspect, let's split the tests into multiple test classes, as described in figure 4.1.

This refactoring would mean that we have one test class focused on verifying the correct default values, one test class for verifying that explicitly setting those values from the command-line works as it should, and a third test class for specifying how erroneous configuration values should be handled. The change is reflected in the following listing.

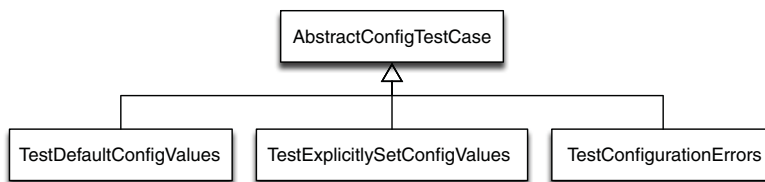


Figure 4.1 Three concrete, focused test classes inherit common setup from an abstract base class.

Listing 4.14 Split personalities extracted into separate test classes

```

public abstract class AbstractConfigTestCase {
    protected Configuration c;

    @Before
    public void instantiateDefaultConfiguration() {
        c = new Configuration();
        c.processArguments(args());
    }

    protected String[] args() {
        return new String[] { };
    }
}

public class TestDefaultConfigValues
    extends AbstractConfigTestCase {
    @Test
    public void defaultOptionsAreSetCorrectly() {
        assertFalse(c.isDebugEnabled());
        assertFalse(c.isWarningsEnabled());
        assertFalse(c.isVerbose());
        assertFalse(c.shouldShowVersion());
    }
}

public class TestExplicitlySetConfigValues
    extends AbstractConfigTestCase {
    @Override
    protected String[] args() {
        return new String[] { "-f", "hello.txt", "-v",
                               "-d", "-w", "--version" };
    }

    @Test
    public void explicitOptionsAreSetCorrectly() {
        assertEquals("hello.txt", c.getFileName());
        assertTrue(c.isDebugEnabled());
        assertTrue(c.isWarningsEnabled());
        assertTrue(c.isVerbose());
        assertTrue(c.shouldShowVersion());
    }
}

public class TestConfigurationErrors
    extends AbstractConfigTestCase {
    @Override
    protected String[] args() {
        return new String[] { "-f" };
    }

    @Test(expected = InvalidArgumentException.class)
    public void missingArgumentRaisesAnError() { }
}

```

Default to empty argument list

Override defaults for specific scenario

Override defaults for specific scenario

The benefit of this refactoring is in each individual test class being focused on just one thing. There's more code now; the boilerplate from having four classes instead of one does add volume. The relative increase in volume is much less when we split a test class with multiple test methods.⁵

Whether the trade-off of introducing an inheritance hierarchy is worth the added complexity is largely dependent on if the concrete test classes share any class-invariant tests through the base class. If you're only interested in splitting out the different test methods and their respective fixtures (and they share pretty much nothing), you could just as well create a distinct test class for each fixture and related tests, avoiding the added complexity of a class hierarchy and the accompanying scent of split logic.

4.5.3 Summary

Readable code makes its purpose clear to the programmer. Split personality is a test smell that fogs up your vision. Both details and the big picture of tests are hidden away as the code confuses you by intertwining multiple aspects into one test.

Splitting a test's multiple personalities—its multiple interests—into their own classes can help highlight the test's multiple meanings such that they're understandable, and when you come back to modify the code you'll be better off, as we can easily see what the code does and which scenarios are missing. In other words, you'll know what you don't know.

An additional benefit is that once you make a mistake (and you will), you have a more precise pointer to what's wrong, as the failing test covers a smaller corner of situations. It's no longer "something's wrong with parsing configuration parameters" but more akin to "configuration parameters that take multiple values don't work with a single value."

Chopping up test classes and methods is often a big improvement in terms of readability and maintainability. The chopping up can go overboard, though. That's when you've gone all the way from split personality to *split logic*.

4.6 Split logic

Nobody likes to read tests that span several screenfuls of code. Part of the reason is because such tests almost invariably exhibit more than a couple of test smells, and you're probably not certain what the test is doing and whether that's what it's supposed to do. Another reason to not like long methods that span several screenfuls is that it means you're forced to switch context, going back and forth between different parts of the source file to keep track of where you are and what's going on.

In short, the code you're looking for is scattered and requires extra effort, adding to your already significant cognitive load.⁶

⁵ Besides, at least I can read a long book in English much faster than a short book in Gaelic.

⁶ Cognitive overload is an especially common problem for programmers, due to programming being intrinsically taxing on the working memory.

The psychology of chunking

Speaking of long methods, the obvious approach to solving that problem is chunking it into smaller pieces. It turns out that *chunking* is an essential concept in cognitive psychology, too.

You may have heard someone point out how much easier it is to remember a phone number such as 326- 3918 than it is to remember an equally long and arbitrary sequence of digits, such as 3 2 6 3 9 1 8. The phone number is easier to remember because we've chunked it into two short sequences and we're much better at remembering short sequences—including short sequences of short sequences.

This could be interpreted to mean that you should split that long method into tiny chunks to improve the code's readability. But that would be jumping to conclusions too fast. For chunking to have a positive impact on our ability to manage information doesn't require the chunks to be meaningful, but being able to identify meaning for chunks significantly enhances the effect.⁷

In terms of code, you shouldn't blindly extract smaller methods, but rather do it mindfully, seeing that each individual chunk (method) consists of a meaningful sequence of chunks (statements) that in themselves share these same properties.

This information scatter is harmful for our cognitive load and makes it slower and more difficult to internalize the meaning and intent of a test. It's not just long methods that exhibit this scatter. An even worse example of information scatter in test code is prevalent in domains where tests involve manipulating larger chunks of data than the occasional float, int, or a String named firstName. Let's take a look at a real-life example of just such scatteritis-infected test code.

4.6.1 Example

The next listing also comes from the excellent JRuby project. This test checks that basic operations, such as variable assignments and method invocations, work as they should when *evaling* snippets of Ruby code using the Ruby class.

Listing 4.15 Tests that have their logic split into multiple places are difficult to grasp

```
public class TestRuby {
    private Ruby runtime;

    @Before
    public void setUp() throws Exception {
        runtime = Ruby.newInstance();
    }

    @Test
    public void testVarAndMet() throws Exception {
```

⁷ Dale Shaffer, Wendy Doube, Juhani Tuovinen. "Applying Cognitive Load Theory to Computer Science Education." Proceedings of First Joint Conference of EASE & PPIG, 2003. Psychology of Programming Interest Group, <http://www.ppig.org/papers/15th-shaffer.pdf>.

```

runtime.getLoadService().init(new ArrayList());
eval("load 'test/testVariableAndMethod.rb'");
assertEquals("Hello World", eval("puts($a)"));
assertEquals("dlroW olleH", eval("puts $b"));
assertEquals("Hello World",
    eval("puts $d.reverse, $c, $e.reverse"));
assertEquals("135 20 3",
    eval("puts $f, \" \", $g, \" \", $h"));
}
}

```

Now, this isn't a long method but it does exhibit severe scatter.⁸ Looking at this test method, it's impossible to tell *why* should `puts $b` return "Hello World" reversed? We're missing some crucial information from this source file because it's scattered to another place in the filesystem: a file named `testVariableAndMethod.rb`. Switching to that source file, we find a whole sequence of operations, shown in this listing.

Listing 4.16 `testVariableAndMethod.rb` contains a block of Ruby code

```

a = String.new("Hello World")
b = a.reverse
c = " "
d = "Hello".reverse
e = a[6, 5].reverse
f = 100 + 35
g = 2 * 10
h = 13 % 5
$a = a
$b = b
$c = c
$d = d
$e = e
$f = f
$g = g
$h = h

```

This explains why the test expects the magic variables from `$a` to `$h` to contain equally magical values. After checking both of these files and going back and forth once or twice, you can see how a total of 16 variables are assigned to inside `testVariableAndMethod.rb`. What makes the degree of scatter even worse than we thought is that the JUnit test only suggests the test involves variables and methods, but it turns out it's also testing mathematical operators—albeit not all of them.

Now, what could you do to reduce this severe information scatter?

4.6.2 What to do about it?

The simplest solution to split logic is often to inline the external information, moving all code and data into the test that uses it. Sometimes this is super convenient; sometimes it's less convenient. Let's see how this approach might look for the situa-

⁸ It also exhibits an acute split personality, but let's not dwell on that right now.

tion described in listings 4.15 and 4.16. My experiment with inlining testVariableAndMethod.rb is shown in this listing.

Listing 4.17 Inlining the split logic inside the test method

```
@Test
public void testVarAndMet() throws Exception {
    runtime.getLoadService().init(new ArrayList());

    AppendableFile script = withTempFile();
    script.line("a = String.new('Hello World')");
    script.line("b = a.reverse");
    script.line("c = ' '");
    script.line("d = 'Hello'.reverse");
    script.line("e = a[6, 5].reverse");
    script.line("f = 100 + 35");
    script.line("g = 2 * 10");
    script.line("h = 13 % 5");
    script.line("$a = a");
    script.line("$b = b");
    script.line("$c = c");
    script.line("$d = d");
    script.line("$e = e");
    script.line("$f = f");
    script.line("$g = g");
    script.line("$h = h");

    eval("load '" + script.getAbsolutePath() + "'");
    assertEquals("Hello World", eval("puts($a)"));
    assertEquals("dlroW olleH", eval("puts $b"));
    assertEquals("Hello World",
        eval("puts $d.reverse, $c, $e.reverse"));
    assertEquals("135 20 3",
        eval("puts $f, \" \", $g, \" \", $h"));
}
```

← Generate
data file from
test code.

This is interesting. Inlining the scattered information into the test method that uses it gets rid of the split logic problem, but now we're starting to have symptoms of other smells, like split personality and setup sermon. And that's all right. These smells are telling us what to do next!

Let's see how our test would look if we deal with the split personality and slice that monster into two tests: one for testing variable assignment and one for testing method invocation. While we're at it, let's trim those complicated sequences of variable assignments and one reverse after another. We don't need such a variety to verify that variable assignments and method calls work. All that extra noise only makes the test more difficult to follow. Listing 4.18 presents the outcome.

Listing 4.18 Slicing a big test into more specific ones

```
@Before
public void setUp() throws Exception {
    runtime.getLoadService().init(new ArrayList());
    script = withTempFile();
}
```

```

@Test
public void variableAssignment() throws Exception {
    script.line("a = String.new('Hello')");
    script.line("b = 'World'");
    script.line("$c = 1 + 2");
    afterEvaluating(script);
    assertEquals("Hello", eval("puts(a)"));
    assertEquals("World", eval("puts b"));
    assertEquals("3", eval("puts $c"));
}

@Test
public void methodInvocation() throws Exception {
    script.line("a = 'Hello'.reverse");
    script.line("b = 'Hello'.length()");
    script.line("c = ' abc '.trim(' ', '_')");
    afterEvaluating(script);
    assertEquals("olleH", eval("puts a"));
    assertEquals("3", eval("puts b"));
    assertEquals("_abc_", eval("puts c"));
}

private void afterEvaluating(AppendableFile sourceFile)
    throws Exception {
    eval("load '" + sourceFile.getAbsolutePath() + "'");
}

```

With this change our code doesn't look too bad. We could probably still find ways to trim those two tests in listing 4.18 even smaller, making them even more focused. For example, our checks for the expected results reek of primitive assertion. We'll leave that for the future.

When to inline data or logic?

Some data and some logic are good to inline, some others are best kept separate. Here's a simple heuristic for quickly determining what you should pull into the test and what to push out to the sidelines:

- 1 If it's short, inline it.
- 2 If it's too long to keep inlined, stash it behind a factory method or a test data builder.
- 3 If that feels inconvenient, pull it out into a separate file.

It's that simple. The best thing to do would be to colocate all knowledge used in a test inside the test. That often leads to too big a test, in which case we typically trade the close proximity of everything being inside the test method in exchange for the test method invoking helper methods on the test class itself. In some cases the nature of the data or logic involved is so complex that it's inconvenient to keep it inside the test class. In those cases it may make more sense to extract that logic or data into a separate data or source file—especially if the same data or logic is duplicated across multiple test classes.

I don't go for an external data file too lightly. The effort of seeing the content of two files simultaneously ranges from slightly inconvenient to next to impossible, depending on which editors and IDEs are hanging from your tool belt. IDEs don't handle non-programming language files too well, which means that you don't get much support for navigating between your test code and such data files. With that said, when I do end up with a separate data file, there are a few guidelines I try to follow:

- Trim your data to bare essentials—just enough to represent the scenario in question. Even though you may decide to accept information scatter as the lesser of two evils, there's no excuse for doing anything but your best to keep that accidental complexity at a minimum.
- Place such data files in the same folder as the tests that use them. That way the data files are easier to find and move around, along with the test code. (Plus, you can load them through Java's classpath rather than by reading a file at a specified file path.) More on this approach in section 5.4.11.
- Whatever structure you end up with, make it a convention with your team and stick to it. You'll make everyone's life that much easier.

Consider looking up such external resources from the classpath rather than with a file path; you won't have to edit your tests when moving the test code and the data file from one package to another. If you decide to go this way, it might be a good idea to make sure that your test data files have unique names, just to avoid a nasty surprise.

4.6.3 Summary

Not too long ago, programmers were taught that source code should be wrapped so that the line length never exceeds 80 characters. Similarly, we were taught that methods shouldn't be longer than 25 lines. Though the exact numbers may have been slightly different, the idea behind these rules has been the simple fact that screen estate is limited, and for a long time people's monitors would only fit around 80 characters on a line and some 25 lines per screen. Anything more than that and you'd have to scroll to see the rest.

Though today's monitors can accommodate many more pixels and our preference for the maximum size of a method may have changed,⁹ the importance of not having to navigate between multiple parts of the code base still applies. The split logic smell warns us from unnecessarily separating the logic or data of a test into multiple places, especially multiple source files.

Though it makes sense in some cases to divide logic or data into separate files, the rule of thumb would be to first seek a way to inline that logic or data inside the test that uses it, and if that doesn't work, look for a way to keep it inside the test class. Accepting split logic should be a last resort rather than a default practice.

⁹ Personally, I prefer methods to be shorter than 10 lines of code, and the ones I really like are almost without exception less than five lines of code.

Whereas split logic may not be among the most common test smells, the next smell is definitely one of the most widespread, even though it's something most programmers are taught to avoid early in their careers.

4.7 Magic numbers

There are a few things that unite programmers all over the world. Programming languages and design patterns are good examples of such things. We're also almost universally of the opinion that *magic numbers* are bad and should be avoided. Magic numbers are numeric values embedded into assignments, method calls, and other program statements.

Literals `123.45` and `42` would be prime examples of such magic numbers. The argument goes that magic numbers are bad because they don't reveal their meaning—reading the code, you're left wondering what's the purpose of that `42`? Why is it `42` instead of, say, `43`? With this in mind, the fundamental problem of magic numbers can be found in magic strings just as well—why is that literal string empty? Does it matter or did someone just need a `String`—*any* `String`?

The conventional advice for treating magic numbers is to replace them with constants or variables that give the number that much-desired meaning, making code easier to read. But that isn't the only option we have. Let's take a look at an example ridden with magic numbers and what might be a surprising alternative for fixing the problem.

4.7.1 Example

Philip Schwarz, in his comment to Jeff Langr's blog post,¹⁰ introduced the following test he'd written for an object representing a game of bowling, shown in the following listing:

Listing 4.19 Why does a `roll(10, 12)` result in a score of 300?

```
public class BowlingGameTest {
    @Test
    public void perfectGame() throws Exception {
        roll(10, 12);
        assertEquals("score", 300, game.score());
    }
}
```

What makes this test worth examining is the way it passes magic numbers to the code under test without explaining what they are. It might be obvious to you while you're writing that test, but it may not be at all obvious what the meaning of `10` and `12` is, and why that input should result in `300`. It might not be obvious to you either when you come back to this code after a couple of weeks.

¹⁰ See Jeff Langr's blog "Goofy TDD Construct," March 25, 2010, <http://langrsoft.com/jeff/2010/03/goofy-tdd-construct/>.

4.7.2 What to do about it?

To improve this test, Philip proceeded to call out those magic numbers, making their meaning and relationship explicit by giving each number a name. As I said, the way Philip did this might be a surprise. This shows how he did it:

Listing 4.20 A-ha! Knocking all 10 pins 12 times in a row yields a score of 300

```
public class BowlingGameTest {
    @Test
    public void perfectGame() throws Exception {
        roll(pins(10), times(12));
        assertEquals("game.score()", is(equalTo(300)));
    }

    private int pins(int n) { return n; }
    private int times(int n) { return n; }
}
```

Though it's more common to see programmers call out their magic numbers using static constants or local variables, the approach Philip's test demonstrates in listing 4.20 has a certain advantage, one that may or may not be relevant for a given situation with a magic number you'd like to call out. For example, if we would've used a constant for giving a name to our magic numbers in listing 4.19, we'd be looking at something like this:

```
roll(TEN_PINS, TWELVE_TIMES);
```

This would be explicit and readable—even easier for a human to read than the `pins(10)` and `times(12)` in listing 4.20. The advantage of the method-based approach comes into play when you have several tests—as is most often the case—where the number of pins and rolls is different. Surely you don't want to have a separate constant for every one of those tests.¹¹

4.7.3 Summary

Magic numbers are numeric values among code that don't communicate their meaning. Without an explicit name or explanation for such a magic number, you're left helpless in trying to figure out why that number is there and why its value is exactly what it is? In short, you should avoid magic numbers because they're about as easy to grok as David Copperfield's magic.

The most common way of calling out magic numbers and making them explicit is to replace the magic number with a local variable or a constant with an expressive name. Listing 4.20 demonstrated a somewhat different but extremely useful technique for documenting the meaning of the magic number.

¹¹ Other programming languages (such as Ruby) support parameter assignment by name, which can be used to achieve similar advantages.

As programmers we've often been taught to think that code should be fast and that compact is better than verbose, and often that would be a fair statement. But there are occasions when the need for readability trumps our desire for conciseness. More often than not, it's worth the extra line or two to make our intent explicit. That's exactly what Philip's `pins()` and `times()` in listing 4.20 are about.

Sometimes, we seem to forget about the virtues of conciseness altogether and end up writing real novels in our tests. Sometimes, we bump into such literary works already in the tests' setup!

4.8 Setup sermon

In section 4.4 you learned about incidental details and how tests should reveal their intent, purpose, and meaning promptly and squarely. Most often we tackle such a problem by extracting the irrelevant details behind helper methods, leaving only the essential bits—the beef—in the test method to make it concise and to the point. Sometimes we move a bunch of code that prepares the scenario for a test into a *setup method*.

Just as we'd be foolish to pretend that test code doesn't need the same degree of professional treatment as production code, we'd be mistaken to think that messy code would be any less dire a problem when it's found in a setup method.

Let's look at an example that illustrates this problem in concrete terms.

4.8.1 Example

The following example is code I wrote some years ago. It's a test for an object responsible for downloading “presentations,” which are listed in an XML-based manifest and missing from the local storage. Look at the following listing and see if it connects with the name of this test smell.

Listing 4.21 Setup sermon is a long setup for a simple test

```
public class PackageFetcherTest {
    private PackageFetcher fetcher;
    private Map downloads;
    private File tempDir;

    @Before
    public void setUp() throws Exception {
        String systemTempDir = System.getProperty("java.io.tmpdir");
        tempDir = new File(systemTempDir, "downloads");
        tempDir.mkdirs();
        String filename = "/manifest.xml";
        InputStream xml = getClass().getResourceAsStream(filename);
        Document manifest = XOM.parse(IO.streamAsString(xml));
        PresentationList presentations = new PresentationList();
        presentations.parse(manifest.getRootElement());
        PresentationStorage db = new PresentationStorage();
        List list = presentations.getResourcesMissingFrom(null, db);
        fetcher = new PackageFetcher();
        downloads = fetcher.extractDownloads(list);
    }
}
```

```

@After
public void tearDown() throws Exception {
    IO.delete(tempDir);
}

@Test
public void downloadsAllResources() {
    fetcher.download(downloads, tempDir, new MockConnector());
    assertEquals(4, tempDir.list().length);
}
}

```

Wow. A lot is going on there, including a case of split logic and a design far from ideal.¹² Leaving that aside and focusing on the symptoms of having such a lengthy and complicated setup, the problem is that the complicated setup is an integral part of the test—which is fairly simple in itself—and effectively makes the test equally complicated.

We sometimes talk about the concept of a fixture, which is the context in which a given test will execute, including the system properties, the constants defined in the test class, and the private members initialized by the `@BeforeClass` and `@Before` setup methods in the test class's hierarchy. In order to understand a test, you need to understand its fixture; and the setup is often the most essential part of that fixture, creating the specific objects and the state the test runs against.

4.8.2 What to do about it?

Though I've brought up the setup sermon as a first-class test smell among the rest, it's really a special case of incidental details. Not surprisingly, we lean on some of the same techniques when encountering a setup sermon. More specifically, we:

- 1 Extract the nonessential details from the setup into private methods
- 2 Give things appropriate, descriptive names
- 3 Strive for a single level of abstraction in the setup

This listing illustrates how these steps might turn out.

Listing 4.22 Extracting details makes the setup easier to grasp

```

public class PackageFetcherTest {
    private PackageFetcher fetcher;
    private Map downloads;
    private File tempDir;

    @Before
    public void setUp() throws Exception {
        fetcher = new PackageFetcher();
        tempDir = createTempDir("downloads");
        downloads = extractMissingDownloadsFrom("/manifest.xml");
    }
}

```

¹² The `PackageFetcher` violates the SRP by assuming responsibility for two things: extracting a list of required downloads and downloading them.

```

@After
public void tearDown() throws Exception {
    IO.delete(tempDir);
}

@Test
public void downloadsAllResources() {
    fetcher.download(downloads, tempDir, new MockConnector());
    assertEquals(4, tempDir.list().length);
}

private File createTempDir(String name) {
    String systemTempDir = System.getProperty("java.io.tmpdir");
    File dir = new File(systemTempDir, name);
    dir.mkdirs();
    return dir;
}

private Map extractMissingDownloadsFrom(String path) {
    PresentationStorage db = new PresentationStorage();
    PresentationList list = createPresentationListFrom(path);
    List downloads = list.getResourcesMissingFrom(null, db);
    return fetcher.extractDownloads(downloads);
}

private PresentationList createPresentationListFrom(String path)
    throws Exception {
    PresentationList list = new PresentationList();
    list.parse(readManifestFrom(path).getRootElement());
    return list;
}

private Document readManifestFrom(String path) throws Exception {
    InputStream xml = getClass().getResourceAsStream(path);
    return XOM.parse(IO.streamAsString(xml));
}
}

```

As you can see, the setup only does three things: instantiates the `PackageFetcher`, creates a temporary directory, and extracts a `Map` of missing downloads from a file named `manifest.xml`. Overall, we now have more code in terms of absolute line count, but the clarity we've gained by factoring out logical steps and giving them expressive names is worth it. After this refactoring, our setup method is at a single level of abstraction and easily understood. If we want to sift through the details, we can do so by navigating down to the private helper methods that carry out the grunt work.¹³

4.8.3 Summary

Setup methods, whether they're annotated with `@Before` or `@BeforeClass`, should be loved and nurtured just as much as the rest of our test code. The setup forms a major part of a test's fixture and creates the context in which a test runs. Without understand-

¹³ Sometimes these helper methods prove to be useful outside of test code, too, and find their way into becoming part of the production code. It happens.

ing the fixture, you don't really understand what the test is about. Therefore, you should attack a setup sermon just like you charge at a case of incidental details in a test method: by extracting out details, giving those steps expressive names, and striving to keep the setup at a single level of abstraction so that it's quick to read and easy to grasp.

It should be noted that sometimes a setup points to a much bigger problem. Sometimes when we find a complicated setup, the problem we should be solving isn't the readability of the test's setup, but rather the design problem that forces the test to do so much work as a way of putting an object under test. Far too often the reason we write so much code is because we've made a less than stellar design decision earlier.

And, sometimes, we simply write too much code, like in the case of our last test smell in this chapter.

4.9 Overprotective tests

One of the most common symptoms of a bug while running Java code is the sudden `NullPointerException` or `IndexOutOfBoundsException`, caused by an argument being a null pointer or an empty string when the method in question expected something else. Software programmers have become so accustomed to bumping into such error messages that they've started to protect themselves against them by adding guard clauses and other kinds of null-checks at the beginning of their methods.¹⁴

Every now and then programmers apply this defensive programming strategy in their test code, too, protecting the test from failing with an ugly `NullPointerException` and, instead, letting the test fail gracefully with a nicely worded assertion error. Let's hurry up to an example that illustrates this failure mode.

4.9.1 Example

The next listing shows a simple test that verifies a correct calculation with two assertions: one for verifying that the `Data` object returned from the `getData()` method isn't null, and another for verifying that the actual count is correct.

Listing 4.23 An overprotective test obsesses over failing even when it would anyway

```
@Test
public void count() {
    Data data = project.getData();
    assertNotNull(data);
    assertEquals(4, data.count());
}
```

This is an *overprotective test* because the `assertNotNull` is superfluous. The test is being overprotective by checking that `data` isn't null (failing the test if it is) before invoking a method on it. That's being overprotective because the test would fail anyway when `data` is null; the test would fail miserably with a `NullPointerException` when the second assertion attempts to invoke `count()` on `data`.

¹⁴ The smarter approach would be to first decide whether the argument not being null should be part of the method's contract—and fixing the bug at its root rather than trying to patch up the symptoms.

4.9.2 What to do about it?

Delete the redundant assertion. It's basically useless clutter that provides no added value.

When asking the programmer who wrote those two assertions about why they felt it necessary to have the null-check there, almost invariably they quote ease of debugging as the primary motivator for having written it. The thinking goes that when one day the `Data` object returned is indeed null, it'll be easier to discover by examining the failed JUnit test's stack trace from within the IDE. You'll simply double-click on the assertion and the IDE will take you directly inside the test method and highlight the line where `assertNotNull` failed.

This thinking is flawed, because even without the `assertNotNull`, you can click on the stack trace of the resulting `NullPointerException`, which will take you to the line where `assertEquals` trips on that null reference. The only real advantage of having checked for null separately would be a situation where the `NullPointerException` is thrown from a line that has a method chain like this:

```
assertEquals(4, data.getSummary().getTotal());
```

In this line of code, you can't immediately tell whether the `NullPointerException` was about `data` being null or the `summary` object being null. Without that explicit check in your test, you'd have to either launch a debugger and step through the code to see which of the objects was null, or you'd have to temporarily add that missing `assertNotNull` and rerun the test.

Personally, I'd rather bear this minor inconvenience once when encountering such a test failure than endure the clutter every time I read my test code.

4.9.3 Summary

Overprotective tests are flush with redundant assertions that check intermediate conditions that are required for the test's real assertion to pass. Such redundant assertions provide no added value and should be avoided. The test will fail anyway; the difference is in the kind of error message you get from that failing test. Though being explicit is generally a good thing for a programmer, being explicit about irrelevant details only confuses the fellow programmer working with your code—that's why they're called incidental details.

4.10 Summary

In this chapter we've studied a number of test smells that hinder test code's readability.

We started by looking at smells that stem from badly written and confusing assertions. Primitive assertions check for desired functionality through overly low-level means, comparing primitive objects miles away from the level of abstraction of the code under test. Hyperassertions demand too much—so much that the test will break if even the smallest detail changes. Hyperassertions extend themselves so widely that the specific logic the test seeks to verify is lost in the sheer volume of the details.

Bitwise assertions are almost the opposite of hyperassertions in that they're often extremely concise. The problem with assertions that lean on bit operators to express a desired condition is that not all programmers live and breathe bit operations. They're too difficult. We're not talking about matters that require a Ph.D, but the added cognitive load is enough to stop a programmer's flow.

We scrutinized a number of smells that stand in the way of the programmer understanding the test he's looking at. Incidental details are where the essence of a test is hidden behind confusing and irrelevant details. Split personality confuses us by intertwining multiple tests into one, checking all sorts of aspects rather than having a clear focus. Split logic derails our attempts at grasping a test by scattering its logic around in multiple files.

Magic numbers are seemingly arbitrary numeric values littered around our test code. These numbers may have significant meaning, but that meaning is lost without the test making it explicit, calling it out by a name. Sometimes, we're too explicit. Setup sermon is a particularly long-worded setup method that spells out too much in too much detail. Overprotective tests go overboard by explicitly asserting technical preconditions required for the ultimate assertion to pass—instead of stating what needs to be true.

All of these smells are common failure modes in writing good tests. The consequences of these failures are different kinds of obstacles for the tests' readability, making it harder for the programmer to understand what's going on and the point of a given test. Code is read much more frequently than it's written and understanding code is essential in maintaining it.

In the next chapter we'll dig into even more test smells that specifically concern the issue of maintainability.

5

Maintainability

In this chapter

- Test smells that add to your cognitive load
- Test smells that make for a maintenance nightmare
- Test smells that cause erratic failures

In this second chapter of part 2 we turn our attention from the readability of test code toward its maintainability.

Many of us have heard the quip that code is read much more often than it's written. That was the case with punch cards and it's still the case today. That is why readability is so crucially important. When we go beyond merely looking at code, we step into the world of writing code—and most often “writing” really means modifying or extending an existing code base. Sometimes we call that *maintenance* and sometimes we call it *development*. In any case, something probably needs to be changed about the code.

What makes code such a fascinating medium to work with and programming such a difficult skill to master is that the nature of code is to be both extremely flexible and extremely fragile. Code is flexible because there are few things that can't be done with software—it's without physical constraints. It's fragile because even the smallest of changes can break it down completely. Code doesn't degrade slowly. It crashes.

Tests are like that. Because they're made up of code just like production code is, tests have the same kind of fundamental brittleness. That's why it's of utmost importance that programmers pay attention to managing this brittleness as they write automated unit tests. We all know what a *maintenance nightmare* is. You don't want your test code to become one.

Just like chapter 4, this chapter is constructed as a sequence of test smells illustrated with examples and followed by a discussion of potential solutions or improvements. We'll begin with some of the most common test smells that increase the cost of keeping your test code up to date, and proceed toward more specific but equally dangerous hazards to your test code's maintainability. First up, let's talk about duplication.

5.1 Duplication

Go to a large industry conference, grab 10 programmers at random, and ask what's the root of all evil? I will bet most of them will respond with a reference to Donald Knuth's classic remark about premature optimization being the root of all evil.¹ The second most popular answer will likely be that *duplication* is the root of all evil—assuming you're attending a conference frequented by agile software developers, test-infected programmers, and software craftsmen who deeply care about clean code.

So what's duplication? What do we mean by duplication? In short, duplication is the existence of multiple copies or representations of a single concept—*needless repetition*.

The most obvious duplication is perhaps the kind where a certain numeric value or a literal string is repeated throughout the code base. Sometimes duplication exists not in the hardcoded data and values sprinkled around the code base but in the logic of short code snippets or in overlapping responsibilities between two classes, objects, or methods.

Duplication is bad because it makes the code base more opaque and difficult to understand as concepts and logic are scattered among multiple places. Furthermore, every bit of duplication is an additional place where a programmer needs to touch the code base to make a change. Each of those bits of duplication also increases the chance of a bug slipping in as you forget to make a change in all necessary places.

5.1.1 Example

Enough talk, I say! Let's look at a concrete example in the next listing that illustrates some of these forms of duplication.

Listing 5.1 A short test class with multiple kinds of duplication.

```
public class TemplateTest {
    @Test
    public void emptyTemplate() throws Exception {
        assertEquals("", new Template("").evaluate());
    }
}
```

¹ Donald Knuth. "Structured Programming with go to Statements." Computing Surveys, Vol. 6, No. 4, 1974. Stanford University.


```

@Test
public void plainTextTemplate() throws Exception {
    assertEquals("plaintext", new Template("plaintext").evaluate());
}
}

```

What kind of duplication do you see in that listing? Perhaps the most trivial examples of duplication in this example are the literal strings in both assertions—the empty string and `plaintext` are both spelled out twice, which I call *literal duplication*. You could remove that literal duplication by defining a local variable. There’s another kind of duplication in this test class which is perhaps more interesting than the obvious case of literal strings. It becomes clear once we extract those local variables, so let’s start cleaning this mess up.

5.1.2 What to do about it?

We already identified the duplication between literal strings, so let’s start by getting rid of it. Our strings may not be the worst case of duplication in the history of computer programming, but cleaning up those small code smells often makes it easier to see the big ones. The next listing shows what we have now.

Listing 5.2 Extracting a local variable highlights the remaining duplication

```

public class TemplateTest {
    @Test
    public void emptyTemplate() throws Exception {
        String template = "";
        assertEquals(template, new Template(template).evaluate());
    }

    @Test
    public void plainTextTemplate() throws Exception {
        String template = "plaintext";
        assertEquals(template, new Template(template).evaluate());
    }
}

```

When you look at the two tests and what they actually *do*, you’ll realize that the literal strings were the only difference between the two test methods. After extracting those differing parts into local variables, the remaining assertions aren’t just similar—they’re identical. They’re both instantiating a `Template` object, evaluating the template, and asserting that the template was rendered as is. This situation where the logic is duplicated but operates on different data is called *structural duplication*. The two blocks of code operate on different values but have identical structure.

Let’s get rid of this duplication and see how the code looks like afterward. The next listing shows the test class after we’ve extracted the duplication we now see into a custom assertion method.

Listing 5.3 The duplication in assertions has been extracted into a custom assertion

```

public class TemplateTest {
    @Test
    public void emptyTemplate() throws Exception {
        assertTemplateRendersAsItself("");
    }

    @Test
    public void plainTextTemplate() throws Exception {
        assertTemplateRendersAsItself("plaintext");
    }

    private void assertTemplateRendersAsItself(String template) {
        assertEquals(template, new Template(template).evaluate());
    }
}

```

Semantic duplication

Sometimes duplication hides from us in plain sight. Though literal and structural duplication are things we can clearly see by identifying similar-looking structures, *semantic duplication* represents the same functionality or concept with different implementations.

Here's an example of such semantic duplication:

```

@Test
public void groupShouldContainTwoSupervisors() {
    List<Employee> all = group.list();
    List<Employee> employees = new ArrayList<Employee>(all);
    Iterator<Employee> i = employees.iterator();
    while (i.hasNext()) {
        Employee employee = i.next();
        if (!employee.isSupervisor()) {
            i.remove();
        }
    }
    assertEquals(2, employees.size());
}

@Test
public void groupShouldContainFiveNewcomers() {
    List<Employee> newcomers = new ArrayList<Employee>();
    for (Employee employee : group.list()) {
        DateTime oneYearAgo = DateTime.now().minusYears(1);
        if (employee.startingDate().isAfter(oneYearAgo)) {
            newcomers.add(employee);
        }
    }
    assertEquals(5, newcomers.size());
}

```

These two methods don't have much literal duplication, apart from both querying `group.list()` for the full list of employees. Yet they're very similar in that they both take the full list of employees and filter that list with some criteria.

(continued)

We tend to approach removing this kind of semantic duplication by first making it structural duplication—changing the way the filtering is done—and then extracting variables and methods to remove the structural duplication. We'll leave doing that as an exercise for the reader.

Looking at listing 5.3, you can see that all of the duplication is now gone. Even the local variables are no longer needed. It's much easier to understand at a glance and, for example, if you wanted to change the template rendering API, you'd only need to make that change in one place.

5.1.3 Summary

Duplication is one of the most common code smells. It's also common in test code, which shouldn't be a surprise considering that the nature of test classes is to specify a variety of aspects of behavior for a small unit of code. Scattering test code with duplication means spending a lot more hours maintaining tests—not to mention hunting down the odd bug resulting from not remembering to update all instances of a duplicated piece of information.

Fortunately, the test-infected programmer has a couple of simple remedies for this kind of duplication. Extracting variable data into local variables, for example, isn't just a way to remove obvious data duplication between literal values it's also an excellent technique for highlighting structural duplication.

Removing duplication can be taken too far, too. After all, your primary interest is to keep the code readable and its intent and function obvious to the reader. With that in mind, there might be a situation every now and then—especially in test code—where you might choose to leave some duplication in place in favor of improved readability.

Our next test smell isn't as common as duplication. The good news is that it's easy to spot and avoid once you've gotten to know it.

5.2 Conditional logic

We write automated tests for multiple reasons. We look for safety from regression, expecting our tests to raise the flag when we stumble. We lean on tests to help us formulate the behavior that we want from our code. We also lean on tests to understand what the code does or what it should do. Particularly related to this last point, the presence of *conditional logic* in our tests is generally a bad thing. It's a smell.

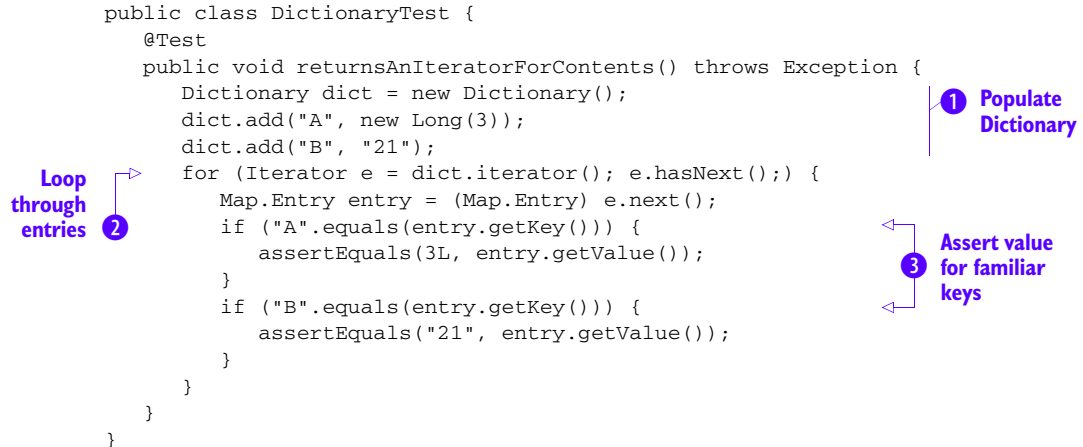
Let's say we're in the middle of a refactoring and running the tests to make sure everything still works, when we find that one of the tests is now failing. This comes as a surprise, we didn't expect our change to affect that test, but that's what happened. We check the stack trace from the test failure and open the test in our trusted editor and try to figure out what's going on, only to realize we can't even figure out what the test is actually doing when it fails.

The struggle we face in this kind of a scenario is the reason we should avoid conditional logic in our test code. Let's illustrate the problem with a more concrete example.

5.2.1 Example

The test shown in the following listing creates a `Dictionary` object, populates it with data, and verifies that the right stuff comes out when asked for an `Iterator` into its content, which is a sequence of `Map.Entry` objects.

Listing 5.4 Conditional logic in test code is a burden to maintain



```

public class DictionaryTest {
    @Test
    public void returnsAnIteratorForContents() throws Exception {
        Dictionary dict = new Dictionary();
        dict.add("A", new Long(3));
        dict.add("B", "21");
        for (Iterator e = dict.iterator(); e.hasNext();) {
            Map.Entry entry = (Map.Entry) e.next();
            if ("A".equals(entry.getKey())) {
                assertEquals(3L, entry.getValue());
            }
            if ("B".equals(entry.getKey())) {
                assertEquals("21", entry.getValue());
            }
        }
    }
}

```

1 Populate Dictionary

2 Loop through entries

3 Assert value for familiar keys

As you can see, this test is surprisingly difficult to parse and understand, even though it's testing a fairly trivial behavior of the `Dictionary` ❶ object. Furthermore, when we do get through the `for` loop and `if` blocks, it turns out that the test populates two entries into the `Dictionary`, loops ❷ through the returned `Iterator`, and checks the entry's value ❸ depending on its key.

What this means in practice is that the test expects both of the two populated entries being found through the `Iterator`. Our goal is to make that intent obvious and reduce the chances of a mistake when editing the test, say, during a refactoring that changes the `Dictionary` class's API. With all of the conditional execution going on in this test code, it's easy to end up thinking that a given assertion is passing while it's actually not even run. That's something we want to avoid.

5.2.2 What to do about it?

As is so often the case when dealing with overly complex code, the first step is to simplify it, usually by extracting a block of code into a separate, well-named method. In listing 5.4 the complexity lies completely within that `for` loop, so let's see how we could extract that.

Listing 5.5 Extracting a custom assertion cleans up the test

```

@Test
public void returnsAnIteratorForContents() throws Exception {
    Dictionary dict = new Dictionary();
    dict.add("A", new Long(3));
    dict.add("B", "21");
    assertContains(dict.iterator(), "A", 3L);
    assertContains(dict.iterator(), "B", "21");
}

private void assertContains(Iterator i, Object key, Object value) {
    while (i.hasNext()) {
        Map.Entry entry = (Map.Entry) i.next();
        if (key.equals(entry.getKey())) {
            assertEquals(value, entry.getValue());
            return;
        }
    }
    fail("Iterator didn't contain " + key + " => " + value);
}

```

① Simple custom assertion

② Found what we we're looking for

③ Fail the test

The listing has replaced the original for loop with a custom assertion ①, making the test itself fairly simple. The loop still exists inside the extracted method but the intent of the test is now clear, which makes a world of difference as far as maintainability goes.

There was another issue with the original test that we've corrected in listing 5.5. More specifically, the original test in listing 5.4 wouldn't have failed even if the Iterator had been empty because of the way the for loop was used. We've corrected that in listing 5.5 by returning from the assertion when we find the expected entry ② and failing the test ③ if a matching entry wasn't found.

This is a common pattern that's good to keep in mind. Incidentally, that final call to fail() is often forgotten, which can cause some head-scratching when tests are passing and yet the code doesn't work. Pay attention!

5.2.3 Summary

We lean on our tests to understand what the code does and what it should do. When changing our code, we rely on our tests to warn us if we're accidentally breaking stuff. Conditional logic makes all of this more difficult and more error-prone.

Conditional execution makes code harder to understand, and you need to understand code before you can change it. If you find yourself debugging tests filled with conditional statements it's hard to figure out what exactly is being executed and how it fails. Furthermore, sometimes you'll find that the conditional execution cuts off your assertions, skipping them altogether, creating a false illusion of correctness.

In short, you should avoid conditional execution structures such as if, else, for, while, and switch in your test methods. This is especially important when the behavior the tests are checking is more complex. There are places for these language

constructs, but when it comes to test code, you should tread that path carefully.² To keep your tests maintainable, you need to keep their complexity well below that of the solution being described.

Conditional logic in test code can be tricky to sort out but fairly easy to spot. Next up we have a test smell that's a tougher nut to crack.

5.3 Flaky test

In many ways automated tests are a programmer's best friend. Like our friends in real life, we should be able to trust our friends in the codebase—the tests that immediately alert us to the occasional mistakes we make. Sometimes, a friend might turn out to be not so trustworthy after all.³ One of the worst test smells I can think of is the simple case of a test that's constantly failing—and is considered normal.

There's a difference between a test failing when you change something—that's a good thing, as the test obviously noticed a change in something you've considered important enough to check—and a test that fails all the time. We're talking about a test that's been in the red for two weeks now and nobody's fixed it. That test is probably going to stay broken for another two weeks or two months until somebody either fixes it or, more likely, deletes it. And that generally makes sense, because a constantly failing test is next to useless. It doesn't tell you anything that you wouldn't know already or that you could trust.

There's another way in which failing tests make our lives unnecessarily complicated: *flaky tests* that fail intermittently. By that I mean flaky in the sense of the *boy who cried wolf*, tests that fail the build but, once you take a look, they're running again, passing with flying colors, but by accident.

Let's look at an example to clarify what the heck I'm trying to convey with all this Aesopian fable reference.

5.3.1 Example

Most of the time when I encounter a flaky, failing test it turns out to involve threads and race conditions, behavior that's dependent on the current date or time, or tests that depend on the computer's performance and, therefore, are influenced by things such as I/O speed or the CPU load during the test run. Tests that access a network resource have acted flaky on occasion, too, when the network or the resource has been temporarily unavailable.

Our example in the following listing showcases a problem I've stumbled on myself more than once: a flawed assumption about filesystem timestamps.

² Loops and other conditional statements can be essential tools for building test helpers. In test methods, though, these structures tend to be a major distraction.

³ For the record, obviously none of *my* friends are like that. (Just in case you're reading this.)

Listing 5.6 Test for aggregated logs being sorted by timestamp

```

@Test
public void logsAreOrderedByTimestamp() throws Exception {
    generateLogFile(logDir, "app-2.log", "one");
    generateLogFile(logDir, "app-1.log", "two");
    generateLogFile(logDir, "app-0.log", "three");
    generateLogFile(logDir, "app.log", "four");
    Log aggregate = AggregateLogs.collectAll(logDir);
    assertEquals(asList("one", "two", "three", "four"),
        aggregate.asList());
}

```

Generate some log files

Expect certain order

The problem with the test is that the logic we're testing relies on files' timestamps. We expect the log files to be aggregated in timestamp-based order, treating the file with the oldest last-modified date as the oldest log file and so forth, not sorting the log files by name in alphabetical order. It turns out that this expectation isn't always fulfilled. This test fails every now and then on Linux, and fails practically all the time on Mac OS X. Let's look at the `generateLogFile()` method our test uses for writing those log files:

```

private void generateLogFile(final File dir, final String name,
    final String... messages) {
    File file = new File(dir, name);
    for (String message : messages) {
        IO.appendToFile(file, message);
    }
}

```

Our downfall with this test and this method of creating the log files for our test is that we've forgotten the passing of time. More specifically, we've forgotten that time passes differently on different computers and on different platforms. On some platforms like Mac OS X, for example, the filesystem timestamps move in one-second increments, which means that it's extremely likely that consequent calls to the `generateLogFile()` will result in identical timestamps on the generated log files, unless it happens to take just long enough to generate them or they're generated at just the right time.

That's a lot of conditionals, so let's talk about what you could do about this problem.

5.3.2 What to do about it?

When it comes to time-related issues, and especially when we want time to pass in our tests, it's surprisingly common for programmers to add a call to `Thread#sleep`. Don't do that. The problem with `Thread#sleep` as a solution is that it's almost as nondeterministic as the problem was—there's still no guarantee that enough time has passed. You can play safe and exaggerate by sleeping for a whole second instead of the 100 milliseconds that's *usually* enough. That bandage comes with a high cost, as your test suite gets increasingly slow with every new `Thread#sleep`.

Instead, when you face a time-related issue in your tests, you should first look at simulating the appropriate conditions through proper APIs. In this case of generating the log files with correct timestamps, you can lean on Java's `File` API and explicitly

increment each log file's timestamp with `setLastModified`. The following listing shows this modification to our utility method.

Listing 5.7 Setting file timestamps clearly apart

```
private AtomicLong timestamp =
    new AtomicLong(currentTimeMillis());

private void generateLogFile(final File dir, final String name,
    final String... messages) {
    File file = new File(dir, name);
    for (String message : messages) {
        IO.appendToFile(file, message);
    }
    file.setLastModified(timestamp.addAndGet(TEN_SECONDS));
}
```

1 Keep track of timestamps

2 Set timestamps clearly apart

With the simple addition of an `AtomicLong` object ❶ and incrementing it explicitly with a call to `File#setLastModified` ❷ for each log file, we're effectively ensuring that each of those log files receives a different timestamp. Doing this instead of relying on implicit and nondeterministic behavior makes our test in listing 5.6 work reliably on all platforms and all computers without slowing down our test runs.

Timestamps aren't the only source of trouble that exhibits itself as intermittently failing tests. Most prominently, multithreading tends to complicate things. The same goes for pretty much anything that involves random number generators. Some of these cases are more obvious, and some aren't that obvious and are thus not easy to protect ourselves from. Regardless of how early we realize the need for protecting ourselves from the random test failure, the way we tend to solve these situations is the same:

- 1 Avoid it.
- 2 Control it.
- 3 Isolate it.

The simplest solution would naturally be to avoid the problem altogether, cutting off any sources of nondeterministic behavior. For example, we might've sorted the log files according to their numeric suffix instead of relying on the timestamp.

If we can't easily go around the tricky bits, we try to control it. For example, we might replace a random number generator with a fake that returns exactly what we want.

Ultimately, if we don't find a way to avoid or control the source of the problem sufficiently, our last resort is to isolate the hard stuff to as small a place in our code base as possible. This is to relieve most of our code from the nondeterministic behavior and to let us tackle the tricky stuff in one place and only in one place.

5.3.3 Summary

Flaky tests that fail at random are about as trustworthy as most casinos, and they're generally more trouble than they're worth. You don't want such a test failing a build and forcing somebody to stop what they're doing only to find out that "it's that test

again.” You shouldn’t fall into despair either, or give up too easily, as these tests are often not that difficult to fix and make trustworthy.

Whether the intermittent failures are due to overreliance on the system clock, timing of execution, concurrent access to shared data, or an object responsible for generating random results, you have a number of options. You can try to work around the problem, control the source of nondeterministic behavior, or isolate and deal with it in all of its ugliness. In particular, testing multithreaded logic can be verbose business, as you need to reach out to all sorts of synchronization objects for help in wrangling the fundamentally unpredictable scheduling of threads during test execution.⁴

Now let’s look at a code odor that’s generally easier to remove.

5.4 Crippling file path

A *crippling file path* is one that immobilizes your code, preventing it from running on anybody else’s computer but yours. We all know this problem. I’d be surprised if you’ve never looked at a piece of code and said, “We shouldn’t hardcode that path here,” knowing that the explicit reference to a certain location in the filesystem is going to come back to haunt you later.

Regardless of how well-known this code smell is, it’s surprisingly common. Furthermore, there’s more than one way to cripple your mobility with file paths, so let’s take a look at an example that lets us explore some of them.

5.4.1 Example

The following example shows a simple test case that reads a document from the filesystem using an absolute file path. We know from chapter 2 that we should avoid file access in our unit tests. With that said, let’s assume that we’ve inherited some test code that accesses the filesystem. The following listing shows what the culprit looks like.

Listing 5.8 Absolute file paths can trip over your tests real quick

```
public class XmlProductCatalogTest {
    private ProductCatalog catalog;

    @Before
    public void createProductCatalog() throws Exception {
        File catalogFile = new File("C:\\workspace\\catalog.xml");
        catalog = new XmlProductCatalog(parseXmlFrom(catalogFile));
    }

    @Test
    public void countsNumberOfProducts() throws Exception {
        assertEquals(2, catalog.numberOfProducts());
    }

    // remaining tests omitted for brevity
}
```

⁴ For a more thorough discussion, check out chapter 7 of *Test Driven* from Manning Publications (2007).

So what's wrong about that file path? First of all, the absolute file path is clearly tied to the Microsoft Windows operating system. Any developer trying to run this test on a Mac or Linux system would quickly stumble on "file not found" style I/O errors. This test code isn't self-contained. You want to be able to simply check out the sources from version control and see all tests passing. That won't be possible with the kind of platform-specific absolute file paths like the one in listing 5.8.

5.4.2 What to do about it?

So we have a Windows-only, absolute file path. That won't work on any Mac or Linux system. Unless we're all strictly Windows users, at the least we should change the path to the platform-agnostic style supported by Java's File API:

```
new File("/workspace/catalog.xml");
```

This *could* work on both families of platforms, Windows and UNIX, provided that on Windows you're on the C: drive and there's a directory called C:\workspace, and that on a UNIX system there's a /workspace directory.

We're still tied to a specific location of the workspace. This isn't quite as bad as being tied to a specific user name (think: /Users/lasse),⁵ but it essentially forces all developers to place their workspace in the same physical location.

In short, you should avoid absolute paths whenever you're dealing with files in your test code. There may be acceptable situations where that's OK, but as a rule of thumb, steer clear of them. Instead, reach for abstract references such as system properties, environment variables, or whenever you can, relative paths.

The reason I recommend opting for a relative path instead of something like `System.getProperties("user.home")`, which returns a valid path on all platforms, is those references are still absolute paths and they still force all developers to place

Replace files with streams

By now you've probably noticed that the `java.io.File` API seems to inflict all kinds of testability issues on you. With that in mind, it might be a good idea to use `java.io.InputStream` and `java.io.OutputStream` in your internal APIs instead of `java.io.File`.

One immediate benefit is that you can always substitute a test double because they're interfaces rather than `final` classes like `java.io.File` is. That means you don't need to worry about file paths and ghost files except where you truly care about those files' physical location on the hard drive.

In your tests you can pass around pure in-memory objects such as `java.io.ByteArrayInputStream` and `java.io.ByteArrayOutputStream` instead of `java.io.FileInputStream` and `java.io.FileOutputStream`.

⁵ Imagine a project where everybody has created a second user in their systems because the lead developer had hardcoded his own home directory throughout the code base. Even the production web servers are running under his user name. Doesn't sound like a professional approach, does it?

their workspace in a specific location. This is particularly difficult in a situation when you're developing a new version of a product and maintaining older versions installed on the field. Whenever you switch from new development to fixing a bug in one of the old branches, you'll have to check out the old branch on top of your ongoing work, because the absolute file path always points to that same location.

Always go for a relative path if at all possible and consider an absolute path reference as the very last option. Relative paths work wonders as long as the files you're dealing with reside within the project's root directory. In our example we might state that the catalog document would reside at `PROJECT_ROOT/src/test/data/catalog.xml`. Then we could refer to it with a relative path like this:

```
new File("./src/test/data/catalog.xml");
```

Running our tests through the IDE or with a build script, that relative path would be evaluated against the project's root directory, allowing me as a Mac user to have my workspace under `/Users/lasse/work` and my Linux-running buddy Sam to stash his working copy under `/projects/supersecret`. Furthermore, Sam could move his entire working copy to another location and everything would still work without changing any file paths in code.

Speaking of moving things around, one more approach is worth mentioning. If the file you're dealing with is tied to a particular test class or a set of test classes in the same package, it might make sense to forget about the filesystem altogether and locate the file through Java's classpath instead. Let's say you have a test class called `XmlProductCatalogTest` residing in the `com.manning.catalog` package. The source code is at `src/test/java/com/manning/catalog/XmlProductCatalogTest.java`. Now, if you also placed your catalog XML document in the same place—in `src/test/java/com/manning/catalog/catalog.xml`—then you could look it up in your test code like so:

```
String filePath = getClass().getResource("catalog.xml").getFile();
File resource = new File(filePath);
```

The classpath API showcased here gives you access to the file that sits on your classpath and, because it's being looked up from a class residing in the same package, you don't even have to hardcode the exact package.⁶ In other words, if you decide to move the test class to another package as part of a refactoring, you don't have to change any file path references, as long as you remember to move the data file over as well.

5.4.3 Summary

You know that absolute file paths which only work for one operating system are generally a bad idea. Even if we all used the same platform today doesn't mean that we wouldn't stumble tomorrow when somebody upgrades to the latest Windows version, switches from Windows to Linux, or becomes the envy of the whole floor with a shiny new Mac.

⁶ Although you can if you want to.

It's not just about operating systems either. A more fundamental problem with absolute paths is that even if you represent them in a cross-platform manner, they still set a hard requirement for all developers to have that resource at that specific location. This can be extremely annoying if you're, for example, switching between multiple branches. Yes, you can check out multiple working copies of the project's code base, but they all point to the same place with that absolute file path.

As a rule of thumb, try to place all project resources under the project's root directory. That gives you the ability to use relative file paths throughout your code base and build files. For resources used by test code, you might even consider putting those data files next to the test code and looking them up through the classpath.

Having said that, there are exceptions to this rule, and not all files should reside under the project root. *Persistent temp file* is one such example.

5.5 Persistent temp files

The whole idea of a temporary file is to be temporary and be thrown away, deleted, after you're done with it. What the *persistent temp file* test smell represents is the case when the temporary file isn't that temporary but rather persistent, to the extent that it won't be deleted before the next test runs.

As they say, assumption is the mother of all screw-ups, and we programmers tend to assume that temporary files are, well, temporary. This can lead to surprising behavior we'd rather avoid debugging. Indeed, you should try to avoid using files altogether where they are not essential to the objects you're testing.

With that said, let's dive into an example that explains the issue with persistent temporary files in more concrete terms.

5.5.1 Example

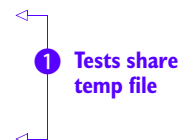
Imagine a sequence of tests exercising a file-based product catalog object. The first test creates an empty temp file containing no items, then initializes the catalog with that empty file, and checks that the catalog claims to be empty as expected. The second test doesn't create a file but instead initializes the catalog on top of a missing file, checking that the catalog behaves as if the file exists but is empty. The third test again creates a temp file—this time with two products—initializes the catalog with that file, and checks that the catalog indeed contains those two products.

This listing shows the interesting bits of such a test class.

Listing 5.9 Tests working on a temp file

```
public class XmlProductCatalogTest {
    private File tempfile;

    @Before
    public void prepareTemporaryFile() {
        String tmpDir = System.getProperty("java.io.tmpdir");
        tempfile = new File(tmpDir, "catalog.xml");
    }
}
```



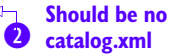
```

@Test
public void initializedWithEmptyCatalog() throws Exception {
    populateCatalogWithProducts(0);
    ProductCatalog catalog = new XmlProductCatalog(tempfile);
    assertEquals(0, catalog.numberOfProducts());
}

@Test
public void initializedWithMissingCatalog() throws Exception {
    ProductCatalog catalog = new XmlProductCatalog(tempfile);
    assertEquals(0, catalog.numberOfProducts());
}

@Test
public void initializedWithPopulatedCatalog() throws Exception {
    populateCatalogWithProducts(2);
    ProductCatalog catalog = new XmlProductCatalog(tempfile);
    assertEquals(2, catalog.numberOfProducts());
}
}

```

 2 Should be no catalog.xml

What happens when JUnit picks up this test class and starts executing is that the private field, `tempfile`, is ❶ initialized with a new `File` object before each test. That's fine. The problem is that when the second test, `initializedWithMissingCatalog`, starts executing, the filesystem has an empty `catalog.xml` ❷ in the system's temporary directory, unlike what the test assumes.

This means that the test is going to pass regardless of whether `XmlProductCatalog` properly deals with the case of a missing file. Not good.

5.5.2 What to do about it?

In general, you should try to keep the use of physical files to a minimum; file I/O makes tests slower than manipulating, say, strings and other types of in-memory blobs. With that said, when you're dealing with temporary files in your tests, you need to remember that temporary files aren't *that* temporary out of the box.

For example, if you obtain a temporary file using the `File#createTempFile` API, the file is no more temporary than any other file reference you have. Unless you explicitly delete it, that file is going nowhere once the test is finished. Instead, it's going to stay around as a *ghost file*, potentially causing seemingly erratic test failures.

From this we can deduce simple guidelines for dealing with temporary files:

- Delete the file in `@Before`.
- Use unique names for temporary files if possible.
- Be explicit about whether or not a file should exist.

Regarding the first guideline, programmers sometimes mark their temporary files to be deleted with `File#deleteOnExit` and happily move on. That's not enough, as the files aren't actually deleted until the JVM process exits. Instead, you need to delete those files *before* each test method that uses them, so let's add a line to the setup:

```

@Before
public void prepareTemporaryFile() {
    String tmpDir = System.getProperty("java.io.tmpdir");
    tempfile = new File(tmpdir, "catalog.xml");
    tempfile.delete();
}

```

Avoid
ghost file

Explicitly deleting the file in @Before makes certain that there's no file at the specific path when the test begins execution.

Second, using `File#createTempFile` is recommended when the exact location and name of the file aren't that important. What you get is a guaranteed unique file name for every test, which means that you don't need to be afraid of inheriting a ghost file from another test. In this example the setup method would be changed to the following:

```

@Before
public void prepareTemporaryFile() {
    tempfile = File.createTempFile("catalog", ".xml");
}

```

Third, it's okay to be silent when you don't care about whether there is or isn't a physical file in the filesystem. But when you do care, you should be explicit about it. One way to do that in our example from listing 5.9 would be this:

```

@Test
public void initializedWithMissingCatalog() throws Exception {
    withMissingFile(tempfile);
    ProductCatalog catalog = new XmlProductCatalog(tempfile);
    assertEquals(0, catalog.numberOfProducts());
}

private void withMissingFile(File file) {
    file.delete();
}

```

Aside from ensuring that the missing file is actually missing, being explicit about what you expect from the execution environment helps express the test's intent.

5.5.3 Summary

One of the problems for tests dealing with files in the filesystem stems from the fact that the filesystem is a shared resource between all tests. This means that tests may trip over each other if one test leaves residue and the next test assumes that it's working against an empty directory or a missing file. Whether this results in tests passing by accident or to a seemingly random test failure when running a set of tests in a particular order, none of the consequences are desirable.⁷

To avoid or alleviate these problems, you should remember to delete any files generated by your tests as part of their teardown method. You should avoid using the

⁷ Though JUnit and its users tend to shun the idea of order-dependent tests, advocating isolation and independence, TestNG (<http://testng.org>), another popular test framework for Java, embraces test-ordering and allows the programmer to explicitly declare that certain tests are to be executed in a certain order.

same file path between multiple tests, significantly reducing the chances of test residue causing problems. Furthermore, you can significantly improve your tests' readability and maintainability by being explicit about which files you expect to exist or not to exist.

Perhaps most importantly—and this warrants repeating—you should try to avoid using physical files altogether where they are not essential to the objects you're testing.

5.6 Sleeping snail

Working with file I/O is dreadfully slow compared to working with in-memory data. Again, slow tests are a major drag to maintainability because programmers need to run the test suite again and again as they work on changes, whether adding new functionality or fixing something that's broken.

File I/O isn't the only source of slow tests. Often a much bigger impact to a test suite's execution time stems from the army of `Thread.sleep` calls put in place to allow other threads to finish their work before asserting on the expected outcomes and side effects. This sleeping snail is remarkably easy to spot: just look out for calls to `Thread.sleep` and keep an eye out for exceptionally slow tests. Unfortunately, getting rid of the stench may not be that straightforward. Having said that, it's doable and definitely worth the trouble.

Let's dig into this code smell and the appropriate deodorant with an example.

5.6.1 Example

Since threads often bump up the complexity of our code, let's pick an example that's fairly simple but that involves testing behavior in a concurrent access scenario. The next listing presents a test for a `Counter` object responsible for generating unique numbers. Obviously we want these numbers to be unique, even if multiple threads invoke our `Counter` simultaneously.

Listing 5.10 Testing multithreaded access to a counter

```
@Test
public void concurrentAccessFromMultipleThreads() throws Exception {
    final Counter counter = new Counter();

    final int callsPerThread = 100;
    final Set<Long> values = new HashSet<Long>();
    Runnable runnable = new Runnable() {
        @Override
        public void run() {
            for (int i = 0; i < callsPerThread; i++) {
                values.add(counter.getAndIncrement());
            }
        }
    };

    int threads = 10;
    for (int i = 0; i < threads; i++) {
        new Thread(runnable).start();
    }
}
```

① Threads increment counter in a loop

② Start threads

```

Thread.sleep(500);
int expectedNoOfValues = threads * callsPerThread;
assertEquals(expectedNoOfValues, values.size());
}

```

What the test does is ❶ set up several threads to repeatedly access the counter and ❷ start all the threads. Since the test has no way of knowing how the threads are scheduled and when they finish, we've added a call to `Thread#sleep`, ❸ making the test wait for half a second before ❹ continuing with the assertion. Though 500 milliseconds might sound exaggerated, it's not really. Even with a sleep of 300 milliseconds, the test would fail roughly 10% of the time on my computer—and that's *flaky* with a capital F!

Now, that 500-millisecond sleep is still not totally reliable; sometimes it might take longer for all threads to complete their job. As soon as the test fails due to that 500-millisecond sleep not being long enough, the programmer is likely to increase the sleep. After all, we don't like flaky tests.

Flakiness is definitely a big issue. But there's another problem with a test sleeping as in listing 5.10: it's pretty darn slow.⁸ Though 500 milliseconds doesn't sound like a lot, it adds up, especially if your code base has multiple tests with `Thread#sleeps` sprinkled all over.

So let's see what we can do about those sleepy tests.

5.6.2 What to do about it?

`Thread#sleep` is slow because there's a delay between the time the worker threads finish and when the test thread knows that they've finished. Even though some of the time the threads would all be finished in 10 milliseconds, sometimes it takes several hundred milliseconds—or more—and therefore we have to wait for hundreds of milliseconds every time, regardless of how much of that waiting is unnecessary.

The next listing presents a better approach, one that replaces the wasteful and unreliable `Thread#sleep`. The fundamental difference is that we use Java's synchronization objects from the `java.util.concurrent` package to make the test thread immediately aware of when the worker threads have completed their work.

Listing 5.11 Testing multithreaded access without sleeping

```

@Test
public void concurrentAccessFromMultipleThreads() throws Exception {
    final Counter counter = new Counter();

    final int numberOfThreads = 10;
    final CountDownLatch allThreadsComplete =
        new CountDownLatch(numberOfThreads);

    final int callsPerThread = 100;
    final Set<Long> values = new HashSet<Long>();
    Runnable runnable = new Runnable() {

```

⁸ If on average the code would take, say, 100 milliseconds to finish, we've just made our test 400% slower than it needs to be.


```

@Override
public void run() {
    for (int i = 0; i < callsPerThread; i++) {
        values.add(counter.getAndIncrement());
    }
    allThreadsComplete.countDown();
}
};

for (int i = 0; i < numberOfThreads; i++) {
    new Thread(runnable).start();
}

allThreadsComplete.await(10, TimeUnit.SECONDS);

int expectedNoOfValues = numberOfThreads * callsPerThread;
assertEquals(expectedNoOfValues, values.size());
}

```

2 Mark thread completed

3 Wait for threads to complete

We’ve highlighted the key differences in listing 5.11. The central piece in this solution is the synchronization **1** object we use for coordinating between the worker threads and the test thread. A `CountDownLatch` is initialized to count down from the number of worker threads, and each worker thread ticks the latch **2** once when it finishes its work. After starting up all worker threads, our test proceeds to wait on the latch **3**. The call to `CountDownLatch#await` blocks until either all threads have notified about their completion or the given timeout is reached.

With this mechanism, our test thread wakes up to perform the assertion immediately when all worker threads have completed. No unnecessary sleeping and no uncertainty about whether 500 milliseconds is long enough for all threads to finish.

5.6.3 Summary

A sleeping snail is a test that’s sluggish and takes (relatively speaking) forever to run because it relies on `Thread#sleep` and arbitrarily long waits to allow threads to execute before performing assertions or continuing with the workflow under test. This smells because thread scheduling and variance in execution speed imply that those sleeps must be much longer than what the threads would typically need to finish their work. All of those seconds add up, making our test suite slower with every single sleep.

The solution to getting rid of this smell is based on the simple idea that the test thread should be notified when each worker thread actually completes. As long as the test thread has this knowledge, the test can resume itself as soon as it makes sense, essentially avoiding the unnecessary wait associated with the `Thread#sleep` hack.

The solution in listing 5.11 builds on one of the standard synchronization objects in the Java API, `CountDownLatch`. The `CountDownLatch` is useful for ensuing test execution only when the required number of other threads has passed the latch. If you work with a system dealing with threads, do yourself a favor and dive into the utilities provided by the `java.util.concurrent` package.

Now, let’s move on from multithreading to something completely different—the unit test equivalent of an obsessive-compulsive disorder. I suppose that requires some explanation, so let’s get to it.

5.7 Pixel perfection

Pixel perfection is a special case of primitive assertion and magic numbers. The reason why I felt it should be included in this catalog of test smells is because it's so frequent in domains dealing with computer graphics, an area where people tend to report difficulties in testing their code.

In essence, the smell of pixel perfection floats around places where a test calls for a perfect match between expected and actual generated images, or when a test anticipates a given coordinate in a generated image to hold a pixel of a certain color. Such tests are notoriously brittle and generally break down with even the smallest changes in the input.

Let's examine this anti-pattern with an example from a project I worked on some 10 years ago.

5.7.1 Example

Once upon a time I was working on a Java application that allowed users to manipulate diagrams onscreen, dragging and dropping boxes on the canvas and connecting them with all sorts of connectors. If you've used any kind of visual diagramming tool, you probably have an idea of what I was doing.⁹ Figure 5.1 is a simplified illustration of the kind of boxes-and-lines diagram that my code would generate from an object graph.

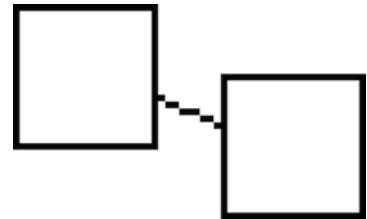


Figure 5.1 Two boxes that should be connected with a straight line

Among the tests I wrote for this code were elaborate definitions for how the borders of a box should look, how many pixels wide a box containing the word *foo* was expected to be, and so forth. Many of those tests suffered from severe pixel perfection. One test that was particularly interesting had to do with the connecting line between two boxes, a bit like figure 5.1. I've reproduced the test I'd initially written in this listing.

Listing 5.12 An extremely brittle test asserting exact pixel location and color

```
public class RenderTest {
    @Test
    public void boxesAreConnectedWithALine() throws Exception {
        Box box1 = new Box(20, 20);
        Box box2 = new Box(20, 20);
        box1.connectTo(box2);

        Diagram diagram = new Diagram();
        diagram.add(box1, new Point(10, 10));
        diagram.add(box2, new Point(40, 20));

        BufferedImage image = render(diagram);
    }
}
```

⁹ Except that mine looked downright ugly, to be honest.

```

    assertThat(colorAt(image, 19, 12), is(WHITE));
    assertThat(colorAt(image, 19, 13), is(WHITE));
    assertThat(colorAt(image, 20, 13), is(BLACK));
    assertThat(colorAt(image, 21, 13), is(BLACK));
    assertThat(colorAt(image, 22, 14), is(BLACK));
    assertThat(colorAt(image, 23, 14), is(BLACK));
    assertThat(colorAt(image, 24, 15), is(BLACK));
    assertThat(colorAt(image, 25, 15), is(BLACK));
    assertThat(colorAt(image, 26, 15), is(BLACK));
    assertThat(colorAt(image, 27, 16), is(BLACK));
    assertThat(colorAt(image, 28, 16), is(BLACK));
    assertThat(colorAt(image, 29, 17), is(BLACK));
    assertThat(colorAt(image, 30, 17), is(BLACK));
    assertThat(colorAt(image, 31, 17), is(WHITE));
    assertThat(colorAt(image, 31, 18), is(WHITE));
}
}

```

First of all, the test in listing 5.12 is hard to validate. I mean, how am I supposed to know whether there should be a black pixel at *(23, 14)*? Moreover, it's not hard to see how easily this test will break. Any change in how wide a diagram's borders are, for example, would likely offset the exact location of the connecting line somewhat and the test would break. This is bad because a change in what a box *looks like* shouldn't affect a test that checks whether two connected boxes in the diagram are also connected visually.

If we were testing the total rendered width of a box, we could tackle the issue of *magic numbers* by composing our assertion from variables like `boxContentWidth + (2 * boxBorderWidth)`, accommodating for changes in the exact border width. But in the case of the connecting line between boxes, it's not as obvious what we could do.

So, what could we do?

5.7.2 What to do about it?

One of the goals in writing tests is to express them at the appropriate level of abstraction. If you're interested in two boxes being connected with a line, you shouldn't be talking about pixels or coordinates. Instead you should be talking about *boxes* and whether they're *connected*. Simple, right?

What this means is that, instead of *primitive assertions*, you should hide the nitty-gritty details behind a custom assertion that speaks at the appropriate level of abstraction. It might also mean that, instead of exact value-to-value comparisons, your assertion needs to be a *fuzzy match*—an actual algorithm.

With this in mind, we could rewrite our test from listing 5.12 to delegate the details of checking whether there's a connection between the two boxes to a custom assertion. The custom assertion would then take care of figuring out how to check that (and preferably in a flexible, nonbrittle manner). The following listing shows what that test might look like.

Listing 5.13 A flexible test talks about stuff at the appropriate level of abstraction

```

public class RenderTest {
    private Diagram diagram;

    @Test
    public void boxesAreConnectedWithALine() throws Exception {
        Box box1 = new Box(20, 20);
        Box box2 = new Box(20, 20);
        box1.connectTo(box2);

        diagram = new Diagram();
        diagram.add(box1, new Point(10, 10));
        diagram.add(box2, new Point(40, 20));

        assertThat(render(diagram),
            hasConnectingLineBetween(box1, box2));
    }

    private Matcher<BufferedImage> hasConnectingLineBetween(
        final Box box1, final Box box2) {
        return new BoxConnectorMatcher(diagram, box1, box2);
    }

    // rest of details omitted for now
}

```

The main difference between listings 5.12 and 5.13 is that the long list of pixel-specific color tests at seemingly arbitrary coordinates has been replaced with a custom assertion that reads like plain English—there should be a connecting line between these two boxes. This is much better from the readability perspective and makes this particular test much less work to maintain.

The complexity hasn't evaporated altogether, though. We still need to tell the computer how it can discern whether two boxes are connected, and that's the responsibility of the mysterious `BoxConnectorMatcher`.

The implementation details of the `BoxConnectorMatcher` could be seen as somewhat of a tangent and unessential for the purpose of this test-smell catalog. After all, it's a solution for one specific case. But I find it useful to take a closer look at the implementation because so many programmers seem to have lost their faith in graphics-related code being at all testable.

With that disclaimer out of the way, the following listing spills the details on `BoxConnectorMatcher` and shows one way of implementing the kind of smart assertion that doesn't break as soon as the boxes' respective locations move a pixel to some direction.

Listing 5.14 Checking the connection between two boxes with a custom matcher

```

public class BoxConnectorMatcher extends BaseMatcher<BufferedImage> {
    private final Diagram diagram;
    private final Box box1;
    private final Box box2;

    BoxConnectorMatcher(Diagram diagram, Box box1, Box box2) {
        this.diagram = diagram;
        this.box1 = box1;
    }
}

```

```

        this.box2 = box2;
    }

    @Override
    public boolean matches(Object o) {
        BufferedImage image = (BufferedImage) o;
        Point start = findEdgePointFor(box1);
        Point end = findEdgePointFor(box2);
        return new PathAlgorithm(image)
            .areConnected(start, end);
    }

    private Point findEdgePointFor(final Box box1) {
        Point a = diagram.positionOf(box1);
        int x = a.x + (box1.width() / 2);
        int y = a.y - (box1.height() / 2);
        return new Point(x, y);
    }

    @Override
    public void describeTo(Description d) {
        d.appendText("connecting line exists between "
            + box1 + " and " + box2);
    }
}

```

① Locate edge pixels

② Delegate!

The essence of the `BoxConnectorMatcher` is in the `matches()` method where it ① finds an edge point—any edge point—for both boxes and delegates to a `PathAlgorithm` ② object to determine whether the two coordinates are connected in the image.

The implementation of `PathAlgorithm` is shown next. A word of warning: it's a lengthy piece of code and I'm not going to explain it in much detail.

Listing 5.15 The `PathAlgorithm` is responsible for finding a path between two pixels

```

public class PathAlgorithm {
    private final BufferedImage img;
    private Set<Point> visited;
    private int lineColor;

    public PathAlgorithm(BufferedImage image) {
        this.img = image;
    }

    public boolean areConnected(Point start, Point end) {
        visited = new HashSet<Point>();
        lineColor = img.getRGB(start.x, start.y);
        return areSomehowConnected(start, end);
    }

    private boolean areSomehowConnected(Point start, Point end) {
        visited.add(start);
        if (areDirectlyConnected(start, end)) return true;
        for (Point next : unvisitedNeighborsOf(start)) {
            if (areSomehowConnected(next, end)) return true;
        }
    }
}

```

```

        return false;
    }

    private List<Point> unvisitedNeighborsOf(Point p) {
        List<Point> neighbors = new ArrayList<Point>();
        for (int xDiff = -1; xDiff <= 1; xDiff++) {
            for (int yDiff = -1; yDiff <= 1; yDiff++) {
                Point neighbor = new Point(p.x + xDiff, p.y + yDiff);
                if (!isWithinImageBoundary(neighbor)) continue;
                int pixel = img.getRGB(neighbor.x, neighbor.y);
                if (pixel == lineColor && !visited.contains(neighbor)) {
                    neighbors.add(neighbor);
                }
            }
        }
        return neighbors;
    }

    private boolean isWithinImageBoundary(Point p) {
        if (p.x < 0 || p.y < 0) return false;
        if (p.x >= img.getWidth()) return false;
        if (p.y >= img.getHeight()) return false;
        return true;
    }

    private boolean areDirectlyConnected(Point start, Point end) {
        int xDistance = abs(start.x - end.x);
        int yDistance = abs(start.y - end.y);
        return xDistance <= 1 && yDistance <= 1;
    }
}

```

The implementation of `PathAlgorithm` is basically a depth-first search, starting from the starting coordinate and recursing into each neighboring coordinate of a matching color, until we either run out of unvisited connected points or we stumble on a coordinate that's a direct neighbor of the ending coordinate.

5.7.3 Summary

Pixel perfection: as its name implies, is a test smell specific to graphics and code-producing graphical output. It's a sort of combination of magic numbers and primitive assertion where a test is made both extremely difficult to read and extremely brittle.

Such tests are next to unreadable because they assert against hardcoded, low-level details such as pixel coordinates and colors, even though the test would semantically be about a much higher-level concept. Whether a pixel at a given coordinate is black or white is different from concepts such as two shapes being connected or one shape being drawn on top of another.

Such tests are extremely brittle because even small and supposedly unrelated changes in the input—whether it's another image, an object graph, or the way a graphical object is rendered onscreen—can affect the output enough to break tests that scrutinize the output by checking pixel-precise coordinates for an exact color match. This

is the same problem we'd be facing with the *golden master* technique of comparing a rendered image to a canned image that we've manually checked to be correct.

These aren't the kinds of tests that we'd like to maintain. We want to write tests without such brittle precision, using fuzzy matching and smart algorithms instead of trivial value comparisons.

What the example solution in section 5.7.2 proves is that we can write assertions against graphical output that express our intent at the appropriate level of abstraction—that is, higher-level concepts than pixels or coordinates. But you've also seen that such custom assertions can be nontrivial to implement. Regardless, when weighed against the extreme brittleness of pixel perfection, the payout is often worth the effort.

Most of the test smells you've seen so far have been undoubtedly bad stuff that you should avoid as a rule of thumb. The next smell is different as it's essentially a good thing turned bad.

5.8 Parameterized mess

As professional programmers, it's our duty to study a wide range of techniques and technologies, improving our ability to develop software. Some of these techniques are so great that we tend to overdo them, as we're always on the lookout for even a remote chance or excuse to apply the latest and greatest tool we've just added to our toolbelt. In the context of unit testing and especially JUnit 4, one of the most commonly over-used techniques is a pattern called parameterized test.¹⁰

The parameterized test pattern

The parameterized test pattern is, in essence, a way to remove duplication from data-oriented tests that only differ in small ways.¹¹ For example, consider a test class where the only difference between its 13 test methods is the simple input and expected output values for the code under test. Let's say the code takes in an `int` and returns a `String`. You could turn this test class into a parameterized test and delegate some of the boilerplate to the test framework itself; you'll see a concrete example of what this looks like in section 5.8.2.

With all the other ways at our disposal for keeping our test code clean and easy to maintain, the context in which the parameterized test pattern truly has an edge over the other alternatives is when the test data isn't known before runtime, which is rarely the case with unit tests.

¹⁰ Other test frameworks such as TestNG also support this pattern, but none of them are as wide spread as JUnit.

¹¹ Gerard Meszaros. *xUnit Test Patterns: Refactoring Test Code*, Addison-Wesley, 2007.

The parameterized test is a fine pattern, but when used too eagerly and in the wrong context, *the parameterized test* turns into a code smell—a *parameterized mess* that's difficult to understand and to pinpoint the actual failure when one of the tests fails.

5.8.1 Example

I'd like to explore an example of this code smell by first looking at the kind of code from which we usually refactor toward the parameterized test pattern. When it comes to parameterized tests, the trigger is generally a situation not unlike our starting point in this listing.

Listing 5.16 Typical starting point for parameterized test pattern

```
public class RomanNumeralsTest {
    @Test
    public void one() {
        assertEquals("I", format(1));
    }

    @Test
    public void two() {
        assertEquals("II", format(2));
    }

    @Test
    public void four() {
        assertEquals("IV", format(4));
    }

    @Test
    public void five() {
        assertEquals("V", format(5));
    }

    @Test
    public void seven() {
        assertEquals("VII", format(7));
    }

    @Test
    public void nine() {
        assertEquals("IX", format(9));
    }

    @Test
    public void ten() {
        assertEquals("X", format(10));
    }
}
```

What we have in the listing is a bunch of one-liner test methods that only differ in the input and expected output value. The code under test is a utility that's responsible for rendering numbers formatted as Roman numerals.

This is exactly the kind of situation where the parameterized test pattern is straightforward to apply, reducing line count mainly by getting rid of the boilerplate, and that's what the test method signatures in listing 5.16 currently are.

The following listing shows the same test rewritten to use the parameterized test pattern and JUnit's Parameterized test runner.

Listing 5.17 The parameterized test pattern reduces boilerplate

```
@RunWith(Parameterized.class)
public class RomanNumeralsTest {
    private int number;
    private String numeral;

    public RomanNumeralsTest(int number, String numeral) {
        this.number = number;
        this.numeral = numeral;
    }

    @Parameters
    public static Collection<Object[]> data() {
        return asList(new Object[][] { { 1, "I" }, { 2, "II" },
            { 4, "IV" }, { 5, "V" }, { 7, "VII" },
            { 9, "IX" }, { 10, "X" } });
    }

    @Test
    public void formatsPositiveIntegers() {
        assertEquals(numeral, format(number));
    }
}
```

← ① Special test runner

③ Store parameters to fields

← List of parameter sets to test with

②

← Tests use data fields

④

The way JUnit's built-in facility for implementing parameterized tests works is simple. We need to tell JUnit to execute our test class with the Parameterized runner ①. When the Parameterized runner takes over, it looks for a no-argument public static Collection<Object[]> method tagged with the @Parameters ② annotation. This method is responsible for returning a list of parameters; that is, the different variations of the data, including the input as well as expected output needed by the test. Looping through this data, the runner instantiates the test class once for each set of parameters, passing the parameters to a constructor that stashes them to private fields ③. Finally, the actual test methods use the fixture stored in private fields ④ to perform their assertions.

The problem with this instance of parameterized test comes down to a balance between two kinds of a readability challenge. In listing 5.16 we observed boilerplate that makes it difficult to see the beef—the one-liner inside each individual test method. In listing 5.17 we've reduced the raw line count by one third, but in turn we now have a cluttered syntax of lists inside lists that's somewhat cumbersome to figure out. That problem only gets worse when the number and complexity of parameters gets bigger and the list grows longer. For example, take a look at this data set I found in the tests for an open source accounting software:

Listing 5.18 The more parameters, the harder to follow and modify

```

@Parameters
public static List<Object[]> getData() {
    return asList(new Object[][] {
        { SCHEDULE_FREQUENCY_EVERY_DAY.toString(),
          getDate(2007, JANUARY, 1), getDate(2007, JANUARY, 31),
          getDate(2007, FEBRUARY, 1), -31001 },
        { SCHEDULE_FREQUENCY_EVERY_DAY.toString(),
          getDate(2007, JANUARY, 1), getDate(2007, JANUARY, 31),
          getDate(2007, MARCH, 1), -31001 },
        { SCHEDULE_FREQUENCY_EVERY_WEEKDAY.toString(),
          getDate(2007, JANUARY, 1), getDate(2007, JANUARY, 31),
          getDate(2007, FEBRUARY, 1), -23001 },
        { SCHEDULE_FREQUENCY_BIWEEKLY.toString(),
          getDate(2007, JANUARY, 1), getDate(2007, JANUARY, 31),
          getDate(2007, FEBRUARY, 1), -2001 },
        { SCHEDULE_FREQUENCY_EVERY_WEEKDAY.toString(),
          getDate(2007, JANUARY, 1), getDate(2007, JANUARY, 31),
          getDate(2007, FEBRUARY, 1), -23001 },
    });
}

```

The good news is that it's still just code, so we have our usual means of making the @Parameters method more accessible. There's an additional problem that's harder to solve.

Figure 5.2 shows the Eclipse test runner reporting a failure of one of our test cases in listing 5.17. Tests dynamically generated by the Parameterized test runner are essentially anonymous and carry no label beyond a running number.

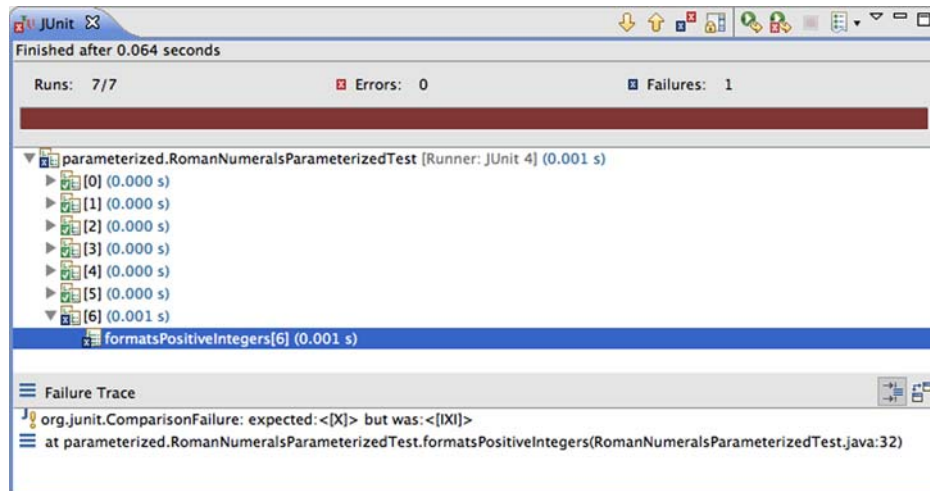


Figure 5.2 Parameterized tests are practically anonymous, which makes it harder to interpret test results.

Without recognizable names or identifiers, when one of these tests fails it can be difficult to determine which of the n parameter sets is failing; the only information available is the assertion failure message. In this example, the error message happens to contain enough details to identify the failing test case. But if we were making more than one value comparison, for example, we'd only know one variable that may or may not be unique within our whole data set.

Now that I've pointed out some of the challenges with parameterized tests that can turn them into a source of code smells and maintainability issues, let's talk about what we can do about those problems.

5.8.2 What to do about it?

You may have heard the joke about a man complaining to the doctor that "My arm hurts when I do this," and the doctor responding with, "Well, don't do that." That can be the easiest solution to a parameterized mess, too.

With that in mind and observing that so many parameterized tests end up causing grey hair and naughty verbal expressions at the office, think twice before applying the parameterized test pattern. For example, an alternative solution might be to reduce the boilerplate-ridden scatter in listing 5.16 by packing multiple test cases inside one test method:

```
public class RomanNumeralsTest {
    @Test
    public void formatsPositiveIntegers() {
        assertEquals("I", format(1));
        assertEquals("II", format(2));
        assertEquals("IV", format(4));
        assertEquals("V", format(5));
        assertEquals("VII", format(7));
        assertEquals("IX", format(9));
        assertEquals("X", format(10));
    }
}
```

Having said that, when you do decide to apply the parameterized test pattern, there are a few simple things you can do to avoid the problems of hard-to-read test data and anonymous test failures that are difficult to identify. Let's start by paying attention to simplifying the test data and making it easier for the eye.

The essence of the readability problem in listing 5.17 comes down to long sequences of anonymous data:

```
new Object[][] { { 1, "I" }, { 2, "II" }, { 4, "IV" },
                 { 5, "V" }, { 7, "VII" }, { 9, "IX" }, { 10, "X" } }
```

To make this kind of multilevel structure easier to follow, you need to find a way to separate the parameter sets from each other. The simplest way to do that would be to add them to a list one by one instead of using the inline-array declaration style, but that would also add the kind of verbosity and duplication we want to get away from.

You could also indent the list so that each data set resides on its own line, but that approach generally prevents you from using your IDE's autoformatting features.¹²

Other solutions to consider might involve wrapping each individual data set with a `varargs` method a bit like this:

```
@Parameters
public static Collection<Object[]> data() {
    return asList(set(1, "I"), //
        set(2, "II"), //
        set(4, "IV"), //
        set(5, "V"), //
        set(7, "VII"), //
        set(9, "IX"), //
        set(10, "X"));
}

private static Object[] set(Object... values) {
    return values;
}
```

We've done two things here. First, we replaced the curly braces with a method call that effectively separates each parameter set with a whole word, `set`. This is generally an easier boundary to distinguish visually. Second, we've added comments between each set to keep our IDE's autoformatting from messing things up the next time somebody wants to fix up code formatting.

Now, what can you do with the problem of parameterized, anonymous tests failing, and not knowing which data set that arbitrary "[123]" in your test report refers to? This problem stems from JUnit's implementation of the `Parameterized` test runner that doesn't provide a way to link an individual test's name to its data set. That's why you need to document the test case's identity in your assertions' failure messages.

```
@Test
public void formatsPositiveIntegers() {
    assertEquals(dataset(), numeral, format(number));
}

private String dataset() {
    StringBuilder s = new StringBuilder();
    s.append("Data set: ").append(this.number);
    s.append(", ").append(this.numeral).append("\n");
    return s.toString();
}
```

You can build a failure message that describes the full data set in a manner similar to how we've constructed it in the `@Parameters` method, making JUnit render that information when this test fails. Even if the test runner's hierarchy still won't identify your test by a meaningful name, you have the next best thing—it's tucked slightly farther away in the failing test's error message.

¹² Having used Eclipse for so many years, I've developed an almost unconscious habit of automatically activating the format shortcut when opening a source file in an editor. That habit tends to not conflict with a convention that requires manual indentation case by case.

5.8.3 Summary

The parameterized test pattern is a way to express repetitive tests in a compact manner when they only differ in terms of input and output values. Domains where this pattern tends to surface most often include validation, translation, string manipulation, and all sorts of mathematics. In many cases, the improvement in reduced verbosity is practically negated by the increased scatter and the reduced ability to trace a test failure back to the specific data set causing that test failure.

For those situations where you decide to go with a parameterized test, I identified three ways of minimizing the undesirable effects of applying the parameterized test pattern.

In order to help you visually distinguish the individual data sets from each other, you can try to wrap each data set into a method call. A method call tends to pop out easier from the surrounding code than the squiggly brackets that come with the usual way of building the list of data sets.

You can also lean on indentation to separate data sets from each other. Modern IDEs give you a powerful ability to automatically indent and format your code according to a uniform style. This is extremely useful for maintaining consistency throughout the code base, but it also tends to destroy carefully hand-indented listings. For this reason you might want to suffix each line with a tiny comment to keep your IDE from ravaging your special indentation.

Finally, since JUnit's parameterized test implementation doesn't give you a way to trace an executed test back to the original data set, you need to rely on the failure message thrown from a failing assertion. Including the full data set for the test case as part of each assertion message helps quickly identify the scenario.

Each of these steps can save you an odd minute, and those minutes add up to big bucks over time. Our next code smell is the same; though it might not seem like a big thing, even small things add up and it's our job as programmers to make sure we're as productive as we can be.

Whereas the parameterized test pattern is specific to test code, our next code smell has wider roots in no less than object-oriented code quality metrics.

5.9 Lack of cohesion in methods

Cohesion is a central property of well-factored object-oriented code. Simply put, cohesion means that a class represents a single *thing*, a single abstraction. Cohesion is good—we want high cohesion—and the *cohesion* is a code smell.

The *in methods* part refers to the way we determine the degree of cohesion by looking at the commonalities between the methods of a class. Though there are many variations in how to calculate the lack of cohesion in methods, the common denominator is that they're all based on the rough idea that perfect cohesion means that every field of a class is used by every method on that class.

In the context of unit tests and unit testing terminology, this could be expressed in such a way that each test in a class should use the same *test fixture*. Conversely, if tests work on different fixtures, they should be refactored into more than one test class.

Let's look at a concrete example of this test smell and discuss the practical problems that stem from lack of cohesion.

5.9.1 Example

The following example in listing 5.19 comes from an open source accounting software package.¹³ The code we're going to zoom in on deals with *split accounting*, which is used to spread the cost of an item across multiple accounting groups. For example, you might split the price of a new database server among three different budgets because the server will be used by three different departments.

The code base handles split accounting operations with three domain objects: Account, Split, and BudgetCategory. The operation itself is represented by a Transaction object. The test class in listing 5.19 initializes a fixture of these domain objects in a @Before method, and the class has a handful of test methods working on those objects. I'm only showing two of those tests for brevity's sake.

Listing 5.19 Test class suffering from lack of cohesion in methods

```
public class SplitsTest {
    Account account;
    Split split;
    BudgetCategory bc1, bc2, bc3, bc4;

    // ...

    @Test
    public void fromSplits() throws Exception {
        List<TransactionSplit> fromSplits =
            new ArrayList<TransactionSplit>();
        fromSplits.add(createSplit(bc3, 1200));
        fromSplits.add(createSplit(bc4, 34));
        Transaction t = createTransaction(split, account);
        t.setFromSplits(fromSplits);
        assertTrue(transactions(t).size() == 1);
    }

    @Test
    public void toSplits() throws Exception {
        List<TransactionSplit> toSplits =
            new ArrayList<TransactionSplit>();
        toSplits.add(createSplit(bc1, 1200));
        toSplits.add(createSplit(bc2, 34));
        Transaction t = createTransaction(account, split);
        t.setToSplits(toSplits);
        assertTrue(transactions(t).size() == 1);
    }
}
```

1 Multiple fixture objects

2 Tests only use subsets of fixture

Let's cut to the chase. The smell in listing 5.19 comes from the test class's fixture consisting of ❶ several BudgetCategory objects and the ❷ test methods only using some

¹³ If you're the worrying type, make sure to never ever see the actual code that's supposed to take care of your pension plan, insurance, or bank account. Trust me. It's better that way.

of them. That's a clear violation of the idea that each test works on the exact same fixture; we have lack of cohesion in (test) methods.

From the grand perspective, having a more complex fixture than necessary makes it harder for the programmer to understand what's going on. More fields means more moving things and more variables to wrap your head around.

It's also hard, looking at one of the tests, to figure out which field it uses, as it's not obvious. Instead we're left wondering, "Which category was bc3 again?" This problem has its roots in one of the fundamental code smells: bad names. Though lack of cohesion in test methods doesn't make it impossible to name your fixture objects well, it tends to make it pretty darn difficult.

For example, each test might work on an input and output, but since there's one pair of these objects for every test, you have to name them along the lines of `empty-Input`, `inputWithOneThing`, `inputWithThreeIdenticalThings`, and so forth. This leads to a situation where it's not trivial to look at the fixture and say, "Of course. These two go together and the rest don't have any part in this test."

So what can you do in these situations?

5.9.2 What to do about it?

We've been talking about the lack of cohesion in methods being an object-oriented metric that carries particular relevance in the context of test code. It also bears a resemblance to another test smell we discussed in chapter 4. That smell was called *split personality*. Only this time it's not a test method suffering from split personality but the test class.

We've already established one solution to this problem—the class should be split. Another, typically much better solution might be to coerce the tests to use the same fixture objects. For example, you could look at the difference between `bc1`, `bc2`, `bc3`, and `bc4` in listing 5.19 and see whether we can get rid of two.

Sometimes that doesn't work; we actually do need more fixture objects than an individual test needs. In those cases it's often best to split the class, looking for suitable boundaries such as the ones illustrated by figure 5.3. The question is, how do we go about doing that? What should we look for as hints of where to split a test class?

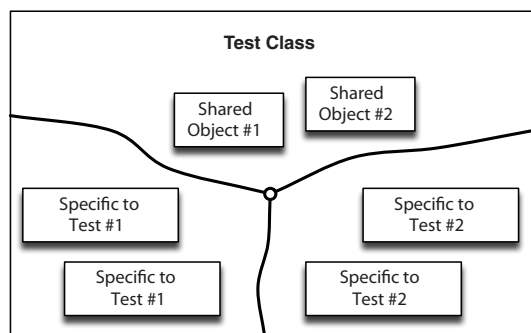


Figure 5.3 Splitting a test class into two, we need to identify which parts of the fixture are shared between all tests and which are specific to one or more tests.

For example, you can only see two tests in listing 5.19, but none of the remaining tests in the original class worked on more than two `BudgetCategory` objects, and always either `bc1` and `bc2` or `bc3` and `bc4`. You should look for clusters like these to find natural boundaries along which to split the class into two or three smaller and more focused—cohesive—test classes.

Sometimes all you need to do then is to create another test class, move some methods and fixture objects over, and dust off your hands. More often than not, the tests depend on the same utility methods, and we don't like duplication. In those cases we also need to either move those utility methods to a separate class that both of your split test classes can use, or extract a common base class that provides these facilities to all the split test classes. This latter approach might result in test classes that look like the next two listings.

Listing 5.20 Split down test classes allow more specific fixtures and better cohesion

```
public class TestSplitsAcrossDifferentBudgetCategories
    extends AbstractSplitsTestCase {
    private BudgetCategory incomeCategory, standardCategory;

    @Before
    public void setup() throws Exception {
        incomeCategory = new BudgetCategoryImpl();
        incomeCategory.setIncome(true);
        standardCategory = new BudgetCategoryImpl();
        standardCategory.setIncome(false);
    }

    @Override
    protected List<BudgetCategory> budgetCategories() {
        return asList(incomeCategory, standardCategory);
    }

    // tests omitted
}
```

Note how simple the fixture has become. Looking at this test class it's obvious which budget categories are involved and how they differ from each other. This is made possible by moving all the common bits into the abstract base class, `AbstractSplitsTestCase`, shown next.

Listing 5.21 Abstract base class for our split down tests

```
public abstract class AbstractSplitsTestCase {
    protected Account account;

    @Before
    public void initializeAccount() throws Exception { ... }

    /**
     * Concrete test classes implement this to give the utility
     * methods access to the BudgetCategory objects in use.
     */
    protected abstract List<BudgetCategory> budgetCategories();
```

① Fixture objects from concrete children


```

protected List<Transaction> transactions(Transaction t)
    throws ModelException {
    Document d = ModelFactory.createDocument();
    for (BudgetCategory bc : budgetCategories()) {
        d.addBudgetCategory(bc);
    }
    d.addAccount(account);
    d.addTransaction(t);
    return d.getTransactions();
}

protected TransactionSplit createSplit(BudgetCategory category,
    int amount) throws InvalidValueException { ... }

protected Transaction createTransaction(Source from, Source to)
    throws InvalidValueException { ... }
}

```

① Fixture objects
from concrete
children

Note how we're acquiring the actual `BudgetCategory` ① objects from our concrete children through an abstract method.

Inheritance and composition

What we're doing here is reuse by inheritance, which is somewhat suspect. After all, inheritance should imply an is-a relationship between the child and parent. The general recommendation would be to choose composition over inheritance; that is, reuse code by composing objects from other objects rather than reuse by sharing data and methods through the inheritance hierarchy.

I wouldn't be worried if you have an abstract test class here and there, but you could go overboard with inheritance, so think twice before introducing abstract base classes for your tests. One of the risks you're facing may be a performance issue, as we'll discuss in chapter 9.

By moving smaller clusters of test methods to their dedicated test classes, we've improved the problem with lack of cohesion and each test class is now focused around a specific fixture that no longer boggles our minds. We decided to extract a common base class in the process to avoid duplicating the plumbing that used to be in the private methods of our non-cohesive behemoth of a test class.

5.9.3 Summary

Lack of cohesion in methods means that the test methods in a test class are only interested in some of the fixture's objects.

The consequence of this is that a programmer developing or modifying such tests has to deal with more moving parts than necessary. This complexity makes it difficult to figure out which field is which, which of the fixture objects are used in which test, and which of them you should edit in the setup. Furthermore, with such overlaps between the kind of objects required by what could be dozens of tests, it tends to be increasingly difficult to name the fixture objects well.

All of these problems boil down and point to a simple truth: those tests don't belong in the same test class. Sometimes the complexity of the bloated test fixture is *accidental* and you can get rid of the whole problem by coercing the tests to use the same fixture objects, shrinking the fixture into something more manageable. Often, we're talking about essential complexity that's not—and shouldn't be—going anywhere.

When you find that those various combinations of fixture objects are necessary, your best remedy to the lack of cohesion tends to be twofold:

- 1 Move the tests to different test classes and (if necessary) extract a common base class to hold any shared plumbing.
- 2 Create the fixture inside each test method with utility methods on a separate class.

Though this kind of refactoring may seem like a lot of work, I assure you that it starts to look like a small price to pay the first time you screw up with a complex fixture lacking cohesion, breaking some functionality without noticing. It certainly did to me.

Mindful test data

This isn't to say that you should always strive to use the exact same data in each test. My friend J.B. likes to point out that mindfully using different test data in each test reduces the likelihood of reusing fixture objects that are only accidentally the same.

Similarly, if the exact value of a string isn't meaningful to a test, it might be a good idea to use a literal "whatever" instead of referring to a fixture object named `productDescription`. Readability trumps maintainability; conveying the meaning is often more important than perfect cohesion and absolute lack of duplication, so use your test data mindfully!

5.10 Summary

In this chapter we've explored some test smells that can make maintenance more difficult than it has to be. Though most code smells derive their stench from different kinds of readability problems, some smells are caused by other kinds of deficiencies that hinder productivity as we maintain and edit our tests.

We established duplication as the root of all evil and pointed out how each bit of duplication between our tests makes them that much harder to update and maintain. We found conditional logic in test code resulting in grey hair and premature aging because we can't figure out what the test is actually doing.

Particularly troubling test smells we looked at include flaky tests that will fail at random and file paths that cripple the mobility of our test code, failing on somebody else's computer. Also related to file paths, we touched on persistent temp files that linger around after their test, wreaking havoc on their successors who assume no such temp file exists.

The smell of a sleeping snail relates to multithreaded code that we've desperately tried to wrangle with calls to `Thread#sleep`, grinding our blazing fast tests almost to a halt as they wait for hundreds of milliseconds for something to happen.

Pixel perfection took us into the land of graphics and overly precise assertions—the computer graphics breed that combines magic numbers and primitive assertions into a hyperassertion that breaks as easily as a soap bubble.

Our analysis of the parameterized mess taught us to tread carefully with JUnit's `Parameterized` test runner, and also taught us some tricks for smoothing out the ride when one of those anonymous tests suddenly starts failing or when we need to edit our test cases.

Finally, we ventured into the land of metrics to reflect on what cohesion means for a test class—and what to do when we suffer from the lack of it.

This is by far not all. In the next chapter we'll dig into another set of test smells: tests that betray our trust in them.

Trustworthiness



In this chapter

- Test smells around code comments
- Test smells around poor expectation management
- Test smells around conditional execution

Trust is good and we generally like it when people are trustworthy. We'd prefer to be able to trust people, as it makes a lot of things in life easier. We'd also like to be able to trust our code—which is one of the reasons we write tests—and the trustworthiness of our tests is a big deal for us. Software development is essentially about modifying, evolving, and maintaining code, and if we can't trust our tests, we can't be too confident about our code working after even the most seemingly innocent change.

This chapter is a walkthrough of some of the biggest detractors to test trustworthiness, a parade of test smells revolving around the problem of tests not being reliable. The parade starts with different ways that code comments can go bad. Though the ability to annotate your code with comments written in plain English can be a real boon, these comments can easily turn from useful information to misinformation. Sometimes it's even worse, when the whole body of a test has been commented-out such that the test is still executed and passes with flying colors—but it doesn't actually do anything.

The remainder of the chapter revolves around various forms of broken (or falsely implied) promises that also lead unsuspecting programmers astray. We look at tests that never fail (even when they should) and tests that suggest a lot but do something completely different. Along the way we'll explore trust issues with conditional statements in our tests.

That's quite a parade so let's get started. I'm anxious to rant about one of my favorite topics—comments in your test code!

6.1 *Commented-out tests*

I don't know who originally came up with the idea that programmers should be able to sprinkle nonexecutable comments into their source code. (If you know, please tell.) I'm glad that we have that ability, as comments can be of great help in trying to understand code.

But often, comments backfire by confusing or even misleading us. Related to tests, a particularly interesting failure mode with comments is a test method that's been commented-out, as it doesn't communicate anything and just confuses people. When we do that, we're not commenting anything—we're using comments as a poor man's version control.

Let's take a look.

6.1.1 *Example*

Not long ago I was digging out code I'd written for a project from half a decade ago. It'd been a while since I looked at the code, let alone developed it, so I was just scanning the code base to remind myself what the architecture was like, and so forth. As usual, I familiarized myself with the various objects by looking at their tests. After all, the tests are what best document what the code actually does.

As I was scouring the tests for a class responsible for certain I/O operations, I found that one of the 15 or so tests in the test class was commented out. There wasn't a single comment in the whole source file except for this test, shown in this listing.

Listing 6.1 Why is this test commented out?

```
// @Test
// public void recognizesFileBeingModifiedWhenItIs() throws
// Exception {
//     File newFile = File.createTempFile(getName(), ".tmp");
//     FileAppendingThread thread = new FileAppendingThread(newFile);
//     thread.start();
//     thread.waitUntilAppendingStarts();
//     try {
//         Thread.sleep(200);
//         assertTrue(IO.fileIsBeingModified(newFile));
//     } finally {
//         thread.interrupt();
//     }
// }
```

I was baffled as I spotted the commented-out test. I scanned through it, biting my lip and blaming the world's IDEs for having messed up the formatting. (IDEs generally treat all comments the same and format them like plain text, not code that's supposed to be indented.) In the end, I still couldn't figure out *why* this test was commented out.

The same problem occurs with tests that have had their `@Test` annotation commented out. The code is there and it compiles and it looks like a test, but JUnit won't run it as a test because the annotation is missing.¹

Assuming that I notice the missing annotation (which isn't necessarily easy to notice), why is it missing? Was it an accident? Did somebody intentionally disable this test temporarily and forgot about it and checked it in? Who knows?

6.1.2 What to do about it?

When you find a test (or parts of a test) commented out, you're essentially looking at dead code. Perhaps it used to have a meaning and a purpose, but today that purpose either no longer exists or is lost, you can't figure out why that commented-out code is there.

The good news is that at this point there's no question as to what you should do. First, you'd ask around if someone else knows what's going on with this test.² If nobody seems to recollect, it really is straightforward:

- 1 Try to understand and validate its purpose. If you figure it out, uncomment the test and refactor it so it communicates its intent better.
- 2 Otherwise, delete it.

That's right. Delete it. That commented-out block of code has no right to stay in your code base for one minute longer. Test code that's never executed is merely noise that doesn't give you any more information about the correctness of your implementation or guide you in its design. It's like trying to make sense of a crazy person engaging in a bad monologue. I don't blame you for first checking whether you can bring that piece of code back from version control later on if need be—but it needs to go!

In the case of listing 6.1, I did eventually manage to figure out the test's purpose. I had to uncomment the test and run it to see whether it passed. When I ran the test, it passed. I still had no idea as to why it was commented out. Then I tried to run it on a Windows computer instead of my Mac and the test failed! It turned out that there was a slight difference in how these different platforms handled time.

It took a long while and I finally ended up determining that I'd probably been working on this test, leaving it commented out and checked it into version control due to switching to some other task, and then forgot all about that test.

¹ What if the test annotation and method signature were intact and only the body of the test were commented out? Your tools would run it and report it as passing even though you didn't check anything. We'll return to this problem in section 6.4.

² You *could* find out by interrogating your version control system. I'd probably not bother.

That case was one where the test actually had a purpose. Sometimes, that purpose is long gone or buried so deep behind the opaque face of not-so-expressive test code that it's better to delete it and move on.

6.1.3 Summary

Code that's commented out is never run. It's dead code. That kind of code tends to have a purpose when it's originally written but the value of commented-out code perishes quickly—especially if you have version control (and you should). Soon the value it had turns into a productivity hit as you bang your head against a brick wall trying to understand and remember why that test was commented out and whether it should or shouldn't exist.

In these situations you need to be pragmatic. If you can't figure out the commented-out test's purpose soon, it's likely that you never will. If that seems to be the case, you're better off deleting the test. If we do manage to decipher its *raison d'être*, you should make sure that the intent comes through better from now on.

In summary, commented-out tests have such a bad smell because they baffle the programmer who's left wondering *why* that test has been commented out. Our next test smell also deals with comments and the missing “why” but from a totally different angle.

6.2 Misleading comments

Though commented-out code tends to be useless and confusing at best, even actual comments—the ones intended to communicate code's purpose—can smell bad. One particularly nasty smell is a *misleading comment*.

Misleading comments are arbitrary in that they may suggest something that's not necessarily true.³ Sometimes they're misleading and the poor programmer reading the comment and relying on that misinformation goes astray.

Let's look at a test where such misleading comments play their tricks on the unsuspecting programmer.

6.2.1 Example

The example in the next listing comes from a debt collection agency's accounting system. It's a test for an Account being (or not being) in good standing, depending on whether it has unpaid debts past the due date. Take a few seconds to read through the listing and to understand what the test is doing.

Listing 6.2 Misleading comments trick the unsuspecting programmer

```
@Test
public void pastDueDateDebtFlagsAccountNotInGoodStanding() {
    // create a basic account
```

³ All comments are like that; they're not executed with the code they're commenting on so they can easily go stale.

```

Customer customer = new Customer();
DelinquencyPlan delinquencyPlan = DelinquencyPlan.MONTHLY;
Account account = new CorporateAccount(customer, delinquencyPlan);

// register a debt that has a due date in the future
Money amount = new Money("EUR", 1000);
account.add(new Liability(customer, amount, Time.fromNow(1, DAYS)));

// account should still be in good standing
assertTrue(account.inGoodStanding());

// fast-forward past the due date
Time.moveForward(1, DAYS);

// account shouldn't be in good standing anymore
assertFalse(account.inGoodStanding());
}

```

So what's the test doing? It clearly consists of several steps, as indicated by the mere presence of those one-line comments and the empty lines used to separate those blocks of code. In the first block we create a basic account, then we register a debt on the account, and then we verify that the account goes from good standing to not good standing when time passes the due date.

Simple? Yes. Except that two of those comments were misleading and probably gave the wrong impression of what the test is actually doing.

First of all, the first block of code is supposedly creating a “basic” account, but the code actually creates a `CorporateAccount` with a specific delinquency plan. That's not what I'd consider “basic.” Second, a later comment suggests we simulate time passing “past the due date” even though we're actually fast-forwarding time to the due date itself. *On* the due date is generally not the same as *past* the due date. Did you notice that detail?

These kinds of discrepancies between what we pick up from reading the comments and what the code actually does are misleading. Sometimes we get by despite those discrepancies; sometimes we end up spending an extra 15 minutes debugging because we overlooked the discrepancy and developed a misinformed understanding of what the code does.

So how do we treat this ailment?

6.2.2 What to do about it?

In section 6.1 I suggested that a commented-out test should either be uncommented and refactored to read better, or deleted. Deleting a comment is almost always worth considering, as in this case of a misleading comment. That comment is doing more harm than good, and it needs to go. But it was likely written for a purpose, so the question is what to replace it with?

If the misleading comment seems to be there to explain what the code block is doing, that smell calls for more readable code. Most often I'd recommend one of these two solutions:

- 1 Replace the comment with better names for variables and methods.
- 2 Extract a method from that commented block of code and name it well.⁴

The fundamental issue with such a comment—one that describes what code is doing—is that it’s really the *code’s* responsibility to communicate its intent. If you feel like you need a comment to explain what the code does, you haven’t refactored the code enough. Maybe you haven’t named the variables well enough. Maybe the block of code should be encapsulated behind a private method that has a descriptive name.

What makes for a good comment?

In short, a good code comment explains the *why*, not the *what*.

Whenever you encounter a comment that explains what code does, that’s a code smell. The code should be readable enough that such a comment is unnecessary. There are a few situations where a comment is truly necessary. Those situations tend to be where the comment explains the *rationale* for a given block of code.

For example, whereas a comment explaining what a complicated `for` loop is doing tends to be a smell, a similar `for` loop might have a good and valid comment explaining that the loop is so ugly because it’s a performance-critical piece of code and you haven’t found a way to improve its readability without sacrificing the performance.

Whenever you find yourself writing a comment, ask yourself whether you’re describing the “what” or the “why.” Then think again whether you should write that comment or refactor the code you’re commenting.

Sometimes programmers add comments inside their methods to visually separate blocks of code from each other. The concern is valid, as the ultimate purpose is to help make the code easier to read. The means are suspect.

In such situations I usually suggest replacing the comment with an empty line—at least that empty line won’t go stale like comments sometimes do. A colleague of mine used to delete all empty lines from methods, too, because they were a code smell. A method was too big if you felt you needed to separate steps with whitespace. Though that may sound extreme, he has a point. That method is likely to be too big and you should seriously consider refactoring it, possibly by extracting a method or two.

6.2.3 Summary

There are good kinds of comments and bad kinds of comments. The latter group clearly outnumber the former. That’s why test-infected programmers should be wary when encountering (or writing) those seemingly helpful comments amidst their test code.

We just explored a particularly nasty kind of a comment: the misleading comment. The main problem with misleading comments is that they’re unreliable and can’t be

⁴ Surprisingly often the new method’s name turns out to be almost identical to the comment it replaced.

trusted. Yet, as we scan through our source code, we tend to read those comments and, being the optimists we are, believe what the comment is saying without checking that assumption against the actual code.

It's not that we'd intentionally misguide ourselves and our colleagues. Most often a comment is born valid and correct but goes stale as time passes and code changes, out of sync with the comment that's now arbitrary and confusing. It's become a misleading comment.

We explored common solutions to dealing with a misleading comment. It all boils down to getting rid of it. As we delete the comment, we also refactor the code so that it's readable without a comment. We may do this by naming variables better or we may extract blocks of code into private methods, each with a descriptive name.

The good kind of comments are those which explain the rationale for code being like it is. That's something we might not be able to express with the constructs of our programming language. Everything else is free game to be deleted and replaced through refactoring.

In the end, test code that gets run tells us more than a comment that's not executed. Our next test smell also lacks informational value. Or how useful would you consider a test that is run but can't fail?

6.3 Never-failing tests

Never-failing tests are like Chuck Norris—they never fail—and that's a bad thing. A test that can't fail has no value, as it will never alert you to a mishap. A test that can never fail is probably worse than not having that test, as it creates a false sense of security.

6.3.1 Example

Perhaps the single most common context where never-failing tests are found is in tests that check for an expected exception to be thrown.⁵ Here is an example of such a test.

Listing 6.3 Test that never fails

```
@Test
public void includeForMissingResourceFails() {
    try {
        new Environment().include("somethingthatdoesnotexist");
    } catch (IOException e) {
        assertThat(e.getMessage(),
            contains("somethingthatdoesnotexist"));
    }
}
```

⁵ Another common type of a never-failing test is a test without assertions.

The result of the test in the previous listing is this:

- 1 If the code works as it should and an exception is thrown, the exception is caught by the catch block and the test passes.
- 2 If the code *doesn't* work as it should and an exception isn't thrown, the method returns, the test passes, and we're oblivious to there being any problem with our code.

6.3.2 What to do about it?

Whenever you test for an exception to be thrown, make sure you don't forget to `fail()` when that exception isn't thrown. This listing points out the correct solution.

Listing 6.4 Adding the missing `fail()` call makes our test useful

```
@Test
public void includeForMissingResourceFails() {
    try {
        new Environment().include("somethingthatdoesnotexist");
        fail();
    } catch (IOException e) {
        assertThat(e.getMessage(),
            contains("somethingthatdoesnotexist"));
    }
}
```

❶ Fail test unless exception is thrown

The simple addition of a call to JUnit's `fail()` method ❶ makes the test useful. Now the test will fail unless that expected exception is thrown.

One of the features introduced in JUnit 4 was the `expected` attribute for the `@Test` annotation. The presence of this attribute declares to JUnit that you expect this test method to throw the specified type of exception and that the test should fail unless such an exception is thrown. Essentially, we get rid of the whole try-catch block and the `fail()` call that's so easy to forget. This listing shows our example from listing 6.4 rewritten to use the annotation-based approach.

Listing 6.5 Declaring an anticipated exception with the `@Test` annotation

```
@Test(expected = IOException.class)
public void includingMissingResourceFails() {
    new Environment().include("somethingthatdoesnotexist");
}
```

Much shorter, much easier to parse, and much less prone to human error and forgetfulness. The downside of this approach is also visible; since we don't have access to the actual exception object being thrown, we can't make any further assertions against that exception. For example, in listing 6.4 we checked that the exception message includes the name of the missing resource that caused the exception.

Aside from "don't make mistakes," the best way to protect yourself from accidentally writing a never-failing test is to develop a habit of running the test so that you see it fail, possibly by temporarily altering the code it tests to intentionally trigger a failure.

6.3.3 Summary

Tests are supposed to fail when they should. That's a truism but an important one when it comes to never-failing tests that give us a false sense of security. We don't want our tests leading us down the wrong path when we're trying to diagnose a problem.

Never-failing tests are mostly seen around tests that check for exceptions being thrown or not thrown, as it's easy to screw up with those `try-catch` blocks. We might forget to `fail()` when an exception isn't thrown, or we might accidentally swallow an exception that should be thrown over to JUnit.

If you're only interested in making sure that an exception is thrown or that it's of a given type, the `expected` attribute for the `@Test` annotation is your friend. If you want to make more specific checks against the thrown exception, you're left with no other option but to wrap your code into a `try-catch` and be careful about doing it right.

6.4 Shallow promises

We recently had elections where I live. Leading up to the vote, you could see politicians debating and interviewing on pretty much all of the national TV channels. You've probably heard enough politician jokes to have formed the same perception that politicians tend to make shallow promises that they don't end up keeping—always for good reasons, but still.

Programmers can be like that, too. Every now and then when you're browsing test code, your senses start hurting like you were Spider-man because you're looking at a test that promises much more than it delivers. This is a problem for the obvious reason of programmers being misled by an untrustworthy test.

There's more than one way to perpetrate this test smell, so we'll take a glimpse at a number of examples.

6.4.1 Example(s)

The underlying theme of *shallow promises* is that the test does much less than what it says it does—or does nothing at all. There are three general categories of how this might happen:

- Test doesn't *do* anything.
- Test doesn't actually *test* anything.
- Test isn't as thorough as its name suggests.

The first and arguably most blatant violation of the programmer's trust is shown in the next listing.

Listing 6.6 A test that doesn't do anything is next to useless

```
@Test
public void filteringObjects() throws Exception {
    //Array array = new Array("joe", "jane", "john");
    //Array filtered = array.filter(new Predicate<String>() {
    //    public boolean evaluate(String candidate) {
```

```
//    return candidate.length() == 4;
// }
//});
//assertEquals(new Array("jane", "john"), filtered);
}
```

What we can see in the listing is a test with its body commented out. Effectively, this test method is a no-op and should be ignored. In practice, our test frameworks will run this test and report its success, potentially misleading a programmer looking at the list of tests in his IDE flagged as passing to think that the functionality described by the test's name is already in there. In other words, it's like a *commented-out test* but even worse!

Furthermore, when we scan this commented-out test we can't ignore it. There's a reason why that code was written and subsequently commented out—but we have no idea as to why. Should this test be passing? Did somebody forget to uncomment it after an experiment of some kind? We just don't know.

Though it's not that common to bump into tests that have been commented out entirely or that don't do anything, our second archetypal example of a shallow promise is much more common. The following listing represents what such tests often look like.

Listing 6.7 A test that doesn't check anything is next to useless

```
@Test
public void cloneRetainsEssentialDetails() throws Exception {
    Document model = ModelFactory.createDocument();
    model.schedule(new Transaction("Test", date("2010-01-20"), 1));
    model = model.clone();
    Transaction tx = model.getScheduledTransactions().get(0);
}
```

On the surface this test looks good. It's short, fairly balanced, and has a clear, understandable name—clone retains essential details. But on a closer look it turns out that the test doesn't actually *check* whether clone retains essential details. There's no `assertTrue`, no `assertEquals`, `assertThat`—no assertions whatsoever. And that means that this test could be passing no matter what the clone behavior actually is. More specifically, the only way this test would notice that the cloning doesn't work as it should would be if an exception is thrown while executing this code.

Some people call such tests *happy path tests* because they tend to never fail and always suggest everything's all right.⁶

Finally, our third example is depicted in the next listing. It represents perhaps the most common pathology of a shallow promise.

⁶ And some of those tests came to be because the test failed so somebody deleted the assertion. I'm not kidding.

Listing 6.8 A test that does less than it suggests

```
@Test
public void zipBetweenTwoArraysProducesAHash() throws Exception {
    Array keys = new Array("a", "b", "c");
    Array values = new Array(1, 2, 3);
    Array zipped = keys.zip(values);
    assertNotNull("We have a hash back", zipped.flatten());
}
```

Reading through the listing, do you see the shallow promise? The name of this test suggests that a zip operation between two array objects would produce a hash. But the test only checks that the zip method returns an array that can be flattened, and that the flattened object isn't null. That's quite different from zip returning a hash, don't you think?

All of these examples of tests that promise one thing and do another can cause grey hair in the poor programmer who's misled into thinking that the described behavior exists and works correctly while in reality, it doesn't. So let's talk about what we should do to avoid that problem.

6.4.2 What to do about it?

Let's start by looking at the situation in listing 6.6 where we have a test with its body commented out.

First of all, why did we comment it out? Maybe we started writing it and realized halfway through that it's too big a step, commented it out, wrote a different test instead, and forgot to come back to flesh out the commented-out test. Maybe we were experimenting with something that broke the test and decided to comment it out instead of deleting it, because we might soon want to restore it back to what it was.

Whatever the reason was for commenting out the test's body, there would've been a better option. The rule of thumb is to *delete code, don't comment it out*. You have version control, right? Plus, modern IDEs give us a local history with which we can undo changes that never reached the version control repository.

An empty test is much better than a commented-out one also because it's a clear placeholder for a test that hasn't been written—for behavior that's currently missing, broken, or unverified.

Maintaining a test list in code

Many programmers maintain a *test list*—a list of tests that should be written (but haven't been, yet). Some programmers prefer the low-tech approach of scribbling notes on a piece of paper beside the keyboard. Others prefer to keep their test lists next to the test code. Though I personally prefer adding a `//TODO` comment into a test class to identify a missing test, others prefer creating empty test method placeholders.

(continued)

Using such empty placeholder methods as the test list can lead to the shallow promise problem when the programmer moves back and forth in the code base, accidentally leaving behind some of those placeholders. With that in mind, I suggest that you mark such placeholders with the JUnit `@Ignore` annotation to make it more clear that those tests haven't been implemented properly and that the behavior described doesn't exist yet.

Even better would be to pick up the good old pen and paper. Aside from a clear separation of what's been implemented and what's still to do, pen and paper don't constrain you to alphabetic notes—you can sketch diagrams, when that's the most natural way of describing what you're thinking.

Our second archetype of shallow promises was shown in listing 6.7: a test without assertions. The solution is simple: make sure you assert something. Though you might dig into your favorite static analysis tools to see if there's a way to check for this automatically, there's also a simple technique that helps in putting in those assertions: start with the assert.

When you start writing the test by first typing in the assertion you eventually want to make, you're making it difficult to accidentally remove that assert from the resulting test. Furthermore, starting with the assertion helps you focus on the essence of the test. What specifically is the behavior we want to check in this test?

Starting with the assertion can also help you steer clear of our third example of a shallow promise, shown in listing 6.8—a test that checks something less thoroughly than its name suggests. After all, when all you have is the name of the test and the assertion statement, it's almost impossible to *not* see if they're in conflict.

Another trick that you might consider is to leave the test's name initially empty or call it, for example, `TODO()`, until you've fully sketched out the test. It's easier to name a test once you've figured out exactly what behavior it's going to check.

The key here, as with all of these tricks, is to make it as easy as possible for the programmer (you) to see and notice when a test isn't doing what it claims. With that in mind, merely keeping tests small and focused is already a big help.

6.4.3 Summary

In this section we talked about tests that don't deliver on their promises. We identified three archetypes of such tests:

- Tests that don't *do* anything
- Tests that don't actually *test* anything
- Tests that aren't as thorough as their names suggest

In all of these cases we're putting ourselves and our colleagues at risk of wasting precious time by making false assumptions, making flawed decisions based on those assumptions, and looking at the wrong places when we find out that things aren't as

they should be. Time is one of the true constraints we have in our lives—don't waste it with such trivial mistakes as a test making a shallow promise.

With that in mind, you should be sure to delete code instead of commenting it out, making sure that you don't leave test code around without assertions, and you should be extra careful in keeping your tests' names in sync with what they're actually checking. Starting with the assertion, naming the test *after* it's written, and keeping tests small can all be of great help. In the end, such tiny bits of discipline can save many hours in a situation where every minute counts.

Speaking of discipline, our next test smell stinks of an interesting type of lapse in discipline.

6.5 Lowered expectations

Programmers tend to be detail-oriented. We also tend to be lazy and often look for the easiest way of doing things. That's good—a lot of the time the easiest thing is what we should do—but sometimes we can overdo it and we end up shooting ourselves in the foot along the way.

This test smell is called *lowered expectations* because it's a case of seeing an easy way out and taking it at the cost of lowered standard of certainty and precision. Taking a shortcut like that can often help you move faster, giving enough certainty that the piece of code you just added does in fact make a difference in the program's output or behavior. In the long run, such tests can maintain a false sense of security precisely because of their imprecision.

Let's look at an example to illustrate this risk.

6.5.1 Example

The example shown next suffers from split logic, a test smell we've covered already, and lowered expectations. See if you can spot them.

Listing 6.9 Calculating cyclomatic complexities for source files

```
public class ComplexityCalculatorTest {
    private ComplexityCalculator complexity;

    ...

    @Test
    public void complexityIsZeroForNonExistentFiles() {
        assertEquals(0.0, complexity.of(new Source("NoSuchFile.java")));
    }

    @Test
    public void complexityForSourceFile() {
        double sample1 = complexity.of(new Source("test/Sample1.java"));
        double sample2 = complexity.of(new Source("test/Sample2.java"));
        assertThat(sample1, is(greaterThan(0.0)));
        assertThat(sample2, is(greaterThan(0.0)));
        assertTrue(sample1 != sample2);
    }
}
```


What we have in the listing is a test class that defines the expected behavior of a `ComplexityCalculator` calculating the cyclomatic complexity of a given source file.⁷

First of all, we clearly have a case of split logic, as we established already. The test is dealing with magic numbers because the reason for that number being 0.0 or greater than 0.0 is hidden in the two source files referenced only by path: `test/Sample1.java` and `test/Sample2.java`.

We already know what to do with split logic, so let's not pay too much attention to that; we mostly care about the lowered expectations right now.

Take a look at `complexityForSourceFile()` in listing 6.9. Note how we make three assertions to verify that the complexity was calculated correctly. First, we verify that for both source files the result is greater than zero, then we verify that both source files received a different result. Assuming that this test passes and that both source files should indeed receive a different result, we can be certain that there's at least *some* logic behind the calculation.

That funny way to “triangulate” our way into a somewhat vague assertion of correctness manifests the lowered expectations. We're essentially saying that it's okay for the actual calculation to be whatever as long as it's not zero and not the same for two different source files. In other words, that particular test is extremely robust for change; so robust that it won't break even when it should.

6.5.2 What to do about it?

The obvious solution for lowered expectations is to raise the bar higher and make the test more specific about what you expect. In the case of listing 6.9 you might be explicit about the exact complexity that the calculation should yield for the two source files. That simple change would turn our five-liner into a two-liner, shown next.

Listing 6.10 Being more explicit simplifies our test

```
@Test
public void complexityForSourceFile() {
    assertEquals(2, complexity.of(new Source("test/Sample1.java")));
    assertEquals(5, complexity.of(new Source("test/Sample2.java")));
}
```

Much more specific, and much simpler. Listing 6.10 still suffers from the split logic and magic number problems, but it's no longer setting such low expectations; this test is a lot more likely to fail if we mistakenly break the algorithm for calculating complexity.⁸

It's worth noting that being totally exact is not a virtue as such. For example, the pixel perfection test smell described in chapter 5 points out the potential downfalls of

⁷ Cyclomatic complexity is a reasonably good indicator for (the lack of) code quality. A high number likely means that the method in question is too big. Though it's not common to run such static analysis for test code, a test method with high cyclomatic complexity would suggest *incidental details* or a *conditional test*.

⁸ What would you do about the split logic in listing 6.10, by the way?

a test being too specific. You should look for an appropriate balance and level of abstraction to express your tests.

6.5.3 Summary

Tests are supposed to fail if the behavior they’re specifying is broken. The lowered expectations test smell leads to tests that are overly robust—they don’t fail when they should.

The essence of lowered expectations is an assertion that’s too vague to properly describe the expected behavior. When the assertions are too vague, you’re contributing to a false sense of security; the functionality might be broken and yet your loosely checked vague assertions might pass with flying colors.

The obvious solution to lowered expectations is to raise the bar by making your assertions more specific and precise. By expressing your interests more directly, you’re making your intent clear, and avoiding the false sense of security because your tests’ assertions will actually fail when the implementation changes in undesirable ways.

With that said, there’s nothing fundamentally bad about leaving out details in your tests. In fact, being too specific with your assertions could mean that you’re risking the problems of pixel perfection.

Next up, we have a couple of test smells that relate to conditional behavior. The first—like so many things in life—is born from good intentions but smells like trouble.

6.6 Platform prejudice

We’ve come a long way since the word computer was synonymous with one specific product that everybody used. Programmers increasingly need to consider multiple alternative platforms and operating systems for their software. Though Java was once pitched at us with the tag line “write once, run anywhere,” we’ve all found ourselves implementing platform-specific functionality because not everything is abstracted away.

With our software products concerned with multiple platforms, our tests must do the same. The *platform prejudice* test smell could be described as a failure to treat all platforms equal. It’s more intricate than that, but perhaps an example will make it crystal clear.

6.6.1 Example

The following example checks that a Configuration class properly identifies the default downloads directory for the underlying platform.

Listing 6.11 Checking platform-specific download folders

```
@Test
public void knowsTheSystemsDownloadDirectory() throws Exception {
    String dir = Configuration.downloadDir().getAbsolutePath();
    Platform platform = Platform.current();
    if (platform.isMac()) {
        assertEquals(dir, matches("/Users/(.*)/Downloads"));
    } else if (platform.isWindows()) {
```

```

        assertThat(dir.toLowerCase(),
            matches("c:\\users\\(.*)\\downloads"));
    }
}

```

The first thing that stands out is the `if-else` hack for carrying out different assertions when running on different platforms. In other words, only one of those branches gets executed on any given platform.

This is problematic for a couple of reasons. First, what if we're running on Linux? The test in listing 6.11 doesn't indicate in any way that it'll silently do nothing when it's executed on a platform other than Mac OS X or Windows. Second, when running the test, we're only checking for the correct behavior on the platform that we happen to run on. That means the test will pass regardless of whether the behavior is broken for *other* platforms.

So what might we do about this problem?

6.6.2 What to do about it?

Listing 6.11 brings up a couple of different problems. First of all, we're hiding crucial information within the test that really should be made visible to the outside; namely, the fact that we have separate branches of checks that are performed for specific platforms but only when running on those specific platforms.

One way to make those branches visible is to split listing 6.11 into multiple tests, one per platform. This is shown next.

Listing 6.12 Different tests for different platforms

```

public class TestConfiguration {
    Platform platform;
    String downloadsDir;

    @Before
    public void setUp() {
        platform = Platform.current();
        downloadsDir = Configuration.downloadDir().getAbsolutePath();
    }

    @Test
    public void knowsTheSystemsDownloadDirectoryOnMacOsX()
        throws Exception {
        assertTrue(platform.isMac());
        assertEquals(downloadsDir,
            matches("/Users/(.*)/Downloads"));
    }

    @Test
    public void knowsTheSystemsDownloadDirectoryOnWindows()
        throws Exception {
        assertTrue(platform.isWindows());
        assertThat(downloadsDir.toLowerCase(),
            matches("c:\\users\\(.*)\\downloads"));
    }
}

```

1 Abort test
if on wrong
platform

Note how we've also added guard clauses that use the `org.junit.Assume#assumeTrue()` assumption API for aborting test execution ^① when not running on a specific platform. When such an assumption fails, JUnit won't execute the remainder of the test. Unfortunately, JUnit marks such tests as passed even though we really don't know whether the functionality is correct on those platforms.⁹

Even with the platform-specific tests now visible to the outside, the second and more fundamental problem remains: only one of these tests will be executed. In order to run all three, one would need three different computers. That problem stems from the design of the code under test and our test smell highlights the need for a refactoring.

We should strive to refactor the code base so that the `Platform` can be substituted on a test-by-test basis, allowing us to run all of the tests in listing 6.12 on any platform. After such refactoring our tests might look like the following.

Listing 6.13 Refactoring the design gets rid of the platform prejudice

```
@Test
public void knowsTheSystemsDownloadDirectoryOnMacOsX()
    throws Exception {
    String downloadsDir = new MacOSX().downloadDir();
    assertEquals(downloadsDir,
        matches("/Users/(.*)/Downloads"));
}

@Test
public void knowsTheSystemsDownloadDirectoryOnWindows()
    throws Exception {
    String downloadsDir = new Windows().downloadDir();
    assertThat(downloadsDir.toLowerCase(),
        matches("c:\\users\\(.*)\\downloads"));
}
```

Note how we're directly instantiating the specific `Platform` subclass instead of going through the indirection of `Platform.current()` and then working around the problem of running conditional tests based on what that `Platform` turns out to be. The platform-detecting behavior of `Platform.current()` is something we need to test somewhere, but those complications don't belong to tests that deal with the behavior of the platform implementations themselves.

On line feeds and carriage returns

If you've worked on systems that need to run on both Windows and UNIX systems, you've probably bumped into one notable difference between these platforms, aside from the different file paths—the newline characters.

⁹ This issue has been on the table with JUnit developers for a while. I remain hopeful that it'll be fixed soon.

(continued)

Traditionally, applications written for Windows have expected all kinds of text files to have their lines separated by two characters (carriage return and line feed—`\r\n`) whereas UNIX-like systems have standardized on just one (the line feed character, `\n`).

Depending on what kind of systems you're working on, it may or may not be important that you write your tests using the platform-specific newline characters instead of hardcoding a `\n`. Just using `\n` will *usually* turn out okay. Usually.

Some programmers choose to use a constant that adapts to the platform the code is executed on, just to be sure, like so: `static final String NL = File.pathSeparatorChar == ':' ? "\n" : "\r\n";`

Note that doing this in your test code also implies that any data files you're processing need to be either generated from the test code or adapted to match the platform-specific newline characters.

6.6.3 Summary

Platform prejudice is a test smell that tends to creep in with the demand to support multiple platforms such as Linux, Mac, and a bunch of Windows variants. When your tests start carrying out different execution paths and different assertions based on which platform you're running on, you're asking for trouble.

Whenever you see those multiple platforms mentioned in your tests, you should make them visible to the outside by pulling them into separate tests. That way at least you're aware of the existence of platform-specific logic.

You still have conditional test execution, as splitting the tests doesn't yet fix the actual problem. Even after splitting the tests and using JUnit's nice `Assume` API instead of brutal `if-else` hacks, you might still be looking at the list of tests in your IDE, falsely thinking that all of that behavior spelled out by those nicely named tests actually works on all platforms you support, because they all seem to be passing even though some of them weren't run.

The real solution can be found in refactoring the code base such that the test smell disappears. Wherever you have platform prejudice in your tests, you should look for ways to streamline the tests so that each test can instantiate and use the specific platform it wants.

In the end, the root cause for platform prejudice can be traced to a tiny fraction of platform detection logic somewhere. That logic should be isolated with a set of tests that are explicit about running on just one particular platform, isolating the problem and making it much more manageable.

Platform prejudice is a special case of conditional tests: executing (or not executing) a test based on a condition that's hidden away inside a test. Let's take a moment to explore such conditional tests in a more generic context.

6.7 Conditional tests

Modern programming languages are amazing. Over the years we've seen languages evolve and APIs expand. Among other things, we've seen language constructs make their way from libraries into the core language. But more power isn't good when it's used wrongly, and the *conditional test* smell is a case in point.

A conditional test is one that's hiding a secret conditional within a test method, making the test logic different from what its name would suggest. Our previous test smell, platform prejudice, featured an example of one specific type of conditional test, but let's make the case for conditional tests in our tests being bad in general, and we'll do that with an example.

6.7.1 Example

The next listing presents a test that checks the behavior of a `Process` object responsible for executing system commands. This particular test has been written to check that arguments are being passed correctly.

Listing 6.14 Checking that a system command was executed correctly

```
@Test
public void multipleArgumentsAreSentToShell() throws Exception {
    File dir = createTempDirWithChild("hello.txt");

    String[] cmd = new String[] { "ls", "-l", dir.getAbsolutePath() };
    Process process = new Process(cmd).runAndWait();

    if (process.exitCode() == 0) {
        assertEquals("hello.txt", process.output().trim());
    }
}
```

Run directory listing ②

① Create temp files

③ Check listing

Let's see what we're doing here. First up, the test creates a temporary directory with a file ① called `hello.txt` and then runs a directory listing ② in that temporary directory. Next, we have the hidden conditional.¹⁰ If the process exited clean (exit code is zero), we assert that the directory listing ③ contains exactly what we expect to see with the specific arguments that were given.

The problem with this conditional branch is that it might evaluate to false and, subsequently, the assertion wouldn't be made and the test would silently pass. For example, we might've broken the `Process#runAndWait()` method in such a way that any system command will fail, yielding a non-zero exit code. Our test in listing 6.14 would happily pass despite nothing actually working.

Somewhat undesirable, no?

¹⁰ It's hiding in plain sight, just invisible from looking at the outline of the class and the names of test methods.

6.7.2 What to do about it?

Again, a test should fail when the behavior it is testing breaks. If something goes wrong, you should loudly announce it, not swallow it in silence.

With that in mind, when stumbling on a conditional in our tests, you should make sure that your test has a chance of failing in every branch of that conditional. In listing 6.14 we could do that by adding an assertion for the process having exited successfully.¹¹ This is shown next.

Listing 6.15 Replacing the conditional with an assertion

```
@Test
public void multipleArgumentsAreSentToShell() throws Exception {
    File dir = createTempDirWithChild("hello.txt");

    String[] cmd = new String[] { "ls", "-l", dir.getAbsolutePath() };
    Process process = new Process(cmd).runAndWait();

    assertEquals(0, process.exitCode());
    assertEquals("hello.txt", process.output().trim());
}
```

With the addition of that assertion, the test will fail unless the process exits properly, solving the problems previously caused by the conditional statement.

In addition to ensuring that each branch of the conditional will trigger a failure when appropriate, you should consider making each branch a test of its own. After all, the conditional wouldn't have made sense if it always evaluated to the same result, so clearly there are multiple potential execution paths the software should expect. In our simple example in listings 6.14 and 6.15, the alternative execution path would be one where the command execution fails. Hence, you might add a new test that checks the exit code being non-zero:

```
@Test
public void returnsNonZeroExitCodeForFailedCommands() {
    String[] cmd = new String[] { "this", "will", "fail" };
    Process process = new Process(cmd).runAndWait();
    assertEquals(1, process.exitCode());
}
```

With this addition, all branches of the conditional of listing 6.14 are fully covered with tests that don't suffer from the same problem. We have the original test for making sure that arguments get passed on to the underlying shell, and we have the test seeing that the exit code signals failure when the system command fails.

¹¹ Semantically, a JUnit *assumption* would be a better fit but, at the time of writing this, the Assume API is still broken and failed assumptions will silently abort the test and mark it as passed, which is exactly what we *don't* want.

6.7.3 Summary

Conditional statements in tests are bad because they tend to be misleading. They might be passing when they shouldn't and even when they pass for the right reasons, we don't know unless we fire up the debugger and see whether the execution path did in fact go through our conditionally executed branch.

As a rule of thumb, all branches in a test method should have a chance to fail. Furthermore, since each of those branches represents a different scenario and different behavior, they really should be split into separate tests.

The conditional structure such as an `if` block represents some kind of uncertainty about how the code's execution will turn out. If that's not the case and there's no uncertainty then the best option might be to delete the conditional and move on.¹² If there's some uncertainty, it tends to be a good idea to replace the conditional with an assertion that confirms and makes explicit the assumptions the test operates under.

6.8 Summary

This chapter walked through a number of test smells dealing with untrustworthy and unreliable tests. Roughly half of the smells in this chapter revolved around tests that won't tell you when something breaks—tests that don't fail when they should. The other half deals with tests that mislead in other ways.

We began with the potential pitfalls of abused comments in the form of commented-out tests and misleading comments. You saw the problems with never-failing tests and tests that make shallow promises. Lowered expectations are a problem because you might not know when things are broken, and the same goes with platform prejudice and other conditional tests.

All of these test smells share the common symptom of misleading you into thinking that everything is fine when it's not, whether that's by claiming that a test passes when it's not even run or by claiming to test some behavior when it's not checking anything whatsoever.

In a way, the test smells in this chapter have the combined downsides of the smells described in chapters 4 and 5: they reduce readability and make maintenance more difficult.

This chapter concludes part 2 and our catalog of test smells. In part 3 we'll shift our attention to totally different topics, such as testable designs and pragmatic approaches to writing and running tests with the latest and greatest that technology has to offer.

¹² There aren't many good reasons to use an `if` in a test method, and even for the good ones there's always a better alternative.

Part 3

Diversions

For a moment, the working name of this third part of the book was “Advanced.” However, reconsidering the use of such a strong word loaded with meaning, we concluded that it would be misleading to call it that. Instead, we decided to call it “Diversions” because that’s what it is—a set of diversions that are interesting and useful to the advanced practitioner but aren’t exactly “required reading” for becoming one.

The topics in this part are about taking what you have and turning the knobs farther toward eleven. In chapter 7 we’ll open the can of worms labeled “design” and try to paint a picture of what constitutes (or doesn’t constitute) as *testable* design. After all, our ability to write tests for our code goes hand in hand with our ability to design our code—period.

In chapter 8 we’ll turn the boat 180 degrees and explore the symbiosis of JVM languages and how it would look if we tested our Java code with tests written in another programming language. This may or may not be a good idea for the project you’re working on today. However, there’s no denying that alternative JVM languages are here to stay, and it could very well be that some time later you’ll thank yourself for writing some of those tests with one. Who knows!

This part concludes with the more down-to-earth topic of sluggish builds. It’s a common phenomenon to see build times grow longer and longer over the lifetime of a project until your feedback loops are next to unbearable. That’s why we’ve dedicated chapter 9 to speeding up our builds, both by tweaking our test code (which is often a major contributor to the build time) and by refining the way our build is run.

Ready to roll?

Testable design



In this chapter

- What is testable design?
- What is untestable design?
- How to create testable designs

In the preface of *Implementation Patterns* (Addison Wesley Professional, 2007), Kent Beck compares programming to an American TV show called *Jeopardy*. In the show the host provides answers and the contestants' job is to guess which question that answer was for. "A short section at the end of a book." "What is an epilogue?" "Correct."

Kent makes the analogy to programming and points out that Java provides answers in the form of its language constructs, and the programmer's job is to figure out what the questions are and which problems are solved by which language construct. He offers the following example: if the answer is "Declare a field as a set," the question might be, "How can I tell other programmers that a collection contains no duplicates?"

The same happens with design. Throughout our education and especially the first years of our professional careers, we're taught solutions. Our senior colleagues show us "the way things are done around here," and we pick up coding conventions from the code we work with, and sometimes we pick up stuff from books like this.

But it's not enough to know solutions. We also need to learn to identify the problems they solve. That's why this book includes a catalog of test smells.

This chapter aims to identify the common testability problems that stem from design decisions, such as making a method `static`, and the use of certain language constructs like `final`. We'll start with a discussion of what we're looking for when we say we want "testable design," a modular design that likely abides by certain design principles and avoids certain implementation details that make it harder to write tests. From there we'll move on to detailing the specific testability issues we try to avoid and we'll conclude the chapter with simple guidelines for designing testable code.

7.1 What's testable design?

The fundamental value proposition of testable design is being better able to test code. Quoting Roy Osherove, the author of *The Art of Unit Testing with Examples in .NET* (Manning Publications, 2009), "a given piece of code should be easy and quick to write a unit test against." More specifically, testable design makes it easy to instantiate classes, substitute implementations, simulate different scenarios, and invoke particular execution paths from our test code.

Testability isn't an absolute yes-or-no property of a design. As Scott Bellware put it, "Testability is not a term that describes *whether* software can be tested. It refers to software that is *easily* tested."¹ It should be a breeze for a programmer to set up scenarios in unit tests. The less testable the design, the more burdensome it is for the programmer to write those tests. The big question is, what are testable designs made of—how do we end up with such designs?

Though you'll often hear quips about software development being a young profession, we have decades of experience to draw from and that experience has taught us a lot about constructing software in a variety of programming paradigms. One of the things we've learned is the virtues of modular design.

7.1.1 Modular design

What makes a design modular is its nature of being composed of separate modules, each serving a particular purpose in the design. By breaking down a program's overall functionality into distinct responsibilities and assigning those responsibilities to separate components, we end up with a design that has plenty of flexibility.

That flexibility builds on the various seams we've introduced by way of composing the overall design from discrete modules, with each individual module containing everything necessary to fulfill its responsibility. This style of programming aims at having as few dependencies *between* modules as possible, as illustrated in figure 7.1.

Constructing software from small components helps groups of people collaborate on larger products because functional changes to the product tend to be more isolated to a specific part of the code base. This follows from the characteristic of

¹ Scott Bellware, "Good Design is Easily-Learned," Jan. 12, 2009, <http://mng.bz/IAMK>.

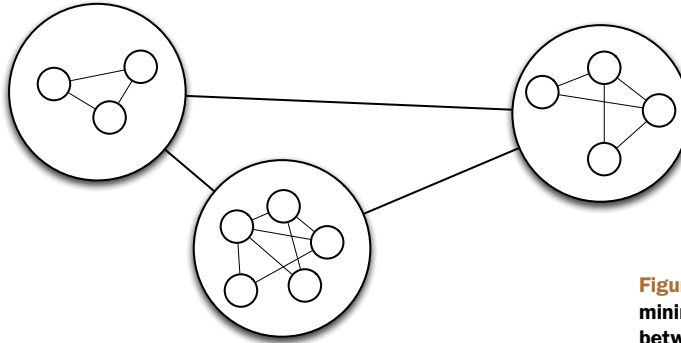


Figure 7.1 Modular designs minimize the dependencies between modules

modular design, where the system is partitioned into discrete functional elements with their respective responsibilities over a specific function or capability.

This structure, encouraged by modular design, enables after-the-fact augmentation of a system, changing or adding new functionality by merely plugging in a new module, as long as the modules are sufficiently self-contained and loosely coupled, and functional concerns are separated within the code base. This also directly aids testability because the properties of modular design are the ones that make code testable.

Generations of programmers have put these ideals in practice and found them a valuable goal to strive for. But the concept of modular design is fairly abstract. Luckily, our elders have also crystallized some of their hard-earned experience into somewhat more concrete design principles for us to keep in mind.

7.1.2 **SOLID design principles**

Plenty of design principles are documented in literature. One of the more widely spread collections of design principles are the *SOLID* principles. *SOLID* is an acronym and stands for a set of five design principles documented by Robert C. Martin.

The nice thing about object-oriented design principles such as *SOLID* is that they mesh well with testability. Keep your code in line with design principles such as *SOLID* and you're that much more likely to end up with a modular design.

So let's see what the *SOLID* principles are and how they contribute to the testability of a design.

SINGLE RESPONSIBILITY PRINCIPLE

The Single Responsibility Principle (SRP) says, "There should never be more than one reason for a class to change." In other words, classes should be small, focused, and have a high cohesion. It goes further than that, as methods should also have just one reason to change. This is what Sandro Mancuso calls the *inside view*.²

Code that follows the SRP is easier to approach and understand, which in turn makes it easier to test because writing tests is essentially an act of specifying the

² See "SRP: Simplicity and Complexity," July 26, 2011, <http://mng.bz/08ks>.

expected behavior, expressing our understanding of the problem the code is intended to solve. Furthermore, taking the inside view, the less complexity a method has, the less complexity is required in a test to thoroughly test that code.

OPEN-CLOSED PRINCIPLE

The Open-Closed Principle (OCP) says that classes should be “open for extension but closed for modification.” Though that may sound abstract, all it means is that you want to be able to change what a class does without changing its source code; for instance, by swapping in an alternative strategy.

Classes that delegate specific responsibilities to other objects allow tests to substitute a test double when needed to simulate a specific scenario.

LISKOV SUBSTITUTION PRINCIPLE

The Liskov Substitution Principle (LSP) says, “Subclasses should be substitutable for their base classes.” In essence, code that uses an instance of class A should continue to function properly if passed an instance of class B, a subclass of A.

LSP is about class hierarchies that exist for the right reason, embodying a valid abstraction, and not merely as a vehicle of code reuse because it happens to be convenient. Though among the less crucial SOLID principles from a testability point of view, violating the LSP does imply a handicap for the test-infected programmer. Class hierarchies that follow the LSP contribute to testability by enabling the use of *contract tests*—tests written for an interface that can be executed against all implementations of that interface.

INTERFACE SEGREGATION PRINCIPLE

The Interface Segregation Principle (ISP) suggests that “Many client specific interfaces are better than one general purpose interface.” In short, you should keep interfaces small and focused.³

Small interfaces improve testability by making it easier to write and use test doubles. For instance, one test might want to stub collaborator A, fake collaborator B, and substitute a spy for collaborator C. With each collaborator having its own small interface, it’s straightforward to implement the test doubles.

DEPENDENCY INVERSION PRINCIPLE

The fifth SOLID principle is the Dependency Inversion Principle (DIP), which says code should “depend on abstractions, not on concretions.” Taken to one extreme, the DIP suggests that a class shouldn’t instantiate its own collaborators but rather have their interfaces passed in.

For writing tests, the ability to pass collaborators in rather than selectively override parts of the code under test is huge. Such *dependency injection* is a boon to testability because not only is substituting collaborators possible, but it’s also made effortless: the tests use the code just like it’s used in production.

It’s important for programmers to understand common design principles and to be able to envision concrete implementations and their implications. Over time, you

³ Sound familiar? I once overheard someone refer to the ISP as “SRP for interfaces.”

gather a sufficient collection of mental notes and learn to pattern-match them against the code you're looking at, being able to draw on that knowledge and experience.

7.1.3 Modular design in context

It's not that easy, because though we're pretty good at identifying suitable implementation alternatives for the problem we're solving, we should also be able to envision those solutions as part of the larger whole—from the perspective of the code that pulls together those smaller components. Tim Berners-Lee, the man who invented the World Wide Web, once wrote the following about the principles of design: “It is not only necessary to make sure your own system is designed to be made of modular parts. It is also necessary to realize that your own system, no matter how big and wonderful it seems now, should always be designed to be a part of another larger system.”⁴

In other words, modularity as such isn't good unless the design is fit for the way you want to use it and grow it. You're in good fortune—I have a trick up my sleeve that makes it easier to arrive at modular designs that respect common design principles and behave well in their larger context.

7.1.4 Test-driving toward modular design

The fastest way to absorb modular design is to start test-driving your code. The act of writing tests before the implementation they call for is essentially a way to ensure that you're taking the client's view on the code you're shaping. That means that you're that much more likely to end up with a design that's fit for purpose. Plus, there's no question of how testable the design is.

It's not just the test-first facet of TDD that spurs modular design. TDD practitioners also refactor their code frequently so they're constantly looking for too big methods to split, better abstractions to introduce, and duplications to remove. J.B. Rainsberger talks about how minimizing duplication and maximizing clarity leads to modular design in his article on “The Four Elements of Simple Design.”⁵

There are many techniques and ways to learn modular design at your disposal. It's only a matter of choosing whether you care about your code enough to adopt disciplined practices, such as test-driven development and merciless refactoring.

Now let's take a look at some of the obstacles that we often find in our way, standing between us and our testable design.

7.2 Testability issues

Most often, a programmer struggling to write a test is facing off against one of a few common show-stoppers. They mostly fall under two categories: restricted access, and inability to substitute certain parts of the implementation. Though there are many

⁴ Tim Berners-Lee, “Principles of Design,” last updated March 2, 2010, <http://www.w3.org/DesignIssues/Principles.html>.

⁵ J.B. Rainsberger, “The Four Elements of Simple Design,” 2009, <http://www.jbrains.ca/permalink/the-four-elements-of-simple-design>

ways for testability to be less than ideal, a lot of the time the first barrier to get over is as trivial as being unable to instantiate a class in your test.

7.2.1 *Can't instantiate a class*

One of the first things we as programmers tend to do when writing a test is to instantiate some objects. It could be an object you want to test that you can't instantiate, but most often the culprit is a collaborator you need to pass in to the object you're testing.

This mostly happens with third-party libraries that weren't designed with testability in mind, but sometimes you only have yourself to blame. Typically, you've been overly conservative with your visibility modifiers and now you're paying for it as the compiler points out your short-sightedness.

Another common cause for the inability to instantiate a class with a constructor is a static initialization block that assumes you're running in a full-blown production environment, leading to a totally unexpected exception blowing up in your face when you try to run your test. For instance, consider the following snippet of testability-destruction:

```
public class DocumentRepository {
    private static final String API_KEY = "d869db7fe62fb07c";
    private static String sessionToken;

    static {
        String serverHostName = System.getenv("ACL_SERVER_HOST");
        SessionClient api = new SessionClientImpl(serverHostName);
        sessionToken = api.openSession(API_KEY);
    }

    public DocumentRepository() {
        ...
    }
}
```

Trying to instantiate this class in a test running on your laptop, you could easily stumble on a `NullPointerException` because you hadn't set an environment variable named `ACL_SERVER_HOST`, or a `UnknownHostException` because your laptop fails to connect to the real server running in the lab behind a firewall. Dependencies like this are business as usual; the real problem is that, since the dependency is hardcoded into a static initialization block, there's not much you can do about it.

7.2.2 *Can't invoke a method*

Even with all objects instantiated and ready to go, your test might stumble on executing the interaction and scenario you want. For instance, you might want to invoke a method that's marked `private`. It's the overly conservative visibility modifier issue again!⁶

Another source for the inability to invoke the method you want might be its opacity; you can't figure out what the method expects to receive as its arguments. This is

⁶ I'd argue that in this case the true issue is that the private method should really be public on a new class. Clearly it's complicated enough, because otherwise you wouldn't want to test it directly.

especially true with APIs that rely on good old `java.util.Map`. Though that may be a “flexible” API, it comes with some serious issues. You can’t intuitively see what the `Map` should contain, so you have to go into the source code or documentation to find out, which grinds your pace to a halt. The same goes with the `private` method you want to test directly. Unless you refactor the design, you’re left with two bad alternatives: don’t test it or whip up the Reflection API to circumvent the visibility modifier.

7.2.3 Can’t observe the outcome

Even if you have the means to invoke the method you want, you might have problems determining whether the right things happened afterward. The “hello world” of unit tests is to invoke a method and assert something about its return value. But if the method is supposed to engage in an interaction with collaborating objects or if it doesn’t return anything as a `void` method, it gets more complicated.

Sometimes you find yourself incapable of intercepting the interaction you’re interested in. This might be because the collaborator is hard-wired into the method you’re testing and can’t be substituted with a test double. Other times, your trouble stems from the method under test starting threads to do the work, but our test code can’t intercept that thread’s execution.

This ties closely with the other major category of testability issues: the trouble you have to go through to selectively substitute parts of the implementation.

7.2.4 Can’t substitute a collaborator

Failure to substitute a collaborator is a common testability problem. It might be because the production code has a hardcoded `new Collaborator()` where you want to test its interaction with that particular collaborator. In other words, you’re missing a “seam” where you can intercept and observe the interaction; you’re technically incapable of substituting the collaborator.

Though such hardcoded collaborator implementation is all too frequent and extremely inconvenient, perhaps even more common an occurrence is a structure that doesn’t make it impossible to substitute a collaborator, but that makes it unnecessarily strenuous. That structure is sometimes referred to as a *method chain*: `getCollaborator().doStuff().askForStuff().doMoreStuff()`.⁷

Even if you could substitute the collaborator, such a method chain means that your test double needs to return a test double that needs to return a test double that needs to... Tedious!

7.2.5 Can’t override a method

Substituting test doubles for collaborators is a crucially important tool in writing unit tests. But it’s not the only tool: sometimes you don’t want to substitute a collaborator

⁷ This situation calls for a reference to the *Law of Demeter*, which warns us about code knowing too much about other units of code: http://en.wikipedia.org/wiki/Law_of_Demeter.

but rather a *part of the object under test*. Say the method you'd like to invoke and test acquires a collaborator through a method like this:

```
private static final Collaborator getCollaborator() { ... }
```

Now, all three of those keywords (`private`, `final`, `static`) essentially mean that you can't do this in your test:

```
@Test
public void test() {
    final Collaborator collaborator = new TestDouble();
    ObjectUnderTest o = new ObjectUnderTest() {
        @Override
        private static final getCollaborator() {
            return collaborator;
        }
    };
    ...
}
```

You can't do that because the compiler won't let you override a method that's `final`, a method that's `private`, nor a method that's `static`.⁸ Again, your options are limited to a range from bad to worse and involve heavy-duty reflection and byte code manipulation—none of which we want to do.

Having faced all of these testability issues again and again over the years, I've come to appreciate a few simple heuristics—guidelines—for testable design. Let's take a look at them.

7.3 Guidelines for testable design

The following is a set of do's and do not's I've gathered and whose importance I've learned through repeated yak shaving.⁹ They aren't in any particular order, and none of them is a universal truth—just things to keep in mind so that you think twice before deciding to go against these guidelines.

With that disclaimer, let's start with some guidelines that relate to what we talked about in the previous section.

7.3.1 Avoid complex private methods

There's no easy way to test a `private` method. Therefore, you should strive to make it so that you don't feel the need to test your `private` methods directly.

Note that I didn't say you shouldn't test your `private` methods, but that you shouldn't test them *directly*. As long as those methods are trivial utilities and shorthands to make your `public` methods read well, it should be perfectly fine that they get tested only through those `public` methods.

⁸ Note that `static` methods can be *hidden* by a subclass, but that's not the same as *overriding*, and largely useless in terms of what we're looking for.

⁹ Yak-shaving defined, http://en.wiktionary.org/wiki/yak_shaving.

When the private method isn't that straightforward and when you do feel like you want to write tests for it, you should refactor your code so that the logic encapsulated by the private method gets moved over to another object responsible for that logic, and where it's a public method as it should be.

7.3.2 Avoid final methods

Few programs need final methods. The odds are you don't need them either.

The main purpose of marking a method as final is to ensure that it isn't overwritten by a subclass. An Oracle Technology Network article on secure coding guidelines for Java¹⁰ suggests that "making a class final prevents a malicious subclass from adding finalizers, cloning and overriding random methods." Though that's true in some contexts, it doesn't mean that *you* should need the final modifier.

There are two problems with the preceding logic. First, those potential subclasses are likely written by the people sitting next to you. Second, the Reflection API can be used to remove the final modifier. In practice, the only situation where you might reasonably want to make a method final is when you load foreign classes at runtime, or you don't trust your colleagues (which sounds like you have much bigger issues to worry about).

You might add a third situation to this list: not trusting yourself; for example, not trusting yourself to not accidentally override the wrong method in a concrete class that's part of a *template method* pattern.¹¹ Even then, your tests should catch that mistake quite easily. (And if you're really paranoid you could go all out and write a *contract test* that checks for proper use with the Reflection API.)

The big question is this: does that final keyword interfere with your tests, and if it does, does the burden of worse testability outweigh the benefit of having the final there.

What about the performance of final?

One of the arguments that people sometimes lean on in support of final methods is performance. Namely, it's said that since final methods can't be overridden, the compiler can optimize the code by inlining the method's implementation.

It turns out that the *compiler* can't safely do this because the method might have a non-final declaration at runtime. But a JIT compiler would be able to inline such methods at runtime, which does present a theoretical performance benefit.

Jack Shirazi and Kirk Pepperdine, the authors of *Java Performance Tuning, 2nd Edition* (O'Reilly Media, January 2003), and javaperformancetuning.com have said, "I wouldn't go out of my way to declare a method or class final purely for performance reasons. Only after you've definitely identified a performance problem is this even worth considering."

¹⁰ "Secure Coding Guidelines for the Java Programming Language, Version 4.0," <http://www.oracle.com/technetwork/java/seccodeguide-139067.html>.

¹¹ "Template method pattern," http://en.wikipedia.org/wiki/Template_method_pattern.

7.3.3 Avoid static methods

Most static methods shouldn't be. The reason to have them is generally either because they don't relate to a specific instance of a class or because we couldn't bother figuring out where it belongs so we made it a static method in this utility class. The former motive is solid but the latter is mere ignorance or lack of interest. Yet another reason, and an unfortunately common one, is to make it easier to provide global access to the method by making it static.

Some methods are naturally static. For example, a utility method that performs a calculation on numbers would likely be a good candidate for a static method. So what distinguishes a method as one that should or shouldn't be static? Perhaps the example in the following listing serves to make the idea more clear.

Listing 7.1 Avoid static methods that you might want to stub out

```
public static int rollDie(int sides) {  
    return 1 + (int)(Math.random() * sides);  
}
```

The `rollDie()` method produces a random result as if you'd rolled a die with a given number of sides. Dealing with random factors in automated tests is generally something you should avoid, so you'll likely want to stub out the `rollDie()` method in your tests.

My rule of thumb is to not make it static if you foresee that you might want to stub it out in a unit test one day. It turns out that I rarely need to stub out a pure calculation. On the other hand, I frequently find myself wanting to stub out static methods that serve as an entry point to a service or a collaborator object.

Pay attention to the kind of methods you declare static. A static method is easy to write, but you'll have a hard time later if you ever need to stub it out in a test. Instead, create an object that provides that functionality through an instance method.

7.3.4 Use new with care

The `new` keyword is the most common form of hardcoding. Every time you "new up" an object you're nailing down its exact implementation. For that reason, your methods should only instantiate objects you won't want to substitute with test doubles. This listing points out this pattern.

Listing 7.2 The new keyword hardcodes the implementation class

```
public String createTagName(String topic) {  
    Timestamp c = new Timestamp();  
    return topic + c.timestamp();  
}
```

The method in the listing builds a string based on the given input and a timestamp generated by a `Timestamp`, which is instantiated inside the method.

Many programmers don't think of `new` as the kind of red flag that `static` and `final` are for test-infected programmers. That doesn't make it completely innocent,

as there's no way (other than byte code manipulation) for a test to override the specific class being instantiated within the method under test; for example, the `Timestamp` in listing 7.2.

When instantiating objects, you should ask yourself: is this object something you'd want to swap out in a test? If it's a true collaborator and you might want to change its implementation on a test-by-test basis, it should be passed into the method somehow rather than instantiated from within that method.

7.3.5 Avoid logic in constructors

Constructors are hard to bypass because a subclass's constructor will always trigger at least one of the superclass constructors. Hence, you should avoid any test-critical code or logic in your constructors.

The next listing is a contrived implementation of a *universally unique identifier* (UUID), which is made up of the generating computer's MAC address and a time-stamp.

Listing 7.3 Doing too much in a constructor

```
public class UUID {
    private String value;

    public UUID() {
        // First, obtain the computer's MAC address by
        // running ipconfig.exe and parsing its output
        long macAddress = 0;
        Process p = Runtime.getRuntime().exec(
            new String[] { "ipconfig", "/all" }, null);
        BufferedReader in = new BufferedReader(
            new InputStreamReader(p.getInputStream()));
        String line = null;
        while (macAddress == null &&
            (line = in.readLine()) != null) {
            macAddress = extractMACAddressFrom(line);
        }

        // Obtain the UTC time and rearrange
        // its bytes for a version 1 UUID
        long timeMillis = (System.currentTimeMillis() * 10000)
            + 0x01B21DD213814000L;
        long time = timeMillis << 32;
        time |= (timeMillis & 0xFFFF00000000L) >> 16;
        time |= 0x1000 | ((timeMillis >> 48) & 0x0FFF);

        ...
    }
}
```

Let's say you'd like to test the `UUID` class shown in listing 7.3. Looking at its bloated constructor, you have a testability issue: you can only instantiate this class (or its subclasses) on a Windows computer because it tries to execute `ipconfig /all`. I happen to be running a Mac so I'm screwed!

A much better approach would be to extract all of that code into protected methods that *can* be overridden by subclasses, as shown here.

Listing 7.4 Moved logic from constructor to protected helper methods

```
public class UUID {
    private String value;

    public UUID() {
        long macAddress = acquireMacAddress();
        long timestamp = acquireUuidTimestamp();
        value = composeUuidStringFrom(macAddress, timestamp);
    }

    protected long acquireMacAddress() { ... }
    protected long acquireUuidTimestamp() { ... }

    private static String composeUuidStringFrom(
        long macAddress, long timestamp) { ... }
}
```

With the modifications in listing 7.4, you can instantiate UUID objects in your tests regardless of which operating system you’re running by overriding the specific bits you want:

```
@Test
public void test() {
    UUID uuid = new UUID() {
        @Override
        protected long acquireMacAddress() {
            return 0; // bypass actual MAC address resolution
        }
    };
    ...
}
```

To summarize this guideline, whatever code you find in your constructors, make sure that it’s not something you’ll want to substitute in a unit test. If there is such code, you’d better move it to a method or a parameter object you can override.¹²

7.3.6 Avoid the Singleton

The *Singleton* pattern has cost this industry millions and millions of dollars in bad code. It was once written down as a pattern for “ensuring a class has one instance, and to provide a global point of access to it.” I suggest you don’t need either half of that proposition.

There are situations where you want just one instance of a class to exist at runtime. But the Singleton pattern tends to prevent *tests* from creating different variants, too. Let’s look at the traditional way to implement a Singleton, shown in the next listing.

¹² Overriding a method that’s called from the constructor is tricky and could result in unexpected behavior. Refer to the Java Language Specification (<http://mng.bz/YFFT>) for details.

Listing 7.5 Traditional implementation of the Singleton pattern

```
public class Clock {
    private static final Clock singletonInstance = new Clock();

    // private constructor prevents instantiation from other classes
    private Clock() { }

    public static Clock getInstance() {
        return singletonInstance;
    }
}
```

Making the constructor private and restricting access to the `getInstance()` method essentially means that you can't substitute the `Clock` instance once it's been initialized. This is a big issue because whenever you want to test code that uses the `Clock` Singleton, you're stuck:

```
public class Log {
    public void log(String message) {
        String prefix = "[" + Clock.getInstance().timestamp() + " ] ";
        logFile.write(prefix + message);
    }
}
```

Wherever the code you want to test acquires the `Clock` instance through the static Singleton accessor, your only option is to inject a new value for the Singleton instance in the Singleton class using reflection (and you don't want to use reflection in your tests), or by adding a setter method for doing that.

SURVIVING A SINGLETON If you do find yourself with a static Singleton accessor, consider making the singleton `getInstance()` method return an interface instead of the concrete class. An interface is much easier to fake when you don't need to inherit from the concrete class!

The much better—and testable—design would be a singleton with a lowercase 's' that doesn't *enforce* a single instance but rather relies on the programmers' agreement that "we'll only instantiate one of these in production." After all, if you need to protect yourself from sloppy teammates you have bigger problems to worry about.

7.3.7 Favor composition over inheritance

Inheritance for the purpose of reuse is like curing a sore thumb with a butcher knife. Inheritance does allow you to reuse code but it also brings a rigid class hierarchy that inhibits testability.

I'll let Miško Hevery, a clean code evangelist at Google, explain the crux of the issue:¹³

¹³ James Sugrue blog, an interview with Miško Hevery, "The Benefits of Testable Code," November 17, 2009, <http://java.dzone.com/articles/benefits-testable-code>.

The point of inheritance is to take advantage of polymorphic behavior NOT to reuse code, and people miss that; they see inheritance as a cheap way to add behavior to a class. When I design code I like to think about options. When I inherit, I reduce my options. I am now sub-class of that class and cannot be a sub-class of something else. I have permanently fixed my construction to that of the superclass, and I am at a mercy of the super-class changing APIs. My freedom to change is fixed at compile time.

On the other hand composition gives me options. I don't have to call superclass. I can reuse different implementations (as supposed to reusing your super methods), and I can change my mind at runtime. Which is why if possible I will always solve my problem with composition over inheritance. (But only inheritance can give you polymorphism.)

Note that Miško isn't saying inheritance is bad; inheritance for polymorphism is totally okay. But if your intent is to *reuse functionality* it's often better to do that by means of composition: using another object rather than inheriting from its class.

7.3.8 Wrap external libraries

Not everybody is as good at coming up with testable designs as you are. With that in mind, be extremely wary of inheriting from classes in third-party external libraries, and think twice before scattering direct calls to an external library throughout your code base.

We already touched on the testability problems with inheritance. Inheriting from an external library tends to be worse, as you don't have as much control over the code you're inheriting from. Whether it's through inheritance or direct calls, the more entangled your code is with the external library, the more likely it is that you'll need those external classes to be test-friendly.

Pay attention to the testability of the classes in the external library's API and, if you see red flags, be sure to wrap the library behind your own interface that's test-friendly and makes it easy to substitute the actual implementation.

INABILITY TO CHANGE THE DESIGN Every now and then you'll find yourself wondering what to do because you're stuck in a situation where your design isn't testable *and you can't change the design* to be more testable. Despite the fact that the code you're looking at is technically "yours," this situation is the same as with external libraries, which you also have little control over. Thus, the approach to dealing with this testability challenge is also much alike: wrap the untestable piece of code into a thin layer that *is* testable.

7.3.9 Avoid service lookups

Most service lookups (like acquiring a Singleton instance through a static method call) are a bad trade-off between a seemingly clean interface and testability. It's only *seemingly* clean because the dependency that could've been explicit as a constructor parameter has been hidden within the class. It might not be technically impossible to substitute that dependency in a test, but it's bound to be that much more work to do so.

Let's take a look at an example. The next listing presents a class responsible for making remote search requests to a web service through an `APIClient`.

Listing 7.6 Service lookups are harder to stub than constructor parameters

```
public class Assets {
    public Collection<Asset> search(String... keywords) {
        APIRequest searchRequest = createSearchRequestFrom(keywords);
        APICredentials credentials = Configuration.getCredentials();
        APIClient api = APIClient.getInstance(credentials);
        return api.search(searchRequest);
    }

    private APIRequest createSearchRequestFrom(String... keywords) {
        // omitted for brevity
    }
}
```

Note how the `search()` method acquires the `APIClient` with a service lookup. This design doesn't drive testability as well as we'd like. This listing shows how you might write a test for this object.

Listing 7.7 Stubbing out a service lookup implies an extra step

```
@Test
public void searchingByKeywords() {
    final String[] keywords = {"one", "two", "three"}
    final Collection<Asset> results = createListOfRandomAssets();
    APIClient.setInstance(new FakeAPIClient(keywords, results));
    Assets assets = new Assets();
    assertEquals(results, assets.search(keywords));
}
```

The indirection implied by the service lookup within `Assets` stipulates that we go through the extra step of configuring the service lookup provider (`APIClient`) to return our test double when someone asks for it. Though it's just a one-liner in this test, it implies several lines of code in `APIClient` that only exist to work around a testability issue.

Now, let's contrast this to a situation where we've employed constructor injection to pass our test double directly to the `APIClient`. This listing shows what that test would look like.

Listing 7.8 Constructor parameters are easier to substitute

```
@Test
public void searchByOneKeyword() {
    final String[] keywords = {"one", "two", "three"}
    final Collection<Asset> results = createListOfRandomAssets();
    Assets assets = new Assets(new FakeAPIClient(keywords, results));
    assertEquals(results, assets.search(keywords));
}
```

Again, it's just one line shorter but that's a full 20% less than listing 7.7 and we don't need the workaround setter method in `APIClient` either. Aside from our code being more compact, passing dependencies explicitly through the constructor is a natural, straightforward means to wire up our objects with their collaborators. Friends don't let friends do static service lookups.¹⁴

7.4 Summary

We started off this chapter by discussing what testable design is, making a corollary to modular design and introducing the SOLID design principles of the *Single Responsibility Principle*, the *Open-Closed Principle*, the *Liskov Substitution Principle*, the *Interface Segregation Principle*, and the *Dependency Inversion Principle*. Abiding to these principles is likely to lead to more modular design and, therefore, more testable design.

Issues with testability boil down to our inability to write tests or the excess trouble we have to go through to get it done. These troubles include:

- Inability to instantiate a class
- Inability to invoke a method
- Inability to observe a method's outcome or side effects
- Inability to substitute a test double
- Inability to override a method

Learning from these issues and drawing from our experience, I described a number of guidelines to avoid the aforementioned testability issues and to arrive at testable designs. Those guidelines teach you to avoid `final`, `static`, and complex `private` methods. You also learned to treat the `new` keyword with care, as it essentially hard-codes an implementation we can't substitute.

You should avoid significant logic in your constructors because they're hard to override. You should avoid the traditional implementation of the Singleton pattern and instead opt for the *just create one* approach. You know to favor composition over inheritance because it's more flexible than the class hierarchy implied by inheritance.

We noted the danger of inheriting from, and making liberal direct use of, external libraries, as such libraries are out of your control and often exhibit a less testable design than your own. Finally, you learned to avoid service lookups and go for passing your dependencies in through constructor parameters instead.

Though they're far from being hard-and-fast rules, keeping these guidelines in mind and stopping to think for a second before going against them is bound to help you devise more testable designs.

¹⁴ Worried that your constructor is already taking in six parameters? It sounds like your code might be missing an abstraction or two—that's what such *data clumps* often reveal after a closer look. If that's the case, perhaps introducing a parameter object (<http://www.refactoring.com/catalog/introduceParameterObject.html>) would help.

And there's more! The internet is chock full of good information related to testable design. My favorites are probably the Google Tech Talk videos. If you have some time to spare, go to youtube.com and search for "The Clean Code Talks." A lot of good stuff there!

When you're done with those videos, we're going to move on to another topic altogether. In the next chapter, you're going to see how you might use other JVM languages for writing tests for your Java code.

Writing tests in other JVM languages

In this chapter

- Using multiple programming languages on the JVM
- Achieving conciseness in test code with a dynamic language
- Added expressiveness with behavior-driven development frameworks

Programming is about expressing your ideas and intent in a language understood by a computer. For a Java programmer, that means writing code in a language that the Java compiler can compile into bytecode to be run on the JVM. More than one programming language can be used for writing code to be run on the JVM, though.

Each alternative JVM language has its distinct syntax and feel but all have something in common: they claim to be more concise and more expressive than Java for creating applications on the JVM.

In this chapter we'll explore what kinds of advantages these modern programming languages might have for the specific task of writing unit tests. You'll witness the ups and downs of writing tests in a dynamic language such as Groovy, and dip your toes into test frameworks that take advantage of these dynamic languages'

expressive power and the BDD style of specifying desired behavior to offer a serious alternative to writing tests in Java and JUnit.

As Dean Wampler said in 2009, “This is an exciting time to be a Java programmer.”

8.1 The premise of mixing JVM languages

The history of alternative JVM languages began almost 15 years ago when Jim Hugunin wrote Jython, an implementation of the Python language for the JVM. Though Jython struggled to gain momentum and its development had practically ceased by 2005, it served as inspiration for many more JVM languages to come.

Soon after the Jython project was announced, other languages joined the race. JRuby, a Java implementation of the Ruby programming language, had been under development since 2001, and James Strachan announced Groovy as a serious alternative for scripting on the JVM in 2003. Also in 2003, Martin Odersky released the Scala programming language, and in 2007 Rich Hickey’s Lisp dialect, Clojure, joined the ranks of serious JVM languages. With all of these languages running on the JVM, I wouldn’t be surprised if the next 10 years was named the decade of polyglot programming.¹

8.1.1 General benefits

Each of these languages offers an alternative approach to writing programs for the JVM. Whereas Groovy promised smoother interoperability with Java code and much-desired conciseness with its stripped-down syntax, JRuby offered a more drastic move away from the curly-brace world of Java syntax. Scala promised to bring the power of functional programming on top of the object-oriented Java programming language, and Clojure presents a pure functional programming language true to its Lisp heritage.

As an overview, some of the potential advantages of these languages include:

- Less boilerplate syntax hiding the beef
- More literal data structures
- Additional methods for standard types
- More powerful language constructs

For instance, getting rid of boilerplate such as unnecessary visibility modifiers and curly braces leaves more space for your intent to shine through from the code you write. Just look at how you’d define a class in Ruby:

```
class Book
  def initialize(title = "Work in Progress")
    @title = title
    @pages = [ ]
  end

  def write page
```

¹ Polyglot programming—combining multiple programming languages and multiple “modularity paradigms” in application development.

```

    @pages << page
  end

  def page_count
    @pages.size
  end
end

```

Not a single curly brace, parentheses are largely optional, no declarations are needed for private fields, and no need to say `return` out loud. Quite succinct, don't you think?

Similarly, being able to define arrays and map structures without the constraints of Java's generics-related, angle-bracket jungle is a relief not just on the programmer's mind but on the wrists, too. Here's how easy it is to define complicated, multi-dimensional data structures in Ruby:

```

my_data = {
  :key => 'value',
  :list => [ 'L', 'I', 'S', 'T' ],
  :map => { :inside => :the_map }
}

```

One of the most common complaints voiced by programmers switching from one language to another is all those handy methods on basic types that Java is missing—string manipulation and collection operations in particular come to mind. Can you imagine how many lines of Java code it'd take to partition a list of objects into two using one of the object's attributes like this piece of Scala does?

```
val (minors, adults) = people.partition(_age < 18)
```

Another one of my favorite features of Scala is the combined power of *case classes* and *pattern matching* for representing nontrivial class hierarchies in a way that not only allows for adding new types into the hierarchy but also new operations. You'll probably have to pick up a book like Joshua Suereth's *Scala in Depth* (Manning Publications, 2012) to get the full depth of case classes, but the next listing gives a taste of what you can do with them.

Listing 8.1 Scala's case classes and pattern matching

```

abstract class Tree
case class Branch(left: Tree, right: Tree) extends Tree
case class Bunch(numberOfBananas: Int) extends Tree

def countBananas(t: Tree): Int = t match {
  case Branch(left, right) =>
    countBananas(left) + countBananas(right)
  case Bunch(numberOfBananas) => numberOfBananas
}

val myPlant = Branch(
  Branch(Bunch(1), Bunch(2)),
  Branch(Bunch(3), Bunch(4)) )

println("I've got " + countBananas(myPlant)
  + " bananas in my tree!")

```

Define
case
classes

Method for
counting
bananas in
tree

Create banana plant
with two branches, each
holding two bunches

New languages, new syntax, and new APIs may seem daunting at first, but you're never left to your own devices. A whole bunch of information and documentation is available about all of these languages, both online and between bound covers. Add to that, the communities forming around these new programming languages are friendly and full of enthusiasm that's hard to not pick up.

These languages place the power of new language constructs at our disposal. Closures, list comprehension, native regular expressions, string interpolation, pattern matching, implicit type conversions—the list goes on and these language features are bound to come in handy sooner or later.

We have a more specific interest in this chapter: how does using these languages help us in writing and maintaining tests for our Java code? Let's talk about that before diving into practice.

8.1.2 Writing tests

All of the general benefits of syntactic sugar and expressiveness, rich APIs, and powerful language constructs apply to writing tests in these other JVM languages. Many companies (mine included) are increasingly moving toward mature languages such as Scala for developing their production systems. Many others are hesitant to take the leap due to a variety of reasons.

READABILITY OVER RAW PERFORMANCE

One of the more common reasons quoted for not writing production code in dynamic languages such as Groovy or Ruby is performance. The joy of expressive and concise syntax sometimes trades off against absolute peak performance. That potential performance hit doesn't tend to matter when it comes to test code. The priority for test code, after all, is to be readable and that's where the power of dynamic languages comes to play. As long as the performance hit isn't massive, you're often better off with the added expressiveness and simplicity of the syntax.²

TESTS AS A "GATEWAY DRUG"

Aside from the advantage of using a language better suited for the job, teams can experiment and learn in a safe environment by initially moving to the new language only for automated tests. With this gateway drug approach, programmers have time to learn the language and its idioms without fear of unforeseen production issues due to unfamiliarity with the technology.

There are downsides to this polyglot approach, too. First, mixing two programming languages in the development environment may add to the complexity of build processes. Second, if you're test-driving your Java code with tests written in Groovy, for example, it's harder to see how well the emerging API is suited for being called from other Java code—the rest of the production code.

² Coincidentally, some 15 years ago many C/C++ programmers dismissed Java due to its inferior performance. Today, the JVM beats many native compilers with its just-in-time compilation and runtime optimizations.

SOME LANGUAGES SUIT TESTS BETTER THAN OTHERS

With the specific purpose of writing tests in an alternative language in mind, I can say that some of these languages are better suited for writing tests than others. I say that because all programming languages are designed, first and foremost, for the purpose of writing applications or systems software. This is where the more advanced language features and constructs show their worth.

At the same time, those language features may be somewhat insignificant for expressing your straightforward tests better or more concisely. For instance, you hardly want to introduce complex class hierarchies in your test code, so Scala's case classes might not be that useful. You probably don't need the safety of side-effect-free functions as much either when writing focused unit tests, so introducing Clojure and its functional programming paradigm into your Java project might not make much sense.

All in all, the languages most suitable for writing tests are those that provide a concise syntax and versatile data structures, not the ones that flex into any kind of complex calculation imaginable. With this in mind, the most likely top candidates to consider would be Ruby and Groovy—the former having an extremely minimal syntax and the latter being more accessible to a Java programmer.

The best way to gauge the suitability of such alternative JVM languages for writing tests for your Java code is through first-hand experience. The next best thing is what we'll do next—study what kind of tools you have to choose from and how your code and tests turn out with that toolset.

We'll start by looking at the simple comparison of writing JUnit tests in Groovy and then expand our exploration further into other test frameworks.

8.2 *Writing unit tests with Groovy*

Groovy's premise for writing unit tests is a significantly lighter syntax for expressing your intent. Though Groovy still resembles Java, its lightness comes from making a lot of Java's syntax optional.

For instance, in Groovy semicolons, parentheses, and the `return` keyword are optional and can be left out, as illustrated by the following listing.

Listing 8.2 Groovy's syntax is like stripped down Java

```
void logAndReturn(String message) {  
    def logMessage = "${new Date()} - ${message}"  
    println logMessage  
    logMessage  
}
```

The listing also illustrates a number of other ways in which Groovy reduces the amount of boilerplate programmers have to write. Because Groovy's default visibility modifier is `public`, you can omit it like in listing 8.2 unless you specifically want to restrict access. You can also see how, in Groovy, you can choose to not declare a variable's type and instead declare it with `def`.

We often find ourselves constructing strings from other strings, numerals, and other objects' string representations. Groovy makes this a lot cleaner as well. You can see that in how the `logMessage` is constructed in listing 8.2.³

These are just some of Groovy's language features that contribute to a nicer syntax overall. We're specifically interested in writing tests, so let's talk about the more specific task of test setup and how Groovy can help there.

8.2.1 Simplified setup for tests

The essence of a unit test's setup boils down to constructing object graphs for input and collaborators for the code under test to interact with. Groovy makes the work of complex object creation much simpler than Java. The following illustrates how your setup ends up being much shorter and easier to read than its Java counterpart would be.

Listing 8.3 Groovy shines at creating objects for testing

```
class ClusterTest {
    def numberOfServers = 3
    def cluster

    @Before
    void setUp() {
        cluster = new Cluster()
        numberOfServers.times {
            def name = "server-${it}"
            cluster.add new Server(name: name, maxConnections: 1)
        }
    }
    ...
}
```

Assigning to fields through default constructor ②

① it is the loop variable

The example illustrates many features that make Groovy such a great language for constructing complex objects. To start with, there are looping constructs ① that let you encapsulate the creation of multiple slightly similar objects for your test fixture.⁴

Groovy also lets you assign field values for objects by passing them through the default constructor ②. `Server` only has a default constructor, for example. This feature can be extremely handy when you want to instantiate JavaBeans-style objects that would otherwise require separate setter calls.

Furthermore, you can create fake implementations of interfaces burdened by getter and setter methods by defining a parameter:

```
class Person { String name }
```

The preceding code automatically generates `getName()` and `setName(String name)`, which makes for more concise implementations of JavaBeans-style interfaces.

³ This is called *string interpolation* and it's a common feature of dynamic scripting languages.

⁴ You shouldn't go overboard with loops, as they tend to invite complexity that you don't want in your test code.

It's not just JavaBeans-style interfaces. Groovy makes it easy to create test doubles for interfaces in general. The following one-liner, for instance, shows how you can implement a `java.lang.Runnable` with a simple block:

```
def fakeRunnable = { println "Running!" } as Runnable
```

For implementing more complicated interfaces you can use a map instead of a block:

```
def reverse = [
    equals: false,
    compare: { Object[] args -> args[1].compareTo(args[0]) }
] as Comparator
Collections.sort(list, reverse)
```

In this code, the `Comparator` object's `equals()` method will always return `false` and its `compare(a, b)` method will do a reverse natural sort (`b.compareTo(a)`).

Other than making it easy to define simple classes, Groovy also allows defining multiple top-level classes in the same source file. This is handy for keeping test doubles next to the tests that use them without cluttering the test class with several inner classes.

Speaking of keeping things in the same source file, Groovy's multiline strings let you rewrite the awkward setup method from listing 4.6 without any `StringBuilders`:

```
class LogFileTransformerTest {
    String logFile;

    @Before
    public void setUpBuildLogFile() {
        logFile = """[2005-05-23 21:20:33] LAUNCHED
[2005-05-23 21:20:33] session-id###SID
[2005-05-23 21:20:33] user-id###UID
[2005-05-23 21:20:33] presentation-id###PID
[2005-05-23 21:20:35] screen1
[2005-05-23 21:20:36] screen2
[2005-05-23 21:21:36] screen3
[2005-05-23 21:21:36] screen4
[2005-05-23 21:22:00] screen5
[2005-05-23 21:22:48] STOPPED"" "
    }

    // ...
}
```

For a more comprehensive introduction to the various syntactic sugar Groovy has to offer, refer to the “Groovy style and language guide” at <http://mng.bz/Rvfs>.

Now, let's conclude our tour in Groovyland by looking at what a regular JUnit 4 test would look like in Groovy.

8.2.2 Groovier JUnit 4 tests

For comparison, let's take a JUnit 4 test and rewrite it using Groovy. The next listing presents a unit test class written in plain old Java.

Listing 8.4 An example of a JUnit test class written in Java

```
public class RegularJUnitTest {
    private ComplexityCalculator complexity = new ComplexityCalculator();

    @Test
    public void complexityForSourceFile() {
        double sample1 = complexity.of(new Source("Sample1.java"));
        double sample2 = complexity.of(new Source("Sample2.java"));
        assertThat(sample1, is(greaterThan(0.0)));
        assertThat(sample2, is(greaterThan(0.0)));
        assertTrue(sample1 != sample2);
    }
}
```

That's a fairly typical JUnit test from the simple end of the spectrum. This listing shows how this same test would look by writing it in Groovy.

Listing 8.5 Groovy removes the syntax clutter from our test code

```
class GroovyJUnitTest {
    def complexity = new ComplexityCalculator()

    @Test
    void complexityForSourceFile() {
        def sample1 = complexity.of new Source("Sample1.java")
        def sample2 = complexity.of new Source("Sample2.java")
        assertThat sample1, is(greaterThan(0.0d))
        assertThat sample2, is(greaterThan(0.0d))
        assertTrue sample1 != sample2
    }
}
```

Notice that the length of the source code is identical. Despite Groovy's plentiful syntactic sugar, most well-written JUnit tests won't turn out much shorter with Groovy. But they'll have fewer parentheses and semicolons, and the more complicated the tests get, the more benefit Groovy yields.

There's a lot more conciseness to reap by switching to a test framework that makes full use of Groovy's features.

8.3 Expressive power with BDD tools

BDD was born out of certain TDD practitioners looking for a better vocabulary to describe the intent of the tests written as part of the TDD cycle. The word *test* didn't quite capture the spirit of specifying desired behavior and it carries a lot of connotations. Instead, the community started talking about *specifications* (or just specs) and behavior rather than tests and test methods.

As a side effect of looking for a better vocabulary, the BDD community wrote a bunch of handy alternatives to established test frameworks like JUnit. Aside from helping you steer clear of the muddy waters of testing when you're really *specifying desired behavior*, these tools tend to highlight the test's intent and fade the syntax into the background.

Let's see how your tests might look if you wrote them using some of these tools instead.

8.3.1 Groovy specs with easyb

One of the key measures of usefulness for a test framework is how easy it is for a programmer to write a test. Another key measure is the signal-to-noise ratio of describing intent and the syntax required to express that intent. The more syntax a test has, the more opaque it tends to be, as it gets harder to scan through the boilerplate and find the essence.

Alternative JVM languages like Groovy, Ruby, and Scala boast leaner syntax, and many test framework authors have taken advantage of that. One of those frameworks is easyb, which embodies the *given-when-then* vocabulary popular with BDD frameworks (<http://easyb.org>). The following listing presents a simple example spec written with easyb, describing a list's desired behavior in two distinct scenarios.

Listing 8.6 Two scenarios specified with the open source easyb framework

```
scenario "New lists are empty", {
    given "a new list"
    then "the list should be empty"
}

scenario "Lists with things in them are not empty", {
    given "a new list"
    when "an object is added"
    then "the list should not be empty"
}
```

Note how the two scenarios read almost as well as natural language. These *scenario outlines* communicate the intent clearly. easyb provides this pleasant balance of freedom (the literal strings) and structure (given-when-then).

The outline doesn't help the computer check that the specification is fulfilled. For that you need to elaborate the scenarios more. This is shown next.

Listing 8.7 easyb scenarios elaborated to let us check for compliance

```
scenario "New lists are empty", {
    given "a new list", {
        list = new List()
    }
    then "the list should be empty", {
        list.isEmpty().shouldBe true
    }
}
```

```

scenario "Lists with things in them are not empty", {
  given "a new list", {
    list = new List()
  }
  when "an object is added", {
    list.add(new Object())
  }
  then "the list should not be empty", {
    list.isEmpty().shouldBe false
  }
}

```

Scanning through the elaborated scenarios' given-when-then steps in the previous listing, you see the meaning of each step in plain English, as well as the technical details of what that step means in terms of code execution.

This can be useful when working with highly specialized domains and opaque APIs that aren't so obvious. It can also feel somewhat redundant if the code you're working with has a good API and flows through your mind as easily as it does through your fingers. As usual, the answer to whether easyb is right for your particular project is, "It depends."

easyb isn't the only BDD-style tool in existence. The Spock Framework (www.spockframework.org) provides a feature set similar to easyb. A couple of its special features warrant a mention so let's take a closer look.

8.3.2 Spock Framework: steroids for writing more expressive tests

At a glance, tests (specs) written with the open source Spock Framework don't look much different from earlier easyb scenarios. To illustrate the similarity, the following listing presents a specification for the same `List` object you saw in listing 8.7.

Listing 8.8 Spock Framework syntax has the same concise feel as easyb

```

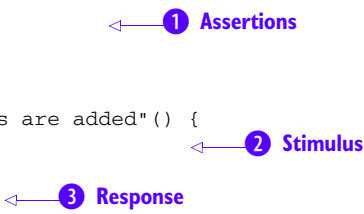
import spock.lang.*

class ListSpec extends Specification {
  def setup() {
    list = new List()
  }

  def "is initially empty"() {
    expect:
    list.isEmpty() == true
  }

  def "is no longer empty when objects are added"() {
    when:
    list.add(new Object())
    then:
    list.isEmpty() == false
  }
}

```



Spock Framework's approach is much more declarative than easyb's, for example. At the core of the framework's behavior is the concept of *blocks* and their execution as part of a *feature method's* lifecycle. The previous listing features three of Spock's six types of blocks.

The contents of a *when* block ❷ can be arbitrary code that's executed as stimulus to the code under test. Statements within an *expect* ❶ or *then* block ❸ are treated as assertions and expected to evaluate to *true*.⁵ This makes a separate assertion API somewhat unnecessary for most situations, although you're still likely to want to extract some of the more elaborate statements into a custom assertion API for your project.

The remaining blocks you didn't see in the previous listing are *setup* (and its alias, *given*), *cleanup*, and *where*.

The *setup* (*given*) is most often omitted, as any code in a feature method before the first block will be treated implicitly as *setup*. A *cleanup* block would be used to free up any resources used by the feature method, similar to what JUnit's *@After* methods are most often used for.

The *where* block is more interesting and is best explained with an example from Spock Framework's documentation.⁶ A *where* block always comes last in a feature method and effectively parameterizes its execution. The following listing illustrates how that works out.

Listing 8.9 Data-driven feature methods with *where* blocks

```
def "computing the maximum of two numbers"() {
    expect:
    Math.max(a, b) == c

    where:
    a << [5, 3]
    b << [1, 9]
    c << [5, 9]
}
```

The feature method essentially says that the *expect* block should be evaluated for two combinations of values for the variables *a*, *b*, and *c*: [5, 1, 5] and [3, 9, 9].

Such parameterized tests can be a convenient way to express multiple variations in one go. They can also become a burden if the number of parameterized variables grows beyond a small handful. Use your newly acquired superpowers with care.

I said that Spock Framework has a couple of features that warrant a special mention. The blocks you've seen so far are one, and the other feature is Spock Framework's support for specifying interactions with test doubles.

⁵ Non-Boolean values are evaluated according to Groovy truth.

⁶ "SpockBasics, Anatomy of a Spock specification," <http://code.google.com/p/spock/wiki/SpockBasics>.

8.3.3 Spock Framework's test doubles are on steroids, too

The Spock Framework comes with built-in support for creating test doubles for interfaces and nonfinal classes. In Spock's taxonomy these are called mocks and they can be created in one of two ways:

```
def screen = Mock(Screen)    // "dynamic" style
Keyboard keyboard = Mock()  // "static" style
```

Both styles are valid—in the latter case the type of the `Mock` is inferred from the type of the variable.

As usual, these test doubles are configured by invoking their methods in a certain context. Whether Spock treats them as stubs or mocks depends on how and in which context they're invoked. For example, to stub a method to return a specific value, you'd say:

```
keyboard.receive() >> "this is stubbed input"
```

The `>>` operator tells the keyboard to return a hardcoded string when its `receive` method is invoked. Similarly, faking an exception would look like this:

```
keyboard.receive() >> { throw new KeyboardDisconnectedException }
```

Checking for intended interactions taking place, the `Mock` instances are invoked within a `then` block specifying how many times you expect the method to be invoked during the test. The following listing shows an example specification with a feature method making use of Spock Framework's `Mock` objects.

Listing 8.10 Spock Framework's `Mock` objects are straightforward

```
import spock.lang.*

class NewsletterSpec extends Specification {
    def newsletter = new Newsletter()
    def frank = Mock(Subscriber)
    def fiona = Mock(Subscriber)
    def issue = Mock(Issue)

    def "delivers an issue to all subscribers"() {
        given: "Fiona and Frank each have subscribed"
        newsletter.subscribe(frank)
        newsletter.subscribe(fiona)

        and: "Frank has a duplicate subscription"
        newsletter.subscribe(frank)

        when:
        newsletter.send(issue)

        then: "Frank receives two copies of the issue"
        1 * fiona.receive(issue)
        2 * frank.receive(issue)
    }
}
```

1 Fiona has one subscription, Frank has two

2 Fiona should receive one copy

3 Frank should receive two copies

Specifying expected interactions

Spock Framework supports more than just the basic “n times” cardinality for specifying the number of expected invocations. It’s also possible to say:

```
(3.._) * frank.receive(issue) // at least 3 times
```

or

```
(_..3) * frank.receive(issue) // at most 3 times
```

Furthermore, you could state that you’re not interested in the arguments of an invocation by saying:

```
2 * frank.receive(_) // any argument will do
```

These are just some of the more common features that Spock Framework’s test doubles support. Do check the official documentation at www.spockframework.com for full disclosure.

In the previous listing, the specification class creates a fixture made of a Newsletter, two Subscribers, and an Issue to be sent to the subscribers.

Though the subscriber Fiona has just one subscription to the newsletter, Frank has two ❶. Therefore, Fiona should receive exactly one copy of the newsletter ❷ and Frank should get two ❸.

This way of specifying the expected number of interactions is both explicit and intuitive. Again, there’s no mention of “asserting” anything—we’re simply stating what we expect to happen. All in all, even the tiny examples of test code you’ve seen in this chapter scream the awesomeness of dynamic JVM languages like Groovy for writing tests. If it’s a great time to be a Java programmer, it’s an even more amazing time to be a test-infected Java programmer.

8.4 Summary

We began this chapter by reviewing some of the proposed benefits of programming in alternative JVM languages such as Groovy, Ruby, and Scala. There tends to be less boilerplate syntax and more powerful language constructs that lead to more readable code overall. Whether it’s for a safe playground for learning a new language or for the language’s advantages as such, there are many test improvements to be had simply by reaching for a dynamic language.

This chapter’s examples revolved around Groovy, one of the more widespread alternative JVM languages. Groovy has managed to hit the sweet spot for many programmers, being as close to Java as it gets and yet arming the programmer with a much thinner syntax and vastly greater power and flexibility of language.

It boils down to small details such as optional semicolons and parentheses, the ability to omit a keyword here and there, choosing to let a variable’s type be inferred from the context, and deciding that the default visibility modifier is `public`. These small things add up and it’s much easier to write concise and articulate tests.

You saw that it's possible to take advantage of the syntactic sugar of dynamic languages like Groovy, continuing to write your tests with good old JUnit. You can also take the red pill and see how far the rabbit hole goes. Recent open source BDD frameworks, such as easyb and the Spock Framework, present a more drastic step away from the traditional JUnit-style of expressing desired behavior for your code. A lot of the time, that step is for the better.

Whether adopting a different language and a new framework for testing your Java code is something you should seriously consider depends a lot on your circumstances. With that said, there's no question that the potential of significantly cleaner code is there, and the main decision factors revolve around the people involved. The technology exists and it's solid.

Speeding up test execution



In this chapter

- Finding the root causes why a build is slow
- How to make test code run faster
- How to tweak an automated build to run faster

By now you've formulated an idea about why you write automated unit tests and what makes (or doesn't make) for a good unit test. One of the properties of a good unit test is that it runs fast. Nevertheless, as you accumulate a bigger and bigger suite of tests, your feedback loop grows too. This chapter explains strategies for speeding up the execution of your test suite—the time it takes to run all of your unit tests—so that you'll get that precious feedback in a timely manner.

First, we'll get to the bottom of why we're looking for faster test execution and faster build times. Then, we'll draw an overall strategy for taking on a slow build. There are two parts to that strategy and the majority of this chapter is focused on those two missions:

- 1 Speeding up tests
- 2 Speeding up a build

Speeding up tests means diving into the code base to find opportunities for making tests run faster. We'll look at slowness inherited from the class hierarchy of our

tests, we'll take a critical look at what code we're executing during our tests, and we'll pay special attention to any place where we access the network, a database, or the plain old filesystem.

In speeding up a build, we raise our focus from the details of test code and look at how our build scripts are running our tests. In our search for a shorter build loop, we'll look at boosting performance with a higher-powered computer—or multiple computers—considering the potential upside of running tests in parallel both locally and in the cloud.

As far as test speed optimization goes, you have options. You can speed up your test code or you can change the way you run that code to get it done faster. Before we go there, let's throw some light on why we're looking for speed in the first place.

9.1 *Looking for a speed-up*

Programmers have traditionally been keen on making their code run as fast as they can. Things may have changed a little over the decades but still we all carry that inner desire for our code to be a bit faster than it is. Back in the day, the rationale was obvious: the hardware set stringent requirements and constraints for any code that would be run, be it the amount of memory or registers available or the cycle with which the processor could carry out instructions.

Today, we aren't nearly as constrained as we used to be and yet we still end up waiting for the computer to do its part. Starting off a build, we lean back and sigh, "If only I had a faster computer." As the tests pass, one by one printing a little dot on the screen, we slowly drift away wondering whether we should get a faster CPU or whether it's really about the number of cores the CPU has. "Maybe it's my hard drive? That thing is getting old. It's started making noise, too."

And that's where the problem lies: our train of thought runs faster than our tests, and that difference creates a conflict.

9.1.1 *The need for speed*

Our need for more speed comes from our test suite taking so long to run that we're faced with a choice between rather bad options. Either we stare at the terminal screen until we lose the flow, or we trigger off the build and task-switch to something else. The former kills our energy and the latter creates trouble down the road as we miss the precious fast feedback and find problems late.

The importance of our tests running fast boils down to the fact that delayed feedback causes trouble. In the small scale, developers have to wait for information—validation for what they're doing—and are held back with this task. In the large scale, our builds take such a long time that developers only run a subset of their test suites before checking in, and leave running the full test suite to a build server while they move on to the next task. When that full test run completes and something does in fact break, developers are faced with another task-switch, losing focus and eating away their productivity.

So we want faster feedback but how do we get it? How should developers approach their tests and their builds to make that feedback cycle shorter?

9.1.2 *Approaching the situation*

What do programmers do when they face a performance issue? They pull out the profiler to identify the hot spots and figure out why the performance is what it is. They apply a fix and profile the system again to see whether the fix actually improves performance.

Slow tests and slow builds should be tackled with the same overall approach: avoid making assumptions about what the problem is and, instead, gather data about the situation by analyzing the test execution. Once you've identified the essential hot spots, pull out your toolbox and get to work.

As you apply your trusted screwdriver and hammer the build into better shape, remember to rerun the build frequently to check whether you're making the kind of progress you're expecting to see.

Now, let's talk about how you can profile your tests and find those hot spots.

9.1.3 *Profiling a build*

When a build is slow the biggest chunk of slow almost always comes from running tests.¹ It's better to be sure, though, so the first thing to do should probably be some kind of a rough performance profiling for the whole build to know where all that time is spent.

Now let's look at some concrete ways of doing this. First, let's assume Apache Ant as the build tool and see how you might go about it.

PROFILING AN ANT BUILD

If your build is automated with Apache Ant, it's easy to get the basic numbers of how long each target takes:

```
ant -logger org.apache.tools.ant.listener.ProfileLogger test
```

Ant comes with a built-in *logger* that emits task and target durations in milliseconds. I ran the ProfileLogger on one fairly big open source project. Filtering out the usual output and leaving just the profiler information about build targets, this is what the report told me:

Listing 9.1 Ant's built-in ProfileLogger reports target durations

```
Target init: started Mon Jan 02 00:54:38 EET 2012
Target init: finished Mon Jan 02 00:54:38 EET 2012 (35ms)
Target compile: started Mon Jan 02 00:54:38 EET 2012
Target compile: finished Mon Jan 02 00:54:41 EET 2012 (2278ms)
Target cobertura-init: started Mon Jan 02 00:54:41 EET 2012
```

¹ The other common sources are the overhead of starting a bunch of external processes and massive filesystem operations.

```

Target cobertura-init: finished Mon Jan 02 00:54:41 EET 2012 (62ms)
Target compile-tests: started Mon Jan 02 00:54:41 EET 2012
Target compile-tests: finished Mon Jan 02 00:54:41 EET 2012 (78ms)
Target jar: started Mon Jan 02 00:54:41 EET 2012
Target jar: finished Mon Jan 02 00:54:41 EET 2012 (542ms)
Target instrument: started Mon Jan 02 00:54:41 EET 2012
Target instrument: finished Mon Jan 02 00:54:43 EET 2012 (2026ms)
Target test: started Mon Jan 02 00:54:43 EET 2012
Target test: finished Mon Jan 02 01:00:55 EET 2012 (371673ms)
Target coverage-report: started Mon Jan 02 01:00:55 EET 2012
Target coverage-report: finished Mon Jan 02 01:01:22 EET 2012 (26883ms)
Target failure-check: started Mon Jan 02 01:01:22 EET 2012
Target failure-check: finished Mon Jan 02 01:01:22 EET 2012 (3ms)

```

Running
testsCollating
coverage
report

Looking at the output in the listing you can see that the vast majority of time (370 seconds) is spent running tests and the second biggest chunk is some 27 seconds of creating a coverage report. The compile and instrument targets, for example, are only slightly over 2 seconds and the rest are smaller. In this particular project any performance optimization you make outside of the test target doesn't have much of an effect on the overall build time. That's what you should be focusing on.

Tidier profiler output with Ant-Contrib

Ant's built-in `ProfileLogger` is handy but its output is rather verbose and takes a bit of legwork to trim down into the essential information shown in listing 9.1. If you find yourself doing this kind of analysis again and again, it might make sense to get help from the Ant-Contrib open source library (<http://ant-contrib.sourceforge.net>).

Ant-Contrib provides a set of additional Ant tasks and listeners, including the `<stopwatch/>` task and the `net.sf.antcontrib.perf.AntPerformanceListener` listener. They both produce a more focused and accessible output for your build-profiling needs.

PROFILING A MAVEN BUILD

Maven doesn't have a built-in utility for measuring how long each part of the build takes. With that said, all it takes to get those measures is to add the *maven-build-utils extension* (<https://github.com/lkoskela/maven-build-utils>) in your POM file, which does the bookkeeping for you, as shown next:²

Listing 9.2 Enabling a Maven extension is straightforward

```

<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0">
  ...
  <build>
    <extensions>

```

² Extensions were added as a feature in Maven 3

```

    <extension>
      <groupId>com.github.lkoskela</groupId>
      <artifactId>maven-build-utils</artifactId>
      <version>1.4</version>
    </extension>
  </extensions>
</build>
...
<properties>
  <maven-build-utils.activationProfiles>
    perfstats
  </maven-build-utils.activationProfiles>
</properties>
...
<profiles>
  <profile>
    <id>perfstats</id>
  </profile>
</profiles>
...
</project>

```

① Register extension

② Name activation profile

③ Register profile

This simple addition activates the extension and produces a listing of execution durations for each Maven lifecycle phase and goal that was executed.

The first thing we're doing in listing 9.2 is to register the build extension ①. This tells Maven to initialize our extension and let it hook to lifecycle events during the build. Next we need to tell maven-build-utils when it should run the analysis (we don't necessarily want to do it all the time) by identifying the *activation profile* ② with the `maven-build-utils.activationProfiles` property—I've chosen to call our profile `perfstats`. Finally, in order for any of this to make sense we need to have a profile by that name ③ in our POM.

I enabled the extension on one open source project by invoking Maven with our `perfstats` profile using the `-P` parameter and got this kind of output:

Listing 9.3 Sample build profile output for an open source project

```

$ mvn package -P perfstats
...
[INFO] ----- BUILD STEP DURATIONS -----
[INFO] [generate-sources                1,2s   3%]
[INFO]   modello-maven-plugin:java      1,1s  93%
[INFO]   modello-maven-plugin:xpp3-reader 0,1s   4%
[INFO]   modello-maven-plugin:xpp3-writer 0,0s   2%
[INFO] [generate-resources              3,0s   8%]
[INFO]   maven-remote-resources-plugin:process 2,4s  79%
[INFO]   buildnumber-maven-plugin:create    0,6s  20%
[INFO] [process-resources               0,3s   0%]
[INFO]   maven-resources-plugin:resources    0,3s 100%
[INFO] [compile                        1,9s   5%]
[INFO]   maven-compiler-plugin:compile      1,9s 100%
[INFO] [process-classes                 9,8s  26%]

```

[INFO]	animal-sniffer-maven-plugin:check	7,8s	79%
[INFO]	plexus-component-metadata:generate-metadata	2,0s	20%
[INFO]	[process-test-resources	0,4s	0%]
[INFO]	maven-resources-plugin:testResources	0,4s	100%
[INFO]	[test-compile	0,1s	0%]
[INFO]	maven-compiler-plugin:testCompile	0,1s	100%
[INFO]	[process-test-classes	0,5s	1%]
[INFO]	plexus-component-metadata:generate-test-metadata	0,5s	100%
[INFO]	[test	19,0s	52%]
[INFO]	maven-surefire-plugin:test	19,0s	100%
[INFO]	[package	0,5s	1%]
[INFO]	maven-jar-plugin:jar	0,5s	100%

The output includes absolute durations for each phase and goal in seconds. We can also see each goal's contribution to its phase as a percentage as well as each phase's contribution to the whole build. Not terribly scientific but more than enough to pinpoint where most of the time is spent. Clearly most of our time is spent running tests (52%) but also a big chunk of time is spent in the *process-classes* phase (26%), mostly by a plugin called *animal-sniffer-maven-plugin*.³

It's important that we understand what kind of improvements we can expect from tweaking our tests and how that fits into the big picture. The tests are often on top of the time-sink list, but it's good to check because sometimes the CPU is choking on something else.

Now let's turn our attention to the situations where you do see a potential improvement from trimming your unit tests' execution times.

9.1.4 Profiling tests

Once you've determined that you should look at optimizing your tests, you need to pinpoint where the current bottleneck is. Once again, you could dust off your trusted Java profiler. With that said, likely you'll be better off getting an overview from the tools and reports you already have. If you still didn't figure out where all that time has gone, by all means, fire up the profiler but don't jump to it too quickly. You could spend hours with a profiler narrowing down the hot spots, whereas a simple test report might tell you immediately where to look.

Let's talk about how exactly you might peruse a test report. Figure 9.1 shows an HTML test report generated with Ant's built-in `<junitreport/>` task.

Looking at the generated report, you can see a column labeled Time(s), which tells you how many seconds it took to execute tests in a specific package, tests in a specific class, or in an individual test method. Though it's not the easiest way of figuring out which tests are the slowest, the information is all there.

If you happen to use Maven, don't worry—there's an equivalent report generator for you, too. Maven generates test reports with the built-in `maven-surefire-plugin`. Figure 9.2 shows a sample Maven test report I generated for an open source project.

³ No, I don't have the slightest idea what that plugin does.

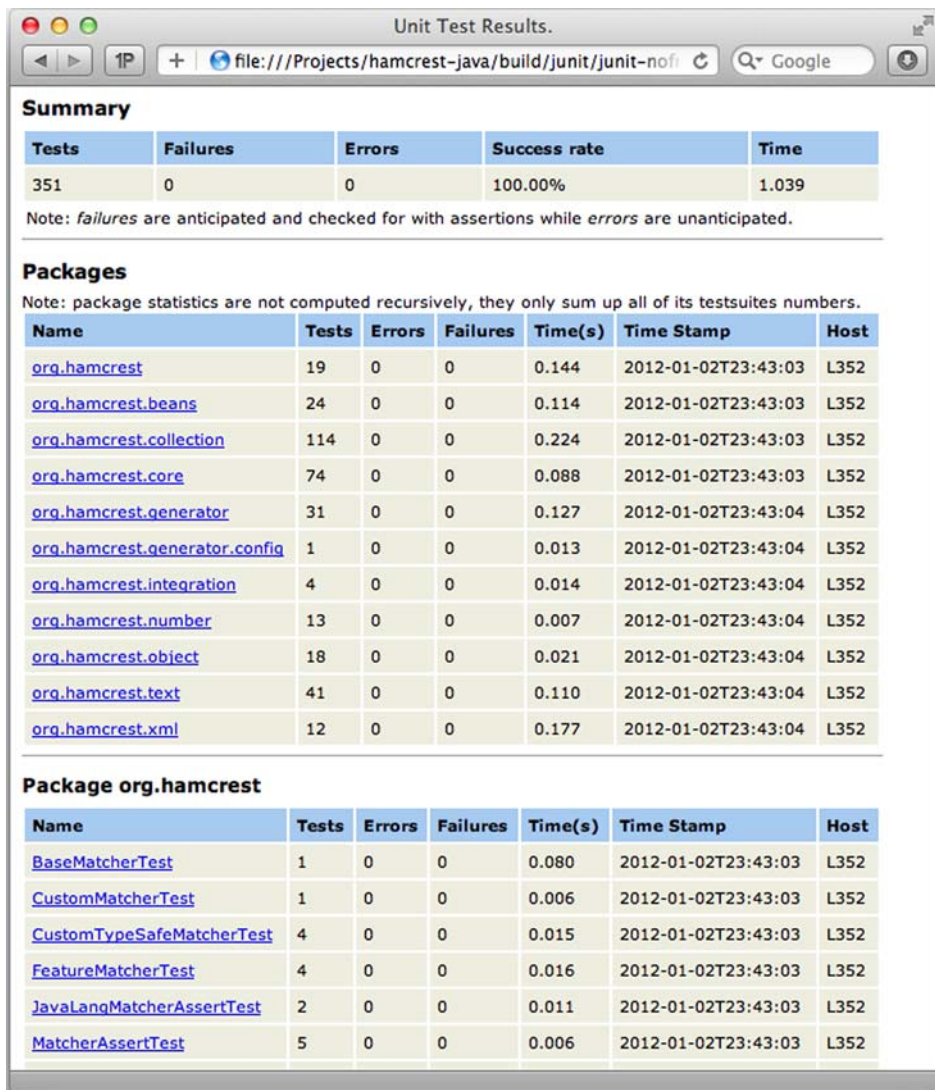


Figure 9.1 Ant's `<junitreport/>` task creates a nice test report.

Maven's test report is much like Ant's, with the statistics for packages followed by package-specific breakdowns, all the way to individual test methods. Again, all the information is there and you just need to scan through the report keeping an eye on the Time(s) column to make note of slow tests that warrant a closer look.

These reports are basic tools, but already a big help in drilling down to the slow tests. Though you want to know where the time goes, you don't want to spend hours making note of every single test method's duration.

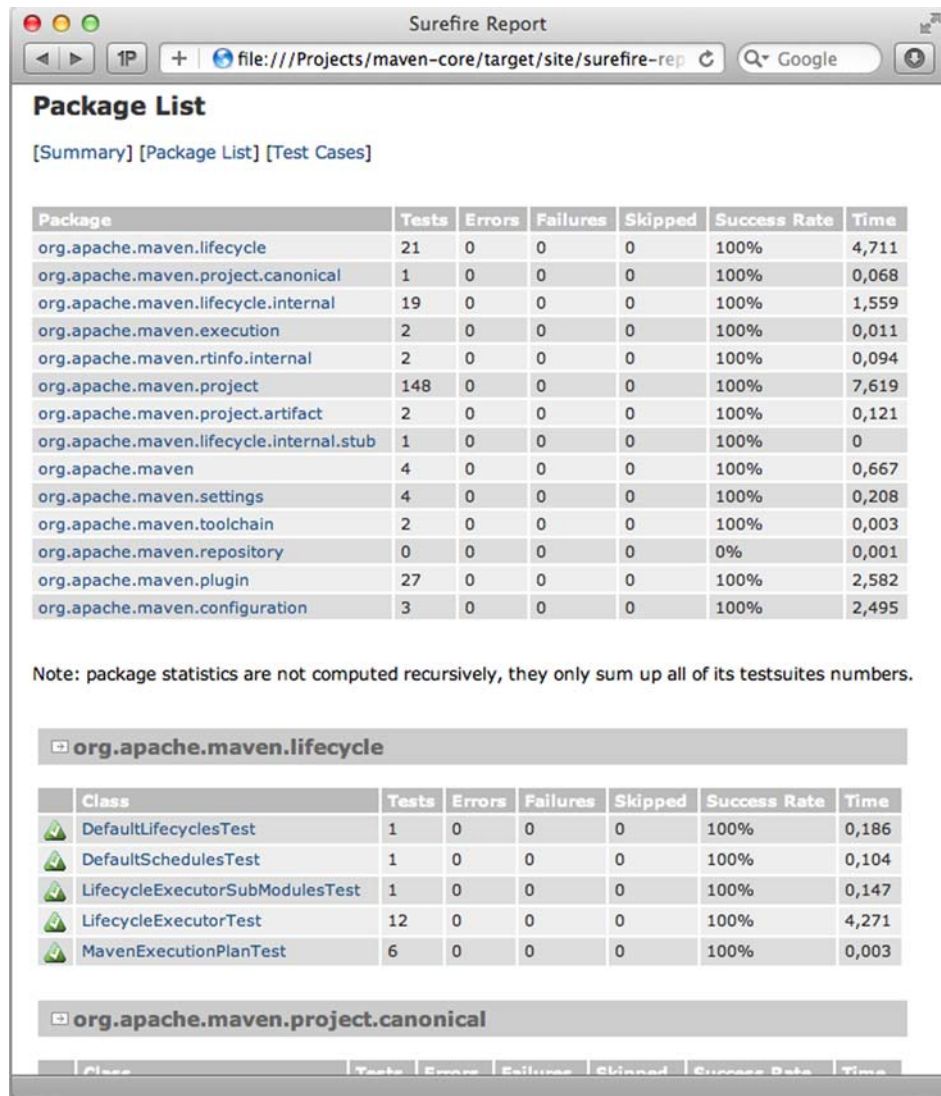


Figure 9.2 Maven's maven-surefire-plugin creates a nice test report, too.

Leaning on a continuous integration server

Continuous integration is the practice of the whole team synchronizing its changes so frequently—several times a day—that it's almost continuous. A related technique is to set up a *continuous integration server* that assists in making sure that the work that the team is synchronizing is solid.

(continued)

A continuous integration server would typically listen for changes committed to version control and, observing a commit, run a full build against that revision of source code. If the compilation, test runs, or anything else in that build fails, the server would then notify the development team that something was wrong with the latest commit.

Having a continuous integration server is generally useful, as it ensures that the code you're writing doesn't just work on your machine but also on other machines. It can also be used to alleviate the problem of a slow build, so that the developer only runs a relevant subset of tests while developing. Whenever the developer commits a change to version control, the continuous integration server runs the full test set, catching any problem that was missed by the developer's chosen subset of tests.

This is something to consider, as it tends to tighten your feedback loop from unit tests a bit. But it comes with a price tag. If there's a problem that your subset doesn't spot, the knowledge of that issue comes much later than it would if you were running the full test suite all the time.

Your toolbox of speed-ups has two compartments. On one side you have battle-tested tricks and checklists around tweaking test code. On the other side you have tips and tricks for manipulating your build scripts for that extra bit of *oomph*. Sometimes the biggest bang for the buck is in tuning your test code and sometimes the code isn't what's bogging your build down.

In summary, there are two distinct areas where you can work on improving the speed of our feedback cycle: code and build. The remainder of this chapter is split between these two pieces of the pie, each presenting tips and strategies for speeding up our test runs. We'll start with code and finish with more grandiose plans for our build infrastructure.

9.2 *Speeding up test code*

The essence of speeding up test code is to find slow things and either make them run faster or not run them at all. This doesn't mean that you should identify your slowest test and delete it. That might be a good move, too, but perhaps it's possible to speed it up so that you can keep it around as part of your safety net of regression tests.

In the next few sections we'll explore a number of common suspects for test slowness. Keeping this short list in mind can take you quickly to the source of slowness and avoid the hurdle of firing up a full-blown Java profiler. Let the exploration begin!

9.2.1 *Don't sleep unless you're tired*

It might seem like a no-brainer but if you want to keep your tests fast, you shouldn't let them *sleep* any longer than they need to. This is a surprisingly common issue in the field, so I wanted to point it out explicitly even though we won't go into it any deeper than that. The problem of sleeping snails and potential solutions to them was dis-

cussed in section 5.6 so there's not much point in reiterating them beyond this: don't rely on `Thread.sleep()` when you have synchronization objects that'll do the trick much more reliably.

There. Now let's move on to more interesting test-speed gotchas.

9.2.2 Beware the bloated base class

One place where I often find sources of slowness is a custom base class for tests. These base classes tend to host a number of utility methods as well as common setup and teardown behavior. All of that code may add to your convenience as a developer writing tests, but there's a potential cost to it as well.

Most likely all of those tests don't need all of that convenience-adding behavior. If that's the case, you need to be careful not to accidentally amortize the performance hit of running that code over and over again for all of your tests in a large code base.

You don't want your test to set up things you'll never use!

THE SLOWNESS IS IN THE STRUCTURE

The reason why a test class hierarchy and its stack of setup and teardown methods are potentially such a drag on test execution time lies in how JUnit treats those hierarchies. When JUnit is about to execute the tests in a given test class, it first looks for any `@BeforeClass` annotated methods *in the whole inheritance tree* with reflection, walking all the way up to `java.lang.Object` (which obviously doesn't have any JUnit annotations). JUnit then executes all of these `@BeforeClass` methods, parents first.

Before each `@Test` method, it does the same with the `@Before` annotation, scanning every superclass and executing all of the setup methods it found. After running the test, it scans for and executes any `@After` methods and, once all tests in the class are executed, scans for and runs any `@AfterClass` annotations.

In other words, if you inherit from another class, that class's setup and teardown will be run alongside your own whether or not they're needed. The deeper the inheritance tree, the more likely that some of those setup and teardown executions and the traversal through the hierarchy are wasted CPU time.

EXAMPLE

Let's consider a scenario depicted in figure 9.3 where you have a hierarchy of base classes that provides a variety of utilities and setup/teardown behavior for your concrete test classes.

What you see in figure 9.3 is an abuse of class inheritance. The `AbstractTestCase` class hosts custom assertions general enough to be of use by many, if not most, of your test classes.

`MiddlewareTestCase` extends that set of utilities with methods for looking up middleware components like Enterprise JavaBeans. The `DependencyInjectingTestCase` serves as a base class for any tests needing to wire up components with their dependencies. Two of the concrete test classes, `TestPortalFacadeEJB` and `TestServiceLookupFactory`, extend from here having all of those utilities at their disposal.

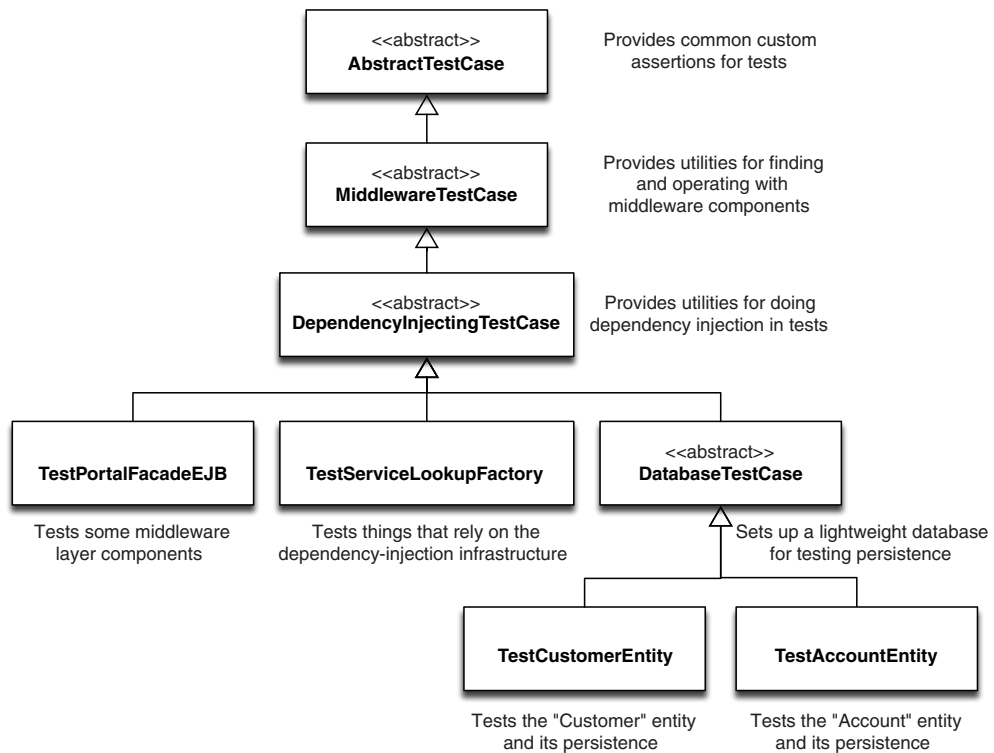


Figure 9.3 A hierarchy of abstract base classes providing various utilities for concrete test classes. Long inheritance chains like this can often cause unnecessary slowness.

Also extending from the `DependencyInjectingTestCase` is the abstract `DatabaseTestCase`, which provides setup and teardown for persistence-related tests operating against a database. Figure 9.3 shows two concrete test classes extending from the `DatabaseTestCase`: `TestCustomerEntity` and `TestAccountEntity`. That's quite a hierarchy.

There's nothing intrinsically wrong about class hierarchies but there are concrete downsides to having big hierarchies. For one, refactoring anything in the chain of inheritance may potentially have a vast effect—that refactoring is going to be that much more laborious to carry out. Second—and this is crucial from a build-speed perspective—it's unlikely that *all* concrete classes at the bottom of that hierarchy actually need and use *all* of those utilities.

Let's imagine that your tests have an average of 10 milliseconds of unnecessary overhead from such an inheritance chain. In a codebase with 10,000 unit tests that would mean a *minute and a half* of extra wait time. In my last project it would've been three minutes. Oh, and how many times a day do you run your build? Likely more than just once a day.

In order to avoid that waste, you should prefer composition over inheritance (just like with our production code) and make use of Java's static imports and JUnit's `@Rule` feature to provide helper methods and setup/teardown behavior for your tests.

In summary, you don't want your test to set up things you'll never use. You also don't want strange people crashing your party and trashing the place, which sort of leads us to the next common culprit of slow tests.

Speaking of setup...

9.2.3 Watch out for redundant setup and teardown

One of the first things you learn about JUnit is how to move common parts of your test methods into special setup and teardown methods tagged with the `@Before` and `@After` annotations. This is an extremely useful utility, but with great power comes the ability to maim yourself, performance-wise.

The code in `@Before` and `@After` methods is run once for each and every single test. If your test class has 12 test methods, its `@Before` and `@After` methods are run 12 times—regardless of whether they *need* to be run every time. Sometimes this might be acceptable, but sometimes that piece of code takes a while to execute and those whiles add up.

When you do have a test class where the setup needs to be done just once, it's delightfully simple to fix by replacing `@Before` with `@BeforeClass`. Sometimes the redundancy is harder to see and sometimes—this is important—your trivial fix doesn't work because there's no redundancy after all.

There's nothing like an example to illustrate the underlying problem so let's look at the following listing, which presents a case that might not be as straightforward as it seems.

Listing 9.4 Some setup and teardown could be done less frequently

```
public class BritishCurrencyFormatTest {
    private Locale originalLocale;

    @Before
    public void setLocaleForTests() {
        originalLocale = Locale.getDefault();
        Locale.setDefault(Locale.UK);
    }

    @After
    public void restoreOriginalLocale() {
        Locale.setDefault(originalLocale);
    }

    // actual test methods omitted
}
```

The setup takes the current locale, stores it for safekeeping, and sets a new locale for the tests' purposes. The teardown restores the original locale so that this test class leaves the environment in the same state that it was.

On the surface it looks like a no-brainer to swap in `@BeforeClass` and `@AfterClass`, followed by a quick celebration of shaving off another millisecond from our build time.⁴ It's true that we could do that—as long as no other tests will be run in between these test methods. You can only rely on `@BeforeClass` and `@AfterClass` if you can trust that other threads aren't going to change the locale and won't expect a specific locale being set.

In other words, what you should be looking for is a setup method that performs an immutable, side-effect-free operation again and again—or one that has side effects nobody else cares about.

One of my technical reviewers, Martin Skurla, provided a great example of this kind of a situation. He had a set of tests and a `@Before` method that would look for source files in a certain directory within the project and parse them into an object graph used by the tests. Because all of this was done in a `@Before` method, those same source files were parsed several times. Once Martin noticed what was going on, he wrote a small utility class that would cache the resulting object graph and skip the parsing when it could. After all, those source files wouldn't change between tests.

Having `@Before` and `@After` at your disposal is next to priceless. Nevertheless, if your build is running slow it might be worth checking how much of that time is spent in setup and teardown—and that they aren't being executed for no good reason.

9.2.4 *Be picky about who you invite to your test*

The more code you execute, the longer it takes. That's a simple way to put it as far as our next guideline goes. One potential way of speeding up your tests is to run less code as part of those tests. In practice that means drawing the line tightly around the code under test and cutting off any collaborators that are irrelevant for the test at hand.

Figure 9.4 illustrates this with an example that involves three classes and their respective tests. Pay attention especially to the top-left corner of the diagram: `TransactionTest` and `Transaction`.

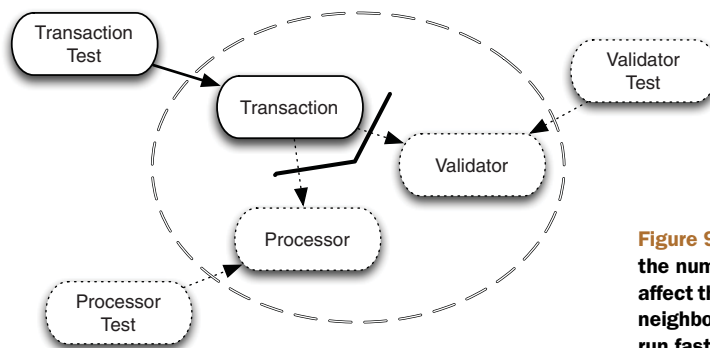


Figure 9.4 Test isolation reduces the number of variables that may affect the results. By cutting off the neighboring components your tests run faster, too.

⁴ Seriously, if we were just talking about 1 millisecond, we probably wouldn't bother trying to optimize it away.

In figure 9.4 you see three classes from a banking system. The `Transaction` class is responsible for creating transactions to be sent for processing. The `Validator` class is responsible for running all kinds of checks against the transaction to make sure that it's a valid transaction. The `Processor`'s job is to process transactions and pass them along to other systems. These three classes represent the production code. Outside of this circle are the test classes.

Let's focus on the `TransactionTest`. As you can see in the diagram, `TransactionTest` operates on the `Transaction` class and has cut off its two collaborators, `Validator` and `Processor`. Why? Because your tests will run faster if they short-circuit a dozen validation operations and whatever processing the `Processor` would do if you let it, and because the specific validations and processing these two components perform aren't of interest to the `TransactionTest`.

Think about it this way: all you want to check in `TransactionTest` is that the created transaction is both validated and processed. The *exact details* of those actions aren't important so you can safely stub them out for `TransactionTest`. Those details will be checked in `ValidatorTest` and `ProcessorTest`, respectively.

Stubbing out such unimportant parts for a specific test can save you some dear time from a long-running build. Replacing compute-intensive components with blazing fast test doubles will make your tests faster simply because there are fewer CPU instructions to execute.

Sometimes, it's not so much about the sheer number of instructions you execute but the *type* of instruction. One particularly common example of this is a collaborator that starts making long-distance calls to neighboring systems or the internet. That's why you should try to keep your tests *local* as far as possible.

9.2.5 Stay local, stay fast

As a baseline, reading a couple of bytes from memory takes a fraction of a microsecond. Reading those same bytes off a web service running in the cloud takes at least 100,000 times longer than accessing a local variable or invoking a method within the same process. This is the essence of why you should try to keep your tests as local as possible, devoid of any slow network calls.

Let's say that you're developing a trading-floor application for a Wall Street trading company. Your application integrates with all kinds of backend systems and, for the sake of the example, let's say that one of those integrations is a web service call to query stock information. Figure 9.5 explains this scenario in terms of classes involved.

What we have here is the class under test, `StockTickerWidget`, calling a `StockInfoService`, which uses a component called `WebServiceClient` to make a web service call over a network connection to the server-side `StockInfoWebService`. When writing a test for the `StockTickerWidget` you're only interested in the interaction between it and the `StockInfoService`, not the interaction between the `StockInfoService` and the web service.

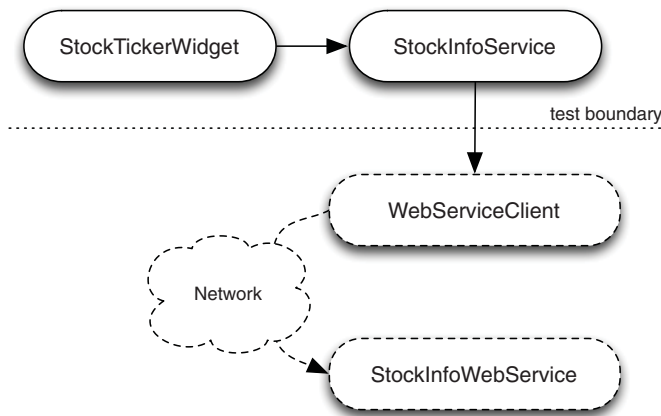


Figure 9.5 Leaving the actual network call outside the test saves a lot of time

Looking at just the transport overhead, that first leg is about one microsecond, the second leg is about the same, and the third might be somewhere between 100-300 milliseconds. Cutting out that third leg would shrink the overhead roughly...a lot.

As we discussed in chapter 3, substituting a test double for the `StockInfoService` or the `WebServiceClient` would make the test more isolated, deterministic, and reliable. It would also allow you to simulate special conditions such as the network being down or web service calls timing out.

From this chapter’s point of view, the big thing here is the vast speed-up from avoiding network calls, whether to a real web service running at Yahoo! or a fake one running in localhost. One or two might not make much of a difference but if it becomes the norm, you’ll start to notice.

To recap, one rule of thumb is to avoid making network calls in your unit tests. More specifically, you should see that the code you’re testing doesn’t access the network because “calling the neighbors” is a seriously slow conversation.

A particularly popular neighbor is the database server, the friend you always call to help you persist things. Let’s talk about that next.

9.2.6 Resist the temptation to hit the database

As we just established, making network calls is costly business in terms of test execution speed. That’s only part of the problem. For many situations where you make network calls, many of those services are slow in themselves. A lot of the time, that slowness boils down to further network calls behind the scenes or *filesystem access*.

If your baseline was a fraction of a microsecond when reading data from memory, reading that same data from an open file handle would take roughly 10 times longer. If we include opening and closing the file handle, we’re talking about 1,000 times longer than accessing a variable. In short, reading and writing to the filesystem is slow as molasses.

Figure 9.6 shows a somewhat typical setup where the class under test, `CardController`, uses a *data access object (DAO)* to query and update the actual database.

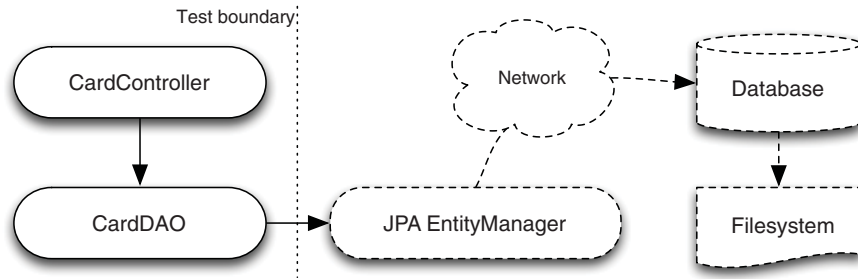


Figure 9.6 Bypassing the actual database call avoids unnecessary network and filesystem access

The `CardDAO` is our abstraction for persisting and looking up persisted entities related to debit and credit cards. Under the hood it uses the standard Java Persistence API (JPA) to talk to the database. If you want to bypass the database calls, you need to substitute a test double for the underlying `EntityManager` or the `CardDAO` itself. As a rule of thumb, you should err on the side of tighter isolation.

It's better to fake the collaborator as close to the code under test as possible. I say this because the behavior you want to specify is specifically the interaction between the code under test and the collaborator, not the collaborator's collaborators. Moreover, substituting the direct collaborator lets you express your test code much closer to the language and vocabulary of the code and interaction you're testing.

For systems where persistence plays a central role, you must pay attention to sustaining a level of testability with your architectural decisions. To make your test-infected life easier, you should be able to rely on a consistent architecture that lets you stub out persistence altogether in your unit tests, testing the persistence infrastructure such as entity mappings of an object-relational mapping (ORM) framework in a much smaller number of *integration tests*.

In summary, you want to minimize access to the database because it's slow and the actual property of your objects being persistent is likely irrelevant to the logic and behavior your unit test is perusing. Correct persistence is something you definitely

Lightweight, drop-in replacement for the database

Aside from substituting a test double for your data access object, the next best thing might be to use a lightweight replacement for your database. For instance, if your persistence layer is built on an ORM framework and you don't need to hand-craft SQL statements, it likely doesn't make much of a difference whether the underlying database product is an Oracle, MySQL, or PostgreSQL.

In fact, most likely you should be able to swap in something much more lightweight—an in-memory, in-process database such as HyperSQL (also known as HSQLDB). The key advantage of an in-memory, in-process database is that such a lightweight alternative conveniently avoids the slowest links in the persistence chain—the network calls and the file access.

want to check for, but you can do that elsewhere—most likely as part of your integration tests.

Desperately trying to carry the calling the neighbors metaphor a bit longer, *friends don't let friends use a database in their unit tests*.

So far you've learned that unit tests should stay away from the network interface and the filesystem if execution speed is a factor. File I/O isn't exclusive to databases, so let's talk more about those other common appearances of filesystem access in our unit tests.

9.2.7 *There's no slower I/O than file I/O*

You want to minimize access to the filesystem in order to keep your tests fast. There are two parts to this: you can look at what your tests are doing themselves, and you can look at what the code under test is doing that involves file I/O. Some of that file I/O is essential to the functionality and some of that is unnecessary or orthogonal to the behavior you want to check.

AVOID DIRECT FILE ACCESS IN TEST CODE

For example, your test code might suffer from split logic (see section 4.6), reading a bunch of data from files stashed somewhere in your project's directory structure. Not only is this inconvenient from a readability and maintainability perspective but, if those resources are read and parsed again and again, it could be a real drag on your tests' performance as well.

If the source of that data isn't relevant to the behavior you want to check, you should consider finding a way to avoid that file I/O with some of the options discussed in section 4.6. At the least, you should look into reading those external resources just once, caching them so that you'll suffer the performance hit just once.

So what about the file I/O that goes on inside the code you're testing?

INTERCEPT FILE ACCESS FROM CODE UNDER TEST

When you look inside the production code, one of the most common operations that involves reading or writing to the filesystem is *logging*. This is convenient because the vast majority of logging that goes on in a typical code base isn't essential to the business logic or behavior you're checking with your unit tests. Logging also happens to be easy to turn off entirely, which makes it one of my favorite build-speed optimization tricks.

So how big of an improvement are we talking about here, turning off logging for the duration of tests? I ran the tests in a code base with the normal production settings for logging and the individual unit tests' execution time ranged between 100-300 milliseconds. Running those same tests with logging level set to `OFF`, the execution times dropped to a range of 1-150 milliseconds. This happens to be a code base littered with logging statements, so the impact is huge. Nevertheless, I wouldn't be surprised to see double-digit percentage improvements in a typical Java web application simply by disabling logging for the duration of test execution.

Since this is a favorite, let's take a little detour and peek at an example of how it's done.

DISABLING LOGGING WITH A LITTLE CLASSPATH TRICKERY

Most popular logging frameworks, including the standard `java.util.logging` facility, allow configuration through a configuration file placed into the classpath. For instance, a project built with Maven likely keeps its logging configuration in `src/main/resources`. The file might look like this:

```
handlers = java.util.logging.FileHandler

# Set the default logging level for the root logger
.level = WARN

# Set the default logging level and format
java.util.logging.FileHandler.level = INFO

# Set the default logging level for the project-specific logger
com.project.level = INFO
```

Most logging configurations are more evolved than this, but it does illustrate how you might have a *handler* that logs any info level or higher messages to a file as well as any warnings or more severe messages logged by the third-party libraries you use.

Thanks to Maven's standard directory structure, you can replace this normal logging configuration with one that you only use when running your automated tests. All you need to do is to place another, alternative configuration file at `src/test/resources`, which gets picked up first because the contents of this directory are placed in the classpath before `src/main/resources`:

```
# Set logging levels to a minimum when running tests
handlers = java.util.logging.ConsoleHandler
.level = OFF
com.mycompany.level = ERROR
```

This example of a minimal logging configuration would essentially disable all logging except errors reported by your own code. You might choose to tweak the logging level to your liking, for example, to report errors from third-party libraries, too, or to literally disable all logging including your own. When something goes wrong, tests are failing, and you want to see what's going on, it's equally simple to temporarily enable logging and rerun the failing tests.

We've noted a handful of common culprits for slow builds. They form a checklist of sorts for what to keep an eye out for if you're worried about your build being too slow. Keeping these in mind is a good idea if your feedback loop starts growing too long. But what if the slowness isn't about the tests being slow? What if you can't find a performance improvement from tweaking your test code? Maybe it's time to give your infrastructure a look and see if you can speed things up over there.

9.3 **Speeding up the build**

When it comes to optimizing build speed outside of the code you're building and testing, the big question is once again, what's the bottleneck? Fundamentally, your build is either CPU-bound or I/O-bound. What that means is that upgrading your old hard disk to a blazing fast solid-state drive (SSD) has much less of an impact to your build

time if it's your CPU that's running white hot throughout the build. Most likely improvements to both I/O and CPU performance will have an impact—the question is *how big* of one.

The following sections will introduce a number of changes to your build infrastructure that may have a positive impact. Your homework is to profile your build and explore which of them has the biggest impact in your particular situation.⁵

Table 9.1 summarizes the six elemental approaches that the techniques described in the remainder of this chapter build on. The left side shows strategies for boosting the CPU-bound parts of your build and the right side shows optimizations for the I/O-bound parts.

CPU-bound?	I/O-bound?
Use a faster CPU	Use a faster disk
Use more CPU cores	Use more threads
Use more computers	Use more disks

Table 9.1 Options for speeding up a CPU or I/O-bound build

More or faster (or both). It sounds simple and a lot of the time it is. Upgrading your CPU or hard drive almost certainly gives you better overall performance. Splitting and distributing the work to multiple CPU cores is often almost as linear an improvement. Running your tests in multiple threads helps you keep the CPU warm while another thread is waiting on the disk.

Throwing better hardware at the problem is almost trivial and it works really well up to a point. But there's a limit to how fast a CPU or how fast a hard drive you can buy. From there on you need to resort to the more sophisticated end of the spectrum.

The remainder of this chapter will explore three such solutions, which are variations of:

- Using a faster disk to accelerate disk operations
- Parallelizing the build to more CPUs and more threads
- Turning to the cloud for a faster CPU
- Distributing the build to multiple computers

Sound interesting? I hope so because I'm thrilled! Now let's jump right in and elaborate on how you might go about implementing these strategies.

9.3.1 *Faster I/O with a RAM disk*

Aside from buying a faster physical hard drive, one potential way of getting a faster disk is to mount your filesystem on a virtual disk backed by your computer's RAM memory.⁶ Once you've allocated part of the computer's physical memory to be used as

⁵ If you're running a UNIX-like system you might want to look at running tools such as `ps`, `top`, `mpstat`, `sar`, `iostat`, `vmstat`, `filemon`, and `tprof` to learn more about your build's characteristics.

⁶ See "RAM drive," http://en.wikipedia.org/wiki/RAM_disk.

such a virtual disk partition, you can read and write files on it as usual—it'll just be a lot faster because you're effectively reading and writing to memory rather than a spinning hard drive.

Some UNIX-like systems come with just such a filesystem, called `tmpfs`, out of the box; for the rest you can easily set up an equivalent memory-backed corner of “disk” that your build can use.

Creating a 128-megabyte file system with `tmpfs` on Linux, for example, is this simple:⁷

```
$ mkdir ./my_ram_disk
$ mount -t tmpfs -o size=128M,mode=777 tmpfs ./my_ram_disk
```

The `tmpfs` filesystem could be mounted in the beginning of your build script, for example, and unmounted or detached after all tests have been executed. All you need to do then is to have your tests run with an effective configuration that directs all file access to files residing on your newly mounted filesystem.

In order to create a similar memory-backed filesystem on Mac OS X (which doesn't support `tmpfs`), you need to reach to another tool called `hddid` before you can mount a memory-backed filesystem:⁸

```
$ RAMDISK=`hddid -nomount ram://256000`
$ newfs_hfs $RAMDISK
$ mkdir ./my_ram_disk
$ mount -t hfs $RAMDISK ./my_ram_disk
```

Reading and writing to a RAM disk created like this is only barely faster than regular filesystems on a proper SSD disk, and modern operating systems are anyway memory-mapping files to improve disk I/O performance, but compared to a plain old traditional spinning disk, the difference can be significant. Try and see it for yourself!

And if a RAM disk doesn't give you enough of a boost, don't fret, we have a couple of tricks left up our sleeves. For instance, maybe parallelizing the build will cut down your build time?

9.3.2 Parallelizing the build

What's almost certain is that your build, whatever it's doing, isn't using *all* of your computer's capacity *all* of the time. This is good news because it means that there's potential for a performance boost from utilizing more of what you already have.

For instance, let's say you have a multicore CPU. (My fairly modest laptop has a dual-core processor.) Are you harnessing both of those cores when you build? If a significant portion of your build is CPU-bound, utilizing two CPU cores instead of one might shrink your build time by almost 50%.

Parallelization can make sense even within the scope of a single CPU core, if your build isn't completely CPU-bound and spends a significant amount of time doing disk

⁷ See `tmpfs`, <http://en.wikipedia.org/wiki/Tmpfs>.

⁸ If you're wondering about the magic number 256,000, the size for `hddid` is given as the number of 512-byte blocks: $256,000 * 512 \text{ bytes} = 128 \text{ megabytes}$.

I/O. Imagine that 80% of your build is about the CPU crunching numbers and the remaining 20% of the time it's your hard drive that's getting busy. Your opportunity lies in that 20% of disk I/O and, more specifically, what the CPU is (or isn't) doing in the meantime.

Running tests in several threads doesn't necessarily parallelize their execution in the sense that a single-core CPU might simply be switching back and forth between the threads rather than running all of them at the same time. What running your tests in several threads does facilitate, even with single-core CPUs, is running code on one thread while another thread is waiting on disk I/O.

Let's take a look at how you might implement such concurrency with your trusted workhorses: Ant and Maven.

PARALLELIZING TESTS WITH MAVEN

When it comes to parallelizing test execution, Maven couldn't make it much easier than it already is. Maven's built-in test runner can be configured to deal out your test suite to an arbitrary number of threads. All that's needed is a bit of configuration in your POM as shown next.

Listing 9.5 Instructing Maven's Surefire plugin to run tests in parallel

```
<project>
...
<build>
  <plugins>
    <plugin>
      <artifactId>maven-surefire-plugin</artifactId>
      <version>2.11</version>
      <configuration>
        <parallel>classes</parallel>
        <threadCount>2</threadCount>
      </configuration>
    </plugin>
  </plugins>
</build>
...
</project>
```

The configuration in the listing instructs Maven's built-in test runner called *Surefire* to run unit tests in parallel using two threads per each CPU core. It's also possible to specify an exact thread count regardless of how many cores there are by disabling the per-core feature:

```
<perCoreThreadCount>false</perCoreThreadCount>
```

The `<parallel/>` attribute accepts three different strategies for allocating tests to the worker threads:

- `classes` runs each test class in a different thread.
- `methods` runs each test method from one test class in a different thread.
- `both` does both.

Of these three alternatives, it's probably easiest to start with `classes` since that's least likely to surface those unwanted dependencies between your tests. It's generally good to get both working sooner or later, though.

In addition to specifying the number of threads to use, you can also tell Surefire to use as many threads as it wishes:

```
<useUnlimitedThreads>true</useUnlimitedThreads>
```

With this configuration, Surefire will look at the `<parallel/>` attribute and launch threads based on the total number of classes and/or methods your tests have. Whether or not this is faster, you'll have to try and see for yourself.

TESTNG AND MAVEN'S `<PARALLEL/>` CONFIGURATION Surefire's support for parallel test execution of TestNG tests isn't quite as configurable as it is for JUnit tests. Most importantly, the `perCoreThreadCount` and `useUnlimitedThreads` configurations aren't available.

Now, what if you *don't* have a `pom.xml` but a `build.xml`?

Parallelizing multimodule Maven builds

The release of Maven 3 brought along a promising experimental feature: the `-T` command-line argument for parallelizing the whole build. Running Maven with `mvn -T 4 clean install`, for example, would analyze the build's dependencies and attempt to parallelize its execution to four threads.

If you'd prefer to be more flexible and use as many threads as you have CPU cores, you could say `mvn -T 1C clean install` and on a quad-core computer you'd get four threads while on a dual-core laptop you'd get two threads.

PARALLELIZING TESTS WITH ANT

Since Apache Ant doesn't do anything by default (unlike Maven) you need to explicitly invoke a certain Ant task in order to execute your tests. The following listing shows a fairly typical Ant target used for running a project's JUnit tests in a serial manner.

Listing 9.6 Typical Ant target for running JUnit tests

```
<target name="test">
  <mkdir dir="build/junit-output"/>
  <junit errorproperty="failed" failureproperty="failed"
    haltonfailure="false" haltonerror="false"
    fork="yes" forkmode="perBatch">
    <formatter type="xml"/>
    <classpath refid="test.classpath"/>
    <batchtest todir="build/junit-output">
      <fileset dir="src/test/java">
        <include name="**/Test*.java"/>
        <include name="**/*Test.java"/>
      </fileset>
    </batchtest>
  </junit>
</target>
```



```

    </batchtest>
  </junit>
</target>

```

The target in the listing *forks* a clean JVM instance for running all unit tests in one big batch. Forking a JVM for running tests is considered a good practice but make sure that you're not forking too many JVM instances. You'll probably want to set `forkmode` as either `once` or `perBatch`; the default value of `perTest` would spin up a new JVM for every test you run.

Unfortunately, Ant doesn't have a built-in way of parallelizing test runs like Maven does. You can do the deed yourself—it's just more legwork. The next listing shows a configuration that divides our test suite into four batches and creates a JVM instance for each batch, effectively running our tests in parallel.

Listing 9.7 Ant script for running JUnit tests in parallel

```

<taskdef resource="net/sf/antcontrib/antcontrib.properties">
  <classpath refid="compile.classpath"/>
</taskdef>

<property name="filter"
  value="com.lassekoskela.ant.selectors.RoundRobinSelector"/>

<target name="test">
  <mkdir dir="build/junit-output"/>
  <parallel threadcount="4">
    <antcallback target="-run-test-batch" return="failed">
      <param name="junit.batches" value="4"/>
      <param name="junit.batch" value="1"/>
    </antcallback>
    <antcallback target="-run-test-batch" return="failed">
      <param name="junit.batches" value="4"/>
      <param name="junit.batch" value="2"/>
    </antcallback>
    <antcallback target="-run-test-batch" return="failed">
      <param name="junit.batches" value="4"/>
      <param name="junit.batch" value="3"/>
    </antcallback>
    <antcallback target="-run-test-batch" return="failed">
      <param name="junit.batches" value="4"/>
      <param name="junit.batch" value="4"/>
    </antcallback>
  </parallel>
  <fail message="Tests failed." if="failed"/>
</target>

<target name="-run-test-batch">
  <junit failureproperty="failed" errorproperty="failed"
    haltonfailure="false" haltonerror="false"
    fork="yes" forkmode="perBatch">
    <formatter type="xml"/>
    <classpath refid="test.classpath"/>
    <batchtest todir="build/junit-output">

```

1 Run set of parallel tasks

2 Invoke JUnit task for specific partition of tests

4 Fail build if some tests failed

3 Parameterized target that triggers standard JUnit task

```

<fileset dir="src/test">
  <include name="**/*Test.java"/>
  <include name="**/Test*.java"/>
  <custom classname="${filter}" classpath="${test.classpath}">
    <param name="partitions" value="${junit.batches}"/>
    <param name="selected" value="${junit.batch}"/>
  </custom>
</fileset>
</batchtest>
</junit>
</target>

```

Let's walk through what goes on in the listing. When we invoke the `test` target we create an output directory for test result files and set up a parallel block ❶ that executes its tasks in four threads. Each of those tasks ❷ calls a parameterized target ❸ that runs one part of our test suite. Once all of our parallel tasks have finished, we check whether the `failure` property was set and fail the build ❹ if it was.

One thing that may strike you as a foreign element in listing 9.7 is the `<custom/>` element, which is our custom extension to Ant's type hierarchy.⁹ The interesting bits of `RoundRobinSelector` are shown in the following listing. (Full source code is available at <https://github.com/lkoskela/ant-build-utils>.) We've omitted the trivial bits that connect those `<param/>` elements into the private fields.

Listing 9.8 Custom Ant type that filters a list of tests in round-robin style

```

public class RoundRobinSelector extends BaseExtendSelector {
    private int counter;
    private int partitions;
    private int selected;

    public boolean isSelected(File dir, String name, File path) {
        counter = (counter % partitions) + 1;
        return counter == selected;
    }

    ...
}

```

The `RoundRobinSelector` selects every n th test class from the full test suite based on which partition was selected. It might not be an optimal distribution of tests among the four threads, but the odds of being a *decent* distribution are pretty good.

You've seen two Ant scripts for running JUnit tests, one of them parallelized. What about the relative performance of these two configurations? How much faster it is to run your tests in four parallel batches instead of one big batch? The specific impact varies between code bases, but running them with one of my projects showed a whopping 20% improvement, largely because those tests included a lot of integration tests that did file and network I/O.

⁹ See "Custom Components," <http://ant.apache.org/manual/Types/custom-programming.html>.

Parallelizing your build to run tests concurrently with multiple cores and threads is relatively easy to do, as you’ve seen. The impact on a typical unit test suite may be “just” a 5% reduction in overall build time. Nevertheless, all of these small improvements add up and there’s no reason *not* to find the optimal number of threads to run your tests with.

With that said, if you’re looking for a bigger boost, a few pieces in this puzzle will give you more leverage than using a faster CPU.

9.3.3 Offload to a higher-powered CPU

A typical build spends most of its duration waiting on the CPU. In other words, the single biggest bottleneck you have is most likely the CPU, and speeding up the CPU would instantly speed up your build, too. Assuming that you can’t just swap in an actual CPU to your computer, how can you get a faster CPU?

A couple of years ago I was on a project that had accumulated a big build. Running a full, clean build would mean compiling thousands of classes and running some 20,000 automated tests across a handful of modules. The full build took some 24 minutes on our laptops, which meant that developers wouldn’t run the full build before checking their changes into version control. This, in turn, led to an almost constantly broken build; there was always some code that wouldn’t compile or a test that had broken. It wasn’t just the slow feedback, though. While the build was running, the computer was practically unusable with the build eating up all of the CPU cores.

The situation was unbearable and several developers had attempted to tweak the build system in the hopes of speeding it up—to no avail—until one of my colleagues came to work one day with an out-of-the-box solution to our challenges. He had tweaked our build scripts just enough to offload all the grunt work to a remote computer. This setup is illustrated in figure 9.7.

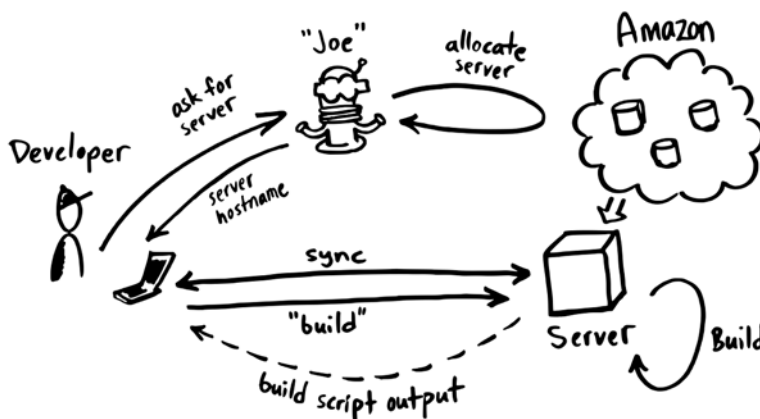


Figure 9.7 A remote build setup on top of a cloud computing platform

All you had to do was to send a message to a bot named Joe in our virtual chat room, asking for a server, and export the hostname you got in return to an environment variable. The next time we ran the build it would magically run it on a remote computer, eating up almost none of our own CPU.

What the bot did was to invoke Amazon Web Services (AWS) <http://aws.amazon.com/API> to allocate a virtual server for the developer. When the build script would detect that a virtual server had been allocated to the developer running the build, it would synchronize the developer's working copy of the codebase to the server with `rsync`, and pipe the build command to be executed on the server with `ssh`. The following listing shows a slightly simplified version of such a build script for a Maven project.

Listing 9.9 UNIX shell script that runs a Maven build on a remote server

```
#!/bin/bash

function usage {
    echo "usage: $0 [options] [arguments for maven]"
    echo ""
    echo "Options:"
    echo "  -h/--help           Display this help."
    echo "  -l/--local          Build locally."
    echo "  -r/--remote [hostname] Build remotely on given server."
    echo ""
    echo "Note: Unless overridden with -l or -r, the remote"
    echo "build machine's hostname is determined from the"
    echo "REMOTE_MACHINE environment variable (if present)."
    echo ""
}

BUILD_CMD="mvn"
while [ "$1" != "" ]; do
    case $1 in
        -l | --local )    REMOTE_MACHINE="" ;;
        -r | --remote )  shift && REMOTE_MACHINE="$1" ;;
        -h | --help )    usage && exit 1 ;;
        * )              BUILD_CMD="$BUILD_CMD $1"
    esac
    shift
done

if [ "" != "$REMOTE_MACHINE" ]; then
    RSYNC="rsync --quiet -az --delete --cvs-exclude"
    RSYNC_UPLOAD="$RSYNC --delete-excluded --exclude target/"
    RSYNC_DOWNLOAD="$RSYNC --include '**/target/**' --exclude '.*'"
    REMOTE_DIR=~ /test

    $RSYNC_UPLOAD ./ $REMOTE_MACHINE:$REMOTE_DIR

    ssh -q -t $REMOTE_MACHINE "cd $REMOTE_DIR ; bash -ic '$BUILD_CMD'"

    $RSYNC_DOWNLOAD $REMOTE_MACHINE:$REMOTE_DIR ./
else
    $BUILD_CMD
fi
```

1 Process command-line arguments

2 Determine whether to build remotely

3 Copy local files to server

4 Run build on server

5 Copy build results from server

Default to local build

The script in listing 9.9 would parse the provided arguments ❶ and determine whether to build locally or remotely ❷ (and, if remotely, on what server). If a remote build was called for, the script would first copy the local files over to the server ❸, then invoke Maven on the server ❹, and finally download the build results ❺ to the developer's local filesystem. The overhead of this upload and download over the network was less than a minute in the first build of the day and the rest of the day it was just a few seconds.

Effect on overall build time? What used to be 24 minutes was now less than 4 minutes, thanks to running the build in parallel threads with the most powerful compute unit Amazon had available.¹⁰ Increased cost? Around \$5 per day per developer. Not a bad deal for crunching your build time from 24 minutes to under 4.

The beauty of this story is that though we couldn't upgrade our laptops in order to get faster CPUs, we could source virtual servers from a cloud computing provider like Amazon. Our solution was essentially a bag of home-grown shell scripts for the server allocation, workspace sync, and remote commands. Though that's doable, there may be an easier way for you to consider. That easier way happens to open the door to using *multiple* remote computers instead of just one. Why don't we go ahead and see if that looks as good as it sounds?

9.3.4 *Distribute the build*

If the biggest leverage for most builds is to use a faster CPU, how about using *seventeen* faster CPUs? That may be overkill for most projects but the notion of using more than one computer (and all of their cores) to do the heavy lifting for your builds does warrant serious thought. Let's start that train of thought by breaking down the problem: what do you need to figure out in order to run your builds remotely with multiple computers?

First, running a build remotely requires that you duplicate the necessary code, data, and other resources to the remote computer you'd like to harness for your personal compute farm. Second, you need to find a way to trigger commands to be executed on that computer and to download the logs and output files back to the developer's computer. In the project illustrated by figure 9.7 we did all of this with some Ruby and shell scripts.

That's not all there is to distributing builds, when it comes to using multiple computers. You also need to find a way to parallelize the build. And in order to parallelize you need to decide how to slice and dice the work so that you can hand out individual slices to a bunch of random computers that don't know about each other.

SLICING THE BUILD INTO DISTRIBUTABLE PARTS

Let's have a look at figure 9.8 to help set some context for this problem.

The figure depicts the essential steps in a build process. Everything starts from a set of source files (code, images, data files, and so on), which are compiled into byte

¹⁰ If memory serves me correctly, it was an 8-core CPU back then.

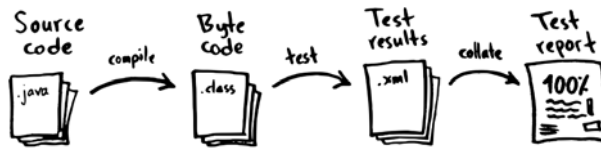


Figure 9.8 A build compiles source code into byte code, runs tests to produce test results, and collates test results into a report.

code. With the compiled code, you can then run tests, producing a set of test result documents. Finally, those test result documents can be collated into a single aggregate report of the whole build’s test results.

So how do you partition this workflow into slices that you can do in parallel? You’ve already seen ways to parallelize a local build. Given that you have the necessary shell scripts for pushing the whole build to be executed on a remote server, it’s not a huge step from our Ant script in listing 9.7 to pushing each of those partitions of your test suite to remote computers.

Similarly, because the test result files produced by Ant’s and Maven’s standard test runners are uniquely named, it’s almost trivial to download whatever output each build produced into a single folder on the developer’s computer for the final step of generating a test report.

What’s more of a challenge is the compilation step. Slicing compilation is difficult because you can’t just randomly compile 50% of the source files on one computer and the other 50% on another computer. The code has dependencies, which means that your slicing algorithm would need to be much more sophisticated. In practice, you’d need to analyze the dependencies and structure your code so that you can compile the shared interfaces first and compile the remaining modules in parallel against the byte code of the shared interfaces.¹¹

Luckily, for most Java projects the compilation step is much faster than test execution, so you’re probably okay with performing the compilation all in one go, on one computer, and focusing your attention on parallelizing and distributing the test execution.

DISTRIBUTED TEST EXECUTION WITH GRIDGAIN

The good news when it comes to distributed computing is that it’s no news—the software development industry has spent decades finding ways of making computers talk to each other. Particularly interesting developments for Java programmers are those that involve the JVM, and any technology that’ll distribute your computation with as little code as possible is sure to get our attention.

One technology that matches that description is *GridGain*.¹² GridGain is an open source middleware product that (among other things) lets you distribute workload on the *grid*: an arbitrary number of computers in the network. In other words, you could set up your own AWS-style cloud computing farm to your local network at the office.

¹¹ Modularizing your code base may be a good idea regardless of whether or not you parallelize the build.

¹² Read more about GridGain at <https://github.com/gridgain/gridgain>.

There's good and bad news about GridGain's offering. The good news is that GridGain provides utilities specifically geared at distributing JUnit tests. The bad news is that those utilities make you jump through hoops to set up a test suite.

To understand those hoops, let's look at the following listing, which shows an example of a test suite annotated to run with GridGain's `GridJunit4Suite` runner.

Listing 9.10 GridGain currently supports a test suite with three fixed partitions

```
import org.gridgain.grid.test.junit4.GridJunit4Suite;

@RunWith(GridJunit4Suite.class)

@SuiteClasses({
    FixedDistributedTestSuite.Partition1.class,
    FixedDistributedTestSuite.Partition2.class,
    FixedDistributedTestSuite.Partition3.class
})

public class FixedDistributedTestSuite {

    @RunWith(Suite.class)
    @SuiteClasses({
        com.manning.PresentationEventTest.class,
        com.manning.PresentationSessionTest.class,
        com.manning.PresentationListTest.class,
        com.manning.PresentationTest.class
    })
    public static class Partition1 { }

    @RunWith(Suite.class)
    @SuiteClasses({
        com.manning.DownloadTest.class,
        com.manning.DownloadItemTest.class,
        com.manning.ProductEqualsTest.class,
        com.manning.ProductTest.class
    })
    public static class Partition2 { }

    @RunWith(Suite.class)
    @SuiteClasses({
        com.manning.HttpDownloaderTest.class,
        com.manning.SocketListenerTest.class,
        com.manning.TcpServerTest.class
    })
    public static class Partition3 { }
}
```

1 Use GridGain
for distributed
execution

2 Enumerate
partitions in this
distributed suite

3 Enumerate test
classes
belonging to
each partition

The test class in listing 9.10 is a fairly standard JUnit test suite class. What makes it somewhat unusual is the `@RunWith` ❶ annotation that instructs JUnit to hand over to GridGain's distributed runner. The other thing that's unusual here is that the suite consists of three test classes described as inner classes of the top-level suite ❷ for the developer's convenience.

The inner classes define three fixed partitions of your test classes ❸. Because we (currently) can't do this programmatically at runtime, we've had to manually split our tests into the partitions and explicitly call them out one by one. Certainly not

something we'd like to do but, for now, the best we can do with GridGain's current limitations.

As of this writing, GridGain only supports the standard Suite runner, which means that you can't use utilities such as JUnit-DynamicSuite for programmatically collecting test classes to execute.¹³

Running `FixedDistributedTestSuite` from listing 9.10 doesn't do anything special yet. It'll just run all of your tests locally on your own computer. The missing piece you still need in order to distribute those tests' execution is to set up the *grid* and line up some worker nodes to do the heavy lifting for you.

GridGain comes with sample configuration and startup scripts for launching nodes for distributed test execution. The output from running the startup script, `bin/ggjunit.sh`, is shown next.

Listing 9.11 GridGain nodes are started with a shell script.

```
$ bin/ggjunit.sh
GridGain Command Line Loader, ver. 3.6.0c.09012012
2012 Copyright (C) GridGain Systems

[01:46:08]      _____
[01:46:08] /  ____/____( )____/ /  ____/_____( )____
[01:46:08] / ( _ // _// _// _// ( _ // _// _// _// _//
[01:46:08] \____//_/ /_/ \_,_/\____/ \_,_//_///_/
[01:46:08]
[01:46:08] -----+ REAL TIME BIG DATA +-----
[01:46:08]                ver. 3.6.0c-09012012
[01:46:08] 2012 Copyright (C) GridGain Systems
[01:46:08]
[01:46:08] Quiet mode.
[01:46:08] << Community Edition >>
[01:46:08] Daemon mode: off
[01:46:08] Language runtime: Java Platform API Specification ver. 1.6
[01:46:08] GRIDGAIN_HOME=/usr/local/gridgain
[01:46:09] Node JOINED [nodeId8=c686a6b7, addr=[192.168.0.24], CPUs=2]
[01:46:11] Topology snapshot [nodes=4, CPUs=2, hash=0xDE6BBF3C]
[01:46:11] Topology snapshot [nodes=2, CPUs=2, hash=0x840EB9D0]
[01:46:11] Node JOINED [nodeId8=fc226dc5, addr=[192.168.0.24], CPUs=2]
[01:46:11] [...]
[01:46:11] Local ports used [TCP:47102 UDP:47200 TCP:47302]
[01:46:11] GridGain started OK
[01:46:11] ^-- [grid=junit, ..., CPUs=2, addrs=[192.168.0.24]]
[01:46:11] ZZZzz zz z...
```

Running the `bin/ggjunit.sh` startup script on a few servers on your network to set up some worker nodes, you could now run your test suite from listing 9.10 and see your tests be executed on several of your grid's nodes in parallel:

```
[01:46:19] Node JOINED [nodeId8=fc226dc5, addr=[192.168.0.24], CPUs=2]
[01:46:19] Topology snapshot [nodes=4, CPUs=2, hash=0xAF0C4685]
```

¹³ Read more about JUnit-DynamicSuite and how to create a dynamic suite of JUnit Tests from a directory at <https://github.com/cschoell/Junit-DynamicSuite>.


```
Distributed test started: equality(ProductTest)
Distributed test finished: equality(ProductTest)
Distributed test started: addingPresentations(ProductTest)
Distributed test finished: addingPresentations(ProductTest)
Distributed test started: removingPresentations(ProductTest)
Distributed test finished: removingPresentations(ProductTest)
```

You already learned about GridGain’s current limitation regarding the supported JUnit runner (Suite). Unfortunately, that’s likely not the only limitation you’ll stumble on. GridGain will likely face challenges in running tests that look for resources from the classpath, for example, not to mention files in the filesystem. For running pure unit tests in a distributed manner on several computers, GridGain is the bee’s knees.

In summary, GridGain’s system is the leading edge of infrastructure for distributed test execution. Yet, it has its limitations that you might find prohibitive. That’s not to say that you should throw away all hope if GridGain doesn’t quite fit the bill. You can always set up our distributed build farm with your good old shell scripts. After all, if you’re looking to speed up your build, there are few approaches with as much potential as using multiple high-powered computers instead of just one!

9.4 **Summary**

This chapter was all about speeding up test execution and getting the feedback on whether anything is broken as fast as possible.

We started our exploration by looking at how you can quickly pinpoint the slow bits—the hotspots where the potentially big speed-ups are hiding. The key word here is *profiling* and we saw a number of tools for surfacing hard facts about what’s taking the most time in your build.

As we moved into examining your test code to see if there’s anything on that front you could optimize, we walked through the most common culprits for slowness: threads sleeping longer than needed, unnecessary code being executed due to a bloated inheritance hierarchy, tests that aren’t isolated enough, tests that talk to the network, and tests that read or write to the filesystem.

We also identified ways of finding speed-ups by changing the way you build. For example, switching a rotating hard drive to a solid-state drive or a RAM disk might give your I/O a boost. As far as the CPU-bound parts of your test execution go, you could parallelize your test suite to execute the tests concurrently on all of your CPUs. Going even further, you could distribute your test runs to remote computers.

All of these options represent potential improvements in your build times. Whether you’ll experience a speed-up or not is context-dependent. The best you can do is to try it and see. Profile your build to understand it better and then try the most promising tricks one by one to see how big of an impact they have.

This chapter also concluded the third and last part of the book. Part 1 introduced the suggested benefits and power of automated unit tests, explored the various aspects that make a unit test a good unit test, and gave you some pointers to using test doubles effectively. Part 2 jumped right into a catalog of test smells—patterns that should

tingle your spidey sense because they often suggest a refactoring is due. Part 3 took you on a real world tour. We started from a discussion of what makes for a testable design. From there we jetted across the globe to visit the world of alternative JVM languages, playing with the idea of writing your tests in a language other than Java. Finally, we took a dive to the deep end of reality: what to do when you realize that your build is slower than you'd like.

Now it's time to get to work and start improving the tests you write. Make those tests better, faster. Make them more meaningful. Make them more useful. It's not going to happen overnight, even if you now know much more about tests and what makes them good. Luckily, even the smallest change for the better can have a big impact on your happiness at work.

Good luck!

appendix A

JUnit primer

In the Java ecosystem, the de facto unit-testing framework today is JUnit. Year after year, you're less and less likely to meet a Java programmer who hasn't at least seen JUnit test code. Nevertheless, there's always the first time for everybody and some of you may have been using a different test framework, so we've included this short appendix as a quick start to writing tests with JUnit.

There are two elemental things to understand about JUnit. First, you must understand the structure and lifecycle of JUnit test code. This is where we'll start. We'll look at how to define *test methods* inside *test classes* and then familiarize you with these tests' lifecycle—how and when JUnit instantiates and invokes your test classes and their methods.

The second elemental thing about JUnit is its `Assertion` API. The basic and most frequently used assertion methods are simple and you'll know how to use them just by looking at their method signatures. That's why we'll only call those methods out by name and focus your attention on the more “opaque” assertions that aren't as self-explanatory.

Overall, JUnit is a small and simple framework and I have no doubt that you'll be running with it quickly. Eventually you'll stumble on situations where you reach out for assistance and that's where dedicated books on JUnit, such as Manning Publications' excellent *JUnit in Action (2nd edition)*, come in handy. Until then, I hope that this appendix is all that you need to get going and keep up with the rest of the book.

A.1 A basic JUnit test class

In short, JUnit test classes are plain old Java classes that have one or more *test methods* and zero or more *setup and teardown methods*. JUnit also defines a simple lifecycle for executing the tests it finds.

The following sections will walk you through these fundamental elements of JUnit test classes one by one, starting with declaring test methods.

A.1.1 Declaring test methods

JUnit tests in their most basic incarnation are regular instance methods marked with a certain annotation. These methods exercise the code they're testing and make assertions about expected conditions and side effects, making use of APIs provided by JUnit. The following listing shows an example of what a simple JUnit test class might look like.

Listing A.1 Test methods annotated with `@org.junit.Test`

```
import org.junit.Test;
import static org.junit.Assert.assertEquals;

public class ExampleTest {
    @Test
    public void thisIsATestMethod() {
        assertEquals(5, 2 + 3);
    }

    @Test
    public void thisIsAnotherTestMethod() {
        WorldOrder newWorldOrder = new WorldOrder();
        assertFalse(newWorldOrder.isComing());
    }
}
```

The listing features a simple JUnit test class with two test methods. If JUnit finds a method that's something other than public, it'll ignore it. If it finds a method that takes arguments, it'll ignore it. If a method returns something other than void, it'll ignore it. If a method is declared static, it'll ignore it. If a method doesn't have a JUnit `@Test` annotation, it'll ignore it.

It's simple, really: a test method must be public void and take no arguments.¹

Now let's look at the lifecycle these tests will go through once JUnit grabs hold of our test class.

A.1.2 JUnit test lifecycle

Pointed to a class like the one in listing A.1, JUnit scans for methods with a signature that matches the preceding constraints and then proceeds to

- 1 Instantiate a fresh instance of the test class
- 2 Invoke any setup methods on the test class instance
- 3 Invoke the test method
- 4 Invoke any teardown methods on the test class instance

JUnit does this for all of the test methods it finds on the test class, including any test methods that were inherited from superclasses. Similarly, the setup and teardown methods may be declared by the test class being run or inherited from its superclasses.

A test is considered to pass if it doesn't throw an exception. For instance, if their conditions aren't fulfilled, the `assertEquals()` and `assertFalse()` assertions in listing A.1 will fail the test by raising an exception. Similarly, if one of the setup or teardown methods throws an exception, that particular test is considered to have failed and reported accordingly.

¹ A test method can declare thrown exceptions. That's okay.

A.1.3 Test setup and teardown

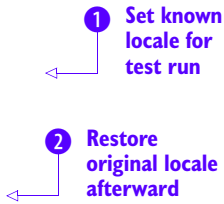
Speaking of setup and teardown, let's look at a small example illustrating their use.

Listing A.2 @Before and @After annotations mark setup and teardown methods

```
public class ExampleTest {
    private Locale originalLocale;

    @Before
    public void setLocaleForTests() {
        this.originalLocale = Locale.getDefault();
        Locale.setDefault(Locale.US);
    }

    @After
    public void restoreOriginalLocale() {
        Locale.setDefault(originalLocale);
    }
}
```



The listing gives an example of a test class ensuring that the active locale **1** during the test execution is `Locale.US` and making sure that the original locale is restored afterward **2**. We're big fans of the Boy Scout Rule² so we want our tests to leave everything in the same condition they were (or better) before the test's execution.

A test class can have any number of these `@Before` and `@After` methods; JUnit will make sure to invoke each of them, although it doesn't guarantee any specific order in which it'll make those invocations.

There are also annotations called `@BeforeClass` and `@AfterClass`, which are somewhat similar except that they're run only once per test class rather than once per test method. These are handy if you want to do something just once before any of the tests in this class are run or after all the tests in this class are run.³

A.2 JUnit assertions

As you've seen in listing A.1, JUnit provides a set of common assertions as public static methods in the `org.junit.Assert` class. It's a common practice to make static imports for these methods to make their use more concise.

There are assertions for checking whether two objects are equal, checking whether an object reference is null, whether two object references point to the same object, whether a condition is `true` or `false`, and so forth. The full API of available assertions can be perused online at www.junit.org but here's a quick overview:

- *assertEquals*—Asserts that two objects (or primitives) are equal.
- *assertArrayEquals*—Asserts that two arrays have the same items.
- *assertTrue*—Asserts that a statement is true.

² See "The Boy Scout Rule" at O'Reilly commons, contributed by Uncle Bob, Nov. 24, 2009, <http://mng.bz/Cn2Q>.

³ Personally, I've used them mostly in the context of integration testing to make sure that a server component is up and running before I start making network connections to it.

- *assertFalse*—Asserts that a statement is false.
- *assertNull*—Asserts that an object reference is null.
- *assertNotNull*—Asserts that an object reference is not null.
- *assertSame*—Asserts that two object references point to the same instance.
- *assertNotSame*—Asserts that two object references do not point to the same instance.
- *assertThat*—Asserts that an object matches the given conditions (see section A.2.2 for a more thorough explanation).

There are a couple of special cases that warrant a special treatment, so let's take a quick peek at those. First, let's check for exceptions you *expect* to be thrown.

A.2.1 Asserting that an exception is thrown

Sometimes the behavior you want your code to exhibit is to raise an exception under certain circumstances. For instance, you might want an `IllegalArgumentException` to be thrown upon invalid input. If the behavior you want is an exception and JUnit interprets exceptions thrown from test methods to suggest test failure, how do you do test for such expected exceptions?

You could do a try-catch inside your test method to catch the expected exception (and fail the test if no exception was thrown) but JUnit provides an even more convenient way of doing this, shown here.

Listing A.3 The `@Test` annotation allows us to declare expected exceptions

```
@Test(expected = IllegalArgumentException.class)
public void ensureThatInvalidPhoneNumberYieldsProperException() {
    FaxMachine fax = new FaxMachine();
    fax.connect("+n0t-a-ph0n3-Numb3r"); // should throw an exception
}
```

In the listing we've used the `expected` attribute of the `@Test` annotation to declare that we *expect* an `IllegalArgumentException` to be thrown when this test method is executed. If such an exception isn't thrown (or some other exception is thrown), the test has failed.

This is a concise way of checking for expected exceptions. But sometimes you want to be more specific about what kind of an exception is thrown. For instance, aside from the thrown exception being of a certain type, you might want to check that it carries specific information in its "message" or that it encapsulates a specific "root cause" exception.

In these situations you need to fall back to the good old try-catch and make those assertions yourself. The next listing shows a variation of our previous example that also checks that the thrown exception mentions specifically what the invalid argument is.

Listing A.4 Using a try-catch lets us examine the thrown exception

```

@Test
public void ensureThatInvalidPhoneNumberYieldsProperException() {
    String invalidPhoneNumber = "+n0t-a-val1d-ph0n3-Numb3r";
    FaxMachine fax = new FaxMachine();
    try {
        fax.connect(invalidPhoneNumber);
        fail("should've raised an exception by now");
    } catch (IllegalArgumentException expected) {
        assertThat(expected.getMessage(),
                    containsString(invalidPhoneNumber));
    }
}

```

Catch exception we expect →

← **Fail test if no exception was thrown**

← **Make further assertions about exception**

This approach gives us the full power of Java and JUnit for asserting whatever it is that we're looking to check. It does get more verbose, though, and it's easy to forget to add that call to `fail()` so if you don't really need to check anything else than that the right type of exception is thrown, go with the annotation-based approach as it's much more clean and concise.

Speaking of concise versus powerful assertions, when the built-in assertions of `org.junit.Assert` don't quite do it for you there's always the option of extending the power of `assertThat()` with custom matchers. Let's take a look at how `assertThat()` and Hamcrest matchers do their magic.

A.2.2 *assertThat()* and Hamcrest matchers

The odd one out among the assertion methods provided by `org.junit.Assert` is `assertThat()`. It's a hook that allows programmers to extend the available basic assertions with their own or with third-party libraries of *matchers*.

The basic syntax is this:

```
assertThat(someObject, [matchesThisCondition]);
```

In other words, the first parameter is the object or value that's the context of the assertion you're making and the second parameter is the matcher which JUnit delegates the actual work of asserting a condition.

These matchers aren't just any regular objects, but Hamcrest matchers. Hamcrest (<https://github.com/hamcrest/JavaHamcrest>) is an open source API and it comes with its own standard collection of matcher implementations that you can use in your tests.

If none of the built-in assertions or Hamcrest matchers are sufficient for what you're trying to express, you can create your own matchers by implementing Hamcrest's `Matcher` interface. Say you want to assert that a particular string looks like a valid international phone number. In this situation, a rudimentary custom Hamcrest matcher might turn out like the following listing.

Listing A.5 Using a custom Hamcrest matcher with `assertThat()`

```

import static org.hamcrest.CoreMatchers.is;
import static org.hamcrest.CoreMatchers.not;
import static org.junit.Assert.assertThat;

import org.hamcrest.BaseMatcher;
import org.hamcrest.Description;
import org.hamcrest.Matcher;
import org.junit.Test;

public class ExampleTest {
    ...

    @Test
    public void providesPhoneNumberInInternationalFormat() {
        String phoneNumber = widgetUnderTest.getPhoneNumber();
        assertThat(phoneNumber, is(internationalPhoneNumber()));
    }

    public Matcher<String> internationalNumber() {
        return new BaseMatcher<String>() {

            /**
             * ITU-T E.123 requires that international phone numbers
             * include a leading plus sign, and allows only spaces to
             * separate groups of digits.
             */
            @Override
            public boolean matches(Object candidate) {
                if (!(candidate instanceof String)) {
                    return false;
                }
                return ((String) candidate)
                    .matches("^\\+(?:[0-9] ?){6,14}[0-9]$");
            }

            @Override
            public void describeTo(Description desc) {
                desc.appendText("ITU-T E.123 compliant " +
                    "international phone number");
            }
        };
    }
}

```

1 Extend BaseMatcher

2 Accept only Strings

3 Match against regular expression

Describe what we expect

When our test invokes `assertThat(phoneNumber, is(internationalPhoneNumber()))` JUnit uses the `BaseMatcher` **1** our helper method has built to carry out the assertion.

Our custom matcher's implementation consists of two methods. The responsibility of `matches()` is to figure out whether a given candidate object passes the assertion. In this example our algorithm is almost trivial: check that the object is indeed a `String` **2** and check that it matches a regular expression **3** for an international phone number.

The second method in our custom matcher, `describeTo()`, is responsible for providing a sensible description of what this matcher is looking for. If the assertion fails, JUnit prints a message saying something like this:

```
java.lang.AssertionError:  
Expected: is ITU-T E.123 compliant international phone number  
got: "+1 (234) 567-8900"
```

This is handy when a test suddenly fails, as it explains what you were looking for and what you got instead. Well, in this case we'd probably have to look at the matcher implementation to know what the requirements for a ITU-T E.123 compliant international phone number is—unless this is an area of the code base we're actively working on.

We've now covered the basic elements of JUnit. With this knowledge you should do fine for some time. When you eventually do bump into a situation where you need something more, refer to appendix B, which explains how to extend JUnit beyond custom matchers.

appendix B

Extending JUnit

Before JUnit reached its venerable 4.0 version, the API was based on your test class inheriting from `junit.framework.TestCase`, and extending JUnit was done by overriding parts of the inherited behavior. With the annotation-based approach adopted by the latest version of JUnit, extensions have a new shape and are hooked up through—you guessed it—annotations.

The main concepts relevant for extending the built-in JUnit behavior are *runners* and *rules*. Though we’re not going to delve too deeply into writing custom extensions, you should be aware of what kind of extensions come built-in with JUnit and where to start when you do want to write a custom extension.

We’ll begin with a brief look at how the runners work. Implementing custom runners isn’t trivial and partly for that reason most of the time we opt for a custom rule instead. Keeping that in mind, the majority of this appendix focuses on how JUnit’s rules work and what rules are at your disposal out of the box.

B.1 Controlling test execution with runners

As we mentioned, when JUnit can’t see an explicit `@RunWith` annotation on your test class, it defaults to a default implementation. But what are these runners and what do they do?

When somebody tells JUnit to “run tests” in a given class, JUnit pulls out a tiny napkin from its back pocket. The napkin contains a list of known ways to identify and run the tests in a given class. Those ways are represented by implementations of the `org.junit.runner.Runner` interface. One by one, JUnit walks through the list until it finds a runner capable of handling the class at hand. For example, on top of the list is a builder that can handle classes with the `@Ignore` annotation. Further down on the list there’s a runner that can handle classes with an explicit `@RunWith` annotation. And at the bottom is the default runner, which is what you mostly end up using. Once JUnit selects one of these runners, it delegates the actual running of tests to that runner, as shown in figure B.1.

From the perspective of extending JUnit, we’re mostly interested in the second type of runner, the one that makes use of `@RunWith`. With the `@RunWith` annotation you can dictate how JUnit treats your test class. You could opt for one of the built-in runners such as `Suite` or `Parameterized` (we’ll get to this little beast later) or you could tell JUnit to use one of your own: a class that implements the abstract `Runner` class.

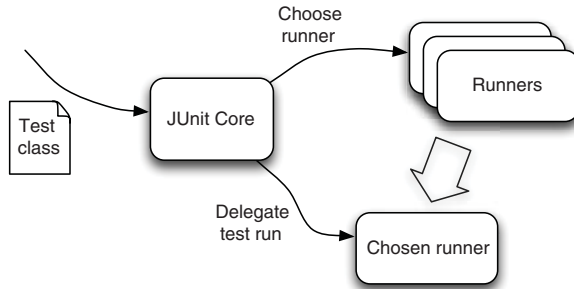


Figure B.1 Given a test class, JUnit figures out which runner implementation to use. One of those runners responds to the `@RunWith` annotation, if present, and delegates test execution to the user-defined implementation of `org.junit.runner.Runner`.

Though writing your custom runner is the single most powerful way to extend JUnit, it can also be more than what you bargained for. Luckily, most of the extension needs for JUnit don't actually warrant an all-new runner. If you only need to modify how individual tests are treated rather than dictate how tests are identified in the first place, perhaps you should first look into using or implementing a *rule* rather than creating a custom runner.

B.2 Decorating tests with rules

Rules are a recent addition to JUnit. They provide the ability to manipulate the execution of tests. A rule implementation might, for example, skip the whole execution or perform some setup or teardown before and after running the test. Rules are applied at class-level and you can apply multiple rules to a single test class. In the case of multiple rules, they're applied one at a time. (JUnit doesn't make any guarantees about the order in which these rules are applied.)

When JUnit looks at a test class, it creates an execution plan for running each test it finds. Rules, which are implementations of `org.junit.rules.MethodRule`, essentially wrap or replace the current execution plan. To clarify these mysterious do-gooders, let's take a look at some of the built-in rules.

STARTING POINT FOR YOUR OWN RULES When you decide to write your own JUnit expressions, you'll likely end up extending `org.junit.rules.TestWatchman` so that's a good place to start reading the JUnit source code.

B.3 Built-in rules

JUnit doesn't just provide the rules API as an extension point for its users: a number of essential JUnit features have been implemented using the same API, including a global timeout for a whole test class, a more sophisticated way of dealing with expected exceptions, and management of temporary filesystem resources. Let's start the tour by looking at how you can configure a global timeout for all tests in a class.

B.3.1 Setting a global timeout

The rules API gives you the option of defining a timeout just once for all tests in the class. This listing shows an example of such a global timeout:

Listing B.1 Specifying a global timeout for all test methods in the class

```
import org.junit.Test;
import org.junit.Rule;
import org.junit.rules.Timeout;

public class GlobalTimeoutTest {
    @Rule
    public MethodRule globalTimeout = new Timeout(20);

    @Test
    public void infiniteLoop1() {
        while (true) { }
    }

    @Test
    public void infiniteLoop2() {
        while (true) { }
    }
}
```

1 Rules are annotated with @Rule.

2 Rules are defined as public fields.

Rules are defined as public fields.

There are three things to note about listing B.1:

- We define a public field 2 of type `MethodRule`.
- The field is annotated with `@Rule` 1.
- The name of the field doesn't matter to JUnit so you can name your rule however you want. Essentially the name should describe what the rule is for.

In the listing, JUnit applies the `Timeout` rule to both test methods, interrupting both when they exceed the configured timeout of 20 milliseconds.

B.3.2 Expected exceptions

JUnit's `@Test` annotation accepts an `expected=` attribute for failing the test unless it throws an exception of the specified type. There is another, slightly more sophisticated way of checking for the expected exception's properties such as the exception message, root cause, and so forth. (The less sophisticated way is to catch the exception with a try-catch block and interrogate the exception object right then and there.) The sophisticated way is the `ExpectedException` rule.

Let's look at an example to clarify its usage.

Listing B.2 Example usage of the `ExpectedException` rule

```
import org.junit.Test;
import org.junit.Rule;
import org.junit.rules.ExpectedException;
```

```

public class ExpectedExceptionTest {
    @Rule
    public ExpectedException thrown = ExpectedException.none();

    @Test
    public void thisTestPasses() { }

    @Test
    public void throwsExceptionAsExpected() {
        thrown.expect(NullPointerException.class);
        throw new NullPointerException();
    }

    @Test
    public void throwsExceptionWithCorrectMessage() {
        thrown.expect(RuntimeException.class);
        thrown.expectMessage("boom");
        throw new NullPointerException("Ka-boom!");
    }
}

```

ExpectedException rule is initialized to "none". ①

② Expect NullPointerException to be thrown

③ Expect exception containing "boom."

In the listing, we have one rule, `ExpectedException`, and three tests. In its initial state, the rule expects no exception ① to be thrown. That's why the first test passes even though rules apply to all test methods in the class they're attached to. In other words, we need to *configure the rule* and tell it what kind of an exception we expect. And that's exactly what we do in the other two tests.

In the second test, we tell the rule to expect a `NullPointerException` ② to be thrown from the test. "If you don't see this type of an exception thrown, please fail this test." What we have here is essentially the same functionality provided by `@Test(expected=NullPointerException.class)` so this isn't that exciting yet. The real value of the `ExpectedException` rule begins to show in the third test.

In the third test, we're not just telling the rule to expect an exception of a certain type. With `expectMessage("boom")`, we're also telling it to verify that the exception's message contains a specific substring—in our case, the word *boom* ③ (case-sensitive). Note that this time we told `ExpectedException` to expect a `RuntimeException` and the test actually throws a `NullPointerException`, a subclass of `RuntimeException`. This test passes because the check performed by the rule is that the thrown exception walks like a duck, not that it actually is a duck.

CHECKING THE ROOT CAUSE

Now you know how to test for an exception to be thrown, that it's of the expected type, and that its message contains the expected bits. Sometimes, you want the thrown exception to wrap a specific *cause*. The following listing shows an example of applying custom logic for determining whether the thrown exception matches our expectations.

Listing B.3 Testing arbitrary details about an expected exception

```

import org.hamcrest.BaseMatcher;
import org.hamcrest.Description;
import org.hamcrest.Matcher;

```

```

import org.junit.Rule;
import org.junit.Test;
import org.junit.rules.ExpectedException;

public class ExceptionWithExpectedRootCauseTest {
    @Rule
    public ExpectedException thrown = ExpectedException.none();

    @Test
    public void throwsExceptionWithCorrectCause() {
        thrown.expect(rootCauseOf(NullPointerException.class));
        throw new RuntimeException("Boom!", new NullPointerException());
    }

    private Matcher<? extends Throwable> rootCauseOf(
        final Class<? extends Throwable> expectedCause) {
        return new BaseMatcher() {
            @Override
            public boolean matches(Object candidate) {
                if (!(candidate instanceof Throwable)) {
                    return false;
                }
                Throwable cause = ((Throwable) candidate).getCause();
                if (cause == null) {
                    return false;
                }
                return expectedCause.isAssignableFrom(cause.getClass());
            }
        };
    }

    @Override
    public void describeTo(Description description) {
        description.appendText("Throwable with cause of type ");
        description.appendText(expectedCause.getSimpleName());
    }
}

```

1 Pass custom Hamcrest matcher to ExpectedException.

2 Create subclass of BaseMatcher.

3 Does thrown exception match our expectations?

In the listing you can see two aspects of JUnit you haven't seen so far. First of all, we're using a different, overloaded version ❶ of `ExpectedException#expect()`. Second, by using this overloaded method we're using something called Hamcrest matchers.

Hamcrest (<http://code.google.com/p/hamcrest>) is an open source library that provides a library of *matcher objects* or *predicates*. JUnit uses Hamcrest's interfaces in its own API, listing B.3 being just one example of such integration.

Let's look at the two variants of `ExpectedException#expect()`:

```

void expect(Class<? extends Throwable> exceptionType)
void expect(org.hamcrest.Matcher<?> matcher)

```

With the first method variant we're saying, "Please check that the thrown exception is of this type." With the latter method variant we're saying, "Please use this predicate to check that the thrown exception is what we expect." What logic that predicate object uses to make this distinction is up to the programmer—that's you.

In our example, we're creating an anonymous subclass of `org.hamcrest.BaseMatcher` ❷ that determines the correctness of the thrown exception ❸. There are two abstract methods we need to implement: `matches(Object)` and `describeTo(Description)`. The former encapsulates the logic. The latter gives JUnit a means to form a meaningful error message in case the predicate evaluates to false, when our assertion about the expected exception fails.

BASEMATCHER Write a mental note to yourself about `BaseMatcher` and the sample of a custom matcher in listing B.3. Most of the custom matchers you're going to write (if you need to write them) are going to extend from that class and will likely look similar to our example here.

In listing B.3, the logic dictates that we first ensure that the given candidate object is an exception. Second, we check that it has a cause and, finally, we check that the cause is of the type we expected. If any of these conditions fails, we return `false` and the `ExpectedException` rule fails the running test.

B.3.3 Temporary folders

Most unit tests should stay away from the filesystem but every now and then you want to write an automated, repeatable test that works with actual files on the filesystem. That's where our last featured rule comes into play. `TemporaryFolder` is a simple utility with which you can easily create temporary files and folders from your tests and have them automatically wiped out after the test has run. Let's look at an example.

Listing B.4 JUnit has a built-in `@Rule` for using temporary folders

```
import java.io.File;
import org.junit.Rule;
import org.junit.Test;
import org.junit.rules.TemporaryFolder;

public class TemporaryFolderTest {
    @Rule
    public TemporaryFolder folder = new TemporaryFolder();

    @Test
    public void thisTempFileIsSquashedAfterTheTest() throws Exception {
        File tempFile = folder.newFile("myTempFile.txt");
        assertTrue(tempFile.exists());
    }

    @Test
    public void allJunkWillBeGoneAfterTheTestHasRun() {
        File tempDir = folder.newFolder("myTempDir");
        createRandomJunkIn(new File(tempDir, "mah-junk.txt"));
        createRandomJunkIn(new File(tempDir, "moar-junk.txt"));
    }

    private void createRandomJunkIn(File file) {
        // omitted for brevity
    }
}
```

❶ Create temporary file

❷ Create temporary folder

Again, we declare a rule—this time a `TemporaryFolder`. Inside a test that needs a temporary file (the first test), we ask the rule for a new file ❶ and we get just that. After the test, our rule takes care that the file is deleted. In the second test, we create a new temporary folder ❷, add a couple of files inside that folder, and again `TemporaryFolder` takes care of cleaning them up afterward. Neat!

Symbols

@After 179, 181, 204
@AfterClass 179, 204
@Before 61, 73, 92, 179, 181, 204
@BeforeClass 73, 179, 181, 204
@BeforeClass method 22
@Ignore 126, 209
@Parameters 104
@Rule 211, 215
@RunWith 198, 209
@Test 117, 122, 205, 211
<junitreport/> 175
>> operator 167

Numerics

1K competitions 16

A

ability
 of a test to fail 53
 to configure objects' behavior 44
 See also testability
 to develop software 102
 See also productivity
 to express intent 121
 See also intent
 to find root cause 108
 to get repeatable results 31
 to keep design simple 11
 to maim oneself 181

 to manage information 65
 See also readability
 to manipulate test execution 210
 to omit keywords 168
 to recognize patterns 21
 to rely on tests 9, 20, 23, 85, 115
 to test 26, 32, 43, 142–143, 154
 to verify behavior 47
aborting test execution 131, 134
absolute path 88
abstract
 class 111–112, 180, 209
 method 112, 214
abstraction 19, 30, 142–143, 185
 a missing 154
 level of 56–57, 59–60, 73–76, 102, 108, 129
 missing 32
abuse
 of comments 135
 of inheritance 179
acceptance test 13–14
access
 concurrent 88, 94
 global 148, 150
 restricted 143, 151, 160
 to a database 171
 to dependencies 43
 to fields 10, 32
 to file system 88, 90, 171, 184–186, 189
 to hidden information 27–28, 32, 44, 122

 to network resources 85, 171, 184–185
accidental complexity 12, 69, 113
accuracy. *See* correctness
activation profile 174
algorithm 30–31, 98, 101–102, 128, 197, 207
allocation
 of tests to worker threads 190
 of virtual servers 195–196
alternative
 implementation 36, 185
 JVM languages 157, 160, 164, 168
 to JUnit 164
 to writing tests in Java 157
Amazon Web Services. *See* AWS
analogy 139
analysis 9, 13, 26, 114, 172–174
 of build dependencies 191, 197
annotation 117, 122, 202, 206, 209
anonymous 105–107, 114, 214
answer 13, 33, 79, 139, 165
Ant. *See* Apache Ant
Ant-Contrib 173
 See also Apache Ant
anti-pattern 45, 97
 See also code smells
Apache Ant
 <junitreport/> 175
 Ant-Contrib 173
 custom programming 193

Apache Ant (*continued*)
 parallelizing tests 191
 ProfileLogger 172
 profiling builds 172
 slicing up builds 197
 Apache Maven
 activation profile 174
 maven-build-utils
 extension 173
 maven-surefire-
 plugin 175
 multi-module builds 191
 parallelizing tests 190
 profiling builds 173
 slicing up builds 197
 Surefire 190
 API 32, 35, 37, 49, 159
 changing 42, 82–83, 133,
 152, 159
 File API 86, 89–90, 92
 Hamcrest 206
 Java Persistence API 185
 JUnit Assertion API 202
 JUnit assertions 204
 Mockito vs. JMock 42
 Reflection API 22, 43, 145,
 147
 time-related 25
 time-related issues 86
 archetype 126
 architecture 10, 21, 35, 116,
 185
 arguments 32, 37, 75, 133–134,
 144, 168, 203, 205
 arrange-act-assert 40, 61
See also given-when-then
 article 5, 13, 143, 151
 assertArrayEquals 204
 assertEquals 51, 53, 76, 204
 assertFalse 51, 205
 assertion 47–48
 about exception 122, 214
 bitwise 56
 custom 84, 98, 179
 duplication in 80
 exceptions 205
 failing 84–85, 106–108, 129,
 133–134
 Hamcrest matchers 206
 hyperassertion 51
 missing 24, 76, 121, 124,
 126–127
 one per test 60–62, 128
 primitive assertion 48, 58
 redundant 76

starting with an 126
See also arrange-act-assert
See also bitwise assertion
See also hyperassertion
See also primitive assertion
 assertNotNull 76, 205
 assertNotSame 205
 assertNull 205
 assertSame 205
 assertThat 49–50, 205–206
 assertTrue 49, 204
 assignment 43, 65–67, 70–71,
 161
 assumeTrue 131
 assumptions
 failed 134
 flawed 85
 asynchronous 23, 25
 automation 4
See also test automation
 awareness 6, 9, 95, 132, 209
 AWS 195–197

B

BDD 41, 163
 Dan North 13
 easyb 164
 outside-in development 13
 Spock Framework 165
 behavior 42
 behavior-driven development.
See BDD
 bit operators 56
 bitwise assertion 77
 how to fix 56
 bugs, cost of fixing 9
 build automation 26, 172
 builds
 Apache Ant 172
 Apache Maven 173
 distributing 196
 GridGain 197
 multi-module 191
 offloading to more powerful
 CPU 194
 parallelizing 189
 profiling 172
 slicing up for
 distributing 196
 speeding up 187
 ByteArrayOutputStream 89
 ByteArrayInputStream 89

C

carriage returns 131
 chunking 65
 classes
 abstract 111, 180
 bloated base classes 179
 JUnit test class 202
 that can't be instantiated 144
 class-invariant tests 64
 Clojure 157
See also JVM languages
 code
 altering 122
 dead 117
 formatting 107–108, 117
 making sense through
 structure 18
 structure 18
 under test 28
 code smells 21
 bitwise assertion 56
 commented-out tests 116,
 123
 conditional logic 82
 conditional tests 133
 crippling file paths 88
 duplication 79
 flaky tests 85
 hyperassertion 51
 incidental details 57
 lack of cohesion in
 methods 108
 lowered expectations 127
 magic number 66, 70
 misleading comments 118
 never-failing tests 121
 overprotective tests 75
 parameterized mess 102
 persistent temp files 91
 pixel perfection 97
 platform prejudice 129
 primitive assertion 48, 58,
 68
 setup sermon 72
 shallow promises 123
 sleeping snail 94
 split logic 64
 split personality 60
 collaborators, that can't be
 substituted 145
 commented-out tests 116, 123
 how to fix 117
 commenting, what versus
 how 120

- comments
 - commented-out tests 116, 123
 - delete code, don't comment it out 125
 - good commenting practice 120
 - misleading 118
- complex designs, problem with 12
- complexity, accidental 12
- composition, instead of inheritance 151
- conditional logic 82
 - eliminating 83
- conditional tests 133
 - how to fix 134
- Configuration 61
- constructors, logic in 149
- contains 50
- continuous integration
 - servers 177
- CountDownLatch 96
- createTempFile 92
- crippling file paths 88
 - how to fix 89
- custom assertions 98
- cyclomatic complexity 128

D

- DAO 184–185
- data
 - external data files 69
 - inline 68
- data access objects. *See* DAO
- databases
 - faster replacements 185
 - speed 184
- debugging 3, 8, 84
 - null check in code 76
- defects 9
- defensive programming 75
- delayed feedback 171
- deleteOnExit 92
- demo parties 16
- dependency injection 43
 - in tests 180
- Dependency Inversion
 - Principle 142
- design
 - activity 11, 13
 - altering 44
 - refactoring 11

- design principles 45, 143
 - abiding to 154
 - Dependency Inversion Principle 142
 - Interface Segregation Principle 142
 - Liskov Substitution Principle 142
 - Open-Closed Principle 142
 - Single Responsibility Principle 141
 - SOLID 141, 154
- distributing builds
 - GridGain 197
 - slicing up 196
- duplication 79
 - eliminating 80
 - literal 80
 - semantic 81
 - structural 80

E

- easyb, scenario outlines 164
- else 84
- exceptions 122, 205
 - that you expect 211
 - with a specific cause 212
- expected attribute 123
- expected= attribute 211
- ExpectedException 211
- external data files 69

F

- fail() 84, 122
- fake objects 35
- fakes 25
 - See also* test doubles
- feature methods 166
- feedback cycle 8
- File API
 - createTempFile 92
 - deleteOnExit 92
- file I/O
 - avoiding direct file access 186
 - disabling logging 187
 - intercepting file access 186
- FileInputStream 89
- FileOutputStream 89
- files
 - persistent temp files 91
 - replacing with streams 89
- fixture 73

- flaky tests 85
 - eliminating 86
- flawed assumptions 85
- for 84
- forkmode 192

G

- getProperties 89
- given-when-then 41
 - and easyb 164
 - See also* arrange-act-assert
- global timeout 211
- grep 48
- GridGain 197
- Groovy
 - and Easyb 164
 - and java.lang.Runnable 162
 - and JavaBeans-style interfaces 161
 - and Spock Framework 165
- creating complex objects 161
- declaring variables with def 160
- field values for objects 161
- optional semicolons, parentheses, and return keyword 160
- public by default 160
- rewriting JUnit 4 tests 163
- string interpolation 161
- test setup 161
- unit tests 160
 - See also* JVM languages
- guard classes 75

H

- Hamcrest 49
- Hamcrest matchers 206
 - and JRuby 50, 213
 - and JUnit 50
- happy path tests 124
- Hellesøy, Aslak 13
- Hoare, Sir Tony 79
- hyperassertion 51, 76, 114
 - how to fix 54

I

- IDE 3, 76
 - code comments 117
- if 84

incidental details 57
 how to fix 58
 independent tests 21
 indexOf 50
 IndexOutOfBoundsException 75
 inheritance 112
 use composition instead 151
 inline data 68
 inline logic 68
 InputStream 89
 integrated development environment. *See* IDE
 intent, expressing 50
 Interface Segregation Principle 142

J

Java Persistence API. *See* JPA
 java.util.concurrent 95
 JavaBeans, and Groovy 161
 JavaServer Pages. *See* JSP
 JMock 39, 42
 vs. Mockito 42–43
 JPA 185
 JRuby 57, 65, 157
See also JVM languages
 JSP 18
 JUnit 4
 @After annotation 166, 181, 204
 @AfterClass annotation 182, 204
 @Before annotation 61, 73–74, 179, 181, 204
 @BeforeClass annotation 73–74, 179, 181, 204
 @Ignore annotation 126, 209
 @Parameters annotation 104
 @Rule annotation 212, 214–215
 @RunWith annotation 198, 209
 @Test annotation 117, 122–123, 203, 205, 211
 asserting that exception will be thrown 205
 Assertion API 202
 assertions 204
 assertThat 49
 assumptions 134
 built-in rules 210

expected exceptions 211–212
 extending 209
 global timeout 211
 Hamcrest matchers 50, 213
 Parameterized 104
 parameterized mess 102
 persistent temp files 92
 rewriting JUnit tests in
 Groovy 163
 rules 210
 runners 209
 temporary folders 214
 test class 202
 test lifecycle 203
 test methods 202
 test setup and teardown 204
 JVM languages 156
 benefits of mixing languages 157
 Groovy 160
 mixing 157
 picking the best one 160
 readability 159
 writing tests in 159

K

Knuth, Donald 79

L

lack of cohesion in
 methods 108
 how to fix 110
 Langr, Jeff 70
 languages, mixing 157
 Law of Two Plateaus 7, 10
 legacy code 17
 libraries, external 152
 line feeds 131
 listeners in Ant-Contrib 173
 literal duplication 80
 logging
 disabling 187
 ProfileLogger 172
 logic
 conditional 82
 inline 68
 look 165
 lowered expectations 127
 how to fix 128

M

magic numbers 70–72
 maintainability 78
 Maven. *See* Apache Maven
 maven-surefire-plugin 175
 methods
 lack of cohesion 108
 static 148
 that can't be invoked 144
 that can't be overridden 145
 misleading comments 118
 how to fix 119
 mock objects 38
 JMock 39, 42
 libraries 42
 Mockito 42
 Mockito 38
 separation of stubbing and verification 42
 vs. JMock 42–43
 mocks. *See* test doubles
 modular programming 140
 and test-driven development 143
 multithreading 87

N

never-failing tests 121
 how to fix 122
 new keyword 148
 newline characters 131
 North, Dan 13
 Now 191
 NullPointerException 75

O

ObjectSpace 57
 Open-Closed Principle 142
 org.junit.rules.MethodRule 210
 org.junit.rules.TestWatchman 210
 org.junit.runner.Runner 209
 OutputStream 89
 overprotective tests 75
 how to fix 76

P

parallelizing builds 189
 with Ant 191
 with Maven 190

Parameterized 104
 parameterized mess 102
 how to fix 106
 parameterized test 102, 105,
 107, 114, 166
 performance hit 159, 179
 persistent temp files 91
 eliminating 92
 pixel perfection 97
 how to fix 98
 place 82
 Platform 131
 platform prejudice 129
 how to fix 130
 Platform.current() 131
 polyglot programming. *See* JVM
 languages
 primitive assertion 48, 58, 68,
 76, 97–98, 101
 how to fix 49
 principle
 of good tests 53
 of test doubles 30
 SOLID 53, 73
 Process 133
 Process#runAndWait 133
 productivity
 accidental complexity 12
 curve of design potential 10
 debugging 8
 feedback cycle 8
 ProfileLogger 172
 profiling tests 175

R

Rainsberger, J.B. 43
 RAM disk 188
 random number generators 87
 readability 16, 55, 159
 versus performance 50
 refactoring 11
 reliable tests 23
 Ruby 157
 See also JVM languages
 rules
 built-in 210
 expected exceptions 211–
 212
 global timeout 211
 Hamcrest matchers 213
 org.junit.rules.TestWatchman
 210
 temporary folders 214
 runners 209

S

Scala 157
 pattern matching 158
 See also JVM languages
 scenario outlines 164
 Schwarz, Philip 70
 semantic duplication 81
 service lookups 152
 setLastModified 87
 setup 67
 setup sermon 72, 77
 how to fix 73
 See also test setup
 shallow promises 123
 how to fix 125
 Singleton design pattern 150
 sleep()
 reasons to avoid 178
 sleeping snail 94
 how to fix 95
 SOLID 141
 Dependency Inversion
 Principle 142
 Interface Segregation
 Principle 142
 Liskov Substitution
 Principle 142
 Open-Closed Principle 142
 Single Responsibility
 Principle 53, 73, 141
 special conditions 32
 specs 165
 speed 170
 Amazon Web Services 195
 analysis 172
 Apache Ant 172
 Apache Maven 173
 avoiding direct file
 access 186
 avoiding sleep() 178
 bloated base classes 179
 continuous integration
 servers 177
 DAOs 184
 database access 184
 delayed feedback 171
 disabling logging 187
 file I/O 186
 forking 192
 GridGain 197
 intercepting file access 186
 irrelevant collaborators 182
 offloading to more powerful
 CPU 194

parallelizing builds 189
 parallelizing tests with
 Ant 191
 parallelizing tests with
 Maven 190
 profiling builds 172
 profiling tests 175
 RAM disk 188
 reasons to speed up tests 171
 speeding up builds 187
 speeding up test code 178
 staying local 183
 stubs 183
 tmpfs 189
 split logic 64
 how to fix 66
 split personality 60
 how to fix 61
 Spock Framework 165
 >> operator 167
 feature methods 166
 specifying number of
 invocations 168
 specs 165
 test doubles 167
 static analysis 126, 128
 streams, replacing files with 89
 string interpolation 161
 structural duplication 80
 structure, useful 18
 stubs 25, 183
 See also test doubles
 Surefire 190
 switch 84
 System.currentTimeMillis() 25

T

TDD 6, 11
 and modular
 programming 143
 benefits 11
 cyclical process 12
 desirable consequences 11
 how it begins 11
 recommended reading 12
 temporary folders 214
 test automation 4–6, 10–11,
 19–20, 55, 82, 187, 214
 learning a new language 159
 manage brittleness 79
 protect against defects 17
 test doubles 23, 25, 27
 accessing hidden
 information 32

- test doubles (*continued*)
 - arrange-act-assert 40
 - behavior versus
 - implementation 42
 - dependency injection 43
 - deterministic execution 31
 - fake objects 35
 - given-when-then 41
 - in Spock Framework 167
 - isolating code under test 28
 - mock objects 38
 - picking the right type 40
 - reasons to use 28
 - special conditions 32
 - speed 30
 - test spies 36
 - test stubs 34
 - types of 33
 - when to use 39
- test fixture, lack of cohesion in
 - methods 108
- test lists 125
- test order 22
- test results 105, 193, 197
 - accuracy of 9
 - aggregated report of 196–197
- test setup 59–60, 73–74, 92–93, 161–162, 179–181, 202–203, 210
 - JUnit 204
 - level of abstraction in 73
 - moving code into 72
 - redundant 181
 - See also* setup sermon
- test smell. *See* code smells
- test spies 36
- test stubs 34
- test teardown 166, 179–181, 202–203, 210
 - JUnit 204
- test, alerted by 22, 85, 121
- testable design 139
 - avoid complex private methods 146
 - avoid final methods 147
 - avoid logic in
 - constructors 149
 - avoid service lookups 152
 - avoid Singleton pattern 150
 - avoid static methods 148
 - classes that can't be
 - instantiated 144
 - collaborators that can't be
 - substituted 145
 - defined 140
 - favor composition over
 - inheritance 151
 - guidelines 146
 - methods that can't be
 - invoked 144
 - methods that can't be
 - overridden 145
 - modular programming 140
 - SOLID 141
 - unobservable outcomes 145
 - use new with care 148
 - wrapping external
 - libraries 152
- test-code-refactor 11
- test-driven development. *See* TDD
- test-first programming. *See* TDD
- test-infected 120, 142, 185
 - and mock object libraries 38
 - and test doubles 25
 - becoming 5
 - remedies for duplication 82
- testing
 - 100% coverage 5
 - aborting test execution 131
 - as design tool 10
 - asynchronously 23
 - avoiding sleep() 178
 - BDD 13
 - benefits of mixing
 - languages 157
 - bloated base classes 179
 - classes that can't be
 - instantiated 144
 - code readability 16
 - code structure 18
 - code under test 28
 - collaborators that can't be
 - substituted 145
 - commented-out tests 116
 - conditional tests 133
 - cyclomatic complexity 128
 - database access 184
 - delayed feedback 171
 - dependencies 22
 - Dependency Inversion
 - Principle 142
 - deterministic execution 31
 - diminishing returns 6
 - distributing builds 196
 - feedback cycle 8
 - file I/O 186
 - fixture 73
 - flaky tests 85
 - guidelines for testable
 - design 146
 - happy path tests 124
 - hidden information 32
 - independent tests 21
 - Interface Segregation
 - Principle 142
 - irrelevant collaborators 182
 - JVM languages 156
 - legacy code 17
 - lengthy tests 19
 - Liskov Substitution
 - Principle 142
 - long methods 19
 - maintainability 78
 - methods that can't be
 - invoked 144
 - methods that can't be
 - overridden 145
 - mixing languages 157
 - modular programming 140
 - multiple test classes 62
 - newline characters 131
 - offloading to more powerful
 - CPU 194
 - only one reason to fail 53
 - Open-Closed Principle 142
 - parallelizing tests with
 - Maven 190
 - parallelizing with Ant 191
 - parameterized tests 166
 - picking the right
 - language 160
 - productivity 8
 - profiling 172, 175
 - quality 10, 16
 - readability 8, 159
 - reasons to speed up 171
 - redundant setup and
 - teardown 181
 - reliability 8, 20, 23
 - repeatability 31
 - setup 161
 - setup and teardown 179
 - Single Responsibility
 - Principle 141
 - SOLID 141
 - special conditions 32
 - speed 9, 30, 178, 187
 - splitting into multiple
 - tests 54
 - staying local 183
 - TDD 6
 - test lists 125
 - test order 22

- testing (*continued*)
 - testability issues 143
 - testable design 139
 - the right things 20
 - to catch mistakes 5
 - trustworthiness 8, 115
 - unobservable outcomes 145
 - value of 5
 - with Java 4
 - with JUnit 202
 - writing 159
- tests
 - as a design tool 10–14
 - independent 21–23
 - making faster 30
 - names are important 20
 - productivity influences 8
- reliable 23–25
 - that always fail 53
 - that do nothing 123
 - that fail constantly 85
 - that never fail 121
 - that rarely fail 24
 - value of 5–10
 - what is good 15
 - write with care 23
- Thread#sleep 86, 94
- timestamps 85
- tmpfs 189
- trustworthiness 115

U

- unit tests
 - best value 55
 - suggested reading 4
 - See also* testing
 - why write 5

V

- varargs method 107

W

- while 84
- Wynne, Matt 13

Effective Unit Testing

Lasse Koskela

Test the components before you assemble them into a full application, and you'll get better software. For Java developers, there's now a decade of experience with well-crafted tests that anticipate problems, identify known and unknown dependencies in the code, and allow you to test components both in isolation and in the context of a full application.

Effective Unit Testing teaches Java developers how to write unit tests that are concise, expressive, useful, and maintainable. Offering crisp explanations and easy-to-absorb examples, it introduces emerging techniques like behavior-driven development and specification by example.

What's Inside

- A thorough introduction to unit testing
- Choosing best-of-breed tools
- Writing tests using dynamic languages
- Efficient test automation

Programmers who are already unit testing will learn the current state of the art. Those who are new to the game will learn practices that will serve them well for the rest of their career.

Lasse Koskela is a coach, trainer, consultant, and programmer. He hacks on open source projects, helps companies improve their productivity, and speaks frequently at conferences around the world. Lasse is the author of *Test Driven*, also published by Manning.

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit manning.com/EffectiveUnitTesting



“A fantastic, definitive guide. It will boost your productivity and deployment effectiveness.”

—Roger Cornejo, GlaxoSmithKline

“Changed the way I look at the Java development process. Highly recommended.”

—Phil Hanna, SAS Institute, Inc.

“A common sense approach to writing high quality code.”

—Frank Crow
Sr. Progeny Systems Corp.

“Extremely useful, even if you write .NET code.”

—J. Bourgeois, Freshly Coded

“If unit tests are a nightmare for you, read this book!”

—Franco Lombardo
Molteni Informatica

