DEVELOPER TESTING
This page intentionally left blank
To my grandfather Romuald, who taught me about books.
This page intentionally left blank
# CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Foreword by Jeff Langr</td>
<td>xiii</td>
</tr>
<tr>
<td>Foreword by Lisa Crispin</td>
<td>xv</td>
</tr>
<tr>
<td>Preface</td>
<td>xvii</td>
</tr>
<tr>
<td>Acknowledgments</td>
<td>xxiii</td>
</tr>
<tr>
<td>About the Author</td>
<td>xxv</td>
</tr>
</tbody>
</table>

## Chapter 1  
**Developer Testing**

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Developers Test</td>
<td>1</td>
</tr>
<tr>
<td>Developer Testing Activities</td>
<td>2</td>
</tr>
<tr>
<td>What Developers Usually Don’t Do</td>
<td>5</td>
</tr>
<tr>
<td>Defining Developer Testing</td>
<td>6</td>
</tr>
<tr>
<td>Developer Testing and the Development Process</td>
<td>7</td>
</tr>
<tr>
<td>Summary</td>
<td>8</td>
</tr>
</tbody>
</table>

## Chapter 2  
**Testing Objectives, Styles, and Roles**

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Testing and Checking</td>
<td>9</td>
</tr>
<tr>
<td>Testing Objectives</td>
<td>10</td>
</tr>
<tr>
<td>Testing Styles</td>
<td>11</td>
</tr>
<tr>
<td>Your Quality Assurance and Developer Testing</td>
<td>18</td>
</tr>
<tr>
<td>Summary</td>
<td>19</td>
</tr>
</tbody>
</table>

## Chapter 3  
**The Testing Vocabulary**

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Errors, Defects, Failures</td>
<td>22</td>
</tr>
<tr>
<td>White Box and Black Box Testing</td>
<td>22</td>
</tr>
<tr>
<td>Classifying Tests</td>
<td>23</td>
</tr>
<tr>
<td>The Agile Testing Quadrants</td>
<td>32</td>
</tr>
<tr>
<td>Some Other Types of Testing</td>
<td>33</td>
</tr>
<tr>
<td>Summary</td>
<td>36</td>
</tr>
<tr>
<td>Chapter</td>
<td>Title</td>
</tr>
<tr>
<td>---------</td>
<td>--------------------------------------------</td>
</tr>
<tr>
<td>4</td>
<td>Testability from a Developer’s Perspective</td>
</tr>
<tr>
<td></td>
<td>Testable Software</td>
</tr>
<tr>
<td></td>
<td>Benefits of Testability</td>
</tr>
<tr>
<td></td>
<td>Testability Defined</td>
</tr>
<tr>
<td></td>
<td>Summary</td>
</tr>
<tr>
<td>5</td>
<td>Programming by Contract</td>
</tr>
<tr>
<td></td>
<td>Contracts Formalize Constraints</td>
</tr>
<tr>
<td></td>
<td>Implementing Programming by Contract</td>
</tr>
<tr>
<td></td>
<td>Enforcing Contracts</td>
</tr>
<tr>
<td></td>
<td>Summary</td>
</tr>
<tr>
<td>6</td>
<td>Drivers of Testability</td>
</tr>
<tr>
<td></td>
<td>Direct Input and Output</td>
</tr>
<tr>
<td></td>
<td>Indirect Input and Output</td>
</tr>
<tr>
<td></td>
<td>State</td>
</tr>
<tr>
<td></td>
<td>Temporal Coupling</td>
</tr>
<tr>
<td></td>
<td>Data Types and Testability</td>
</tr>
<tr>
<td></td>
<td>Domain-to-Range Ratio</td>
</tr>
<tr>
<td></td>
<td>Summary</td>
</tr>
<tr>
<td>7</td>
<td>Unit Testing</td>
</tr>
<tr>
<td></td>
<td>Why Do It?</td>
</tr>
<tr>
<td></td>
<td>What Is a Unit Test?</td>
</tr>
<tr>
<td></td>
<td>The Life Cycle of a Unit Testing Framework</td>
</tr>
<tr>
<td></td>
<td>Naming Tests</td>
</tr>
<tr>
<td></td>
<td>Structuring Tests</td>
</tr>
<tr>
<td></td>
<td>Assertion Methods</td>
</tr>
<tr>
<td></td>
<td>Testing Exceptions</td>
</tr>
<tr>
<td></td>
<td>Behavior-driven Development–Style Framework</td>
</tr>
<tr>
<td></td>
<td>Summary</td>
</tr>
<tr>
<td>8</td>
<td>Specification-based Testing Techniques</td>
</tr>
<tr>
<td></td>
<td>Equivalence Partitioning</td>
</tr>
<tr>
<td></td>
<td>Boundary Value Analysis</td>
</tr>
<tr>
<td></td>
<td>Edge Cases and Gotchas for Some Data Types</td>
</tr>
<tr>
<td></td>
<td>State Transition Testing</td>
</tr>
<tr>
<td></td>
<td>Decision Tables</td>
</tr>
<tr>
<td></td>
<td>Summary</td>
</tr>
</tbody>
</table>
## Chapter 9  Dependencies

- Relations between Objects 119
- System Resource Dependencies 125
- Dependencies between Layers 129
- Dependencies across Tiers 132
- Summary 133

## Chapter 10  Data-driven and Combinatorial Testing

- Parameterized Tests 138
- Theories 139
- Generative Testing 141
- Combinatorial Testing 145
- Summary 149

## Chapter 11  Almost Unit Tests

- Examples 152
- Impact 156
- Summary 157

## Chapter 12  Test Doubles

- Stubs 159
- Fakes 162
- Mock Objects 164
- Spies 170
- Dummies 171
- Verify State or Behavior? 173
- Summary 176

## Chapter 13  Mocking Frameworks

- Constructing Test Doubles 177
- Setting Expectations 179
- Verifying Interactions 183
- Misuse, Overuse, and Other Pitfalls 185
- Summary 189

## Chapter 14  Test-driven Development—Classic Style

- Test-driving a Simple Search Engine 192
- Order of Tests 204
- Red- to Green-bar Strategies 205
<table>
<thead>
<tr>
<th>Contents</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test-driven Development</td>
<td>282</td>
</tr>
<tr>
<td>Beyond Unit Testing</td>
<td>287</td>
</tr>
<tr>
<td>Bibliography</td>
<td>289</td>
</tr>
<tr>
<td>Index</td>
<td>295</td>
</tr>
</tbody>
</table>
This page intentionally left blank
Ten years ago, I became the manager and tech lead for a small development team at a local, small start-up after spending some months developing for them. The software was an almost prototypically mired mess of convoluted logic and difficult defects. On taking the leadership role, I began to promote ideas of test-driven development (TDD) in an attempt to improve the code quality. Most of the developers were at least willing to listen, and a couple eventually embraced TDD.

One developer, however, quit two days later without saying a word to me. I was told that he said something to the effect that “I’m never going to write a test, that’s not my job as a programmer.” I was initially concerned that I’d been too eager (though I’d never insisted on anything, just attempted to educate). I no longer felt guilty after seeing the absolute nightmare that was his code, though.

Somewhat later, one of the testers complained to me about another developer—a consultant with many years of experience—who continually submitted defect-riddled code to our QA team. “It’s my job to write the code; it’s their job to find the problems with it.” No amount of discussion was going to convince this gentleman that he needed to make any effort to test his code.

Still later and on the same codebase, I ended up shipping an embarrassing defect that the testers failed to catch—despite my efforts to ensure that the units were well tested. A bit of change to some server code and an overlooked flipping of a boolean value meant that the client—a high-security chat application—no longer rang the bell on an incoming message. We didn’t have comprehensive enough end-to-end tests needed to catch the problem.

Developer tests are tools. They’re not there to make your manager happy—if that’s all they were, I, too, would find a way to skip out on creating them. Tests are tools that give you the confidence to ship, whether to an end customer or to the QA team.

Thankfully, 10 years on, most developers have learned that it’s indeed their job to test their own code. Few of you will embark on an interview where some form of developer testing isn’t discussed. Expectations are that you’re a software development professional, and part of being a professional is crafting a high-quality product. Ten years on, I’d squash any notions of hiring someone who thought they didn’t have to test their own code.

Developer testing is no longer as simple as “just do TDD,” or “write some integration tests,” however. There are many aspects of testing that a true developer must embrace in order to deliver correct, high-quality software. And while you can find a good book on TDD or a good book on combinatorial testing, *Developer Testing:...*
Building Quality into Software overviews the essentials in one place. Alexander surveys the world of testing to clarify the numerous kinds of developer tests, weighing in on the relative merits of each and providing you with indispensable tips for success.

In Developer Testing, Alexander first presents a case for the kinds of tests you need to focus on. He discusses overlooked but useful concepts such as programming by contract. He teaches what it takes to design code that can easily be tested. And he emphasizes two of my favorite goals: constructing highly readable specification-based tests that retain high documentation value, and eliminating the various flavors of duplication—one of the biggest enemies to quality systems. He wraps up the topic of unit testing with a pragmatic, balanced approach to TDD, presenting both classical and mockist TDD techniques.

But wait! There’s more: In Chapter 18, “Beyond Unit Testing,” Alexander provides as extensive a discussion as you could expect in one chapter on the murky world of developer tests that fall outside the range of unit tests. Designing these tests to be stable, useful, and sustainable is quite the challenge. Developer Testing doesn’t disappoint, again supplying abundant hard-earned wisdom on how to best tackle the topic.

I enjoyed working through Developer Testing and found that it got even better as it went along, as Alexander worked through the meaty coding parts. It’s hard to come up with good examples that keep the reader engaged and frustration free, and Alexander succeeds masterfully with his examples. I think you’ll enjoy the book too, and you’ll also thank yourself for getting a foundation of the testing skills that are critical to your continued career growth.
The subtitle says it all—“Building Quality into Software.” We’ve always known that we can’t test quality in by testing after coding is “done.” Quality has to be baked in. To do that, the entire delivery team, including developers, has to start building each feature by thinking about how to test it. In successful teams, every team member has an agile testing mind-set. They work with the delivery and customer teams to understand what the customers need to be successful. They focus on preventing, rather than finding, defects. They find the simplest solutions that provide the right value.

In my experience, even teams with experienced professional testers need developers who understand testing. They need to be able to talk with designers, product experts, testers, and other team members to learn what each feature should do. They need to design testable code. They need to know how to use tests to guide coding, from the unit level on up. They need to know how to design test code as well as—or even better than—production code, because that test code is our living documentation and our safety net. They need to know how to explore each feature they develop to learn whether it delivers the right value to customers.

I’ve encountered a lot of teams where developers are paid to write production code and pushed to meet deadlines. Their managers consider any time spent testing to be a waste. If these organizations have testers at all, they’re considered to be less valuable contributors, and the bugs they find are logged in a defect tracking system and ignored. These teams build a mass of code that nobody understands and that is difficult to change without something breaking. Over time they generally grind to a halt under the weight of their technical debt.

I’ve been fortunate over the years to work with several developers who really “get” testing. They eagerly engage in conversations with business experts, designers, testers, analysts, data specialists, and others to create a shared understanding of how each feature should behave. They’re comfortable pairing with testers and happily test their own work even before it’s delivered to a test environment. These are happy teams that deliver solid, valuable features to their customers frequently. They can change direction quickly to accommodate new business priorities.

Testing’s a vast subject, and we’re all busy, so where do you start? This book delivers key testing principles and practices to help you and your team deliver the quality your customers need, in a format that lets you pick up ideas quickly. You’ll learn the language of testing so you can collaborate effectively with testers, customers, and other delivery team members. Most importantly (at least to me), you’ll enjoy your work a lot more and be proud of the product you help to build.
This page intentionally left blank
I started writing this book four years ago with a very clear mental image of what I wanted it to be and who my readers were going to be. Four years is quite a while, and I’ve had to revise some of my ideas and assumptions, both in response to other work in the field and because of deepening understanding of the subject. The biggest thing that has happened during the course of those years is that the topic has become less controversial. Several recent books adopt a stance similar to this one, and there’s some reassuring overlap, which I interpret as being on the right track.

**Why I Wrote This Book**

I wrote this book because this was the book I should have read a decade ago! Ten years is a long time, but believe it or not, I still need this book today—although for other reasons.

Roughly 10 years ago I embarked on a journey to understand software quality. I wasn’t aware of it back then; I just knew that the code that I and my colleagues wrote was full of bugs and made us sad and the customers unhappy. I was convinced that having testers execute manual routines on our software wouldn’t significantly increase its quality—and time has proven me right! So I started reading everything I could find about software craftsmanship and testing, which led to two major observations.

First, to my surprise, these topics were often totally separated back then! Books about writing software seldom spoke of verifying it. Maybe they mentioned one or two testing techniques, but they tended to skip the theory part and the conceptual frameworks needed for understanding how to work systematically with testing in different contexts. That was my impression at least. On the other hand, books on testing often tended to take off in the direction of a testing process. Books on test-driven development focused on test-driven development. This applied to blogs and other online material too.

Second, writing testable code was harder than it initially appeared, not to mention turning old legacy monoliths into something that could be tested. To get a feeling for it, I had to dive deep into the areas of software craftsmanship, refactoring, legacy code, test-driven development, and unit testing. It took a lot of deliberate practice and study.

Based on these observations and my accumulated experience, I set some goals for a book project:
Make the foundations of software testing easily accessible to developers, so that they can make informed choices about the kind and level of verification that would be the most appropriate for code they’re about to ship. In my experience, many developers don’t read books or blogs on testing, yet they keep asking themselves: When have I tested this enough? How many tests do I need to write? What should my test verify? I wanted these to become no-brainers.

Demonstrate how a testing mind-set and the use of testing techniques can enrich the daily routines of software development and show how they can become a developer’s second nature.

Create a single, good enough body of knowledge on techniques for writing testable code. I realized that such a work would be far from comprehensible, especially if kept concise, but I wanted to create something that was complete enough to save the readers from plowing through thousands of pages of books and online material. I wanted to provide a “map of the territory,” if you will.

This is why I should have had a book written with these goals in mind a decade ago, but why today? Hasn’t the world changed? Hasn’t there been any progress in the industry? And here comes the truly interesting part: this book is just as applicable today as it would have been 10 years ago. One reason is that it’s relatively technology agnostic. Admittedly it is quite committed to object-oriented programming, although large parts hold true for procedural programming, and some contents apply to functional programming as well. Another reason is that progress in the field it covers hasn’t been as impressive as in many others. True, today, many developers have grasped the basics of testing, and few, if any, new popular frameworks and libraries are created without testability in mind. Still, I’d argue that it’s orders of magnitude easier to find a developer who’s a master in writing isomorphic JavaScript applications backed by NoSQL databases running in the cloud than to find a developer who’s really good at unit testing, refactoring, and, above all, who can remain calm when the going gets tough and keep applying developer testing practices in times of pressure from managers and stressed-out peers.

Being a consultant specializing in software development, training, and mentoring, I’ve had the privilege to work on several software development teams and to observe other teams in action. Based on these experiences, I’d say that teams and developers follow pretty much the same learning curve when it comes to quality assurance. This book is written with such a learning curve in mind, and I’ve done my best to help the reader overcome it and progress as fast as possible.
Target Audience

This is a book for developers who want to write better code and who want to avoid creating bugs. It’s about achieving quality in software by acknowledging testability as a primary quality attribute and adapting the development style thereafter. Readers of this book want to become better developers and want to understand more about software testing, but they have neither the time nor support from their peers, not to mention from their organizations.

This is not a book for beginners. It does explain many foundations and basic techniques, but it assumes that the reader knows how to work his development environment and build system and is no stranger to continuous integration and related tooling, like static analysis or code coverage tools. To get the most out of this book, the reader should have at least three years of experience creating software professionally. Such readers will find the book’s dialogues familiar and should be able to relate to the code samples, which are all based on real code, not ideal code.

I also expect the reader to work. Even though my ambition is to make lots of information readily available, I leave the knowledge integration part to the reader. This is not a cookbook.

About the Examples

This book contains a lot of source code. Still, my intention was never to write a programming book. I want this to be a book on principles and practices, and as such, it’s natural that the code examples be written in different languages. Although I’m trying to stay true to the idioms and structure used in the various languages, I also don’t want to lose the reader in fancy details specific to a single language or framework; that is, I try to keep the examples generic enough so that they can be read by anyone with a reasonable level of programming experience. At times, though, I’ve found this stance problematic. Some frameworks and languages are just better suited for certain constructs. At other times, I couldn’t decide, and I put an alternative implementation in the appendix. The source code for the examples in the book and other related code are available on the book’s companion website—http://developertesting.rocks.

How to Read This Book

This book has been written with a very specific reader in mind: the pressed-for-time developer who needs practical information about a certain topic without having to read tons of articles, blogs, or books. Therefore, the underlying idea is that each chapter should take no more than one hour to read, preferably less. Ideally, the reader should be able to finish a chapter while commuting to work. As a consequence, the
Preface

Chapters are quite independent and can be read in isolation. However, starting with the first four chapters is recommended, as they lay a common ground for the rest of the material.

Here’s a quick overview of the chapters:

- **Chapter 1: Developer Testing**—Explains that developers are engaged in a lot of testing activities and that they verify that their programs work, regardless of whether they call it testing or not. Developer testing is defined here.

- **Chapter 2: Testing Objectives, Styles, and Roles**—Describes different approaches to testing. The difference between testing to critique and testing to support is explained. The second half of the chapter is dedicated to describing traditional testing, agile testing, and different versions of behavior-driven development. Developer testing is placed on this map in the category of supporting testing that thrives in an agile context.

- **Chapter 3: The Testing Vocabulary**—This chapter can be seen as one big glossary. It explains the terms used in the testing community and presents some commonly used models like the matrix of test levels and test types and the agile testing quadrants. All terms are explained from a developer’s point of view, and ambiguities and different interpretations of some of them are acknowledged rather than resolved.

- **Chapter 4: Testability from a Developer’s Perspective**—Why should the developer care about testability? Here the case for testable software and its benefits is made. The quality attribute *testability* is broken down into observability, controllability, and smallness and explained further.

- **Chapter 5: Programming by Contract**—This chapter explains the benefits of keeping *programming by contract* in mind when developing, regardless of whether tests are being written or not. This technique formalizes responsibilities between calling code and called code, which is an important aspect of writing testable software. It also introduces the concept of assertions, which reside at the core of all testing frameworks.

- **Chapter 6: Drivers of Testability**—Some constructs in code have great impact on testability. Therefore, being able to recognize and name them is critical. This chapter explains direct and indirect input/output, state, temporal coupling, and domain-to-range ratio.

- **Chapter 7: Unit Testing**—This chapter starts by describing the fundamentals of xUnit-based testing frameworks. However, it soon moves on to more advanced topics like structuring and naming tests, proper use of assertions, constraint-based assertions, and some other technicalities of unit testing.
Chapter 8: Specification-based Testing Techniques — Here the testing domain is prevalent. Fundamental testing techniques are explained from the point of view of the developer. Knowing them is essential to being able to answer the question: “How many tests do I need to write?”

Chapter 9: Dependencies — Dependencies between classes, components, layers, or tiers all affect testability in different ways. This chapter is dedicated to explaining the different kinds and how to deal with them.

Chapter 10: Data-driven and Combinatorial Testing — This chapter explains how to handle cases where seemingly many similar-looking tests are needed. It introduces parameterized tests and theories, which both solve this problem. It also explains generative testing, which is about taking test parameterization even further. Finally, it describes techniques used by testers to deal with combinatorial explosions of test cases.

Chapter 11: Almost Unit Tests — This book relies on a definition of unit tests that disqualifies some tests that look and run almost as fast as unit tests from actually being called by that name. To emphasize the distinction, they’re called “fast medium tests”. They typically involve setting up a lightweight server of some kind, like a servlet container, mail server, or in-memory database. Such tests are described in this chapter.

Chapter 12: Test Doubles — This chapter introduces typical test doubles like stubs, mocks, fakes, and dummies, but without using any mocking frameworks. The point is to understand test doubles without having to learn yet another framework. This chapter also describes the difference between state-based and interaction-based testing.

Chapter 13: Mocking Frameworks — Here it gets very practical, as the mocking frameworks Moq, Mockito, and the test double facilities of Spock are used to create test doubles for different needs and situations — especially stubs and mocks. This chapter also includes pitfalls and antipatterns related to the use of mocking frameworks.

Chapter 14: Test-driven Development — Classic Style — Here, classic test-driven development is introduced through a longer example. The example is used to illustrate the various details of the technique, such as the order in which to write tests and strategies for making them pass.

Chapter 15: Test-driven Development — Mockist Style — There's more than one way to do test-driven development. In this chapter, an alternative way is described. It's applicable in cases where test driving the design of the system is more important than test driving the implementation of a single class or component.
- Chapter 16: Duplication—This chapter explains why code duplication is bad for testability, but sometimes a necessary evil to achieve independence and throughput. Two main categories of duplication are introduced and dissected: mechanical duplication and duplication of knowledge.

- Chapter 17: Working with Test Code—This chapter contains suggestions on what to do before resorting to comments in test code and when to delete tests.

- Chapter 18: Beyond Unit Testing—Unit testing is the foundation of developer testing, but it’s just one piece of the puzzle. Software systems of today are often complex and require testing at various levels of abstraction and granularity. This is where integration, system, and end-to-end tests come in. This chapter introduces such tests through a series of examples and discusses their characteristics.

- Chapter 19: Test Ideas and Heuristics—This final chapter, on the border of being an appendix, summarizes various test heuristics and ideas from the book.

Register your copy of Developer Testing at informit.com for convenient access to downloads, updates, and corrections as they become available. To start the registration process, go to informit.com and log in or create an account. Enter the product ISBN (9780134291062) and click Submit. Once the process is complete, you will find any available bonus content under “Registered Products.”
Writing a book is a team effort. The author is the one who writes the text and spends the most time with it, but many people make their contributions. This book is no exception. My first thanks go to Joakim Tengstrand, an expert in software development with a unique perspective on things, but above all, my friend. He’s been giving me continual and insightful feedback from very early stages of writing to the very end.

Another person who needs a special mention is Stephen Vance. He helped me by doing a very exhaustive second-pass technical review. Not only did he offer extensive and very helpful feedback, he also found many, if not all, places where I tried to make things easy for myself. In addition, he helped me broaden the book by offering alternatives and perspectives.

As a matter of fact, this entire book wouldn’t exist in its present form without Lisa Crispin’s help. She’s helped me to get it published, and she has supported me whenever I needed it throughout the entire process. I’m honored to have her write one of the forewords. Speaking of which, Jeff Langr also deserves my deepest gratitude for writing a foreword as well and for motivating me to rewrite an important section that I had been postponing forever. Mike Cohn, whom I’ve never had the pleasure of meeting, has accepted this book into his series. I can’t even express how grateful I am and what it means to me. Thanks!

While on the topic of publication, I really need to thank Chris Guzikowski at Addison-Wesley. He’s been very professional throughout the process and, above all, supportive beyond all limits. I don’t know how many e-mails I started with something akin to: “Thanks for your patience! There’s this thing I need to do before handing in the manuscript . . .” During the process of finalizing the book, I’ve had the pleasure to work with very professional and accommodating people, who really made the end of the journey interesting, challenging, and quite fun. Many thanks to Chris Zahn, Lisa McCoy, Julie Nahil, and Rachel Paul.

My reviewers, Mikael Brodd, Max Wenzin, and Mats Henricson, have done a huge job going through the text while doing the first-pass technical review.

Carlos Blé deserves special thanks for taking me through a TDD session that ended up producing a solution quite different from the one in the chapter on TDD. It sure gave me some things to think about, and it eventually led to a rewrite of the entire chapter. Ben Kelly has helped me enormously in getting the details of the testing terminology right, and he didn’t let me escape with dividing some work between developers and testers. Dan North has helped me get the details straight about BDD and ATDD. Frank Appel has helped me around the topic of unit testing and related
material. His well-grounded and thorough comments really made me stop and think at times. Many thanks. Alex Moore-Niemi has widened the book’s scope by providing a sidebar on types, a topic with which I’m only superficially familiar.

I’d also like to extend my thanks to Al Bagdonas, my first-pass proofreader and copy editor for his dedication to this project.

In addition, I’d like to thank other people who have helped me along the way or served as inspiration: Per Lundholm, Kristoffer Skjutare, Fredrik Lindgren, Yassal Sundman, Olle Hallin, Jörgen Damberg, Lasse Koskela, Bobby Singh Sanghera, Gojko Adzic, and Peter Franzen.

Last, but not least, I’m joining the scores of authors who thank their wives and families. Writing a book is an endeavor that requires a lot of passion, dedication, and above all, time away from the family. Teresia, thanks for your patience and support.
Alexander Tarlinder wrote his first computer program around the age of 10, sometime in the early nineties. It was a simple, text-based role-playing game for the Commodore 64. It had lots of GOTO statements and an abundance of duplicated code. Still, to him, this was the most fantastic piece of software ever conceived, and an entry point to his future career.

Twenty-five years later, Alexander still writes code and remains a developer at heart. Today, his professional career stretches over 15 years, a time during which he shouldered a variety of roles: developer, architect, project manager, Scrum-Master, tester, and agile coach. In all these roles, he has gravitated toward sustainable pace, craftsmanship, and attention to quality, and he eventually got test infected around 2005. In a way, this was inevitable, because many of his projects involved programming money somehow (in the banking and gaming industry), and he always felt that he could do more to ensure the quality of his code before handing it over to someone else.

Presently, Alexander seeks roles that allow him to influence the implementation process on a larger scale. He combines development projects with training and coaching, and he shares technical and nontechnical aspects of developer testing and quality assurance in conferences and local user groups meetings.
Testability means different things to different people depending on the context. From a bird’s eye view, testability is linked to our prior experience of the things we want to test and our tolerance for defects: the commercial web site that we’ve been running for the last five years will require less testing and will be easier to test than the insulin pump that we’re building for the first time. If we run a project, testability would be about obtaining the necessary information, securing resources (such as tools and environments), and having the time to perform various kinds of testing. There’s also a knowledge perspective: How well do we know the product and the technology used to build it? How good are our testing skills? What’s our testing strategy? Yet another take on testability would be developing an understanding of what to build by having reliable specifications and ensuring user involvement. It’s hard to test anything unless we know how it’s supposed to behave.¹

Before breaking down what testability means to developers, let’s look at why achieving it for software is an end in itself.

**Testable Software**

Testable software encourages the existence of tests—be they manual or automatic. The more testable the software, the greater the chance that somebody will test it, that is, verify that it behaves correctly with respect to a specification or some other expectations, or explore its behavior with some specific objective in mind. Generally, people follow the path of least resistance in their work, and if testing isn’t along that path, it’s very likely not going to be performed (Figure 4.1).

That testable software will have a greater chance of undergoing some kind of testing may sound really obvious. Equally apparent is the fact that lack of testability, often combined with time pressure, can and does result in bug-ridden and broken software.

Whereas testable software stands on one side of the scale, The Big Ball of Mud (Foote & Yoder 1999) stands on the other. This is code that makes you suspect that

---

¹. For an in-depth breakdown of testability, I recommend James Bach’s work on the subject (2015).
the people who wrote it deliberately booby-trapped it with antitestability constructs to make your life miserable. A very real consequence of working with a system that’s evolved into The Big Ball of Mud architecture is that it’ll prevent you from verifying the effects of your coding efforts. For various reasons, such as convoluted configuration, unnecessary start-up time, or just the difficulty to produce a certain state or data, you may actually have a hard time executing the code you’ve just written, not to mention being able to write any kinds of tests for it!

For example, imagine a system that requires you to log in to a user interface (UI) and then performing a series of steps that require interacting with various graphical components and then navigating through multiple views before being able to reach the functionality you’ve just changed or added and want to verify. To make things more realistic (yes, this is a real-life example), further imagine that arriving at the login screen takes four minutes because of some poor design decisions that ended up having a severe impact on start-up time. As another example, imagine a batch program that has to run for 20 minutes before a certain condition is met and a specific path through the code is taken.

Honestly, how many times will you verify, or even just run, the new code if you have to enter values into a multitude of fields in a UI and click through several screens
(to say nothing of waiting for the application to start up), or if you must take a coffee break every time you want to check if your batch program behaves correctly for that special almost-never-occurring edge case?

Testers approaching a system with The Big Ball of Mud architecture also face a daunting task. Their test cases will start with a long sequence of instructions about how to put the system in a state the test expects. This will be the script for how to fill in the values in the UI or how to set the system up for the 20-minute-long batch execution. Not only must the testers author that script and make it detailed enough, they must also follow it . . . many times, if they are unlucky. Brrr.

**Benefits of Testability**

Apart from shielding the developers and testers from immediate misery, testable software also has some other appealing qualities.

**Its Functionality Can Be Verified**

If the software is developed so that its behavior can be verified, it’s easy to confirm that it supports a certain feature, behaves correctly given a certain input, adheres to a specific contract, or fulfills some nonfunctional constraint. Resolving a bug becomes a matter of locating it, changing the code, and running some tests. The opposite of this rather mechanical and predictable procedure is playing the guessing game:

Charlie: Does business rule X apply in situation Y?
Kate: Not a clue! Wasn’t business rule X replaced by business rule Z in release 5.21 by the way?
Charlie: Dunno, but wasn’t release 5.2 scrapped altogether? I recall that it was too slow and buggy, and that we waited for 5.4 instead.
Kate: Got me there. Not a clue.

Such discussions take place if the software’s functionality isn’t verifiable and is expressed as guesses instead. Lack of testability makes confirming these guesses hard and time consuming. Therefore, there’s a strong probability that it won’t be done.

And because it won’t be done, some of the software’s features will only be found in the lore and telltales of the organization. Features may “get lost” and, even worse, features may get imagined and people will start expecting them to be there, even though they never were. All this leads to “this is not a bug, it’s a feature” type of arguments and blame games.
It Comes with Fewer Surprises

Irrespective of the methodology used to run a software project, at some point somebody will want to check on its progress. How much work has been done? How much remains? Such checks needn’t be very formal and don’t require a written report with milestones, toll gates, or Gantt charts. In agile teams, developers will be communicating their progress at least on a daily basis in standup meetings or their equivalents.

However, estimating progress for software that has no tests (because of poor testability) ranges between best guesses and wishful thinking. A developer who believes he is “95 percent finished” with a feature has virtually no way of telling what fraction of seemingly unrelated functionality he has broken along the way and how much time it’ll take to fix these regressions and the remaining “5 percent”. A suite of tests makes this situation more manageable. Again, if the feature is supposedly “95 percent finished” and all tests for the new functionality pass, as well as those that exercise the rest of the system, the estimate is much more credible. Now the uncertainty is reduced to potential surprises in the remaining work, not to random regressions that may pop up anywhere in the system. Needless to say, this assumes that the codebase is indeed covered by tests that would actually break had any regression issues taken place.2

It Can Be Changed

Software can always be changed. The trick is to do it safely and at a reasonable cost. Assuming that testable software implies tests, their presence allows making changes without having to worry that something—probably unrelated—will break as a side effect of that change.

Changing software that has no tests makes the average developer uncomfortable and afraid (and it should). Fear is easily observed in code. It manifests itself as duplication—the safe way to avoid breaking something that works. When doing code archaeology, we can sometimes find evidence of the following scenario:

At some point in time, the developer needed a certain feature. Alas, there wasn’t anything quite like it in the codebase. Instead of adapting an existing concept, by generalizing or parameterizing it, he took the safe route and created a parallel implementation, knowing that a bug in it would only affect the new functionality and leave the rest of the system unharmed.

2. A slight variation of this is nicely described in the book Pragmatic Unit Testing by Andrew Hunt and David Thomas (2003). They plot productivity versus time for software with and without tests. The productivity is lower for software supported by tests, but it’s kept constant over time. For software without tests, the initial productivity is higher, but it plummets after a while and becomes negative. Have you been there? I have.
This is but one form of duplication. In fact, the topic is intricate enough to deserve a chapter of its own.

**Why Care about Testability**

Ultimately, testable software is about money and happiness. Its stakeholders can roll out new features quickly, obtain accurate estimates from the developers, and sleep well at night, because they’re confident about the quality. As developers working with code every day, we, too, want to be able to feel productive, give good estimates, and be proud of the quality of our systems. We also want our job to feel fulfilling; we don’t want to get stuck in eternal code-fix cycles, and, above all, we don’t want our job to be repetitive and mind numbing. Unfortunately, unless our software is testable, we run that risk. Untestable software forces us to work *more* and *harder* instead of *smarter*.

---

**Tests Are Wasteful**

by Stephen Vance

This may sound heretical in a book on developer testing and from the author of another book on code-level testing, but bear with me. Agile methods attempt to improve the software we write, or more generally, the results of our knowledge work. I’m very careful to phrase this in a way that highlights that the results are more important than the methods. If some magical Intention Machine produced the software we want without programming, this entire book would be academic. If we could achieve the results without software altogether at the same levels of speed and convenience, our entire discipline would be irrelevant. In some sense, as advanced as we are compared to the course of human history, the labor-intensive-approach trade we ply is quite primitive. Before we wither at the futility of it all, we realize we can only achieve this magical future through improvement.

Most Agile methods have some basis in the thinking that revolutionized manufacturing at the end of the twentieth century. Lean, Total Quality Management, Just-in-time, Theory of Constraints, and the Toyota Production System from the likes of Juran, Deming, Ohno, and Goldratt completely changed the state of manufacturing. Agile methods take those insights and apply them to a domain of inherent invention and variability. Although the principles must be significantly adapted, most of them still apply.

A key principle is the elimination of waste. The Toyota Production System even has three words for waste, *muda*, *mura*, and *muri*, and *mura* has at
least seven subcategories captured in the acronym TIMWOOD. Much of our testing focuses on the waste of defects, but does so by incurring inventory and overprocessing.

We incur inventory waste when we invest capital (i.e., coding time) in product that has not yet derived value. Since tests are never delivered, they are eternal inventory. They are an investment with no direct return, only indirect through the reduction and possible prevention of defects.

We incur overprocessing waste by spending the extra attention required to write the tests as compared to the raw production code. The extra attention may pay off compared to the debugging time to get it right at first, the rework for the defects we don't catch, and the refamiliarization on each maintenance encounter. It is clearly additional to getting the code right naturally from the start.

The previous alternatives clearly show that tests are better than the problems they address. That just means they’re the best thing we have, not the best we can do. Ultimately, we care about correctness, not tests. We need to keep looking for better ways to ensure the correctness of our software.

I haven’t found the answer yet, but there are some interesting candidates.

**Domain-Specific Languages**

Domain-specific languages (DSLs) have promise. They simplify the work for their users and avoid the repetitive creation of similar code. They bring us closer to being able to say exactly what we mean in the language of the problem we are solving by encapsulating potentially complex logic in a higher-order vocabulary. If the author guarantees the correctness of the elements of the DSL, whole layers of code are correct before we try to use them.

However, good DSLs are notoriously hard to write. Arguably, almost every API we use should be a good DSL, but how many are? Creating a good DSL requires not only taking the time to understand the domain, but also playing with different models of the domain and its interactions to optimize its usability and utility. Additionally, there may be multiple characteristic usage patterns, differing levels of relevant abstractions, varying levels of user expertise, and impactful technological changes over time.

Take, for example, the Capybara acceptance test framework for Ruby, often cited as an example of a well-crafted DSL in the context of its host language. With a set of actions like `visit`, `fill_in`, `click_button` and matchers like `have_content`, it is well suited to static web pages. Under the covers, it has adapted to the rapid evolution of underlying tools like Selenium, but not without challenges at times. However, it still has difficulty dealing with the dynamic, time-dependent behaviors of single-page applications.

**Formal Methods**

Formal methods sound good. They provide formal proof of the correctness of the code. Unfortunately, we have had a hard time adapting them to larger
problems, they are very labor intensive, and most programmers I’ve met prefer not to deal in that level of mathematical rigor. The research continues, but we’re not there yet.

Types
Types bridge the gap between mainstream languages and formal methods in my opinion. By using a subset of formal specification, they help you ensure correctness by cleanly and compactly expressing your illegal “corner cases” in the context they can be most readily applied.

Others
Other approaches provide partial, complex, or laborious solutions. If you’re so inclined, maybe you can find that great breakthrough. Until then, keep testing.

Testability Defined
Testability is a quality attribute among other “ilities” like reliability, maintainability, and usability. Just like the other quality attributes, it can be broken down into more fine-grained components (Figure 4.2). Observability and controllability are the two cornerstones of testability. Without them, it’s hard to say anything about correctness. The remaining components described next made it to the model based on my practical experience, although I hope that their presence isn’t surprising or controversial.

When a program element (see “Program Elements”) is testable, it means that it can be put in a known state, acted on, and then observed. Further, it means that this can be done without affecting any other program elements and without them interfering. In other words, it’s about making the black box of testing somewhat transparent and adding some control levers to it.

Program Elements
From time to time I’ll be using the term program element. The meaning of the term depends on the context. Sometimes it’s a function, sometimes a method, sometimes a class, sometimes a module, sometimes a component, or sometimes all of these things. I use the generic term to avoid clumsy sentences.

Using a catch-all term also solves the problem of emphasizing the difference between programming paradigms. Although the book favors object-oriented code, many techniques apply to procedural and functional constructs too. So instead of writing “class” and “method” everywhere, I can use “program element” and refer to “function” or “module” as well, like a C file with a bunch of related functions.
Observability

In order to verify that whatever action our tested program element has been subjected to has had an impact, we need to be able to observe it. The best test in the world isn’t worth anything unless its effects can be seen. Software can be observed using a variety of methods. One way of classifying them is in order of increasing intrusiveness.

The obvious, but seldom sufficient, method of observation is to examine whatever output the tested program element produces. Sometimes that output is a sequence of characters, sometimes a window full of widgets, sometimes a web page, and sometimes a rising or falling signal on the pin of a chip.

Then there’s output that isn’t always meant for the end users. Logging statements, temporary files, lock files, and diagnostics information are all output. Such output is mostly meant for operations and other more “technical” stakeholders. Together with the user output, it provides a source of information for nonintrusive testing.

To increase observability beyond the application’s obvious and less obvious output, we have to be willing to make some intrusions and modify it accordingly. Both testers and developers benefit from strategically placed observation points and various types of hooks/seams for attaching probes, changing implementations, or just peeking at the internal state of the application. Such modifications are sometimes frowned upon, as they result in injection of code with the sole purpose of increasing observability. At the last level, there’s a kind of observability that’s achievable only by
developers. It’s the ability to step through running code using a debugger. This certainly provides maximum observability at the cost of total intrusion. I don’t consider this activity testing, but rather writing code. And you certainly don’t want debugging to be your only means of verifying that your code works.

Too many observation points and working too far from production code may result in the appearance of Heisenbugs—bugs that tend to disappear when one tries to find and study them. This happens because the inspection process changes something in the program’s execution. Excessive logging may, for example, hide a race condition because of the time it takes to construct and output the information to be logged.

Logging, by the way, is a double-edged sword. Although it’s certainly the easiest way to increase observability, it may also destroy readability. After all, who hasn’t seen methods like this:

```java
void performRemoteReboot(String message) {
    if (log.isDebugEnabled()) {
        log.debug("In performRemoteReboot:" + message);
    }
    log.debug("Creating telnet client");
    TelnetClient client = new TelnetClient("192.168.1.34");
    log.debug("Logging in");
    client.login("rebooter", "secret42");
    log.debug("Rebooting");
    client.send("/sbin/shutdown -r now " + message + " ");
    client.close();
    log.debug("done");
}
```

As developers, we need to take observability into account early. We need to think about what kind of additional output we and our testers may want and where to add more observation points.

Observability and information hiding are often at odds with each other. Many languages, most notably the object-oriented ones, have mechanisms that enable them to limit the visibility of code and data to separate the interface (function) from the implementation. In formal terms, this means that any proofs of correctness must rely solely on public properties and not on “secret” ones (Meyer 1997). On top of that, the general opinion among developers seems to be that the kind of testing that they do should be performed at the level of public interfaces. The argument is sound: if tests get coupled to internal representations and operations, they get brittle and become obsolete or won’t even compile with the slightest refactoring. They no longer serve as the safety net needed to make refactoring a safe operation.
Although all of this is true, the root cause of the problem isn’t really information hiding or encapsulation, but poor design and implementation, which, in turn, forces us to ask the question of the decade: Should I test private methods?\(^3\)

Old systems were seldom designed with testability in mind, which means that their program elements often have multiple areas of responsibility, operate at different levels of abstraction at the same time, and exhibit high coupling and low cohesion. Because of the mess under the hood, testing specific functionality in such systems through whatever public interfaces they have (or even finding such interfaces) is a laborious and slow process. Tests, especially unit tests, become very complex because they need to set up entire “ecosystems” of seemingly unrelated dependencies to get something deep in the dragon’s lair working.

In such cases we have two options. Option one is to open up the encapsulation by relaxing restrictions on accessibility to increase both observability and controllability. In Java, changing methods from private to package scoped makes them accessible to (test) code in the same package. In C++, there’s the infamous friend keyword, which can be used to achieve roughly a similar result, and C# has its InternalsVisibleTo attribute.

The other option is to consider the fact that testing at a level where we need to worry about the observability of deeply buried monolithic spaghetti isn’t the course of action that gives the best bang for the buck at the given moment. Higher-level tests, like system tests or integration tests, may be a better bet for old low-quality code that doesn’t change that much (Vance 2013).

With well-designed new code, observability and information hiding shouldn’t be an issue. If the code is designed with testability in mind from the start and each program element has a single area of responsibility, then it follows that all interesting abstractions and their functionality will be primary concepts in the code. In object-oriented languages this corresponds to public classes with well-defined functionality (in procedural languages, to modules or the like). Many such abstractions may be too specialized to be useful outside the system, but in context they’re most meaningful and eligible for detailed developer testing. The tale in the sidebar contains some examples of this.

**Testing Encapsulated Code**

Don’t put yourself in the position where testing encapsulated code becomes an issue. If you’re already there and can’t escape in the foreseeable future, test it!

---

3. Or functions, or modules, or any program element, the accessibility to which is restricted by the programming language to support encapsulation.
The Tale of the Math Package

Let’s assume that we’re setting out to build a math package with a user interface. Users will enter different expressions or equations somehow, and the software will compute the result or perform a mathematical operation like differentiation or integration.

If built iteratively in increments, possibly in a test-driven manner, the entire application may initially start in a single class or module, which will do everything: accept input, parse it, evaluate it, and eventually output the results. Such a program can easily be tested via its public interface, which would be somewhere around accepting unparsed input and returning the results of the computation. Maybe like so:

```java
DisplayableResult evaluate(String userInput)
```

However, as the code grows, new program elements will be introduced behind this public interface. First a parser may appear, then something that evaluates the parsed input, then a bunch of specialized math functions, and finally a module that presents the output somehow—either graphically or using some clever notation. As all these building blocks come into existence, testing them through only the first public entry point becomes ceremonious, because they’re standalone abstractions with well-defined behavior. Consequently, all of them operate on their own data types and domains, which have their own boundary values and equivalence partitions (see Chapter 8, “Specification-based Testing Techniques”) and their own kind of error and exception handling. Ergo, they need their share of tests. Such tests will be much simpler than the ones starting at the boundary of the public interface, because they’ll hit the targeted functionality using its own domains and abstractions. Thus, a parsing module will be tested using strings as input and verified against some tree-like structure that represents the expression, whereas an evaluation module may be tested using this tree-like representation and returning something similar. If the underlying math library contains a tailor-made implementation of prime number factorization, that, too, will need specific testing.

If built with some degree of upfront design (be it detailed or rough), that design will reveal some interesting actors, like the parser or the evaluation engine, and their interfaces from the start. At this stage it will be apparent that these actors need to work together correctly, but also exhibit individual correctness. Enter tests of nonpublic behavior . . .
Controllability

Controllability is the ability to put something in a specific state and is of paramount importance to any kind of testing because it leads to reproducibility. As developers, we like to deal with determinism. We like things to happen the same way every time, or at least in a way that we understand. When we get a bug report, we want to be able to reproduce the bug so that we may understand under what conditions it occurs. Given that understanding, we can fix it. The ability to reproduce a given condition in a system, component, or class depends on the ability to isolate it and manipulate its internal state.

Dealing with state is complex enough to mandate a section of its own. For now, we can safely assume that too much state turns reproducibility, and hence controllability, into a real pain. But what is state? In this context, state simply refers to whatever data we need to provide in order to set the system up for testing. In practice, state isn’t only about data. To get a system into a certain state, we usually have to set up some data and execute some of the system’s functions, which in turn will act on the data and lead to the desired state.

Different test types require different amounts of state. A unit test for a class that takes a string as a parameter in its constructor and prints it on the screen when a certain method is called has little state. On the other hand, if we need to set up thousands of fake transactions in a database to test aggregation of cumulative discounts, then that would qualify as a great deal of state.

Deployability

Before the advent of DevOps, deployability seldom made it to the top five quality attributes to consider when implementing a system. Think about the time you were in a large corporation that deployed its huge monolith to a commercial application server. Was the process easy? Deployability is a measure of the amount of work needed to deploy the system, most notably, into production. To get a rough feeling for it, ask:
“How long does it take to get a change that affects one line of code into production?” (Poppendieck & Poppendieck 2006).

Deployability affects the developers’ ability to run their code in a production-like environment. Let’s say that a chunk of code passes its unit tests and all other tests on the developer’s machine. Now it's time to see if the code actually works as expected in an environment that has more data, more integrations, and more complexity (like a good production-like test environment should have). This is a critical point. If deploying a new version of the system is complicated and prone to error or takes too much time, it won’t be done. A typical process that illustrates this problem is manual deployment based on a list of instructions. Common traits of deployment instructions are that they’re old, they contain some nonobvious steps that may not be relevant at all, and despite their apparent level of detail, they still require a large amount of tacit knowledge. Furthermore, they describe a process that’s complex enough to be quite error prone.

### Manual Deployment Instructions

A list of instructions for manual deployment is a scary relic from the past, and it can break even the toughest of us. It’s a sequence of steps written probably five or more years ago, detailing the procedure to manually deploy a system. It may look something like this:

1. Log in to prod.mycompany.com using ssh with user root, password secret123.
2. Navigate to the application server directory:
   
   ```bash
cd /data/opt/extras/appserver/jboss
   ```
3. Stop the server by running the following:
   
   ```bash
./stop_server_v1_7.sh
   ```
4. On your local machine, run the build script:
   
   ```bash
cd c:\projects\killerapp, ant package
   ```
5. Use WinSCP version 1.32 to copy killerapp.ear to the deployment directory.
6. Remove the temporary files in /tmp/killerapp.
7. Clear the application cache:
   
   ```bash
rm -rf server/killerapp/cache*)
   ```
8. More steps ...
Being unable to deploy painlessly often punishes the developers in the end. If deployment is too complicated and too time consuming, or perceived as such, they may stop verifying that their code runs in environments that are different from their development machines. If this starts happening, they end up in the good-old “it works on my machine” argument, and it never makes them look good, like in this argument between Tracy the Tester and David the Developer:

Tracy: I tried to run the routine for verifying postal codes in Norway. When I entered an invalid code, nothing happened.
David: All my unit tests are green and I even ran the integration tests!
Tracy: Great! But I expected an error message from the system, or at least some kind of reaction.
David: But really, look at my screen! I get an error message when entering an invalid postal code. I have a Norwegian postal code in my database.
Tracy: I notice that you’re running build 273 while the test environment runs 269. What happened?
David: Well . . . I didn’t deploy! It would take me half a day to do it! I’d have to add a column to the database and then manually dump the data for Norway. Then I’d have to copy the six artifacts that make up the system to the application server, but before doing that I’d have to rebuild three of them . . . I forgot to run the thing because I wanted to finish it!

The bottom line is that developers are not to consider themselves finished with their code until they’ve executed it in an environment that resembles the actual production environment.

Poor deployability has other adverse effects as well. For example, when preparing a demo at the end of an iteration, a team can get totally stressed out if getting the last-minute fixes to the demo environment is a lengthy process because of a manual procedure.

Last, but not least, struggling with unpredictable deployment also makes critical bug fixes difficult. I don’t encourage making quick changes that have to be made in a very short time frame, but sometimes you encounter critical bugs in production and they have to be fixed immediately. In such situations, you don’t want to think about how hard it’s going to get the fix out—you just want to squash the bug.

What about Automated Deployment?

One way to ensure good deployability is to commit to continuous integration and then adapt the techniques described in the book Continuous Delivery. Its authors often repeat: “If it’s painful, do it more often” (Humble & Farley 2010), and this certainly refers to the deployment process, which should be automated.
Isolability

Isolability, modularity, low coupling—in this context, they’re all different sides of the same coin. There are many names for this property, but regardless of the name, it’s about being able to isolate the program element under test—be it a function, class, web service, or an entire system.

Isolability is a desirable property from both a developer’s and a tester’s point of view. In modular systems, related concepts are grouped together, and changes don’t ripple across the entire system. On the other hand, components with lots of dependencies are not only difficult to modify, but also difficult to test. Their tests will require much setup, often of seemingly unrelated dependencies, and their interactions with the outside world will be artificial and hard to make sense of.

Isolability applies at all levels of a system. On the class level, isolability can be described in terms of fan-out, that is, the number of outgoing dependencies on other classes. A useful design rule of thumb is trying to achieve a low fan-out. In fact, high fan-out is often considered bad design (Borysowich 2007). Unit testing classes with high fan-out is cumbersome because of the number of test doubles needed to isolate the class from all collaborators.

Poor isolability at the component level may manifest itself as difficulty setting up its surrounding environment. The component may be coupled to other components by various communication protocols such as SOAP or connected in more indirect ways such as queues or message buses. Putting such a component under test may require that parts of it be reimplemented to make the integration points interchangeable for stubs. In some unfortunate cases, this cannot be done, and testing such a component may require that an entire middleware package be set up just to make it testable.

Systems with poor isolability suffer from the sum of poorness of their individual components. So if a system is composed of one component that makes use of an enterprise-wide message bus, another component that requires a very specific directory layout on the production server (because it won’t even run anywhere else), and a third that requires some web services at specific locations, you’re in for a treat.

Smallness

The smaller the software, the better the testability, because there’s less to test. Simply put, there are fewer moving parts that need to be controlled and observed, to stay consistent with this chapter’s terminology. Smallness primarily translates into the quantity of tests needed to cover the software to achieve a sufficient degree of confidence. But what exactly about the software should be “small”? From a testability perspective, two properties matter the most: the number of features and the size of the codebase. They both drive different aspects of testing.
Feature-richness drives testing from both a black box and a white box perspective. Each feature somehow needs to be tested and verified from the perspective of the user. This typically requires a mix of manual testing and automated high-level tests like end-to-end tests or system tests. In addition, low-level tests are required to secure the building blocks that comprise all the features. Each new feature brings additional complexity to the table and increases the potential for unfortunate and unforeseen interactions with existing features. This implies that there are clear incentives to keep down the number of features in software, which includes removing unused ones.

A codebase’s smallness is a bit trickier, because it depends on a number of factors. These factors aren’t related to the number of features, which means that they’re seldom observable from a black box perspective, but they may place a lot of burden on the shoulders of the developer. In short, white box testing is driven by the size of the codebase. The following sections describe properties that can make developer testing cumbersome without rewarding the effort from the feature point of view.

Singularity

If something is singular, there’s only one instance of it. In systems with high singularity, every behavior and piece of data have a single source of truth. Whenever we want to make a change, we make it in one place. In the book *The Pragmatic Programmer*, this has been formulated as the DRY principle: Don’t Repeat Yourself (Hunt & Thomas 1999).

Testing a system where singularity has been neglected is quite hard, especially from a black box perspective. Suppose, for example, that you were to test the copy/paste functionality of an editor. Such functionality is normally accessible in three ways: from a menu, by right-clicking, and by using a keyboard shortcut. If you approached this as a black box test while having a limited time constraint, you might have been satisfied with testing only one of these three ways. You’d assume that the others would work by analogy. Unfortunately, if this particular functionality had been implemented by two different developers on two different occasions, then you wouldn’t be able to assume that both are working properly.

<table>
<thead>
<tr>
<th>The tester sees . . .</th>
<th>The developer implemented . . .</th>
</tr>
</thead>
<tbody>
<tr>
<td>![Copy]</td>
<td>EditorUtil.copy</td>
</tr>
<tr>
<td></td>
<td>currentEditorPanel.performCopy</td>
</tr>
<tr>
<td>A third version?</td>
<td></td>
</tr>
</tbody>
</table>

This example is a bit simplistic, but this scenario is very common in systems that have been developed by different generations of developers (which is true of pretty much every system that’s been in use for a while). Systems with poor singularity
appear confusing and frustrating to their users, who report a bug and expect it to be fixed. However, when they perform an action similar to the one that triggered the bug by using a different command or accessing it from another part of the system, the problem is back! From their perspective, the system should behave consistently, and explaining why the bug has been fixed in two out of three places inspires confidence in neither the system nor the developers’ ability.

To a developer, nonsingularity—duplication—presents itself as the activity of implementing or changing the same data or behavior multiple times to achieve a single result. With that comes maintaining multiple instances of test code and making sure that all contracts and behavior are consistent.

Level of Abstraction

The level of abstraction is determined by the choice of programming language and frameworks. If they do the majority of the heavy lifting, the code can get both smaller and simpler. At the extremes lie the alternatives of implementing a modern application in assembly language or a high-level language, possibly backed by a few frameworks. But there’s no need to go to the extremes to find examples. Replacing thread primitives with thread libraries, making use of proper abstractions in object-oriented languages (rather than strings, integers, or lists), and working with web frameworks instead of implementing Front Controllers[^4] and parsing URLs by hand are all examples of raising the level of abstraction. For certain types of problems and constructs, employing functional or logic programming greatly raises the level of abstraction, while reducing the size of the codebase.

The choice of the programming language has a huge impact on the level of abstraction and plays a crucial role already at the level of toy programs (and scales accordingly as the complexity of the program increases). Here’s a trivial program that adds its two command-line arguments together. Whereas the C version needs to worry about string-to-integer conversion and integer overflow . . .

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
    int augend = atoi(argv[1]);
    int addend = atoi(argv[2]);

    // Let's hope that we don't overflow...
    printf("*drum roll* ... %d", augend + addend);
}
```

[^4]: https://en.wikipedia.org/wiki/Front_Controller_pattern
... its Ruby counterpart will work just fine for large numbers while being a little more tolerant with the input as well.

```ruby
puts "*drum roll* ... #{ARGV[0].to_i + ARGV[1].to_i}"
```

From a developer testing point of view, the former program would most likely give rise to more tests, because they’d need to take overflow into account. Generally, as the level of abstraction is raised, fewer tests that cover fundamental building blocks, or the “plumbing,” are needed, because such things are handled by the language or framework. The user won’t see the difference, but the developer who writes the tests will.

**Efficiency**

In this context, efficiency equals the ability to express intent in the programming language in an idiomatic way and making use of that language’s functionality to keep the code expressive and concise. It’s also about applying design patterns and best practices. Sometimes we see signs of struggle in codebases being left by developers who have fought valorously reinventing functionality already provided by the language or its libraries. You know inefficient code when you see it, right after which you delete 20 lines of it and replace them with a one-liner, which turns out idiomatic and simple.

Inefficient implementations increase the size of the codebase without providing any value. They require their tests, especially unit tests, because such tests need to cover many fundamental cases. Such cases wouldn’t need testing if they were handled by functionality in the programming language or its core libraries.

**Reuse**

Reuse is a close cousin of efficiency. Here, it refers to making use of third-party components to avoid reinventing the wheel. A codebase that contains in-house implementations of a distributed cache or a framework for managing configuration data in text files with periodic reloading\(^5\) will obviously be larger than one that uses tested and working third-party implementations.

This kind of reuse reduces the need for developer tests, because the functionality isn’t owned by them and doesn’t need to be tested. Their job is to make sure that it’s plugged in correctly, and although this, too, requires tests, they will be fewer in number.

---

5. Now this is a highly personal experience, but pretty much all legacy systems that I’ve seen have contained home-grown caches and configuration frameworks.
Mind Maintainability!
All of the aforementioned properties may be abused in a way that mostly hurts maintainability. Singularity may be taken to the extreme and create too tightly coupled systems. Too high a level of abstraction may turn into some kind of “meta programming.” Efficiency may turn into unmotivated compactness, which hurts readability. Finally, reuse may result in pet languages and frameworks being brought in, only to lead to fragmentation.

A Reminder about Testability
Have you ever worked on a project where you didn’t know what to implement until the very last moment? Where there were no requirements or where iteration planning meetings failed to result in a shared understanding about what to implement in the upcoming two or three weeks? Where the end users weren’t available?

Or maybe you weren’t able to use the development environment you needed and had to make do with inferior options. Alternatively, there was this licensed tool that would have saved the day had but somebody paid for it.

Or try this: the requirements and end users were there and so was the tooling, but nobody on the team knew how to do cross-device mobile testing.

After having dissected the kind of testability the developer is exposed to the most, I’m just reminding that there are other facets of testability that we mustn’t lose sight of.

Summary
If the software is designed with testability in mind, it will more than likely be tested. When software is testable, we can verify its functionality, measure progress while developing it, and change it safely. In the end, the result is fast and reliable delivery.

Testability can be broken down into the following components:

- **Observability**—Observe the tested program element in order to verify that it actually passes the test.
- **Controllability**—Set the tested program element in a state expected by the test.
- **Smallness**—The smaller the system or program element—with respect to the number of features and the size of the codebase—the less to test.
INDEX

A
Abstraction, level of
  high-level considerations for testing, 273
  programming language/frameworks
    impacting, 53–54
Acceptance test-driven development (ATDD), 15–17
Acceptance tests
  end-to-end, as double-loop TDD, 221–222
  functional testing via, 27
  overview of, 26
  of services/components, 248–249
Accessor, in state verification, 173–174
Act, in Triple A test structure, 88–89
Actions
  in decision tables, 115
  in state transition model, 113–114
Activities, developer testing, 2–5
ACTS (Advanced Combinatorial Testing System) tool, pairwise testing, 149
Age checks, data types and testability, 72–76
Agile testing
  BDD, ATDD, and specification by example, 15–17
  summary, 19
  understanding, 13–15
Agile Testing Quadrants, 32–33
Algorithmic errors, in behavior testing, 175–176
Almost unit tests
  examples, 152
  impact of, 156–157
  overview of, 151–152
  summary, 157
  test-specific mail servers, 153–154
  using in-memory databases, 152–153
  using lightweight containers, 154–155
  of web services, 155–156
APIs (application programming interfaces)
  in components, 24
  deciding on developer testing strategy, 268
  discovering for simple search engine, 193–194
  domain-specific languages for, 42
  error/exception handling for public, 63
  testing web services, 155–156
  in tests using in-memory databases, 152
  using/testing vendor payment gateways, 250–251
Archetype, considerations for testing, 273
Argument matchers, stubs in mocking framework, 181–182
Arguments
  contracts, 61
  stubs in mocking framework, 181–182
Arrange-Act-Assert, Triple A test structure, 88–89
Assert, in Triple A test structure, 88–89
assertEquals method
  as assertion method, 89, 106
  data-driven and combinatorial testing, 136–137, 280
  generative testing, 143
  implementing mockist style TDD, 213–214
  mock objects, 164, 167–168
  spies, 171
  working with test code, 238
Assertions
  assumptions vs., 141
  constraints and matchers, 94–99
  contract verification, 62–63
  of equality, 93–94
  exceptions to one per test, 90–92
  fluent, 96–97
Assertions (continued)
- methods, 89–90
- one per test, 90
- overview of, 89
- removing need for comments, 238–239
- specialized, 94–96
- in state-based tests, 173–174
- test-driving search engine, 196–197
- verbosity of, 92–93
- verifying in more complex tests, 266

AssertThat method
- data-driven and combinatorial testing, 280–281
- defined, 91
- fluent assertions, 97
- mock objects, 167–168
- specialized assertions, 96
- spies, 171
- tests enclosed in transactions, 246

Assumptions, theory testing, 140–141

Asynchronicity, UI tests failing, 252

ATDD (Acceptance test-driven development), 15–17

Attacks, CIA security triad for resilience to, 29

Audit, security testing as, 28

Authentication, in-memory database, 152–153

Author bias, critique-based testing and, 11

Automation
- acceptance test, 17
- agile testing, 14
- of checks, 9–10
- deployment, 50
- providing infrastructure for, 5
- smoke test, 34
- as support testing, 11
- unit test, 82

Availability
- of CI servers, 157
- in CIA security triad model, 29
- enforcing contracts, 62
- micro-services across tiers for, 132

B

Behavior
- benefits of testable software, 39
- in characterization testing, 34–35
- defining component, 27
- mock objects testing. See Mock objects
- naming unit test for expected, 87
- unit tests specifying tested code, 81
- verification, 174–176

Behavior-driven development (BDD)
- frameworks
  - double-loop TDD as, 222
  - matchers, 103–105
  - more fluent syntax of, 104
  - naming tests, 103
  - overview of, 102
  - test structure, 102–103
  - testing style, 15–17
  - unit testing in some languages with, 103–106

The Big Ball of Mud, testable software vs., 37–39

Black box testing
- implementing system tests, 26
- integration test vs., 25
- overview of, 22–23
- when singularity has been neglected, 52–53

Block copy and paste, 229–230

Blocks, Spock framework, 90

Blueprint, construction phase in traditional testing, 12

Boundary value testing
- defined, 116
- edge cases/gotchas for some data types, 111–113
- specification-based technique, 110

Broken window syndrome, in duplication, 225, 233

Brown-field business applications, testing, 258

Buffer overflow
- developer understanding of, 5
- from lenient/missing parameter checking, 61
- strings and, 111
Bugs/defects
- copy and paste introducing, 228–232
- double-mode faults, 147–148
- duplication introducing, 225–226
- fixed by developers, 3–4
- fixed in agile testing, 14–15
- in language of testing, 22
- leading to software failures, 22
- regression testing and, 30–31
- single-mode faults, 146–147

Builders
- controlling dependency between collaborators, 123–125
- removing need for comments, 239–240
- tests invoking systems via, 256–257
- for tests that are not unit tests, 265–266

Business rules
- for data types and testability, 72–76
- decision tables showing gaps/inconsistencies in, 115–116
- verifying indirect output with mock objects, 167–169
- why duplication introduces bugs, 226

C
- Canonical test method, 159–161, 228–229
- Capybara acceptance test framework, Ruby, 42
- Case insensitivity, creating search engine, 202
- Challenges, of test-driven development, 206–209, 211

Change
- benefits of testable software, 40
- making people responsible for code, 81
- oververifying in mocking frameworks, 186
- unit tests enabling, 80

Characteristics of tests that are not unit tests
- complexity, 258–259
- environmental dependence, 261–262
- error localization, 260–261
- overview of, 257–258
- performance, 261
- stability, 259–260
- target audience, 262–263

- testing brown-field business applications, 258
- Characterization testing
  - of legacy code, 3–4
  - overview of, 34–35
  - state verification with, 147
- Checking
  - benefits of testable software, 40
  - developer testing vs., 9–10
  - CI. See Continuous integration (CI)
  - CIA security triad, 29
- Class invariants, contracts, 59–60
- Classes
  - avoid mocking concrete, 187–188
  - duplicating similar functionality in different, 233–234
  - introducing test-driven development into legacy code, 206
  - mocking value-holding, 188
  - removing need for comments by splitting test, 240–241
  - removing need for comments using factory, 239–240
- Classic style TDD. See Test-driven development (TDD) - classic style
- Classification of tests
  - almost unit tests as unclassified tests, 151
  - overview of, 23
  - test levels, 23–26
  - test types. See Test types
- The Clean Coder (Martin), 206
- Cleanup methods (teardown), 84
- Clients
  - contract building blocks and, 59–60
  - implementing contracts, 60–62
  - overview of, 57–58
- Clock, dependencies of system, 127–128
- Code Contracts, 63–64
- Collaboration
  - absence of, 15
  - agile testing, 13–15
  - dependencies between objects, 119–125, 133
- Collaborator isolation, 24
- Collaborators
  - creating stubs in unit tests, 160
Collaborators (continued)
  defined, 119
  fakes replacing, 162–164
  implementing mockist style TDD, 215–216
  objects replacing. See Test doubles
  passing around, 121
  verifying indirect input transformations, 169–170
Collections
  as edge case worth checking, 112–113
  low-level test considerations, 275
Combinatorial testing
  beyond double-mode faults, 149
  overview of, 145–146
  single-mode faults, 146–147
  summary, 149
Command/Query Separation principle, 93
Command shell, tests invoking systems via, 255–257
Commenting test code
  adjusting test name vs., 237
  deleting tests that are commented out, 242
  overview of, 237
  splitting up test classes vs., 240–241
  using asserts with messages vs., 238–239
  using factories or builders vs., 239–240
  using variables/constant to clarify test vs., 238
Compile, deleting tests that do not, 242
Complexity, tests that are not unit tests, 258–259
Components
  defining behavior of, 27
  elusive definition of, 24–25
  implementing mockist style TDD, 213–214
  introducing TDD into legacy code, 206
  poor isolability of, 51
  system test of, 26
  tests exercising, 248–249, 271
Concrete classes, avoid mocking, 187–188
Condition alternatives, decision tables, 115
Conditions, decision tables, 115
Confidence of team, and testing, 11
Confidentiality, CIA security triad model, 29
Confusion, running almost unit tests with unit tests, 156
Consistency
  of pure functions, 69
  of unit tests, 82
Constants, removing need for comments with, 238
Constraints
  assertion, 94–99
  enforcing contracts with, 57–60, 62–65
  parameterized tests, 139
  search engine, 192
  verifying interactions in mocking framework, 183
Construction phase, in traditional testing, 11–13
Constructor
  copy and paste, 230–231
  creating stubs in unit tests, 160
  passing in collaborators, 121
  unit testing frameworks, 85
Containers
  almost unit tests using lightweight, 154–155
  new school approach to embedded, 155
  testing brown-field business applications, 258
Context method, BDD-style test framework, 102–103
Context, naming standard and, 87
Continuous Delivery—Reliable Software Delivery through Build, Test, and Deployment Automation (Humble & Farley), 24
Continuous integration (CI)
  developers implementing, 4–5
  running almost unit tests with unit tests and, 155–156
  running unit tests in environment of, 82
  TDD exposing deficiencies in, 209
  in traditional testing, 12–13
Contracts, Programming by Contract
Controllability
  defined, 55
  deployability and, 48–50
increasing through encapsulation in old systems, 46
isolability and, 51
overview of, 48
test first or test last and, 209–210
as testability quality attribute, 48–51
UI tests failing, 252
Convergence, as traditional testing risk, 12
Copy and paste programming
of blocks of code, 229–230
breeding duplication, 225–226
of constructors, 230–231
example of, 228–229
generally the wrong thing to do, 227
as mechanical duplication, 228
messing up metrics, 226
method duplication, 231–232
when to use, 227
Coupling
behavior tests introducing, 175–176
black box reducing, 22
singularity introducing, 227
temporal, 71–72
test independence introducing, 263–264
during verification of mock object, 186
CPU performance, 261
Critique
Agile Testing Quadrants for product, 32–33
testing to, 10–11
Cross-checks, high-level tests, 274
Cross–time zone tests, 112
CUnit unit testing framework, 83
Custom constraints, assertions, 94–99
Customer
in BDD, ATDD, and specification by example, 15–17
registration in mockist style TDD, 214–219
use of term in this book, 17
using ubiquitous language of, 15–17
D
Data
CIA security triad model for, 29
dividing into equivalence partitions, 107–110
UI test failure to control, 252–253
Data-driven and combinatorial testing
beyond double-mode faults/all pairs, 149
and combinatorial testing, 145–149
generative testing, 141–144
high-level considerations on format, 273
overview of, 135–137
parameterized tests, 138–139
source code, 279–282
summary, 149
theories, 139–141
Data helpers, complex tests, 265–266
Data points, in theory tests, 140–142
Data types
edge cases/gotchas for some, 111–113
and testability, 72–76
Databases
almost unit tests using in-memory, 152–153
as piles of state, 70
testing brown-field business applications, 258
tests enclosed in transactions, 247
Date pickers, choosing, 112
Dates
boundary values for, 112
low-level test considerations, 276
Daylight saving time (DST), 112
Debuggers, 44–45
Decision tables, 115–117
Decoupling layers, 131
Defects. See Bugs/defects
Degenerate case, order of tests in TDD, 85
Deleting tests, 241–243
Dependencies
across tiers, 132
between collaborating objects, 119–125
isolability as fan-out of, 51
between layers, 129–132
overview of, 119
summary, 133
on system resources, 125–129
test doubles dealing with. See Test doubles unit test. See Test doubles
Dependency injection frameworks, 131
Dependency inversion, between layers, 131–132
Deployment
  adverse effects of poor, 49–50
  automated, 50
  double-loop TDD forcing, 222
  manual instructions for, 49
  overview of, 48–49
  testing brown-field business applications, 258
Describe function, BDD-style test framework, 102
Design
  duplicating similar functionality in different classes, 234
  duplicating similar functionality in different methods by, 233
  efficiency in patterns of, 54
  unit tests for better, 79–80
Destructors, 85
Detail, high-level considerations for test, 273
Developer, clarifying meaning of, 1
Developer mind-set, 10
Developer testing
  activities, 2–5
  BDD, ATDD, and specification by example, 15–17
  defining, 6–7
  deleting learning tests in, 241–242
  development process and, 7–8
  high-level considerations, 271–274
  low-level considerations, 274–276
  overview of, 1–2
  quality assurance and, 18
  strategy for, 267–269
  summary, 8
  what they usually do not do, 5–6
Development process, 7–8, 32–33
Diagrams, state, 114
Direct input
  as drivers of testability, 68
  pure functions having no, 69
  stubs controlling, 160
Direct output
  as drivers of testability, 68
  pure functions having no, 69
Document IDs, test-driving search engine design phase, 192–193
  finding words in multiple documents, 197
  introducing ranking, 199–202
  searching more sophisticated documents, 196–197
  searching multiple documents, 195–196
Domain classes, mockist style TDD, 215–216
Domain models, competing duplication in, 234–235
Domain-specific languages (DSLs), testing with, 42
Domain-to-range ratio (DRR), as driver of testability, 77–78
Double-loop TDD, 220–222
Double-mode faults, 147–149
Drivers of testability
  data types and testability, 72–76
  direct input and output, 68
  domain-to-range ratio, 77–78
  indirect input and output, 68–69
  overview of, 67–68
  state, 70–71
  summary, 68
  temporal coupling, 71–72
DRR (domain-to-range ratio), as driver of testability, 77–78
DRY principle: Don’t Repeat Yourself, 52
DSLs (domain specific languages), 42
DST (daylight savings time), 112
Dummies, 171–173, 176
Duplication
  assertions introducing, 92–93
  breeding, 225
  factory classes introducing, 240
  knowledge, 232–235
  mechanical, 228–232
  overview of, 225
  singularity vs., 53
  summary, 235
  taking advantage of, 227
  testable software and, 40–41
  why it is bad, 225–227
Dynamic proxies, mocking frameworks, 178
Index

E
E-mails, testing delivery, 153–154
Edge cases, 110, 111–113
Effectiveness, high-level considerations for test, 271
Efficiency, testability and, 54
Eiffel, 60, 74–76
Elimination of waste, 41–42
Embedded containers, 155
Encapsulated code, 46–47
End-to-end tests
  effectiveness of, 271
  of features, 52
  level of abstraction/detail, 273
  overview of, 34
  preparing brown-field business applications for, 258
  UI tests as, 252–254
End users
  acceptance testing by, 26
  observability of output for, 44
Enforcing contracts, 62–65
Environmental dependence, tests that are not unit tests, 261–262
Equality
  in BDD-style frameworks, 104
  errors in mocking frameworks, 182
  in unit tests, 93–94
Equals method, 93–94, 181–182
Equivalence partitioning, 107–110, 116
Errors
  exceptions in unit tests, 99–102
  forgetting equals method in unit tests, 93–94
  in language of testing, 22
  low-level test considerations, 275
  order of tests in TDD, 85
  temporal coupling, 72
  for tests that are not unit tests, 260–261
  in unit testing frameworks vs., 90
  from violation of contracts, 57
Events, state transition model, 113–114
Exceptions
  in copy/paste programming, 227
  low-level test considerations, 275
  in number of assertions per test, 89–90
  stubs in mocking framework, 183
  in unit tests, 99–102
Execution speed
  critique-based testing of, 10
  in tests that are not unit tests, 261
  unit testing and, 24, 82
Expectations
  configuring stubs, 180–183
  setting, 179–180
  verifying, 186–187
Expected behavior, naming unit tests, 87
ExpectedException rule, JUnit, 100–101
Experimenting, with test names, 88
Exploratory testing, cross-functional teams, 5
External factory, 123–125
Extract method of refactoring, 229–230
F
Factory classes, removing need for comments, 239–240
Factory methods
  controlling dependency between collaborators, 122–123
  as data helpers for tests outside domain of unit tests, 265–266
  removing need for comments, 239–240
Fail-safe activities, as support testing, 11
Failures
  errors in unit tests vs., 90
  software bugs/defects leading to. See Bugs/defects
Faking
  in classic style TDD, 211
  defined, 176
  in mockist style TDD, 216–217
  overview of, 205
  as test double, 162–164
  tests interacting with other systems via, 250–251
Fan-out, isolability as, 51
Fast medium tests, 151
Features
  added complexity of, 52
  BDD-style frameworks, 105
  benefits of testability for, 39–41
Features (continued)

double-loop TDD verifying new, 222
fluent assertion, 97
mocking framework, 178
smallness with respect to number of,
52, 55
unit tests enabling change of, 80
Feedback
benefits of double-loop TDD, 221–222
running almost unit tests with unit tests,
155
with short iterations in TDD, 191
File dependencies, 125–127
Find method

discovering API, 193
finding words in multiple documents, 197
happy path, 194–195
removing duplicate matches, 198–199
searching more sophisticated documents,
196–197
searching multiple documents, 195
Floating point numbers, boundary values,
111
Fluent assertions, 96–97
Format, high-level considerations on test, 273
4.x unit testing framework, 83
Fragmentation, as risk in traditional testing,
12
Frameworks

BDD, Behavior–driven development
(BDD) frameworks
dependency injection, 131
mocking. See Mocking frameworks
possibly deleting tests using older,
242–243
TDD exposing deficiencies in testing, 209
test method names mandated by, 83–84,
86
unit testing. See Unit tests
Functional testing
black box testing similar to, 36
nonfunctional testing vs., 28
overview of, 27
security testing as, 30
Functionality

benefits of testable software, 39–40
critique-based testing of, 10
double-loop TDD verifying finished, 222
duplication of different classes with
similar, 233–234
duplication of different methods with
similar, 232–233
efficiency for, 54
evironmental dependence of tests that
are not unit tests, 261–262
testing in old systems, 46
tests exercising across several systems,
249–251
Functions
encoding business logic out of
preconditions, 74–76
measuring information loss, 77–78
pure functions vs., 69
state verification of, 174
testing exceptions in higher-order, 101
Fundamental test process, 12

G

General properties, generative test results,
144
Generative testing

defined, 149
high-level considerations on format, 273
overview of, 141–143
verifying results, 143–144
Generators, QuickCheck test, 143
Green bar, test-driven development

defined, 191
implementing mockist style TDD, 216–217
inspiration for, 206
turning from red bar to, 205
Groovy, 90, 101
Growing Object-Oriented Software, Guided by
Tests (Freeman & Price), 221
Guard assertions, 90
Guava, contract programming, 63–64

H

Handovers, agile testing with no, 14
Happy path tests
order of tests in TDD, 85
as positive testing, 35
search engine design, 194–195
Heisenbugs, 45
“Hello World” of smoke testing, 33
Heuristics. See Test ideas and heuristics
High-level test considerations, 271–274
How, nonfunctional tests targeting, 28
HtmlUnitDriver, WebDriver testing, 253, 255
HTTP
tests exercising services/components, 248–249
tests that are not unit tests, 260
I
I/O-related errors, nasty test cases, 6
Ignorance, duplicating similar functionality, 233, 234
Ignored tests, deleting, 242
IllegalStateException, 101, 256
Implementation
in classic style TDD, 205, 211
competing duplication in, 234
In-memory databases
almost unit tests using, 152–153
almost unit tests with unit tests and, 156
tests enclosed in transactions vs. tests of, 246
Index, search engine
creating case insensitivity, 203
dealing with punctuation marks, 203
designing, 192–193
discovering API, 193–194
happy path, 194–195
introducing ranking, 199–202
Indirect input
pure functions having no, 69
testability driven by, 68–69
verifying transformations, 169–170
Indirect output
mock objects verifying, 164–169
pure functions having no, 69
testability driven by, 68–69
Information
hiding, 45–46
order of tests in TDD, 85
Initializer, lifecycle of unit tests, 83–85
Inspiration, TDD, 206
Integration. See Continuous integration (CI)
Integration tests
developer testing via, 2–3
functional tests at level of, 27
increasing observability in old code, 46
preparing brown-field business applications for, 258
specification-based techniques for. See Specification-based testing techniques
test level of, 25–26
for tests enclosed in transactions, 246–248
Integrity, CIA security triad model for, 29
Interaction tests
double-loop TDD verifying all, 221–222
test double response to expectations, 179–180
tests of web services, 155–156
verifying indirect output, 164–169
Interactions
arguments against behavior testing, 175–176
in mockist style TDD, 215–217
oververifying in mocking framework, 186–187
spies capturing, 170–171
verifying in mocking framework, 183–185
Interface
mocking, 188
mockist style TDD, 215–216
Intertwining layers, dependencies, 130
Invariants, enforcing contracts, 64–65
Inventory waste, in testing, 42
Inverse functions, 144, 274
Invocation
mockist style TDD, 215–216
test double response to expectations, 179–180
Isolability, 51
Isolation, unit tests, 82, 84
It function, naming BDD-style framework tests, 103
Iterations, 113, 191

J
Jasmine, 104
Jetty, 154–155
JUnit testing framework
  exception testing, 101–102
  ExpectedException rule, 100–101
  matchers determining outcome of assertions, 94–96
  MSTest assertions vs., 89–90
  source code, sample TDD session, 282–284
  test methods, 83
  theory tests, 140

K
Knowledge duplication
  competing domain models, 234–235
  competing implementations, 234
  overview of, 232
  similar functionality in different classes, 233–234
  similar functionality in different methods, 232–233
  summary, 235
Knowledge, order of tests in TDD, 85

L
Large tests, 35
Layers
  dependencies, 129–133
  using mockist style TDD with, 219–220
Laziness
  duplicating functionality in different methods, 233
  reasons for almost unit tests, 151
Legacy code
  controlling dependency using factory, 123
  defining, 3
developer testing strategy for, 268–269
enforcing contracts in, 61
faking, 163–164
information hiding/observability in, 45–46
introducing TDD into, 206–207
safe way of working with, 3–4
using test double, 179
Level of abstraction, testability and, 53–54
Libraries
  as data helpers for more complex tests, 265–266
  implementing contract programming, 63–64
  resources, 277–278
  specialized fluent assertion, 97
  TDD exposing deficiencies in, 209
  in tests that are not unit tests, 259
  UI tests relying on, 252–254
Lifecycle, unit testing framework, 83–85
Lightweight containers, 154–155
Load balancers, 258
Load testing, performance, 28
Log servers, 258
Logging, increasing observability via, 45
Logical concept, unit tests testing single, 82
Login, smoke tests for, 33–34
Low coupling, isolability and, 51
Low-level tests, 52, 274–276

M
Mail servers, in almost unit tests, 153–154
Maintenance
  by developers, 3–4
  nonfunctional testing of, 28
  patching/bug fixing for, 3–4
  smallness of test for, 55
Manual testing, of features, 52
Master database, 258
Matchers
  assertions in unit tests, 94–99
  BDD-style test framework, 103–105
  verifying interactions in mocking framework, 183
Matching arguments, stubs in mocking framework, 181–182
Math package, testing, 47–48
Maximum values for data types, 111
Mechanical duplication
    block copy and paste, 229–230
    constructor copy and paste, 230–231
    copy and paste, 228–229
    method duplication, 231–232
    overview of, 228
    summary, 235
Medium tests, 35, 151
Memory corruption, 111
Messaging middleware, 258
Metadata, unit test methods via, 83
Method duplication, 231–232
Methods
    assertion, 89–90
    cleanup, 84
    controlling dependency using factory, 122–123
    duplication of similar functionality in different, 232–233
    limitations of testing with formal, 42–43
    test, 83–84
Metrics, duplication messing up, 226
Micro-services, dependencies across tiers, 133
Mind-set, in critique-based testing, 10–11
Minimum values for data types, 111
Mirroring business logic, complex stubs, 162
Misuse, of mocking framework, 185–189
Mobile applications, UI tests for, 252–254
Mocha for JavaScript, BDD-style test, 102
Mock objects
    for behavior verification, 174
    defined, 176
    implementing with mocking frameworks. See Mocking frameworks
    oververifying in mocking frameworks, 186–187
    response to expectations, 179–180
    returning mocks, 189
    spies vs., 170–171
    as test doubles, 164–170
    verifying interactions in mocking framework, 183–185
Mocking frameworks
    constructing test doubles, 177–179
    misuse, overuse, and other pitfalls, 185
    mocking concrete classes, 187–188
    mocking value-holding classes, 188
    mocks returning mocks, 189
    oververifying, 186–187
    overview of, 177
    setting expectations, 179–180
    stubbing, 180–183
    summary, 189
    verifying interactions, 183–185
Mockist style TDD. See Test-driven development (TDD) - mockist style
Mockito, 180–184
Modifications, increasing observability, 44
Modularity, isolability and, 51
Moq for C#
    configuring stubs in mocking framework, 180–183
    constructing test doubles, 178
    mocks behaving like spies in, 180
    verifying interactions in mocking framework, 184
MSTest unit testing framework, 83, 89–90
Multitiered applications, dependency across tiers, 133
Mutator, state-based tests, 173–174

N
Naming conventions
    BDD-style tests, 103
    duplication of similar functionality in different methods, 232–233
    method duplication dangers, 231
    removing need for comments, 237
Naming conventions, unit tests
    behavior-driven development-style, 86
    mandated by framework, 86
    overview of, 85–86
    picking naming standard, 87–88
    structuring unit tests, 88–89
    test methods using, 83
    unit of work, state under test, expected behavior, 87
Nasty test cases, 5, 6
Negative testing, 35, 85
Nested contexts, RSpec for Ruby, 102–103
Network performance, tests outside domain of unit tests, 261
Nice mocks, 180
Nomenclature, contract programming, 58
Nonfunctional testing, 28, 30
Normal mocks, 180
Nuking, coding stability for tests that are not unit tests, 259
Null check, enforcing contracts, 65
Null value, boundary values for strings, 111
Nulls
   indicating dummy, 172
   low-level test considerations, 274
Numbers
   finding boundary values for, 111
   low-level test considerations, 275
NUnit testing framework
   constraints and assertions, 94–96
   exception testing, 101
   parameterized tests, 138–139
   test methods, 83
   theory tests, 140

0
Object equality
   asserting in BDD-style tests, 104
   unit test assertion checking for, 93–94
Object-oriented languages
   contracts blending with, 61
   data types/testability in, 72–73
   data types/testability in non, 74–76
   raising level of abstraction, 53
   temporal coupling in, 72
Objectives. See Testing objectives
Objects, dependencies between collaborating, 119–125, 133
Observability
   defined, 55
   test first or test last, 209–210
   as testability quality attribute, 44–48
Obvious implementation, classic style TDD, 205, 211
Optimization, ranking, 201–202
Oracles, 144
Order of tests, TDD, 204
Outcome, naming unit tests to convey expected, 85
Outgrown tests, deleting, 243
Output
   of developers, 1
   observability via developer, 44
Overprocessing waste, incurring in testing, 42
Overuse, mocking framework, 185–189
Overuse, of dummies, 173
Oververifying, in mocking frameworks, 186–187
P
Page Objects, UI tests, 254
Pair programming, and legacy code, 4
Pairwise testing
   beyond, 149
   for combinatorial explosions, 147–149
defined, 149
Pairwise.pl program, 149
Parallel implementations, 227
Parameterized tests
   defined, 149
   overview of, 138–139
   reporting results from, 141
   theories vs., 139–141
   using parameterized stubs, 161–162
Parentheses, expressing intervals, 109
Partial verification, unit tests, 98–99
Partitioning
   boundary value analysis of, 110
   equivalence, 107–110
   knowledge duplication with deliberate, 233
Pass-through tests, mockist style TDD, 218–219
Patching, by developers, 3–4
Paving, 259–260
Payment gateways, 250–251
PCI DSS security standard, 250
Penetration tests, 28
Performance testing
   impact of assertions on, 63
   nonfunctional testing of, 28
not usually done by developers, 5
overview of, 28
of tests that are not unit tests, 261
Persistence operations, tests enclosed in transactions, 246–248
PHPUnit unit testing framework, 83
Pitfalls, of mocking frameworks, 185–189
Portability
  nonfunctional testing of, 28
  running almost unit tests with unit tests, 156
  of unit tests across all environments, 82
Positive testing, 35
Postconditions, enforcing contracts, 59, 64–65
The Pragmatic Programmer, 52
Pragmatic Unit Testing (Hunt & Thomas), 40
Preconditions
  as contract building block, 59
  encoding business logic out of, 74–76
  enforcing contracts with assertions, 62
  enforcing contracts with Guava, 63–64
  enforcing contracts with unit tests, 64–65
  for tests that are not unit tests, 259
Predicates
  configuring stubs in mocking framework, 181–182
  determining outcome of assertions, 94
  high-level test considerations, 274
Prefixes, naming tests, 86, 87
Primitive integer types, boundary values, 111
Privacy, in CIA security triad model, 29
Proactive role, of tester in agile testing, 14
Processes, traditional testing requiring well-defined, 12
Program elements, testable, 43
Programming by Contract
  contract building blocks, 59–60
  contracts defining constraints, 57–58
  enforcing with assertions, 62–63
  enforcing with specialized libraries, 63–64
  enforcing with static analysis, 65
  enforcing with unit tests, 64–65
  overview of, 57
  summary, 65
Programming languages
  efficiency (intent) of, 54
  level of abstraction, 53–54
  minimum and maximum values in, 111
Properties, unit test, 81–82
Provisioning, in tests that are not unit tests, 259–260
Proxies, dynamic, 178
Punctuation marks, search engine, 203
Pure functions, side effects, 69

Q
Quality
  attributes, 28, 43–44
  developer testing for, 6–7
  why we care about testability, 41
Quality assurance
  developer testing and, 18
  in traditional testing, 12–13
QuickCheck, using test generator, 143

R
Randomness, making tests nondeterministic, 143–144
Range, test considerations, 273, 274
Ranking, test-driving search engine designing, 192–193
  introducing, 199–202
  removing duplicate matches to prepare for, 197–198
Readability, logging and, 45
Red bar, test-driven development defined, 191
  implementing mockist style TDD, 216–217
  inspiration for, 206
  never refactor in, 199
  turning into green bar, 205
Redundant tests, deleting, 242
Refactoring
  deleting tests that have not kept up with, 241
  oververifying in mocking frameworks, 186
Index

Refactoring, test-driven development
    dealing with punctuation marks, 203–204
    defined, 191
    introducing ranking, 199–202
    legacy code, 206–207
    order for adding tests, 210
    removing duplicate matches, 198
Regression testing, 30–31, 35
Regression, unit tests preventing, 80, 81
Regulations, critique-based testing of, 10
Relations between objects, dependency as, 119–125, 133
Reliability, performance tests targeting, 28
Repeatability, of unit tests, 82
Reproducibility, controllability paramount to, 48
Resources, CIA security triad model for availability of, 29
Responsiveness, performance tests targeting, 28
RESTful web service, 155–156, 248–249
Rewrites, testing brown-field business applications, 258
Risk, 11, 12
Role, of tester in agile testing, 13–14
Rollbacks, tests enclosed in transactions, 246–247
RSpec for Ruby, BDD-style tests, 102–105

S
Safety, testing for, 11
Scaling, unit tests enabling, 79
Scope
    critique-based testing of, 10
    in functional testing, 27
    unit testing and, 24
Seams, breaking dependencies, 120
Search engine, test-driving
    dealing with punctuation marks, 203–204
    designing, 192–193
    discovering API, 193–194
    finding words in multiple documents, 197
    happy path, 194–195
    ignoring case, 202
    introducing ranking, 199–202
    more sophisticated documents, 196
    removing duplicate matches, 197–199
    searching in multiple documents, 195–196
Security, payment gateways, 250
Security testing, 5, 28–30
Semantics, number of assertions per unit test, 91
Server configuration, 258
Service tests, 248–249, 267–269, 271
Setters, passing in collaborators with, 121
Setup, of higher-level tests, 264–266
Should, starting test name with, 86
Side effects
    faking, 162–163
    implementing empty stub to get rid of, 162
    pure functions and, 69
Simplicity, deleting tests for, 243
Single-mode faults, 146–147, 149
Single Responsibility Principle, 85
Single value, high-level test considerations, 273
Singleton pattern, 69
Singularity
    bottleneck/coupling in, 227
    testability and, 52–53
Small tests, 35
Smallness
    defined, 55
    efficiency and, 54
    level of abstraction and, 53–54
    maintenance and, 55
    reuse and, 54
    singularity and, 52–53
    of test suite in almost unit tests, 151
    as testability quality attribute, 51–52
Smoke testing, 33–34
SMTP port, almost unit tests using mail servers, 153–154
Social dimension, of continuous integration, 4–5
Source code
    beyond unit testing, 287
    data-driven and combinatorial testing, 279–282
    integration tests coupled to, 26
JUnit version, TDD, 282–284
Spock version, TDD, 284–287
  test doubles, 279
  test levels express proximity to, 23–26
  white box vs. black box testing, 22
Special code, in test-driven development, 207
Specialized assertions, unit tests, 94–96
Specification-based testing techniques
  based on decision tables, 115–116
  boundary value analysis, 110
  edge cases/gotchas for some data types, 111–113
  equivalence partitioning, 107–110
  overview of, 107
  state transition testing, 113–114
  summary, 116–117
Specification by example
  as double-loop TDD, 222
  testing style, 15–17
  as tests exercising services/components, 248–249
Speed. See Execution speed
Spies
  defined, 176
  implementing with mocking frameworks, 177
  as test doubles, 170–171
Spike testing, performance, 28
Spock framework
  differentiating stubs and mocks, 178
  mocks behaving like spies in, 180
  parameterized tests, 138
  source code for TDD, 284–287
  using blocks as assertions, 90
  verifying interactions in mocking framework, 185
Spring Boot, starting embedded containers, 155
SQL-compliant in-memory databases, 152–153, 156
Square brackets, expressing intervals, 109
Stability, tests that are not unit tests, 259–260
Stacking, stubs in mocking framework, 182
Startup, complex test, 264
State
  controllability and, 48
  as driver of testability, 70–71
  mock objects shifting focus to, 164
  setting up higher-level tests, 264–265
  temporal coupling vs., 71–72
  unit testing from known, 83–84
  verification of, 173–174, 176
State-based tests, 173–174
State transition testing, 113–114, 116
State under test, 87
Statements, verifying tested code with theories, 139–141
Static analysis, contracts, 65
Stderr (standard err), 255
Stdin (standard input), 255
Stdout (standard output), 255
Steady pace of work, in unit tests, 80
Storage performance, tests outside domain of unit tests, 261
Stored procedures, tests enclosed in transactions, 247
Stress testing, of performance, 28
Strict mocks, 180
Strings
  finding boundary values for, 111–112
  low-level test considerations, 275–276
Structuring
  BDD-style tests, 102–103
  unit testing frameworks, 88–89
Stubs
  configuring in mocking framework, 180–183
  defined, 176
  flexibility of, 161–162
  getting rid of side effects with, 162
  implementing with mocking frameworks, 177–179
  as test doubles, 159–162
Subsystems, TDD for legacy code, 206
Suppliers, in contract programming
  contract building blocks and, 59–60
  implementing contracts, 60–62
  overview of, 57–58
Support, testing to, 11
Switch coverage, state transition testing, 114
Syntax
  BDD-style frameworks with fluent, 105
  number of assertions per unit test, 91–92
System
  CIA security triad model for integrity of,
  29
clock, 127–128
resource dependencies, 125–129, 133
System boundary, mockist style TDD,
  214–215
System tests
  considering effectiveness of, 271
  considering level of abstraction/detail, 273
  end-to-end testing vs., 34
  of features, 52
  functional tests at level of, 27
  increasing observability in old code, 46
  test level of, 26
  UI tests as, 252–254
Systems
  tests invoking, 255–257
  tests that interact with other, 249–251

T
Tables
  decision, 115–116
  double-mode faults, 147–149
  single-mode faults, 147
Tabular/data-driven tests, 273
Target audience, tests that are not unit tests,
  262–263
TDD. See Test-driven development (TDD) -
  classic style; Test-driven development
  (TDD) - mockist style
Team
  agile testing experts on development,
  13–14
  automated acceptance tests written by, 27
  TDD exposing deficiencies in, 209
Teardown (cleanup methods), 84
Technical debt, of intertwining layers, 130
Technical side, of continuous integration, 4–5
Technology-facing tests, 32–33
Temporal coupling, 71–72, 263–264
Termination of failed assertions, 61
Test automation pyramid, 267–269
Test classes, lifecycle of unit tests, 84–85
Test code, working with
  commenting tests, 237–241
  deleting tests, 241–243
  overview of, 237
  summary, 243–244
Test coverage, deleting duplicated code to
  increase, 226–227
Test doubles
  behavior verification, 174–176
  constructing with mocking frameworks.
    See Mocking frameworks
  dealing with dependencies in unit tests,
    159
dummies, 171–173
fakes, 162–164
mock objects, 164–170
for more complex tests, 267
replacing entire system with, 251
source code, 279
spies, 170–171
state verification, 173–174
stubs, 159–162
summary, 176
Test-driven development (TDD) - classic style
  alternatives and inspiration, 206
  challenges, 206–209
  order of tests, 204
  overview of, 191
  red to green bar strategies, 205
  resources on, 206
Spock version source code, 284–287
summary, 210–211
switching between mockist and, 220
test-driving simple search engine. See
  Search engine, test-driving
test first or test last, 209–210
Test-driven development (TDD)—mockist
  style
  adding more tests, 219–220
  different approach to design, 213–214
  double-loop TDD, 220–222
  focusing on design of system, 213
  summary, 223
  switching between classic and, 220
test-driving customer registration,
Index

Test first or test last, TDD, 209–210
Test fixture, unit tests, 83–84
Test ideas and heuristics
  high-level considerations, 271–274
  low-level considerations, 274–276
  overview of, 271
Test initializers
  BDD-style frameworks, 103
  for tests that are not unit tests, 259
  unit tests, 83–85
Test levels
  acceptance test, 26
  defined, 23
  integration test, 25–26
  putting to work, 31
  system test, 26
  unit test, 23–25
Test recipes, high-level considerations, 272
Test types
  defined, 26–27
  functional testing, 27
  nonfunctional testing, 28
  performance testing, 28
  putting to work, 31
  regression testing, 30–31
  requiring different amounts of state, 48
  security testing, 28–30
Testability
  benefits of, 39–43
  from developer's perspective, 37
  reminder about, 55
  summary, 55
  test-driven development exposing deficiencies in, 209
  test first or test last, 209–210
  testable software, 37–39
Testability, as quality attribute
  controllability, 48–51
  observability, 44–48
  overview of, 43–44
  program elements, 43
  smallness, 51–55
Testable software, The Big Ball of Mud vs., 37–39
Test-driven Development by Example (Beck), 205, 206
Tested object
  in behavior verification, 174–175
  creating fakes, 162–164
  creating stubs in unit tests, 160
  in state verification, 173–174
  verifying indirect input transformations, 169–170
  verifying indirect output with mock objects, 165
Tester mind-set, 5, 10
Testing behavior, 176
Testing objectives
  of test types. See Test types
testing to critique, 10–11
testing to support, 11
testing vs. checking, 9–10
Testing styles
  Agile testing, 13–15
  BDD, ATDD, and specification by example, 15–17
  traditional testing, 11–13
Tests enclosed in transactions, 246–248
Tests exercising services/components, 248–249
Tests invoking system, 255–257
Theory tests
  adding generative testing to, 142–143
  defined, 149
  overview of, 139–141
  reporting results from, 141
Third parties
  reuse by implementing, 54
  tests interacting with other systems, 250–251
Throughput
  and duplication, 227
  performance tests targeting, 28
Tiers, dependencies across, 132, 133
 Tight coupling, 175–176
Time
  boundary values for, 112
  unit tests freeing up testing, 80–81
Tools
checking vs. testing, 9–10
resources for, 277–278
Toyota Production System, elimination of
waste, 41–42
Traditional testing, 11–13, 19
Transactions, tests enclosed in, 246–248
Transformation Priority Premise (Martin), 206
Transitions, state, 113–114
Transparency, deleting tests for, 243
Triangulation, 205, 211, 243
Triple A test structure, 88–89
Truthfulness
deleting tests for, 243
of test result, 273
Try-catch statement, testing exceptions in
higher-order functions, 101
Types, limitations of testing with, 43

U
UAT (user acceptance testing), 26
Ubiquitous language, 15–17
UI (user interface) tests, 252–254, 267–269
Unclassified tests, almost unit tests as, 151
Unicode characters, strings, 112
Unit of work, 24, 87
Unit tests
in agile testing, 15
assertion methods, 89–99
with BDD-style frameworks, 102–105
characteristics of tests that are not, 257–263
in characterization testing, 34
data-driven. See Data-driven and
combinatorial testing
definition of, 81–83
developers writing, 2
effectiveness of, 271
enforcing contracts with, 64–65
for exceptions, 99–102
functional tests as, 27
level of abstraction/detail, 273
lifecycle of, 83–85
naming, 85–88
in old system, 46
overview of, 79
reasons to perform, 79–81
as small tests, 35
specification-based. See Specification-
based testing techniques
structuring, 88–89
summary, 105–106
system tests vs., 26
TDD exposing deficiencies in, 209
in test automation pyramid, 267–269
test level of, 23–25
tests that are almost. See Almost unit tests
in traditional testing, 12–13
unit testing frameworks not running only, 83
Unit tests, beyond
developer testing strategy decisions,
267–269
overview of, 245
pointers and practices, 263–267
source code, 287
summary, 269–270
test independence, 263–264
tests enclosed in transactions, 246–248
tests exercising services/components,
248–249
tests interacting with other systems,
249–251
tests invoking system, 255–257
tests running through user interface,
252–255
tests that are not unit tests, 257–263
@Unroll annotation, parameterized tests,
138
Usability testing
nonfunctional testing, 28
not usually done by developers, 5
User acceptance testing (UAT), 26
User interface (UI) tests, 252–254, 267–269
Users, critique-based testing of, 10
Utility methods, duplication, 231–232,
265–266
V
Validation, contracts not replacing, 57
Value-holding classes, 188
Values
dummies indicated by simple, default, 171–172
high-level test considerations, 273–274
stubs, 160, 161–162
Variable delays, UI tests failing, 252
Variables, removing need for comments, 238
Verbosity, of assertions in unit tests, 92–93
Verification. See also Developer testing
of behavior, 174–175
The Big Ball of Mud preventing, 38–39
of contracts, 62–63
in generative testing, 143–144
of indirect output with mock objects, 164–169
in mocking framework, 183–187
in more complex tests, 266
of state, 173–174
in testable software, 39
with theories, 139–141
in traditional testing, 11–13
in unit testing, 82, 98–99
Verify method, 164–169
Virtualization, tests that are not unit tests, 259–260
Vocabulary, test key terms
Agile Testing Quadrants, 32–33
black box testing, 22–23
characterization testing, 34–35
d e nd-to-end testing, 34
errors, defects, and failures, 22
negative testing, 35
overview of, 21
positive testing, 35
putting test levels/test types to work, 31
small, medium, and large tests, 35
smoke testing, 33–34
summary, 36
test levels, 23–26
test types, 26–31
white box testing, 22–23
W
Waste, elimination of, 41–42
Wasteful, tests as, 41–43
Web applications
reality of layers in, 130
UI tests for, 252–254
Web frameworks, raising level of abstraction, 53
Web services, almost unit tests of, 155–156
WebDriver testing, 253–255, 259
“What,” functional tests targeting, 28
White box testing, 22–23, 52
Word frequency, and ranking, 200–202
Working Effectively with Legacy Code
(Feathers), 3
X
XCTest unit testing framework, 83
XUnit.net framework, 85
Z
Zero-one-many, test coverage of, 274