Graham Lee

Test-Driven iOS Development

Developer's Library

Test-Driven iOS Development

Developer's Library

ESSENTIAL REFERENCES FOR PROGRAMMING PROFESSIONALS

Developer's Library books are designed to provide practicing programmers with unique, high-quality references and tutorials on the programming languages and technologies they use in their daily work.

All books in the *Developer's Library* are written by expert technology practitioners who are especially skilled at organizing and presenting information in a way that's useful for other programmers.

Key titles include some of the best, most widely acclaimed books within their topic areas:

PHP & MySQL Web Development
Luke Welling & Laura Thomson
ISBN 978-0-672-32916-6

MySQL
Paul DuBois
ISBN-13: 978-0-672-32938-8

Linux Kernel Development
Robert Love
ISBN-13: 978-0-672-32946-3

Python Essential Reference
David Beazley
ISBN-13: 978-0-672-32862-6

Programming in Objective-C
Stephen G. Kochan
ISBN-13: 978-0-321-56615-7

PostgreSQL
Korry Douglas
ISBN-13: 978-0-672-33015-5

Developer's Library books are available at most retail and online bookstores, as well as by subscription from Safari Books Online at **safari.informit.com**

Developer's Library
informit.com/devlibrary

Test-Driven iOS Development

Graham Lee

Addison-Wesley

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

Library of Congress Cataloging-in-Publication Data is on file

ISBN-13: 978-0-32-177418-7
ISBN-10: 0-32-177418-3

Text printed in the United States on recycled paper at R.R. Donnelley in Crawfordsville, Indiana.

First printing, April 2012

Editor-in-Chief
Mark Taub

Senior Acquisitions Editor
Trina MacDonald

Managing Editor
Kristy Hart

Project Editor
Andy Beaster

Copy Editor
Barbara Hacha

Indexer
Tim Wright

Proofreader
Paula Lowell

Technical Reviewers
Richard Buckle

Patrick Burleson

Andrew Ebling

Alan Francis

Rich Wardwell

Publishing Coordinator
Olivia Basegio

Book Designer
Gary Adair

Compositor
Gloria Schurick

❖

This book is for anyone who has ever shipped a bug. You're in great company.

❖

This page intentionally left blank

Contents at a Glance

Table of Contents

Preface

My experience of telling other developers about test-driven development for Objective-C came about almost entirely by accident. I was scheduled to talk at a conference on a different topic, where a friend of mine was talking on TDD. His wife had chosen (I assume that's how it works; I'm no expert) that weekend to give birth to their twins, so Chuck—who commissioned the book you now hold in your hands—asked me if I wouldn't mind giving that talk, too. Thus began the path that led ultimately to the year-long project of creating this book.

It's usually the case that reality is not nearly as neat as the stories we tell each other about reality. In fact, I had first encountered unit tests a number of years previously. Before I was a professional software engineer, I was a tester for a company whose product was based on GNUstep (the Free Software Foundation's version of the Cocoa libraries for Linux and other operating systems). Unit testing, I knew then, was a way to show that little bits of a software product worked properly, so that hopefully, when they were combined into big bits of software, those big bits would work properly, too.

I took this knowledge with me to my first programming gig, as software engineer working on the Mac port of a cross-platform security product. (Another simplification—I had, a few years earlier, taken on a six-week paid project to write a LISP program. We've all done things we're not proud of.) While I was working this job, I went on a TDD training course, run by object-oriented programming conference stalwart Kevlin Henney, editor of *97 Things Every Programmer Should Know*, among other things. It was here that I finally realized that the point of test-driven development was to make me more confident about my code, and more confident about changing my code as I learned more. The time had finally arrived where I understood TDD enough that I could start learning from my own mistakes, make it a regular part of my toolbox, and work out what worked for me and what didn't. After a few years of that, I was in a position where I could say yes to Chuck's request to give the talk.

It's my sincere hope that this book will help you get from discovering unit testing and test-driven development to making it a regular part of how you work, and that you get there in less time than the five years or so it took me. Plenty of books have been written about unit testing, including by the people who wrote the frameworks and designed the processes. These are good books, but they don't have anything specifically to say to Cocoa Touch developers. By providing examples in the Objective-C language, using Xcode and related tools, and working with the Cocoa idioms, I hope to make the principles behind test-driven development more accessible and more relevant to iOS developers.

Ah, yes—the tools. There are plenty of ways to write unit tests, depending on different features in any of a small hoard of different tools and frameworks. Although I've covered some of those differences here, I decided to focus almost exclusively on the capabilities Apple supplies in Xcode and the OCUnit framework. The reason is simply one of applicability; anyone who's interested in trying out unit tests or TDD can get on straight

away with just the knowledge in this book, the standard tools, and a level of determination. If you find aspects of it lacking or frustrating, you can, of course, investigate the alternatives or even write your own—just remember to test it!

One thing my long journey to becoming a test-infected programmer has taught me is that the best way to become a better software engineers is to talk to other practitioners. If you have any comments or suggestions on what you read here, or on TDD in general, please feel free to find me on Twitter (I'm @iamleeg) and talk about it.

Acknowledgments

It was Isaac Newton who said, "If I have seen a little further it is by standing on the shoulders of giants," although he was (of course!) making use of a metaphor that had been developed and refined by centuries of writers. Similarly, this book was not created in a vacuum, and a complete list of those giants on whom I have stood would begin with Ada, Countess Lovelace, and end countless pages later. A more succinct, relevant, and bearable list of acknowledgements must begin with all of the fine people at Pearson who have all helped to make this book publishable and available: Chuck, Trina, and Olivia all kept me in line, and my technical reviewers—Saul, Tim, Alan, Andrew, two Richards, Simon, Patrick, and Alexander—all did sterling work in finding the errors in the manuscript. If any remain, they are, of course, my fault. Andy and Barbara turned the scrawls of a programmer into English prose.

Kent Beck designed the xUnit framework, and without his insight I would have had nothing to write about. Similarly, I am indebted to the authors of the Objective-C version of xUnit, Sente SA. I must mention the developer tools team at Apple, who have done more than anyone else to put unit testing onto the radar (if you'll pardon the pun) of iOS developers the world over. Kevlin Henney was the person who, more than anyone else, showed me the value of test-driven development; thank you for all those bugs that I didn't write.

And finally, Freya has been supportive and understanding of the strange hours authors tend to put in—if you're reading this in print, you'll probably see a lot more of me now.

About the Author

Graham Lee's job title is "Smartphone Security Boffin," a role that requires a good deal of confidence in the code he produces. His first exposure to OCUnit and unit testing came around six years ago, as test lead on a GNUstep-based server application. Before iOS became the main focus of his work, Graham worked on applications for Mac OS X, NeXTSTEP, and any number of UNIX variants.

This book is the second Graham has written as part of his scheme to learn loads about computing by trying to find ways to explain it to other people. Other parts of this dastardly plan include speaking frequently at conferences across the world, attending developer meetings near to his home town of Oxford, and volunteering at the Swindon Museum of Computing.

1

About Software Testing and Unit Testing

To gain the most benefit from unit testing, you must understand its purpose and how it can help improve your software. In this chapter, you learn about the "universe" of software testing, where unit testing fits into this universe, and what its benefits and drawbacks are.

What Is Software Testing For?

A common goal of many software projects is to make some profit for someone. The usual way in which this goal is realized is directly, by selling the software via the app store or licensing its use in some other way. Software destined for in-house use by the developer's business often makes its money indirectly by improving the efficiency of some business process, reducing the amount of time paid staff must spend attending to the process. If the savings in terms of process efficiency is greater than the cost of developing the software, the project is profitable. Developers of open source projects often sell support packages or use the software themselves: In these cases the preceding argument still applies.

So, economics 101: If the goal of a software project is to make profit—whether the end product is to be sold to a customer or used internally—it must provide some value to the user greater than the cost of the software in order to meet that goal and be successful. I realize that this is not a groundbreaking statement, but it has important ramifications for software testing.

If testing (also known as *Quality Assurance*, or QA) is something we do to support our software projects, it must *support the goal of making a profit*. That's important because it automatically sets some constraints on how a software product must be tested: If the testing will cost so much that you lose money, it isn't appropriate to do. But testing software can show that the product works; that is, that the product contains the valuable features expected by your customers. If you can't demonstrate that value, the customers may not buy the product.

Notice that the purpose of testing is to show that the product works, not discover bugs. It's Quality *Assurance*, not Quality *Insertion*. Finding bugs is usually bad. Why? Because it costs money to fix bugs, and that's money that's being wasted because you were being paid to write the software without bugs in in the first place. In an ideal world, you might think that developers just write bug-free software, do some quick testing to demonstrate there are no bugs, and then we upload to iTunes Connect and wait for the money to roll in. But hold on: Working like that might introduce the same cost problem, in another way. How much longer would it take you to write software that you knew, before it was tested, would be 100% free of bugs? How much would that cost?

It seems, therefore, that appropriate software testing is a compromise: balancing the level of control needed on development with the level of checking done to provide some confidence that the software works without making the project costs unmanageable. How should you decide where to make that compromise? It should be based on reducing the risk associated with shipping the product to an acceptable level. So the most "risky" components—those most critical to the software's operation or those where you think most bugs might be hiding—should be tested first, then the next most risky, and so on until you're happy that the amount of risk remaining is not worth spending more time and money addressing. The end goal should be that the customer can see that the software does what it ought, and is therefore worth paying for.

Who Should Test Software?

In the early days of software engineering, projects were managed according to the "waterfall model" (see Figure 1.1).[1] In this model, each part of the development process was performed as a separate "phase," with the signed-off output of one phase being the input for the next. So the product managers or business analysts would create the product requirements, and after that was done the requirements would be handed to designers and architects to produce a software specification. Developers would be given the specification in order to produce code, and the code would be given to testers to do quality assurance. Finally the tested software could be released to customers (usually initially to a select few, known as *beta testers*).

1. In fact, many software projects, including iOS apps, are still managed this way. This fact shouldn't get in the way of your believing that the waterfall model is an obsolete historical accident.

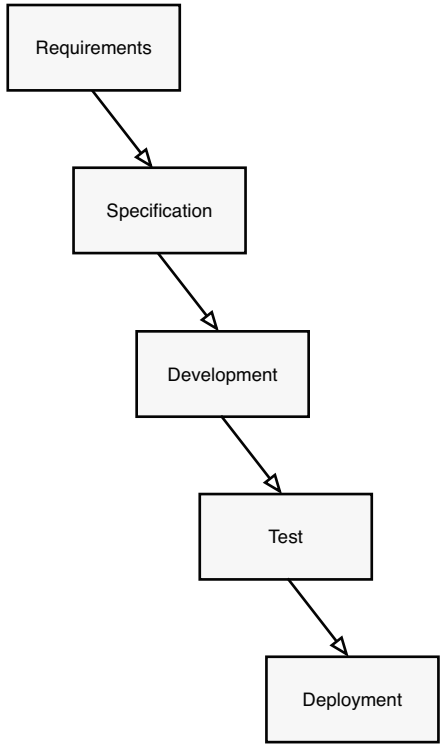

Figure 1.1 The phases of development in the waterfall software project management process.

This approach to software project management imposes a separation between coders and testers, which turns out to have both benefits and drawbacks to the actual work of testing. The benefit is that by separating the duties of development and testing the code, there are more people who can find bugs. We developers can sometimes get attached to the code we've produced, and it can take a fresh pair of eyes to point out the flaws. Similarly, if any part of the requirements or specification is ambiguous, a chance exists that the tester and developer interpret the ambiguity in different ways, which increases the chance that it gets discovered.

The main drawback is cost. Table 1.1, reproduced from *Code Complete*, 2nd Edition, by Steve McConnell (Microsoft Press, 2004), shows the results of a survey that evaluated the cost of fixing a bug as a function of the time it lay "dormant" in the product. The table shows that fixing bugs at the end of a project is the most expensive way to work, which makes sense: A tester finds and reports a bug, which the developer must then interpret and attempt to locate in the source. If it's been a while since the developer worked on that project, then the developer must review the specifications and the code. The bug-fix version of the code must then be resubmitted for testing to demonstrate that the issue has been resolved.

Table 1.1 **Cost of Fixing Bugs Found at Different Stages of the Software Development Process**

Cost of Bugs	Time Detected				
Time Introduced	**Requirements**	**Architecture**	**Coding**	**System Test**	**Post-Release**
Requirements	1	3	5–10	10	10–100
Architecture	-	1	10	15	25–100
Coding	-	-	1	10	10–25

Where does this additional cost come from? A significant part is due to the communication between different teams: your developers and testers may use different terminology to describe the same concepts, or even have entirely different mental models for the same features in your app. Whenever this occurs, you'll need to spend some time clearing up the ambiguities or problems this causes.

The table also demonstrates that the cost associated with fixing bugs at the end of the project depends on how early the bug was injected: A problem with the requirements can be patched up at the end only by rewriting a whole feature, which is a very costly undertaking. This motivates waterfall practitioners to take a very conservative approach to the early stages of a project, not signing off on requirements or specification until they believe that every "i" has been dotted and every "t" crossed. This state is known as *analysis paralysis*, and it increases the project cost.

Separating the developers and testers in this way also affects the type of testing that is done, even though there isn't any restriction imposed. Because testers will not have the same level of understanding of the application's internals and code as the developers do, they will tend to stick to "black box" testing that treats the product as an opaque unit that can be interacted with only externally. Third-party testers are less likely to adopt "white box" testing approaches, in which the internal operation of the code can be inspected and modified to help in verifying the code's behavior.

The kind of test that is usually performed in a black box approach is a *system test*, or *integration test*. That's a formal term meaning that the software product has been taken as a whole (that is, the system is integrated), and testing is performed on the result. These tests usually follow a predefined plan, which is the place where the testers earn their salary: They take the software specification and create a series of test cases, each of which describes the steps necessary to set up and perform the test, and the expected result of doing so. Such tests are often performed manually, especially where the result must be interpreted by the tester because of reliance on external state, such as a network service or the current date. Even where such tests can be automated, they often take a long time to run: The entire software product and its environment must be configured to a known baseline state before each test, and the individual steps may rely on time-consuming interactions with a database, file system, or network service.

Beta testing, which in some teams is called *customer environment testing*, is really a special version of a system test. What is special about it is that the person doing the testing probably isn't a professional software tester. If any differences exist between the tester's system configuration or environment and the customer's, or use cases that users expect to use and the project team didn't consider, this will be discovered in beta testing, and any problems associated with this difference can be reported. For small development teams, particularly those who cannot afford to hire testers, a beta test offers the first chance to try the software in a variety of usage patterns and environments.

Because the beta test comes just before the product should ship, dealing with beta feedback sometimes suffers as the project team senses that the end is in sight and can smell the pizza at the launch party. However, there's little point in doing the testing if you're not willing to fix the problems that occur.

Developers can also perform their own testing. If you have ever pressed Build & Debug in Xcode, you have done a type of white-box testing: You have inspected the internals of your code to try to find out more about whether its behavior is correct (or more likely, why it isn't correct). Compiler warnings, the static analyzer, and Instruments are all applications that help developers do testing.

The advantages and disadvantages of developer testing almost exactly oppose those of independent testing: When developers find a problem, it's usually easier (and cheaper) for them to fix it because they already have some understanding of the code and where the bug is likely to be hiding. In fact, developers can test as they go, so that bugs are found very soon after they are written. However, if the bug is that the developer doesn't understand the specification or the problem domain, this bug will not be discovered without external help.

Getting the Requirements Right

The most egregious bug I have written (to date, and I hope ever) in an application fell into the category of "developer doesn't understand requirements." I was working on a systems administration tool for the Mac, and because it ran outside any user account, it couldn't look at the user settings to decide what language to use for logging. It read the language setting from a file. The file looked like this:

```
LANGUAGE=English
```

Fairly straightforward. The problem was that some users of non-English languages were reporting that the tool was writing log files in English, so it was getting the choice of language wrong. I found that the code for reading this file was very tightly coupled to other code in the tool, so set about breaking dependencies apart and inserting unit tests to find out how the code behaved. Eventually, I discovered the problem that was occasionally causing the language check to fail and fixed it. All of the unit tests pass, so the code works, right? Actually, wrong: It turned out that I didn't know the file can sometimes look at this:

```
LANGUAGE=en
```

> Not only did I not know this, but neither did my testers. In fact it took the application crashing on a customer's system to discover this problem, even though the code was covered by unit tests.

When Should Software Be Tested?

The previous section gave away the answer to the preceding question to some extent—the earlier a part of the product can be tested, the cheaper it will be to find any problems that exist. If the parts of the application available at one stage of the process are known to work well and reliably, fewer problems will occur with integrating them or adding to them at later stages than if all the testing is done at the end. However, it was also shown in that section that software products are traditionally only tested at the end: An explicit QA phase follows the development, then the software is released to beta testers before finally being opened up for general release.

Modern approaches to software project management recognize that this is deficient and aim to continually test all parts of the product at all times. This is the main difference between "agile" projects and traditionally managed projects. Agile projects are organized in short stints called *iterations* (sometimes *sprints*). At every iteration, the requirements are reviewed; anything obsolete is dropped and any changes or necessary additions are made. The most important requirement is designed, implemented, and tested in that iteration. At the end of the iteration, the progress is reviewed and a decision made as to whether to add the newly developed feature to the product, or add requirements to make changes in future iterations. Crucially, because the agile manifesto (http://agilemanifesto.org/) values "individuals and interactions over processes and tools," the customer or a representative is included in all the important decisions. There's no need to sweat over perfecting a lengthy functional specification document if you can just ask the user how the app should work—and to confirm that the app does indeed work that way.

In agile projects then, all aspects of the software project are being tested all the time. The customers are asked at every implementation what their most important requirements are, and developers, analysts, and testers all work together on software that meets those requirements. One framework for agile software projects called *Extreme Programming* (or XP) goes as far as to require that developers unit test their code and work in pairs, with one "driving" the keyboard while the other suggests changes, improvements, and potential pitfalls.

So the real answer is that software should be tested all the time. You can't completely remove the chance that users will use your product in unexpected ways and uncover bugs you didn't address internally—not within reasonable time and budget constraints, anyway. But you can automatically test the basic stuff yourself, leaving your QA team or beta testers free to try out the experimental use cases and attempt to break your app in new and ingenious ways. And you can ask at every turn whether what you're about to

do will add something valuable to your product and increase the likelihood that your customers will be satisfied that your product does what the marketing text said it would.

Examples of Testing Practices

I have already described system testing, where professional testers take the whole application and methodically go through the use cases looking for unexpected behavior. This sort of testing can be automated to some extent with iOS apps, using the UI Automation instrument that's part of Apple's Instruments profiling tool.

System tests do not always need to be generic attempts to find any bug that exists in an application; sometimes the testers will have some specific goal in mind. *Penetration testers* are looking for security problems by feeding the application with malformed input, performing steps out of sequence, or otherwise frustrating the application's expectation of its environment. *Usability testers* watch users interacting with the application, taking note of anything that the users get wrong, spend a long time over, or are confused by. A particular technique in usability testing is *A/B Testing*: Different users are given different versions of the application and the usages compared statistically. Google is famous for using this practice in its software, even testing the effects of different shades of color in their interfaces. Notice that usability testing does not need to be performed on the complete application: A mock-up in Interface Builder, Keynote, or even on paper can be used to gauge user reaction to an app's interface. The lo-fi version of the interface might not expose subtleties related to interacting with a real iPhone, but they're definitely much cheaper ways to get early results.

Developers, particularly on larger teams, submit their source code for review by peers before it gets integrated into the product they're working on. This is a form of white-box testing; the other developers can see how the code works, so they can investigate how it responds to certain conditions and whether all important eventualities are taken into account. Code reviews do not always turn up logic bugs; I've found that reviews I have taken part in usually discover problems adhering to coding style guidelines or other issues that can be fixed without changing the code's behavior. When reviewers are given specific things to look for (for example, a checklist of five or six common errors—retain count problems often feature in checklists for Mac and iOS code) they are more likely to find bugs in these areas, though they may not find any problems unrelated to those you asked for.

Where Does Unit Testing Fit In?

Unit testing is another tool that developers can use to test their own software. You will find out more about how unit tests are designed and written in Chapter 3, "How to Write a Unit Test," but for the moment it is sufficient to say that unit tests are small pieces of code that test the behavior of other code. They set up the preconditions, run the code under test, and then make *assertions* about the final state. If the assertions are valid (that is, the conditions tested are satisfied), the test passes. Any deviation from the

asserted state represents a failure, including exceptions that stop the test from running to completion.[2]

In this way, unit tests are like miniature versions of the test cases written by integration testers: They specify the steps to run the test and the expected result, but they do so in code. This allows the computer to do the testing, rather than forcing the developer to step through the process manually. However, a good test is also good documentation: It describes the expectations the tester had of how the code under test would behave. A developer who writes a class for an application can also write tests to ensure that this class does what is required. In fact, as you will see in the next chapter, the developer can also write tests *before* writing the class that is being tested.

Unit tests are so named because they test a single "unit" of source code, which, in the case of object-oriented software, is usually a class. The terminology comes from the compiler term "translation unit," meaning a single file that is passed to the compiler. This means that unit tests are naturally white-box tests, because they take a single class out of the context of the application and evaluate its behavior independently. Whether you choose to treat that class as a black box, and only interact with it via its public API, is a personal choice, but the effect is still to interact with a small portion of the application.

This fine granularity of unit testing makes it possible to get a very rapid turnaround on problems discovered through running the unit tests. A developer working on a class is often working in parallel on that class's tests, so the code for that class will be at the front of her mind as she writes the tests. I have even had cases where I didn't need to run a unit test to know that it would fail and how to fix the code, because I was still thinking about the class that the test was exercising. Compare this with the situation where a different person tests a use case that the developer might not have worked on for months. Even though unit testing means that a developer is writing code that won't eventually end up in the application, this cost is offset by the benefit of discovering and fixing problems before they ever get to the testers.

Bug-fixing is every project manager's worst nightmare: There's some work to do, the product can't ship until it's done, but you can't plan for it because you don't know how many bugs exist and how long it will take the developers to fix them. Looking back at Table 1.1, you will see that the bugs fixed at the end of a project are the most expensive to fix, and that there is a large variance in the cost of fixing them. By factoring the time for writing unit tests into your development estimates, you can fix some of those bugs as you're going and reduce the uncertainty over your ship date.

Unit tests will almost certainly be written by developers because using a testing framework means writing code, working with APIs, and expressing low-level logic: exactly the things that developers are good at. However it's not necessary for the same developer to write a class and its tests, and there are benefits to separating the two tasks.

2. The test framework you use may choose to report assertion failures and "errors" separately, but that's okay. The point is that you get to find out the test can't be completed with a successful outcome.

A senior developer can specify the API for a class to be implemented by a junior developer by expressing the expected behavior as a set of tests. Given these tests, the junior developer can implement the class by successively making each test in the set pass.

This interaction can also be reversed. Developers who have been given a class to use or evaluate but who do not yet know how it works can write tests to codify their assumptions about the class and find out whether those assumptions are valid. As they write more tests, they build a more complete picture of the capabilities and behavior of the class. However, writing tests for existing code is usually harder than writing tests and code in parallel. Classes that make assumptions about their environment may not work in a test framework without significant effort, because dependencies on surrounding objects must be replaced or removed. Chapter 11, "Designing for Test-Driven Development" covers applying unit testing to existing code.

Developers working together can even switch roles very rapidly: One writes a test that the other codes up the implementation for; then they swap, and the second developer writes a test for the first. However the programmers choose to work together is immaterial. In any case, a unit test or set of unit tests can act as a form of documentation expressing one developer's intent to another.

One key advantage of unit testing is that running the tests is automated. It may take as long to write a good test as to write a good plan for a manual test, but a computer can then run hundreds of unit tests per second. Developers can keep all the tests they've ever used for an application in their version control systems alongside the application code, and then run the tests whenever they want. This makes it very cheap to test for *regression* bugs: bugs that had been fixed but are reintroduced by later development work. Whenever you change the application, you should be able to run all the tests in a few seconds to ensure that you didn't introduce a regression. You can even have the tests run automatically whenever you commit source code to your repository, by a *continuous integration* system as described in Chapter 4, "Tools for Testing."

Repeatable tests do not just warn you about regression bugs. They also provide a safety net when you want to edit the source code without any change in behavior—when you want to *refactor* the application. The purpose of refactoring is to tidy up your app's source or reorganize it in some way that will be useful in the future, but without introducing any new functionality, or bugs! If the code you are refactoring is covered by sufficient unit tests, you know that any differences in behavior you introduce will be detected. This means that you can fix up the problems now, rather than trying to find them before (or after) shipping your next release.

However, unit testing is not a silver bullet. As discussed earlier, there is no way that developers can meaningfully test whether they understood the requirements. If the same person wrote the tests and the code under test, each will reflect the same preconceptions and interpretation of the problem being solved by the code. You should also appreciate that no good metrics exist for quantifying the success of a unit-testing strategy. The only popular measurements—code coverage and number of passing tests—can both be changed without affecting the quality of the software being tested.

Going back to the concept that testing is supposed to reduce the risk associated with deploying the software to the customer, it would be really useful to have some reporting tool that could show how much risk has been mitigated by the tests that are in place. The software can't really know what risk you place in any particular code, so the measurements that are available are only approximations to this risk level.

Counting tests is a very naïve way to measure the effectiveness of a set of tests. Consider your annual bonus—if the manager uses the number of passing tests to decide how much to pay you, you could write a single test and copy it multiple times. It doesn't even need to test any of your application code; a test that verifies the result `"1==1"` would add to the count of passing tests in your test suite. And what is a reasonable number of tests for any application? Can you come up with a number that all iOS app developers should aspire to? Probably not—I can't. Even two developers each tasked with writing the same application would find different problems in different parts, and would thus encounter different levels of risk in writing the app.

Measuring code coverage partially addresses the problems with test counting by measuring the amount of application code that is being executed when the tests are run. This now means that developers can't increase their bonuses by writing meaningless tests—but they can still just look for "low-hanging fruit" and add tests for that code. Imagine increasing code coverage scores by finding all of the `@synthesize` property definitions in your app and testing that the getters and setters work. Sure, as we'll see, these tests do have value, but they still aren't the most valuable use of your time.

In fact, code coverage tools specifically weigh against coverage of more complicated code. The definition of "complex" here is a specific one from computer science called *cyclomatic complexity*. In a nutshell, the cyclomatic complexity of a function or method is related to the number of loops and branches—in other words, the number of different paths through the code.

Take two methods: `-methodOne` has twenty lines with no `if`, `switch`, `?:` expressions or loops (in other words, it is minimally complex). The other method, `-methodTwo:(BOOL)flag` has an `if` statement with 10 lines of code in each branch. To fully cover `-methodOne` only needs one test, but you must write two tests to fully cover `-methodTwo:`. Each test exercises the code in one of the two branches of the `if` condition. The code coverage tool will just report how many lines are executed—the same number, twenty, in each case—so the end result is that it is harder to improve code coverage of more complex methods. But it is the complex methods that are likely to harbor bugs.

Similarly, code coverage tools don't do well at handling special cases. If a method takes an object parameter, whether you test it with an initialized object or with nil, it's all the same to the coverage tool. In fact, maybe both tests are useful; that doesn't matter as far as code coverage is concerned. Either one will run the lines of code in the method, so adding the other doesn't increase the coverage.

Ultimately, you (and possibly your customers) must decide how much risk is present in any part of the code, and how much risk is acceptable in the shipping product. Even if the test metric tools worked properly, they could not take that responsibility away from

you. Your aim, then, should be to test while you think the tests are being helpful—and conversely, to stop testing when you are not getting any benefit from the tests. When asked the question, "Which parts of my software should I test?" software engineer and unit testing expert Kent Beck replied, "Only the bits that you want to work."

What Does This Mean for iOS Developers?

The main advantage that unit testing brings to developers of iOS apps is that a lot of benefit can be reaped for little cost. Because many of the hundreds of thousands of apps in the App Store are produced by micro-ISVs, anything that can improve the quality of an app without requiring much investment is a good thing. The tools needed to add unit tests to an iOS development project are free. In fact, as described in Chapter 4, the core functionality is available in the iOS SDK package. You can write and run the tests yourself, meaning that you do not need to hire a QA specialist to start getting useful results from unit testing.

Running tests takes very little time, so the only significant cost in adopting unit testing is the time it takes you to design and write the test cases. In return for this cost, you get an increased understanding of what your code should do *while you are writing the code*. This understanding helps you to avoid writing bugs in the first place, reducing the uncertainty in your project's completion time because there should be fewer show-stoppers found by your beta testers.

Remember that as an iOS app developer, you are not in control of your application's release to your customers: Apple is. If a serious bug makes it all the way into a release of your app, after you have fixed the bug you have to wait for Apple to approve the update (assuming they do) before it makes its way into the App Store and your customers' phones and iPads. This alone should be worth the cost of adopting a new testing procedure. Releasing buggy software is bad enough; being in a position where you can't rapidly get a fix out is disastrous.

You will find that as you get more comfortable with test-driven development—writing the tests and the code together—you get faster at writing code because thinking about the code's design and the conditions it will need to cope with become second nature. You will soon find that writing test-driven code, including its tests, takes the same time that writing the code alone used to take, but with the advantage that you are more confident about its behavior. The next chapter will introduce you to the concepts behind test-driven development: concepts that will be used throughout the rest of the book.

This page intentionally left blank

Index

A

B

C

D

E

F

G

H-I

J-K-L

M

N

O

P

Q

R

S

T

U

V

W

X-Y-Z