Extracted from:

iOS 8 SDK Development

Creating iPhone and iPad Apps with Swift

This PDF file contains pages extracted from *iOS 8 SDK Development*, published by the Pragmatic Bookshelf. For more information or to purchase a paperback or PDF copy, please visit http://www.pragprog.com.

Note: This extract contains some colored text (particularly in code listing). This is available only in online versions of the books. The printed versions are black and white. Pagination might vary between the online and printed versions; the content is otherwise identical.

Copyright © 2015 The Pragmatic Programmers, LLC.

All rights reserved.

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

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

iOS 8 SDK Development

Creating iPhone and iPad Apps with Swift

Chris Adamson
Janie Clayton

edited by Rebecca Gulick

iOS 8 SDK Development

Creating iPhone and iPad Apps with Swift

Chris Adamson
Janie Clayton

The Pragmatic Bookshelf
Dallas, Texas • Raleigh, North Carolina

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and The Pragmatic Programmers, LLC was aware of a trademark claim, the designations have been printed in initial capital letters or in all capitals. The Pragmatic Starter Kit, The Pragmatic Programmer, Pragmatic Programming, Pragmatic Bookshelf, PragProg and the linking *g* device are trademarks of The Pragmatic Programmers, LLC.

Every precaution was taken in the preparation of this book. However, the publisher assumes no responsibility for errors or omissions, or for damages that may result from the use of information (including program listings) contained herein.

Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at *https://pragprog.com*.

The team that produced this book includes:

Rebecca Gulick (editor)
Potomac Indexing, LLC (indexer)
Liz Welch (copyeditor)
Dave Thomas (typesetter)
Janet Furlow (producer)
Ellie Callahan (support)

For international rights, please contact *rights@pragprog.com*.

Copyright © 2015 The Pragmatic Programmers, LLC.
All rights reserved.

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

Printed in the United States of America.
ISBN-13: 978-1-941222-64-5
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—March 2015

CHAPTER 4

Testing Apps

We have come a long way in a short time. We've got an app that can send tweets and show our Twitter web page. We now have a stable app that isn't going to crash on us, right?

Well, how do we know that? We have run the app a few times, but have we really pushed the limits of the app? Have we really tried everything that anyone could possibly do to our app? How do we prove that our app is not going to crash before we ship it off to Apple?

And as we start adding features, what proves that those changes work, or that they're not going to have weird side effects that break the stuff that had been working?

The way we deal with this is to use *unit tests*.

Unit Tests

Unit tests are exactly what they sound like. Unit tests are small, self-contained segments of code that test very small, targeted units of functionality. Rather than check to see if the whole application works, we can break the functionality into pieces to pinpoint exactly where errors and bugs are occurring.

Unit tests are designed to either pass or fail. Is this feature working the way you want it to, yes or no?

The Parable of the Dinosaur

Here is an example of unit testing gone bad.

In *Jurassic Park* (the book, not the movie), Dr. Grant asks the scientists how they can be sure that the dinosaurs are not breeding.

The scientists assure Dr. Grant that every precaution has been taken. They engineered the dinosaurs to all be female. They had the island blanketed with motion detectors to count each and every dinosaur every five minutes. They created a computer algorithm to check the number

• Click HERE to purchase this book now. discuss

and types of dinosaurs found by the motion sensors and the number only changed when a dinosaur died. There had been no escapes. They knew everything happening on the island and they were completely in control.

Dr. Grant asks them to change the parameters of the computer program to look for more dinosaurs than they were expecting to have. The scientists humor Dr. Grant and change the algorithm to search for more dinosaurs. Lo and behold! There are more dinosaurs. After running the program several more times with increasing numbers they eventually discovered there are over 50 extra dinosaurs on the island. Oops!

The program had been set up with the expectation that the number of dinosaurs could only go down, never up. Once the program reached the number of dinosaurs it was expecting to find, it stopped counting and the scientists never knew there was an issue. The program anticipated the outcome of dinosaurs dying or escaping the island, but never the possibility that life could find a way.

Reasons We Unit Test

Bugs, like life, do find a way. The first thing to remember in computer programming is that the computer is stupid. The computer only does what you tell it to do. It can't infer what you meant. It is important to verify that you are giving the right directions to the computer and the best way to do that is to test your apps.

One major reason to unit test an application is to eliminate crashes. The single biggest reason that most app submissions are rejected by Apple is because they crash. Even if Apple doesn't catch your crash, users have a talent for finding the one combination of things that will cause your app to crash. These are the users who tend to leave one-star reviews on the store, which is something we want to avoid if at all possible.

Unit tests also expose logic errors in our code. In the Jurassic Park example, the code being run had a logic error that prevented the scientists from discovering the problem until it was too late. We don't want that to happen to you.

Writing tests also helps you write your code. Have you ever started writing a piece of code only to figure out that one feature you spent days working on wasn't really going to work out in your project? By thinking critically about what specifically you want your application to do, you can avoid writing overly complicated and unnecessary code. They can inform the design of our code: what part of the code has what responsibilities, and how we recover if something unexpected happens.

• Click HERE to purchase this book now. discuss

Designing Good Unit Tests

As we will soon discover, writing a unit test is not difficult. Writing a good unit test is another story altogether.

There are generally three types of unit tests:

- *Debugging:* These tests are built around bugs to ensure that when you change the code these bugs do not reappear. Sometimes when we are coding we make changes to the code that affect bugs that we have already resolved. Since we do not want to see that bug again, we need to write tests to make sure that the bug has not reappeared when we change anything.
- *Assert Success:* We are testing to make sure you are getting a result you want.
- *Assert Failed:* We are testing to make sure you are not getting a result you don't want.

We might wonder why you would need a test to assert failure. Isn't the point of testing to make sure that features we created work properly?

Think back to the Jurassic Park example. The scientists created tests to make sure they were finding all of the dinosaurs they were looking for. They asserted success once the number of dinosaurs they were looking for was reached.

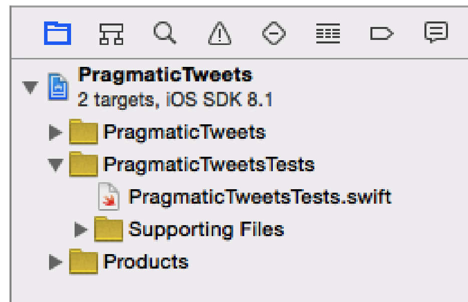
Sometimes it is as important to write a test that we expect to fail to make sure that we are not getting a result we don't want. Had the scientists also included a failure assertion test, they would have discovered that they were getting results that made no sense: there are more dinosaurs in the park than there are supposed to be.

Creating Tests in Xcode

Testing functionality was introduced in Xcode 5. Apple based many of its built-in functions on accepted and open source frameworks and has been working very hard to make testing a vital and useful tool in your developer utility belt.

We are going to go over several aspects of testing in Xcode in this chapter. Since you have spent a great deal of time creating and developing your PragmaticTweets app, let's run it through some tests to see how it works.

Let’s direct our attention to the File Navigator, shown in the figure. There is a group titled PragmaticTweetsTests. Xcode has conveniently created this group and sample template class, PragmaticTweetsTests.swift, for our first two tests.



Before we move on to actually looking at the included test file, let’s also look at the Test Navigator (⌘5). Rather than showing test files, this shows the tests themselves, and whether they passed or failed the last time they ran. This is another location in Xcode that makes it easy for you to get an overview of what tests you have and whether or not they are passing.

Click on the PragmaticTweetsTests.swift file in either the Project or the Test Navigator. There are four methods within this class: setUp(), tearDown(), testExample(), and testPerformanceExample(). Every test class that we create will have a setUp() and a tearDown() method. setUp() is used to instantiate any boilerplate code you need to set up your tests, and tearDown() is used to clear away any of the setup you needed to do for your tests. Whenever we find ourselves repeating code in multiple tests, it’s a candidate for moving into setUp() and tearDown(). This is the principle of DRY: Don’t Repeat Yourself.

Every test method we create will start with the word “test,” just as the testExample() and testPerformanceExample() methods demonstrate. Additionally, every method in a test class will have no return type. We are doing this to make sure that our tests are found by the test compiler and that they get run. A test passes if it returns normally, and fails if it fails an assertion method before it returns.

For fun, let’s just run the test included in the template. There are several ways to run your unit tests:

- Keyboard command: ⌘U
- Main Menu: Product→Test
- Clicking on the diamond icon next to either the test class or the specific test in Xcode

The first two ways of running tests will run all of your tests, whereas the third way will allow us to run selected tests. This is useful if you have one test that is failing and we want to focus on that one without having to run all the others.

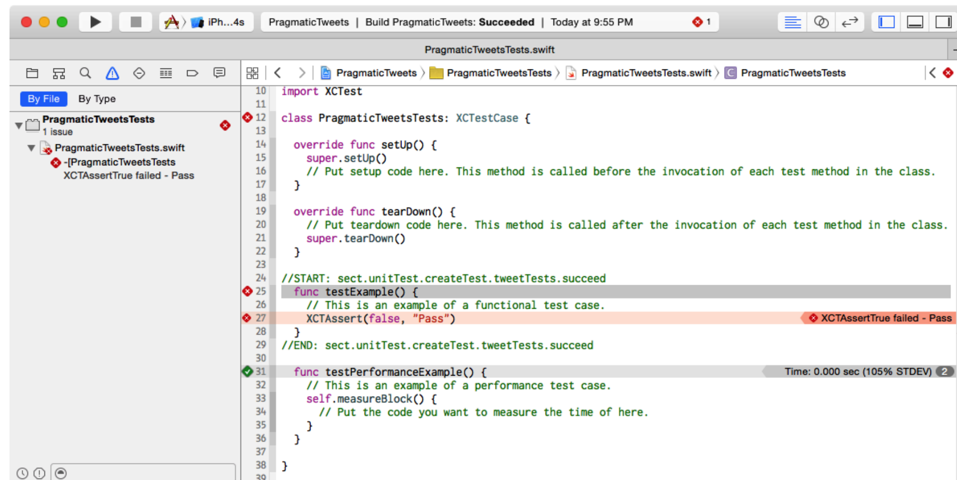
Run the test in the manner of your choice.

Let's take a closer look at testExample().

Testing/PragmaticTweets-4-1/PragmaticTweetsTests/PragmaticTweetsTests.swift

```
func testExample() {
  // This is an example of a functional test case.
  XCTAssert(true, "Pass")
}
```

Just for fun, go in and change the first parameter in the method to false from true to see what happens.



Oh no! The test stopped working! What happened?

Well, we just changed the conditions of the test. XCTAssert must pass a true condition through the test to pass. Since we programmed the condition to false, the test fails. Option-clicking on XCTAssert doesn't give us nice documentation like most Swift methods, but we'll cover the most useful XCTest assertions later in *OCUnit and XCTest,* on page 10.

At first blush this might seem like a useless exercise. Why would we want to write a test that always fails when you run it?

We run a test that is designed to fail so that we verify that the testing framework itself is working properly. If we simply create nothing but tests that are supposed to pass, we can't know for certain that the tests are passing because

the code is correct. There could be an error and the tests would pass regardless. By prompting a failure, we now verify that when we write a test that passes that our code is, in fact, working correctly. As one wise person put it, "How do you know your smoke detector works if it never goes off?"

OCUnit and XCTest

At this point you may be wondering where we got XCTAssert() and the other testing methods from.

Prior to iOS 6, unit testing was done using an open source framework called OCUnit. As of iOS 8, OCUnit is deprecated, and as of Xcode 5, the preferred testing framework is XCTest. XCTest is built on top of OCUnit, so anyone already familiar with OCUnit should find translating from OCUnit to XCTest simple and straightforward.

As we saw earlier, the method in XCTest to assert a true condition is XCTAssert(). In OCUnit, the assertion call for a failing method was STAssert(). It is a convention that any method you have in OCUnit can be converted to XCTest() by changing the ST at the beginning to XCT. So, what was STFail() is now XCTFail().

There are about twenty different assertion methods in XCTest, but the ones we will be using most often are

- XCTAssert()
- XCTAssertFalse()
- XCTAssertEqual()
- XCTAssertNotNil()
- XCTAssertThrowsSpecificNamed()

There is a complete list of every assertion in the "Testing with Xcode" programming guide in your Xcode documentation, if you want to see how deep the rabbit hole goes.

Test-Driven Development

Now that we have a good handle on how to create a unit test, we are going to delve into the realm of test-driven development (TDD). TDD, in a nutshell, is figuring out the least number of objects you need to create in order to get your application to work the way you want it to. TDD utilizes the idea that you will write your tests first rather than after you have already completed your application.

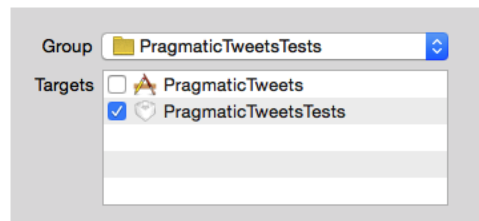
If we write tests for the app now, we'll just be checking functionality we already know works. In TDD, we write the test first, fail for lack of any working functionality, then press ahead and actually create the functionality.

Why do we want to do all this extra work before we write a line of code? Let's jump in the Way Back machine and visit your elementary school English class. Remember back when you were learning how to write stories your teacher told you to write an outline. We write outlines for our stories so that we have an idea about how our story is going to go. We want to figure out the beginning, middle, and end so that we can write a tight and cohesive story that follows a path and has an ending that makes sense. If you go into a story not sure about what is going happen, you'll wind up writing lots and lots of plot where nothing happens.

Our time is valuable. It is in our best interest to figure out exactly which features are important and which ones are not before you spend a week trying to figure out and debug a feature that we figure out later doesn't fit in with what we want our app to do.

So let's add a new feature, TDD style. Let's say we want to have the web view load itself when the app starts up, without having to tap Show My Tweets. We'll start by writing a test to make sure the web view got populated, initially failing because it's not being populated, then go back and add the feature. When the test passes, our feature is good to go.

We'll start by creating a new test class. Before you create this class, click on the PragmaticTweetsTests group. Use the menu item File→New→File to bring up a template of file types. Choose iOS Source from the left pane, and on the right, select the Test Case Class template. Name our new class WebViewTests. Make sure this class is a subclass of XCTestCase and that it is attached to the PragmaticTweetsTests target before creating the class, as shown in the following figure:



Xcode Targets

In Xcode, it is possible to create multiple applications based on the same codebase. If we wanted to make, for example, a game where we had "full" and "lite" versions where the only difference is how many levels are included, we could create two targets that mostly differed by which level files were or weren't included.

Since the primary application does not know what to do with a testing class, we don't want to include it in the codebase for that application; it would just take up space on the end user's device. Putting the test classes in their own target help us segregate them out.

Targets can also be used for other sophisticated build tasks, like running arbitrary shell scripts prior to or after building our code. They can also be set as dependencies of one another. For example, the tests target is dependent on the main app target, so any time we run tests, any changes to the app's code will be built first.