Extracted from:

iOS Unit Testing by Example

XCTest Tips and Techniques Using Swift

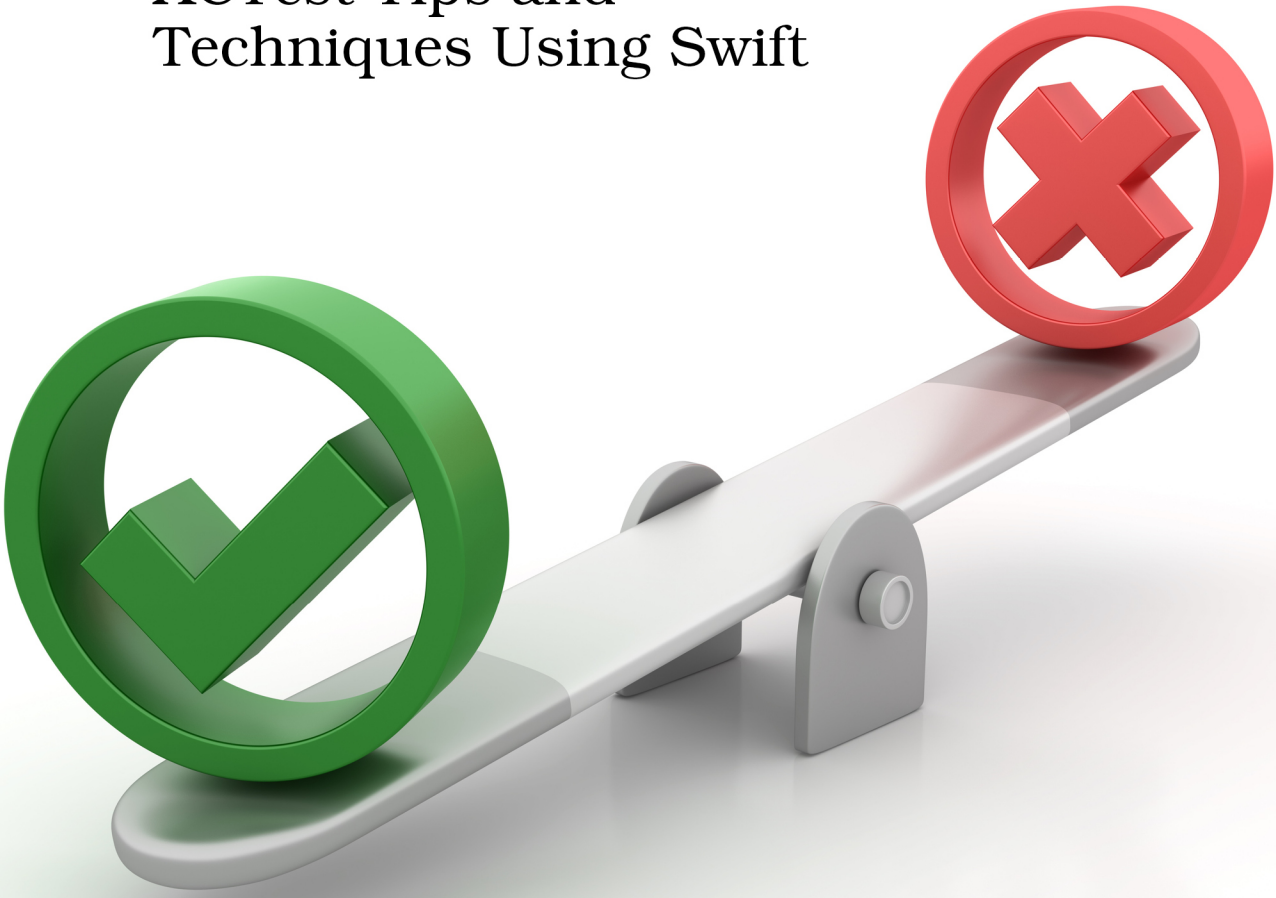
The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

iOS Unit Testing by Example

XCTest Tips and Techniques Using Swift



Jon Reid

edited by Michael Swaine

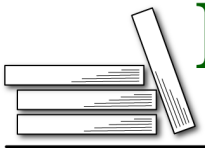
iOS Unit Testing by Example

XCTest Tips and Techniques Using Swift

Jon Reid

The Pragmatic Bookshelf

Raleigh, North Carolina



Our Pragmatic books, screencasts, and audio books can help you and your team create better software and have more fun. Visit us at *https://pragprog.com*.

The team that produced this book includes:

Publisher: Andy Hunt
VP of Operations: Janet Furlow
Executive Editor: Dave Rankin
Development Editor: Michael Swaine
Copy Editor: Adaobi Obi Tulton
Indexing: Potomac Indexing, LLC
Layout: Gilson Graphics

For sales, volume licensing, and support, please contact *support@pragprog.com*.

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

ISBN-13: 978-1-68050-681-5
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—June 2020

For Kay, who believed in me

CHAPTER 1

Assert Yourself

Every company wants to reduce their costs. In software, making changes is inexpensive: we wiggle our fingers on keyboards. So where do the costs lie? Aside from development time, they lie in errors, and how much time it takes to detect these errors. (They also lie in building the wrong thing, which is beyond the scope of this book.)

To detect problems, mobile developers use all kinds of feedback loops. For example, we keep an eye on crash reports and customer complaints. But that's the longest loop. After making an incorrect change, it takes a long time to get that feedback.

To try to prevent errors from making it all the way to customers, companies use manual testing. The best quality experts apply talent and creativity to do exploratory testing. Let's not waste their time asking them to follow steps in mind-numbing repetition. Besides, the time between making an error and getting feedback from testers is still long.

What if we could do a large amount of testing using computers? In fact, what if the developer's own computer could provide feedback? And what if this feedback were so quick, you could get it on every change you made? This kind of rapid feedback is a game changer. It not only catches problems quickly, it can change the way you code.

This is what unit tests are for. Maybe you haven't done any unit testing in your iOS apps yet. Or maybe you've been able to test some logic, but your tests don't cover the iOS-specific parts. (And those are important parts.) Wherever you are in your unit testing journey, the goal for this book is the same: to reduce your costs.

What Are Unit Tests Anyway?

There's some confusion about what makes a test a *unit test*. Many people try to focus on the "unit" part of the name, thinking it describes testing a unit of production code. I'll continue to use the term because it's widespread, but let's forget about asking "What's a unit?" Instead, here's my definition:

Unit tests are a subset of automated tests where the feedback is quick, consistent, and unambiguous.

Quick: A single unit test should complete in milliseconds. We want thousands of such tests.

Consistent: Given the same code, a unit test should report the same results. The order of test execution shouldn't matter. Global state shouldn't matter.

Unambiguous: A failing unit test should clearly report the problem it detected.

In our first chapter, we'll explore the fundamental tool of unit testing: assertions. You'll learn the most common assertions in the Swift XCTest framework in a hands-on way.

If you're a seasoned unit test writer, you may want to skip ahead to the Key Takeaways, on page ?. But even if you've written some tests, it can be good to go over the fundamentals. What are assertions for? What do they report? Do you know how to choose the right assertion for the right job? This chapter will help you get familiar with these tools, which we're going to be using all the time.

Create a Place to Play with Tests

Assertions give unit tests a way to state their expectations. The tests fail if these expectations aren't met. Let's make a place outside of your actual projects where we can experiment with how they work. Throughout this book, you'll learn new concepts by playing in these safe spaces. Then in the exercises at the end of each chapter, you'll begin applying these concepts to your own code.

When it comes to learning, reading doesn't come close to *doing*. If you take the code from the examples and type them into your computer, your learning will go deeper. So I encourage you to open your IDE of choice and give it a go. (The examples will use Xcode.)

Let's start by making a place where we can play with tests. Xcode playgrounds are tricky to use with XCTest, so we won't do that. Instead, we'll make a new project. In the Xcode menu, select *File ▶ New ▶ Project…* or press Shift-⌘-N.

It doesn't matter what type of project we make as long as it comes with unit test support. But since we're going to focus on testing iOS apps, we may as well get used to what that feels like. First, create an iOS Single View App.

Next, choose any options you like for your new project. In the examples that follow, we'll use the project name AssertYourself. But make sure to do the following:

- Choose "Swift" as the language.
- Choose "Storyboard" as the user interface. (Don't select "SwiftUI.")
- Select the check box for "Include Unit Tests."

You now have a project set up to run unit tests on an iOS app, which we'll use for our learning experiments.

Select the initial test file that the new project created. Its name will be the project name followed by Tests. So for this project, find AssertYourselfTests.swift.

Delete every method in the AssertYourSelfTests class, leaving only an empty shell:

```
class AssertYourselfTests: XCTestCase {
}
```

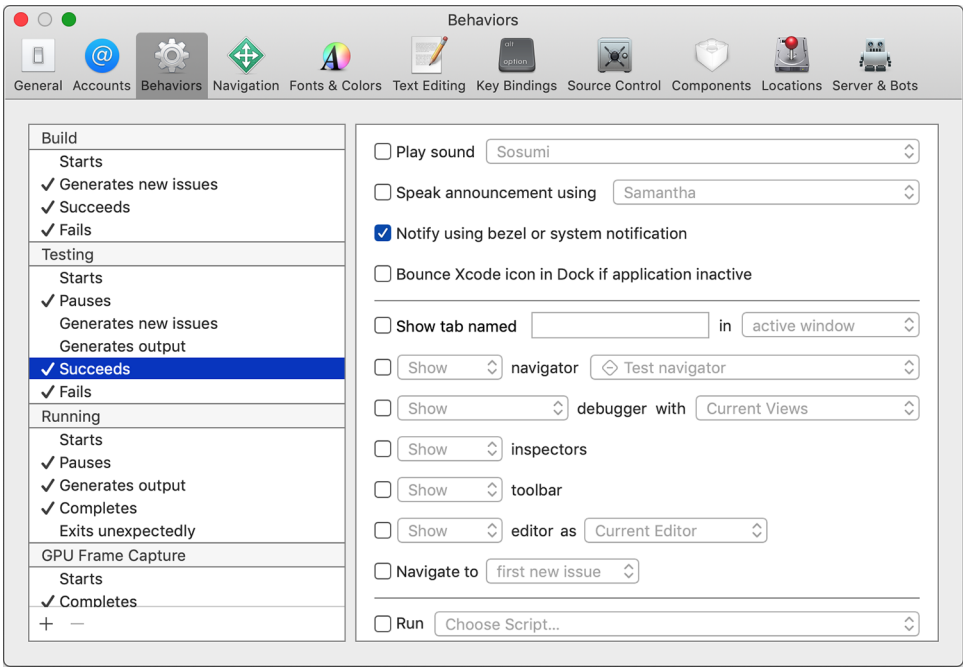
Make sure your destination is set to an iOS simulator. Any simulator will do.

Now in the Xcode menu, select *Product* ▶ *Test* or press ⌘-U. You might want to learn this keyboard shortcut—you'll be doing this often. Think U for "unit test" to remember it.

This will perform several steps and then run the tests. You won't see any test failures because there are no tests. You may see this image show briefly on your screen:

If you didn't see that image, go to Xcode Preferences and select the Behaviors tab. There you can customize what happens when testing succeeds. To display the image, select the check box "Notify using bezel or system notification," as shown in the image on page 10.

Now we're ready to play. In the following sections, we'll experiment with assertions to learn more about them.



Write Your First Assertion

Now that we have a home for tests, let's go over how to use the testing mechanism. How does a test communicate success or failure? What does Xcode show you when a test fails? What does it show when a test succeeds?

The way a test reports a failure to XCTest is through assertions. Let's start with the simplest assertion. Add the following method to the AssertYourselfTests class:

AssertYourself/AssertYourselfTests/AssertYourselfTests.swift

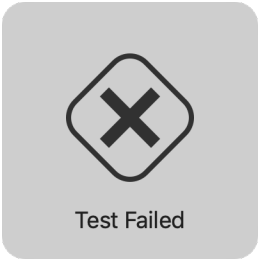
```
func test_fail() {
    XCTFail()
}
```

First, what makes this function a test?

- It lives within a subclass of XCTestCase.
- It isn't declared private.
- Its name starts with test.
- It takes no parameters.
- It has no return value.

Why the underscore in the test name? This goes against Swift's normal "camel case" naming conventions. But good test names often contain three parts. I like to use underscores to separate these parts and camel case within each part. I'll explain this further when we have a test name describing its inputs and expected output. For now, know that the underscores separate the test name into parts, which we'll look at in Add Tests for Existing Code, on page ?.

This test does nothing but fail. Run it by pressing ⌘-U and observe what happens. First, you may see this image show briefly on your screen:



(If you didn't see that image, go back to the Behaviors tab in Xcode preferences. Only this time, customize what happens when testing fails.)

Looking at the earlier source file within Xcode, you'll see the *Test Status Icon* in the left-hand gutter, like the image to the right.

X marks the spot in two places: the method and the class containing the method. The method is a *test*, also known as a *test case*. The class represents a *test suite*, which is a collection of tests. The X icon shows a failure at both the test level and the suite level. You'll also see that Xcode highlighted the XCTFail() line and added an annotation to its right.

```
XCTFail()                    failed
```

So Xcode has marked the following:

- The class containing a failing test
- The method defining a failing test
- The line with the failed assertion

Now add // before XCTFail() to comment out the assertion. Press ⌘-U to run the tests. You'll see the following:

- The annotation disappears from the assertion line
- The test status icons change from red Xs to green check marks, like the image to the right.

This may look trivial, but it's significant. It means we have a way to fail a test, with Xcode showing us where the test reported the failure. You can also see that when a test finishes without triggering any assertions, the test passes.

As you progress in your testing ability, you'll even be able to write assertions defining what you *want* the code to do. Then you can change the production code until it passes the tests. We'll return to this topic in Chapter 20, Test-Driven Development Beckons to You, on page ? at the very end of the book.

Add a Descriptive Message

Seeing the location of a test failure is a good start. But when a test fails, we have to diagnose what went wrong. We can save time for ourselves in the future by having the assertion explain anything we know at the point of failure.

`XCTFail()` can take a `String` parameter as an *assertion message*. Let's see how it works. Add the following method to the class:

AssertYourself/AssertYourselfTests/AssertYourselfTests.swift

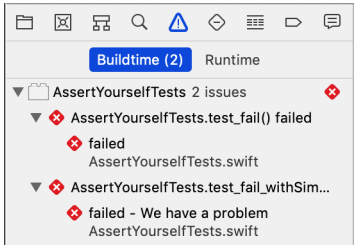
```
func test_fail_withSimpleMessage() {
    XCTFail("We have a problem")
}
```

Run the tests. Note how Xcode puts the message in the annotation:

```
XCTFail("We have a problem")    failed - We have a problem
```

Since the annotation is on the same line as the failure, you may ask, "Couldn't we have put a message to ourselves in a code comment?" But this isn't the only place the message appears.

In the Xcode menu, select *View ▶ Navigators ▶ Show Issue Navigator* (or press ⌘-5). The Navigator column on the left will show any issues, including test failures. You may need to click the Buildtime selector, shown here:



As you can see, the descriptive failure message appears in the Issue Navigator. It also appears in the test logs, which other tools may process—especially on *continuous integration* servers.

Thanks to Swift's string interpolation, XCTFail() can do more than spit out a string literal. Add this to the suite:

AssertYourself/AssertYourselfTests/AssertYourselfTests.swift

```
func test_fail_withInterpolatedMessage() {
    let theAnswer = 42
    XCTFail("The Answer to the Great Question is \(theAnswer)")
}
```

(Strings are italicized in code samples. That's a backslash \ for string interpolation, not a pipe |.)

Run the tests, and you'll see the following:

```
failed - The Answer to the Great Question is 42
```