HACKING WITH SWIFT

# TESTING SWIFT

## COMPLETE TUTORIAL COURSE

Learn how writing better tests can help you to build better apps

Paul Hudson

# Testing Swift

*Paul Hudson*

Version: 20190526

# Contents

# Preface

# Welcome

When developers started buildings apps for iPhoneOS – before Apple had even licensed the iOS trademark from Cisco – it's fair to say the common approach to development was more than a little Wild West. Whether it was bizarre apps like the $999 "I Am Rich" app or the seemingly endless range of fart apps, it was clear there was a gold rush and many people – myself included! – made a lot of money by getting apps out the door as fast as possible rather than planning for long-term development.

These days things could not be more different. Apps are very clearly here to stay: any serious company has an app in the App Store, and will continue to have one there for many years yet. As a result, the idea that app code is ephemeral just isn't true for the vast majority of us – we work on apps that are 10,000 lines of code, 100,000 lines of code, or even more, and we've spent years honing architectural practices, design patterns, and code re-use to support business-critical apps that millions of users rely on.

But while our app development skills have advanced dramatically, many developers for Apple platforms continue to struggle with testing. In fact, it's fairly common to find people proud of the fact that they don't write tests, as if skipping tests were a badge of elite coding prowess. Although every community has opponents to testing, it seems particularly common in the world of Apple platforms. This is probably partly because Apple itself has historically not paid a great deal of attention to testing, but I do wonder how much of it is a legacy of the Wild West culture of the early App Store.

The fact that we're now dealing with codebases that are 1000x bigger ought to be a giveaway that we need bigger and better tools at our disposal, and yet if you look at many app projects you'll find they contain the same two tests:

```
func testExample() {
    // This is an example of a functional test case.
    // Use XCTAssert and related functions to verify your tests
produce the correct results.
}
```

```
func testPerformanceExample() {
    // This is an example of a performance test case.
    self.measure {
        // Put the code you want to measure the time of here.
    }
}
```

Yes, those are the default tests Xcode provides with its template. The developer who selected Xcode's "Include Unit Tests" checkbox might have intended to actually write some tests as part of their work, but as Andy Stanley said in *The Principle of the Path* "it's your direction, not your intention, that shapes your destination." It's not enough to *intend* to write some tests, or *hope* to write some tests – *you need to be moving in that direction by sitting down and actually writing some tests.*

By reading this book you have demonstrated the intention to do better at testing, which is a step forward. However, you need more than just good intentions: you need to be heading in the direction of applying every day the techniques covered here in your own projects.

At first this means you will write code slower, partly because you'll be reading the book, and probably even referring back to it on occasion. But over time you'll become more adept at test-first thinking and soon it will become second nature. I realize that initial slowdown will annoy you – most of us would rather being developing awesome new features rather than struggling to change long-standing culture. But there's a quote attributed to Mich Ravera that you would do well to remember: "if it doesn't work, it doesn't matter how fast it doesn't work."

## About this book

This book is designed to give you a broad introduction to the world of testing. I have tried to cover everything you need to go from someone fresh to the culture of testing to someone who is able to write unit tests, UI tests, and performance tests, who can use test-driven development, who can configure continuous integration systems, and more.

If you haven't worked with tests before, you'll find it's a whole world unto itself: there's a lot of new terminology, new approaches to working, and new Xcode features to learn. As a result, chances are you'll need to read this book through at least twice, because there are so many new things inside. That's OK, though – if you read through from start to finish and learn 10 new things you apply immediately that's a huge win. If you then re-read and find another 10 things you can apply immediately because you already did the original 10 and understand things more now, that's even better.

Don't be too hard on yourself: your direction is what matters!

**Note:** This book uses iOS for test examples, but all the techniques presented here work equally well anywhere Swift runs.

## Frequent Flyer Club

You can buy Swift tutorials from anywhere, but I'm pleased, proud, and very grateful that you chose mine. I want to say thank you, and the best way I have of doing that is by giving you bonus content above and beyond what you paid for – you deserve it!

Every book contains a word that unlocks bonus content for Frequent Flyer Club members. The word for this book is **SPECTRUM**. Enter that word, along with words from any other Hacking with Swift books, here: **https://www.hackingwithswift.com/frequent-flyer**

## Dedication

This book is dedicated to Graham Lee of **http://www.sicpers.info**, Joe Masilotti of **http://masilotti.com**, and Jon Reid of **https://qualitycoding.org**.

When I looked over all the tests I had written in the past, all the notes I had taken both for this book and elsewhere, and all the workshops I had run, I realized that so many of the techniques I was applying were learned either directly or indirectly from either Graham, Joe, or Jon. These three have done so much to encourage a healthy culture of testing in the iOS world, and I'm hugely grateful for all their effort – our community is a stronger place thanks to them.

(For the avoidance of doubt: any mistakes or errors in this book are wholly mine!)

## Copyright

# Chapter 1
## The Basics of Testing

# Why test?

Why do cars have brakes? That question has been asked so many times in so many places and by so many people, but the answer always manages to make listeners do a double take: *cars have brakes so they can go faster.*

This of course seem counter-intuitive: surely brakes are there so that drivers can slow down? Well, yes, but if cars didn't have brakes at all they would simply drive slowly in order to maintain safety – it's the fact that brakes have been added that allows us to drive extremely fast, because we know we can safely stop as needed.

This same logic applies to testing: we write tests so that we can work faster. Again, it's counterintuitive, because it sounds like all the extra work writing tests would ultimately slow down development efforts. However, the cost of building good tests is ultimately more than offset by the ongoing benefits those tests give us.

My all-time favorite quote from noted iOS tester Jon Reid is "we want tests to give us confidence so that we can make bold changes." That really sums it up for me: the tests we write shouldn't just suggest that our code does what we think, but should instead provide us with so much certainty that we can make bold changes to our code without fear of surprise breakages.

Although there are many varieties of tests (more on *that* in later chapters) fundamentally they come in two forms: tests that can be run automatically by a computer, and tests that require manual processes. Both are an important part of any app's testing strategy, but it's the automated side that is the main force behind our ability to make bold changes: with computers able to run hundreds of tests every second it becomes possible to verify a program's behavior with every small change we make.

Good tests mean we can *refactor* code (change the underlying implementation details to be more efficient, more maintainable, or more extensible without changing the external behavior) because our tests should continue to pass throughout the process.

Good tests mean bugs we've fixed in the past stay fixed – we don't accidentally introduce

regressions along the way. Even if you already have a large app that doesn't include tests, you can start adding them each time you fix a bug just for this reason.

Finally, good tests effectively become good documentation, or at least part of it. Each test you write is a direct assertion that you expect the code to have some specific output when it's given some specific input – it literally codifies how various parts of the code are expected to do when run.

So, I hope the answer to the question "why test?" is a little clearer: by giving us the ability to make bold changes, by helping us detect and resolve regressions early, and by providing a comprehensive and practical description of the expectations we have of our app's functionality, tests allow us to work faster. Rather than finding and resolving regression problems by hand – the equivalent of driving a brake-less car slowly because we daren't take any risks – we can be confident that our code works as expected even after we've made changes, because our automated tests are backing us up.

Before I move on, that last point carries with it a subtle extra meaning that is easily missed: tests show us that our code works as expected, but that isn't the same thing as helping uncover bugs. After all, automated tests only verify the expectations you provide them with, rather than to discover new bugs you hadn't anticipated. As Graham Lee put it, testing provides "Quality Assurance, not Quality Insertion."

# Your first test

If you've read other books I've written, you'll know how much I value hands-on learning. So, although most of this introductory chapter is really about giving you a fundamental grounding in tests and their usefulness, I also want to throw you in at the deep end and start writing tests immediately.

Launch Xcode, then create a new iOS project using the Single View App template. Give it the name First, make sure Include Unit Tests is checked, then click Next and create it somewhere like your desktop.

We're going to start with something nice and simple. Taylor Swift's song "Shake it off" tells us that "haters gonna hate", but how can be sure that haters are indeed gonna hate?

Xcode's template gives us a single view controller, but we're going to ignore that for the time being – testing views and view controllers is a more advanced topic that is best left for later. Instead, I'd like you to press Cmd+N to create a new Swift file named Hater.swift, then give it this code:

```
struct Hater {
    var hating = false

    mutating func hadABadDay() {
        hating = true
    }

    mutating func hadAGoodDay() {
        hating = false
    }
}
```

That defines a trivial **Hater** struct with a single property and two methods to manipulate that property. I know it's not complicated, but it gives us enough of a starting point to work with.

We want to test that this **Hater** struct behaves as expected, so what kind of tests should we write? Obviously I'm going to tell you because this is potentially the first time you've sat down and written a test, but before I do I'd like you to look at the code and think for yourself.

Hopefully you've come up with three suggestions:

1. Instances of **Hater** should start with their **hating** property set to **false**.
2. After calling **hadABadDay()** the property should be **true**.
3. After calling **hadAGoodDay()** the property should be **false**.

Let's try coding those now. In the project navigator, look for the FirstTests group and open FirstTests.swift inside there. It should have code similar to this:

```
import XCTest
@testable import First

class FirstTests: XCTestCase {
    override func setUp() {
        // Put setup code here. This method is called before the
invocation of each test method in the class.
    }

    override func tearDown() {
        // Put teardown code here. This method is called after
the invocation of each test method in the class.
    }

    func testExample() {
        // This is an example of a functional test case.
        // Use XCTAssert and related functions to verify your
tests produce the correct results.
    }
```

```
    func testPerformanceExample() {
        // This is an example of a performance test case.
        self.measure {
            // Put the code you want to measure the time of here.
        }
    }
}
```

That's all boilerplate code that we'll be investigating more closely soon, but for now let's get right into writing a test.

Underneath the **testPerformanceExample()** method, but still inside the **FirstTests** class, add this method:

```
func testHaterStartsNicely() {
    let hater = Hater()

    XCTAssertFalse(hater.hating)
}
```

Now I'd like you to press a very important keyboard shortcut: Cmd+U. You're probably already familiar with Cmd+B ("build project") and Cmd+R ("run project"), and Cmd+U should join those two in your muscle memory because it asks Xcode to test the project.

Our new method above is one very small test that checks whether the initial state of **Hater** objects matches what we expect, so Xcode will build the project, launch it in the simulator, then run that test. All being well, you'll see a green check mark appear next to the test when it finishes, which is Xcode's way of saying that the test passed. You'll also see green checkmarks next to the placeholder tests Xcode provided us with, along with another one next to the **FirstTests** class as a whole to mean that all test in the class passed.

Of course, we also need to verify that our two methods work as expected, so please add these two further tests:

```
func testHaterHatesAfterBadDay() {
    var hater = Hater()

    hater.hadABadDay()

    XCTAssertTrue(hater.hating)
}

func testHaterHappyAfterGoodDay() {
    var hater = Hater()

    hater.hadAGoodDay()

    XCTAssertFalse(hater.hating)
}
```

Again, press Cmd+U to have Xcode build and run them all, and you should see more green checkmarks. Testing is easy! Well, sort of – there's quite a lot happening even in this small amount of code, so let's start to unpick what we've seen so far…

# The anatomy of a test

Even though we've only written a tiny amount of code, we've already seen several important parts of Swift testing. Let's start with something nice and easy:

```
import XCTest
```

This imports the XCTest framework, which is Apple's framework for doing all parts of testing. In this book we'll be looking at a variety of test approaches, but they are all covered in this single framework.

Next, we have another **import** line:

```
@testable import First
```

That imports our First module – the main app in our project – but does so using the keyword **@testable**. By default, all types in your code – and all their properties – use the **internal** protection level, which means only code inside your main app module can read it. As that would cause all sorts of problems for testing, the **@testable** attribute automatically gives your tests the same access to code as the rest of your app's code.

Note: this doesn't change the way **private** and **fileprivate** behave – both of those will still keep implementation details inaccessible externally. That's OK, though: private code is *supposed* to be private, and so it shouldn't be tested. If you find yourself wanting to test private code it's a sign you need to rethink your architectural choices.

Moving on, next we have our **FirstTests** class, which defines a block of tests that are grouped logically together. Often these classes have a 1:1 mapping to the classes in your app's code, but it's not required. All test classes must inherit from the **XCTestCase** class: when you run a test Xcode will automatically scan your test bundle for all **XCTestCase** subclasses to find out what should be tested.

Next we have two methods: **setUp()** and **tearDown()**, which are called alongside every test method that is run and provide us the opportunity to create (**setUp()**) and destroy (**tearDown()**)

any objects required for all our tests. These two also have class variants – **class func setUp()** and **class func tearDown()** – that are called once before and after all tests, respectively.

So, excluding the example tests Apple's template provided us with, the testing lifecycle looks like this:

```
class func setUp()
setUp()
testHaterHatesAfterBadDay()
tearDown()
setUp()
testHaterHatesAfterGoodDay()
tearDown()
setUp()
testHaterStartsNicely()
tearDown()
class func tearDown()
```

Now, there are two questions you might have about that code. First, how come those tests aren't run in the order we wrote them? And second, why do we have **setUp()** when we can just create properties in the **FirstTests** class?

The answer to the first question is easy enough: by default Xcode runs them in alphabetical order. That might seem like a logical thing to do, but in practice it can cause all sorts of headaches because it's easy to get into the situation where the result of one test accidentally pollutes another test – test B might pass only because test A ran first, and if the running order were flipped they'd fail. If you ever see tests named **test001addUser()** or similar, you know there's a problem.

As for the second question, this is a little more complex. In order to run one single test, Xcode creates a complete instance of its test class. So, for our three tests plus Apple's two example tests, it will create five instances of the **FirstTests**, and once they are all ready it will begin running the tests using the order shown above.

If we create test objects as properties, this means they will all be created immediately, before any test has run. It also means they will carry on existing all the way through all tests, and in fact won't even be destroyed because **XCTestCase** just terminates the test rather than deallocating its test instances. So, if your test objects are performing important setup and tear down work you have no control over how and when that happens, which makes writing good, isolated tests harder.

As a result, if you want an object to be created for use in tests, it's a better idea to declare it as an optional property that gets created during **setUp()** and destroyed using **tearDown()**, thus guaranteeing you the work is under your control.

Moving on, we have our test methods. Apple provided us with **testExample()** and **testPerformanceExample()**, plus we wrote our own on top to test out the **Hater** struct. Each of these test methods have three common features:

1. Their names all start with the word "test".
2. They all accept no parameters.
3. They all return nothing.

Earlier I said that Xcode automatically scans your test bundle for all **XCTestCase** subclasses to find out what should be tested, and this is the next step: once it has found all your **XCTestCase** subclasses it then scans for test methods using the criteria above. Anything that matches all three points is considered a test, and will be run as such.

A useful side effect of the first point is that you can disable any test by writing anything before "test" in its name so that Xcode will ignore it. Whatever you choose to use, make it consistent – I've seen "DISABLED", "INACTIVE", "SKIP", and "DEAD" all used, and even "BROKEN" a few times when someone is having a particularly hard time. So, you might rename a test method to be **DISABLED_testHaterHappyAfterGoodDay** so that it's clear the test is disabled by choice.

How you choose to name your tests is down to you, but please try to make them clear so that when something fails you can look at the test name to see why. There are a few common

styles, with the most basic looking like this: **testEmptyTableRowAndColumnCount()** – that's actually taken from Apple's documentation. This naming style works fine when you don't have many tests, but as your code grows you might find it easier to be clear what you expect to happen – something like **testEmptyTableRowAndColumnCountIsZero()** makes the expectation clear.

Don't get me wrong: the test code itself also repeats the expectations we had of the code, and indeed that's one of the benefits of writing tests. However, do you really want to dig into the body of the test method when you could just get the summary from the method name?

One trick you would do well to apply is to look for a verb outside of "test" – something saying the action that is happening in the test. For example, **testDeathStarFiredLaser()** or **testMeaningOfLifeShouldBe42()** – both make it clear what we expect to happen, and "should" in particular makes a particularly clear assertion.

As your skill with testing increases, you might find it useful to adopt Roy Osherove's naming convention for tests: **[UnitOfWork_StateUnderTest_ExpectedBehavior]**. If you follow that precisely it would create test method names like this: **test_Hater_AfterHavingAGoodDay_ShouldNotBeHating()**.

**Note:** Mixing PascalCase and snake_case might hurt your head at first, but at least it makes clear the UnitOfWork – StateUnderTest – ExpectedBehavior separation at a glance. You might also see camelCase being used, which would give **test_Hater_afterHavingAGoodDay_shouldNotBeHating()**

Some people like to add method names to their test names, to make it clear which method is being tested. If that works for you that's great, but I don't use this approach because it would cause problems during refactoring – Xcode isn't smart enough to rename associated tests just yet.

We're going to be writing a lot of tests in this book, so I'm going to try to strike a balance between descriptive and wordy test names. Although **long_andDescriptive_testNames()** look and work great in Xcode, they cause all sorts of formatting problems in books, so instead I'm

going to try to keep test names as short as possible while still being useful. In your projects you should experiment to find a style that suits you!

We're nearing the end of our breakdown of the test code, so let's move onto the penultimate items: **XCTAssertFalse()** and **XCTAssertTrue()**. These are test assertions: tests that XCTest will run, then use the result to decide whether the test was successful. There are quite a few of these:

- You can check a value was or wasn't nil using **XCTAssertNil()** and **XCTAssertNotNil()** respectively.
- You can check two values are the same or not using **XCTAssertEqual()** and **XCTAssertNotEqual()**, both of which have special variants for allowing some degree of variation between the values being checked.
- There are **XCTAssertGreaterThan()**, **XCTAssertGreaterThanOrEqual()**, **XCTAssertLessThan**, and **XCTAssertLessThanOrEqual()** for checking specific values.
- You can use **XCTAssertThrowsError()** and **XCTAssertNoThrow()** to check that some expression either throws or doesn't throw an error.
- And finally, there's a general **XCTAssert()** that lets you create any condition you want.

Now, you might think that you can effectively ignore most of those and just use **XCTAssertTrue** for everything. That is, these two pieces of code check the same result:

```
XCTAssertTrue(2 == 3)
XCTAssertEqual(2, 3)
```

While that's true, there are two things to keep in mind. First, good tests act as part of your overall documentation, so if you have the opportunity to express your intent more clearly it's a good idea to take it. Second, those two assertions write out different error messages if your tests fail. The first one will print **XCTAssertTrue failed**, whereas the second will print **XCTAssertEqual failed: ("2") is not equal to ("3")** – I think it's pretty clear the second one is much more valuable than the first!

So, you should always aim to use the most precise assertions XCTest offers – future you will

be most grateful!

All forms of XCTest assertion let you add a custom message to use when tests fail. This is highly recommended, because it's the single best place to clarify exactly what should have happened when the test ran. For example:

```
func testHaterStartsNicely() {
    let hater = Hater()
    XCTAssertFalse(hater.hating, "New Haters should not be
hating.")
}
```

As an alternative, sometimes you'll see folks add a message that acts as a unit for the values being checked, resulting in code like this:

```
XCTAssertEqual(correctLengthInMeters, testedLengthInMeters,
"meters")
```

If that test failed because the correct value was 2 whereas the test returned 3, you'd get output like this: **XCTAssertEqual failed: ("2") is not equal to ("3") - meters**. It's a small thing, but the addition of "meters" there at least adds some context to what's going on.

**It is usually a good idea to use this message string in one of those two forms.** I won't always be doing it here because code in books is already hard enough to read without adding message strings, but please do it in your own code.

The final thing I want to look at is how the tests are structured:

```
func testHaterHappyAfterGoodDay() {
    var hater = Hater()

    hater.hadAGoodDay()

    XCTAssertFalse(hater.hating)
```

```
}
```

Notice that I've added two empty lines in there? That's not an accident! One of the smartest test practices you can adopt is the Arrange, Act, Assert paradigm, which is where you break down your tests into three states:

1. You're putting things into place ready for the test.
2. You're executing the code you want to test.
3. You're evaluating the results of that test.

This is commonly phrased as "given, when, then" and it's very common to see those terms added as comments in tests, like this:

```
func testHaterHappyAfterGoodDay() {
    // Given
    var hater = Hater()

    // When
    hater.hadAGoodDay()

    // Then
    XCTAssertFalse(hater.hating)
}
```

Although again this is a style issue that you'll need to decide for yourself, at least keeping the *structure* of arrange, act, assert is a good idea. This means you should avoid asserting something, then doing some more acting, then asserting again – it creates confusion when your tests fail, and massively increases your chance of your tests not being isolated.

**Tip:** Many testers feel you should only have one XCTest assertion in your test methods, and will instead call helper methods that wrap multiple assertions. Personally I feel that can sometimes obfuscate things a little, but it does have a lot of value when you're making the same assertions in various places.

# The testing pyramid

Our code gets tested in lots of different ways, and all combine together to help us ship a solid, stable product. What you've just written are called *unit tests* because they are designed to test one individual *unit* of functionality – they are the most common and I feel most *important* type of test in our arsenal, but as your skill grows you'll learn the value of the other test types as well.

We've just gone into great detail on what all our test code means, so now I want to turn to a little theory and discuss the various ways our code gets tested. These ways naturally form a pyramid: at the bottom of the pyramid are the fastest, most stable, and most common tests, and at the top are the slowest, least stable, and least written tests.

## Unit tests

You've already met unit tests, and you'll be getting a lot more time with them in upcoming chapters. Unit tests are designed to test that tiny, individual parts of your program work as you expect.

Tim Ottinger and Jeff Langr wrote an article that gives some criteria good unit tests, summing up their findings using the acronym FIRST. So, good tests should be:

- Fast: you should be able to run dozens of them every second, if not hundreds. As they say in the article, "the faster your tests run, the more often you'll run them."
- Isolated: they should not depend on another test having run, or any sort of external state.
- Repeatable: they should always give the same result when they are run, regardless of how many times or when they are run. If unit tests intermittently fail – usually called "flaky tests" – developers stop trusting them as being an accurate measure of code health.
- Self-verifying: the test must unambiguously say whether it passed or failed, with no room for interpretation.
- Timely: they should be written before or alongside the production code that you are testing. This leads into test-driven development, which is covered much later in this book.

*You can read their full article at **https://pragprog.com/magazines/2012-01/unit-tests-are-first**.*

None of the words in FIRST are any more or less important than the others – there's no point in a test being fast if it isn't repeatable, or being isolated if it isn't self-verifying, for example. It's the combination of all these things that makes unit test useful: we can have thousands of tiny tests verifying that everything works individually, and when we have that we can start to build bigger components on top.

I love the way Graham Lee phrased this in his book Test-Driven iOS Development: "Unit testing is 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." (Lee, G., *Test-Driven iOS Development*, 2012)

Note that these tests by necessity preclude combining code together. Following the I in FIRST we are testing "little bits" of our program in isolation – trying to test multiple components in a single unit test will cause problems, because you'll end up less sure of what caused failures. So, unit tests often use a system of test doubles to simulate other components in the system so they can't complicate the test; these are covered later on.

**Tip:** I'll be referring back to the various letters in FIRST a lot during this book, so it's worth writing them down and putting them somewhere you can see!

## Integration tests

On the next level of our testing pyramid we have *integration* tests, which are when parts of your program combine together to complete a specific task in the app. So, you might have unit tests that verify A, B, and C work in isolation, then you might write an integration test that verifies A -> B -> C leads to D.

I explained that unit tests should be fast, isolated, repeatable, self-verifying, and timely, but not all those things apply to integration tests:

- They can't always be fast, because the nature of integration tests is that they are doing real work using real components.

- They can and should be isolated: one integration test should not affect another.
- They can and should be repeatable: we want integration tests to yield the same tests whenever they are run.
- They can't always be self-verifying, because sometimes humans need to check the results by hand.
- They are rarely timely, at least by the standards of Ottinger and Langr – it's unlikely you'll write them alongside your production code.

Obviously the more integration tests *can* be fast and self-verifying, the more useful they will be. Still, it's common to see integration tests done manually, often as a build of the app is being readied for release.

## UI testing

At the top of our testing pyramid are user interface tests, where your actual app is manipulated to check that things work as you expect. Although this can and should be done manually to some degree, you'd be surprised how much can also be done automatically – XCTest has a lot of functionality dedicated to performing automatic verification of app state using your UI, and there are extremely common libraries for performing visual verifications of your layouts.

You've seen how unit tests have direct access to your app's code thanks to the **@testable** attribute – we can literally reach in, grab classes and structs, then call their methods and read their properties all we want. However, UI tests can't do that: they literally run your real app in the simulator, then tap and swipe around just like a real user would.

On one hand this means they more accurately reflect real-world usage of your app, because your tests have no way to short- circuit behavior or inject alternate functionality – they must do, and can only do, what real users can.

However, on the other hand user interface testing is inherently more likely to cause problems for three main reasons:

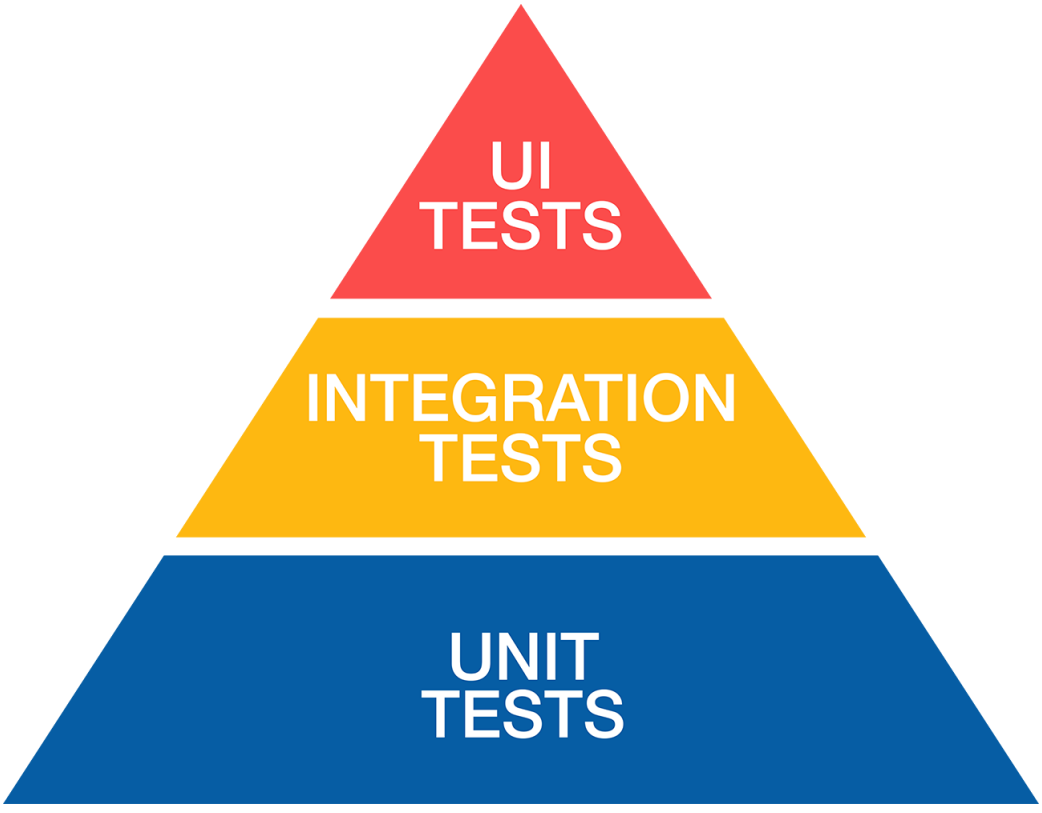
1. The technology itself is less mature, so it might take more work to make your tests pass.

2. It's harder to track what caused the error when a test fails. Should have swiped when it tapped, or vice versa? Whenever you change your user interface a little you might break tests.
3. Visual verification – known as *snapshot* or *screenshot* tests – relies on two images being absolutely identical. If iOS changes font rendering just a tiny amount between releases, your snapshot tests will fail.

All those are why UI testing sits at the top of our pyramid: they are the slowest, least stable, and least written type of test.

## Using the pyramid in practice

As you've seen, these three tests form a neat pyramid, where the most common, fastest, and most reliable tests form a strong bedrock under our code, and the higher layers are less (or at least ought to be!) common because they are slower and a little harder to rely on.



If you're familiar with Maslow's hierarchy of needs, the same proviso applies here: you'll have an uphill battle writing integration tests unless you first have unit tests, and you'll have a similar problem writing UI tests unless you first have integration tests. That doesn't mean it's *impossible*, just that it's not *recommended*.

The pyramid metaphor works well enough at a high level, but in practice things are more complex: there's a huge range of testing that happens, both in terms of test types and test approaches, and different companies use different mixes.

You might say that the complete flow of testing looks something like this:

- The compiler evaluates our code, making sure that we don't mix up data types, forget return values, misuse optionals, forget enum cases, fail to handle thrown errors, and so on.
- Unit tests, integration tests, and UI tests run to make sure our code work as we expect. Some companies prefer different people to write tests and production code and others consider such an approach worthy of damnation.
- A continuous integration service runs those same tests externally to make sure they aren't reliant on our system. Sometimes these only run subsets of tests known as "smoke tests", which are tests that prove the most important functionality works as expected. (Note: "Continuous integration" is a fancy term meaning a computer somewhere monitors every change we make, then automatically runs a suite of tests.)
- QA engineers look at our app, perhaps running through test sheets (lists of actions to perform that exercise the app thoroughly), or perhaps using fuzz testing (press every button, swipe every screen, type nonsense into text fields, often done extremely rapidly) to trigger failures.
- Hallway testing and/or dogfooding happens. Hallway testing is when you ask other folks in the company outside your development group – e.g. "Sam in marketing" – to try the app, and "dogfooding" is a reference to the term "eat your own dog food", which is the idea that makers of a product should be forced to use the product as real end users.
- You might use a service such as TestFlight to deploy the app to a group of users who have signed up to early access. They can provide feedback and report crashes before you ship the thing to the majority of end users.

- Finally, there's production testing, also known as "ship at and hope for the best." As my friend Zack Falgout said, "it all gets tested eventually." Hopefully you've caught the vast majority of bugs before this point!

Sadly many Swift developers rely heavily on the compiler to "prove" their code is correct, then go straight to manual and production testing. It's certainly true that the compiler will do a number of very useful checks for us, and Swift's ability to synthesize conformances for protocols such as **Equatable**, **Hashable**, and **Codable** means that's less code for us to write, and in turn less code for us to *test*. However, in the words of Chris Eidhof: "Types are not a silver bullet. You still need to test your code. But wouldn't you rather test interesting parts, and leave the boring stuff to the compiler?"

# So what's the problem?

So far we've written a handful of simple tests, walked through how they work, and taken a brief look at the bigger picture of testing. You're probably thinking this doesn't look too complicated, and with such clear benefits why would anyone in their right mind not want to add tests to their code?

Recently I was having dinner with a couple of friends, one of whom had worked at Apple for a Very Long Time. We were talking about the importance of testing, and one friend said "didn't you previously work on [redacted]? You must have had so many unit tests on that!" The ex-Apple developer sighed and said, "or zero – guess which one it is."

Spoiler: it was zero. And this is on a component I guarantee you've used many times – a component we all rely on every day.

At the other end of the scale, just a week earlier I was speaking to developers who work on the app for a major department store, who have over 7000 tests. Every developer who joins the team is given a top-spec MacBook Pro to work with, but even then it takes 25 minutes to run the complete test suite. They said recently someone joined the team and, thinking a MacBook Pro was too heavy for them, asked for a MacBook Air – I'll leave it to you to imagine how long their tests took to run!

Going back to Ottinger and Langr's advice on tests, "the faster your tests run, the more often you'll run them." If your full test suite takes 25 minutes to run, you clearly can't run it more than once a day – you're just burning through too much time. As a result, you end up picking and choosing subsets of tests to run, so the onus is on you to make sure you've chosen the right tests.

So, it's clear that companies big and small have problems with testing. Sometimes it's having startlingly few tests, sometimes it's having so many they are effectively out of control, but I think it illustrates that testing is far from a solved problem.

### "We can't afford tests"

One of the most common reasons you'll hear for a poor approach to testing is that they are too expensive. This comes in a variety of forms, and chances are you'll meet all of them:

- "The client doesn't want to pay for tests."
- "We're a startup; we don't have the resources to write tests."
- "This needs to ship quickly; we don't have the time for tests."

Of course, the responses should be pretty clear at this point: if the client doesn't want to pay for tests, are they willing to pay to fix the crashes from the less-stable software you write, or willing to pay for a dozen bug fix releases after their initial launch as you try to clear up the mess? If your company "doesn't have time" for tests, are they willing to put their developers on a three-month crunch – sometimes called a *death march* – as they try to track down and fix all the bugs that accumulated during their rushed development?

Bluntly, we don't ask clients whether we should use Scrum or Kanban in our development, and neither do we ask them whether MVC or MVVM is a better architecture to choose. The same is true for tests: we don't ask clients whether they want to pay for tests, because our jobs – our reputations – rely on us shipping high-quality software, on spec, and on time, and that means writing high-quality tests.

Let me put it this way: which parts of your app do you need to write test for? Only the parts you don't want to crash.

## "We have too many tests"

Some people, particularly those approaching test-driven development for the first time, write some tests, then write some code, then write some more tests and so more code, and really feel like they are making good progress – until they realize they have a hundred tests, lots of duplication, and their ability to refactor has been *lowered*.

Remember that quote from Jon Reid: "we want tests to give us confidence so that we can make bold changes." If your app code and test code is so unwieldy that you can't make bold changes, you've missed the point.

Think about how much time we spend reading about and applying knowledge about architecture, generics, protocols, and so on. The Swift language provides us with a dazzling array of language features and design patterns to help us create smart, safe, reusable code, but for some reason these get completely ignored when writing tests – we just write them the brute force way, with a thousand functions named **testXYZ()**.

At WWDC17, Xcode engineer Greg Tracy said "Treat your test code with the same amount of care as your app code." Simple, but straight to the point: all the coding principles we apply to writing our app code apply just as much to our test code.

If you speak to companies who find their tests have gotten out of control, they'll usually talk proudly about their approach to code review – they might do pair programming, they might do GitHub reviews, they might have in-person code review sessions. But if you ask whether they do that same code review for their tests, you'll find the problem: they usually don't.

The code you write is designed to solve a problem, and the tests you write are there to prove the code does what it says – the two are equally important, and should be treated as such. If companies aren't reviewing their tests, why not? If companies aren't testing their tests – feeding them some bad data to make sure they work as intended – why not?

## We're all on the same team

In the next chapter we're going to dive deep into unit tests, but before I move on I want to add one last thing.

Sometimes development and QA teams eye each other with suspicion – one team inserts bugs and the other breaks software is a common viewpoint. Neither of those things are true – we're all on the same team, and we're all doing our best to ship useful, stable software to the world.

Tests alone aren't enough to demonstrate that the code you've written does what customers actually need. It *does* prove that the functions you're testing behave in the way you intended, but tests can't explain client specifications to you or clear up ambiguities – they can't show you've understood or even considered the fundamental problem you were trying to solve. So we muddle along, doing our best.

As for QA, there's a great quote by James Bach: "We don't break the software, we break illusions about the software." You can write all the tests you want, and you can even point to metrics that show things like 100% code coverage in your unit tests, but if your tests weren't thorough enough and QA hits problems it's not because they don't like you.

To paraphrase Patrick Lencioni in his book *The Advantage*, we're all in the same boat, and we're all supposed to be rowing in the same direction. There's no point behaving like a fisherman who looks at his friend sat at the other end of the boat and says "watch out – your side of the boat is sinking." That means it's not good enough for developers to shirk responsibility for testing, because it's a team effort.

OK, lecture over – let's take a look at some serious unit testing!

# Chapter 2
## Unit Testing

# Organizing unit tests

As you've learned, unit testing forms the bedrock of our testing pyramid: they are the first tests you write, the most common tests you write, and the most useful test to *run*.

So, over the next few chapters we're going to go into much more detail about unit tests: how they work, how to test unusual or complex circumstances, and how to assess the quality and usefulness of your tests.

## Splitting tests by task

Let's write a little code that is actually useful: a Fahrenheit to Celsius converter. Start by creating a new Swift file in our project, naming it Converter.swift – make sure you add it to the "First" group, not to "FirstTests".

We don't need much in there, so just give it this code for now:

```
struct Converter {
    func convertToCelsius(fahrenheit: Double) -> Double {
        return (fahrenheit - 32) * 5 / 9
    }
}
```

In order to test our **Converter** class we need a new **XCTestCase** subclass. So, press Cmd+N, choose Unit Test Case Class, then click Next and name it "ConverterTests". Make sure your group is "FirstTests", then click Create.

You'll see more or less the same code we had in our first example unit test from earlier, but this time it's missing one critical line from the top: **@testable import First**. Without that we can't create and test data types from our main app bundle, so, please add that now.

We need to test that our simple converter works as we anticipated. This is best done by feeding it a few different values to make sure output manages input.

Two values that I remember off by heart are the freezing and boiling points of water: 30 Fahrenheit / 0 Celsius, and 210 Fahrenheit / 100 Celsius, respectively.

So, we could add this to our new test case:

```
func testFahrenheitToCelsius() {
    let sut = Converter()

    let input1 = 30.0
    let output1 = sut.convertToCelsius(fahrenheit: input1)
    XCTAssertEqual(output1, 0)

    let input2 = 210.0
    let output2 = sut.convertToCelsius(fahrenheit: input2)
    XCTAssertEqual(output2, 100)
}
```

**Note:** You'll see **sut** a lot in unit tests. It's short for "system under test" and its used to refer to whatever object you are testing in the code so it can be identified at a glance. To the best of my knowledge, this term was first introduced by Gerard Meszaros.

Press Cmd+U to build and run that test, and you'll see it fails. Don't worry, the failure is intentional – I lied about the two input values, because I wanted to demonstrate an important behavior of XCTest: when one test fails it will carry on running tests. This behavior is usually what you want, because if you had a number of test failures it would be quite tiresome to see them only one at a time.

If you'd prefer Xcode to halt as soon as any test fails, add this line to the **setUp()** method:

```
continueAfterFailure = false
```

You'll usually see that in UI tests, because a failed test usually means the UI is in an unexpected or incorrect state so it isn't a good idea to continue.

To make these tests pass we need to use the correct values for Celsius: 32 and 212. So, change it to this:

```
func testFahrenheitToCelsius() {
    let sut = Converter()

    let input1 = 32.0
    let output1 = sut.convertToCelsius(fahrenheit: input1)
    XCTAssertEqual(output1, 0)

    let input2 = 212.0
    let output2 = sut.convertToCelsius(fahrenheit: input2)
    XCTAssertEqual(output2, 100)
}
```

If you press Cmd+U the tests will now pass, but that doesn't mean the test code is good. In fact, this test still has a problem: it doesn't follow the given-when-then pattern. If we were to annotate it with comments it would look like this:

```
func testFahrenheitToCelsius() {
    // given
    let sut = Converter()
    let input1 = 32.0

    // when
    let output1 = sut.convertToCelsius(fahrenheit: input1)

    // then
    XCTAssertEqual(output1, 0)

    // given
    let input2 = 212.0
```

```
    // when
    let output2 = sut.convertToCelsius(fahrenheit: input2)

    // then
    XCTAssertEqual(output2, 100)
}
```

So, we have given-when-then-given-when-then – we repeat the whole process twice. We *could* rearrange the lines to look more like this:

```
func testFahrenheitToCelsius() {
    // given
    let sut = Converter()
    let input1 = 32.0
    let input2 = 212.0

    // when
    let output1 = sut.convertToCelsius(fahrenheit: input1)
    let output2 = sut.convertToCelsius(fahrenheit: input2)

    // then
    XCTAssertEqual(output1, 0)
    XCTAssertEqual(output2, 100)
}
```

However, that's just missing the point a little. Yes, it is always preferable to be able to see the structure of your tests clearly, but what would happen if calling **convertToCelsius()** changed our system state? It might adjust some state in our app or it might mutate the **Converter** we had created, for example, but either way it would stop our tests from being Isolated and Repeatable – the "IR" in FIRST.

**Warning:** I'm repeating this because it's important. If Xcode is configured to continue after a test has failed (which it is, see above), and you try to test more than one thing in a single test

case, you run the risk that one failing test will pollute the results of a later test in the same method. This might make the test fail because things aren't set up the way it expects, or, worse, make a broken test *pass*. If you intend to test more than one thing in a single method, please do so carefully otherwise you might come face to face with a problem sometimes called *assertion roulette*.

Plus, is **testFahrenheitToCelsius()** a particularly good test name? It doesn't make it clear what we expect to happen, so if the test fails in the future we need to read the test code to figure out what was supposed to happen.

We can fix both these problems by splitting this one test out into two, each making their own assertions:

```
func test32FahrenheitIsZeroCelsius() {
    // given
    let sut = Converter()
    let input = 32.0

    // when
    let celsius = sut.convertToCelsius(fahrenheit: input)

    // then
    XCTAssertEqual(celsius, 0)
}

func test212FahrenheitIs100Celsius() {
    // given
    let sut = Converter()
    let input = 212.0

    // when
    let celsius = sut.convertToCelsius(fahrenheit: input)
```

```
    // then
    XCTAssertEqual(celsius, 100)
}
```

Those tests can now be run independently without fear of the results of one influencing the other. Plus they now have nice and clear test names: if **test212FahrenheitIs100Celsius()** because the result is 100 Celsius, we can see the correct value at a glance.

## Common set up

Now that we have two independent tests that both use an instance of **Converter**, we can start to use the **setUp()** method. Remember, this gets called before every test is run, with **tearDown()** being called *after* each test is run. So, rather than creating our **Converter** instance inside each test method we can create a property that gets instantiated and destroyed automatically.

First, add this property to the **ConverterTests** class:

```
var sut: Converter!
```

Now modify your **setUp()** and **tearDown()** methods to this:

```
override func setUp() {
    sut = Converter()
}

override func tearDown() {
    sut = nil
}
```

Even though it's now a property on the class, every test method will get its own converter created and destroyed as the tests run – just make sure you remove the **let sut = Converter()** lines from each test, because it's no longer needed.

This system works because Xcode creates one unique copy of **ConverterTests** for every test, so each test has its own unique **sut** property.

**Note:** As mentioned previously, you can use **class func setUp()** and **class func tearDown()** to do one-time setup and teardown for all tests in a class. Normally I prefer to avoid static properties because they risk the isolation of our tests, but in this instance **Converter** has no state of its own – its functionality can't be affected with settings, because it's just data in and data out.

## Time for some refactoring

I don't recall when I first learned the difference between refactoring and rewriting, but it's a distinction that served me for so long and so well in my career that if you remembered nothing else from this book I hope it's this:

**If you spend a few hours changing your code to make it better, smarter, more maintainable, or whatever, that's great. But that doesn't mean it's refactoring. Refactoring is when you've made modifications to your code without changing its external behavior, and the only way to be sure you haven't changed its external behavior is using tests.**

That's the difference between refactoring and rewriting: both involve making changes to your code, but you can only really call it refactoring when it's backed by tests.

We've written tests for our temperature converter, so we're now in a position to do some refactoring: we're going to ditch our custom Fahrenheit to Celsius converter and replace it with the one from Foundation.

Replace the **convertToCelsius()** method with this:

```
func convertToCelsius(fahrenheit: Double) -> Double {
    let fahrenheit = Measurement(value: fahrenheit, unit:
UnitTemperature.fahrenheit)
    let celsius = fahrenheit.converted(to: .celsius)
```

```
    return celsius.value
}
```

That code is functionally identical to what we had, except it now uses Foundation's **Measurement** type to create a Fahrenheit measurement, then converts it to Celsius. We've completely changed our code, but this is safe because we have tests, right?

Well, let's find out: press Cmd+U to run the tests against this new code. Chances are you'll find it's failing, and you'll see messages like this:

- XCTAssertEqual failed: ("2.2168933355715126e-12") is not equal to ("0.0")
- XCTAssertEqual failed: ("100.00000000000301") is not equal to ("100.0")

What we're seeing here is the natural variance in accuracy of floating point numbers: 100.00000000000301 is, for all intents and purposes, 100.0, and the "e-12" in 2.2168933355715126e-12 means you're actually looking at a very tiny number: 0.000000000022168933355715126 – effectively 0.0.

Floating point numbers are tricky beasts, and in this situation strict equality isn't going to cut it. Fortunately, our tests caught the problem – if we didn't have tests those tiny changes could have caused all sorts of problems in our app.

In this instance we're going to use a slightly different form of **XCTAssert()** that takes an optional extra parameter specifying an accuracy – how much plus or minus the value we're willing to accept as variation. In this instance, accepting 1/1000000th of a degree Celsius - one millionth of a degree – allows us to ignore floating-point mathematics issues, while still ensuring accuracy.

So, replace your two calls to **XCTAssertEqual()** with these two:

```
XCTAssertEqual(celsius, 0, accuracy: 0.000001)
XCTAssertEqual(celsius, 100, accuracy: 0.000001)
```

This accuracy parameter is really useful when working with numbers that have recurring

digits. For example, 1/3 as a decimal is 0.333333333 recurring. Swift can't represent recurring digits, so we'll probably end up with something like 0.3333333333334, but we don't want to *rely* on that so it's a good idea to use the accuracy parameter instead.

As a practical example, 100 degrees Fahrenheit is 37.777777 recurring degrees Celsius. We could write a test like this:

```
func test100FahrenheitIs37Celsius() {
    // given
    let input = 100.0

    // when
    let celsius =
ConverterTests.sut.convertToCelsius(fahrenheit: input)

    // then
    XCTAssertEqual(celsius, 37.777777, accuracy: 0.000001)
}
```

# Custom setup and teardown

Trying to stick with the given-when-then format can help focus your mind: it's clear what the setup is, you've cleanly isolated the behaviors in your test, and the expected results of your checks can be seen at a glance. As you've seen, it's generally a good idea to assert only one thing in your tests, but there's another problem you're likely to meet early in your testing career: complicated setup.

For example, you might have a **User** type that needs to be created, then some instances of a **Project** type inside your user, then some **ToDo** instances inside those projects, and you want to test that having one item tells the user "1 item" but having two or more says "(2/3/4/etc) items" – that the pluralization is correct across all projects.

You might write the latter like this:

```
// given
let sut = User(name: "Taylor Swift")

for album in 1...3 {
    let project = Project(name: "Album #\(album)")
    sut.addProject(project)

    for song in 1...10 {
        let item = ToDoItem(name: "Write song #\(song)")
        project.addItem(item)
    }
}

// when
let rowTitle = sut.outstandingTasksString

// then
XCTAssertEqual(rowTitle, "30 items")
```

The “when” and “then” parts of that test are trivial, but the “given” part takes a fair chunk of code – we need to do a lot of work just to set up our app so that it’s in the correct state to be tested.

This gets compounded more if you want to check assumptions before you start testing. Assumptions are a special variety of assertions – they use the same XCTest assert calls you’ve seen already, but now they are used at the start of a test in order to make sure the environment is correct before the test even starts.

For example, you might rewrite the above like this:

```
// given
let sut = User(name: "Taylor Swift")
XCTAssertEqual(sut.projects, 0)

for album in 1...3 {
    let project = Project(name: "Album #\(album)")
    XCTAssertEqual(project.items, 0)
    sut.addProject(project)

    for song in 1...10 {
        let item = ToDoItem(name: "Write song #\(song)")
        project.addItem(item)
    }
}

// when
let rowTitle = sut.outstandingTasksString

// then
XCTAssertEqual(rowTitle, "30 items")
```

There's nothing *wrong* with this approach, but I would say two things.

First, these sorts of states really ought to be covered in individual tests rather than asserted in an unrelated test. You ought to be able to have sufficient confidence in your tests that you don't need to re-assert these fundamentals here.

Second, if you *do* choose to add assertions for your assumptions – and it is quite common, honest! – at least keep it brief. Your tests are likely to fail at some point in the future, as you set about making bold changes to your code. That's OK; that's just a sign your tests are doing their job. But when tests fail, you don't really want to have to read through a lot of code to figure out what's happening.

In his book *xUnit Test Patterns: Refactoring Test Code*, Gerard Meszaros outlines a solution he calls *creation methods*. These are effectively setup helpers that isolate the kind of setup you see above, and they allow us to "encapsulate the mechanics of object creation so that irrelevant details do not distract the reader." (Meszaros, G., *xUnit Test Patterns: Refactoring Test Code*, 2007)

The goal of these setup helpers is to make sure our system under test is created and configured in a simple, standardize way, so that all tests using that configuration don't have to repeat themselves.

For example, we might take the set up code from above and put it into a creation method like this one:

```
func createTestUser(projects: Int, itemsPerProject: Int) ->
User {
    let user = User(name: "Taylor Swift")
    XCTAssertEqual(sut.projects, 0)

    for album in 1...projects {
        let project = Project(name: "Album #\(album)")
        XCTAssertEqual(project.items, 0)
        user.addProject(project)
```

```
        for song in 1...itemsPerProject {
            let item = ToDoItem(name: "Write song #\(song)")
            project.addItem(item)
        }
    }

    return user
}
```

That now allows to dramatically simplify our test set up:

```
// given
let sut = createTestUser(projects: 3, itemsPerProject: 10)

// when
let rowTitle = sut.outstandingTasksString

// then
XCTAssertEqual(rowTitle, "30 items")
```

The important thing to remember is that subclasses of **XCTestCase** are data types in their own right – you can add to them all the properties and methods you like, although I would caution you to avoid creating unnecessary state.

What is *state*? Well, it's funny you should ask that. "State" is what we call stored values in an app. Mostly its benign because we create state inside a method then toss it away once the method finishes, but sometimes we also store state as properties to a type and those values get used again and again.

Managing state in tests is pretty much like managing state in app code: if you can avoid it you should, otherwise manage it extremely carefully. Specifically, it's down to you to make sure that every one of your tests creates its data during **setUp()** and destroys it in **tearDown()** – that

there is no surprise shared state that might weaken the isolation between your tests. As I said, this is the primary reason why I don't use **static func setUp()** or **static func tearDown()** unless I'm working with guaranteed fixed data that is slow to create, such as parsing a load of JSON into a constant array of objects.

Sometimes you'll find that a test has taken some action that has the potential to pollute the rest of the tests in your suite, but because it happens only in one test you can't clean it up using **tearDown()**. For this situation, XCTest gives us an extremely useful method: **addTeardownBlock()**. Call that with whatever custom code you want to be used to clean up the current test, and it will be done regardless of how the test resulted.

It has a number of very useful behaviors:

- You can have one custom teardown block or ten of them, it doesn't matter.
- They are run in the reverse order you add them, so the last teardown block to be added will be the first to be run.
- All your teardown blocks are guaranteed to have been run by the time **tearDown()** is called.
- Teardown blocks are executed regardless of the result of your test.
- Teardown blocks are executed even if you've set **continueAfterFailure** to false.

To demonstrate the complete lifecycle of tests, including multiple teardown blocks, create a new Unit Test Case Class called LifecycleTests, making sure to add it to the FirstTests group.

Give it this code:

```
import XCTest

class LifecycleTests: XCTestCase {
    override class func setUp() {
        print("In class setUp.")
    }
```

```
    override class func tearDown() {
        print("In class tearDown.")
    }

    override func setUp() {
        print("In setUp.")
    }

    override func tearDown() {
        print("In tearDown.")
    }

    func testExample() {
        print("Starting test.")

        addTeardownBlock {
            print("In first tearDown block.")
        }

        print("In middle of test.")

        addTeardownBlock {
            print("In second tearDown block.")
        }

        print("Finishing test.")
    }
}
```

When that runs you'll see the following in Xcode's log:

```
In class setUp.
Test Case '-[FirstTests.LifecycleTests testExample]' started.
```

```
In setUp.
Starting test.
In middle of test.
Finishing test.
In second tearDown block.
In first tearDown block.
In tearDown.
Test Case '-[FirstTests.LifecycleTests testExample]' passed
(0.001 seconds).
In class tearDown.
Test Suite 'LifecycleTests' passed at 2018-11-10 13:13:31.263.
```

That's the complete test lifecycle – note particularly the way **class setUp()** is called before test case even starts, and how the teardown blocks called in reverse.

So, if a test opens a file it needs to close, or if it writes some data it needs to delete, the simplest, guaranteed way to clean up is with **addTeardownBlock()**.

# Control your inputs

When you're writing tests what you want to be able to do is provide specific input and expect to receive back specific output - regardless of what state your app is in, or what tests have run previously. This satisfies the "I" and "R" in FIRST: tests should be isolated and repeatable.

The most important thing to do is make sure the code you're testing relies solely on the input you're providing. This means you should provide all the values the test code needs either as initializers or properties – a technique called dependency injection, which we'll be looking at in the Test Doubles chapter. As soon as the code you're testing accesses something you didn't provide – such as reading from a singleton – it becomes significantly harder to test.

This will all sound very familiar to functional programmers. In my book *Pro Swift* I laid down five rules of functional programming, and rules 3, 4, and 5 are all helpful to apply when writing testable code:

1. Functions are first-class data types. That means they can be created, copied, and passed around like integers and strings.
2. Because functions are first-class data types, they can be used as parameters to other functions.
3. In order to allow our functions to be re-used in various ways, they should always return the same output when given specific input, and not cause any side effects.
4. Because functions always return the same output for some given input, we should prefer to use immutable data types rather than using functions to change mutable variables.
5. Because our functions don't cause side effects and variables are all immutable, we can reduce how much state we track in our program – and often eliminate it altogether.

You'll know when you're creating highly testable code because you find writing unit tests for it is trivial.

**Tip:** Although realistically having to do this is a bit of a code smell (a sign that your tests aren't as simple as they ought to be), you can place breakpoints in tests and step through them just like regular app code. This can help you narrow down problems when your tests aren't

behaving as you expect.

# Making assertions

You've now seen the variety of XCTest assertions you can use to verify that tests work as you intend, and we've also looked at the importance of allowing some degree of variable accuracy when working with floating-point numbers.

Here's another excellent piece of advice from Jon Reid that you would do well to remember: "tests should be sensitive to things that are important, but ignore things that are unimportant." Does it matter if the temperature returned was 100.00000000000301 rather than 100.0? If it matters, make your assertion check for it; if it *doesn't* matter then add a little bit of float in your accuracy to make it clear.

My friend Paul Ardeleanu once gave me a piece of advice that has proved immensely useful in stress-testing my code: avoid hard-coded values. So, when creating users, don't give them the name "Taylor Swift" or "Paul Hudson", but instead give them a random string of letters and numbers – something like a UUID, for example:

```
let user = User(name: UUID().uuidString)
```

Every time that test runs it will create users with seemingly random names, such as "53C7B0D1-17DF-4346-9813-F47721932D30." This can help uncover all sorts of hidden assumptions in your tests, and you could easily extend it to other fields that need to be thoroughly tested. For example:

```
func randomAge() -> Int {
    return Int.random(in: 0...120)
}
```

Allowing a little float comes in other forms. For example, if you were testing a game like Civilization you might want to write a test that if a player builds a city then their city count should be 1. But what if later on you decide all players should start with one city to help them get into the game faster? Checking for 1 would no longer work, but if you add a little float the problem goes away:

```
// given
let initialCities = player.cities.count

// when
player.buildCity(name: "London")

// then
XCTAssertEqual(player.cities.count, initialCities + 1)
```

So, rather than checking for precisely 1 city, we read the start value, perform our work, then check that our player now have one more than they started with.

Or what if you were checking whether declaring war on other players worked properly? You might write code like this:

```
// given
let humanPlayer = Player(nation: "Romans")
let computer1 = Player(nation: "Babylonians")
let computer2 = Player(nation: "Vikings")

// when
humanPlayer.declareWar(on: computer1)
humanPlayer.declareWar(on: computer2)

// then
XCTAssertEqual(humanPlayer.enemies, [computer1, computer2])
```

That test might work great for now, but in the future your algorithm might change subtly. Rather than being at war with **[computer1, computer2]** the human player might now be at war with **[computer2, computer1]**. (It's the same computer enemies, just switched in the array.)

Does that difference matter? *Possibly* – it depends on the code you were building. However, in this case we don't really care in what order the enemies are stored, as long as they are both

there, so our test might break without warning when really its fine.

In this instance you have two options: either use the **Set** type, which implements == so that elements in the set can appear in any order without affecting the result of the check, or add an extension to **Array** like this:

```
extension Array where Element: Comparable {
    func fuzzyMatches(other: Array) -> Bool {
        let sortedSelf = self.sorted()
        let sortedOther = other.sorted()
        return sortedSelf == sortedOther
    }
}
```

That requires the array in question to have elements conforming to **Comparable**, because it allows us to sort the arrays first then do a straight check for equality.

If your type only conforms to **Equatable** rather than **Comparable** (which includes **Equatable**), you need to write a slightly more in-depth method:

```
extension Array where Element: Equatable {
    func fuzzyMatches(other: Array) -> Bool {
        guard self.count == other.count else {
            return false
        }

        var selfCopy = self

        for item in other {
            if let index =
                selfCopy.firstIndex(of: item) {
                selfCopy.remove(at: index)
            } else {
```

```
                return false
            }
        }

        return true
    }
}
```

That takes a copy of the original array, then loops over the other array removing each item it finds until there are none left. If any are missing it returns false, but if it's going through all items successfully it returns true.

## Custom assertions

Previously I explained that it's often useful to isolate common setup code in a method by itself, partly it makes your tests easier to read but partly also because it allows other methods to use that same shared setup code without applying it to all tests.

When making assertions you will usually make one assertion per test, but there are many times when you'll want to make more than one. For example, if you were tasked with writing a method that calculated the result and remainder of dividing one number by another, you might write something like this:

```
func divisionRemainder(of number: Int, dividedBy: Int) ->
(quotient: Int, remainder: Int) {
        let quotient = number / dividedBy
        let remainder = number % dividedBy
        return (quotient, remainder)
}
```

It would be awfully tiresome to have to write two tests for each pair of input values you provided. Really you'd want to write a test like this:

```
// given
let dividend = 10
let divisor = 3

// when
let result = divisionRemainder(of: dividend, dividedBy:
divisor)

// then
XCTAssertEqual(result.quotient, 3)
XCTAssertEqual(result.remainder, 1)
```

Of course, just writing one test isn't enough, because for all we know our method might struggle with numbers over 100 or have some other problems, so you'll write three or four methods like the above providing a range of input. Each time, though, you need to make sure you check both the result and quotient otherwise your tests won't be as thorough as you hoped.

In this instance you might want to write what Gerard Meszaros called a *verification method*: a method whose job it is to wrap multiple assertions for specific values to make our lives easier. In this simple instance, you might write a verification method like this:

```
func verifyDivision(_ result: (quotient: Int, remainder: Int),
expectedQuotient: Int, expectedRemainder: Int) {
    XCTAssertEqual(result.quotient, expectedQuotient)
    XCTAssertEqual(result.remainder, expectedRemainder)
}
```

That makes your tests simpler because you can now call **verifyDivision()** and have it do all the work:

```
// then
verifyDivision(result, expectedQuotient: 3, expectedRemainder:
1)
```

However, there's a subtle problem. Take a look at this test:

```
func testDivisors() {
    // given
    let dividend = 10
    let divisor = 3

    // when
    let result = divisionRemainder(of: dividend, dividedBy:
divisor)

    // then
    verifyDivision(result, expectedQuotient: 3,
expectedRemainder: 2)
}
```

That expects 10 divided by 3 to yield a quotient of 3 and a remainder of 2. This incorrect, but by design: when you run that test, you'll get Xcode's red warning line on **XCTAssertEqual()** call inside **verifyDivision()**, which isn't what you want – it would be much better to know which call to **verifyDivision()** was failing so we could check the values being passed in.

Earlier I mentioned that all XCTest assertion functions have a **message** provider that we can use to provide some clarification when the test fails. Well, they also have two other parameters, **file** and **line**, that say where the problem occurred. Both of these use default values so that we don't have to worry about them for the most part: **#file** is a Swift compiler directive that refers to the name of the current code file, and **#line** is another Swift compiler directive that refers to the line number of the current line of code.

These two values – a string and an integer – are provided for us using those compiler directives, but we can specify our own values when we want to report a specific file and line. Even better, we can use these as default parameters in our verification methods, which means whichever line of code calls them will be used for **#file** and **#line**.

Before I show you the code, there's one small thing you need to know: even though a filename sounds like it could be a regular Swift string, and line number sounds like it could be an integer, Swift actually uses **StaticString** and **UInt** here. The unsigned integer makes sense because code line numbers are never negative, but **StaticString** is a little more unusual – it refers to a string that must have its value known at compile time, thus excluding things like string interpolation. That makes sense here, because all our filenames aren't exactly going to be changing once the app is running.

Anyway, modify the **verifyDivision()** method to this:

```
func verifyDivision(_ result: (quotient: Int, remainder: Int),
expectedQuotient: Int, expectedRemainder: Int, file:
StaticString = #file, line: UInt = #line) {
   XCTAssertEqual(result.quotient, expectedQuotient, file:
file, line: line)
   XCTAssertEqual(result.remainder, expectedRemainder, file:
file, line: line)
}
```

That method now takes two more parameters, both of which have default values so that the call site – anywhere calling **verifyDivision()** – doesn't have to change. Those parameters then get passed straight into the **XCTAssertEqual()** calls, untouched.

With that simple change, running the test again will now show failure on the *call* to **verifyDivision()** rather than on the assertions inside it, making it clear exactly where failures happened.

Verification methods might seem trivial when working with only two assertions, but if you ever need to have three, four, or even five they are incredibly helpful.

# Handling errors

When working with throwing functions it's important that we can test them both working correctly and throwing, so we can make sure specific errors are thrown as they ought to be.

There are at three ways of working with throwing functions, and we're going to look at all three: catching errors in our test, asserting on throws, and writing throwing tests.

We're going to use the same test data for each of these examples we're looking at: a **Game** struct that can be initialized with the string of a game's name. When we call **play()** on a game instance, one of four things will happen:

- If the player asks to play BioBlitz (a two-player strategy game from *Hacking with tvOS*) we'll throw a "not purchased" error.
- If they asked to play Blastazap (a space shooter also from *Hacking with tvOS*) we'll throw a "not installed" error.
- If they asked to play Dead Storm Rising (a top-down strategy game from *Advanced iOS: Volume One*) we'll throw a "parent controls disabled" error.
- All other games will print out a message saying the game is OK to play.

Start by making a fresh Unit Test Case Class named ThrowingTests, then give it this code:

```
import XCTest

enum GameError: Error {
    case notPurchased
    case notInstalled
    case parentalControlsDisallowed
}

struct Game {
    let name: String
```

```
    func play() throws {
        if name == "BioBlitz" {
            throw GameError.notPurchased
        } else if name == "Blastazap" {
            throw GameError.notInstalled
        } else if name == "Dead Storm Rising" {
            throw GameError.parentalControlsDisallowed
        } else {
            print("\(name) is OK to play!")
        }
    }
}

class ThrowingTests: XCTestCase {
}
```

I've put the **Game** struct directly into the same file as the test for convenience – in your own app you'd be testing something from your main app, so you'd need to use **@testable import First**.

## Catching errors in tests

Let's start with the easiest option: catching errors directly in tests, using the same **do**/**try**/**catch** syntax you're used to in regular app code.

So far we've been using XCTest's assertion suite of functions to evaluation a condition and pass a test only if the condition was true. When checking throwing functions, we're going to use a new function called **XCTFail()**, which doesn't evaluate anything – if that code is run the test is immediately considered a failure. Like the regular assertion functions this optionally takes a message along with a filename and line number.

We know that playing BioBlitz should result in an error being thrown. If no error is thrown then code in the **do** block will continue executing, at which point we can safely call **XCTFail()**

because clearly something has gone wrong.

So, we could write a test like this:

```
func testPlayingBioBlitzThrows() {
    let game = Game(name: "BioBlitz")

    do {
        try game.play()
        XCTFail("BioBlitz has not been purchased.")
    } catch { }
}
```

That **XCTFail()** line shouldn't be hit if **game.play()** threw an error, which it ought to if everything is working correctly.

Notice how there's no **XCTSuccess()** call? That function doesn't even exist, because a test is presumed to pass as long as no assertions have failed. So, as long as **game.play()** throws an error that test will pass. This breaks our given-when-then sequence a little because is no "then" – that **XCTFail()** is the closest we have.

Of course, that test isn't perfect. We know that **game.play()** will throw an error, but we also know precisely which kind of error it ought to throw: **GameError.notPurchased**. So, we have a slightly more complicated condition to test: if **game.play()** doesn't throw the test should fail, if it throws **notPurchased** then the test is a success, but if it throws anything else then again the test should fail.

In code it looks like this:

```
func testPlayingBioBlitzThrows() {
    let game = Game(name: "BioBlitz")

    do {
        try game.play()
```

```
            XCTFail()
        } catch GameError.notPurchased {
            // success!
        } catch {
            XCTFail()
        }
    }
```

Much better!

## Asserting on throws

XCTest has two assertions we can use to check whether expressions throw: **XCTAssertThrowsError()** and **XCTAssertNoThrow()**. We know that trying to play Blastazap should play a "not installed" error, so we could write a test as simple as this:

```
func testPlayingBlastazapThrows() {
    let game = Game(name: "Blastazap")
    XCTAssertThrowsError(try game.play())
}
```

However, this is a similar problem to the one we had before: **XCTAssertThrows()** is consuming the error there, so we're not testing the *type* of error thrown.

To make this better we can call **XCTAssertThrowsError()** with a trailing closure that will specify how to handle any thrown errors. It will hand us some sort of **Error**, so we can typecast that to **GameError** and check whether it matches **notInstalled**:

```
func testPlayingBlastazapThrows() {
    let game = Game(name: "Blastazap")

    XCTAssertThrowsError(try game.play()) { error in
        XCTAssertEqual(error as? GameError,
```

```
GameError.notInstalled)
    }
}
```

Again, that's much better.

Like I said, there's also **XCTAssertNoThrow()**, so we can use that to test all other games run without throwing:

```
func testPlayingExplodingMonkeysDoesntThrow() {
    let game = Game(name: "Exploding Monkeys")
    XCTAssertNoThrow(try game.play())
}
```

In case you were curious, "Exploding Monkeys" is a game from the original Hacking with Swift book. I'm always on-brand!

## Writing throwing tests

The third and final option for testing throwing code is to write throwing *tests* – our test methods are just regular methods, so we can mark them as **throws** and let Xcode figure it out. Using this approach, any code that runs without throwing is considered a pass as long as you don't add any assertions that fail, but if any code throws then the test is considered a failure.

Our final throwing game is Dead Storm Rising, and we could test it like this:

```
func testDeadStormRisingThrows() throws {
    let game = Game(name: "Dead Storm Rising")
    try game.play()
}
```

That makes no attempt to handle errors caused by the call to **play()** – they just propagate upwards to XCTest, which will consider that test a failure if the game throws.

That game will always throw, which means that test will always fail. That's by design, though, because I want you to look at the results of the failing test. If you wade through the XCTest output you'll see this:

```
Test Case '-[FirstTests.ThrowingTests
testDeadStormRisingThrows]' started.
<unknown>:0: error: -[FirstTests.ThrowingTests
testDeadStormRisingThrows] : failed: caught error: The
operation couldn't be completed. (FirstTests.GameError error
2.)
```

So, an error was thrown causing the test to fail, and the error was "operation couldn't be completed." That's hardly useful, because it makes no mention of the parental controls error we actually threw.

This can be fixed in surprisingly easily: if we add an extension to **LocalizedError** that provides a default implementation of its **errorDescription** property, we can return the stringified form of the case name that was used for throwing:

```
extension LocalizedError {
    var errorDescription: String? {
        return "\(self)"
    }
}
```

Now we just need to make our error enums conform to **LocalizedError** rather than **Error**, like this:

```
enum GameError: LocalizedError {
```

And now, without making any changes to our test code, the error that appears in the XCTest log is this:

```
Test Case '-[FirstTests.ThrowingTests
testDeadStormRisingThrows]' started.
<unknown>:0: error: -[FirstTests.ThrowingTests
testDeadStormRisingThrows] : failed: caught error:
parentalControlsDisallowed
```

That's a big improvement, but it's still not perfect because that particular game will always throw. This approach is really only a good idea for tests that aren't designed to throw, because as soon as you get into handling errors by hand you might as well assert on throws yourself.

So, let's rewrite the test so that it checks Crashy Plane – another Hacking with Swift project, and one that ought not to throw:

```
func testCrashyPlaneDoesntThrow() throws {
    let game = Game(name: "CrashyPlane")
    try game.play()
}
```

I'll leave it up to you to write a test for Dead Storm Rising using one of the other two methods!

## Repeating tests quickly

You've just been running tests, making changes, then running them again. In small test apps like ours this isn't too much of a pain, but as you progress you'll find yourself using three important keyboard shortcuts:

- Cmd+U: Run all tests.
- Ctrl+Alt+Cmd+U: Run whichever test my text caret is currently in.
- Ctrl+Alt+Cmd+G: Run whichever test I last ran. That might be one test method, a whole test class, or all tests classes depending on your last action.

# Testing the tricky stuff

So far everything we've tested has been relatively straightforward: do some setup, run some code, then check the return value matched what we expected.

However, there are three whole classes of tests you'll need to write that aren't so simple: testing calls to **NotificationCenter**, testing view controllers, and testing anything at all that's asynchronous – i.e., work being done on different threads.

The most common type of asynchronous work is hitting the network – reading or writing data from a remote server – and while testing that is important it's also not what I would call a unit test. Think back to the FIRST rules of unit tests: reading from the network is definitely not fast, and it's also often not repeatable because of network glitches and failures.

In the Test Doubles chapter I'll be walking you through a number of ways you can test networking code effectively using unit tests, but here we're going to focus on general asynchronous work such as calculating prime numbers. This isn't fast (or at least we won't make it fast for the purpose of this chapter!), but it *is* repeatable because it doesn't rely on external functionality.

If you're thinking you can skip asynchronous testing, you're wrong: the iPad Pro has eight CPU cores in total, and all eight can be run simultaneously. This means it's more important than ever to be skilled at concurrent testing, so take the time to learn it right!

## Working with view controllers

View controllers can sometimes be the hardest things to test, but I'll let you into a little secret: I hardly write any unit tests for my view controllers. This isn't because I think they are unimportant (they are anything but!), but because I focus on making them as simple as possible.

In my code, as much as possible, view controllers are as close to being inert as possible. They connect model data to view data, handle view lifecycle events (**viewDidLoad()**, **viewWillAppear()**, etc), and occasionally store some outlets.

Connecting model data to view data is usually just your view controller acting like a pipe: it sends data from one place to the other, then sends updates back the other way if something happened. There is no knowledge encapsulated here – they are effectively inert data carriers, like a piece of string that carries a child's voice between two tin cans. Follow the advice of Brian Gesiak: "push as much logic into your model layer as possible."

As for lifecycle events, it's true that you will sometimes need to test them. For example, you might do a little configuration in **viewDidLoad()**: hiding a button based on some state, or perhaps setting up some labels so they show the right text.

The trick here is that **viewDidLoad()**, **viewWillAppear()** and so on aren't special in any way: you can write a unit test that creates a view controller, call **viewWillAppear()** by hand, and check the results. The only thing you need to be careful with is making sure the view is actually created, like this:

```
let sut = ViewController()
sut.loadViewIfNeeded()
```

Once that's done, you can call **sut.viewWillAppear(false)** or whatever you want to make your lifecycle events happen.

When it comes to testing navigation flow – "does **ViewControllerA** show **ViewControllerB** when the button is tapped?" – you can unit test that by removing **ViewControllerB** entirely from the equation by using the coordinator pattern. This is discussed in more detail in the chapter Dependency Injection.

The final thing you might want to do is check your outlets. This can be a nice part of smoke tests (tests that prove the most important functionality works as expected), because it's easy to disconnect outlets by accident if you're using storyboards.

So, once your view has been loaded go ahead and use **XCTAssertNotNil()** on all your outlets, like this:

```
XCTAssertNotNil(sut.submitButton)
```

These are the simplest tests to write, but they just make sure your outlets remain wired up now and in the future.

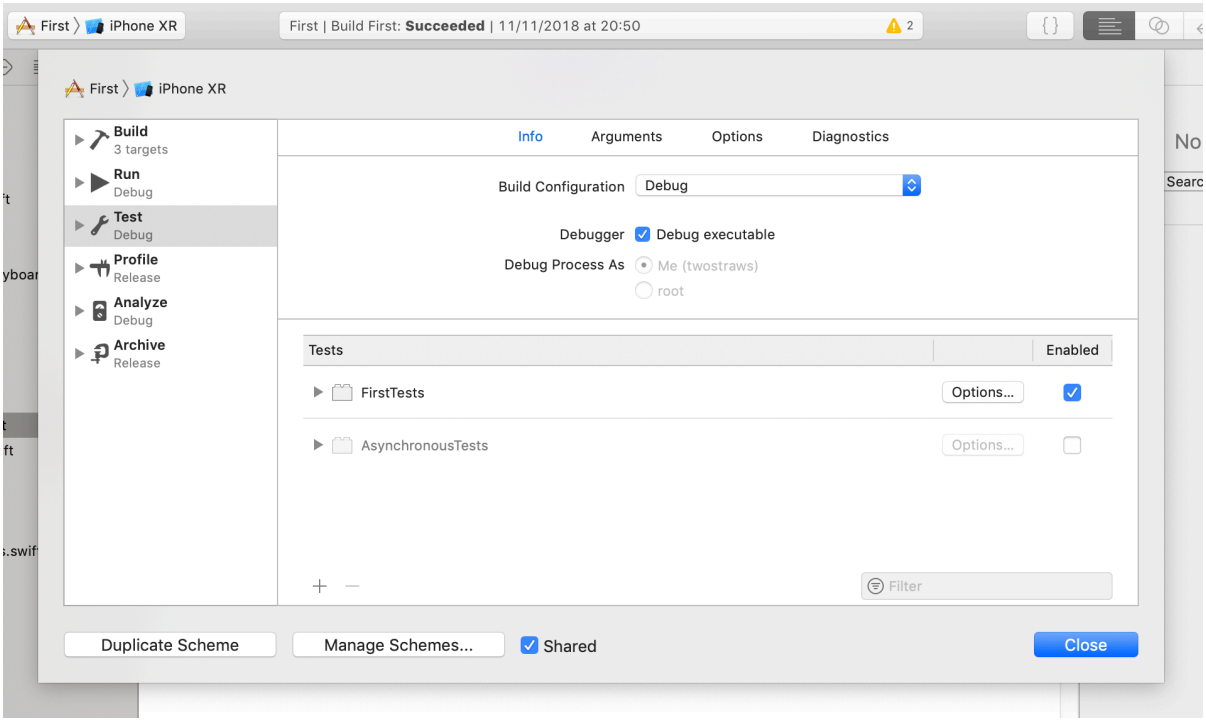
When trying to write tests for view controllers, my main advice is this: as soon as you add **import UIKit** to a file it becomes harder to test. That doesn't make *impossible* – we'll be looking at UI testing in a chapter all of its own – but it's always, *always* better to pull your logic out into separate, smaller data types that are just pure data models.

## Testing asynchronous work

Asynchronous tests are hard for the two reasons I already outlined: they aren't fast, and they often aren't reliable. Even if they *are* reliable, you need to use asynchronous tests carefully so they don't interrupt your daily work flow – if you run tests regularly (which you should!) even reliable asynchronous tests can be annoying, as you'll see.

We're going to start a little differently: we're *not* going to create a new Unit Test Case Class. Instead, I'd like you to go to the File menu and choose New > Target, then choose iOS Unit Testing Bundle. Name it AsynchronousTests, then click Finish.

This creates a whole other separate collection of tests from what we had before. This means we can create all our unit tests in the FirstTests group, and all the asynchronous tests in the AsynchronousTests group. They are both test groups, so by default Xcode will run them both every time we ask all tests to be run (Cmd+U), but it's common practice to run asynchronous tests only on demand so they don't slow down your day-to-day coding. To do that, go to the Product menu and choose Scheme > Edit Scheme. Now choose Test from the left-hand list of options, select the Info tab, then uncheck Enabled next to AsynchronousTests.




**Please leave it enabled for the purpose of this book.** You will want to disable asynchronous tests for your production work, but it's fine for this book because it's all carefully controlled.

Next we need some code to test. You can put this into AsynchronousTests.swift if you like, but obviously in a real app you'd have this in your main app target and use **@testable import First**.

The code we're going to work is a simple implementation of an algorithm called the Sieve of Eratosthenes, which calculates prime numbers up to a maximum number. I like to teach it to children because it's so smart it feels like cheating. Not only does it run screamingly fast, but it's easy enough you can explain in under a minute and remember for the rest of your life.

Here's how it works: mark your entire range of numbers as being prime, so let's say that's 0 to 10. We know that 0 and 1 can't be prime by definition, so we mark those as not prime. Now we loop from 2 up to the maximum of our range: if that number is currently marked prime, then we can mark all its multiples as not prime. So, 2 is prime, which means 4, 6, and 8 are not, so we mark them as not prime. We then continue to the next number, which is 3, and mark its multiples as not prime: 6 and 9. We then continue to 4, but it's already been marked as not

prime so we can continue to 5, and so on.

That's all there is to it, but when it finishes you'll have an array of all prime numbers.

Now, for small values – say up to 100,000 – this algorithm is fast enough that you can run it on the main thread on modern devices, and it will complete so fast you won't drop any frames. But for larger numbers – say a million or even a billion – this is a much slower task, so it's the kind of thing you'll want to run on a background thread to avoid problems. As a result, we'll make our sieve implementation call a completion closure with its result rather than returning a value.

Here's the code – put it into PrimeCalculator.swift if you want to organize your project like a real-world one, or put it into AsynchronousTests.swift if you prefer:

```
struct PrimeCalculator {
   static func calculate(upTo max: Int, completion: @escaping
([Int]) -> Void) {
      // push our work straight to a background thread
      DispatchQueue.global().async {
         guard max > 1 else {
            // if the input value is 0 or 1 exit immediately
            return
         }

         // mark all our numbers as prime
         var sieve = [Bool](repeating: true, count: max)

         // 0 and 1 are by definition not prime
         sieve[0] = false
         sieve[1] = false

         // count from 0 up to the ceiling…
         for number in 2 ..< max {
```

```
            // if this is marked as prime, then every multiple
of it is not prime
            if sieve[number] == true {
                for multiple in stride(from: number * number,
to: sieve.count, by: number) {
                    sieve[multiple] = false
                }
            }
        }

        // collapse our results down to a single array of
primes
        let primes = sieve.enumerated().compactMap { $1 ==
true ? $0 : nil }
        completion(primes)
    }
  }
}
```

Now let's write a test case for that, counting the number of primes from 0 through 100:

```
func testPrimesUpTo100ShouldBe0() {
    // given
    let maximumCount = 100

    // when
    PrimeCalculator.calculate(upTo: maximumCount) {
        // then
        XCTAssertEqual($0.count, 0)
    }
}
```

That sets a maximum value of 100, then asks the prime calculator to figure out all the primes up to that number. Finally, it asserts that zero primes were found.

Go ahead and run that test and it should pass – awesome!

Next, let's move on t— what's that you say? There *aren't* zero primes from 0 through 100? You're right! Numbers like 2, 3, 5, 7, 11, 13, and 17 are all primes, all the way up to 83, 89, and 97.

Either our prime calculator is bad or our test is bad. Given that the former correct implements the Sieve of Eratosthenes – an algorithm that has been around for over 2000 years – then Occam's Razor tell us the test is probably at fault. (Spoiler: it is.)

What's happening here is that XCTest isn't aware that the completion block matters. To be fair, often it doesn't: in iOS you can use **present()** to show a view controller, and it has a completion block that will be run when the show operation finishes. This is sometimes useful, but more often than not we don't use.

To fix this, we need to tell XCTest to wait until our asynchronous code completes using a system of *expectations*. We *expect* that our asynchronous code will complete at some point in the future, and when it does we can make the same kinds of assertions you've seen previously.

To make an expectation you start by creating an instance of **XCTestExpectation** like this:

```
let expectation = XCTestExpectation(description: "Run some asynchronous work")
```

You then start your asynchronous work as normal, and call the **wait()** method afterwards, like this:

```
wait(for: [expectation], timeout: 10)
```

That tells XCTest it should wait up to 10 seconds for the asynchronous code to have finished running.

Finally, the important part: XCTest considers the code to have finished running when we call the **fulfill()** method on it. This happens whenever we want, but it's our way of notifying Xcode that the work has finished so the test can continue.

**As a result of the above, it's important you have completed your assertions before calling `fulfill()` otherwise the test is likely to be considered a pass rather irrespective of your assertions.**

Let's put all this together in a proper test to replace the broken one from before:

```
func testPrimesUpTo100ShouldBe25() {
    // given
    let maximumCount = 100
    let expectation = XCTestExpectation(description: "Calculate
primes up to \(maximumCount)")

    // when
    PrimeCalculator.calculate(upTo: maximumCount) {
        XCTAssertEqual($0.count, 0)
        expectation.fulfill()
    }

    wait(for: [expectation], timeout: 10)
}
```

This time the test will fail, which is good: there aren't zero primes up to 100, there are 25 like the test method name says. It's often useful to try sending your tests incorrect data now and then, just to make sure they work as intended!

Go ahead and change **XCTAssertEqual($0.count, 0)** to **XCTAssertEqual($0.count, 25)** and re-run the test, and this time it should pass.

What value you use for a timeout greatly depends on the task at hand, but if have many asynchronous tests then the worst case runtime for them all is a sum of your timeouts. That is,

if you have 100 asynchronous tests each with a timeout of 10 seconds, the worst case runtime for them is 1000 seconds – about 15 minutes. Hopefully you can see why asynchronous tests are not considered to be unit tests, particularly if they are doing live networking.

# Advanced expectations

Expectations have three useful properties that you might want to use from time to time.

The first is **isInverted**, which flips the expectation around so that fulfilling it is considered a bad thing. For example, if you built an AI for a game you might want to make it wait a couple of seconds between starting its move and making its move, so the player can visually see the phone is "thinking".

To test such a scenario, you might write code like this:

```
// given
let ai = AI()
let expectation = XCTestExpectation(description: "Computer must
wait at least two seconds before taking its turn")
expectation.isInverted = true

// when
ai.startMove() {
    // then
    // pause competed; mark that we finished our turn
    expectation.fulfill()
}

wait(for: [expectation], timeout: 1.8)
```

The expectation timeout is 1.8 seconds, which is a little less than the amount of time the AI has to wait before it makes its move, so in theory that expectation should never be fulfilled. But if in the future we accidentally made the AI jump in faster, it *would* be fulfilled and the test would be considered a failure.

The second useful property is **expectedFulfillmentCount**, which requires that you fulfill an expectation multiple times before it's considered complete. For example, we could write a

second method of our **PrimeCalculator** called **calculateStreaming()**, which takes the same maximum value but calls its closure every time a prime number is found – it streams the results, rather than sending them all back at once.

It would look like this:

```
static func calculateStreaming(upTo max: Int, completion:
@escaping (Int) -> Void) {
   DispatchQueue.global().async {
      guard max > 1 else {
         return
      }

      var sieve = [Bool](repeating: true, count: max)
      sieve[0] = false
      sieve[1] = false

      for number in 2 ..< max {
         if sieve[number] == true {
            for multiple in stride(from: number * number, to:
sieve.count, by: number) {
               sieve[multiple] = false
            }

            completion(number)
         }
      }
   }
}
```

Testing that requires us to set **expectedFulfillmentCount** with a value of 25, because we expect the completion closure to be called 25 times for the numbers 0 through 100:

```
func testPrimesUpTo100ShouldBe25() {
    // given
    let maximumCount = 100
    let expectation = XCTestExpectation(description: "Calculate
primes up to \(maximumCount)")
    expectation.expectedFulfillmentCount = 25

    // when
    PrimeCalculator.calculateStreaming(upTo: maximumCount)
{ number in
        expectation.fulfill()
    }

    wait(for: [expectation], timeout: 3)
}
```

That now checks that the completion closure was called 25 times. If we had set **expectedFulfillmentCount** to 26, the test would wait for its timeout to pass before failing.

If you wanted, you could add individual checks for all the primes: create an array of the correct results, add a counter so we know which one to check each time the closure is called, then use **XCTAssertEqual()** for each one, like this:

```
func testPrimesUpTo100ShouldBe25() {
    // given
    let maximumCount = 100
    let primes = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37,
41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97]
    var primeCounter = 0

    let expectation = XCTestExpectation(description: "Calculate
primes up to \(maximumCount)")
    expectation.expectedFulfillmentCount = 25
```

```
    // when
    PrimeCalculator.calculateStreaming(upTo: maximumCount)
{ number in
        XCTAssertEqual(primes[primeCounter], number)
        expectation.fulfill()
        primeCounter += 1
    }

    wait(for: [expectation], timeout: 3)
}
```

That has the added advantage of checking the primes are sent back in the correct order.

**Tip: expectedFulfillmentCount** is ignored when used alongside **isInverted**, because it doesn't make sense to combine the two together.

The third useful property of **XCTExpectation** is **assertForOverFulfill**, which is important whether or not you've set a custom value for **expectedFulfillmentCount** – even if you don't set that, you're still using the default value of 1 for **expectedFulfillmentCount**.

**assertForOverFulfill** will cause XCTest to exit if you fulfill an expectation too many times. **This is not pleasant**. Quite the contrary: your test will be considered to have passed because you fulfilled at least as many times you said, but this setting literally causes your test to crash. You'll get a long debug log in your terminal, and somewhere in there you'll see this error: **API violation - multiple calls made to -[XCTestExpectation fulfill] for Calculate primes up to 100.**

Annoyingly, this doesn't say how many times it *should* have been called, but at least that's fairly easy to figure out by looking at the test code.

## Watching for progress

Using completion closures is a standard, Swifty approach to working with asynchronous operations, but some code uses an API called **Progress** that requires special handling. **Progress** is a dedicated API for monitoring progress, but hopefully you figured that much out for yourself. It's useful because it is able to carry much more data about some in-flight action, such as whether it has been cancelled or paused by the user. It also allows you to compose task progression, so you could say that task 1 is 100% complete, task 2 is 50% complete, and task 3 is 0% complete, so therefore the whole overall set of tasks is 50% complete.

Testing progress is done with a specific **XCTestExpectation** subclass called **XCTNSPredicateExpectation**, which lets you specify a custom test using string syntax. In our code we're going to check that the **completedUnitCount** of our **Progress** object is equal to 100, which means our expectation will automatically be fulfilled as soon as that becomes true:

```
let predicate = NSPredicate(
   format: "%@.completedUnitCount == %@", argumentArray:
[progress, maximumCount]
)
```

We can then wrap that in an instance of **XCTNSPredicateExpectation**, also providing a progress instance that got returned from somewhere else:

```
let expectation = XCTNSPredicateExpectation(predicate:
predicate, object: progress)
```

To make this work with a real test we need to start by rewriting the **calculate()** method of **PrimeCalculator** so that it returns a progress object that gets manipulated inside the method.

Here's the new code, with comments pointing out the important changes:

```
static func calculate(upTo max: Int, completion: @escaping
([Int]) -> Void) -> Progress {
   // create a Progress object that counts up to our maximum
```

```
number
    let progress = Progress(totalUnitCount: Int64(max))

    DispatchQueue.global().async {
        guard max > 1 else {
            completion([])
            return
        }

        var sieve = [Bool](repeating: true, count: max)
        sieve[0] = false
        sieve[1] = false

        // add 2 to our progress counter, because we already went
through 0 and 1
        progress.completedUnitCount += 2

        for number in 2 ..< max {
            // every time we've checked one number, add 1 to our
completed unit count
            progress.completedUnitCount += 1

            if sieve[number] == true {
                for multiple in stride(from: number * number, to:
sieve.count, by: number) {
                    sieve[multiple] = false
                }
            }
        }

        let primes = sieve.enumerated().compactMap { $1 == true ?
$0 : nil }
```

```
        completion(primes)
    }

    // send back the Progress object
    return progress
}
```

That **return progress** line will be hit almost immediately, because it happens on the main thread while the actual prime number calculation is happening. That gives us the value to feed into **XCTNSPredicateExpectation**, so now we can rewrite our test to use it:

```
func testPrimesUpTo100ShouldBe25() {
    // given
    let maximumCount = 100

    // when
    let progress = PrimeCalculator.calculate(upTo: maximumCount)
{
        XCTAssertEqual($0.count, 25)
    }

    // then
    let predicate = NSPredicate(
        format: "%@.completedUnitCount == %@", argumentArray:
[progress, maximumCount]
    )

    let expectation = XCTNSPredicateExpectation(predicate:
predicate, object: progress)
    wait(for: [expectation], timeout: 10)
}
```

Personally I don't think this approach is anything like as clean as completion closures, but sometimes you won't have a choice - many of Apple's own APIs use **Progess** objects, so there is no other option.

## Testing notifications

I've now shown you multiple ways of testing asynchronous code, and to be honest the remaining two things – notifications and view controllers – are like a walk in the park in comparison!

First, let's look at testing notifications – writing a test that needs to check a notification has been received from somewhere else in the system. This is done using another **XCTestCaseExpectation** subclass called **XCTNSNotificationExpectation**, which will automatically be fulfilled successfully if it detects a specific notification being posted.

As a demonstrating, let's take an app like LinkedIn: there are regular users and there are premium users, and if you're a premium user you get access to a variety of bonus features. Only one part of the app actually deals with the premium upgrade happening, but all parts of the app need to know that it took place so they can enable their own piece of bonus functionality.

So, we might have a **User** class that performs a premium upgrade, and when it finishes posts a notification saying so. Here's a simple example we can test with:

```
struct User {
    static let upgradedNotification =
Notification.Name("UserUpgraded")

    func upgrade() {
        DispatchQueue.global().async {
            Thread.sleep(forTimeInterval: 1)
            let center = NotificationCenter.default
            center.post(name: User.upgradedNotification, object:
```

```
nil)
        }
    }
}
```

I've made that push its work to a background thread so we don't lock up our tests, but then after a one-second sleep it posts the **User.upgradedNotification** notification to the main notification center. It's a simple mock up of the real thing, but it does the job.

Writing a test for this is surprisingly easy: create an instance of **User** and the expectation that **User.upgradedNotification** will be posted, call **upgrade()**, then wait for the expectation to complete successfully.

In code it becomes this:

```
func testUserUpgradedPostsNotification() {
    // given
    let user = User()
    let expectation = XCTNSNotificationExpectation(name:
User.upgradedNotification)

    // when
    user.upgrade()

    // then
    wait(for: [expectation], timeout: 3)
}
```

That test should pass immediately, because our notification fires as expected.

Our solution works for this one particular situation, but it's easy to imagine something more complex: maybe there are gold, silver, and bronze packages that unlock different levels of functionality, for example. In this more advanced scenario the default behavior of

**XCTNSNotificationExpectation** isn't good enough, because it just tells us the user upgraded rather than saying what they upgraded *to*.

In this situation we can attach a custom handler to **XCTNSNotificationExpectation** that lets us evaluate the notification more closely: we can read the name that was posted, the object it was attached to, and also dig through its user info. If we're happy with what we see we just need to return true from the closure to signal success, or return false to signal failure.

First let's update the **User** class so that it still posts one kind of notification, but now attaches some user info saying what level was chosen:

```
struct User {
    static let upgradedNotification =
Notification.Name("UserUpgraded")

    func upgrade() {
        DispatchQueue.global().async {
            Thread.sleep(forTimeInterval: 1)
            let center = NotificationCenter.default
            center.post(name: User.upgradedNotification, object:
nil, userInfo: ["level": "gold"])
        }
    }
}
```

Now we can upgrade our test so that it adds a handler to our **XCTNSNotificationExpectation**. This will be given the notification that was posted and needs to return true or false, so we'll make it return false if there was no "level" setting in our user info or if it wasn't set to "gold":

```
func testUserUpgradedPostsNotification() {
    // given
    let user = User()
    let expectation = XCTNSNotificationExpectation(name:
```

```
User.upgradedNotification)

    expectation.handler = { notification -> Bool in
        guard let level = notification.userInfo?["level"] as?
String else {
            return false
        }

        if level == "gold" {
            return true
        } else {
            return false
        }
    }

    // when
    user.upgrade()

    // then
    wait(for: [expectation], timeout: 3)
}
```

Later on in the Test Doubles chapter we'll be looking at dependency injection as a way of controlling how tests are tracked, but I want to give you a little teaser here because it's particularly important when dealing with notifications.

You see, the problem with notifications is that they are both loosely coupled and extremely distributed. That is, they are designed so that any part of your app can talk to any other part of your app easily, without even knowing what if anything is listening.

While this has many advantages, it causes problems with testing because anything can post notifications and it might confuse our test. We need to be certain that the notification was posted as a direct result of our call to **login()** and not any other reason, and the way to do that is

by using a custom notification center.

Lots of people think that **NotificationCenter** is a singleton, which isn't accurate. Sure, there's a shared notification center we can all post to, but you can also create custom instances of it that act as private communications channels. For this test, we can create a custom **NotificationCenter** instance for our **XCTNSNotificationExpectation**, then send that to our **User** so that it posts notifications only to there.

Even better, we can use a default value of **NotificationCenter.default** when working with our user, so its external actual API hasn't changed – we can inject a value for testing (hence the name "dependency injection") without affecting the rest of our app.

First, change the **User** struct to this:

```
struct User {
    static let upgradedNotification =
Notification.Name("UserUpgraded")

    func upgrade(using center: NotificationCenter =
NotificationCenter.default) {
        DispatchQueue.global().async {
            Thread.sleep(forTimeInterval: 1)
            center.post(name: User.upgradedNotification, object:
nil, userInfo: ["level": "gold"])
        }
    }
}
```

Take particular note of the way I'm allowing a **NotificationCenter** instance to be passed in, but also providing a default value so that most users don't need to care. The actual posting code is the same, but now I've removed the **let center = NotificationCenter.default** line because we no longer need it.

With that new **User** struct we can rewrite our test so that it creates its own instance of **NotificationCenter**, uses that when creating an **XCTNSNotificationExpectation** so we can monitor the results, then pass that to the **upgrade()** method of our user:

```
func testUserUpgradedPostsNotification() {
    // given
    let center = NotificationCenter()
    let user = User()
    let expectation = XCTNSNotificationExpectation(name:
User.upgradedNotification, object: nil, notificationCenter:
center)

    expectation.handler = { notification -> Bool in
        guard let level = notification.userInfo?["level"] as?
String else {
            return false
        }

        if level == "gold" {
            return true
        } else {
            return false
        }
    }

    // when
    user.upgrade(using: center)

    // then
    wait(for: [expectation], timeout: 3)
}
```

The test will still pass, just like it passed before, but now we've removed the risk of something else accidentally posting a notification and causing problems.

**Tip:** Sometimes you might want to add a notification center observer during a test, and that's a perfectly valid solution. However, please make extra sure you remove that observer when you're done, otherwise you might be introducing all sorts of problems by accident – use **addTeardownBlock()** to make sure your observers always get removed. Something like **NotificationCenter.default.removeObserver(yourObserver)** is usually enough.

# Performance testing

Xcode gives us some useful tools for monitoring the performance of our code. While these tests can and should be used to test individual units of code, that doesn't necessarily qualify as being true unit tests. This is partly because you can use them to test whatever you like – unit tests or integration tests – but mainly because Xcode runs each performance test 10 times to avoid performance fluctuations causing hiccups. Even better, Xcode lets you mark performance baselines, and it will consider tests to be a failure if your performance falls a certain distance away.

First, let's resurrect our prime number calculator from earlier on. I've simplified it a little so that it's synchronous, but the concept is the same:

```
struct PrimeCalculator {
   static func calculate(upTo max: Int) -> [Int] {
      guard max > 1 else {
         return []
      }

      var sieve = [Bool](repeating: true, count: max)
      sieve[0] = false
      sieve[1] = false

      for number in 2 ..< max {
         if sieve[number] == true {
            for multiple in stride(from: number * number, to:
sieve.count, by: number) {
               sieve[multiple] = false
            }
         }
      }

      let primes = sieve.enumerated().compactMap { $1 == true ?
```

```
$0 : nil }
        return primes
    }
}
```

Now that we have something to test, let's write a performance test so that we can track performance over time.

Performance tests are done using a single method call: **measure()**. Provide this with a block of work to do and Xcode will make repeat it 10 times, noting down how fast it ran each time. **You should only put code inside `measure()` if you want it be measured** – you might want to put any setup code should outside, for example.

Try adding this performance test:

```
func testPrimePerformance() {
    measure {
        _ = PrimeCalculator.calculate(upTo: 1_000_000)
    }
}
```

That asks the prime calculator to find all primes from zero through to a million, which is likely to take around a second or so depending on your system configuration. Because that runs 10 times, the whole process is likely to take about 10 seconds – see what was saying about these things not really being unit tests?

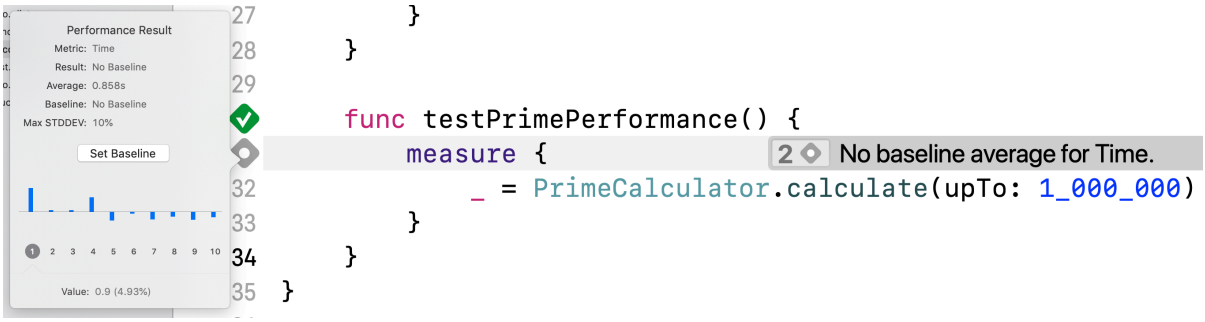
Go ahead and run the unit test, and wait for it to complete. It should show a green checkmark when it finishes, which at this point doesn't mean much. However, you should also see a gray diamond next to the **measure()** call – this is Xcode showing us that it has performance results we can dig into. When you click that gray diamond a popover should appear, offering more details:

- There was no baseline for the test – Xcode has nothing to compare it again to say whether

it was faster or slower than expected.

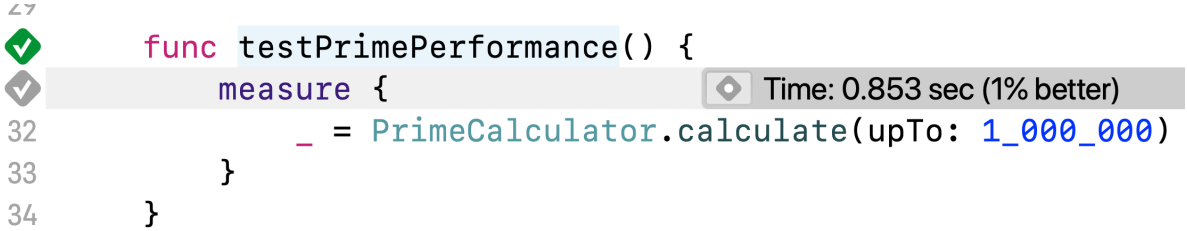
- The average time shows how long each test run took as a mean average – I got 0.858 seconds.
- Max STDDEV is how much float from the baseline Xcode is willing to accept before it considers the test a failure. The default is 10%, which allows for system conditions to fluctuate just a little.

Below that you'll see a bar chart of 10 items, which represents the ten times the test ran. You can click on any of the 10 numbers to see how long each test took to run in seconds, and as a performance difference from the average. For example, my first test run completed in 0.9 seconds, which was 4.93% above the average – well within acceptable.




The most important thing on that screen is the Set Baseline button. This tells Xcode to accept the current results as being a yardstick against which it can measure other tests. So, go ahead and click Set Baseline now.

With that baseline in place, try running the test again – just click the green checkmark next to **testPrimePerformance()**. After another ten seconds have passed you should see the green checkmark return, but this time Xcode will show you how the tests performed against your benchmark: how much faster or slower it was in absolute and relative terms.



Remember, Xcode will only consider the test a failure if it performs significantly slower than the 10% float that is allowed by default. That's usually a good value to start with, but you'll probably want to increase it for very short tasks where even small variations are a lot in percentage terms, and likewise you might want to decrease it for long-running tasks where you can expect much more stable test run times.

Performance testing won't help you find bugs, and it shouldn't really be part of your daily test suite. However, it can help stop your code from getting progressively worse over time – it's a constant background reminder to looking at your app's performance from time to time.

# Monitoring tests

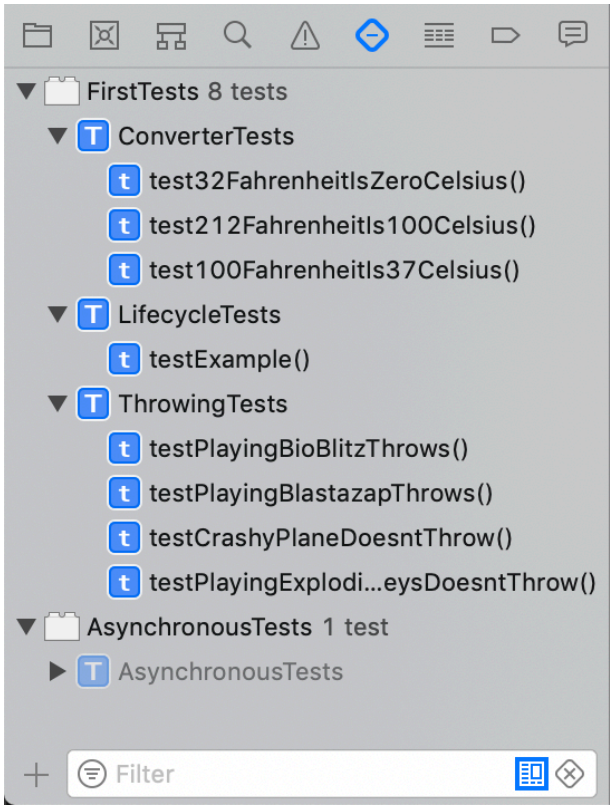
At this point you know enough to write unit tests for a variety of parts of an app, and perhaps you're even raring to go. But before you dive in head first, I'd like to show you a couple more things that will help guide you so you write tests effectively.

In particular, we're going to look at three things: Xcode's test reports, code coverage, and some ways to assess the quality of your tests. Combining these three will help you get a better idea of how well your tests are working, because there's no point writing tests if they aren't pulling their weight!

## Reading Xcode's test reports

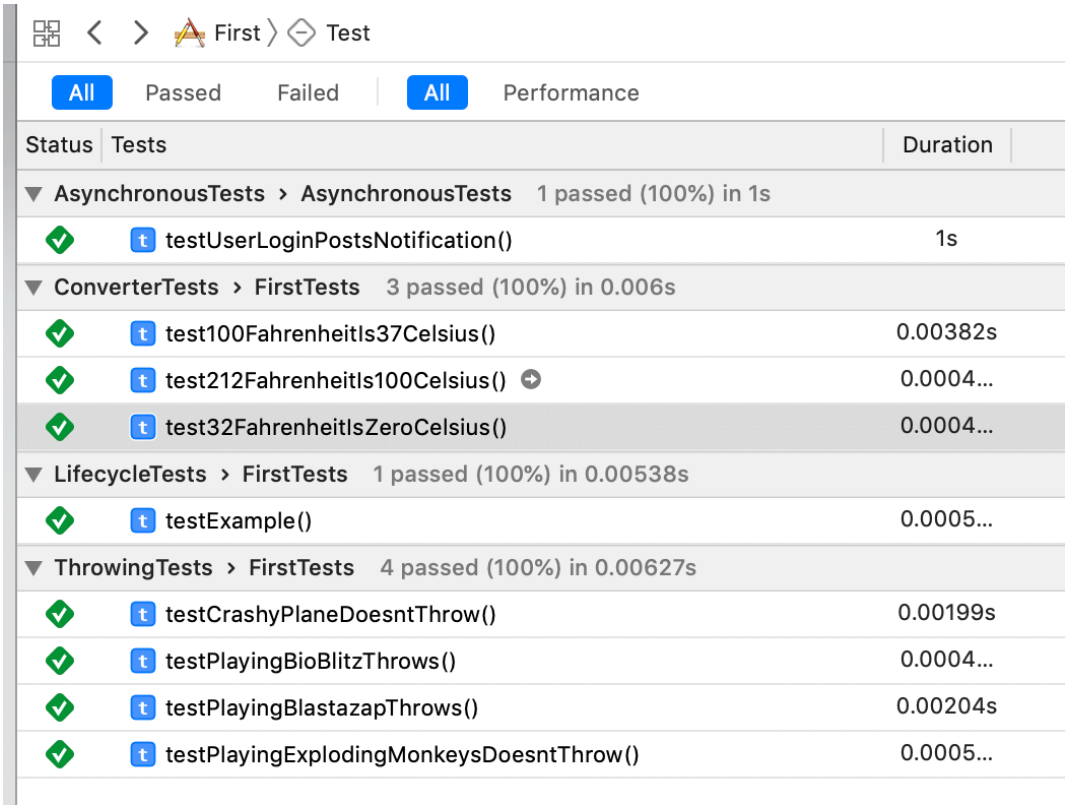
Whenever you run tests Xcode will show you a summary of its results using the test navigator – that's the pane on the left-hand side of the Xcode window that shows lots of green checks and red crosses depending on how things are going.

For larger classes, the filter text field at the bottom will undoubtedly be useful. However, regardless of project size there's one button down there that *will* be useful: it's a little X inside a diamond, and toggles "Show Only Failing Tests". When you have a sea of green checks it can sometimes be hard to see what isn't working, so if you select that option it's much easier to see what's actually happened.




This is a nice overview of your test results, but for even more detail you can right-click on any test there – or any green checkmark in the gutter while you're editing code – and choose Jump to Report. This will show you a complete breakdown of the tests that ran and how long each one took, along with an arrow that takes you straight to the test code.

**Note:** Xcode's test report window has a number of options across the top. Start by selecting the two "All" buttons so you can see the results for all your tests, then select subsets as needed.




There's no hard rule on how fast each unit test ought to be other than "as fast as you can make them" – certainly your computer ought to be able to run at least 100 a second if not significantly more. If you've designed your application and tests extremely well, you might even be able to run 1000 tests in a second. Remember, the faster your tests run the more likely you are to run them regularly.

## Code coverage: a warning

One of the most commonly cited ways of monitoring tests is through code coverage, which is a report telling you how much of your code is covered by unit tests. The logic for this is simplistic, which on one hand means that it's easy for Xcode to figure out, but on the other hand makes it a less than perfect metric.

Code coverage is measured like this: Xcode launches your tests, runs through them all, and simply tracks how many of your lines of code were executed. It can then provide a single figure such as 65%, telling you that 65% of your code was run by your unit tests.

Now, the problem is that running code isn't the same as testing code. To demonstrate this, here's a **House** struct that can be created with a certain number of bedrooms and bathrooms, then checked for suitability by comparing how many bedrooms and bathrooms folks want:

```
struct House {
    var bedrooms: Int
    var bathrooms: Int
    var cost: Int

    init(bedrooms: Int, bathrooms: Int) {
        self.bedrooms = bedrooms
        self.bathrooms = bathrooms
        self.cost = bedrooms * bathrooms * 50_000
    }

    func checkSuitability(desiredBedrooms: Int,
desiredBathrooms: Int) -> Bool {
        if bedrooms >= desiredBedrooms && bathrooms >=
desiredBathrooms {
            return true
        } else {
            return false
        }
    }
}
```

**As I've said, when you're just trying things out it's OK to paste code directly into your Swift tests file. This time, though, I'd like you to add that code to a dedicated House.swift file in your main bundle.** If you don't have it already, make sure you add **@testable import First** to your test case file.

To test out that code, we could write a test like this:

```
func test4Bed2BathHouse_Fits3Bed2BathRequirements() {
    // given
    let house = House(bedrooms: 4, bathrooms: 2)
    let desiredBedrooms = 3
    let desiredBathrooms = 2

    // when
    let suitability = house.checkSuitability(desiredBedrooms:
desiredBedrooms, desiredBathrooms: desiredBathrooms)

    // then
    XCTAssertTrue(suitability)
}
```

That test successfully and correctly evaluates most of the lines in the **House** struct, and if you added a second test with the same house and **desiredBedrooms** of 10(!) then *all* lines in **House** would be run during testing.

However, that doesn't mean our code is being thoroughly tested. Let's take a look at the initializer again:

```
init(bedrooms: Int, bathrooms: Int) {
    self.bedrooms = bedrooms
    self.bathrooms = bathrooms
    self.cost = bedrooms * bathrooms * 50_000
}
```

All three of those lines of code must be run each time we create a **House**, which means they will all be run by the tests above and our code coverage for that initializer is therefore 100%. However, we aren't actually testing those lines: we don't check that **bedrooms** and **bathrooms** are actually stored, and we certainly don't check that **cost** is calculated correctly.

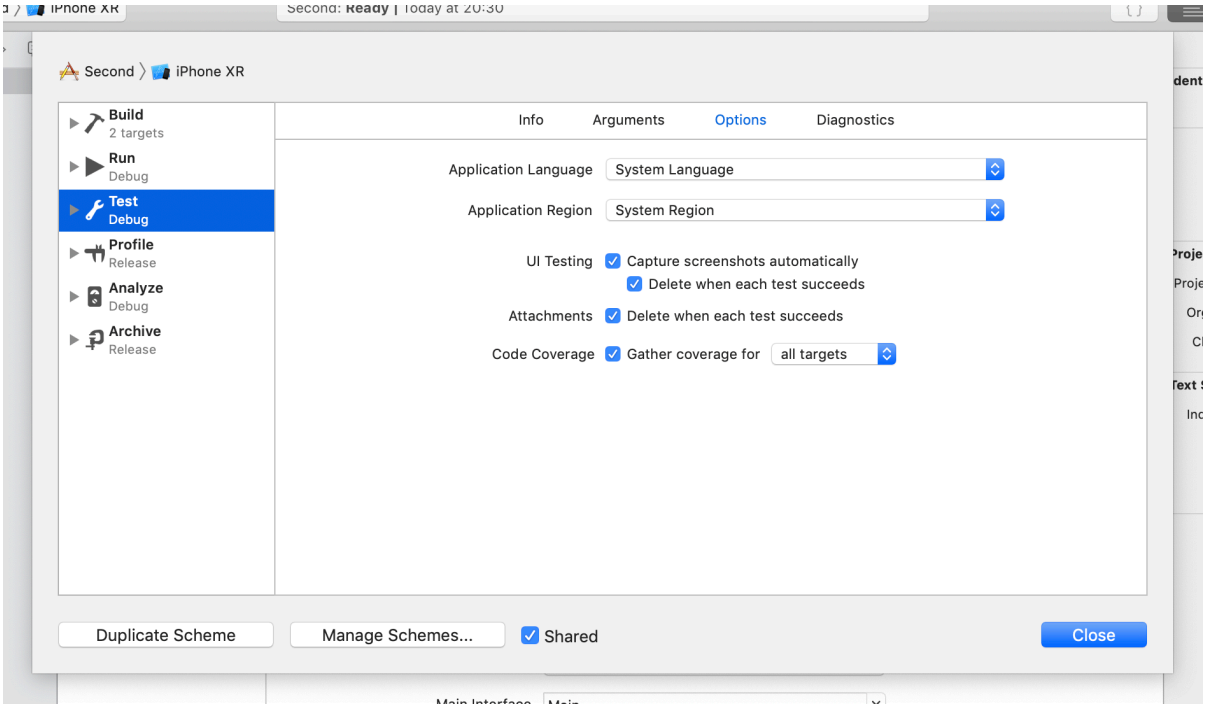
So, even though code coverage can be of some use, I want to make it clear before we dive into

it that it is a thoroughly flawed metric – if it says code isn't covered then fine that's accurate, but if it says code *is* covered that serves as no guarantee that those tests do anything useful.

## Tracking code coverage

OK, now you know the pitfalls of code coverage, let's take a look at it in action. I'd like you to create a new iOS project using the Single View App template, calling it Second. We've been doing a fair amount of creating, deleting, and rewriting tests so far, but to make code coverage easier to discuss it's a good idea to have a clean slate. Make sure you keep the "Include Unit Tests" box checked!

We're not going to write any code or any tests just yet. Instead, the first thing we're going to do is see what kind of code coverage we have right out of the box. So, go to the Product menu and choose Scheme > Edit Scheme. Next, choose the Test scheme and activate its Options tab. Finally, check the Code Coverage box, so that Xcode will gather code coverage data while your tests are running. This is disabled by default for performance reasons.




Now just go ahead and run your tests like normal – pressing Cmd+U ought to build and run the

default test suite, which will contain empty **setUp()**, **tearDown()**, **testExample()**, and **testPerformanceExample()** tests.

When the tests finish, activate the report navigator – it's the last one in the list of navigators, and has an icon like a small speech bubble. Each time you build, run, or test your project, a new entry will be added to this navigator, and you can look in more detail to see exactly what took place.

You've just run this new app's tests, so you should see a test entry in the report navigator. If you open that up you'll see entries for Build, Coverage, and Log – please go ahead and click the "Coverage" option.


All being well, Xcode should report that Second.app has 44% coverage, and if you expand the disclosure indicators you'll see AppDelegate.swift is on 33% and ViewController.swift is on 100% thanks to it having 100% coverage for its lone **viewDidLoad()** method.

Again, saying 100% coverage doesn't mean there are any actual tests for **viewDidLoad()**, it just means that its code was run during the test phase – the app was launched and its view controller created and shown. And even if there *were* test for all that code, having 100% code coverage doesn't mean our app is bug-free, it just means it passes all the tests we wrote.

Let's take code coverage for a test drive using the **House** example I gave earlier. So, create a new Swift file called House.swift in your new project, then give it this content:

```
struct House {
    var bedrooms: Int
    var bathrooms: Int
    var cost: Int

    init(bedrooms: Int, bathrooms: Int) {
```

```
        self.bedrooms = bedrooms
        self.bathrooms = bathrooms
        self.cost = bedrooms * bathrooms * 50_000
    }

    func checkSuitability(desiredBedrooms: Int,
desiredBathrooms: Int) -> Bool {
        if bedrooms >= desiredBedrooms && bathrooms >=
desiredBathrooms {
            return true
        } else {
            return false
        }
    }
}
```

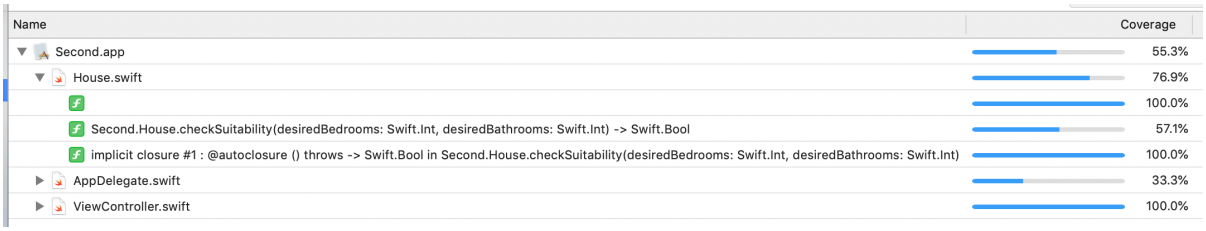
Now open SecondTests.swift and paste in this test method:

```
func test4Bed2BathHouse_Fits3Bed2BathRequirements() {
    // given
    let house = House(bedrooms: 4, bathrooms: 2)
    let desiredBedrooms = 3
    let desiredBathrooms = 2

    // when
    let suitability = house.checkSuitability(desiredBedrooms:
desiredBedrooms, desiredBathrooms: desiredBathrooms)

    // then
    XCTAssertTrue(suitability)
}
```

Finally, press Cmd+U to run all our tests, collecting code coverage along the way. This is a nice example to work with when learning about code coverage, because it shows us three important things.

Open the latest coverage report that was generated by your test run, then open the disclosure indicators so you can see inside House.swift. You should see three things:

- An unnamed function (the green "F" with nothing next to it) is actually the initializer for **House** along with its properties. It's marked as having 100% code coverage even though it's not directly being tested, because we initialize a **House** before calling **checkSuitability()**.
- The call to **checkSuitability()** has 57.1% coverage, which is a bit of a strange number given that it only contains 5 lines of code.
- There's a third entry: "implicit closure #1 : @autoclosure () throws -> Swift.Bool in Second.House.checkSuitability……"




It's easy enough to see why our properties and initializer lines have all been run, but the other two might be confusing.

The call to **checkSuitability()** clearly cannot be said to have 100% code coverage: it has a conditional statement in there, and our test only exercises one half of the code. Xcode reaches its score of 57.1% by counting the following lines:

```
func checkSuitability(desiredBedrooms: Int, desiredBathrooms:
Int) -> Bool {
    if bedrooms >= desiredBedrooms {
        return true
    } else {
```

If you include the function signature and closing brace the method is 7 lines in total, so each line of coverage is worth 14.29%. It's counting the above four in its total, giving 57.16% in total.

Head back to House.swift for a second, then go to the Editor menu and choose Show Code Coverage. This will cause Xcode to show colored bands along the right edge of your source code, telling you which code was and was not run – no color means it was run, a salmon color means it was not run, and salmon stripes mean it was partially run, such as in the case of **} else {**.

The last strange part is the third line of coverage, "implicit closure #1". This is shown as having 100% code coverage, which is lovely – but what is it?

To answer that question I want to show you a small snippet from the Swift standard library. This might sound odd, but it's probably my favorite piece of code in the Swift standard library, and if you've read my other books you'll know I talk about it unrelentingly!

Here's the code:

```
public static func && (lhs: Bool, rhs: @autoclosure () throws -
> Bool) rethrows -> Bool {
   return lhs ? try rhs() : false
}
```

That is the source code for the **&&** operator, which is most commonly used to check multiple conditions in a single **if** statement.

What this code does is take some sort of boolean on its left, and some sort of code that returns a boolean on the right, and returns true if both of them return true. But if the thing on the left is *false* then the thing on the right is never actually checked. This behavior is made possible by autoclosures: Swift silently wraps the right-hand side of the code in a miniature closure so that it's only run when needed.

Now let's return to what the code coverage report is telling us: we have 100% coverage in "implicit closure #1 : @autoclosure () throws -> Swift.Bool in Second.House.checkSuitability……" What this means is that the right-hand side of our **&&** call (**bathrooms >= desiredBathrooms**) is being wrapped in an implicit autoclosure, and *that* has 100% code coverage.

Or, in plainer English: when we wrote **bedrooms >= desiredBedrooms && bathrooms >= desiredBathrooms**, both the **bedrooms >= desiredBedrooms** part and the **bathrooms >= desiredBathrooms** were run.

Try changing the **checkSuitability** method to use || rather than **&&**, like this:

```
func checkSuitability(desiredBedrooms: Int, desiredBathrooms:
Int) -> Bool {
    if bedrooms >= desiredBedrooms || bathrooms >=
desiredBathrooms {
        return true
    } else {
        return false
    }
}
```

Press Cmd+U to re-run all tests, then view the new coverage report - this time you'll see the implicit closure (this time caused by ||) has 0% coverage, because the right-hand side of || is never being evaluated.

So, that "implicit closure #1" line might look weird, but it's extremely helpful for pointing out subtle places where you don't have full coverage.

## What is a good amount of code coverage?

This is a question you will either be asking now or at some point in the near future. And the answer is: there is no answer. Even if we accept that code coverage has some value as a metric

– which it does, unless you actively set out to game it – we still can't really say that a project with 85% code coverage is meaningfully better tested or more stable than a project with 75% code coverage.

Instead, code coverage is at best an indicator of the health of your tests. If some code isn't covered that doesn't mean it's bad or broken, but it does give you a good starting place when you want to write fresh tests. You might look at the code and decide it's not worth testing, and that's OK - in the push to get 100% code coverage results I've seen strange workarounds that take away more than they add.

For example, consider this piece of iOS code:

```
func showDetailScreen() {
    guard let viewController =
storyboard?.instantiateViewController(withIdentifier: "Detail")
else {
        return
    }

    present(viewController, animated: true)
}
```

That's all perfectly sound code, but that **return** line will never to be hit – it will only be run if loading a view controller from your storyboard failed, which ought never to happen. The same goes for using **fatalError()** to abort from situations that must never occur, such as trying to create an **NSRegularExpression** from a hard-coded string. Calls to **fatalError()** cannot be tested because their very nature is that they crash your app, and yet I've seen people jump through hoops trying to create workarounds just to maximize their code coverage number, which is sort of missing the point.

There's a well-respected test author called Brian Marick, who once wrote this about code coverage: "If you treat their clues as commands, you'll end up in the fable of the Sorcerer's Apprentice: causing a disaster because your tools do something very precisely, very

enthusiastically, and with inhuman efficiency - but that something is only what you thought you wanted."

If for some reason you have a company policy around code coverage, I think there are only two life hacks I can recommend. First, turn on code coverage gathering selectively, rather than for all targets – after all, the best way to get high code coverage numbers is to check only the parts that matter, right? More seriously, there is value in only using code coverage selectively, because there's no point having Xcode gather coverage data for third-party frameworks.

Second, you can (and should!) test your view controllers receiving **didReceiveMemoryWarning()**. I use a weird and entirely undocumented hack for doing this, because sadly Apple doesn't give us any sort of official way of triggering memory warnings. Instead, I add this to any test that should be evaluating memory warning work:

```
UIApplication.shared.perform(Selector("_performMemoryWarning"))
```

It's ugly, and there's no doubt it's a hack, but it works.

## Assessing test quality

So far we've learned how Xcode can show us time taken for each test, along with how much of our code has been run. Neither of these tell us anything about how good our tests are, but there is one thing that *might*: mutation testing. This is the practice of introducing deliberately incorrect data into a passing test and running it again to make sure that the mutant copy fails. If the mutant *doesn't* fail it suggests your test might not be as thorough as you had hoped.

Apple provides no support for mutation testing in Xcode, but there is work happening on an open-source project called Mull. You can follow its development here: **https://github.com/mull-project/mull**.

In the meantime, there's nothing stopping you from doing a little mutation testing of your own. I already showed you a few deliberate mistakes: saying that 30 Fahrenheit was 0 Celsius, and saying that there are zero primes from 0 through 100. Try changing **true** to **false** in a test, try looking for different integers coming back, try changing > to <, and so on.

As an example, here's a **Cake** struct that makes a cake baked from a list of ingredients, then sends back a cost calculated by multiplying the ingredients by 3 and adding 1:

```
struct Cake {
    func bake(ingredients: [String]) -> Int {
        for ingredient in ingredients {
            print("Adding \(ingredient).")
        }

        let cost = 1 + (ingredients.count * 3)
        print("The cake is ready; please pay \(cost).")
        return 10
    }
}
```

To test out that code, we could write a test like this:

```
func testThreeIngredientCakeCosts9() {
    // given
    let cake = Cake()
    let ingredients = ["chocolate", "cherries", "frosting"]

    // when
    let cost = cake.bake(ingredients: ingredients)

    // then
    XCTAssertEqual(cost, 10)
}
```

That test will pass, which is great, right? Wrong: try sending in a single ingredient while leaving the cost at 10 and the test will still pass. This shouldn't be the case: with only one ingredient the cost should come down to 4, so the test should fail.

This mutation has found a problem with our cost calculation code. Here it is again:

```
let cost = 1 + (ingredients.count * 3)
print("The cake is ready; please pay \(cost).")
return 10
```

That *should* read **return cost**, but someone hard-coded 10 by accident and the test passed so no one picked up on it.

I'm not advocating that anyone start building complex tests to try to catch every conceivable problem, because there is such a thing as overtesting and at some point you're likely to start introducing bugs into your tests. Instead, be pragmatic: be guided by the tools but not driven by them and you're on the right track.

# Random and parallel testing

Before we wrap up with unit testing, I want to touch on two useful tools in Xcode that can help you with unit tests: random testing and parallel testing.

First, as I said earlier Xcode runs all our tests in alphabetical order, one by one. This sometimes gets implicitly abused by test writers, because one test earlier in the alphabet sets up some state that a test later in the alphabet relies on. If someone later renamed a test so the alphabetical order changed, they would find tests were failing despite Worse, it sometimes gets *explicitly* abused by test writers, which is when you'll see tests named something like **test001addUser()** – they want that one to run before others so that everything works correctly.

Having links between tests, intentional or otherwise, goes against the "I" in FIRST – the idea that tests should be isolated. To solve this, Xcode has a built-in option for running tests in random order: every time the tests run they'll be executed in a different order, which means there's almost no chance of surprise interactions between your apps.

To enable random testing, go to the Product menu and choose Scheme > Edit Scheme. Make sure your Test profile is selected from the options on the left, then choose the Info tab. Now click the Options button next to your test suite and check the box next to Randomize Execution Order. That's it! Next time you run your tests, they'll be randomized. This can help improve the quality of your tests, which will in turn make your tests easier to maintain as your app grows larger.

While this will certainly improve the quality of your tests, it introduces a subtly different problem: if running test A then B then C passes, B, C, A passes, but C, A, B fails, how can you easily verify the fix once you have one? Although Xcode is capable of running your tests in a different order every time, it has no way of saying "please run my tests in the order you used that one time when I hit a failure" so you can verify your fix.

The other useful testing feature is the ability to run your tests in parallel. This is enabled in the same place as random testing, so start by going back to the Product menu and choosing Scheme > Edit Scheme. Make sure your Test profile is selected from the options on the left,

then choose the Info tab. Now click the Options button next to your test suite and check the box next to Execute In Parallel On Simulator.

Parallel testing launches a bunch of simulator clones so that it can run multiple test cases simultaneously. Your tests are then divided up roughly equally among the clones so they do a similar amount of work.

Xcode's approach here is based on **XCTestCase** subclass: it doesn't send individual tests to its simulator clones, but instead whole test classes. This approach allows Xcode to call **class func setUp()** and **class func tearDown()** only once for each class because the code is always on the same simulator clone, but it also helps make sure you don't have any surprise dependencies between test classes – they are literally running in different, isolated simulators.

On the flip side, this means your tests will always take at least as long as your slowest-running test class, so if you're finding there's a bottleneck you should split up that big class into multiple smaller ones. Xcode's test report will tell you which test classes got distributed to which simulator clone, so you should be able to spot problems there.

**Parallel testing may help speed up your test times, but it is not some sort of silver bullet.**

Specifically, there are three areas where it's a bad idea. First, performance testing is an area of testing that relies on precise benchmarking to work correctly – you shouldn't try to run performance tests in parallel, because you'll get bad results almost all the time.

Second, any tests that access some sort of shared resource: a cache file, a database, and so on. If your tests are creating and destroying shared resources such as files or database tables, there's a high chance they will tread on each others' toes and cause sporadic test failures.

Third, if you have very few unit tests and they run quickly, it's possible the CPU cost of spinning up several simulator clones outweighs the cost of your unit tests – you might find parallel testing actually slows you down!

# Chapter 3
## Test Doubles

# A little terminology

Test doubles are a family of solutions to an important problem: allowing us to create unit tests that are truly isolated from the rest of our code. If you recall, "isolated" is the I in FIRST, because the point of unit testing is to test small units of our code in isolation so we can be sure they work independently. Writing integration tests that combine components is important and useful, but it's quite separate from unit testing.

In chapter 2 I said, "the most important thing to do is make sure the code you're testing relies solely on the input you're providing. This means you should provide all the values the test code needs either as initializers or properties." The most important thing is that the system under test – the SUT – is being tested by itself, and not being forced to work with other parts of our code that might introduce problems.

In this chapter we're going to look closely at how this is done, and – more importantly – how it is *faked*. But first, we need to define a little terminology. This is important because there is some confusion around what all these words mean, so if we can work to a common definition for this book it will help makes things smoother.

**Don't skip over this bit.** We'll be using these terms a lot over the coming pages, so it's better to spend a few extra minutes here rather than be confused later.

Let's start with an easy one and move up to the most complex. At the bottom of the test double food chain is a **dummy**: an object that does nothing, but is required for your code to compile because you have parameter lists to fill. So, it might be a class or a struct, and it might conform to any number of protocols, but it literally has no functionality – all its method implements are empty, or as empty as they can be.

Moving on, a **fake** is an object that are actual working implementations of your component, but take some sort of important shortcut that makes testing easier. The most common example is using an in-memory database: it requires no setup or teardown because that will automatically happen in memory. Fakes are particularly useful when they make your tests run faster – an in-memory database will be much faster than a real one, for example.

Going up to the next member of our food chain, a **stub** is an object that always returns fixed data. If you stub your networking code, it means you create a tiny replacement for something like **URLSession** that always returns some fixed JSON so that your tests have something consistent to work with. This is helpful when you need something really simple, because it has no logic or no knowledge other than some hard-coded values. Stubs can return a different kind of test double if you want, such as a fake or a dummy.

Next in line is a **mock**, which has the job of tracking that you called the object correctly and with meaningful parameters. You might use this to check that an object was initialized correctly, that specific methods were called inside it, that specific methods were called a certain number of times, that methods were called in a particular order, and so on. The mock object just tracks all this data so you can verify it later, but doesn't take any meaningful action – its only job is to act like the flight recorder on airplanes and keep track of everything that is happening.

Finally we have **spy** objects, which are similar to mocks except they make actual work happen. So, if you've written a spy to check that your analytics service is tracking data correctly, the spy will monitor that while also sending data to your analytics service.

That completes our food chain. "Test double" is used to describe the whole of those things – the collective word for them all. It's another term from Gerard Meszaros, who likens it to stunt doubles in movies: highly trained substitutes who look similar to actors. So, when someone says "we should put a test double in there," they are leaving open exactly which double should be used. Regardless of which implementation you choose, all test doubles need to conform to the correct protocols required for it to stand in place for a real type.

While all types of test double are useful, in practice I think it's fair to say that mocks and stubs are used the most. Mocks in particular are powerful because they give us just enough feedback to build real things with: we set them up with a series of expectations, trigger some action, then inspect all the work that happened in a single, sealed box, with no actual work happening. Mocks are so common in tests that the word "mocking" is more or less synonymous with "substituting for a test double of some sort."

## Building good test doubles

The point of test doubles is to make unit tests fast, isolated, and repeatable – the FIR in FIRST, as it were. Accessing real files, real databases, or real server cause problems for all three of those things, which is why test doubles are so useful – you *will* use them, and frequently.

When you're just starting out, it's a good idea to place your test double implementations directly into the test you're writing as nested types, like this:

```
func testThatTableViewsRockMySocks() {
    class TableViewDataSourceStub: UITableViewDataSource {
        override func tableView(_ tableView: UITableView,
numberOfRowsInSection section: Int) -> Int {
            return 1
        }

        /* etc */
    }

    // given
    /* etc */
}
```

This helps you get the hang of test doubles in a safe, encapsulated way, with no risk of polluting other tests or trying ahead of time to make re-usable code when you aren't really sure what works yet.

As you progress you'll start to find that you use the same test doubles in more than one place, and realize there's opportunity to refactor your test code. First, awesome – not enough people take the time to polish their test code, and if you're considering doing some test refactoring it's a great sign that you're thinking careful about maintainability. Second, if you carve off your test doubles into separate types, I highly recommend you add their type to the filename and class or struct name: use **TableViewDataSourceStub** for your class name, and store it in

TableViewDataSourceStub.swift. Remember, it's really useful to be able to see at a glance what test code does, and encoding your test double type in its name is a valuable piece of information that couldn't be any easier to add.

As you continue further, you'll find that you can use test doubles as stand ins that call real objects, or you can have real objects calling test doubles. Both are common, and both are important. But don't get carried away with yourself: remember that the ultimate point of this is to simulate your production environment, where real objects communicate with other real objects.

So, only do the minimum amount of work to prove the unit you're testing works as it ought to. The root of Occam's razor is the Latin phrase *entia non sunt multiplicanda praeter necessitatem*, which translates as *more things should not be used than are necessary.* I think Occam was a natural test writer…

# Dependency injection

Dependency injection is such a simple thing, but for some reason the name puts people off – maybe it sounds medically invasive?

Of all the things written about dependency injection, I think this quote from James Shore does more to demystify it than anything else: “Dependency injection is a 25-dollar term for a 5-cent concept.” That is, it might sound complicated, but it’s actually a) obvious, b) probably what you’re already doing a lot of the time, and c) extremely common.

This is the complete, unsimplified description of dependency injection: when you create an object you tell it what data it should work with, rather than letting it query the environment to decide for itself. Martin Fowler calls this “tell, don’t ask”, which is a perfect summary.

One of the most important books written on testing is called *Working Effectively with Legacy Code* by Michael C. Feathers. In there Feathers introduces the concept of hidden dependencies:

*”Some classes are deceptive. We look at them, we find a constructor that we want to use, and we try to call it. Then, bang! We run into an obstacle. One of the most common obstacles is hidden dependency; the constructor uses some resource that we just can’t access nicely in our test harness.”* (Feathers, M.C., *Working Effectively with Legacy Code*, 2005)

Does that sound familiar? This is the problem dependency injection is designed to resolve: hidden dependencies are anything a type draws on that we can’t effectively replace in a test, and they cause problems because suddenly we no longer has complete control over all the inputs and outputs of our system. Instead, some unknown and effectively *hidden* object can affect our test results, which stops them from being as isolated as we expected.

Good dependency injection – giving an object *everything* it needs to function – is such a central part of writing testable code, and leads directly to writing better tests. So, let’s dig into how dependency injection works and how you can use it…

# Interfaces, not implementations

There's a classic computer science book called *Design Patterns* by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides – it's so commonly cited in our industry that it's normally just called the *Gang of Four book*, after its four authors.

Anyway, this book puts forward a number of guidelines for creating better software, and one of the most important is that we should design to interfaces, not implementations. From a Swift perspective this means using protocols rather than concrete types.

Let's start with a nice and simple example to get us moving: a user that can buy apps. For them to be able to buy an app they must have enough money in their account, be over the minimum age for the app, and the app itself must be released.

In code, we could represent the app like this:

```
struct App {
    var price: Decimal
    var minimumAge: Int
    var isReleased: Bool

    func canBePurchased(by user: User) -> Bool {
        guard isReleased else {
            return false
        }

        guard user.funds >= price else {
            return false
        }

        if user.age >= minimumAge {
            return true
        } else {
```

```
            return false
        }
    }
}
```

And the user like this:

```
struct User {
    var funds: Decimal
    var age: Int
    var apps: [App]

    mutating func buy(_ app: App) -> Bool {
        let possible = app.canBePurchased(by: self)

        if possible {
            apps.append(app)
            funds -= app.price
            return true
        } else {
            return false
        }
    }
}
```

Now let's think about how we might unit test that. Remember, unit tests should test one single unit of code at a time, which means we can test either the **User** or the **App** struct, but not both at the same time – that would be an *integration* test.

The **buy()** method of **User** explicitly requires an **App** instance, which means we can't use a test double instead. As for our **App** struct, its **canBePurchased()** method explicitly looks for a **User** instance, so we can't use a test double there either.

This is where designing to interfaces rather than implementations becomes important. When writing a test for this, we want to be able to substitute a test double for either **User** or **App** so that we are testing something that's truly isolated. Designing to an interface means that we should be looking for protocols rather than concrete types, so that we can create some sort test double that conforms to the same protocol and test freely.

For the code above, that would mean starting with a protocol that defines what users and apps should look like:

```
protocol UserProtocol {
    var funds: Decimal { get set }
    var age: Int { get set }
    var apps: [AppProtocol] { get set }

    mutating func buy(_ app: AppProtocol) -> Bool
}

protocol AppProtocol {
    var price: Decimal { get set }
    var minimumAge: Int { get set }
    var isReleased: Bool { get set }

    func canBePurchased(by user: UserProtocol) -> Bool
}
```

I've just taken the property and method names directly from the original structs – in practice you might only choose a subset, or you might create protocol extensions to provide default values.

Once we have those protocols, we can make both **User** and **Struct** conform to them, but we also need to make them *use* these protocols rather than looking for concrete types in their methods. So, we'd replace the existing structs with these:

```
struct User: UserProtocol {
    var funds: Decimal
    var age: Int
    var apps: [AppProtocol]

    mutating func buy(_ app: AppProtocol) -> Bool {
        let possible = app.canBePurchased(by: self)

        if possible {
            apps.append(app)
            funds -= app.price
            return true
        } else {
            return false
        }
    }
}

struct App: AppProtocol {
    var price: Decimal
    var minimumAge: Int
    var isReleased: Bool

    func canBePurchased(by user: UserProtocol) -> Bool {
        guard isReleased else {
            return false
        }

        guard user.funds >= price else {
            return false
        }
```

```
        if user.age >= minimumAge {
            return true
        } else {
            return false
        }
    }
}
```

**Note:** There is a performance cost to using protocols, not least in incurring the hit of dynamic dispatch. Broadly speaking I prefer to start with concrete types where possible, working upwards to a protocol where needed.

Now that our **User** struct can work with anything that conforms to **AppProtocol**, and our **App** struct can work with anything that conforms to **UserProtocol**, we can start to write meaningful unit tests: can the user buy an unreleased app?

First, we need a test double we can work with. Stubs are a great choice for this test, because all we really care about is that the **canBePurchased()** method returns false. So, we might define something like this:

```
struct UnreleasedAppStub: AppProtocol {
    var price: Decimal = 0
    var minimumAge = 0
    var isReleased = false

    func canBePurchased(by user: UserProtocol) -> Bool {
        return false
    }
}
```

It's a minimal implementation of the protocol, with hard-coded values that make it do a specific thing – a *stub*.

Once we have that we can create an instance of our **User** struct and ask it to try buying our stub object:

```
// given
var sut = User(funds: 100, age: 21, apps: [])
let app = UnreleasedAppStub()

// when
let wasBought = sut.buy(app)

// then
XCTAssertFalse(wasBought)
```

That's a nice, isolated test: one object is real and one is a double, we call precisely one method, then make one assertion at the end.

Like I said earlier, to begin with it's good practice to make your test doubles nested type inside your test, to avoid confusing yourself as your tests expand. You can change this easily enough by refactoring those nested types into top-level types, but you should only start doing that when you're found your footing.

So, you might write the complete test like this:

```
func testUserCantBuyUnreleasedApp() {
    struct UnreleasedAppStub: AppProtocol {
        var price: Decimal = 0
        var minimumAge = 0
        var isReleased = false

        func canBePurchased(by user: UserProtocol) -> Bool {
            return false
        }
    }
```

```
    // given
    var sut = User(funds: 100, age: 21, apps: [])
    let app = UnreleasedAppStub()

    // when
    let wasBought = sut.buy(app)

    // then
    XCTAssertFalse(wasBought)
}
```

Because we refactored both **User** and **App** to work with protocols rather than concrete types, we've made it possible to test them in any number of different ways using mocks, stubs, spies, and more. Even better, any surprise dependencies between the two – where **User** expects an **App** to behave in a specific way – simply won't work any more, so the whole thing is a big improvement.

When working with protocols, you might sometimes need to call a static method rather than an instance method. This requires a special method invocation, but it's nothing too fancy.

As an example, we're going to add a static **printTerms()** method to **AppProtocol** so that we can print out the latest iteration of Apple's App Store legalese when someone wants to buy an app. To make things easier, I'll add a default implementation using a protocol extension so that we don't need to add it to our types and stubs unless we specifically want it.

Here's the new **AppProtocol** code:

```
protocol AppProtocol {
    var price: Decimal { get set }
    var minimumAge: Int { get set }
    var isReleased: Bool { get set }
```

```
    func canBePurchased(by user: UserProtocol) -> Bool
    static func printTerms()
}

extension AppProtocol {
    static func printTerms() {
        print("Here are 50 pages of terms and conditions for you
to read on a tiny phone screen.")
    }
}
```

Because that's a static method we can't call it using **app.printTerms()** – we need to call it on the type itself rather than an instance of it. Fortunately, Swift gives us the **type(of:)** method for just this purpose: call it on an instance of any type and you'll get back its class or struct. We can then call **printTerms()** on *that* to get our static method call.

In code you'd write this:

```
type(of: app).printTerms()
```

While I have nothing against static methods – in fact, they serve an important purpose – just make doubly sure that you aren't creating or referencing some shared state there.

# From protocols to injection

Designing to interfaces rather than implementations leads us to dependency injection. At its core, dependency injection lets us provide an object with the data it requires to do its job, but that's an architectural problem rather than a *testing* problem. However, when we're designing to interfaces rather than implementations, dependency injection lets us provide an object with the *protocols* it needs to do its job.

So, in production we use dependency injection to inject production dependencies: real users, real apps, and so on. But in tests we use dependency injection to inject test dependencies: we give real users fake apps, or fake users real apps, and so on. That usually means injecting test doubles using protocols, but sometimes you might inject real objects that have fixed values – a real string, for example, that is fixed to make your tests repeatable. Even better, Swift allows us to provide default values so that production code doesn't even really care that dependency injection is happening.

What you saw above – making a method accept a protocol so that we can inject a stub – is called *method injection*, and it's one of several dependency injection flavors we have to work with. You might also use the following:

- *Constructor injection*, which is providing a value when your object is created.
- *Property injection* is where you set a property directly, after it has been created.
- *Closure injection* is a variant of either property injection or constructor injection depending on how you want to work, where the value in question is a closure that gets run at some important moment.
- *Factory injection* is another variant of property injection, where the property in question is designed to create objects that get used inside the type.

You will almost certainly find yourself using all those at some point, but of all them the first one – constructor injection – is the most useful. Constructor injection is when we create custom Swift initializers that set particular properties in an object, and it's useful because it's crystal clear: you're literally saying "this is the data I want to this type to work with, and nothing else." You can't forget to provide values with constructor injection, because the

initializer requires them.

If you intend to use property injection, make sure you give all your properties sensible default values to avoid the risk of you getting strange behavior because you forget to set something.

When working with constructor injection you should also try to provide sensible default values where possible, because this makes your code feel less clumsy in production. To demonstrate this, let's look at a simple example of constructor injection in practice.

Here's a **Tweet** struct that can be initialized from some text and an author name. When it's created, it pulls out the current date and time and stashes it away, so that all tweets are dated when they are posted:

```
struct Tweet {
    var text: String
    var author: String
    var date: Date

    init(text: String, author: String) {
        self.text = text
        self.author = author
        self.date = Date()
    }
}
```

That's not a complicated implementation, but it reveals a problem: what if we wanted to test certain dates? For example, you might add a method that writes dates in a relative format like "5 minutes ago" or "3 hours ago", and for testing that you'd really need to be able to create tweets with specific dates.

So, you might change the struct to this:

```
struct Tweet {
    var text: String
```

```
    var author: String
    var date: Date

    init(text: String, author: String, date: Date) {
        self.text = text
        self.author = author
        self.date = date
    }
}
```

At this point you might think that all the initializer is doing is assigning each parameter straight to a property, so you could ditch the initializer entirely and rely on Swift's memberwise initializer for structs, like this:

```
struct Tweet {
    var text: String
    var author: String
    var date: Date
}
```

Such a change might seem simple, but it has two problems. First, we've changed the API for that struct: previously we didn't require dates to be provided and now we do, so we need to update all the code elsewhere in the app to provide a date. Second, we've made a change that makes our code more complicated even though almost all the time – every time we create a new tweet – we want to use the current date.

A better idea is to allow a date to be injected, but provide a sensible default of the current date. So, everywhere you post a new tweet you don't need to worry about the date, but when you're testing you can inject a custom date and get guaranteed results every time.

In code, it looks like this:

```
struct Tweet {
```

```
    var text: String
    var author: String
    var date: Date

    init(text: String, author: String, date: Date = Date()) {
        self.text = text
        self.author = author
        self.date = date
    }
}
```

That makes all the dependencies for **Tweet** clear up front, so we won't get any unwelcome surprises later on. This approach allows us to go further and lock down our object even more: we can make all the properties constants so they can be initialized once and never changed again:

```
struct Tweet {
    let text: String
    let author: String
    let date: Date

    init(text: String, author: String, date: Date = Date()) {
        self.text = text
        self.author = author
        self.date = date
    }
}
```

Dependency injection with default parameters is particularly useful when working with singletons – a technique I'll cover in the Mocking chapter.

# Where constructor injection fails… and succeeds

If there's one problem iOS developers are familiar with, it's Massive View Controller syndrome – when one view controller ends up doing far too many things at the same time.

For example:

```
class MegaController: UIViewController, UITableViewDataSource,
UITableViewDelegate, UIPickerViewDataSource,
UIPickerViewDelegate, UITextFieldDelegate,
WKNavigationDelegate, URLSessionDownloadDelegate {
```

What is that view controller responsible for? If someone asked you what it did, could you answer in one breath? Probably not – this feels more like The One Controller To Rule Them All, rather than something that follows the Single Responsibility Principle.

If you wanted to use dependency injection with that view controller, you might end up with something like this monstrosity:

```
class MegaController: UIViewController, UITableViewDataSource,
UITableViewDelegate, UIPickerViewDataSource,
UIPickerViewDelegate, UITextFieldDelegate,
WKNavigationDelegate, URLSessionDownloadDelegate {
    var tableDataSource: UITableViewDataSource
    var tableDelegate: UITableViewDelegate
    var pickerDataSource: UIPickerViewDataSource
    var pickerDelegate: UIPickerViewDelegate
    var navigationDelegate: WKNavigationDelegate
    var downloadDelegate: URLSessionDownloadDelegate

    init(tableDataSource: UITableViewDataSource, tableDelegate:
UITableViewDelegate, pickerDataSource: UIPickerViewDataSource,
```

```
pickerDelegate: UIPickerViewDelegate, navigationDelegate:
WKNavigationDelegate, downloadDelegate:
URLSessionDownloadDelegate) {
        self.tableDataSource = tableDataSource
        self.tableDelegate = tableDelegate
        self.pickerDataSource = pickerDataSource
        self.pickerDelegate = pickerDelegate
        self.navigationDelegate = navigationDelegate
        self.downloadDelegate = downloadDelegate
        super.init(nibName: nil, bundle: nil)
    }

    required init?(coder aDecoder: NSCoder) {
        fatalError("Not supported")
    }
}
```

Can you imagine yourself creating that kind of view controller? I hope not – it's grown way too big for its boots, and to get any sort of meaningful unit tests done you'd need to create so many test doubles. My heads is even spinning after reading the signature of the initializer with all its parameters.

This kind of code makes constructor injection pretty grim, but that's not the fault of constructor injection – this is just bad code. In fact, it's possible that trying to use constructor injection is a lightbulb moment for you: if you're sighing heavily while trying to inject lots of dependencies, that would be a good time to re-think your architectural choices and break things up more. As Andy Matuschak (ex-UIKit engineer, now working at the Khan Academy) once said, "decompose everything."

(If you were curious, the smart thing to do above is to use view controller containment to separate work out. View controllers should be small and composable!)

# Injecting closures

Closure injection is an attractive and relatively simple technique for creating loosely coupled code - code where two things are linked in such a way that you can pull them apart easily because they don't mix up implementation details or more.

Going back to the problem of massive view controllers again, one important weapon in our arsenal is the ability to create custom **UIView** subclasses and use them for our view controllers. So, rather than creating our layout in code inside **viewDidLoad()**, create a custom **UIView** subclass that encapsulates all our layout work.

But this raises a further problem: if our **UIView** subclass creates all our layouts, how do we report back to the view controller when something happened – if a button was clicked, for example? You could use delegates if you wanted, but when you're working with only one or two callbacks a cleaner solution is to use closure injection: to send into the view a closure that should be run when some action has happened.

Let's demonstrate this with a real **UIView** subclass. This needs a few special things:

1. We're going to give it a closure property **(String) -> Void**, which is a function that accepts a string and returns nothing.
2. We'll give it a **shareTapped()** method that we can trigger when a button is tapped. UIKit does this using the target/selector pattern, so we need to mark the method as **@objc**.
3. Inside **shareTapped()** we'll call the closure that was passed in. Sadly there's no way of attaching closures directly to button taps, which is why we have this layer of indirection.

**Note:** If you wanted the action closure to be optional you need to wrap it in parentheses before adding a question mark, like this: **((String) -> Void)?**. Without the extra parentheses you have **(String) -> Void?**, which looks like a function that accepts a string and returns an optional nothing, and I doubt you want a function with an existential crisis in your code.

Here's how we might write such a view in code:

```
class ShareView: UIView {
```

```
    var shareAction: (String) -> Void
    var textField: UITextField!

    init(shareAction: @escaping (String) -> Void) {
        self.shareAction = shareAction
        super.init(frame: .zero)

        let textField = UITextField()
        textField.sizeAndPositionHoweverYouWant()
        addSubview(textField)

        let button = UIButton(type: .system)
        button.sizeAndPositionHoweverYouWant()
        button.addTarget(self, action: #selector(shareTapped),
for: .touchUpInside)
        addSubview(textField)
    }

    required init?(coder aDecoder: NSCoder) {
        fatalError("init(coder:) is not supported.")
    }

    @objc func shareTapped() {
        guard let text = textField.text else {
            return
        }

        shareAction(text)
    }
}
```

When it comes to implementing a view controller to load and show that view, you should start

by overriding **loadView()** so that you can create and use a **ShareView** instance. This is where we're going to inject a method that can be used for the button tap in **ShareView**.

You might start by writing something like this:

```
class ShareViewController: UIViewController {
    override func loadView() {
        view = ShareView(shareAction: shareContent)
    }

    func shareContent(text: String) {
        print("Sharing text…")
    }
}
```

That will build, run, and work just fine, but it's actually caused a subtle problem: the view controller owns the view that it is showing, and we've now given the view a strong reference to a method in the view controller, which means the view now owns the view controllers. Our little closure injection example has caused a retain cycle!

It doesn't take much work to fix this: just create a fresh closure using **weak self**, and call your method from inside there. In code we'd write this:

```
class ShareViewController: UIViewController {
    override func loadView() {
        view = ShareView { [weak self] in
            self?.shareContent(text: $0)
        }
    }

    func shareContent(text: String) {
        print("Sharing text…")
    }
```

```
}
```

This approach becomes particularly useful if you want to make a type more testable without having to create a full test double. Swift really blurs the lines between functions, methods, closures, and operators, and we can use that flexibility in all sorts of ways.

As an example, let's look at a **URLOpener** struct that checks whether a URL has the specific prefix of **internal://**. If it does we'll imagine that it takes some sort of app-specific behavior, such as showing a particular screen or maybe launching an internal browser. All other URLs will get forwarded straight to **UIApplication**, which will perform app switching and open the URL in Safari.

Here's the code:

```
struct URLHandler {
    func open(url: URL) {
        if url.absoluteString.hasPrefix("internal://") {
            // run some app-specific code
        } else {
            UIApplication.shared.open(url, options: [:],
completionHandler: nil)
        }
    }
}
```

That might solve the problem well enough, but hopefully you can see that the call to **UIApplication.shared** represents a hidden dependency – behavior we can't control.

Now, we *could* wrap `UIApplication in a protocol that gets injected. First, we'd create a protocol that defined the part we cared about:

```
protocol ApplicationProtocol {
    func open(_ url: URL, options:
[UIApplication.OpenExternalURLOptionsKey: Any],
```

```
completionHandler completion: ((Bool) -> Void)?)
}
```

Next we'd make **UIApplication** conform to it, which doesn't require any new method implementations because it already has **open()**:

```
extension UIApplication: ApplicationProtocol { }
```

Finally we'd make **URLHandler** store an instance of something that conforms to **ApplicationProtocol**, then use that to call our URL opener:

```
struct URLHandler {
    let application: ApplicationProtocol

    func open(url: URL) {
        if url.absoluteString.hasPrefix("internal://") {
            // run some app-specific code
        } else {
            application.open(url, options: [:], completionHandler:
nil)
        }
    }
}
```

Of course, that really *isn't* the end, because now we need to create some sort of test double that also conforms to **ApplicationProtocol** so that we can test it.

This is a situation where closure injection is much, much simpler. Rather than creating an **ApplicationProtocol** protocol, making **UIApplication** conform to it, building a test double and making *that* conform to it, then adding an **ApplicationProtocol** property to our type, we can just make **URLHandler** have a single property specifying what code should be run to open external URLs:

```
struct URLHandler {
   let urlOpener: (URL,
[UIApplication.OpenExternalURLOptionsKey: Any], ((Bool) ->
Void)?) -> Void

   func open(url: URL) {
      if url.absoluteString.hasPrefix("internal://") {
         // run some app-specific code
      } else {
         urlOpener(url, [:], nil)
      }
   }
}
```

You could even provide a default value to make that closure effectively invisible outside of testing:

```
let urlOpener: (URL, [UIApplication.OpenExternalURLOptionsKey:
Any], ((Bool) -> Void)?) -> Void = UIApplication.shared.open
```

**Tip:** When working with long type signatures like the above, you might want to consider wrapping it in a type alias, like this:

```
typealias URLOpening = (URL,
[UIApplication.OpenExternalURLOptionsKey: Any], ((Bool) ->
Void)?) -> Void
```

You can then refer to **(URL, [UIApplication.OpenExternalURLOptionsKey: Any], ((Bool) -> Void)?)** as just **URLOpening**, which makes your intention clearer:

```
let urlOpener: URLOpening = UIApplication.shared.open
```

# Injecting everything

When working with dependency injection, one term you might hear a lot is "dependency struct" – the idea of passing one struct around that contains your dependencies. Although it's not something I use myself, I can see the appeal: rather than passing around a sea of dependencies that must be swapped in and out in testing as needed, you can instead pass in one single environment struct like this:

```
struct Environment {
    var singleton1: SomeThing
    var singleton2: SomeOtherThing
    var singleton3: AThirdThing
}
```

As you've seen, singletons create hidden dependencies because they allow our objects to read state that wasn't explicitly passed in. The dependency struct approach works around this by having us create one set of singletons when the app launches, and passing that from place to place. Of course, singletons are objects that are created once and only once inside an app, but that doesn't stop different *runs* of the app changing its singleton implementations as seen here.

With that change, the singletons are no longer hidden dependencies – they are passed in explicitly as part of the dependency struct. However, they are no longer painful to change: if we want to test the app in a staging situation, we simply switch the above **Environment** struct for a **StagingEnvironment** struct that has all our singletons swapped out for test doubles, and we're done. You could even mix and match a little: mock several components while working with one live component.

Although this approach is used in many places, it's most famously used in the Kickstarter iOS app – see **http://bit.ly/ksenvironment** for their code on GitHub.

Dependency structs definitely solve an important problem in a useful and clean way, but to me at least they violate the interface segregation principle – that's the "I" in the SOLID principles. This is commonly phrased as "many client-specific interfaces are better than one general-

purpose interface" – it's the idea that each piece of your code should only know about the things it directly relies on, to make your code easier to refactor.

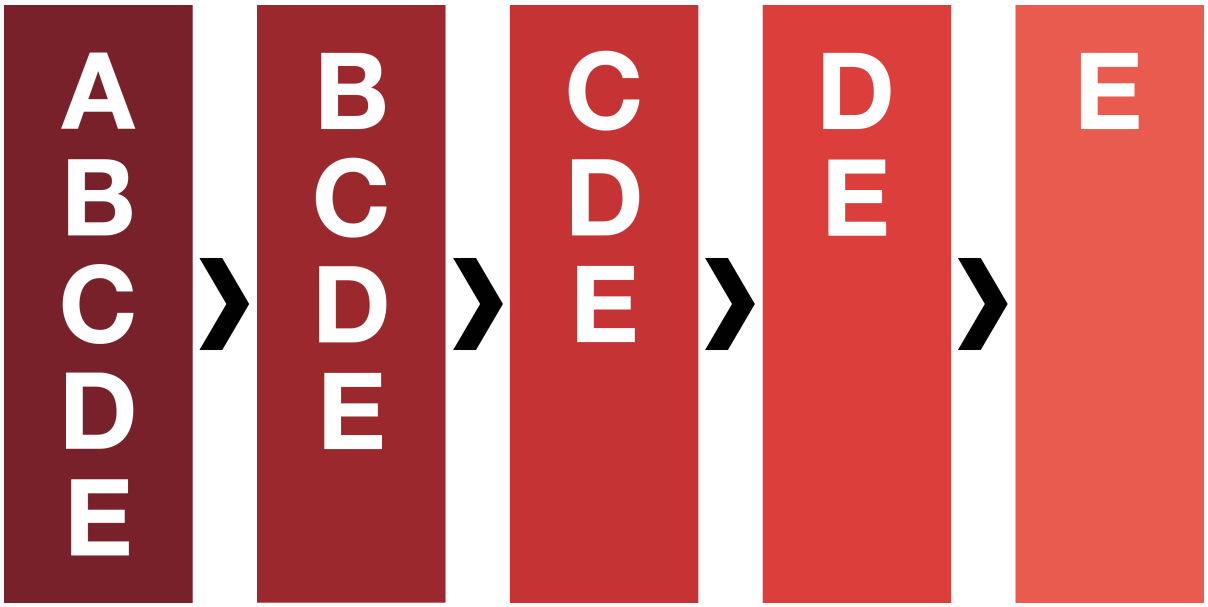
# Coordinators

One helpful way to reduce massive view controllers is the *coordinator pattern*, first introduced by Soroush Khanlou at **http://khanlou.com**. With coordinators each view controller in your app has no idea which view controller comes next in the flow of your application, or even that it's part of a flow at all. Instead, the coordinator uses dependency injection to set itself on every view controller it creates so that the view controller can ask the coordinator to take whatever action should happen next.

I'm mentioning coordinators here, because they teach three useful things at the same time:

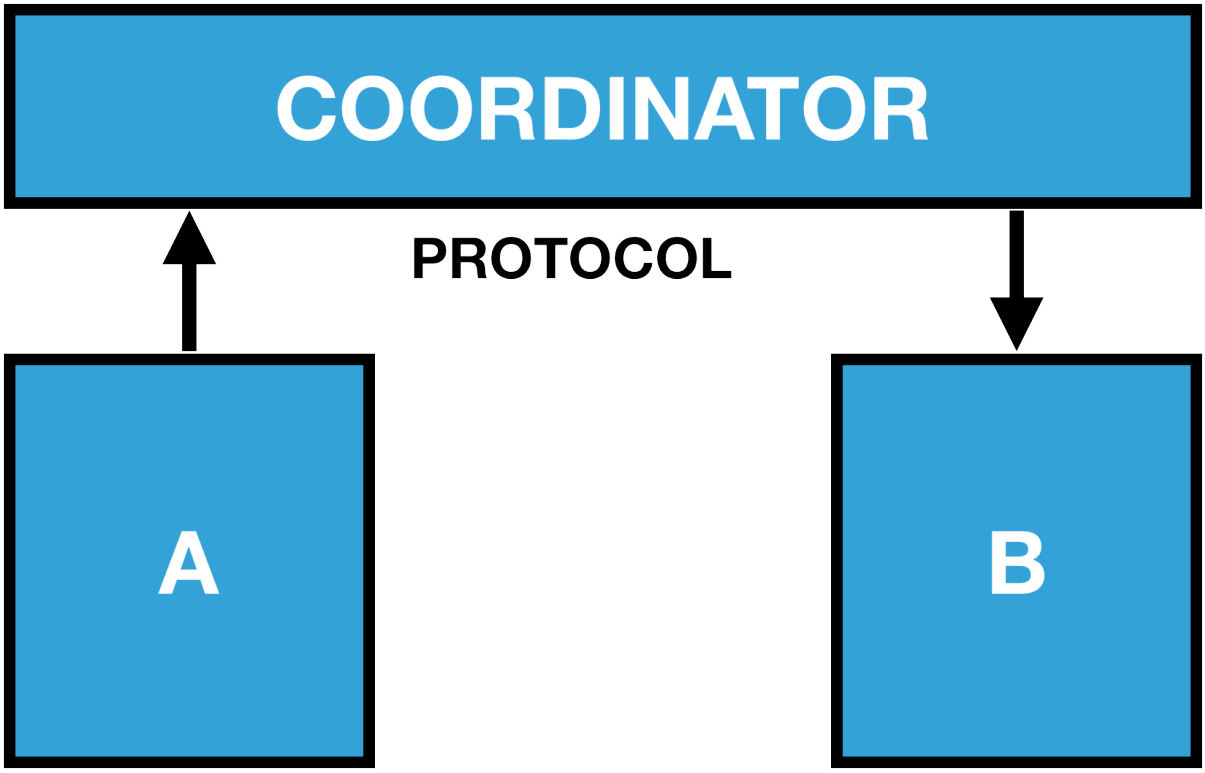
1. They are a great example of property injection: the coordinator creates its view controller, then assigns itself to a **coordinator** property of the new controller.
2. You can switch your coordinators over to using closure injection to create even looser coupling if you prefer.
3. Best of all, it solves the problem of trickledown dependencies: where one view controller has to have its own dependencies along with all those for view controllers it shows, along with all the view controllers shown by view controllers it shows, and so on.

If you're not familiar with trickledown dependencies, perhaps this picture might help jog your memory:




In that diagram, the grandparent view controller on the far left has to know all the dependencies for the view controllers that get presented from it either directly or indirectly. It use dependencies B, C, D, and E, but it also might not; either way it doesn't have a choice: we need to store them there so we can pass them on to children.

Coordinators solve this problem by isolating view controllers: rather than one view controller having to figure out dependencies, it only holds the parts it needs. When it's ready, the view controller calls up to the coordinator saying that's done and the coordinator can create and configure the next view controller.




Whether or not your use property injection or closure injection, the coordinator pattern is a great way to restructure your code to be more flexible.

For the purposes of this book, the best thing about coordinators is that they allow you to unit test your navigation flow because you can create an isolated environment easily.

To see this in action, here's the kind of code we see almost every day as iOS developers:

```
override func tableView(_ tableView: UITableView,
didSelectRowAt indexPath: IndexPath) {
    let someValue = someArray[indexPath.row]

    guard let detailVC =
storyboard?.instantiateViewController(withIdentifier:
"DetailViewController") as? DetailViewController else {
        return
    }
```

```
    detailVC.someProperty = someValue
    navigationController?.pushViewController(detailVC, animated:
true)
}
```

Even though that kind of code is extremely common it can't (reasonably) be unit tested, not least because it reaches into two hidden dependencies (**storyboard** and **navigationController**).

On the other hand, if we used coordinators we could inject a closure such as this one:

```
var showValueAction: ((SomeValue) -> Void)?

override func tableView(_ tableView: UITableView,
didSelectRowAt indexPath: IndexPath) {
    let someValue = someArray[indexPath.row]
    showValueAction?(someValue)
}
```

Now *that* is testable: we can create an instance of our data source, passing in a custom **showValueAction** closure that asserts our code was called. Boom – we're done.

**Tip:** We'll walk through converting an app to using coordinators in the test-driven development chapter.

# Dependency injection vs encapsulation

When you first start using dependency injection, it's common to have your brain rail against it a little bit – particularly if you come from an object-oriented background.

One of the core principles of OOP is *encapsulation*: the idea that classes should be black boxes that we don't reach into. This approach has many benefits, not least in stopping us from reaching into implementation details to create all sorts of coupling mess.

In some respects, dependency injection feels like the antithesis of encapsulation: rather than letting objects work autonomously we are making external objects dictate how they should behave. Surely both approaches can't be right?

If you feel your brain getting sucked into this dilemma, it might help to understand that this is a false dichotomy. That is, it's representing the two problem as two competing states when that isn't really how things are.

The reason we're injecting dependencies is because those dependencies are needed, not because we just want to have them around. So, the thing we're injecting them into has to either have those dependencies sent in, or it has to find them for itself – either way, the dependencies have to come from *somewhere*. The idea that the object was perfectly encapsulated was never actually accurate: it either digs around to find them (implicit, hidden dependencies) or we send them in (explicit, injected dependencies.)

I don't know about you, but I'd much prefer the latter!

# Mocking

Of all the technical skills required by a good testers, I'd say the ability to make good mocks is probably up there in the top three. It's not an easy skill to master, and there are a number of sub-skills inside it that each require time and practice to learn thoroughly, but if you spend the time to master mocking it's a skill that will pay you back for years to come.

The job of a mock is to verify that some sort of behavior happened to the mock. You use mocks just like any other test double, and they straddle the position between dummies and stubs (more or less empty implementations of protocols) and spies (mocks that actually have side effects because they do real work.) This means they must be good enough to behave like the real thing they are standing in for, but still won't cause any actions to take place – it's a sweet spot, really, which is why they deserve extra focus here.

**Important reminder:** I know I've said this a few other times already, but if you're writing tests for **ComponentA** and **ComponentB**, and one of those two isn't mocked or otherwise substituted with a test double, you're writing an integration test not a unit test. There's nothing wrong with that (quite the contrary!) but it's important to keep the distinction clear: unit tests must test one single unit of code at a time.

## Mocking by example

Let's start with an example so simple that hopefully it leaves little room for confusion: people buying houses can view houses they are interested in, and every time they view one we'll add one to a counter:

```
class Buyer {
    func view(_ house: House) {
        house.conductViewing()
    }
}

class House {
```

```
    var numberOfViewings = 0

    func conductViewing() {
        numberOfViewings += 1
    }
}
```

Now, we can test **House** by itself easily enough:

```
func testViewingHouseAddsOneToViewings() {
    // given
    let house = House()

    // when
    house.conductViewing()

    // then
    XCTAssertEqual(house.numberOfViewings, 1)
}
```

This is one of those situations where assuming zero viewings probably makes sense, but you could also write that test to check difference rather than an absolute value:

```
func testViewingHouseAddsOneToViewings() {
    // given
    let house = House()
    let startViewings = house.numberOfViewings

    // when
    house.conductViewing()

    // then
    XCTAssertEqual(house.numberOfViewings, startViewings + 1)
```

```
}
```

Regardless of which approach you choose, testing **Buyer** is trickier: we want to be able to test that viewing a house correctly adds 1 to the viewing count, but we don't want the internal implementation of **House** to get in the way.

As we saw previously, the solution here is to create a protocol that **House** conforms to so that we can put a test double in its place. Yes, even though the **House** class couldn't get any simpler it still needs to be mocked – our **House** class might look simple in this example, but it's likely to expand over time and get other behaviors, whereas our mock will remain small and focused.

So, start by adding this protocol:

```
protocol HouseProtocol {
    var numberOfViewings: Int { get set }
    func conductViewing()
}
```

Next, make **House** conform to that:

```
class House: HouseProtocol {
    var numberOfViewings = 0

    func conductViewing() {
        numberOfViewings += 1
    }
}
```

We'll also need to update **Buyer** so that it uses **HouseProtocol** rather than the concrete **House** implementation:

```
class Buyer {
    func view(_ house: HouseProtocol) {
```

```
        house.conductViewing()
    }
}
```

And now it's time to make a mock of our house. Have a think about this for yourself before continuing: what should the **House** mock look like? What properties or methods does it need?

Don't rush ahead; think about it for a moment.

Still here?

OK, here's my answer: any **House** mock will look identical to the current **House**. Literally identical – it will have the same single property and single method, because there's no way we can make it any simpler. This isn't me being obtuse as you'll see shortly.

For now, though, let's add a **HouseMock** class that is identical to **House**:

```
class HouseMock: HouseProtocol {
    var numberOfViewings = 0

    func conductViewing() {
        numberOfViewings += 1
    }
}
```

With that in place we can write a test for **Buyer**, using our mock rather than a real house:

```
func testBuyerViewingHouseAddsOneToViewings() {
    // given
    let buyer = Buyer()
    let house = HouseMock()

    // when
    buyer.view(house)
```

```
    // then
    XCTAssertEqual(house.numberOfViewings, 1)
}
```

As you can see, **HouseMock** is effectively acting like a flight recorder – it tracks what's happening for us so we can check the sequence of events more easily.

Now, I didn't just choose this simple example to confuse you. You see, not only could the behavior of our **House** class change over time (perhaps relying on more information, or just plain breaking), but it could actually become even *simpler*. For example, we might make **House** perform some sort of action rather than adjusting its state:

```
protocol HouseProtocol {
    func conductViewing()
}

class House: HouseProtocol {
    func conductViewing() {
        print("The buyer looked around the house.")
    }
}
```

That prints a message to the Xcode log, although in a real app you might update the UI, show an alert, or similar. With that small change our original **House** type becomes much harder to test – how can we track whether buyers have viewed a house if houses don't track their view count?

Fortunately, our mock hasn't changed: we can still use it just as before, and it will still correctly track how many times it has been viewed – it's a flight recorded, remember?

**Tip:** One mistake I used to make with mocks was to add a boolean property that got set to true when a method was called. At try! Swift Tokyo 2017, Jon Reid gave a great talk called

“Making Mock Objects More Useful”, where he explicitly calls out this practice as throwing away valuable information, and suggests tracking with an integer instead. This lets us check whether a method was or wasn’t called (1 run or 0 runs), but also whether it was run more than once or precisely once – a subtle but important distinction.

# Partial mocks vs full mocks

What you're seeing with our **House** and **HouseMock** classes are *full* or *complete* mocks, because they create wholly separate types for our testing purposes. The alternative is called a *partial* mock, and involves subclassing some type and overriding whichever parts you want to monitor.

Both types of mocks are important, and I want to demonstrate them side by side so you can see for yourself the difference.

Let's start with the system we're testing: tired of staring at a pile of old iPhones and iPads, you've decided to put them to use by making an app that tracks power outages. The concept is simple: you launch your power monitoring app, which immediately sets **UIApplication.shared.isIdleTimerDisabled** to true so that the screen doesn't turn off, and it then reads **UIDevice.current.batteryState** every 10 seconds to detect whether the device is connected to mains power or not.

Here's how a small slice of the code might look:

```
struct PowerMonitor {
    func getStatus() -> String {
        if UIDevice.current.batteryState == .unplugged {
            return "Power is down"
        } else if device.batteryState == .unknown {
            return "Error"
        } else {
            return "Power is up"
        }
    }
}
```

As you can see, **UIDevice** is a hidden dependency there: we aren't passing it in, but it's directly affecting how the code operates.

We can get a slight improvement by extracting the hidden dependency into an explicit one:

```
struct PowerMonitor {
   var device: UIDevice

   func getStatus() -> String {
      if device.batteryState == .unplugged {
         return "Power is down"
      } else if device.batteryState == .unknown {
         return "Error"
      } else {
         return "Power is up"
      }
   }
}
```

With **device** as a property we must use constructor injection to send in **UIDevice.current**:

```
let sut = PowerMonitor(device: UIDevice.current)
```

Now, how might we test that class? To be able to test all conditions inside **getStatus()** we need to be able to send in all three battery states that are checked: **.unplugged**, **.unknown**, and everything else. We can't rely on **UIDevice** providing that unless you want to sit next to a physical device connecting and disconnecting the power cable, so instead we're going to mock it using partial mocks.

As a reminder, a partial mock is when you subclass a type and override only the parts you want to control. In this case we need to create three mocks, each one returning a different battery state:

```
class UnpluggedDeviceMock: UIDevice {
   override var batteryState: BatteryState {
      return .unplugged
```

```
    }
}

class UnknownDeviceMock: UIDevice {
    override var batteryState: BatteryState {
        return .unknown
    }
}

class ChargingDeviceMock: UIDevice {
    override var batteryState: BatteryState {
        return .charging
    }
}
```

**Note:** I know it sounds obvious, but subclassing only works with classes, which makes partial mocking unsuitable for structs.

With those in place we can finally start writing tests for **PowerMonitor**. Each test must instantiate one of our three device mocks depending on which state it wants to check, send that mock into our real **PowerMonitor** struct, then check the result of its **getStatus()** method.

The three tests are almost identical so there's no point me showing more than one here. This is the test for unplugged devices, which is when our app should show the power is down:

```
func testUnpluggedDeviceShowsDown() {
    // given
    let sut = PowerMonitor(device: UnpluggedDeviceMock())

    // when
    let message = sut.getStatus()

    // then
```

```
    XCTAssertEqual(message, "Power is down")
}
```

That code all works correctly, but we have an opportunity to refactor our tests here. Do we really need three independent classes for doing device mocking? In this situation all we're doing is treating our **UIDevice** subclasses like stubs: we've hard-coded values that also happen to be part of a wider class. A better solution is to make one subclass and tell it what behavior we expect, like this:

```
class DeviceMock: UIDevice {
    var testBatteryState: BatteryState

    init(testBatteryState: BatteryState) {
        self.testBatteryState = testBatteryState
        super.init()
    }

    override var batteryState: BatteryState {
        return testBatteryState
    }
}
```

With that in place, all three of our tests can use the same mock type because they are able to specify what results they expect to see:

```
func testUnpluggedDeviceShowsDown() {
    // given
    let sut = PowerMonitor(device:
DeviceMock(testBatteryState: .unplugged))

    // when
    let message = sut.getStatus()
```

```
    // then
    XCTAssertEqual(message, "Power is down")
}
```

Always take the time to refactor your tests, folks!

So, that's an example of a partial mock. A *full* mock is where we wrap **UIDevice** in a protocol, then send in a complete replacement that conforms to the same protocol.

This takes the same steps we've gone over previously. First, create a protocol that isolates the behavior we care about:

```
protocol DeviceProtocol {
    var batteryState: UIDevice.BatteryState { get }
}
```

Second, make our target type conform to that:

```
extension UIDevice: DeviceProtocol { }
```

Third, add a mock type that implements the same protocol:

```
struct DeviceMock: DeviceProtocol {
    var testBatteryState: UIDevice.BatteryState

    var batteryState: UIDevice.BatteryState {
        return testBatteryState
    }
}
```

Fourth, change our **PowerMonitor** type so that it's designed to an interface not an implementation:

```
struct PowerMonitor {
```

```
    var device: DeviceProtocol

    func getStatus() -> String {
        if device.batteryState == .unplugged {
            return "Power is down"
        } else if device.batteryState == .unknown {
            return "Error"
        } else {
            return "Power is up"
        }
    }
}
```

And finally, write a test that uses the device mock struct. I've actually written the struct to be a drop-in replacement for the **UIDevice** subclass we had before, so our previous test should continue to work just fine.

# Mocking preconditions and assertions

One interesting challenge is how we deal with functions that would otherwise be fatal, such as **precondition()** or **assert()**. (Note: you *could* add **fatalError()** to this list, but I think that's missed the point of fatal error – it's not supposed to be reached!)

Thanks to the way partial mocks work, testing these functions is actually not so hard.

Let's start with some more code we can work with: a **User** class that would store some sort of user details, then a **Store** class that is responsible for letting users buy products. Inside the store's **buy()** method we'll check whether we have a valid user set, and use **assert()** to make sure – it's a programmer error for this method to be called without a user.

In code we'd have this:

```
class User { }

class Store {
    var user: User?

    func buy(product: String) -> Bool {
        assert(user != nil, "We can't buy anything without a
user.")
        print("The user bought \(product).")
        return true
    }
}
```

We could write a test for that like so:

```
func testStoreBuyingWithoutUser() {
    // given
```

```
    let store = Store()

    // when
    let success = store.buy(product: "War of the Worlds")

    // then
    XCTAssertTrue(success)
}
```

However, that test won't pass – it won't even *finish*. You see, the **assert()** call will always fail because we haven't set up a real user, so the **assert()** call will cause our test code to halt. This is by design, because it's a programmer error to try calling this method without a user.

When testing, though, we don't actually care what the method does as long as it correctly halts when called without a user. So, we can apply a technique that I *think* first came from Jon Reid, at least in the Swift world: add our own instance method called **assert()** that matches Swift's own **assert()**, calling straight through to Swift's version in production code but silently acting as a flight recorder while testing.

First we modify our **Store** type so that it has its own **assert()** function. This needs to exactly match Swift's assertion code, so we need to paste the function signature for **assert()** from the Swift source code straight into the class. Inside there we're just going to pass the work on to **Swift.assert()** – the *actual* assert function, not our little wrapper.

You can find the code for **assert()** inside the Swift standard library, under **stdlib/public/core/Assert.swift** – you can browse it online at **http://bit.ly/swiftassert** if you don't have a local clone.

You should find this code:

```
public func assert(
    _ condition: @autoclosure () -> Bool,
    _ message: @autoclosure () -> String = String(),
```

```
    file: StaticString = #file, line: UInt = #line
) {
    // Only assert in debug mode.
    if _isDebugAssertConfiguration() {
        if !_branchHint(condition(), expected: true) {
            _assertionFailure("Assertion failed", message(), file:
file, line: line, flags: _fatalErrorFlags())
        }
    }
}
```

You can get a little sniff of the magic of **assert()** in there – it only actually causes a crash while your program is operating in debug mode.

Anyway, all we care about is the function signature, because we need to put it into **Store**. Here's the code:

```
class Store {
    var user: User?

    func assert(_ condition: @autoclosure () -> Bool, _ message:
@autoclosure () -> String = String(), file: StaticString =
#file, line: UInt = #line) {
        Swift.assert(condition, message, file: file, line: line)
    }

    func buy(product: String) -> Bool {
        assert(user != nil, "We can't buy anything without a
user.")
        print("The user bought \(product).")
        return true
    }
}
```

Now we can create our mock. This will subclass **Store**, then override **assert()** so that it just tracks whether the assertion was successful or not:

```
class StoreMock: Store {
    var wasAssertionSuccessful = false

    override func assert(_ condition: @autoclosure () -> Bool, _
message: @autoclosure () -> String = String(), file:
StaticString = #file, line: UInt = #line) {
        wasAssertionSuccessful = condition()
    }
}
```

With that in place we can now write a test where assertions won't cause a crash. We don't actually care what **buy()** will return, because the thing being tested here is whether our assertion was successful or not:

```
func testStoreBuyingWithoutUser() {
    // given
    let store = StoreMock()

    // when
    _ = store.buy(product: "War of the Worlds")

    // then
    XCTAssertFalse(store.wasAssertionSuccessful)
}
```

The marvelous thing about this approach is that our call site – how **Store** is used in production code – is completely unchanged, and will still call **assert()** as it did before, while in testing we now have the ability to control what assertions do.

# Mocking networking

Networking is where mocking really comes into its own, because it allows us to run simulated network requests that have none of the performance or reliability problems of real network requests. Dominik Hauser and Joe Masilotti did a lot of work to pioneer network mocking in Swift, and their approach is now pretty much the standard.

Let's start with some more example code:

```
class News {
    var url: URL
    var stories = ""

    init(url: URL) {
        self.url = url
    }

    func fetch(completionHandler: @escaping () -> Void) {
        let task = URLSession.shared.dataTask(with: url) { data,
response, error in
            if let data = data {
                self.stories = String(decoding: data, as:
UTF8.self)
            }

            completionHandler()
        }

        task.resume()
    }
}
```

If we want to test that, we need to remove the hidden dependency of **URLSession.shared**. So,

a sensible first step would be to inject the **URLSession** like this:

```
func fetch(using session: URLSession = .shared,
completionHandler: @escaping () -> Void) {
   let task = session.dataTask(with: url) { data, response,
error in
      if let data = data {
         self.stories = String(decoding: data, as: UTF8.self)
      }

      completionHandler()
   }

   task.resume()
}
```

Next we need to create a protocol that lets us mock **URLSession**, which means we need something that lets us call that **dataTask(with:)** method, passing in a completion handler. This works much like the regular partial mocking seen above:

```
protocol URLSessionProtocol {
   func dataTask(with url: URL, completionHandler: @escaping
(Data?, URLResponse?, Error?) -> Void) -> URLSessionDataTask
}

extension URLSession: URLSessionProtocol { }
```

With that in place we can update the **News** class so that it uses the **URLSessionProtocol** instead:

```
func fetch(using session: URLSessionProtocol =
URLSession.shared, completionHandler: @escaping () -> Void) {
```

Now for the tricky part: what do we want to test? There are a few things you might want to check:

1. What was the URL that was requested?
2. Was a network request actually started?
3. Did the request come back with certain data?
4. Did the request come back with a specific error?

We'll work through all four of those, so you can see each one in action.

First, checking the URL that was requested. This requires creating a new concrete implementation of the **URLSessionProtocol** we made above, which will store the URL that was requested. We don't need this thing to do anything else, so when its **dataTask()** method is called we'll stash away the URL, send back an empty data task to satisfy the API, then call the completion handler immediately.

Foundation doesn't let us use **URLSessionDataTask** directly, because it's technically an abstract class. So, we're going to create a mock version that just has an empty **resume()** method – it won't do anything. Technically speaking this is a *dummy*: it doesn't do anything other than satisfy the need for **dataTask()** to return something.

Here's the mocked **URLSessionDataTask**:

```
class DataTaskMock: URLSessionDataTask {
    override func resume() { }
}
```

Now that we have that we can create a **URLSession** mock that tracks the last URL that was requested. We're going to use **defer** so that it automatically calls its completion handler with nil values – we don't care what comes back, because we're just checking that the URL is being requested correctly.

Here's the new mock:

```
class URLSessionMock: URLSessionProtocol {
    var lastURL: URL?

    func dataTask(with url: URL, completionHandler: @escaping
(Data?, URLResponse?, Error?) -> Void) -> URLSessionDataTask {
        defer { completionHandler(nil, nil, nil) }
        lastURL = url
        return DataTaskMock()
    }
}
```

With those two we're now in a position to write a test: does calling **fetch()** on our **News** instance actually request the URL we asked for?

Here's the test code:

```
func testNewsFetchLoadsCorrectURL() {
    // given
    let url = URL(string: "https://www.apple.com/newsroom/rss-
feed.rss")!
    let news = News(url: url)
    let session = URLSessionMock()
    let expectation = XCTestExpectation(description:
"Downloading news stories.")

    // when
    news.fetch(using: session) {
        XCTAssertEqual(URL(string: "https://www.apple.com/
newsroom/rss-feed.rss"), session.lastURL)
        expectation.fulfill()
    }

    // then
```

```
    wait(for: [expectation], timeout: 5)
}
```

Even though that uses **XCTExpectation**, the work is no longer really asynchronous because we've hard-coded it to complete immediately.

Let's move on to the next testable thing: was a network request actually started?

This takes a little more thought, because this code is no longer good enough:

```
func dataTask(with url: URL, completionHandler: @escaping
(Data?, URLResponse?, Error?) -> Void) -> URLSessionDataTask {
    defer { completionHandler(nil, nil, nil) }
    lastURL = url
    return DataTaskMock()
}
```

That uses Swift's **defer** to immediately call our completion handler, which means our mocked networking code will be "completed" before we've even called **resume()** on the mock data task. If you're just checking that the URL was set that's not a problem, but now we need to make sure **resume()** was called on our mocked task.

To correctly test **resume()** being called, we need to extend **DataTaskMock** so that it has a boolean variable storing whether **resume()** has actually been called, but we also need to pass it the completion handler from **dataTask()** so that it can be called only when **resume()** has actually been triggered.

So, the new **DataTaskMock** looks like this:

```
class DataTaskMock: URLSessionDataTask {
    var completionHandler: (Data?, URLResponse?, Error?) -> Void
    var resumeWasCalled = false

    // stash away the completion handler so we can call it later
```

```
    init(completionHandler: @escaping (Data?, URLResponse?,
Error?) -> Void) {
        self.completionHandler = completionHandler
    }

    override func resume() {
        // resume was called, so flip our boolean and call the
completion
        resumeWasCalled = true
        completionHandler(nil, nil, nil)
    }
}
```

That again calls the completion handler with nil values, because we aren't checking what comes back – all we care about is that **resume()** was called.

Now that **DataTaskMock** has been upgraded to track when **resume()** was called, we also need to upgrade **URLSessionMock** so that it can store the task that got created. Remember, this is going to pass the **dataTask()** completion handler straight to the **DataTaskMock** instance so we can trigger it later.

Here's the updated class:

```
class URLSessionMock: URLSessionProtocol {
    var dataTask: DataTaskMock?

    func dataTask(with url: URL, completionHandler: @escaping
(Data?, URLResponse?, Error?) -> Void) -> URLSessionDataTask {
        let newDataTask = DataTaskMock(completionHandler:
completionHandler)
        dataTask = newDataTask
        return newDataTask
    }
```

```
}
```

Finally, we can rewrite our test to check that **resumeWasCalled** is set to true on our session's data task:

```
func testNewsFetchCallsResume() {
    // given
    let url = URL(string: "https://www.apple.com/newsroom/rss-
feed.rss")!
    let news = News(url: url)
    let session = URLSessionMock()
    let expectation = XCTestExpectation(description:
"Downloading news stories triggers resume().")

    // when
    news.fetch(using: session) {
        XCTAssertTrue(session.dataTask?.resumeWasCalled ?? false)
        expectation.fulfill()
    }

    // then
    wait(for: [expectation], timeout: 5)
}
```

The final two things you might want to test are that the requested returned some meaningful data and/or an error. The **fetch()** method we wrote in **News** does this:

```
if let data = data {
    self.stories = String(decoding: data, as: UTF8.self)
}
```

So, when valid data comes back from the URL request, it decodes it to a string and stores it. So, we could make a **URLSessionProtocol** mock with a property that lets us send back

specific data, turning it into a neatly customizable stub. We can then use **defer** to call the completion handler with that pre-set data, leaving **DataTaskMock** back to being a dummy.

Here's the code for that:

```
class URLSessionMock: URLSessionProtocol {
    var testData: Data?

    func dataTask(with url: URL, completionHandler: @escaping
(Data?, URLResponse?, Error?) -> Void) -> URLSessionDataTask {
        defer {
            completionHandler(testData, nil, nil)
        }

        return DataTaskMock()
    }
}

class DataTaskMock: URLSessionDataTask {
    override func resume() { }
}
```

Try extending **URLSessionMock** so it supports a property-injected error as well - it's just a matter of adding a second property then passing that to the completion handler.

Finally, we can write a test that injects some test data and verifies that it comes back successfully:

```
func testNewsStoriesAreFetched() {
    // given
    let url = URL(string: "https://www.apple.com/newsroom/rss-
feed.rss")!
    let news = News(url: url)
```

```
    let session = URLSessionMock()
    session.testData = Data("Hello, world!".utf8)
    let expectation = XCTestExpectation(description:
"Downloading news stories triggers resume().")

    // when
    news.fetch(using: session) {
        XCTAssertEqual(news.stories, "Hello, world!")
        expectation.fulfill()
    }

    // then
    wait(for: [expectation], timeout: 5)
}
```

The nice thing about this solution is that **URLSessionMock** is a complete mock – there are no worries that the real **URLSession** will be firing off network requests in the background because it's never actually used.

# Mocking networking: an alternative

Foundation provides us with an alternative way to test networking, but it's harder to configure because it uses a class rather than an instance of a class. On the flip side, it doesn't require us to mock any part of **URLSession**, because the system is designed to let us inject our own code right in there.

Underneath **URLSession** lies a whole series of **URLProtocol** implementations that handle reading and writing requests. When we ask for something over HTTPS, for example, a **URLProtocol** gets activated to do the actual heavy lifting of making that happen.

Helpfully, **URLProtocol** is designed to be subclassed, and you can then provide your custom subclasses to the **protocolClasses** property of **URLSession** to have them take part in the standard system networking – it's transparent, and gives us complete control over the way networking requests are handled.

Cunningly, Apple's official documentation says "the URLSession object searches the default protocols first and then checks your custom protocols until it finds one capable of handling the specified request." However, in practice I tried it out and our custom protocols get used first: using a default configuration I can fetch data over HTTPS, but injecting a custom protocol class means it gets created instead.

While all this sounds very clever, the problem is that we specify our custom protocol as a class rather than an instance of a class, so if we want to provide test data to our mock then it needs to exist at the class level. Anyway, you might find this solves your needs much more cleanly than mocking **URLSession**, so let's try it out.

Most of the work is done by our **URLProtocol** subclass – and in case you were wondering, despite its name **URLProtocol** is a class rather than a protocol.

This needs to:

- Define a static property that can be used when setting up our tests. It will then be returned when the request begins.
- Return true from **canInit()**, saying that it can handle all kinds of network requests.
- Implement the **canonicalRequest()** method. Apple's documentation says that "it is up to each concrete protocol implementation to define what "canonical" means" – in our case we don't care, so we'll just send back whatever was sent in.
- Implement the **startLoading()** method, which is when the actual URL fetching should begin. We're going to use this to send back our test data and end the loading immediately.
- Implement the **stopLoading()** method. We don't need this but it's required, so we'll just write an empty method.

Here's the code:

```
class URLProtocolMock: URLProtocol {
    // this is the data we're expecting to send back
    static var testData: Data?

    override class func canInit(with request: URLRequest) ->
Bool {
        return true
    }

    override class func canonicalRequest(for request:
URLRequest) -> URLRequest {
        return request
    }

    // as soon as loading starts, send back our test data or an
empty Data instance, then end loading
    override func startLoading() {
        self.client?.urlProtocol(self, didLoad:
URLProtocolMock.testData ?? Data())
```

```
        self.client?.urlProtocolDidFinishLoading(self)
    }

    override func stopLoading() { }
}
```

When it comes to writing a test, we no longer need to create any sort of **URLSession** mock. Instead, we configure our **URLProtocolMock** class with whatever test data we expect to receive back, then ask **URLSession** to use that class as part of its networking. This is done in three steps:

1. Create an instance of **URLSessionConfiguration**, using **ephemeral** as our starting point so we don't get any sort of caching.
2. Set our configuration's **protocolClasses** property to an array containing **URLProtocolMock.self**.
3. Create a real **URLSession** from that configuration, and pass it into our **fetch()** method.

Here's the code:

```
func testNewsStoriesAreFetched() {
    // given
    let url = URL(string: "https://www.apple.com/newsroom/rss-
feed.rss")!
    let news = News(url: url)
    let expectation = XCTestExpectation(description:
"Downloading news stories triggers resume().")
    URLProtocolMock.testData = Data("Hello, world!".utf8)

    let config = URLSessionConfiguration.ephemeral
    config.protocolClasses = [URLProtocolMock.self]
    let session = URLSession(configuration: config)

    // when
```

```
    news.fetch(using: session) {
        XCTAssertEqual(news.stories, "Hello, world!")
        expectation.fulfill()
    }

    // then
    wait(for: [expectation], timeout: 5)
}
```

Because we no longer have any sort of **URLSession** mock, our **fetch()** method is back to how it was originally:

```
func fetch(using session: URLSession = .shared,
completionHandler: @escaping () -> Void) {
```

## Mocking many requests

While the code for this is both cleaner and simpler than a complete mock of **URLSession**, I feel that the static property on **URLProtocolMock** is silently judging me. So, a better thing to do is change our static **Data** instance to a dictionary that maps URLs to data. We can then register any URL we like and provide it the data to return, allowing us to mock multiple URLs at the same time.

Here's the code:

```
class URLProtocolMock: URLProtocol {
    // this is the data we're expecting to send back
    static var testURLs = [URL: Data]()

    override class func canInit(with request: URLRequest) ->
Bool {
        return true
    }
```

```
    override class func canonicalRequest(for request:
URLRequest) -> URLRequest {
        return request
    }

    // as soon as loading starts, send back our test data or an
empty Data instance, then end loading
    override func startLoading() {
        if let url = request.url {
            if let data = URLProtocolMock.testURLs[url] {
                self.client?.urlProtocol(self, didLoad: data)
            }
        }

        self.client?.urlProtocolDidFinishLoading(self)
    }

    override func stopLoading() { }
}
```

To use that, just register your dictionary of URLs and data ahead of time, like this:

```
let url = URL(string: "https://www.apple.com/newsroom/rss-
feed.rss")!
URLProtocolMock.testURLs = [url: Data("Hello, world!".utf8)]
```

# What not to mock

You've now seen how to create a variety of mocks, both partially and completely, so hopefully you now understand why they are so important. I know this is probably starting to sound like a broken record, but unit tests really do need to be Fast, Isolated, Repeatable, Self-Verifying, and Timely if they are to be useful, and mocks do a fantastic job of helping us with the first three of those.

Which mocking approach you choose depends greatly on what you're trying to do. You've seen how we were able to subclass **UIDevice** to return specific battery states for the power tracking app, but the subclass-and-override approach won't work everywhere – **UIApplication**, for example, causes a hard crash if you try to create one, so you need to create a complete mock with a protocol instead.

Testing mocks is a bit like sitting an IQ test: in theory a high score on an IQ test means you're really clever, but in practice it just means you're good at taking IQ tests. When you test with mocks, in theory it means your code is working as you intended, but in practice it just means your mocks are working as you intended – it's possible your actual production code might choke on situations that your mocks are fine with.

So, when you're mocking the best piece of advice I can give you is to do it as little as possible and as lightly as possible. That means only mock when you really need to, and even then only mock the parts that you absolutely must. This ought to increase the usefulness of your mocks: they aren't detailed, they aren't fancy, and they don't do anything more than the absolute minimum required.

# Working with test data

Being able to work with test data is useful in any test environment, and there are two ways that you might come across.

First, if you're using Core Data you can set up a fake, in-memory persistent store by asking for an **NSInMemoryStoreType** description:

```
let container = NSPersistentContainer(name: "User")
let description = NSPersistentStoreDescription()
description.type = NSInMemoryStoreType
container.persistentStoreDescriptions = [description]
```

The rest of your Core Data code is unaffected, and in fact blissfully unaware that it's all going to be zapped as soon as the test finishes – the memory just gets wiped and reset each time.

More common is the need to load specific test data from the bundle – an example JSON response that you want to use in a network mock, for example.

Your test bundle is an instance of **Bundle** its own right, but it's *not* **Bundle.main** – that's our app bundle. If you want to load a file in your tests, start by dragging it into your test bundle in the project navigator – below your test case file is fine.

Next, use **Bundle(for: type(of: self))** in your test code to find the **Bundle** instance responsible for the current test case. Once you have that, you can load assets from there as you normally would – it's just a regular **Bundle**.

You might find a helper method such as this one is a useful addition to your code – it looks for and loads any file you've added to your test bundle, returning a **Data** instance if it was found or nil otherwise:

```
func data(for filename: String) -> Data? {
    let bundle = Bundle(for: type(of: self))
```

```
    guard let url = bundle.url(forResource: filename,
withExtension: nil) else {
        return nil
    }

    return try? Data(contentsOf: url)
}
```

# Chapter 4

# User Interface Testing

# A UI testing primer

You've seen how unit tests are designed to evaluate one single piece of your app at a time, with other components needing to be mocked in order that we can get true isolation of functionality. I've also explained that when you remove the mocks – if you use two of your code units together once – then you have integration testing. There's nothing magic about integration tests: they are just unit tests without mocking.

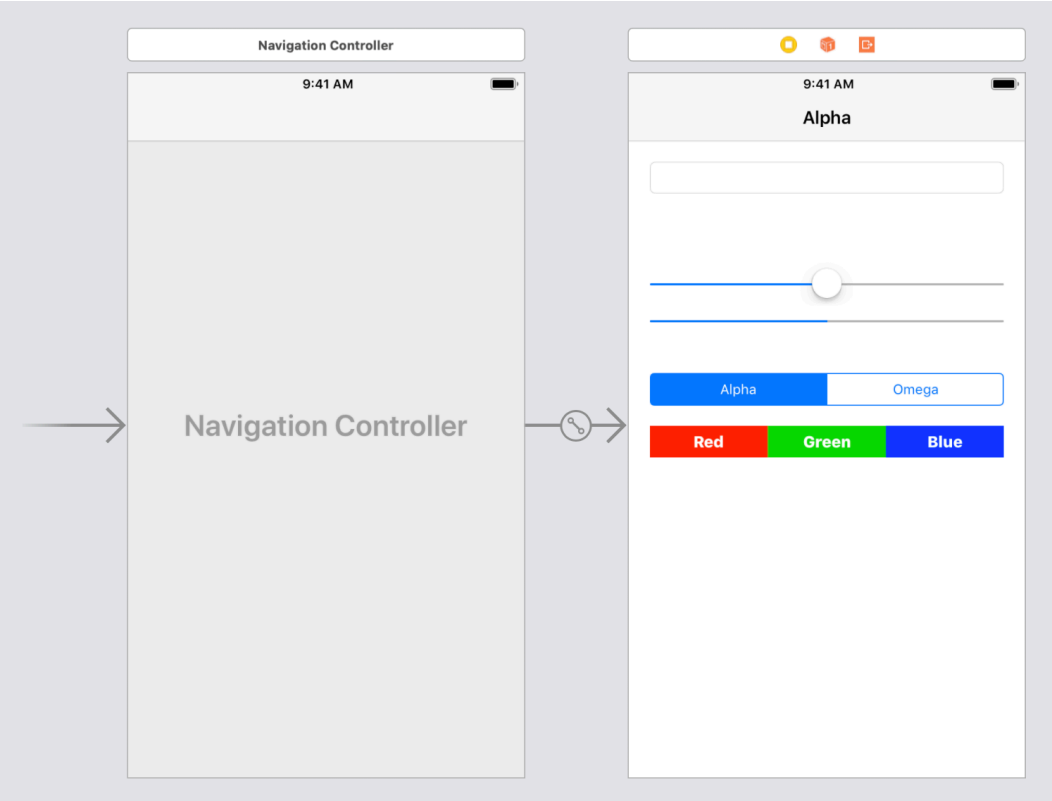
Well, it's time to move up to the top of our pyramid and look at user interface testing. This is effectively the ultimate form of integration testing, because it tests that everything you have written comes together to launch and run a real app – that your users gets an actual app with actual functionality, just as you planned.

Unit tests have explicit knowledge about the workings of your code: you can create instances of your custom types, inject properties, call methods, and so on. At the other end of the spectrum, user interface testing has nothing like that at all: it has no view into your code whatsoever, and instead is forced to interact with your app just like a user does. You're testing that the finished product looks and works as you expect, which means you're running over a lot of functionality at the same time.

Obviously any user interface tests must have some sort of user interface to work with, so in the source code for this book I've provided you with XCUITestSandbox – a project that has a simple user interface that gives us enough to get started with:

- If you enter text into the text box you'll see the same text appear in the label below.
- Dragging the slider will update the progress view; they should move in opposite directions.
- Toggling the segmented control will change the title in the navigation bar.
- Clicking any of the three colored buttons will present an alert.

This sandbox gives us a variety of controls to work with, which should allow for some interesting user interface tests.




If you didn't see the on-screen keyboard when the text field was selected, you should go to the Hardware menu and uncheck Keyboard > Connect Hardware Keyboard. XCUITest does *not* work great with hardware keyboards, so you should use the on-screen one to be sure.

The sandbox app I've given you doesn't have anywhere to add user interface tests, which is intentional: if you're reading this chances are your own app also doesn't have any UI tests, so you need to know how to get started!

The first step is to add a new target: go to the File menu and choose New > Target. You'll see a long list of possible targets, but I'd like you to scroll down and select iOS UI Testing Bundle then click Next. Xcode's default options might be OK, but check the team option to make it's configured for your team – you're using a project project I created, after all.

Finally, look inside the XCUITestSandboxUITests group and open XCUITestSandboxUITests.swift for editing. This is where we'll be putting all our user interface testing code.

## Setting an initial state

Like all the regular UI tests we've looked at so far in this book, you'll see **setUp()** and **tearDown()** methods that run before and after your tests to make sure they are in a consistent state. You'll also see an empty test method to get us started, called **testExample()**.

For now I want to focus on the **setUp()** method, which should contain code like this:

```
// In UI tests it is usually best to stop immediately when a failure occurs.
continueAfterFailure = false

// UI tests must launch the application that they test. Doing this in setup will make sure it happens for each test method.
XCUIApplication().launch()

// In UI tests it's important to set the initial state - such as interface orientation - required for your tests before they run. The setUp method is a good place to do this.
```

That third comment is important: you need to make sure the app is configured in a very specific way, otherwise you're likely to have tests fail sporadically.

While you *can* put code after that comment to effectively force your app to be in a specific state, most good tests I've seen use a much more robust approach: pass in a command-line parameter that the app can read and adapt to.

To pass in a testing flag, replace the call to **XCUIApplication().launch()** with this:

```
let app = XCUIApplication()
app.launchArguments = ["enable-testing"]
app.launch()
```

That will pass "enable-testing" to the app when it's launched, which we can then make it

detect and respond to. Our sandbox app doesn't have any initial state to worry about, but if you needed to configure your app in a certain way then you'd add something like this to your main app:

```
#if DEBUG
if CommandLine.arguments.contains("enable-testing") {
    configureAppForTesting()
}
#endif
```

What **configureAppForTesting()** does is down to you – you might load some fixed data, for example, so that your test stubs work as expected.

## Recording tests

To begin with, we're going to use Xcode's test recording system to write some test code. This is where Xcode will literally watch what you do inside the simulator, and attempt to translate your taps and swipes into Swift code. I say "attempt" because your results are likely to be mixed at best, as you'll see

Click inside the **testExample()** method, and you should see a red recording button become active directly below the code area – next to where the breakpoint and debugging buttons are. Go ahead and press that button now, and you should see Xcode build and launch your app.

Here's what I'd like you to do when the app launches:

1. Click inside the text field, enter "test", then press enter.
2. Drag the slider all the way to the right.
3. Click "Omega" from the segmented control.
4. Click the Blue button then dismiss the alert.

As you take those actions, code will be written directly into the **testExample()** method, more or less matching what you're doing on-screen. I say "more or less" because in practice this is

the flakiest part of XCTest: it generates terrible code that often won't compile.

Here's the code it generated for me when I followed the above actions:

```
let app = app2
app.children(matching: .window).element(boundBy:
6).children(matching: .other).element.swipeLeft()

let tKey = app.keys["t"]
tKey.twoFingerTap()

let app2 = app
app2.keys["e"].twoFingerTap()
app2.keys["s"].twoFingerTap()
tKey.twoFingerTap()
app2.buttons["Return"].twoFingerTap()
app.sliders["50%"].swipeRight()
app2.buttons["Omega"].twoFingerTap()
app.buttons["Blue"].twoFingerTap()
app.alerts["Blue"].buttons["OK"].twoFingerTap()
```

**Note:** Wherever you see some code with a blue background color, click the micro-sized arrow to the right of it to see other possible interpretations of that code.

As bad as that is, don't be surprised if your own code is substantially worse – it's luck of the draw, from what I can tell. I don't know what code was generated for you, but my code above has a number of problems:

1. The very first line, **let app = app2**, is meaningless because **app2** isn't defined.
2. The second line inserts **(boundBy: 6)** for some unknown reason – that's its way of reading item 6 in all the window.
3. It claims I tapped everything using two fingers.
4. The return key is accessed as a button rather than a key. While this might work, it means

any other buttons with the text “Return” will confuse the system.

5. Our slider is referred to as **sliders["50%"]** – that’s its value, which means it will change.
6. The slider movement is referred to simply as **swipeRight()**, rather than a precise movement to a value.
7. The “Omega” segmented title is under “Buttons”, which again will work but might confuse things later on.

This code doesn’t actually *test* anything yet, but before we can write tests we need to make it work. So, please change it to this:

```
func testExample() {
    let app = XCUIApplication()
    app.textFields.element.tap()

    app.keys["t"].tap()
    app.keys["e"].tap()
    app.keys["s"].tap()
    app.keys["t"].tap()
    app.keyboards.buttons["Return"].tap()

    app.sliders["50%"].swipeRight()
    app.segmentedControls.buttons["Omega"].tap()
    app.buttons["Blue"].tap()
    app.alerts["Blue"].buttons["OK"].tap()
}
```

That does all the work we want, but it’s significantly less code, and for bonus points actually compiles.

**Note:** Just like all the other tests we’ve written, user interface tests are stored inside a subclass of **XCTestCase**, and use method names that start with “test”, take no parameters, and return nothing.

## Finding elements to test

XCUITest is built around the concept of *queries*: you navigate through your UI using calls that get evaluated at runtime. You can navigate in a variety of ways, which is why both **app.buttons["Omega"].tap()** and **app.segmentedControls.buttons["Omega"].tap()** are valid – as long as XCUITest finds a button with the title "Omega" somewhere then it's happy.

If you noticed, there are two ways of accessing a specific element:

```
app.textFields.element.tap()
app.buttons["Blue"].tap()
```

The first option is used when the query – **textFields** – only matches one item. If our app had only one button, we could have used **app.buttons.element.tap()** to tap it.

The second option is used when you need to select a specific item. Using **app.buttons["Blue"]** effectively means "the button that has the title 'Blue'", but this approach is problematic as can be seen here:

```
app.sliders["50%"].swipeRight()
```

Sliders don't have titles, so Xcode identified it using its value. This was 50% to begin with, but that's a really confusing way to refer to user interface elements, so iOS gives us a better solution called *accessibility identifiers*. All UIKit interface objects can be given these text strings to identify them uniquely for user interface testing.

To try this out, open Main.storyboard and select the slider. Select the identity inspector, then enter "Completion" for its accessibility identifier.

**Note:** The accessibility identifier is designed for internal use only, unlike the other two accessibility text fields that are read to the user when Voiceover is activated. Xcode's user interface system actually scans the accessibility identifiers to navigate its way around our app, so make sure and attach useful identifiers to things!

With that change we can refer to our slider regardless of what value it might have, like this:

```
app.sliders["Completion"].swipeRight()
```

## Writing our own tests

So far we've used Xcode's event recorder, cleaned up the code it generated, and added an accessibility identifier to avoid problems when accessing the slider. However, we still don't have any *tests*, so let's write those now.

Take a copy of the **testExample()** method and call it **testLabelCopiesTextField()**, like this:

```
func testLabelCopiesTextField() {
    let app = XCUIApplication()
    app.textFields.element.tap()

    app.keys["t"].tap()
    app.keys["e"].tap()
    app.keys["s"].tap()
    app.keys["t"].tap()
    app.keyboards.buttons["Return"].tap()

    app.sliders["Completion"].swipeRight()
    app.segmentedControls.buttons["Omega"].tap()
    app.buttons["Blue"].tap()
    app.alerts["Blue"].buttons["OK"].tap()
}
```

Now delete its last four lines, but make some space where we can write a test:

```
func testLabelCopiesTextField() {
    let app = XCUIApplication()
    app.textFields.element.tap()
```

```
    app.keys["t"].tap()
    app.keys["e"].tap()
    app.keys["s"].tap()
    app.keys["t"].tap()
    app.keyboards.buttons["Return"].tap()

    // test goes here
}
```

As for the test itself, this uses the same XCTest assertion functions we've been using in our unit tests. In this case we want to check whether the label contains the text "test", which is done like this:

```
XCTAssertEqual(app.staticTexts.element.label, "test")
```

Put that directly below the // **test goes here** comment. You might be wondering why we must use **staticTexts** rather than **labels**, but keep in mind XCUITest is cross-platform – AppKit on macOS blurs the lines between text fields and labels, so using something generic like "staticTexts" makes sense on both platforms.

You can try that test now if you want – if you press Cmd+B to build your code you should see an empty gray diamond to the left of **func testLabelCopiesTextField()**, and clicking that will run the test.

However, it's likely the test will fail because Xcode can find more than one label on the screen – the clock, for example, counts as a label, so when we **app.staticTexts.element** XCTest doesn't known which one we mean. To fix that, open Main.storyboard, select our label, and use the identity inspector to change the accessibility identifier to "TextCopy". Remember, this is only used internally in apps so users never see it, but it lets us reference specific elements in a user interface.

With that change, we can now write a passing test:

```
XCTAssertEqual(app.staticTexts["TextCopy"].label, "test")
```

Next we're going to test the slider and progress view, because moving the slider to the right should move the progress view to the left. Copy the **app.sliders["Completion"].swipeRight()** line into its own test called **testSliderControlsProgress()**, like this:

```
func testSliderControlsProgress() {
    app.sliders["Completion"].swipeRight()
}
```

You'll need to copy the definition for **app** too, making this:

```
func testSliderControlsProgress() {
    let app = XCUIApplication()
    app.sliders["Completion"].swipeRight()
}
```

The **swipeRight()** method call was generated by Xcode, but it's really not fit for purpose here because it doesn't mention how far the test should swipe. To fix this we need to replace **swipeRight()** with a call to **adjust(toNormalizedSliderPosition:)**. As the name suggests, this takes normalized slider positions, meaning that you refer to the leading edge as 0 and the trailing edge as 1 even if your slider counts from 0 to 100.

Here's how that looks in code:

```
func testSliderControlsProgress() {
    let app = XCUIApplication()
    app.sliders["Completion"].adjust(toNormalizedSliderPosition:
1)
}
```

Now for the complicated part: writing a test. This is complicated by three things:

1. You won't find a **progressViews** property inside **app**. Instead, you must use **progressIndicators**.
2. Once you find your progress view, its value is stored as **Any?** – anything at all, or perhaps nothing.
3. In this case, the actual value is stored as a string with "%" on the end. This means we should be checking for "0%", because our slider was dragged all the way to the right.
4. We need to typecast the **Any?** to a string, and if that fails for some reason we'll call **XCTFail()** then end the test.

Here's how the test should look:

```
func testSliderControlsProgress() {
    let app = XCUIApplication()
    app.sliders["Completion"].adjust(toNormalizedSliderPosition:
1)

    guard let completion = app.progressIndicators.element.value
as? String else {
        XCTFail("Unable to find the progress indicator.")
        return
    }

    XCTAssertEqual(completion, "0%")
}
```

For our last test we're going to make sure that tapping colors shows an alert. Start by copying **app.buttons["Blue"].tap()** and **app.alerts["Blue"].buttons["OK"].tap()** into their own **testButtonsShowAlerts()** test, then app the usual **let app = XCUIApplication()** line to the top. It should look like this:

```
func testButtonsShowAlerts() {
    let app = XCUIApplication()
    app.buttons["Blue"].tap()
```

```
    app.alerts["Blue"].buttons["OK"].tap()
}
```

Between the second and third lines of that method we need to write a new test: does an alert exist with the title "Blue"? XCUITest has a dedicated **exists** property for this purpose, so we can test for the alert in just one line of code:

```
func testButtonsShowAlerts() {
    let app = XCUIApplication()
    app.buttons["Blue"].tap()
    XCTAssertTrue(app.alerts["Blue"].exists)
    app.alerts["Blue"].buttons["OK"].tap()
}
```

**Note:** As well as **exists**, you might also want to try using **isHittable** to check whether something can actually be pressed – just existing is often not enough!

At this point we've refactored almost all the code from **testExample()** into individual tests – see if you can add one more to test the segmented control, using what you've learned so far. When you're done, you can delete the original **testExample()** entirely, because it's no longer needed.

That's all our code tested, so we can now run all the tests at the same time. To make that happen, scroll up to the top of XCUITestSandboxUITests and click to the left of **class XCUITestSandboxUITests** – it might be a green checkmark or a red cross depending on what state your tests are in.

As Xcode runs all the tests you'll see it start and stop the app multiple times, ensuring that your code runs from a clean state each time. If everything has gone to plan, once the test runs finish you should see a green checkmark next to your class.

# Working with queries

Now that you have a rough idea of how user interface testing works, let's take a look at some more advanced functionality to help you write better, faster tests.

## What can you check?

Using Xcode's test recorder system can help you get started, but you should be prepared to do a fair amount of hand editing of the "code" it generates – and I'm putting code in quotes for good reason, as you've seen.

Once you get the hang of how UI tests look and work you'll be able to write them by hand. So, let's kick off with a quick breakdown of the various functionality we have at our disposal.

First, tapping on things:

- Call **tap()** to tap directly on something. If you call this on a text field it will begin editing, but if you call this on a button its action will be triggered.
- Call **doubleTap()** to tap twice on something in quick succession.
- Use **twoFingerTap()** to tap once with two fingers on something.
- For real flexibility, use **tap(withNumberOfTaps:, numberOfTouches:)** – it lets you specify a finger count as well as how many taps to perform.
- There's also **press(forDuration:)** for triggering long press gesture recognizers.

As for complex gestures, you can choose from the following:

- There's **swipeLeft()**, **swipeRight()**, **swipeUp()**, and **swipeDown()** for swipe directions. None of these work with any sort of distance.
- You can use **pinch(withScale:, velocity:)** to pinch and zoom on something like a map. Specify a scale between 0 and 1 to zoom out, or a scale greater than 1 to zoom in. Velocity is specified as "scale factor per second"; send in zero to make your pinch precise.
- Use **rotate(_: withVelocity:)** to rotate around an element. The first parameter is a **CGFloat** specifying an angle in radians, and the second is specified as radians per second.

There are two more element-specific methods you'll want to use:

- For pickers, use **adjust(toPickerWheelValue: String)** to make a picker scroll through to select a particular value.
- For text boxes, use **typeText()** to insert some text quickly. Xcode's previous recording typed each letter individually, which is significantly slower than typing whole strings.

## Waiting for existence

The above commands give us a working vocabulary that can tap, swipe, pinch, and rotate our way around an app. Plus, in the previous walkthrough we briefly used the **exists** property to check that a view exists as part of our test. Put together, this means we can navigate around an app, come to a particular screen, then check that everything is configured correctly.

However, the nature of user interfaces is that they are fluid compared to the nice, basic truths of the kind of model data you'd use with unit tests. It's common to create views and move them around dynamically before destroying them again, so it's important to give your user interface tests a little bit of flexibility to help make them less fragile.

One simple thing you can do is exchange the **exists** check for a call to **waitForExistence(timeout:)**. This is similar to waiting for expectations in unit tests, except it's designed to return true as soon as it finds a specific UI element in your app – or return false if the timeout is reached without the element being created.

For example, here's how we checked whether an alert with the title "Blue" exists:

```
XCTAssertTrue(app.alerts["Blue"].exists)
```

Although that works fine here, other times you're likely to want to add that little bit of flexibility by using **waitForExistence()**, like this:

```
XCTAssertTrue(app.alerts["Blue"].waitForExistence(timeout: 1))
```

A one-second timeout won't do much to slow down your tests, but will help make your user interface tests that little bit more reliable.

## Finding what you're looking for

XCTest gives us six ways of looking up specific elements:

- The **children(matching:)** method looks through all the direct children of a specific element, trying to find something.
- The **descendants(matching:)** method looks through all children, grandchildren, great-grandchildren, and more, of a specific element, trying to find something. Obviously this is more intense than using **children(matching:)**, because it might have to search your whole user interface.
- There are properties that group together specific types of elements. We've seen **app.buttons**, **app.alerts**, and more.
- You can read various dictionaries to look for strings that appear in your user interface, such as **app.buttons["Blue"]**.
- If you append **.element** to a query it means "there is only one of these" – you can manipulate it as a single result, rather than a collection of results. If you use **.element(boundBy:)** it lets you pick a specific element from a collection.
- If you append **.firstMatch** to a query it means "use the first one you find, and let me manipulate it as a single result."

Xcode literally needs to scan your user interface to figure out what's available and what can be pressed. The code we write in UI tests is a series of instructions telling Xcode how to scan the user interface and what to look for, so the more precise you can be the better.

In particular, this means using **children(matching:)** rather than **descendants(matching:)** where possible, and using **firstMatch** as often as you can – Xcode will return your element as soon as finds any matching item, rather than continuing its search across all the user interface.

**Warning:** If you use **element** with a regular query that matches multiple items, XCTest will throw an error warning you that your test isn't unique. If you use **firstMatch** instead, XCTest

*won't* throw an error because it stopped looking for more elements as soon as it found the first – your test still isn't unique, but now you aren't being warned about it!

# Screenshots and attachments

User interface testing is naturally more flaky than unit tests, partly because you're testing something out of your control – how a virtual computer navigates around a virtual iPhone – but also because you're testing whole swathes of functionality at once. If you change even one small thing in your code it could break the way your app works, in turn breaking many if not all of your tests.

To help you narrow down problems, Xcode is able to take screenshots as it works. If your tests succeeds – if everything went to plan – those screenshot will be deleted because you probably won't want to do any debugging. But if the tests *fails* then Xcode will keep the screenshots to help you step through visually and figure out what went wrong.

To see this in action, let's try a failing test:

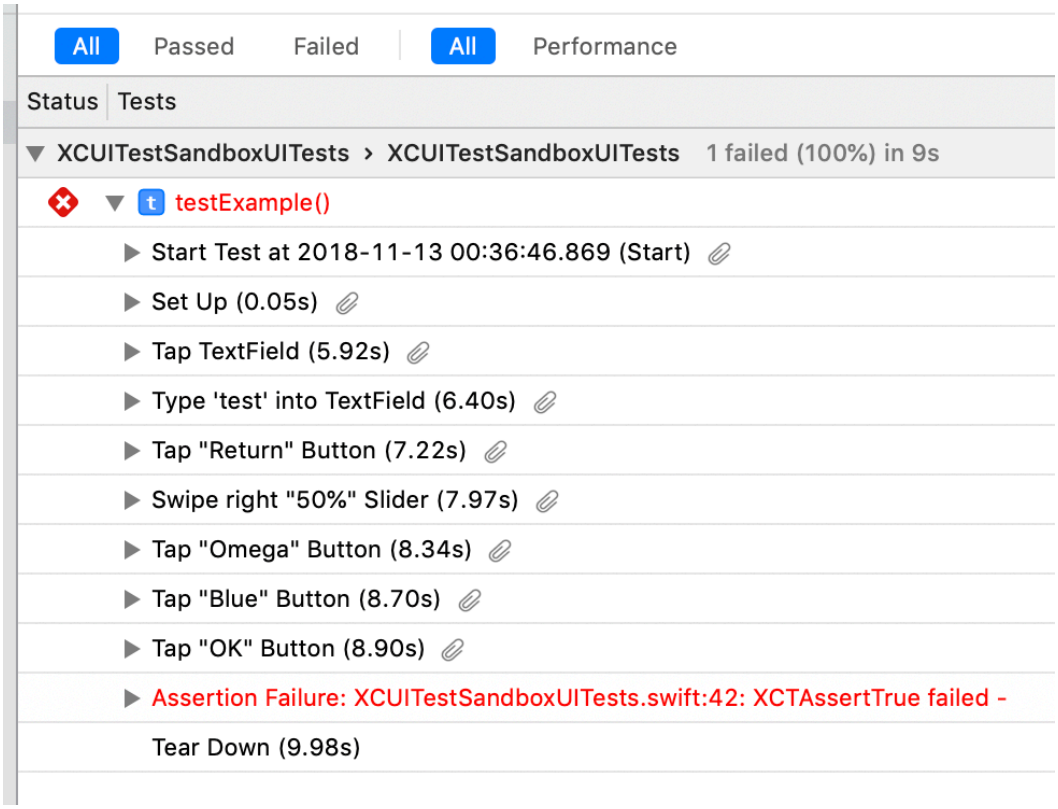
```
func testExample() {
    let app = XCUIApplication()
    app.textFields.element.tap()
    app.textFields.element.typeText("test")
    app.keyboards.buttons["Return"].tap()

    app.sliders["50%"].swipeRight()
    app.segmentedControls.buttons["Omega"].tap()
    app.buttons["Blue"].tap()
    app.alerts["Blue"].buttons["OK"].tap()

    XCTAssertTrue(app.alerts["Blue"].exists)
}
```

That adds an assertion to the end of our test saying that the "Blue" alert must exist even after its OK button has been tapped to dismiss it. This will fail, as you might expect, but that's a good thing because it gives you a chance to look over Xcode's screenshots.

First, right-click on the red and white cross next to **testExample()**, which is Xcode telling you that test failed. Now choose Jump To Report, and you should see "testExample" in red. If you open up its disclosure indicator you'll see all the steps that Xcode took, with the assertion failure near the end.

Each one of those steps should have a small paperclip icon next to it, which is Xcode's way of saying there's some sort of attachment in that step. Try going into "Tap OK Button" then "Synthesize Event". You should see "Automatic Screenshot" in there, with a quick look eye icon next to it – tap that eye to see the screenshot Xcode took as it was about to synthesize tapping the OK button.

As you can see just by looking at all the paperclip icons, there are a *lot* of automatic screenshots being taken even for trivial tests – it's no wonder that Xcode deletes them for passing tests by default.




If you want, you can take screenshots by hand when something important has happened. This

has the advantage that you can name your screenshot something useful – "About to click on the button" or "Showing authentication screen", for example – and you can configure it to remain even when the tests have passed.

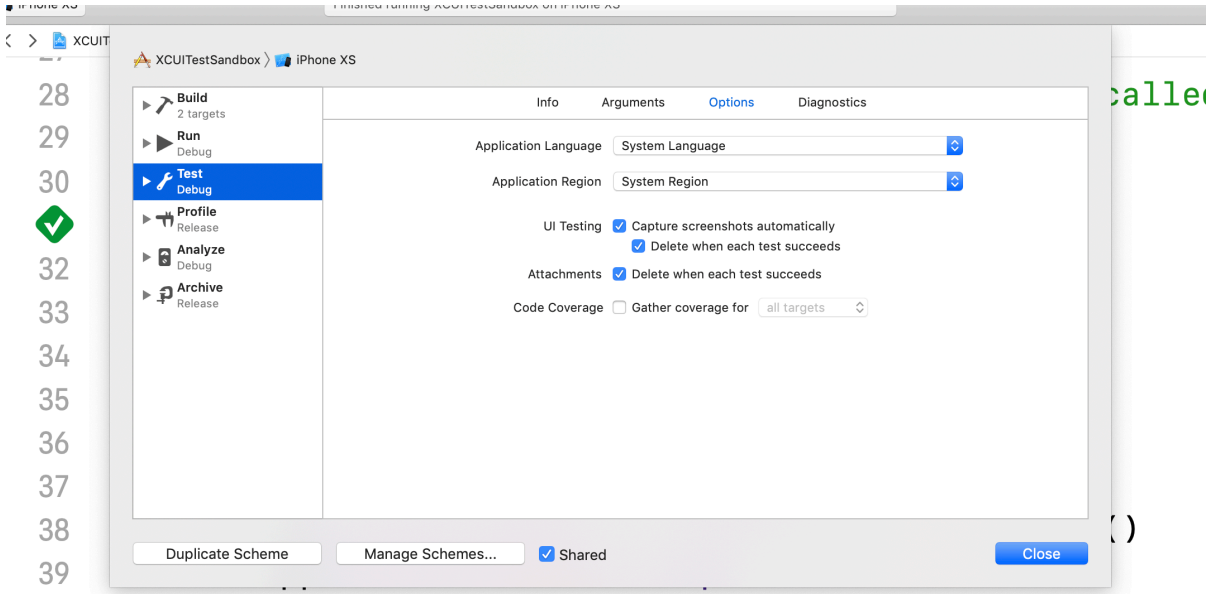
To create a screenshot, just call **screenshot()** on any element. That might be a single control in your app, or it might be the whole app itself, in which case you'll get the whole screen. Once you have that, wrap it in an instance of **XCTAttachment**, optionally add a name and lifespan, then call **add()** to add it to your test run.

For example:

```
let screenshot = app.screenshot()
let attachment = XCTAttachment(screenshot: screenshot)
attachment.name = "My Great Screenshot"
attachment.lifetime = .keepAlways
add(attachment)
```

**XCTAttachment** is good for storing things other than screenshots: it has convenience initializers for any **Codable** and **NSCoding** objects, or you can just write a **Data** instance into there.

**Tip:** When your screenshots and attachments are set to automatically be deleted on passing tests, there's no reason not to add attachments regularly – take screenshots if it helps, and name them something sensible so you can find them easily. You can change the lifetime policy for your project by going to the Product menu and choosing Scheme > Edit Scheme. Now choose Test from the list of schemes, and select the Options tab. Finally, adjust the UI Testing and Attachments screenshot until you're happy. Be warned: saving lots of screenshots can be resource intensive, particularly when tests run often. Don't save too much!



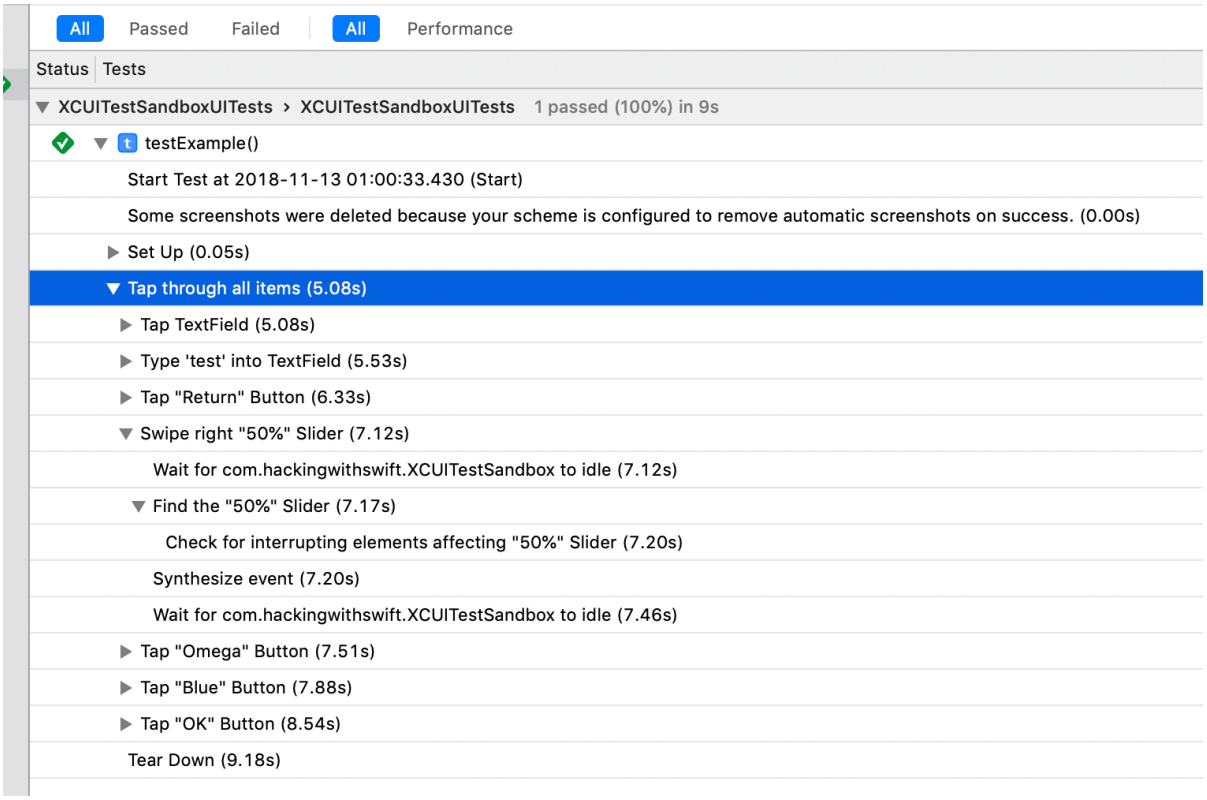

```
28
29
30
32
33
34
35
36
37
38
39
40          app.alerts["Blue"].buttons["OK"].tap()
```

# Tips and tricks

I've said it several times now, and this will be the last: user interface testing is flaky, so you need to be careful with it. Don't treat UI testing as the bottom layer in your pyramid, because that role is naturally taken by unit tests.

That being said, I have four tips that will help you write better, faster, and more stable unit tests.

First, you can create groups of interactions by using **XCTContext.runActivity(named:)**. This doesn't do anything other than allow you to logically group your instructions together: you might start an activity called "Tap through all items", where you tap all the correct controls to authenticate a user. All those individual actions will still be tracked, but now they'll be grouped under the "Tap through all items" heading so you can find them more easily.




Second, user interface testing allows you to toggle between apps if you like. This is primarily

going to be useful if you have multiple apps in the same group and want to check they work together – Gmail might want to launch Google Maps to see if a hotel reservation has appeared, for example.

To jump between apps, create instances of **XCUIApplication** using the **bundleIdentifier** initializer. You can then call **launch()** on the result to launch a new instance, or **activate()** to bring to the front an existing instance.

Third, some people prefer to disable animations when user interface tests are being run, because it removes a common source of delays. Each time XCTest simulates an action it has to wait for the UI to come to rest – to finish all its updates – before it can continue, so by disabling animations you're removing all the little pauses as things happen.

If you want to try this out, try using the command-line argument approach to detect "enable-testing", then call **UIView.setAnimationsEnabled()** with false to disable animations across the board:

```
if CommandLine.arguments.contains("enable-testing") {
    UIView.setAnimationsEnabled(false)
}
```

Finally, XCTest provides us with a specific way of dismissing user interface interruptions – a nice way of saying all those alerts that appear when we ask for things like location permission.

Using the **addUIInterruptionMonitor()** method we can automatically run some code whenever an interruption begins. This might be because we asked for permission for something, or because we showed an alert ourselves. Either way, we can pass a closure to run that can evaluate the interruption and take any action. If you successfully handled the interruption you should return true; if not return false, and other interruption monitors can be run – you can have as many as you want, and they are run in reverse order that you added them.

Here's some sample code to get you started:

```
addUIInterruptionMonitor(withDescription: "Automatically allow
location permissions") { alert in
    alert.buttons["OK"].tap()
    return true
}
```

# Chapter 5

# Test-Driven Development

# Why test first?

Too many companies suffer from what I call **Ready, Fire, Aim** – they have a great team and a great idea, but charge off writing production code without taking the time to plan a more careful approach. Put more simply, weeks of coding can save hours of testing.

Test-driven development (TDD) is designed to solve this problem by having us write tests before we write any code. In fact, you don't even get to write a whole test: you write just part of it, then part of your production code, then more test, then more production code, and so on – you write the two of them side-by-side, in tandem, so it's really easy to keep them both in your head while you're working.

This approach gives two important side effects.

First, you find and fix bugs straight away, while you're still writing the tests. Writing the test first means you can be clear about your expectations – "when I create A and send it B, I expect C back" – and if your code doesn't match that then you fix the code until it does. It's much, *much* easier to fix bugs like this as soon as you find them, rather than having months go by and the issue being reported when you have long forgotten how that piece of code worked.

Now, obviously writing tests as you go bears a cost because rather than spending all your time writing production code you're now dedicating a not-insubstantial amount of time to writing tests. However, think about it: if there are bugs in the code you write would you rather discover them now or when you're much closer to your deadline? And would you prefer to run your app repeatedly to check that everything works as you intended, or would you rather have tests that do it for you and verify everything is correct?

The second side effect of TDD is that it forces you to break up long tasks into a series of small, testable chunks. I don't know about you, but I feel much happier when I'm able to cross small tasks off a to do list – it helps me build up momentum because I can say "all that code is written and the tests pass," and feel confident that I'm building new code on top of a firm foundation.

Now, while that does wonders for morale (and certainly makes your burndown charts more

useful!) breaking up tasks like this also means you're naturally writing smaller, simpler units of code. That is, if you leave your tests to the end, you might find you've written code that's very hard to test: perhaps one method does five different things, perhaps you've leant heavily on hidden dependencies, or perhaps you've made view controllers do more heavy lifting than they ought to, for example.

When it comes to writing tests you'll realize the problem and have two choices: declare the code "untestable" and move, or try to pull it apart and rewrite it. You might even try to call this process "refactoring", but like I said earlier the difference between refactoring and rewriting is that you can only really call it refactoring when it's backed by tests. We don't have any tests here because we decided to write them after the code, so now we have a complicated rewrite ahead of us.

On the other hand, if you write tests alongside the code you're forced to write testable code at every step along the way. This means you solve one small problem and have it fully tested, then solve another one that's also fully tested, and so on – no more risk of ending up with huge untestable chunks.

Back in 2008, Microsoft and IBM ran a study in 2008 that followed various development teams who were trying out TDD – IBM's people were working on device drivers, and Microsoft's were working on Windows, MSN, and Visual Studio. What they found was that the "defect density" – a fancy term meaning how many bugs were found compared to total number of lines of code – decreased by between 40% and 90% compared to other projects that were being run without TDD. That bears repeating: the *lowest* improvement they saw was a 40% lower defect density.

On the flip side, those same teams estimated that were they spending 15-35% more time writing their code, because of the time costs of learning and adapting to TDD, but we can do a little bit of napkin mathematics: if you could reduce the number of bugs in your code by between 40-90%, would you accept a 15-35% increase in development time?

If your answer is "yes!" then TDD is for you. If your answer is "no!" then TDD is also for you – you just don't know it yet.

## The laws of TDD

Robert C. Martin describes test-driven development using three fundamental laws:

1. You are not allowed to write any production code unless it is to make a failing unit test pass.
2. You are not allowed to write any more of a unit test than is sufficient to fail, and compilation failures are failures.
3. You are not allowed to write any more production code than is sufficient to pass the one failing unit test.

Those laws might seem simple enough, but they cause a number of fundamental shifts in development if you follow them closely:

- We stop writing a test as soon as it fails, which it will almost immediately. For example, testing that a user can buy a product will fail because we don't have a **User** struct.
- We stop writing production code as soon as our test passes, which in the previous example it will as soon as we create an empty **User** struct.
- This means we'll bounce between test code and production code repeatedly: try to use a non-existent **User** struct, create the empty **User** struct, try to to use a non-existing **Product** struct, create an empty **Product** struct, try to call a non-existent method on the **User** struct, write the method on the **User** struct, and so on.
- You must *not* skip ahead and write the complete test. As soon as your test no longer compiles, you stop.
- Similarly, you don't carry on writing production code as soon as your test passes. Any code written once outside of your tests is, by its very definition, untested, and we don't want that, do we?

One acronym closely related to TDD is YAGNI: "You Ain't Gonna Need It." This is another Extreme Programming principle, but it's one I apply every day to every project I work on: there's no point trying to predict what functionality you might need in the future, because there's a high chance you'll be wrong.

So, don't try writing things outside of your tests: not only are they not tested (boo!), but there's a good chance you won't end up needing it.

## Test-driven Swift

In the next chapters we're going to look at how TDD works in practice, and apply it to Swift code. However, there's one warning I need to say up front: Swift does not have a reputation for particularly fast compile times, which is a pretty big speed bump in the road towards TDD.

Now, to be fair to Swift things have gotten hugely better since it launched, and work is constantly underway to make compilation faster and faster. However, even slightly slow compilation times can be annoying when you're using TDD: you want to run your tests all the time, and you want quick feedback to tell you whether you can move onto the next step. So, if you're constantly getting five-second delays it can be frustrating.

One mitigation here is that most of us are working in the iOS simulator, and even without Swift's slow compile time there's a fix cost to copying our app to the simulator and launching it. Unless you're writing deeply intertwined code, all being well the time spend compiling your Swift should still be pretty small compared to the fixed cost of launching your app in the simulator.

## A note on TDD zealotry

Although I believe TDD can do wonders for improving your workflow, it's not my job to convince you to use it. I'm going to show you how it works, and I'll give you practical examples of TDD using iOS and Swift, but I'm leaving it up to you to decide whether it works for you and your team.

One thing I ask, though, is that you give it a meaningful try. It doesn't have to be for a whole project, or even for a whole sprint, but maybe just try it out for three days and see how you get on. As long as you give it a fair try during that time, you'll be in a good position to judge how well it fits with you and your colleagues.

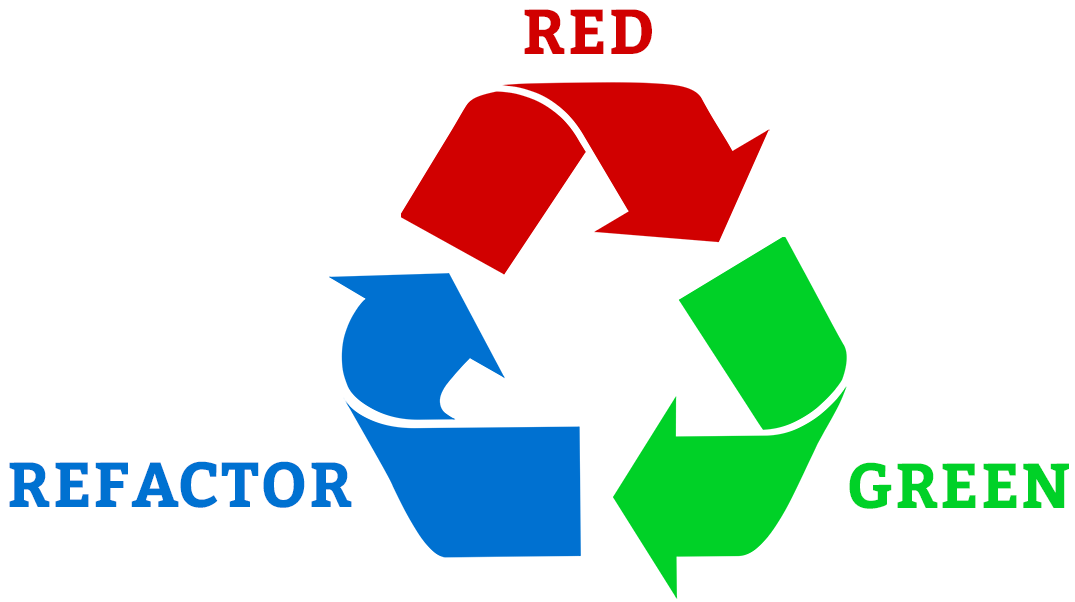
# The basics of TDD

I've already explained what test-driven development is along with the benefits it can bring. Now I want to turn to the details: *how* it works.

At the wider scale, TDD works by having us write small features for our code at the same time as we write tests. However, if we zoom in a little you'll see that TDD breaks down into smaller steps usually called Red-Green-Refactor:

- **Red:** Create some sort of test that fails, even if it's only a compilation failure because your production code is missing the required functionality.
- **Green:** Write just enough of your production code required to make your test pass.
- **Refactor:** Take a closer look at your code: can you make it faster or more efficient? Can you remove duplication? This is your chance to fix it up.

The "red" and "green" names come from the colors most development environments assign to failing and passing tests – some apps even have a colored bar that's visible at all times, leading to Gerard Meszaros's saying, "keep the bar green to keep the code clean.




Even though TDD breaks down to just three steps, folks hit so many problems and misunderstandings with them. In particular, there are five problems folks normally hit:

1. They forget about the refactor step, and instead iterate tightly around a Red-Green loop. While this definitely allows you to make seemingly good progress, you're also at extreme risk of writing messy code that will cause you pain the long-term. There are three steps in TDD, and you forget the last one at your peril.
2. You jump around the three laws of TDD continuously while you're working inside the phases of Red-Green-Refactor. You might go through Red-Green-Refactor once for a single unit test, but inside that single test you might have bounced around the three laws ten times.
3. Refactoring doesn't happen until you've been through the green phase – all tests passing – and by the time refactoring completes your tests should still all be passing. Jon Reid calls this "start in green, end in green."
4. They try to apply Red-Green-Refactor to large components of code, which will just cause confusion – trying to refactor 1000 lines at once is a recipe for problems, even with good tests. Instead, break your tasks down into small chunks and tackle *those* with the Red-Green-Refactor approach.
5. Red-Green-Refactor isn't the only thing we do to design applications. Before you even start writing tests it's a good idea to have an idea of the bigger picture: how the overall thing is going to fit together. Without this you're likely to blunder forward somewhat blindly and burn through a lot of time.

So, TDD has three laws, and a three-step working process, but in practice it takes a little more thought!

## A practical example

OK, enough talk: let's put TDD into action with a small, practical example. In the next chapter we'll be going through a longer example with real iOS code, but at this point let's work with something small so you can really see the fundamentals of TDD in action.

Start by creating a new iOS project using the Single View App template. Name it TDD, make sure you have Include Unit Tests checked, and save it to your desktop.

We're going to write some code that tests a user buying a book, which will involve creating a user, buying a book, then checking that the book is in the user's library afterwards.

As you might imagine, this means starting with a test, so please open TDDTests.swift in the TDDTests group, and give it this initial code:

```
func testReadingBookAddsToLibrary() {

}
```

We're going to start by setting up our "given" code, like this:

```
// given
let bookToBuy = "Great Expectations"
let user = User()
```

Creating the simple **bookToBuy** string is fine, but creating an instance of **User** will immediately cause Xcode to flag up an error: **Use of unresolved identifier 'User'.**

At this point the second law of TDD kicks in: **you are not allowed to write any more of a unit test than is sufficient to fail, and compilation failures are failures.**

So, let's pause writing the test for now, and instead create our **User**. To do that, press Cmd+N to create a new file, choose Swift File, then name it User.swift. **Make sure you place this inside your TDD group and not the TDDTests group.**

Add this code to User.swift:

```
struct User {

}
```

Now for the important part: what code should we put inside that **User** struct? Well, we know it needs to buy books, so maybe let's start with a **buy()** method tha— what's that? Oops! We

almost just broke the third law of TDD: *you are not allowed to write any more production code than is sufficient to pass the one failing unit test.*

By adding a **User** struct our test now passes, so trying to add any more production code is a bad idea. Instead, we should return to writing more of the test until it fails again, so go back to TDDTests.swift and let's add the *when* phase of our test:

```
// when
user.buy(bookToBuy)
```

That causes our test to fail, again: it won't compile because we don't have a method called **buy()**.

So, head back to User.swift and we'll add the method:

```
struct User {
    func buy(_ product: String) {

    }
}
```

Remember, we're not going to fill in that method at all – we don't write any more code than is required to make the test pass in its current form.

The last step for our test is to add an assertion to the end: we've set up our user, performed some action, and now want to check that the user has "Great Expectations" in their library.

There are a number of ways you could code this, but we're going to take a neatly encapsulated option:

```
// then
XCTAssertTrue(user.owns(bookToBuy))
```

To make that compile we need to add an **owns()** method to our **User** struct, that accepts a

string and returns a boolean. Now, we could just cheat here and make it return **true** every time, but we're not actually out to *cheat* – we want to write the simplest possible solution that actually works as intended.

So, first add this property to the **User** struct, so that we can track which products the user owns:

```
var products = [String]()
```

Now we can implement the **owns()** method like this:

```
func owns(_ product: String) -> Bool {
    return products.contains(product)
}
```

That allows our test code to compile, but it won't actually pass because the **buy()** method is still empty. So to finish up we need to implement the **buy()** method properly:

```
mutating func buy(_ product: String) {
    products.append(product)
}
```

That in turn will cause our test code to stop compiling, because we've declared **user** as a constant and we're now trying to mutate it. We need one last change, this time in our test code:

```
var user = User()
```

With that our test code compiles, runs, and passes successfully, so we have one complete test. That's just the first of many tests you would write in this program, but I think it shows you how rapidly we need to move between the TDD laws.

We are now in the green state for our program, because all our tests pass. This means we should be looking for opportunities to refactor: how can we make our code more efficient, or remove duplication? Well, in this case there isn't any duplication we can get rid of, but there *is*

a place we could make our code faster.

Right now we're using an array for products, which is fine for small instances but if we have a library of hundreds or even thousands of books that **contains()** call will be *slow*. A better idea is to change that property to a set, like this:

```
var products = Set<String>()
```

We then need to use **insert()** rather than **append()**, because sets don't have the concept of ordering:

```
mutating func buy(_ product: String) {
    products.insert(product)
}
```

Make that change, then run the test again. We started in green and ended in green, which is exactly what refactoring ought to look like – good job!

## Three Xcode tips

With TDD having three laws and a three-step process, I feel it's only right to give you three Xcode tips to help make your life easier when using TDD.

First, configuring your Xcode layout correctly will save you a lot of hassle when writing tests. More specifically, you should consider activating the assistant editor and configuring it so you have your test on the left and the matching code unit on the right. This allows you to see them both at the same time, and to make quick edits between the two. Failing that, at least try using the Ctrl+Cmd+Left and Ctrl+Cmd+Right keyboard shortcuts to move between files.

Second, the Ctrl+Alt+Cmd+G shortcut is *extremely* helpful when using TDD, because it means "run the previous test again." It's helpful in general when you're writing tests, but it's *doubly* helpful when you're continuously running the same test.

Finally, Xcode has an automatic breakpoint that triggers whenever a test fails, and will point

out exactly where it happened. This is really useful a lot of the time, but when you're writing TDD is actually counter-productive: we write tests *knowing* they are going to fail, so having Xcode come in and take over isn't helpful. So, as useful as this breakpoint might be elsewhere, you should turn it while writing tests.

## Focusing on tests

When you're just starting out, the biggest problem you'll face is trying to write testable code at every step of the way. Don't worry: that's normal. Flipping back and forth between production and test code will stop feeling like a bump in the road after you've written a few tests, and will instead start to feel fluid. As you get better, you'll almost develop a spider sense for when you're writing code that can't be tested, or for when you start straying away from the laws.

If everything goes to plan, you'll start to discover another important side benefit of TDD: test-first development is effectively *requirements*-first TDD, because you start by outlining what you expect to happen when your code is run: "when I create A and send it B, I expect C back." This approach forces you to keep the end-game in sight at all times: you can't "sort of" know what should happen, because unless you understand the complete requirements you can't begin to write a test for them.

Finally, I should add that even though it does many things brilliantly, Xcode is far from the state of the art when it comes to supporting test-driven development. Alternatives such as AppCode are specifically designed to support the TDD workflow: it could have generated our **User** struct, generated the **buy()** method stub, and so on – it recognizes that they don't exist as we're writing the test, and will do the job for us.

So, if you find TDD works well for you then you should at least consider a trial version of AppCode: go to **https://www.jetbrains.com/objc** and see what you think!

# A test-driven case study

We've just walked through the principles of TDD, but you're still a long way from being a test-driven developer. You see, writing one, two, or even ten tests in isolation is a great start, but it doesn't really map to writing full apps with TDD – building view controllers, writing data sources, and so on.

While these use the same skills, the same laws, and the same red-green-refactor circle, the *context* is completely different – once we're transplanted from simple scenarios to view controllers, it's natural for the mind to revert a little and switch back to the old ways of working, particularly when you start butting heads with the practicalities of UIKit development.

So, in this chapter we're going to recreate the first project from my original Hacking with Swift series, except this time we'll try doing it using test-driven development. If you haven't read that book, the first project is called Storm Viewer: it lets users browse through a selection of storm images, then tap to see one.

In so far as it's possible, I'm going to try to keep the structure of this code the same as the Hacking with Swift original, so it's easier to compare the two. As we progress you'll start to see how this causes cracks to show in the original app design, but I hope it will cause you to think a little more closely in your own code!

## Step 1: Loading data

Create a new iOS project using the Single View App template, naming it Project1 and making sure Include Unit Tests is checked but nothing else is.

I've included the assets for this project in the source code for this book, in the project1-files directory – please drag the Content folder that's inside there into your project navigator, choosing Create Groups. These pictures will be used inside the app, so please make sure they are inside the app group and not the tests group.

Open Project1Tests.swift, then go ahead and delete all the placeholder methods so all you have

left is this:

```
import XCTest
@testable import Project1

class Project1Tests: XCTestCase {

}
```

OK! Let's start writing some tests.

First, let's test that our view controller is able to load the ten pictures we just added. In the original project, these files were loaded straight from the app bundle, so we're going to follow that approach here. Of course, we *aren't* going to write any code until we first have a failing test, so let's start with that:

```
func testLoadingImages() {
    // given
    let sut = ViewController()

    // when
    sut.loadViewIfNeeded()

    // then
    XCTAssertEqual(sut.pictures.count, 10, "There should be ten
pictures.")
}
```

The first line creates a new **ViewController** instance, the second makes it load fully, and the third checks that it has correctly found all 10 pictures. However, the third one won't compile, because we haven't given our view controller a **pictures** property.

So, let's pause there and write just enough code to continue. In this case, that means opening

ViewController.swift (now would be a good time to activate the assistant editor!) and giving it this property:

```
var pictures = [String]()
```

That will make our test code compile, but the test will still fail because we're looking for 10 pictures and the view controller's array has none.

To fix this, we need to implement the **viewDidLoad()** method so that it correctly looks through all the "nssl" pictures in the app bundle, loading each one into the **pictures** array. There are a number of ways of doing this, but I'm just going to take the original Hacking with Swift code:

```
override func viewDidLoad() {
    super.viewDidLoad()

    let fm = FileManager.default
    let path = Bundle.main.resourcePath!
    let items = try! fm.contentsOfDirectory(atPath: path)

    for item in items {
        if item.hasPrefix("nssl") {
            pictures.append(item)
        }
    }
}
```

## Step 2: Adding a table view

Our flat white view controller needs to be transformed into a table if this app is going to be useful. So, let's write a test that identifies a problem:

```
func testTableExists() {
    // given
```

```
    let sut = ViewController()

    // when
    sut.loadViewIfNeeded()

    // then
    XCTAssertNotNil(sut.tableView)
}
```

That creates an instance of our view controller, forces it to load its view for real, then asserts that its table view property isn't nil. Once again, that last line won't compile because we're making an assumption that isn't currently true: our view controller doesn't have a **tableView** property.

This happens because our view controller is currently a direct subclass of **UIViewController**, when really it needs to come from a **UITableViewController**. So, open ViewController.swift and change it to this:

```
class ViewController: UITableViewController {
```

Now the test will compile, but it won't work correctly: if you try running it, the app will crash because our storyboard expects a view controller but our code expects a table view controller.

To fix that we need to modify the storyboard a little: open Main.storyboard, delete the current view controller scene, and replace it with a new table view controller. Using the identity inspector, change the class of the new table view controller to "ViewController", then make sure you check the "Is Initial View Controller" box to put things back as they were.

Now press Cmd+U to run our tests again, and this time they are working – progress!

## Step 3: Counting table rows

We've loaded our picture data and added a table view, so now the next step is to verify that our

table is showing the correct number of rows. In this app there will be one row for each picture that was loaded, so we should be able to query the table view and check that its row count is equal to **pictures.count**.

The easiest way to do this is to talk to the table view directly using the same methods that **UITableView** uses: **numberOfRowsInSection**, in this instance.

So, go ahead and add this test:

```
func testTableViewHasCorrectRowCount() {
    // given
    let sut = ViewController()

    // when
    sut.loadViewIfNeeded()

    // then
    let rowCount = sut.tableView(sut.tableView,
numberOfRowsInSection: 0)
    XCTAssertEqual(rowCount, sut.pictures.count)
}
```

If everything is working correctly, **rowCount** and **pictures.count** should be equal, but if you run the test you'll find they aren't – **pictures.count** is 10 to reflect all the pictures we loaded, but **rowCount** is 0 because we haven't actually done the work of loading table view cells yet.

Go back to ViewController.swift and give it this method:

```
override func tableView(_ tableView: UITableView,
numberOfRowsInSection section: Int) -> Int {
    return pictures.count
}
```

That *ought* to be enough for the test to build and pass, but if you try it now you'll see one of

UIKit's many fun quirks: now that we've told it there are some rows in the table, it expects us to respond to **cellForRowAt** with meaningful data.

We *could* pause here to implement the method fully, but realistically all we want to do is just enough work to make the current test pass – we'll look at other functionality in a moment.

So, the least we need to do is implement **cellForRowAt** so that it returns some sort of table view cell:

```
override func tableView(_ tableView: UITableView, cellForRowAt
indexPath: IndexPath) -> UITableViewCell {
    return UITableViewCell()
}
```

With that change our current test passes, and we can move on.

## Step 4: Do the cells have the correct text?

Returning an empty table view cell helped our previous test pass, but it's clearly not a long-term solution. Instead, we need to make sure we're loading the correct data into each cell, which in this case means the name of a storm filename from the **pictures** array.

There are three ways we could tackle this:

1. Write ten tests, with each one checking a different cell in the table view is correct.
2. Write one or two tests, checking some specific rows in the table view.
3. Write one test, using a loop to check *all* table views.

Like I've said in the past, it's often tricky to have multiple assertions in a single test, but in this case I don't think it's a problem because even with the loop we still only have one assertion being made.

So, this test will go through all cells in our table view to make sure they have the correct text:

```
func testEachCellHasTheCorrectText() {
    // given
    let sut = ViewController()

    // when
    sut.loadViewIfNeeded()

    // then
    for (index, picture) in sut.pictures.enumerated() {
        let indexPath = IndexPath(item: index, section: 0)
        let cell = sut.tableView(sut.tableView, cellForRowAt:
indexPath)
        XCTAssertEqual(cell.textLabel?.text, picture)
    }
}
```

That test will fail because our current view controller just returns empty table view cells, so let's write just enough code to fix it.

First, we need to register table view cell with a reuse identifier, so place this into **viewDidLoad()**:

```
tableView.register(UITableViewCell.self,
forCellReuseIdentifier: "Cell")
```

Second, we're going to implement **cellForRowAt** so that it creates one of those cells, configures it using the correct title, and sends it back:

```
override func tableView(_ tableView: UITableView, cellForRowAt
indexPath: IndexPath) -> UITableViewCell {
    let cell = tableView.dequeueReusableCell(withIdentifier:
"Cell", for: indexPath)
    cell.textLabel?.text = pictures[indexPath.row]
```

```
    return cell
}
```

That's it – only four lines of code, really, but now the test passes.

Now would be a good time to pause for refactoring. Our code works fine (we're in a green state), so it's important we stop to look for duplication otherwise our code might get messy.

In this instance we're calling **register()** on our table view to register a new table view cell identifier, but we also have an empty prototype cell in our storyboard – we don't need both. You're welcome to use whichever one you prefer but we *don't* need both, so in my case I removed the prototype cell from the storyboard.

## Step 5: Adding disclosure indicators

In this project, tapping one of the rows is going to show the selected picture in more detail. In iOS, this behavior is usually signaled to users by adding a light gray chevron to the right-hand edge of each table view cell, so let's write a test to look for that.

The code for this is almost the same as before: create a view controller, make it load, then loop over each cell and perform our check. This time we're going to check that the **accessoryType** property is set to **.disclosureIndicator**.

Add this test now:

```
func testCellsHaveDisclosureIndicators() {
    // given
    let sut = ViewController()

    // when
    sut.loadViewIfNeeded()

    // then
    for index in sut.pictures.indices {
```

```
        let indexPath = IndexPath(item: index, section: 0)
        let cell = sut.tableView(sut.tableView, cellForRowAt:
indexPath)
        XCTAssertEqual(cell.accessoryType, .disclosureIndicator)
    }
}
```

That test will fail because we haven't given our table view cells any accessory type. To fix that, go back to the **cellForRowAt** method and add this line before **return cell**:

```
cell.accessoryType = .disclosureIndicator
```

## Step 6: Finishing up with this view controller

I'm not a big fan of using TDD to verify design, because to me that feels better suited to UI testing. However, we're here so we might as well have a go, so we'll write two tests: one to verify that it has large titles enabled, and one to verify that it has the title "Storm Viewer". Spoiler alert: the first of these might sound easy, but you'd be surprised!

We can write the first test like this:

```
func testViewControllerUsesLargeTitles() {
    // given
    let sut = ViewController()

    // when
    sut.loadViewIfNeeded()

    // then

XCTAssertTrue(sut.navigationController?.navigationBar.prefersLa
rgeTitles ?? false)
}
```

That attempts to read the **prefersLargeTitles** property if it can be found, otherwise it uses false. If you run that test you'll see it fails because we haven't provided any sort of value for that property.

So, open ViewController.swift and add this to **viewDidLoad()**:

```
navigationController?.navigationBar.prefersLargeTitles = true
```

That sets **prefersLargeTitles** to true so that our test should pass. I say "should" because of course it still won't: even though we've attempted to change the property, in reality nothing happened because our view controller isn't actually inside a navigation controller.

To fix *that* we need to open Main.storyboard, select the table view controller, then go to the Editor menu and chose Embed In > Navigation Controller.

If you run the test again you'll see it *still* fails, and it's *still* because there's no navigation controller to work with even though we tried embedding it in one.

The problem is a direct result of the way we're creating the view controller: we're doing it by instantiating our own instance of **ViewController** rather than reading the storyboard.

This is where TDD and UIKit can get a little hairy. On the one hand, **navigationController** is a read-only property so we can't inject a mock into there, but on the other hand I don't particularly want to start using a storyboard because that's a whole other component of code and I want to keep this test focused on one thing. An easier solution is to create a dummy navigation controller and send it in, like this:

```
func testViewControllerUsesLargeTitles() {
    // given
    let sut = ViewController()
    _ = UINavigationController(rootViewController: sut)

    // when
```

```
    sut.loadViewIfNeeded()

    // then

XCTAssertTrue(sut.navigationController?.navigationBar.prefersLa
rgeTitles ?? false)
}
```

Using this approach, the view controller has a navigation controller that it can manipulate freely and UIKit is satisfied.

The second test – that our view controller has the correct title – is trivial in comparison:

```
func testNavigationBarHasStormViewerTitle() {
    // given
    let sut = ViewController()

    // when
    sut.loadViewIfNeeded()

    // then
    XCTAssertEqual(sut.title, "Storm Viewer")
}
```

The production code to make the test pass is equally easy – add this to **viewDidLoad()**:

```
title = "Storm Viewer"
```

## Step 7: Creating a detail view controller

So far this has all been fairly easy, but now we come to our second screen: a new view controller, with a large image view in the center showing whatever the user selected.

So, let's start by writing a test that the image view exists. You can put this into Project1Tests.swift if you want, but really it's better placed into a new **XCTestCase** subclass – don't forget to add **@testable import Project1** if you create a new test case.

Anyway, we might start writing a test like this:

```
func testDetailImageViewExists() {
    // given
    let sut = DetailViewController()
}
```

…at which point we have our first compile error, because **DetailViewController()** doesn't exist. So, we need to stop writing our test and create that class: press Cmd+N to create a new Cocoa Touch Class, naming it "DetailViewController" and making it subclass from "UIViewController".

That's enough to make our test pass, so we can continue with it:

```
func testDetailImageViewExists() {
    // given
    let sut = DetailViewController()

    // when
    sut.loadViewIfNeeded()

    // then
    XCTAssertNotNil(sut.imageView)
}
```

That will again fail, because we haven't added an **imageView** property to the view controller. So, let's do that now – add this property to the **DetailViewController** class:

```
let imageView = UIImageView()
```

That's enough for our test to pass, but it doesn't really capture the *spirit* of the test: an empty image view isn't much use. At this point it's not uncommon to see folks write tests to check Auto Layout constraints for particular views, but it's not something I'm a big fan of – that kind of thing will break your code every time a small UI change is made, and I think it's better encapsulated using UI tests.

In this case, I'd probably talk to the designer to figure out what they wanted things to look like, and try to implement that design in fool-proof way. In the original Hacking with Swift project the image view took up the full space on the screen, so we can implement that here using another test:

```
func testDetailViewIsImageView() {
    // given
    let sut = DetailViewController()

    // when
    sut.loadViewIfNeeded()

    // then
    XCTAssertEqual(sut.view, sut.imageView)
}
```

That will fail because we're creating the image view but doing nothing special with it, but we can make it pass by adding one new method to **DetailViewController**:

```
override func loadView() {
    view = imageView
}
```

That code by itself isn't enough to complete **loadView()**, because the image view needs to have a white background color and use aspect fill for its content mode – can you figure out how to write tests for them? Here's the code you should end up with in **loadView()**:

```
override func loadView() {
    imageView.backgroundColor = .white
    imageView.contentMode = .scaleAspectFit
    view = imageView
}
```

## Step 8: Selecting images

When the detail view controller is presented, we need to tell it which image was selected so it can be shown in its image view. Let's start writing another test to make sure all that works correctly:

```
func testDetailLoadsImage() {
    // given
    let filenameToTest = "nssl0049.jpg"
    let imageToLoad = UIImage(named: filenameToTest, in:
Bundle.main, compatibleWith: nil)

    let sut = DetailViewController()
    sut.selectedImage = filenameToTest
}
```

That starts by picking a filename to work with and preparing a **UIImage** we can compare against. It then creates an instance of **DetailViewController** and sets its **selectedImage** property so it knows which image to load – and of course that won't work because we haven't created a **selectedImage** property yet.

So, modify **DetailViewController** so that it has this property:

```
var selectedImage: String?
```

Now we can get back to writing our test:

```
// when
sut.loadViewIfNeeded()

// then
XCTAssertEqual(sut.imageView.image, imageToLoad)
```

If you run that you'll see that it fails, because our image view doesn't actually load anything. Sure, we accept the filename that we *want* to load, but nothing actually happens with it.

Of course, the fix is to update **DetailViewController** so that it actually loads the image it's supposed to. Modify its **viewDidLoad()** method to include this:

```
if let imageToLoad = selectedImage {
    imageView.image  = UIImage(named: imageToLoad)
}
```

## Step 9: Handling table view taps

At this point we have written code to make most of our app work, but there's no integration between our components – we can't get from the first view controller to the second. This needs an integration test: a test where we put together two or more live code units to make sure they work together. I still don't particularly want to work with storyboards, so here's what we're going to do:

1. Create a live instance of **ViewController**, then wrap it in a navigation controller inside our test.
2. Trigger the **didSelectRowAt** method by hand, passing in a specific index path.
3. Wait a tiny fraction of a section.
4. Check that the navigation controller is showing a **DetailViewController**.

The only interesting part there is step 3, "wait a tiny fraction of a second." You see, when you push a new view controller onto a navigation controller stack UIKit doesn't do it that second – it waits until all your other code has finished executing before it happens. So, if we try to check

immediately we'll see the wrong view controller on the stack.

Fortunately, the fix for this is easy, and does nothing to make our unit test slow: we create an **XCTestExpectation** then fulfill it immediately using an asynchronous closure. Unless you provide a specific deadline, asynchronous closures are executed on the next run loop, which means it will happen after UIKit has finished its work of creating and showing our view controller. It's not ideal – in fact, I'd say it's a "workaround" if I were putting it nicely, or a "hack" otherwise – but it does the job.

Here's the code:

```
func testSelectingImageShowsDetail() {
    // given
    let sut = ViewController()
    let navigationController =
UINavigationController(rootViewController: sut)
    let testIndexPath = IndexPath(row: 0, section: 0)

    // when
    sut.tableView(sut.tableView, didSelectRowAt: testIndexPath)

    // create an expectation…
    let expectation = XCTestExpectation(description: "Selecting
a table view cell.")

    // …then fulfill it asynchronously
    DispatchQueue.main.async {
        expectation.fulfill()
    }

    // then
    wait(for: [expectation], timeout: 1)
    XCTAssertTrue(navigationController.topViewController is
```

```
DetailViewController)
}
```

Yes, that has a nominal timeout of one second, but in practice it will be fulfilled immediately – we don't even need to specify an artificial delay in there, because it's guaranteed to be done on the next runloop.

Of course, that test won't actually pass yet because calling **didSelectRowAt** does nothing yet.

So, let's write the code to make it pass:

```
override func tableView(_ tableView: UITableView,
didSelectRowAt indexPath: IndexPath) {
    let vc = DetailViewController()
    navigationController?.pushViewController(vc, animated: true)
}
```

We're in the green state now because our tests all pass, but I hope you're feeling a bit uncertain at this point – our test isn't as clean cut as the previous ones we wrote.

## Step 10: Loading a real image

We've written tests for our two view controllers in isolation, and we've written an integration test that puts them together to test cell selection. For the final integration test, we're going to trigger cell selection in the first view controller, and make sure it loads an image in the second view controller. This is really just a matter of combining the **testDetailLoadsImage()** and **testSelectingImageShowsDetail()** tests into one.

Let's take a look at a starting implementation:

```
func testSelectingImageShowsDetailImage() {
    // given
    let sut = ViewController()
    let navigationController =
```

```
UINavigationController(rootViewController: sut)
    let testIndexPath = IndexPath(row: 0, section: 0)
    let filenameToTest = "nssl0049.jpg"
    let imageToLoad = UIImage(named: filenameToTest, in:
Bundle.main, compatibleWith: nil)

    // when
    sut.tableView(sut.tableView, didSelectRowAt: testIndexPath)

    let expectation = XCTestExpectation(description: "Selecting
a table view cell.")

    DispatchQueue.main.async {
        expectation.fulfill()
    }

    // then
    wait(for: [expectation], timeout: 1)

    guard let detail = navigationController.topViewController
as? DetailViewController else {
        XCTFail("Didn't push to a detail view controller.")
        return
    }

    detail.loadViewIfNeeded()

    XCTAssertEqual(detail.imageView.image, imageToLoad)
}
```

I don't know about you, but that's not particularly nice code to my eyes. However, that's OK: the point here isn't to write beautiful code first time, but just to write an initial pass at the test.

The above test will fail because when we tap on a row, we're pushing a **DetailViewController** without setting its **selectedImage** property. So, change the **didSelectRowAt** method to this:

```
override func tableView(_ tableView: UITableView,
didSelectRowAt indexPath: IndexPath) {
    let vc = DetailViewController()
    vc.selectedImage = pictures[indexPath.row]
    navigationController?.pushViewController(vc, animated: true)
}
```

Now the test will pass, so we're in the green stage and it's time to refactor. Can we do this better? Can we make it more efficient, or remove duplication?

In this instance, a few places seem decided uncomfortable and by putting them together the whole thing just sits uneasily:

- Back in step 6 we created a testing navigation controller so we could check large title mode was working. You could argue that was a sensible choice, because if you were to create a mock it would just duplicate the property in question.
- However, in step 8 we started pushing view controllers onto the navigation controller, which meant dabbling with async calls to make sure something got pushed correctly.
- We also started calling **didSelectRowAt** by hand so that we could monitor what happened.

And now we're putting all that together: we're creating a test navigation controller, triggering a table view action, pushing a view controller, dabbling with async, then examining the result. This is *complicated*, and complicated often morphs into *flaky*. That doesn't mean keeping this test is a bad idea, because you could make a clear case that as integration tests go this one really is the fundamental test that the app is working as expected. However, it is a significant code smell that our code needs to be refactored.

In this particular instance, there's one particular thing that's annoying me: the use of **navigationController**. This property is a really common hidden dependency in view controllers, and it goes away entirely if we use coordinators. In this case, we can use

coordinators with closure injection because there's only one action being triggered, and it would allow us to substitute the coordinator with a mock in our tests – no more need for the async futzing.

## Refactoring our tests with coordinators

We're currently in the green state, which means we're in a safe place to refactor. So, we're going to do something big and important: we're going to remove the **navigationController** hidden dependency from our view controller, at least when it comes to navigation - we can't remove it for handling large titles because that's literally hard-coded into UIKit.

First, we need to switch our app over to using coordinators. So:

1. Open Main.storyboard and delete the navigation controller.
2. Select the remaining view controller and make it the initial view controller for the storyboard.
3. Go to your project's settings – click "Project1" at the top of the project navigator, then select the General tab. Now look for "Main" next to the Main Interface setting and delete it.
4. Create a new Swift file called MainCoordinator.swift.

Now give MainCoordinator.swift this code:

```
import Foundation
import UIKit

class MainCoordinator {
    var navigationController = UINavigationController()

    func start() {
        let storyboard = UIStoryboard(name: "Main", bundle: nil)

        guard let viewController =
storyboard.instantiateInitialViewController() as?
```

```
ViewController else {
         fatalError("Missing initial view controller in
Main.storyboard.")
      }

      navigationController.pushViewController(viewController,
animated: false)
   }
}
```

When that coordinator is created, it will also create a navigation controller so it can host itself, and it's also capable of creating the initial table view controller for the app.

The last conversion step is to boot strap our app so that it uses the coordinator. When you don't use Main.storyboard for creating your layout, you need to create your **UIWindow** by hand. This takes a few lines of boilerplate code in AppDelegate.swift, but it's nothing tricky.

First, open AppDelegate.swift and give it this new property:

```
var coordinator: MainCoordinator?
```

Now fill in the **didFinishLaunchingWithOptions** method like this:

```
func application(_ application: UIApplication,
didFinishLaunchingWithOptions launchOptions:
[UIApplication.LaunchOptionsKey: Any]?) -> Bool {
   // create our coordinator
   coordinator = MainCoordinator()
   coordinator?.start()

   // create the window for our app and make it use our
navigation controller
   window = UIWindow(frame: UIScreen.main.bounds)
```

```
    window?.rootViewController =
coordinator?.navigationController
    window?.makeKeyAndVisible()

    return true
}
```

That embeds a coordinator at the core of our app, but it doesn't actually *use* it yet. Before we write that code, though, press Cmd+U to run all your tests: all being well everything should still pass, which is a good sign that our work is safe.

Here's where the important part comes in. To make our code more testable, we need to remove the **navigationController** hidden dependency for all navigation code. Instead, we want to call a closure that will be injected by our coordinator so that we can replace it with a mock during testing.

So, open ViewController.swift and give it this property:

```
var pictureSelectAction: ((String) -> Void)?
```

Now go down to its **didSelectRowAt** method and cut these three lines to your clipboard:

```
let vc = DetailViewController()
vc.selectedImage = pictures[indexPath.row]
navigationController?.pushViewController(vc, animated: true)
```

We'll need them again in a moment, but first I'd like you to replace them with this:

```
pictureSelectAction?(pictures[indexPath.row])
```

So, your entire **didSelectRowAt** method should be this:

```
override func tableView(_ tableView: UITableView,
didSelectRowAt indexPath: IndexPath) {
```

```
    pictureSelectAction?(pictures[indexPath.row])
}
```

Now head back to MainCoordinator.swift, and add this new method:

```
func showDetail(for filename: String) {
    let vc = DetailViewController()
    vc.selectedImage = filename
    navigationController.pushViewController(vc, animated: true)
}
```

You can get the middle three lines by pasting in the code from your clipboard, then modifying it just a little – we need to use **filename** rather than **pictures[indexPath.row]**, and the **navigationController** property is no longer optional.

Finally, we need to connect that method to the **pictureSelectAction** on our view controller. So, add this to the **start()** method, just before the call to **pushViewController()**:

```
viewController.pictureSelectAction = { [weak self] in
    self?.showDetail(for: $0)
}
```

That assigns a new closure to the **pictureSelectAction** property on our view controller – it uses a new closure so that we don't accidentally get a retain cycle between our view controller and coordinator.

Go ahead and run your tests again, and you'll see… yes, they fail. This isn't great: ideally you want to start your refactor in green and end it in green, but now we have broken tests so how can we verify that our changes our good?

Well, the problem here is that we weren't writing great tests - we were leaning heavily on a hidden dependency, and all the asynchronous code really should have been a give away that we were getting into increasingly detailed integration tests rather than smaller, simpler unit

tests.

That's why I titled this section "refactoring our tests": it's not possible to change this code to make it more testable without also breaking a few of our tests – we've ripped out a big chunk of our app, and it's time to do the matching work to upgrade our tests.

The first test that fails is **testSelectingImageShowsDetail()**, but with our closure injection approach we can now dramatically simplify it. This test is here to verify that when a particular row is tapped, the view controller triggers its picture selection action with that image, so we can inject a closure to read that value:

```
func testSelectingImageShowsDetail() {
    // given
    let sut = ViewController()
    var selectedImage: String?
    let testIndexPath = IndexPath(row: 0, section: 0)

    // when
    sut.pictureSelectAction = {
        selectedImage = $0
    }

    sut.tableView(sut.tableView, didSelectRowAt: testIndexPath)

    // then
    XCTAssertEqual(selectedImage, "nssl0049.jpg")
}
```

The second failing test, **testSelectingImageShowsDetailImage()**, is trickier, because this really is the ultimate integration test. Still, we could write a slightly simpler integration test by removing our coordinator and using a custom closure, like this:

```
func testSelectingImageShowsDetailImage() {
```

```
    // given
    let sut = ViewController()
    let testIndexPath = IndexPath(row: 0, section: 0)
    let filenameToTest = "nssl0049.jpg"
    let imageToLoad = UIImage(named: filenameToTest, in:
Bundle.main, compatibleWith: nil)

    // when
    sut.pictureSelectAction = {
        let detail = DetailViewController()
        detail.selectedImage = $0
        detail.loadViewIfNeeded()
        XCTAssertEqual(detail.imageView.image, imageToLoad)
    }

    sut.tableView(sut.tableView, didSelectRowAt: testIndexPath)
}
```

It's better than what we had, while still being good enough to count as an integration test.

All this could and should have been avoided earlier if we had listened to the warning bells: hidden dependencies regularly cause problems, and when you start relying on them the testability of code can easily take a nosedive.

# Adopting a test-first mentality

We've now looked at TDD in two ways: one where we had a simple, clean piece of code that we could write tests for, and one where we had to navigate around some of UIKit's quirks. In practice you'll find yourself writing far more of the latter than the former, which is why it's so important to listen to your TDD spider senses whenever they start tingling – don't get into the situation where you write one slightly grungy test, then write another even grungier test, and so on, because you're just going to cause pain for yourself further down the line.

In our UIKit example we ripped out a large part of the app and replaced it with coordinators. That in turn caused tests to break, so we had to rewrite tests while we were also rewriting the code – a situation that makes it much harder to track whether our changes were actually safe or not. Of course, the *right* thing to do would have been to use coordinators from the start, because it avoids the hidden dependency for navigation.

You're going to get yourself into these sorts of holes a lot, and that's OK – don't be too hard on yourself. I encourage folks to adopt a technique I call TDDBut – it's like regular TDD, except you make exceptions for things that don't fit in well with your team or your personal approach. So, you can say "I use TDD but sometimes I hold off refactoring until I've written a few tests." Is it ideal? Not really – refactoring ought to be an integral step in the process. But if it gets you closer to where you want to be, go for it! Don't get stuck in a hole of fighting with UIKit if it's not helping you write stable, repeatable tests.

All being well, you'll start to learn how to marry up TDD with your own coding style and find something that works great for you and your team. Over time you'll get faster and more comfortable with a test-driven approach, and you might even grow to rely on it for your own work – let me know how you get on!

## Test writing tips

I've already given you some Xcode tips that help make TDD easier, but there are a few more general tips that I think will help.

First, even if you go down the route of TDDBut you'll find it helpful to keep a clear and close mapping between your test code and production code. When Robert C. Martin learned TDD from Kent Beck (a key proponent of TDD), he described Beck like this: "he would write one line of a failing test, and then write the corresponding line of production code to make it pass. Sometimes it was slightly more than one line; but the scale he used was was very close to line by line." You might have a 1:1 line mapping in your code, or maybe a 1:1 mapping between methods and tests, but you'll likely find some sort of consistent relationship helpful.

Second, if you ever write a complete test and it passes straight away – with no production code changing – you should be very worried indeed. This means your app is behaving in an unexpected way, and you'll likely find yourself in the bizarre position of *trying to make your test fail*.

Third, even when you're writing small, simple tests it's important that they still stick to the FIRST principles of unit tests. Remember, you'll be running these tests a lot, so if you write a test that sometimes returns false positives or false negatives people you run the risk of people starting to ignore them – at which point the value of automated testing decreases.

Fourth, you'll find that some teams prefer to use behavior-driven development (BDD) rather than TDD. BDD is a variant of TDD that focuses on a standardized language that describes tests from the perspective of user stories. So, rather than focusing on "when this method is called this thing should be returned," you instead focus on "when the user buys a product they should have the product in their library." Honestly, all that changes is the language.

Finally, watch out for the trap of over-testing. When tests are the first thing you write and you're deep into the Red-Green-Refactor cycle, it can be easy to forget to stop and look around now and then – to look at the bigger picture. Back in chapter 1 I mentioned a company that had over 7000 tests that took 25 minutes to run. Those developers *know* they can delete some of their tests, but they don't know *which*. So, when you're refactoring, every so often stop to look at the bigger picture: can you refactor more than just the tests you were working on? Can you make improvements to your test code without reducing test coverage? Test code takes just as much maintenance as production code, so treat it with the same amount of love!

# Chapter 6
# Continuous Integration

# Always be testing

Continuous integration is the name we give to the process of checking code into a central repository several times each day, and – for modern developers – the configuration of servers that respond to those check ins by automatically running all tests.

This solves two major problems at the same time:

1. Developers are much less likely to create monster patches – change sets consisting of thousands of lines or more become almost impossible if check ins happen frequently.
2. Problems are spotted earlier in the development process, because automatic test runs catch problems faster.

Now, I should make it clear that continuous integration (CI) does not stop you from running your own tests locally before you commit code – quite the contrary! However, it's fair as your project increases in size it becomes harder for developers to run all the tests in a usefully short period of time, particularly when running across multiple devices.

Having a dedicated CI server – one or more computers able to build your code and run all tests – allows developers to run some tests locally, knowing that the full suite will be run on their server, across all supported devices.

Xcode comes with the ability to act as a CI server, but there are also third-party services such as CircleCI and Bitrise that specialize in taking the load off your hands.

These third-party components have several advantages of using Xcode Server. For example, they integrate directly into GitHub, which means you can make sure all tests pass before a pull request can be merged. These tools usually have some sort of free plan, although you might find it's for open-source work only – that means you'll need to make sure your Git repository is public.

However, Xcode Server has the advantage of being built right into the tool we use every day, and of course it's completely free. This means you can use it for your private work without paying a subscription fee to an online service, or just to play around with CI without any

watching.

In this chapter I'm going to walk you through setting up and using Xcode Server for continuous integration so you can benefit from the safety of regular, automated test runs. We're also going to briefly look at an example third-party service, CircleCI – this is not me endorsing them, but they are as good an example as any for such services.

These tools give us the power to run testing much more aggressively than would be possible on local machine: all commits and all pull requests can be tested, across all devices, as often as you want. However, they *can't* force your developers to commit code regularly: that's a hygiene issue you need to solve inside your team if you want CI to work.

**Note:** For the avoidance of doubt, the very first step in continuous integration is having developers check their code into a source control system. If you aren't using source control you need to stop and set that up before continuing – create a new private repository on GitHub or Bitbucket, if you like.

# Xcode server

Your version of Xcode has been leading a secret double life all this time. Alongside Interface Builder, the assistant editor, and the new project templates lies a whole subsystem called Xcode Server. This is installed alongside the regular Xcode, and can act as your own personal CI system. While it can't have some of the super-smart GitHub integration powers that online services have, Xcode's option is free, configured through a familiar user interface, and entirely private.

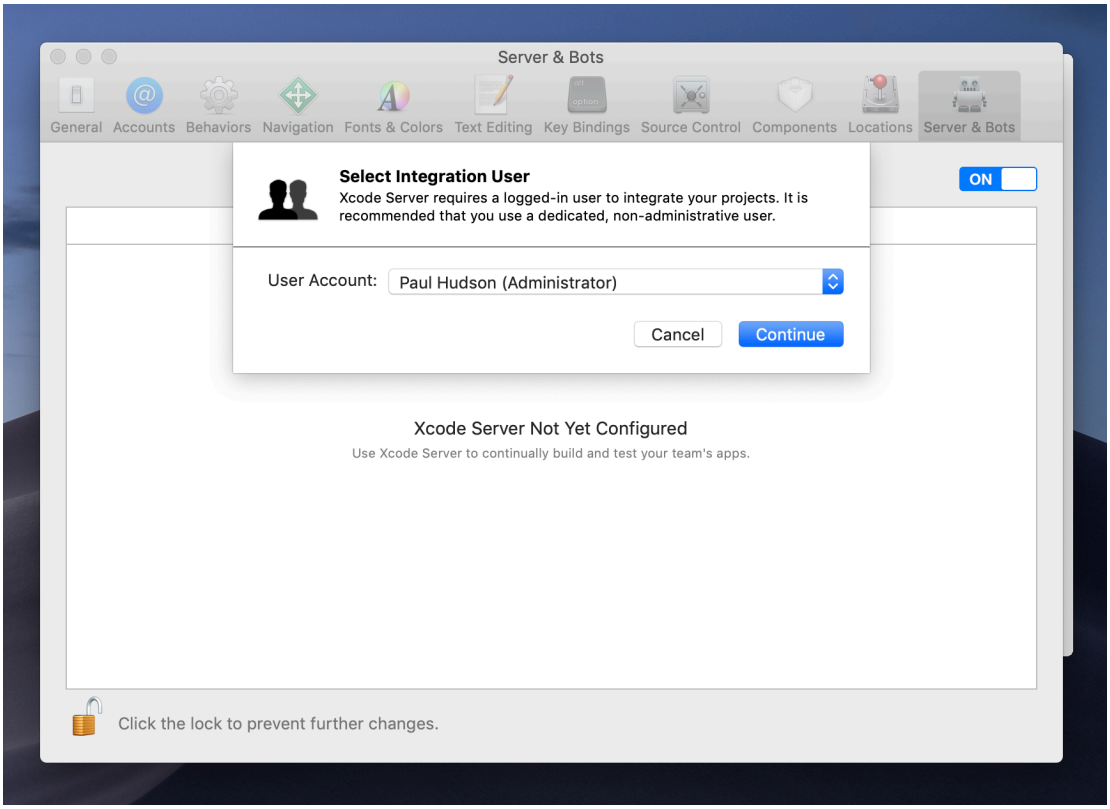
In this chapter we'll look at how to set up Xcode Server, configure it with some integration tasks, then monitor how it's working. **This will almost certainly take longer than you expect** – there are many options to go through, and although you can change your mind easily later on you will still want to at least understand what's happening.

## Enabling Xcode Server

The first step is to configure Xcode Server so that it's running on a Mac somewhere. If you're just testing it out this might be the same Mac you use for Xcode development, but you might also have a spare Mac that can be used instead. Either way, this computer should be online and connected to the network all the time, otherwise it won't be able to monitor your source control system for changes.

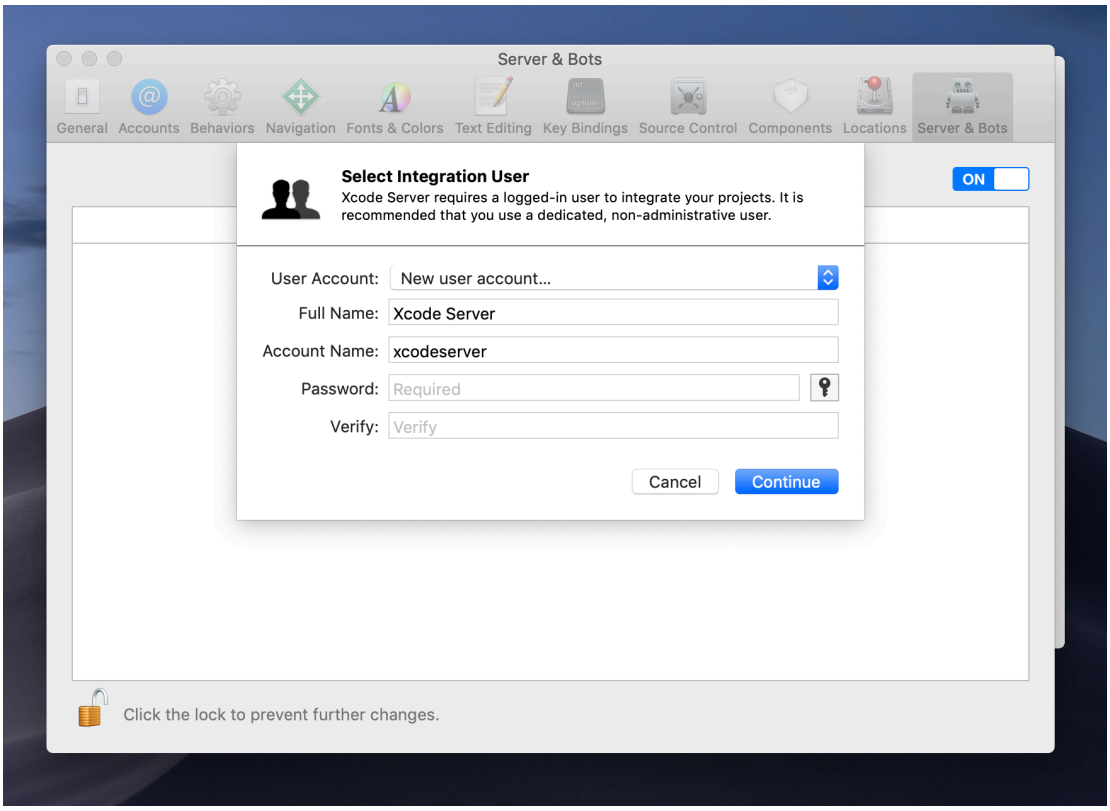
Using whichever Mac you want to configure for Xcode Server, go to Xcode's preferences window and look for the Server & Bots tab on the far right. Turn the switch to On, then enter your administrator password.

Next you'll be asked to select the integration user. This will be the user account that runs your tests, and Xcode will offer some clear, important advice: create a dedicated, non-administrative user. This user needs to be logged in at all times, so creating a dedicated account with limited privileges helps keep your computer fast without exposing it to security problems.




**Note:** To avoid confusion, I've used macOS's light mode for screenshots on my Xcode Server Mac, and dark mode for my main development Mac.

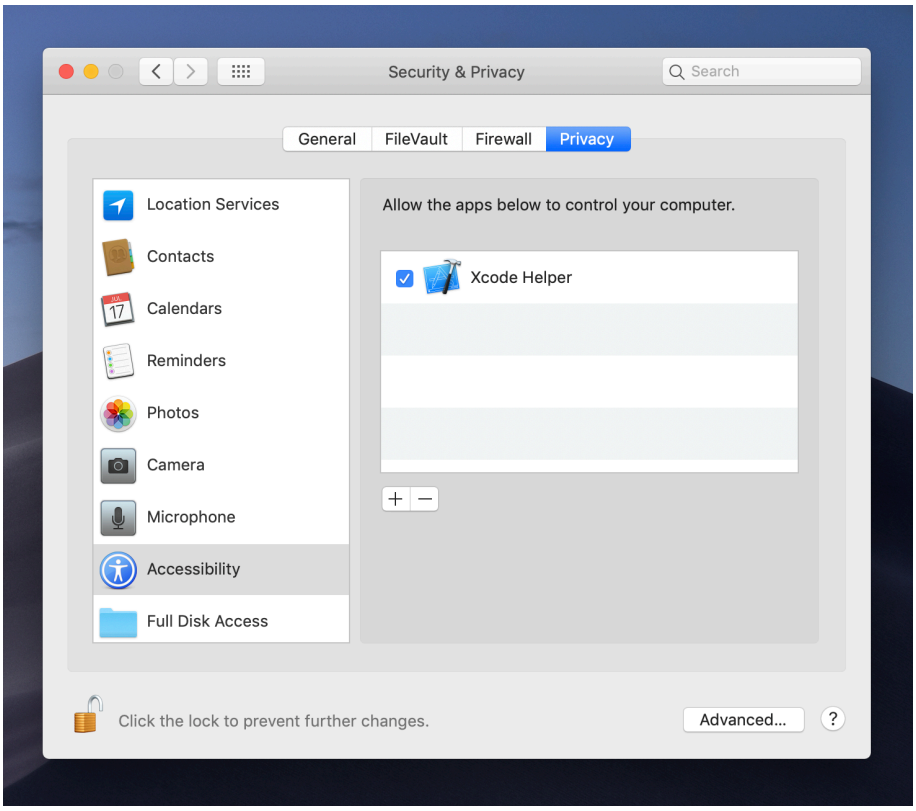
If you open the User Account dropdown menu you should see New User Account, and when that's selected Xcode will fill in the account name "xcodeserver", leaving you to enter and verify the password. Once that's done Xcode will do a little work to set this up: you'll see "Enabling Developer Mode", "Configuring SSL Certificates", "Starting API server", and more fly by.



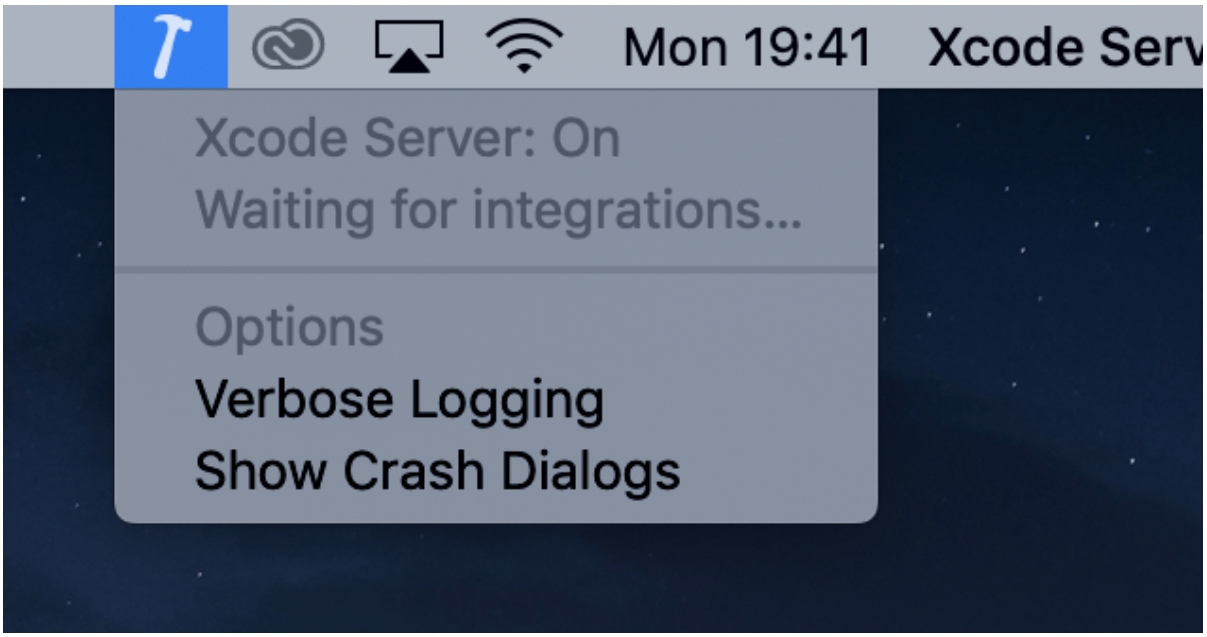

Once Xcode Server is configured, the next step is to set it running. You'll probably see a message along the lines of "You must log in as xcodeserver to run integrations" – this appears because the Xcode Server account needs to be actively running, in the background or foreground, in order to run tests. So, click Log In, then enter the password you created for your Xcode Server account.

Logging into a new macOS account for the first time (even one that isn't a real person) inevitably means you'll see the usual welcome messages about privacy and iCloud – just click your way through those using whatever "Skip" options you can see.

**When login completes you'll probably see an alert that tells you Xcode Helper wants to control the computer.** This is important: click Open System Preferences, click the lock to make changes, enter your password, then check the box next to Xcode Helper.




At last, this computer is now fully configured for Xcode Server – you should see an Xcode hammer icon appear in your macOS status bar, giving you access to logs.



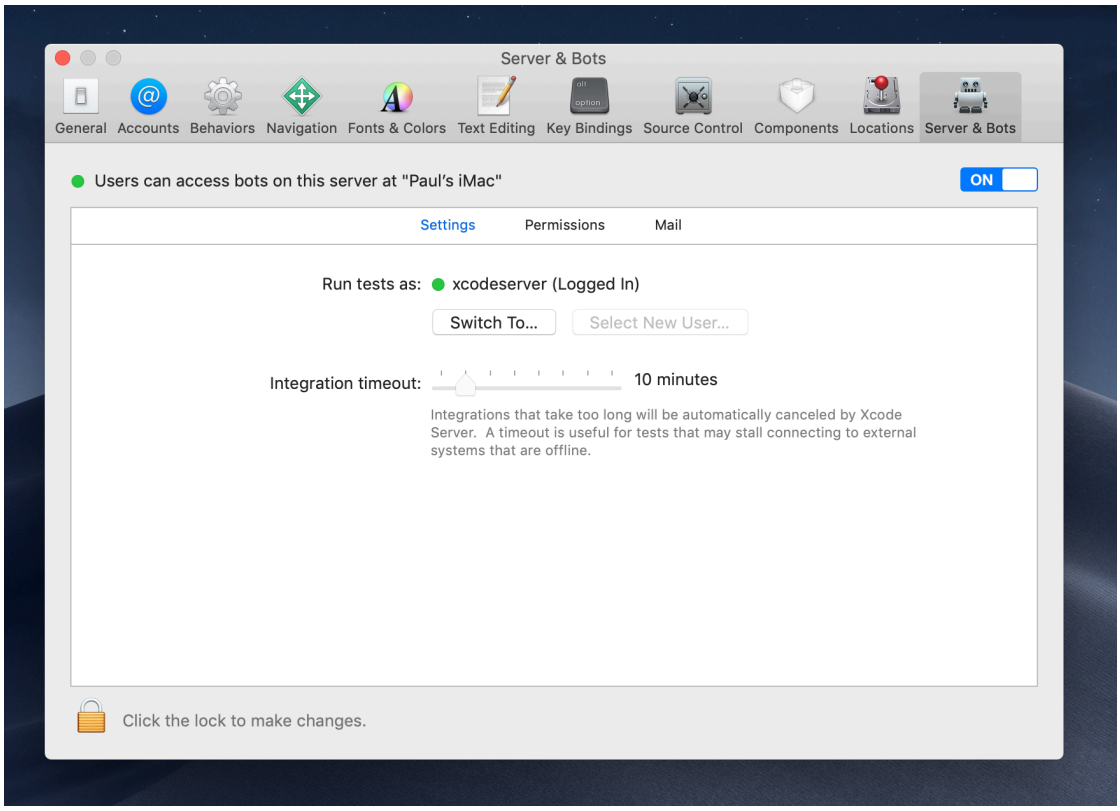
However, here's the catch: the Xcode Server account needs to stay logged in for your integrations to run. That doesn't mean it needs to be the *active* account – i.e., the one currently in control of the computer – but it does need to be logged in.

If you have a dedicated Mac then you have nothing to worry about: you're logged in as the Xcode Server account, and can just walk away.

If not – if you're just testing out Xcode Server on the same Mac you intend to build apps – then you need to make sure you have Fast User Switching enabled. If you see "Xcode Server" in the top-right corner of the window it means you have Fast User Switching enabled already – select that, then click your usual account name from the menu.

If you *don't* see Xcode Server in the top-right corner of your Mac's status bar, go to System Preferences > Users & Groups, click Login Items, click the lock to make changes, then click the box next to "Show fast user switching menu as". Now click your usual account name – this will switch to your usual account while leaving Xcode Server's account active.

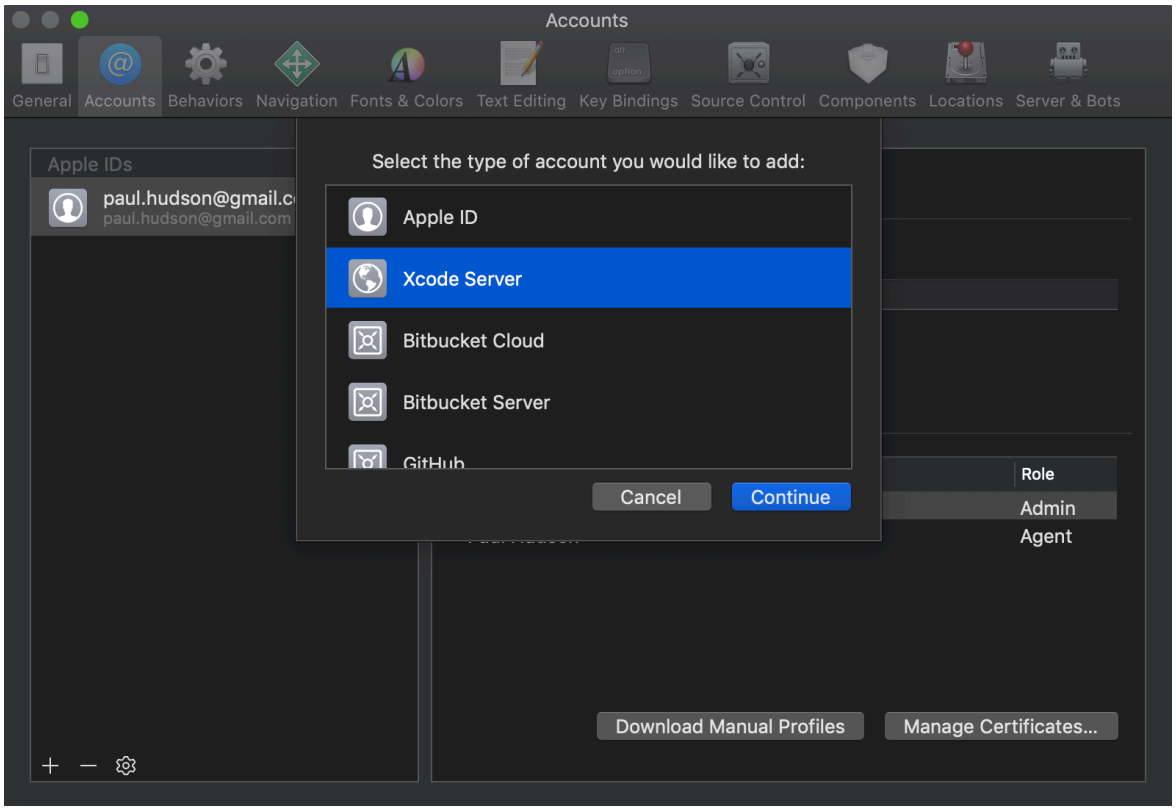
Once you're back to your usual account, you'll see Xcode's Server & Bots tab has updated: you should see a green light near the top saying "Users can access bots on this server at Paul's iMac" – or whatever your Mac is named.



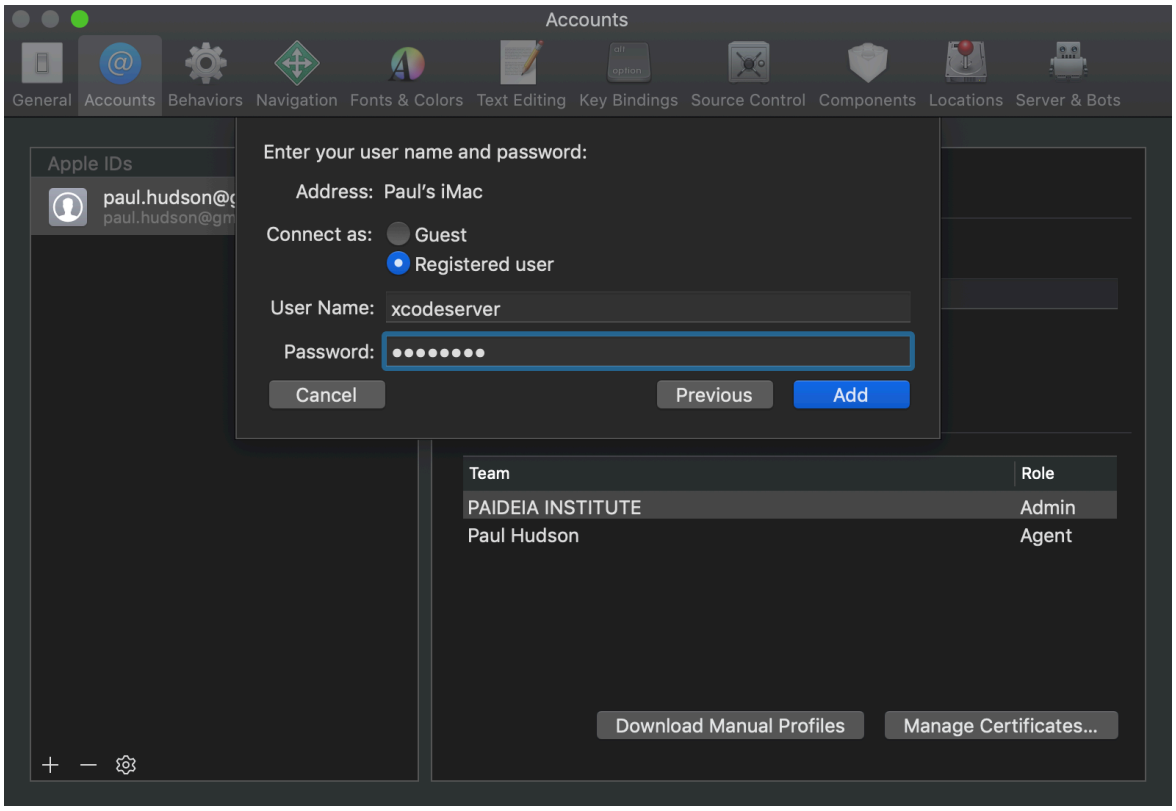

## Adding integrations

Now that we have a server – either remote or local – the next step is to configure *our* Xcode to use it. All those steps above were just to get Xcode Server working on our machine. The *real* work is to add some integrations – to tell Xcode how to actually monitor and test our code. Xcode calls these *bots*, because they are little pre-configured robots taking automated actions on your behalf.

First, go to Xcode's preferences then its Accounts tab. You'll see your usual Apple ID in there if you have a paid-for Apple developer account, but I'd like you to click the + button in the bottom-left of that window. From the list that appears, please choose Xcode Server, and you should find your Xcode Server's Mac is listed on the following screen – Xcode has scanned the network and found all possible candidates. If it hasn't, you can enter a server address by hand.




Once you're ready – either by selecting your Mac from a list or by entering an address – click Next to provide some credentials. I used "xcodeserver" as the username, with a strong password, so please enter whatever you chose here.




Now that Xcode knows where bots will be run, we can go ahead and create one: open whatever Xcode project or workspace you want to use, then go to the Product menu and choose Create Bot.

**Be warned: creating a bot involves clicking through several screens.** Fortunately, you can edit the bot at any later point and change your mind – in fact, because this is all private you can create, modify, run, and delete bots as often as you want until you feel you understand it.

When creating a bot, the first thing you'll be asked to do is give it a name and select which server it should run on. Name it whatever works for you, but obviously please make sure your Xcode server is selected. When you click Next, Xcode will try to authenticate with your source control repository, which probably means it's going to connect to GitHub.

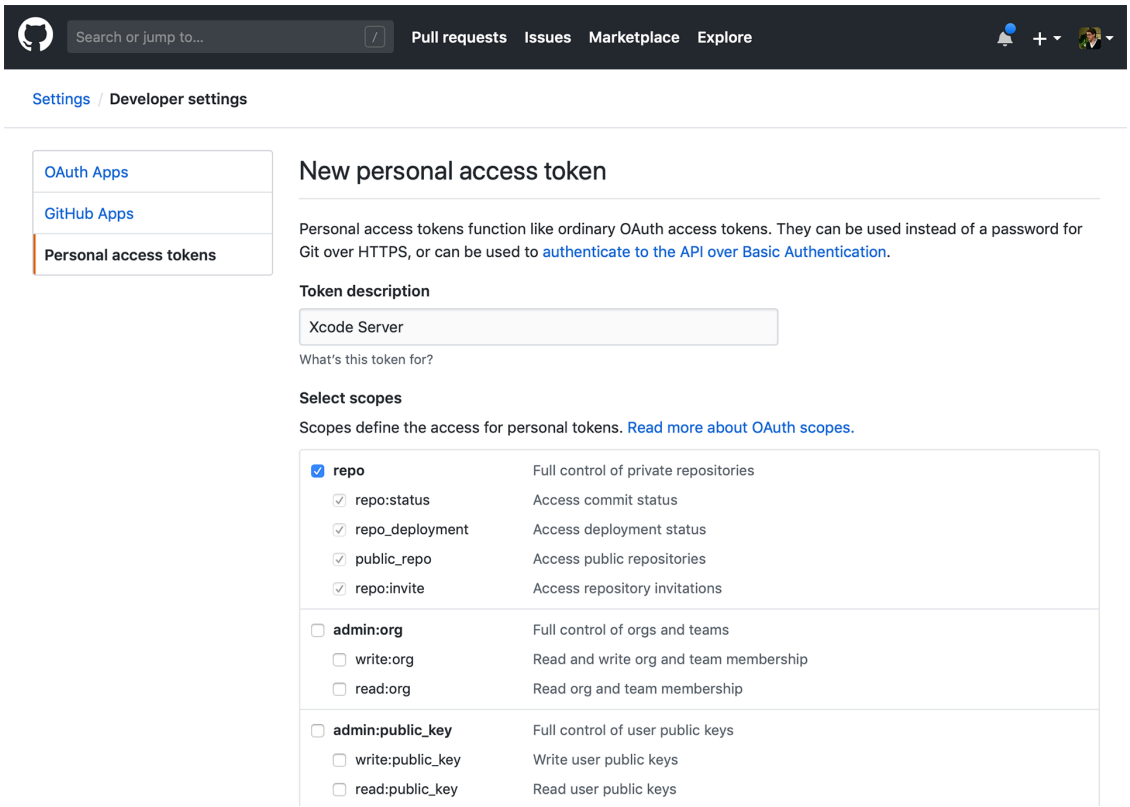
**This may well fail.** If you're accessing a repository that Xcode can't read, you'll need to provide it with your GitHub credentials. However, just to make things that little bit trickier, you can't use your actual GitHub password here. Instead, you need to create a personal access

token – it's effectively a new, random password generated for external services, so you can revoke access later on without compromising your GitHub account.

**Note:** Obviously there are lots of source control services in the world so if your own provider isn't GitHub then you might find you can ignore this section entirely.**

If you *do* use GitHub for source control, visit **https://github.com/settings/developers** to go to your developer settings screen, then select Personal Access Tokens from the left-hand menu. In that screen, click Generate New Token and enter your password, and you'll see a screen full of checkboxes that let you decide exactly what this new access token should be able to do with your GitHub account.

There are lots of options here, but the only ones that matter are in the "repo" section, so check the "repo" box to enable them all. As for the token description, this needs to be something memorable so you know what it is later on – "Xcode Server" is fine.



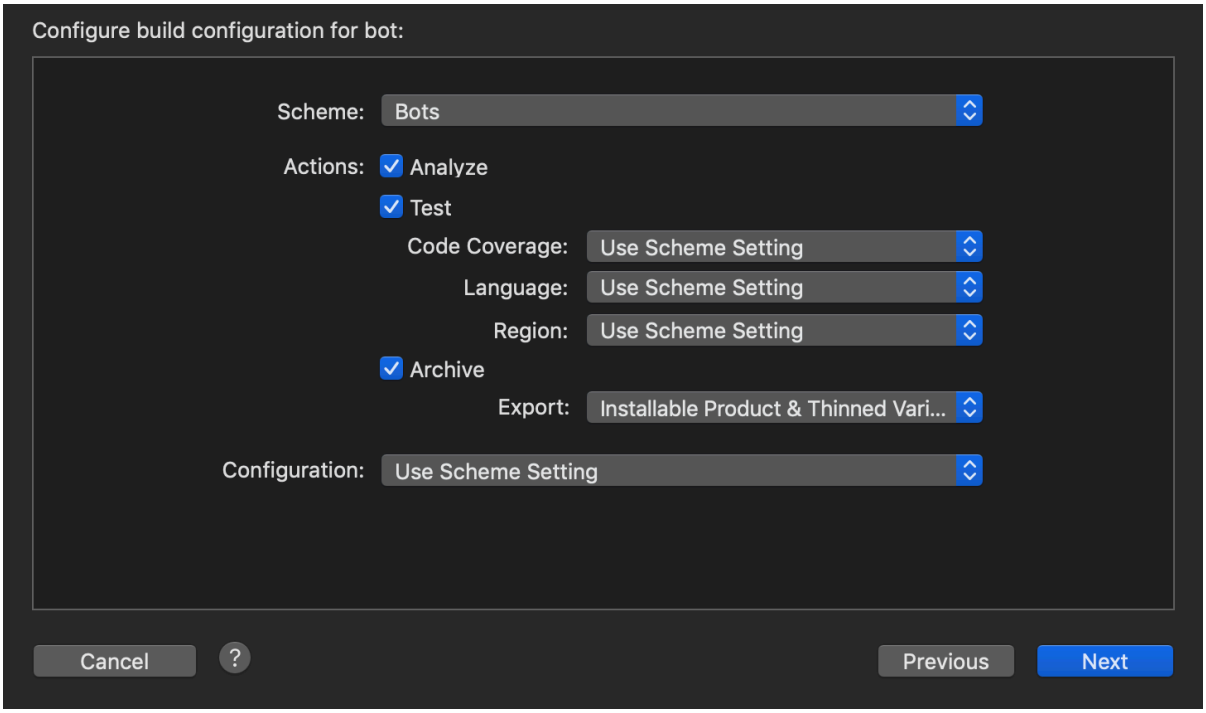

**Important:** When you click Generate Token GitHub will show you the token on-screen. This

is a long string of hexadecimal digits, and you need to copy it to your clipboard *now*. This is effectively the password that Xcode needs to use in order to access your account, and GitHub won't show it to you again - you need to revoke the token and issue a new one if you lose this.

So, copy the token somewhere safe, then back in Xcode use *that* for your GitHub password rather than your regular password. Xcode will try authenticating again, and this time it will succeed – select your source control target, which will be your GitHub repository.

Next you'll be asked to set up the build configuration for your new bot. This can be helpful when you want your bots to be more thorough than your regular settings, but "Use Scheme Setting" is the default for a reason – it will do whatever you configured in Xcode, which is sensible when you're just getting started.

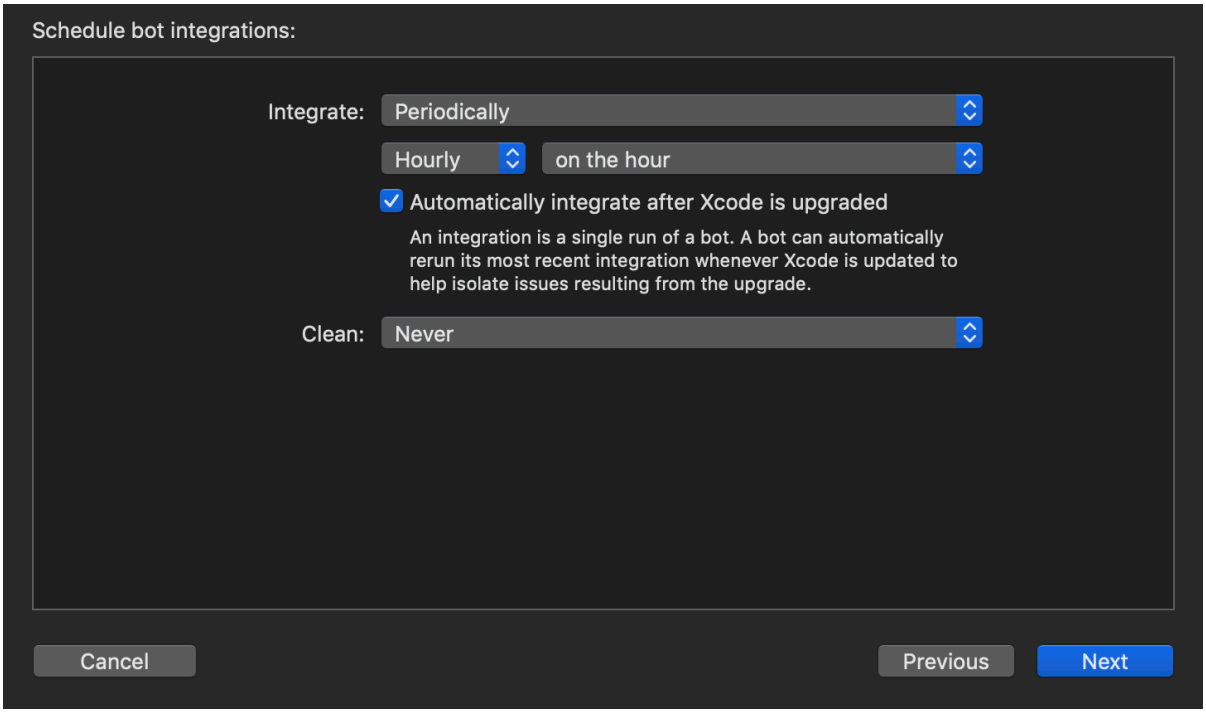
Moving on, Xcode will ask you whether it should Analyze, Test, and/or Archive your project. The Analyze step used to be really important back when Objective-C was widespread, but it's effectively irrelevant in Swiftland so you can uncheck it. As for the Archive step, again this used to be useful in years gone by before we had Fastlane, but these days doesn't get much use. When you're starting out "Test" is the one you care about most, so uncheck the other two.



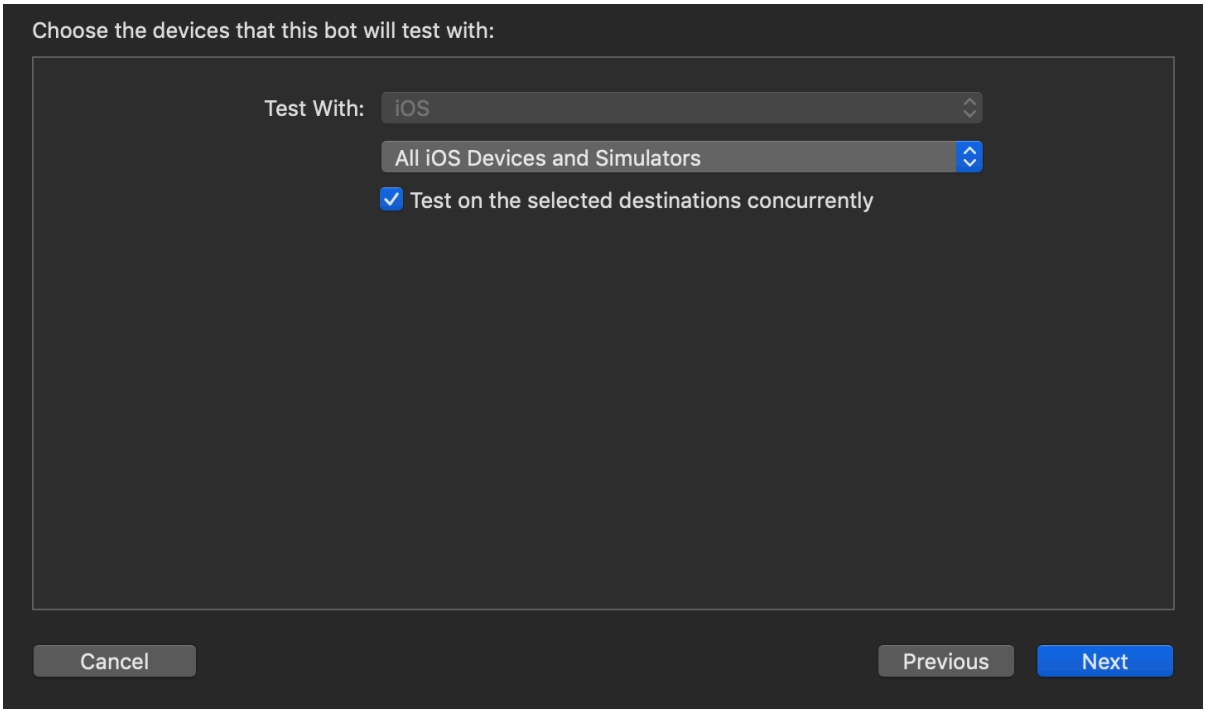

When you click Next you'll be asked how often Xcode should integrate – i.e., run all your tests. It's set to hourly by default, but you can also use On Commit to have Xcode check Git regularly to see if changes have happened, and trigger an integration as appropriate. If you're just testing it out, using Manual mode gives you that little bit of extra control, and is a smart choice.

At the bottom of this screen you'll see a dropdown menu marked "Clean", which determines how often Xcode should wipe out its build cache. It's set to Never by default, but it's a good idea to give this some *sort* of value depending on the complexity of your project. If it takes more than a minute to build your project then go for Once A Day, otherwise you could try Always and see how you get on. Let's face it, unless you have a team of 40 or above your build servers are going to be sitting around doing nothing most of the day.

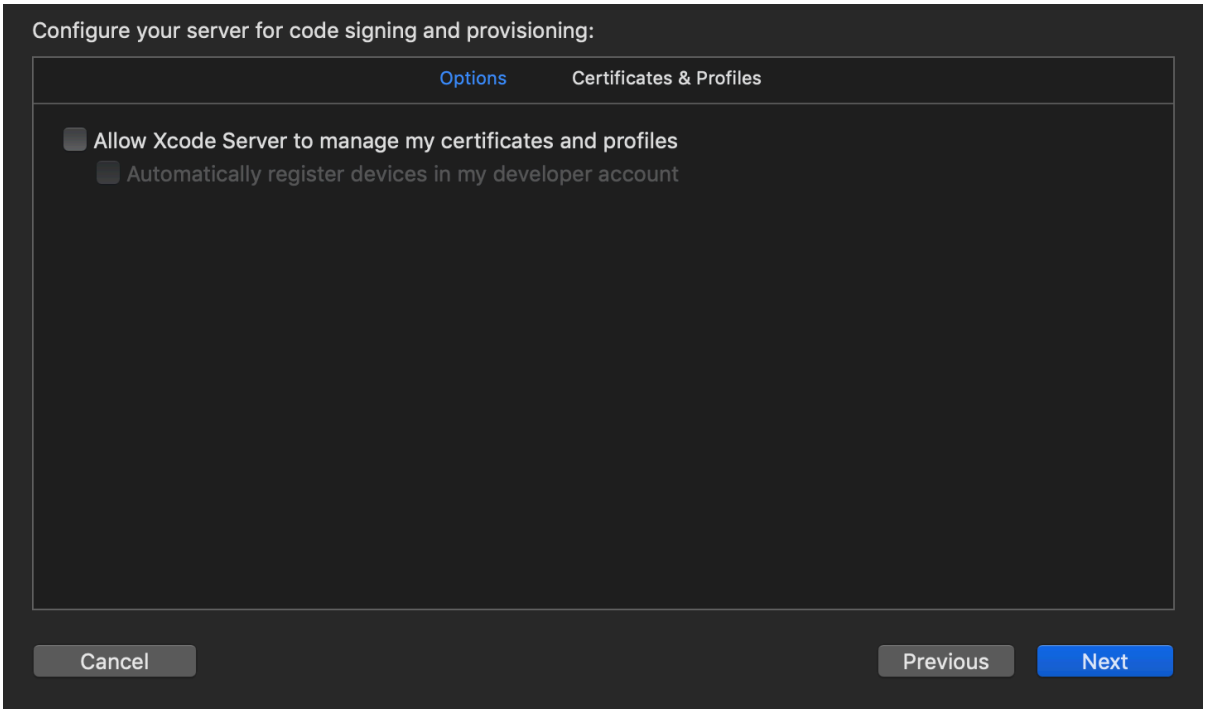
**Tip:** You'll see that "Automatically integrate after Xcode is upgraded" is checked by default. This is because upgrading Xcode often introduces warnings and errors that weren't present previously, so these need to be caught immediately and handled specially.




When you click Next you'll be asked which devices should be tested. It's set to All iOS Devices and Simulators by default, which you might find rather intensive depending on how many tests you have and how frequently you asked Xcode to integrate. While it's true that some APIs work different on devices than in simulators, using real devices creates extra complications – I prefer to work without them if possible, at least for unit tests. Don't feel bad about selecting only a subset of devices: the latest you support, the oldest you support, and one in between is fine.

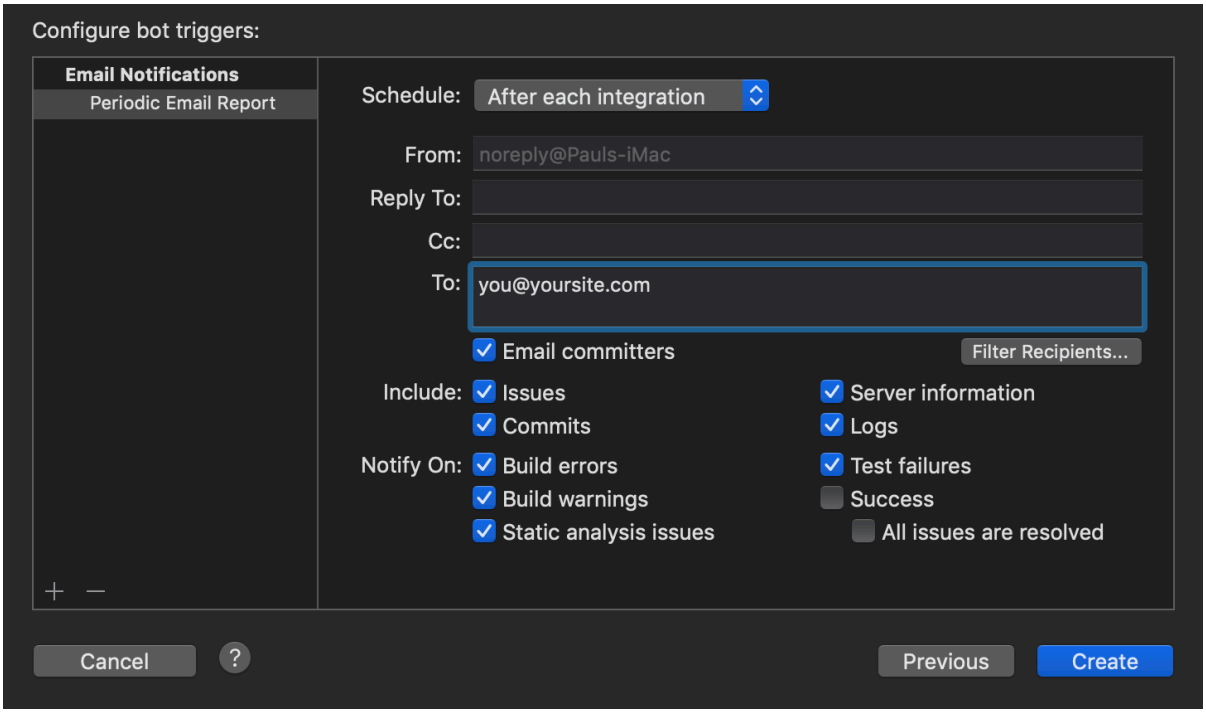



Next you have the option of configuring the server for code signing and provisioning. If you haven't yet discovered Fastlane and intend to use the "Archive" option for integrations, or if you're testing with actual devices, then you should check this box to avoid problems. Otherwise, it's best to leave Xcode Server as simple as possible.




Next you can pass custom environment variables for your bot, in case you want your code to detect that it's being run by CI. For example, you might set environment variables that mark the app as in test mode so that it can install a default configuration that's pre-set for testing. If you aren't using these just click Next.

Finally, you can create triggers, which are Xcode's name for things that should happen before or after integration, or from time to time. For example, you might want periodic reports sent out to specific people every day, every week, or every after each integration completes, or you might prefer individual emails sent to someone who breaks the build. The two script options – pre-integration and post-integration – let you write any shell scripts you like, for complete customizability, but if you aren't already using these in Xcode you won't need them here either.




We've finally reached the last step so click Create, and we're done.

## Running integrations

As I said before, you can adjust your bots at any point later on, and now that you've seen the full range of options out there it's likely you'll want to go back and revisit them later on. For now, though, Xcode will try running the bot you just made: Xcode Server will check out your code, build it using the configuration you provided, test it on all the devices you selected, then upload the results so you can view them in your regular Xcode window.

Depending on the size of your project and the complexity of your configuration, running an integration could take anywhere from one minute to over an hour – I expect it's fairly common for new developers to create complex bots with many test configurations, only to realize later on that they need a 10x more powerful machine to accomplish their such a plan.

As you'll see, running automated tests takes a lot longer than running them locally, but that's sort of the point: they run on their own time, without you sitting around waiting for them. You check in your code, and move on: the server will do its thing at some point, and can email you

if there was a problem.

As your integrations complete, Xcode will build up information on what passed and failed, what your success rate is, how many commits you've had, what your code coverage is like, and more. This all gets shown in the Report navigator, which is the last navigator tab on the left of your Xcode window – Cmd+9 should bring it up. This usually shows just your build information, but now it will also scan your network for activate Xcode servers and their bots. If everything has worked correctly you should see your server and, inside it, your bot.

When you've selected a bot you'll see a graphical read out of how things are going: all the numbers I cited above, plus a few charts showing key information. If you were using Xcode Server as part of a team of developers, Xcode has a neat way to help you show all this information to your team at a glance: right-click on any bot, and choose View Bot in Browser.

This requests our bot using a simple web server that's built into Xcode Server. You will almost certainly see complaints that the connection isn't really secure, which is fair enough – you're just talking to a local computer, rather than a site with its own secure certificate. However, once you convince your browser to let you see the site you should click the Big Screen button in the top-right corner. This will show you all the bots you have configured, along with how many build errors, warnings, analyzer problems, and test failures they had. There's even a full-screen button you can press here, to make the thing take up all available space – it's perfect for teams that rely on Xcode Server for CI.


Core Bot 11 hrs ago
0
0
1
of 2
0% Coverage
Unwrap Bot 8 min ago
0
0
1
of 52
0% Coverage

# Using a third-party service

As you've seen, Xcode gives us a huge amount of control over our CI system: what it does, how it works, when it works, and more – all privately on a local Mac.

However, it's fair to say that most companies prefer to use online services for their continuous integration needs. These integrate deeply into GitHub, allowing you to make sure all tests pass before merging pull requests, for example.

There are lots of these companies out there, and I don't wish to endorse a particular one. However, clearly I need to use *something* otherwise this chapter will be utterly abstract. So, for the purposes of demonstration we'll be looking at CircleCI: **https://circleci.com**. This is *not* free for macOS users, however if your project is open source then you can contact them to get a free account.

So, if you're an open-source project you get all the benefits of CI, plus GitHub integration, all for free. The downside, of course, is that your work must all be public, so if you just want to noodle around with CI to get the hang of it, you might not like Circle.

## Setting up

If you don't already have a CircleCI account, the first step is to make one: go to **https://circleci.com** now.

**I strongly recommend you create your account using GitHub, because it will automatically configure both GitHub and CircleCI so they can communicate.** This saves you having to do the work yourself, so please use this approach.

As I said, CircleCI does have a free tier for open source projects, but you need to email their team to ask them to enable this for you. Seriously, just write an email to **billing@circleci.com** giving them your username on GitHub and your repository URL, and they'll set you up with free access – this isn't some sort of elaborate prank! Alternatively, try tweeting **@circleci** with the same information.

Because CircleCI is able to talk to GitHub through your account, after log in you'll see a message like this: "No projects building on CircleCI. Skip to set up new projects" – look for the Add Projects button on the right, and click that. You should see a list of all your GitHub projects, public or private, so find the project you want to configure and click Set Up Project.

As this will be a macOS project, and as you only just sent in your request for a free plan, the best thing to do at this point is select one of their paid-for plans – choose the Growth plan for macOS, because it comes with a two-week free trial. Don't worry: you won't be asked for credit card information, so there's no chance of you incurring fees by accident. Selecting the free trial of a paid-for plan lets you continue while the billing team at CircleCI validates your email request.

Once you have a plan set up and working – free or paid trial – the CircleCI site should give you a comprehensive set of instructions to follow to get things moving. Here's the condensed form of their instructions:

1. Make a directory in your project called **.circleci** – yes, that starts with a period. macOS considers directories that start with a period to be hidden, so you can create this one by running **mkdir .circleci** using Terminal.
2. Now run the command **touch .circleci/config.yml** to create a configuration file in there.
3. Run **open .circleci/config.yml** to open config.yml in a text editor – this is the configuration file that CircleCI will look at when working with your project.
4. Click the Copy To Clipboard button in the CircleCI instructions. This will copy their default configuration data to your Mac's clipboard, and you should paste that into your config.yml file.
5. Change their Xcode version to be 10.1, or whichever version matches your local version.
6. Change the **SCAN_DEVICE** line to "iPhone 8", then change **SCAN_SCHEME** to be the name of your Xcode target – this is probably your project name.
7. Save that file, then run **git add .** from your terminal to add that configuration file to Git's staging area.
8. Now run **git commit -m "Adding CircleCI"** then **git push** to commit that change and push it to GitHub.

9. Click the Start Building button in CircleCI.

Now leave CircleCI to do its work: it will check out your source code from GitHub, fetch any CocoaPods used in your project, build the whole thing, run your tests, and report back how it all went. At the end, you'll automatically get an email showing you what went wrong if any failures occurred.

At this point, CircleCI is configured to watched our project, which means that every new commit in the future will automatically trigger a new test run. If you want to try this for yourself, wait until the previous test finished then try making a minor change to your code. As soon as it's pushed, you'll see your CircleCI dashboard update to say another test is running.

## GitHub integration

At this point CircleCI is performing *reactive* checking of code changes, rather than *proactive* checking – we learn when our build breaks rather than stopping such a thing from happening, and it's preferable to avoid that entirely where possible.

The smart way to fix this is to lock down the master branch of your GitHub repository, which stops anyone – including you – from pushing code directly to it. Instead, everyone will need to make a branch, push changes there, then open a pull request to merge this back with master.

This is where third-party source control services like CircleCI start to really shine: they can integrate directly into GitHub so that anyone opening a pull request in our repository will automatically have their code checked. We can then make sure all tests pass on CircleCI's servers before merging to master becomes possible.

Here's how to set it up:

1. On the homepage of your GitHub repository, go to the Settings tab then look for Branches.
2. You should heading Branch Protection Rules, and below that a button saying "Choose A Branch" – click that button now.
3. Select master from the list of branches in your repository.
4. Select the checkbox marked "Protect this branch".

5. Select the checkbox marked “Require status checks to pass before merging”, then select “ci/circleci” in the list of checks.
6. If you want these rules to apply to you too, you should also select the checkbox “Include administrators”.

Click Save Changes, and enter your password to confirm your new settings. You can change these settings later if you want, but they are a good idea to start with.

We’ve just told GitHub that no one – including ourselves – should be allowed to push changes directly to the master branch of our project.

To try it out, make any change to your project and try using **git push** to send it up to master. You should get errors back saying “remote: error: GH006: Protected branch update failed for refs/heads/master” and “remote: error: Required status check "ci/circleci" is expected” – that’s GitHub’s way of saying it’s enforcing the rules we set.

Instead, you needf to push changes up to a remote branch, using something like this:

```
git push origin master:new_branch_name
```

That pushes our local master branch up to the remote “new_branch_name” branch – creating it as necessary.

CircleCI will detect the new branch and immediately start running our change through all the tests we have. If you refresh your GitHub repository page you’ll see “Your recently pushed branches” followed by the “new_branch_name” branch that was just created.

At the right side of that message you’ll see “Compare & pull request”, which starts GitHub’s process of merging your new branch with master. This is where you would normally add information telling your team what changed and why – click Create Pull Request when you’re ready

Without CircleCI, this is where you would normally see a green button saying “Merge Pull Request” that handles the process of merging the branch you created with the master branch,

but it's now going to be disabled – GitHub is refusing to let you merge the pull request because CircleCI's tests haven't finished yet.

This shows CircleCI doing exactly what it was designed to do: we can no longer break our build by merging in some bad code, because GitHub's merge button won't be enabled until CircleCI passes all our tests. GitHub will only enable its button when CircleCI completes its tests – wait for it to finish, then complete the merge and delete your branch.

At this point CircleCI is automatically running our tests and letting us know if any fail, and it's also automatically verifying pull requests so no one can break our build by accident – this is a huge step forward for project stability.

# Chapter 7

## Tips

# Upgrading to tests

At this point of the book, we've covered unit tests, test doubles, UI tests, test-driven development, continuous integration, and more – you know the full range of testing approaches and techniques that will help you do a great job on projects of all sizes.

In this last part of the book, I want to outline some advice to help you make the most of what you've learned, and the first topic I want to cover is how to introduce testing into a project.

You see, it's possible you've read through large parts of this book with a faint sense of despair: you like the sound of unit tests, UI tests, and more, but all along you've known that the codebase at your company is untested and quite possibly *untestable*.

Such a codebase can be challenging to work with: you tread carefully around code you know is complicated, you spend hours crafting patches that you hope will work in production, and you actively avoid Jira issues that you know will cause pain.

But before you decide to throw it away, let's see if we can take steps to improve. Yes, I know it's tempting to think a complete rewrite will solve your problems, and to be honest it *might*. But there's also a big chance you'll spend months working hard, only to replace your long list of known bugs with a long list of unknown bugs, and that's *worse*.

## Start with bugs

The thought of going from no tests to 100% coverage in an existing project fills many people with dread – and quite rightly too. In fact, this kind of thought process often leads to people continuing to write no tests, because "the code is already in such a bad state it wouldn't help much."

This isn't true – far from it. My all-time favorite Graham Lee quote is this: "Each test you add will prove that one more little piece of your app is working as you expect, and the first test takes you from no proof to one little piece of proof – an infinite enhancement." (Lee, G., *Test-Driven iOS Development*, 2012)

I love how encouraging this is – how often do you get to say that you've gone from nothing an infinite enhancement? Almost never, that's how often. So if you're imagining that your current project is somehow beyond hope for testing, please practice a little self-compassion and realize it's never too late to make up for past mistakes – you retroactively claim many of the benefits from testing, although I admit it might take a little more work than if you had started earlier.

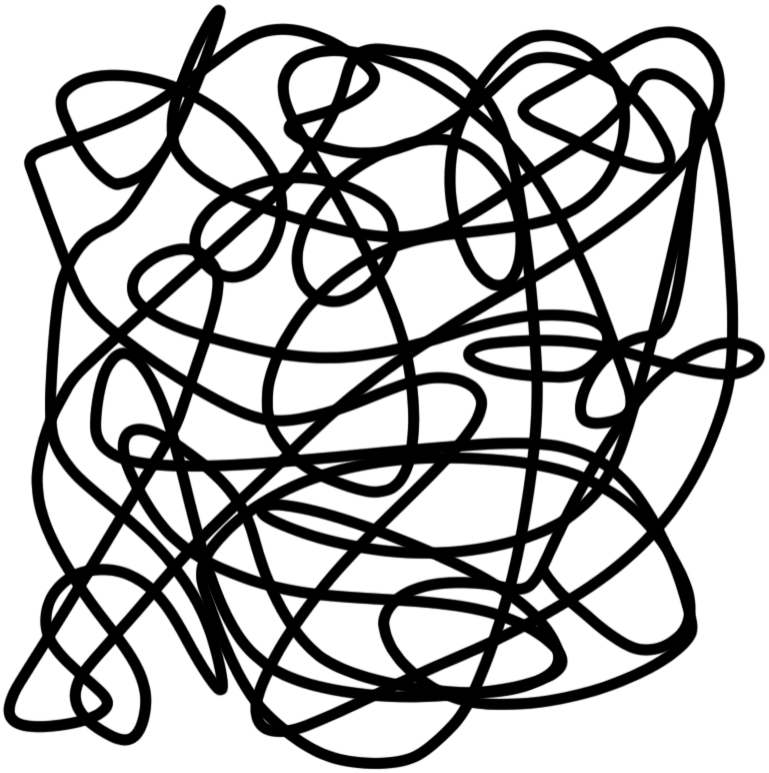
Time after time, I've found the easiest place to begin is the next time you're assigned a bug to fix. It doesn't matter how big or small it is: it's a bug, which means you have some code that isn't behaving the way it should. This makes for a perfect starting point, because a clear behavior we can test: when method X is called we expected it to return Y, but instead it returned Z.

So, you know the code is wrong, which means you can write a test for that: `testXReturnsY()" is trivial to write, but will immediately fail because it will get back a Z. When this test fails – and it certainly ought to – you have hard proof that you've understood the bug enough to isolate it neatly, and can be sure you know when the bug is fixed because your test will pass. Even better, that test will remain in your code going forward, which means it will help catch *regressions* – it will stop the bug from coming back in the future.

If you *really* want to make this work, get into the habit of connecting your code, commits, and Jira IDs. So, when you write a test that identifies a bug, add a comment before it with the Jira ID so you can look back at the original report in the future. Reference the Jira ID in your Git commit message, then update the Jira ID to say when the bug was fixed. These connections allow you to go from Jira to commit, or code to Jira, etc, because each of these things provide a different part of the story going forward.

## Pulling code apart

When I was a kid, one of my favorite jokes was to scribble over a piece of paper and ask someone "what's that a picture of?" They might guess "a scribble" or "modern art", but my hilarious-for-an-8-year-old punchline was "it's an explosion in a spaghetti factory."



Comedy gold, right? But that "joke" wouldn't work today, because for many developers that scribble looks more or less like their architecture diagram and might cause their involuntary eye twitch to return.

Don't take this personally: if you think your code is particularly entangled, you're not alone. In fact, Joe Armstrong, the creator of Erlang, once wrote that "the problem with object-oriented languages is they've got all this implicit environment that they carry around with them. You wanted a banana but what you got was a gorilla holding the banana and the entire jungle."

And he's right: although OOP has helped teach millions structured software development, it can often lead to a situation where each piece of code relies on some other piece of code – a situation known as *tight coupling*.

Tightly coupled code is hard to test, because we want things to be isolated from other parts of our system. If component A relies heavily on components B through Q, writing a test for just A will be hard if not impossible. Yes, you can attempt to create mocks, but do you really want

to mock that many things?

When writing software with TDD, you naturally write tests as you go and so structure your code to be testable. When you approach the problem in reverse, trying to add tests after writing the code, you might find it's almost impossible to isolate a single component simply because they weren't designed to work in isolation.

This can be a tricky problem to solve, because it can really feel like there's no simple entry point to bring about change. The key is to start *small*, so here are some tips:

1. Take a method that reads state somehow – such as reading a property in its struct – and make that value be passed in as a method parameter.
2. Take a method that *modifies* state somehow – such as adjusting a property in its struct – and make that change be passed back as a return value.
3. Find a method more that's more than 100 lines of code and break it up into four smaller methods. Leave the original method intact and just make it call the four smaller methods internally.
4. "Whenever you are tempted to type something into a print statement or a debugger expression, write it as a test instead" – Martin Fowler

We'll be looking more closely at writing testable code shortly, but those four are relatively easy places to start. Try finding an example from the list above, break it apart, then see if you can write a test for it – that's the ultimate test of your success.

This approach lets you chip away at a large, complex codebase piece by piece. Each test you write adds another small piece of certainty that your code behaves as you expect, and by reducing the complexity of your code you're also making it easier to understand.

## Getting to tested

The key thing here is that you're making incremental improvement. I know it's tempting to say "we're going to stop everything and spend the next four weeks writing tests," but that's surprisingly hard to do. Years ago I used to work as a journalist for a Linux magazine, and I

published an interview with one of the core developers of KDE – a Linux desktop environment perhaps best known to us as the originators of the WebKit engine that now powers all modern browsers. We asked him why KDE releases focused so heavily on adding new features rather than fixing bugs or tidying up existing software, and his response was remarkable: "we have some developers who would prefer to write nothing if they weren't able to write new features."

It's common for all development teams to have a similar problem: tackling new challenges is *fun*, coming up with solutions to problems is *exciting*, delivering user-facing features is… well, you get the point. But fixing bugs, documenting code, and adding tests? They are just as important, but significantly less attractive to most of us.

Because of this, it's just not possible to ask folks to focus on testing for an extended period of time. A one-day hack session can work, not just in boosting tests but also helping reframe the focus of developers to think of testing as more important, but if you start going beyond that interest wanes quickly – developers will be less focused and more demoralized, and that's not going to get you where you want.

# Writing testable code

Marco Contino gave me this wonderful summary of the problem we're facing: "Always assume your code is not working – prove it otherwise."

When code is in a great state, writing tests that prove your code is working is easy – in fact, it can be so easy that they feel almost *obvious*, because of course calling function A will always return B. But *getting* to that point is easier said than done: we all make choices every day, and sometimes the culmination of those choices lead us to code that's hard to write tests for.

In this chapter I want to walk you through a number of tips that will help you write testable code, because ultimately that's the fastest way to make writing tests easier. If you're able to structure your code with these tips in mind, you will find your unit tests easier to write and more reliable, as well as massively reducing coupling in your code, so it's a win all around.

## Use dependency injection

Some years ago, the Demeter Project outlined its Law of Demeter, which has since become known as the principle of least knowledge:

- Each unit of your code should know only about other units that are closely related to it.
- Each unit should talk only to those units, not to units that aren't related.

This will seem familiar to you as dependency injection: we strip away hidden dependencies and pass them in explicitly, making clear the relation between units of our code. So, if a method you write calls out to **UIApplication** or **UIDevice**, for example, these are effectively strangers to that method – code that isn't related. If you pass in those using construction injection, the relationship becomes clear.

As I've mentioned previously, it's usually a good idea to provide sensible default values for construction injection to avoid making your call sites a mess.

## Avoid state where possible

State is something that comes naturally to object-oriented developers: we define a class called **Dog**, and of course that has **name** and **breed** properties. I'm not saying we should eliminate *all* state: just take a step back and think "is there a better way of storing this?"

I talk a lot about functional programming in my book Pro Swift, and I don't really want to repeat that here. However, I do at least want to reiterate the importance of having fixed input and output for your functions: if you give a function A, B, C it should always return 1, 2, 3. It doesn't matter what state your program is, the value of your properties, or what code has already been run – that code will always do the same thing.

These functions, known as *pure* functions, should form the backbone of your code for a number of reasons:

1. You can call functions in any order without worrying about what else might have changed.
2. You can combine small functions into larger ones, knowing that the combination won't produce any surprises.
3. You can write trivial tests because the output will never vary.

## Treat singletons with care

Singletons make a lot of sense in some specific situations, but that doesn't mean a) they are your first port of call, b) they are even your tenth port of call, or c) you need to expose them in your code.

Singletons are effectively *global state in sheep's clothing* – they are global variables wrapped up in some accessors, and often become hidden dependencies that would be better off replaced with dependency injection.

In my book Swift Design Patterns we look at a protocol-oriented approach to hiding singletons, but ultimately you should be questioning their need in the first place. Analytics and logging are often given as singleton examples, and I agree they both make sense. However, for most other places you should be looking for alternatives: adding something as a property that can be injected isn't hard to do, and clarifies the relationship between your objects.

## Watch out for side effects

Functions are considered to have side effects when they do something that you wouldn't expect by looking at their signature. For example, if a method with the signature **sum(numbers: [Int])** also happens to check whether the current user is authenticated, that's a side effect – something we wouldn't expect.

Of course, that's a rather *obvious* side effect, so let's look at something more subtle: what if **sum(numbers:)** writes that sum to a property in the struct it belongs to? This is still a side effect, and still likely to cause problems: you have mutated your test environment, which means you might introduce subtle problems in your code and/or tests.

The definition of a side effect is a little hazy, and that's just something we need to accept and work with. For example, if **authenticate(user:)** returns true or false depending on whether a user's credentials are valid, but also writes to a log file with the result of the check, is the logging behavior a side effect or just part of the method?

There's no simple answer so you'll need to draw that line yourself, and in particular try to come to a pragmatic conclusion that ways the effects of the other tips presented here.

## Manage your dependencies

Switching from hidden dependencies to injection is a huge step forward for most projects, but it's sometimes easy to go too far in a bad direction: you end up with many properties for a type, and initializing them all using construction injection is problematic – yes, even with default values.

Solving this problem starts by figuring out why the type has so many dependencies in the first place. In my experience there are usually two reasons:

- The class does a number of different things, and needs to pull in data from lots of places.
- The class creates a variety of other objects that need those dependencies, so it needs to have those ready to pass on.

These problems manifest themselves in the same way – a type with lots of properties – but they have different underlying causes and different solutions.

In the first instance, our solution lies in the S of SOLID: the single-responsibility principle, which is the idea that each part of our code should be responsible for one thing. Clinging to this dogmatically will cause problems, but more people have the opposite problem – we make types do so many things that we can't explain all their functionality in one breath.

Some developers take this a long way, saying that if you need to use the word "and" to describe what your type does, break it up. Again, I encourage you to avoid dogma here: the single-responsibility principle is a guideline to help us structure our thinking, not a law that results in prison time when you don't follow it from time to time.

In the second instance, we have a dependency pyramid:

1. View controller E needs a property.
2. View controller D needs a property, but it will also show view controller E so it needs to have E's property too.
3. View controller C needs a property, but will also show view controller D, which in turn will show view controller E, so it needs to have all the properties for those too.
4. View controller B… well, I think you've got the point by now.

This is a dependency pyramid: whatever lies at the base of the pyramid must support everything above it, even if it doesn't need all the information itself.

The solution here is to take those properties away and store them somewhere else. We already looked at coordinators back in the Test-Driven Development chapter, but it's just one possible solution to this issue – using view controller containment can also work well, for example.

The goal here is to simplify your types: don't overwhelm them with tasks, and certainly don't give them dependencies they don't actually use.

## Measure your complexity

We have a dedicated metric for measuring the complexity of functions, and it's called *cyclomatic complexity*. This measures how many edges (*E*), nodes (*N*), and connected components (*C*) we have in a given function, and applies a simple alglorithm: your cyclomatic complexity is equal to **E - N + 2C**.

Put more simply:

- A connected component is when one part of your code moves onto another with no special logic.
- A node is a decision point in your code – a loop or a condition, for example.
- An edge is the flow of code: if a condition is true or false, for example.

To make it even more simple, if your function has no branching in it at all – if it just flows linear from top to bottom with all code being executed – then it has a complexity of 1, and every time you see a condition or a loop, add 1 to that number. (In practice things are significantly more complicated – think about **didSet**, subscript methods, KVO, and more!)

I realize this is all very abstract, and unless you enjoy graph theory it's unlikely you'll want to calculate your cyclomatic complexity by hand. So, let's look at something more concretely useful:

- If a function accepts a boolean parameter and uses that to decide which value to return, it will run through two possible states: one when the boolean is true and one when it's false.
- If the same function accepts two boolean parameters, then it could have as many as four states: true/true, false/false, true/false, and false/true.
- If we extend it to four booleans, now it could have as many as sixteen states – and these are just booleans!

Now, I've said "could have as many as" because *how* those booleans are used matters, because there's a subtle but important distinction between *branch* coverage and *path* coverage.

The classic example of this can be seen here:

```
func process(user: String, uppercase: Bool, addGreeting: Bool)
```

```
-> String {
    let processedUser: String
    let result: String

    if uppercase {
        processedUser = user.uppercased()
    } else {
        processedUser = user
    }

    if addGreeting {
        result = "Welcome, \(processedUser)!"
    } else {
        result = user
    }

    return result
}
```

That accepts a string plus two booleans, and returns one of four values. For example, if the user was “Paul”, it could return:

1. Paul
2. PAUL
3. Welcome, Paul
4. Welcome, PAUL

Branch coverage is determined by how much of our conditional branch code is executed, and in this case we need to write only two unit tests to get 100% branch coverage:

```
process(user: "Paul", uppercase: true, addGreeting: true)
process(user: "Paul", uppercase: false, addGreeting: false)
```

Those two calls will exercise both cases of both conditions. However, it's clear from our return values that we aren't testing all possible responses, which is where *path coverage* comes in: we have exercised only two of four possible paths through the code, missing both true/false and false/true.

Once again, there is no magic number to aim for here – no cyclomatic complexity value beyond which lies certain peril. Instead, try to see how far your path coverage varies from your branch coverage: if it's trivial to cover all branches but much harder to cover all paths, you should look to break things up.

## Constants rather than variables

This simple change will help you eliminate a raft of problems, mostly by enforcing making it easier to enforce the other guidelines here. It's common to see a property marked variable, set to a value in an initializer or something similar – perhaps **viewDidLoad()** in a view controller, for example – then never changed again.

This kind of code makes it possible for code to behave unpredictably: that state can be mutated anywhere, so you're opening little holes that can cause problems later on. If you're able to take those variables and make them constants, your code will now behave much more predictably: once your code is set up, it's not going to change by surprise.

## Make impossible conditions impossible

Here's a well-used method from Foundation's **URLSession** class:

```
func dataTask(with request: URLRequest, completionHandler:
@escaping (Data?, URLResponse?, Error?) -> Void) ->
URLSessionDataTask
```

That expects us to pass a completion handler closure as its second parameter, and that closure must accept three parameters: an optional **Data**, an optional **URLResponse**, and an optional **Error**. So, it might give us some data, a response, and an error, or any combination of the

three – even though it's clearly impossible to get back both a valid result and an error.

The problem is that Objective-C didn't have a way to send back an either/or state – something we would call a *sum* type if we were algebraically inclined. Swift has always had the **Optional** type, which lets us represent two possible states: having a value or not. However, if you're lucky enough to be using Swift 5.0 or later you'll have access to the **Result** type instead, which does much the same thing: we either get back a valid value, or we get back an error of some sort.

In code, it looks like this:

```
public enum Result<Success, Failure: Error> {
    case success(Success)
    case failure(Failure)
}
```

This precludes the possibility of getting back both a success and failure, which makes your tests easier to write.

**Note:** If you aren't lucky enough to be using Swift 5 or later, try one of the open-source **Result** alternatives such as **https://github.com/antitypical/Result**.

## Trap on bad state

When you get right down to it, bugs are effectively our code behaving differently from how we expected it to behave: we think doing X will return Y, but instead it returns Z.

Often you'll see folks add **print()** calls to their code as part of their debugging process, verifying that their program's state is what they expected. But what they are doing is adding a milestone: "at this point, I expect the array to have 5 items" or "I expect the number to be over 10." This is sensible, but it has a problem: we need to evaluate these milestones by hand, reading the output of **print()** to make sure it is what we expect.

A better idea is to use the **assert()** function, which is a milestone that Swift evaluates on our

behalf. You can all **assert()** whenever and as often as you want: if the condition you're checking turns out to be false, Swift will crash your code and print a message of your choosing.

That might sound bad, but:

1. If your program somehow ended up in invalid state, you have a fundamental problem in your logic.
2. The app will only crash in debug mode – i.e., when you're running from Xcode. When you ship release apps to the App Store the crash will no longer happen.
3. In fact, Swift completely compiles out assertions in release mode, so any slow checks you run won't exist at all in release code – they don't happen at all.

You can, and should, add assertions throughout your code. Again, they have no impact on your shipping code, but they help you ensure your code is working as it should – they eliminate a whole raft of possibilities that should be impossible, allowing you to focus on writing tests for the things that matter.

## Simplify your surface area

Your tests should check the full range of functionality your code exposes, but that doesn't mean you need to test all the code directly. Instead, spend time trying to cut back on the number of properties and methods that are exposed publicly: mark them as **private** or **fileprivate**, then you can focus on testing the parts that are actually exposed.

Remember, we're testing behavior, not lines of code: I don't really care if a public method internally calls five private methods, only that it ultimately returns what I expect. So, pass values into your type using a constructor, and let it do with them whatever it needs to – we don't need to poke around in there afterwards.

## Separate your concerns

When one unit of code attempts to mixes two different responsibilities at the same time, it

inevitably results in problems. This commonly takes two forms:

1. Blending our application logic with some code triggered by a framework such as UIKit.
2. Mixing view code with model code.

These are effectively the same thing, although they manifest themselves in different ways. The former looks like this:

```
func webView(_ webView: WKWebView, decidePolicyFor
navigationAction: WKNavigationAction, decisionHandler:
@escaping (WKNavigationActionPolicy) -> Void) {
    if let host = navigationAction.request.url?.host {
        if allowedSites.contains(where: host.contains) {
            decisionHandler(.allow)
            return
        }
    }

    decisionHandler(.cancel)
}
```

That's a WebKit delegate callback, asking us whether a specific URL is safe to load. Inside the code we check whether an **allowedSites** array contains the URL, and allow or deny the request based on that – it's trivial logic, but it would still be nice to write tests passing in some good and bad sites to make sure it works.

Even though this code is purposefully simple, it contains a fundamental problem: to test that method we must create **WKWebView** and **WKNavigationAction** instances, when really we just want to validate that our URL-checking works as intended.

The smart solution here is to pull this code apart into two methods, so that your application logic can expand freely while still remaining testable – separating logic from effects. For example:

```
func isAllowed(_ host: String) -> Bool {
    return allowedSites.contains(where: host.contains)
}

func webView(_ webView: WKWebView, decidePolicyFor
navigationAction: WKNavigationAction, decisionHandler:
@escaping (WKNavigationActionPolicy) -> Void) {
    if let host = navigationAction.request.url?.host {
        if isAllowed(host) {
            decisionHandler(.allow)
            return
        }
    }

    decisionHandler(.cancel)
}
```

The same problem hits us in a different way when we mix our *own* model and presentation layers. Yes, our methods might be slightly easier to call, but the problem is the same: to test pure logic, we need to create and manipulate views.

In my book Swift Design Patterns I looked at a number of ways to break up functionality so this problem was less likely, but the abridged version is this: put your nouns (what objects you have) in models, your verbs (make things happen) in controllers, and your adjectives (describing how things look) in your views. If you find you start adding action code to your views, you've mixed up your concerns and tests will be harder to write.

# Final advice

Now that we're near the end of the book I hope you're feeling ready to get out there and start building a healthy culture of testing in your workplace – even if it's just you in a home office.

But before we're done, I have a handful of small tips I want to drop in – things that wouldn't fit elsewhere into this book, but are still worth mentioning.

**Test your code like you hate your code.** Putting to one side all your unit tests, integration tests, UI tests, and more, when you finally have a build of your app running on a device, you need to whack it with everything you've got.

I like Bill Sempf's joke: "A QA Engineer walks into a bar. Orders a beer. Orders 0 beers. Orders 999999999 beers. Orders a lizard. Orders -1 beers. Orders a sfdeljknesv." This is *fuzz testing* and it forces you to make your code rock solid.

I once published an interview with Alan Cox, who was a core developer on the Linux kernel. He described how he tested the Linux networking stack by plugging in cables randomly until something broke!

**Try putting your test code next to your app code.** It's easy – and indeed common! – to put your test code in a different Xcode group, but that can sometimes cause more problems than it solves. Try rearranging your project so that your tests are organized directly next to the code they are testing: you'll be able to see more clearly which parts have tests, and it makes it trivial to open both the app code and test code side by side in the assistant editor.

**Prototype tests in playgrounds.** Just like regular application code, you can write tests in Xcode's playgrounds if it helps you. Start by importing the XCTest framework, go ahead and create your test class as a subclass of **XCTestCase**, then call **YourTestClass.defaultTestSuite.run()** to have all the tests run.

For example:

```
import UIKit
```

```
import XCTest

class ExampleTest: XCTestCase {
    override func setUp() {
        print("Setting up")
    }

    func testPlaygroundsRock() {
        print("Running test")
    }

    override func tearDown() {
        print("Tearing down")
    }
}

ExampleTest.defaultTestSuite.run()
```

Many people find that working in playgrounds lets them iterate faster – give it a try and see what works for you.

**Customize behavior with environment variables**

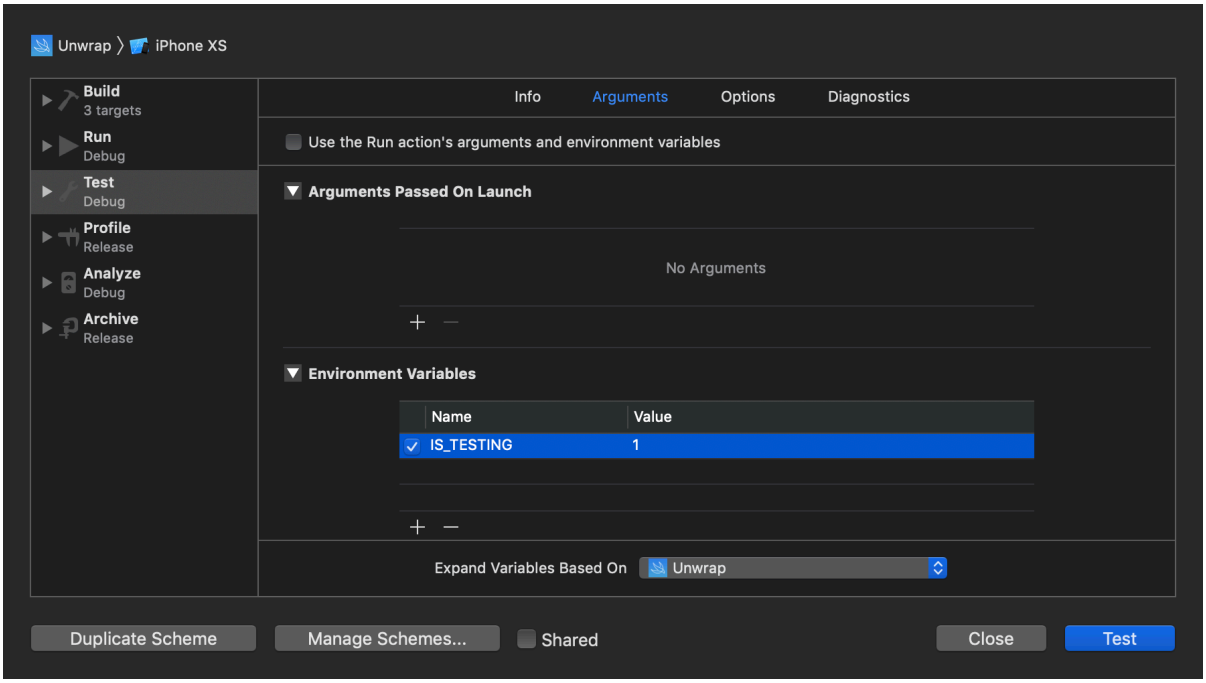
Xcode lets you set environment variables for each of your schemes, then read those schemes to customize the way your code works. For example, you might want to set an **IS_TESTING** environment variable for your Test scheme, then add some Swift code to detect that and reset user settings at launch.

To try this out:

1. Hold down the Option key, go to the Product menu, and choose "Test…" This will let you customize the way the Test scheme executes.
2. Go to the Arguments tab, then uncheck the checkbox marked "Use the Run action's

arguments and environment variables" so that we can specify custom environment variables.
3. Click + under the Environment Variables heading, to add a new variable.
4. Name it "IS_TESTING" and give it the value "1".

That will set the IS_TESTING environment variable only when unit tests are being run.




We can now detect that in code like this:

```
if ProcessInfo.processInfo.environment["IS_TESTING"] == "1" {
    print("We're in test mode")
} else {
    print("We're not in test mode")
}
```

**Avoid flaky tests at all costs**. If you have a test that sometimes passes and sometimes fails, rewrite it so that it's continuously successful, fix the underlying code so that the flakiness never happens, or remove it. Flaky tests – tests that are unreliable indicators of the health of your code – do more damage than good, and should be fixed or excised.

Common problems include:

- The environment is changing around you, and you aren't properly compensating or controlling it.
- You aren't testing a code unit in true isolation, and something else is getting in the way.
- A cache is storing an earlier value that pollutes your results.
- You're using multi-threading and have a race condition.

If you have a flaky test, try checking each of those to see if you can nail down the problem and get it resolved.

**Test your utility code.** Previously I mentioned Gerard Meszaros's *creation methods* and *verification* methods: helpers that encapsulate the creation of a test harness and wrap multiple assertions to make testing easier.

These methods are important and helpful, but they are code in their own right and should be tested as such. Yes, I'm saying you need to write tests for your tests.

**Finally, don't assume Xcode is as good as it gets.** I love Xcode a lot and have been using it for a very long time, but it's far from the cutting edge of testing. Many well-respected testers, particularly those fond of test-driven development, prefer to use AppCode instead, largely because it supports TDD's natural way of working: write a failing test, then write the code to make it succeed. It literally looks at the code of your failing test, figures out what production code needs to be written to make it pass, and helps create skeleton versions – it's really impressive.

If you've never seen it before, give AppCode a try – you'll be surprised how effective it can be. This isn't me suggesting that everyone ditch Xcode – I just encourage you to give it a try and come to your own conclusion.