HACKING WITH SWIFT

TESTING SWIFT

COMPLETE TUTORIAL COURSE

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

FREE SAMPLE

Paul Hudson

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:

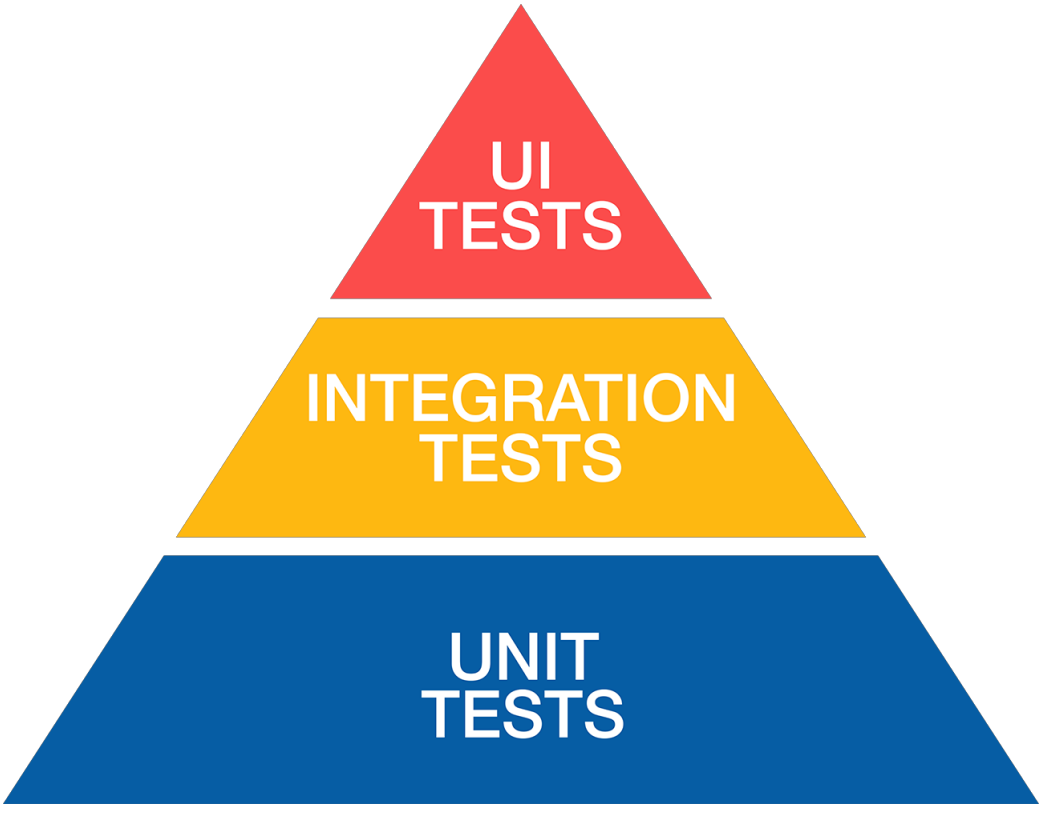
1. The technology itself is less mature, so it might take more work to make your tests pass.

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.