Extracted from:

iOS Unit Testing by Example

XCTest Tips and Techniques Using Swift

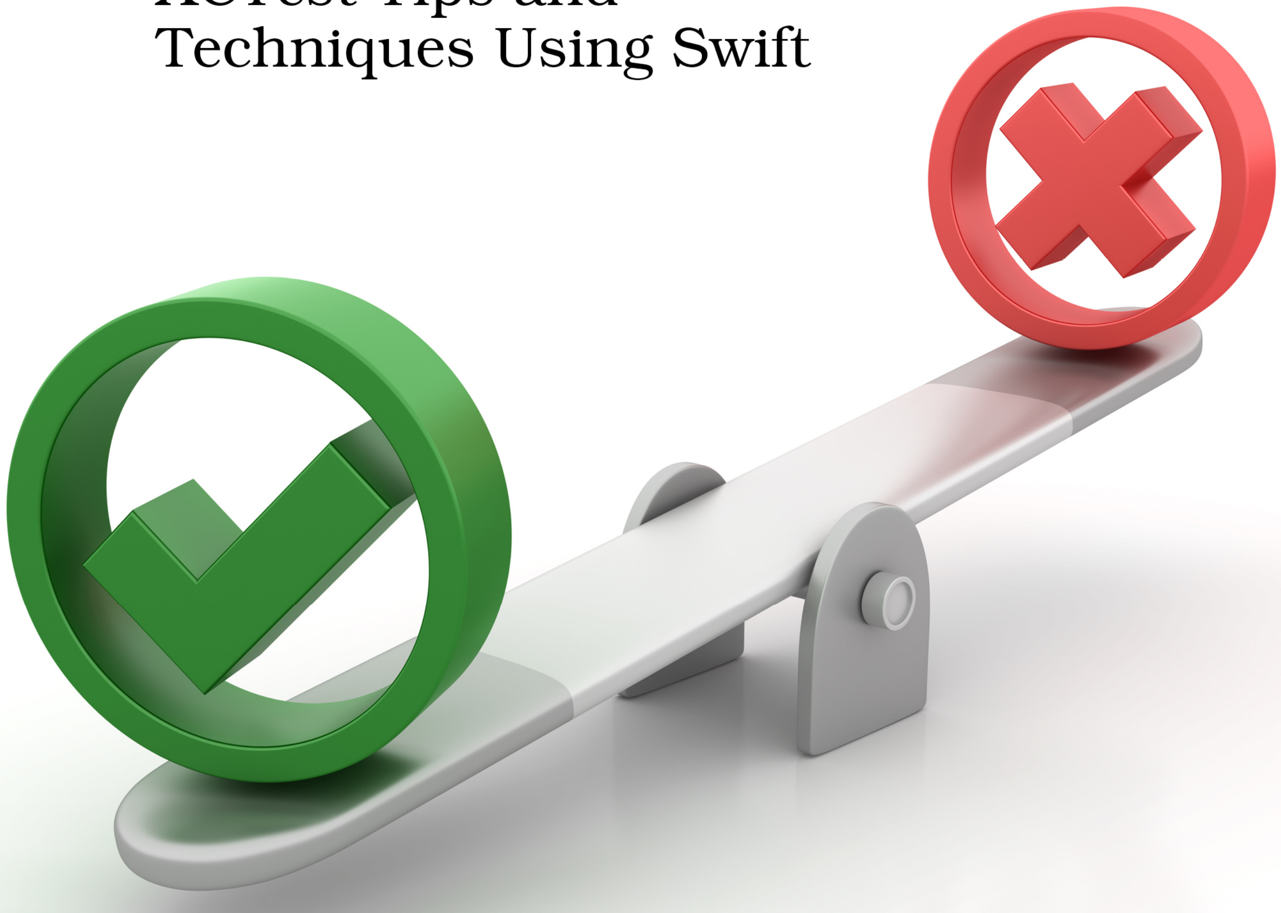
The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

iOS Unit Testing by Example

XCTest Tips and Techniques Using Swift



Jon Reid

edited by Michael Swaine

iOS Unit Testing by Example

XCTest Tips and Techniques Using Swift

Jon Reid

The Pragmatic Bookshelf

Raleigh, North Carolina

The team that produced this book includes:

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

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

For Kay, who believed in me

CHAPTER 6

Manage Difficult Dependencies

When you realize you can write unit tests against view controllers, it's exciting. At first, you may think this will unlock your codebase to automated testing: "I can test anything!"

Unfortunately, as you try to make progress, you'll experience setbacks and frustrations. Sure, you can write a test that accesses a particular view controller. But as soon as you try to have a test call some method, you find the code inside the view controller is fighting you.

This is true of any types, not only view controllers. Code written without tests often has implicit hardwired dependencies. These dependencies can complicate testing. It so happens that view controllers are especially susceptible to such problems. It's easy to lump functionality (and the dependencies needed to perform it) into a view controller.

When testing is difficult, this reveals flaws in the architectural design of the code. By making changes to enable testing, you'll be shaping the code into cleaner design. Design decisions that were once hidden and implicit will become visible and explicit.

In this chapter, we'll learn how to identify difficult dependencies. Having identified them, we'll explore some techniques for isolating these dependencies. This will give you ways to write unit tests against previously untestable code.

Be Okay with Problem-Free Dependencies

Law enforcement agents learn how to detect counterfeit money by studying genuine money. Let's apply this idea to dependencies. Before looking at difficult dependencies, let's see what makes some dependencies problem-free.

Consider the following function:

```
func shoutHello(to name: String) -> String {
    return "HELLO, \(name.uppercased())!"
}
```

What dependencies does it have? This is a trick question because it's easy to reply, "It has no dependencies." But it *does* depend on the Swift String type. Our function calls the uppercased() method. Swift takes the result and does string interpolation. String interpolation prefers calling the description property of the CustomStringConvertible protocol.

There are word lawyers who insist that a test is not a unit test if it exercises more than one type. Yet no one would blink an eye at writing tests for shoutHello(to:) and calling them unit tests. So why is this dependency okay to test without isolating it?

To answer this, let's start with three of the FIRST unit test principles.[1] The first three apply to dependencies:

F for Fast Both functions—the uppercased() method and the description computed property—are fast. We're not in any danger of pushing up against the rule of thumb from *Working Effectively with Legacy Code [Fea04]*:

> A unit test that takes 1/10th of a second to run is a slow unit test.

I for Isolated Neither function has any side effects that would persist beyond the test run. Tests that exercise shoutHello(to:) won't get different results due to external factors. And the tests won't have any effect on each other.

R for Repeatable Calling these two functions with the same input will always yield the same output. There are no external services that might fail. There are no race conditions. The time of day (or phase of the moon) will make no difference.

(The last two of the FIRST principles don't apply to dependencies. So you're not left wondering what they are, S is for self-verifying. This means using assertions to pass or fail without human verification. And T is for timely. This means tests have more value when written before the production code.)

After fast, isolated, and repeatable, there's one more question that helps us classify dependencies.

Easy to Test? When something calls a dependency, how can we know if the call was correct? If there's a return value, it's easy. We can check the

1. https://pragprog.com/magazines/2012-01/unit-tests-are-first

return value, or any computation that uses the return value. For shoutHello(to:), the calls to String's uppercased() and description affect the function's return value. Tests can simply check the return value.

What if there is no return value? Chances are good the call causes some state to change. If we can check a property of the dependency for an expected value, that's also easy.

But if a call has an external effect we can't access, that dependency is harder to test.

If we take fast, isolated, and repeatable and combine it with easy to test, we get FIRE. If a dependency satisfies the FIRE rules, we can use it as is. Writing tests with it won't be difficult.

Identify Difficult Dependencies

Now that we have some rules to gauge if a dependency is problem-free, let's break each FIRE rule. This will help us learn which kinds of dependencies get in the way of simple tests.

F for Fast iOS programs often include code that will execute in response to some external trigger. In later chapters, we'll see how to unit test delegate methods. But if there's no way for tests to trigger the code execution immediately, that's a slow dependency. Examples include the following:

- Calls to web services
- Timers

I for Isolated Dependencies break the rule of isolation in two common ways: global variables and persistent storage.

Global variables come in different varieties:

- Variables defined outside of any type
- Singletons
- Static properties

Globals aren't a problem if they're read-only, such as string constants. It's when we can change the value of a global that we run into the challenges of *shared mutable state*. One test can set a value that affects a following test.

Persistent storage is similar, except that we store the state in something that outlasts the app's life cycle. This includes the following:

- The file system
- UserDefaults
- The keychain
- A local database
- A remote database

Recall from Chapter 2, Manage Your Test Life Cycles, on page ? that we need each test to run in a clean room. Earlier test runs or manual testing should not change the outcome of automated tests. And automated tests should leave no trace that affect later manual testing.

R for Repeatable What dependencies are there that yield different results when called? We expect different results for the following:

- Current time or date
- Camera or microphone input
- Face ID or Touch ID
- Core Motion sensors
- Random numbers

We can anticipate those differences. But there are also unpredictable differences:

- External services—they can fail.
- Writing to a log file—we can run out of disk space.
- Time zone of the machine running tests—when writing tests, it's easy to assume they'll always run in your own time zone. Hidden problems will surface if your development team grows globally.

Easy to Test? It's not hard to test functions that return values or change properties. But there are also functions that cause side effects outside of the invoked type. Such dependencies take commands but offer no way to access the effects of those commands. Examples include the following:

- Analytics
- Playing audio or video

Analytics includes any system of logging events to a server. We can send events, but there's no way for the mobile API to ask for the last batch of events you sent.

This isn't a complete list of difficult dependencies. But they illustrate guidelines that will help you identify most of them. Next, we'll see how to isolate them.

Create Boundaries to Isolate Dependencies

Once we've identified dependencies that make testing difficult, what do we do with them? We need to find ways to isolate them behind boundaries. Having isolated them, we can replace them with substitutes during testing.

In well-structured code, we can summarize our code as boxes of functionality. An arrow line from one box to another represents a dependency. With careful design, these boxes and arrows form a *directed acyclic graph*. By avoiding cycles, we make it easier to replace functionality. This brings benefits to ongoing maintenance that extend beyond testability.

We can implement boundaries using Swift protocols. With protocols in place, we can substitute different concrete types. But to even begin using a protocol, we need a place where we make the current type explicit. Once we spell out the type, we'll be able to switch it to a protocol.

There are various techniques for making dependencies explicit. To illustrate them, let's make another project for our experiments.

Make a New Place to Play

Now we're ready to create a new project for this chapter. Follow the steps for Create a Place to Play with Tests, on page ?, but name the project HardDependencies. Also delete that initial test file, HardDependenciesTests.swift.

We don't need to apply Chapter 4, Take Control of Application Launch, on page ? to bypass the app delegate, since this is an experiment. (This will be true for most of the book, so I won't continue to repeat this.)

To simulate a difficult dependency, let's pretend we're using an analytics API to track events. Make a new file in production code named Analytics.swift:

HardDependencies/HardDependencies/Analytics.swift

```swift
class Analytics {
    static let shared = Analytics()

    func track(event: String) {
        print(">> " + event)

        if self !== Analytics.shared {
            print(">> ...Not the Analytics singleton")
        }
    }
}
```

This API provides a shared instance to use as a singleton. As a "soft" singleton, it doesn't restrict us from creating separate instances.

Let's pretend the track(event:) instance method sends the event to a web service. We'll simulate it with a print(_:) statement, and observe the results in the console log. It also prints a message if the Analytics instance is not the singleton.

Besides an API we can't control, we'll also see an approach for singletons we own and can change. Make a second file in production code named MySingletonAnalytics.swift:

HardDependencies/HardDependencies/MySingletonAnalytics.swift

```
class MySingletonAnalytics {
    static let shared = MySingletonAnalytics()

    func track(event: String) {
        Analytics.shared.track(event: event)

        if self !== MySingletonAnalytics.shared {
            print(">> Not the MySingletonAnalytics singleton")
        }
    }
}
```

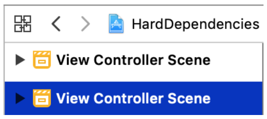
It's similar but wraps a call to the original Analytics class. We'll use this for Add Backdoors to Singletons You Own, on page ?.

Add Storyboard-Based View Controllers

To experiment with different techniques, let's make several view controllers. First let's add two view controllers to the storyboard. Select the HardDependencies group. Make a new file, selecting Cocoa Touch Class. Name it InstancePropertyViewController and make it a subclass of UIViewController. In the Save dialog, double-check that the app target is selected, not the test target.

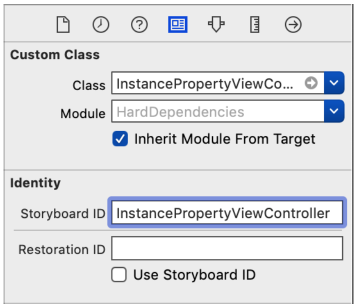
Now let's add this view controller to the storyboard. Open Main.storyboard and select *View* ▶ *Libraries* ▶ *Show Library* from the Xcode menu, or press Shift-⌘-L. This will bring up the Object Library. Double-click "View Controller" to add a new view controller to the storyboard.

This will create a generic view controller, which we need to change to our specific type. Select the second "View Controller Scene" that we just added, like you see here:



In the Xcode menu, select *View* ▶ *Inspectors* ▶ *Show Identity Inspector* or press ⌥-⌘-4. In the Identity Inspector on the right, the Custom Class section will show that the class of the selected view controller is UIViewController. (If it shows ViewController, that's the wrong one.) Click the down arrow for Class to reveal the pop-up menu, and select InstancePropertyViewController.

We're going to have a test load this view controller, so we need to apply the trick from Load a Storyboard-Based View Controller, on page ?. In the Identity Inspector, copy and paste the Class name into the Storyboard ID.


Custom Class
Class InstancePropertyViewCo...
Module HardDependencies
☑ Inherit Module From Target
Identity
Storyboard ID InstancePropertyViewController
Restoration ID
☐ Use Storyboard ID

Now we have one storyboard-based view controller we can use in a test. Repeat these steps to create another view controller, naming it ClosurePropertyViewController. You should end up with three scenes in Main.storyboard—the first there by default, and the two you just added.

Add XIB-Based View Controllers

Now we'll add view controllers that use XIBs. Select the HardDependencies group in the Project Navigator. Make a new file, selecting Cocoa Touch Class. Name it OverrideViewController and make it a subclass of UIViewController. This time, select the check box labeled "Also create XIB file."

In the Save dialog, double-check that the app target is selected, not the test target. Pressing Create will add OverrideViewController.swift and OverrideViewController.xib to the project.

Now we have one XIB-based view controller we can use in a test. Repeat these steps to make the following additional view controllers:

- InstanceInitializerViewController
- ClosureInitializerViewController
- MySingletonViewController