Extracted from:

iOS Unit Testing by Example

XCTest Tips and Techniques Using Swift

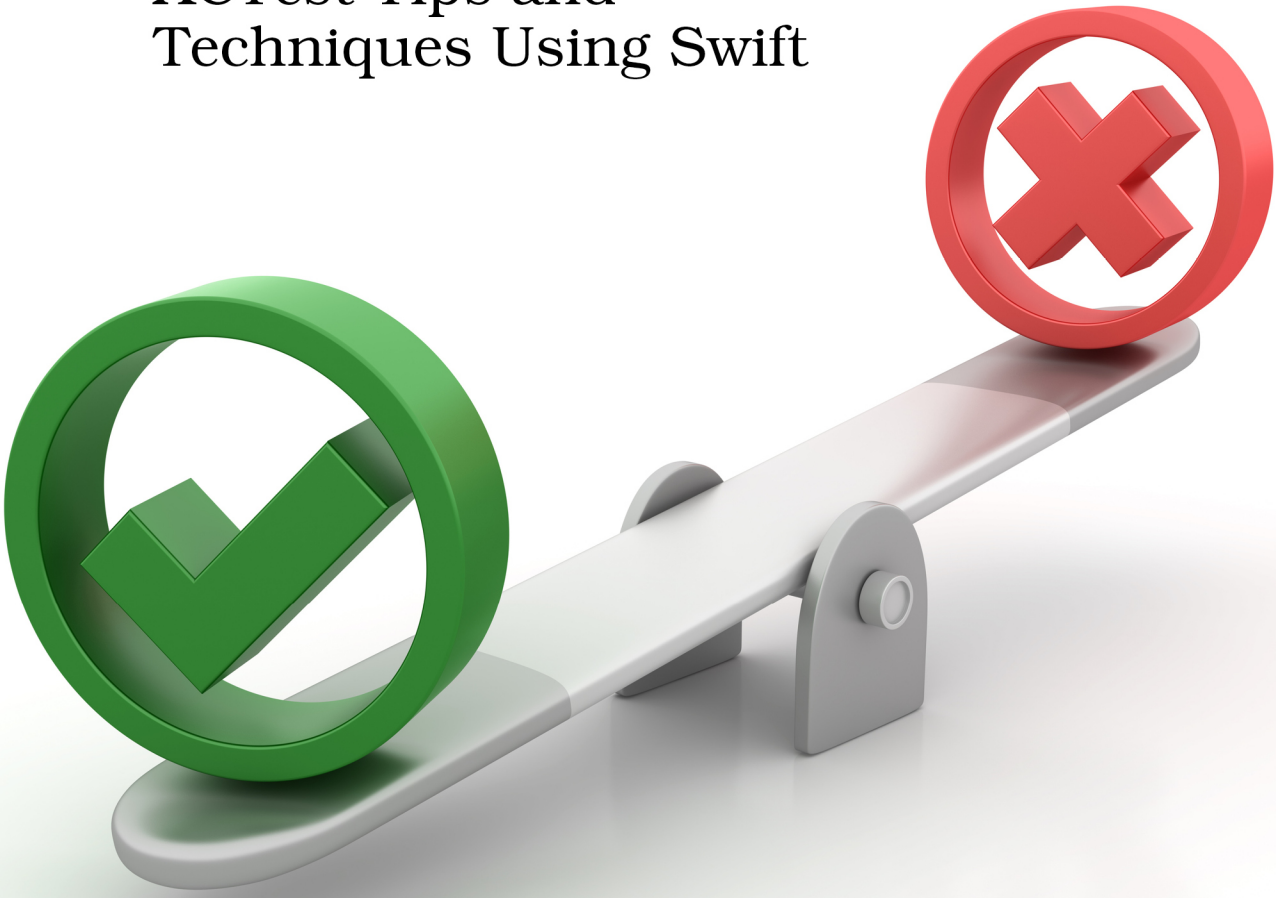
The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

iOS Unit Testing by Example

XCTest Tips and Techniques Using Swift



Jon Reid

edited by Michael Swaine

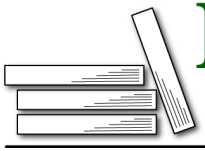
iOS Unit Testing by Example

XCTest Tips and Techniques Using Swift

Jon Reid

The Pragmatic Bookshelf

Raleigh, North Carolina



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

The team that produced this book includes:

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

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

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

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

For Kay, who believed in me

CHAPTER 8

Testing Button Taps (Using Actions)

Tapping a button is one of the most basic things users do with controls. If we can write unit tests for button taps, we'll open up an entire world for your tests.

In this chapter, you'll learn how to write a unit test that taps a button. This will let you confirm the behavior of button taps on your view controllers without manual tests or even UI tests.

Make a Place to Play with a Button

As usual, let's make a new project. Follow the steps for Create a Place to Play with Tests, on page ?, but use the project name ButtonTap. Also delete the initial test file ButtonTapTests.swift.

Let's use the predefined storyboard-based view controller. Edit ViewController.swift, adding a button outlet. Many people declare outlets to be private, so let's do that for this example:

ButtonTap/ButtonTap/ViewController.swift
```
@IBOutlet private var button: UIButton!
```

Add an action, making it private. The action prints to the console to simulate doing something useful:

ButtonTap/ButtonTap/ViewController.swift
```
@IBAction private func buttonTap() {
    print(">> Button was tapped")
}
```

We need to do three things in the storyboard:

1. Add a button to the view controller.
2. Connect the button to its outlet.
3. Connect the button to its action.

First, let's add the button. Open Main.storyboard and drag a button onto the view controller. For this experiment, don't worry about positioning it or setting any Auto Layout constraints.

Second, let's connect it to the outlet. Show ButtonTapViewController.swift in the Assistant Editor, click in the open circle next to @IBOutlet, and drag it to the button on the storyboard.

Finally, connect it to the action. Click in the open circle next to @IBAction, and drag it to the button on the storyboard to connect them.

Let's confirm the button tap code with a manual test. Press ⌘-R to run the app. Then select *View ▶ Debug Area ▶ Activate Console* or press Shift-⌘-C to show the console on the bottom right. Tap the button in the running app. In the console, you'll see the following message:

```
>> Button was tapped
```

Now we're ready to see how to write a unit test that does the button tap.

Test Button Taps

Since buttons are UI elements, many assume you need UI tests to exercise them. But you can tap buttons using unit tests, which are much faster than UI tests.

Let's start by adding a test suite ViewControllerTests. Use Test Zero as temporary scaffolding to confirm that you hooked up the test suite. (See Start from Test Zero, on page ?.) Delete Test Zero once you see its expected failure message.

Add a test named test_tappingButton(). Recall from the tip in Add Tests for Existing Code, on page ? that a good test name states the expected result. But this experimental test won't have any assertions, so we can skip the should clause. Instead, we'll check the console output for the printed message.

Inside the test, add code to load the view controller. (See Chapter 5, Load View Controllers, on page ?.) For our example, ViewController is a storyboard-based view controller. We'll use the simpler forced downcast approach:

ButtonTap/ButtonTapTests/ViewControllerTests.swift

```
func test_tappingButton() {
    let storyboard = UIStoryboard(name: "Main", bundle: nil)
    let sut: ViewController = storyboard.instantiateViewController(
            identifier: String(describing: ViewController.self))
    sut.loadViewIfNeeded()
}
```

Run the tests. Since this is a storyboard-based view controller, you'll see a failure message:

```
failed: caught "NSInvalidArgumentException", "Storyboard
    (<UIStoryboard: 0x6000001f08c0>) doesn't contain a view controller with
    identifier 'ViewController'"
```

Recall from Load a Storyboard-Based View Controller, on page ? that for tests, we need to set the view controller's Storyboard ID. Open Main.storyboard and select the view controller. In the Xcode menu, select *View ▶ Inspectors ▶ Show Identity Inspector* or press ⌥-⌘-4. Then in the Identity Inspector on the right, copy and paste the Class name ViewController into the Storyboard ID field.

Run tests again to confirm that we can now load the view controller. So far, we've been following the principle *Take a small step, get feedback.*

To have a unit test tap a button, the button outlet must be accessible to test code. It's currently private, which limits its visibility to the view controller alone. Let's do the same thing we did for Test Outlet Connections, on page ? and change the outlet's access to private(set):

ButtonTap/ButtonTap/ViewController.swift

```
@IBOutlet private(set) var button: UIButton!
```

Now we're ready to add code to tap the button. Add the following lines to the end of the test case:

ButtonTap/ButtonTapTests/ViewControllerTests.swift

```
sut.button.sendActions(for: .touchUpInside)

// Normally, assert something
```

Run tests and find the console output. (See Examine Console Output, on page ?.)

```
Test Case '-[ButtonTapTests.ViewControllerTests test_tappingButton]' started.
>> Button was tapped
Test Case '-[ButtonTapTests.ViewControllerTests test_tappingButton]' passed
    (0.014 seconds).
```

This shows that the unit test successfully did a button tap! All you need to do is make the outlet accessible, then call sendActions(for:). The event .touchUpInside is the correct event for button taps.

Of course, this means you can send any event to any control using the same trick.

Make a Test Helper for Button Taps

Tapping buttons is something we'll do quite a bit when unit testing view controllers. But sendActions(for: .touchUpInside) isn't very descriptive. We can improve the readability of our tests by extracting a helper to tap UIButtons.

Putting test helpers in a separate file makes it easier to find them. In the Project Navigator, select the ButtonTapTests group and press ⌘-N to make a new file. Select Swift File, name it TestHelpers.swift, and set its target to the test target. Give it the following code:

ButtonTap/ButtonTapTests/TestHelpers.swift

```
import UIKit

func tap(_ button: UIButton) {
    button.sendActions(for: .touchUpInside)
}
```

Change the test to use this new helper:

ButtonTap/ButtonTapTests/ViewControllerTests.swift

```
tap(sut.button)
```

Run the tests and confirm that the console output says Button was tapped.

UIBarButtonItems aren't UIControls, so we can't call sendActions(for:) on them. We can make a separate helper for them. We won't use it until the example in Chapter 17, Unleash the Power of Refactoring, on page ?, where we'll work through an example with a UIBarButtonItem. But for reference, here's a test helper to tap them:

ButtonTap/ButtonTapTests/TestHelpers.swift

```
func tap(_ button: UIBarButtonItem) {
    _ = button.target?.perform(button.action, with: nil)
}
```

Swift's function overloading lets us support the common abstraction of tap(_:) for two types. This also means that if we change a button type from UIButton to UIBarButton, the test code can stay the same.

Extract test helpers to make your tests more readable. They'll also make the tests less fragile.

Key Takeaways

The point of this chapter's exercise was to show you the following:

- That unit tests can send actions to controls.
- That doing so is simple. You just have to make controls non-private so tests can talk to them.
- That test helpers can make tests more readable.

But you may be wondering why I recommend writing unit tests to verify UI behavior such as button taps. Shouldn't we use Apple's UI testing to test UI?

It's helpful to know how the two testing paradigms differ. In UI testing, the tests run in a separate test runner app, sending UI events to the app under test. The app under test is a black box, revealing only the UI elements on the screen.

But in the unit testing paradigm, the tests can have full access to all code that isn't declared `private`. This brings many advantages:

- Unit tests don't have to start from your app's initial screen and navigate to specific screens. Instead, they create whatever view controllers they want.
- Unit tests can inject different dependencies to the system under test. These dependencies can provide canned inputs, or they can record outputs. For example, they can intercept network calls.
- Unit tests are orders of magnitude faster than UI tests.

So test behaviors (and even appearance, as we'll see in Chapter 16, Testing View Appearance (with Snapshots), on page ?) using unit tests as much as you can. They'll give you fast feedback you can incorporate into your coding workflow. Reserve UI testing for anything the unit tests don't already verify, especially end-to-end testing.

Activities

Try the following five-step activity to sink this chapter into your brain:

1. Find a view controller where tapping a button changes some state. If the action method contains difficult dependencies, skip it for now.
2. Write a test that loads the view controller. If you face difficult dependencies in `viewDidLoad()`, move on to a different view controller for now.

3. Change the access control of the button outlet to `private(set)`.
4. Call `sendActions(for:.touchUpInside)` on the button. Use a test helper to make this easier.
5. Add an assertion to make a characterization test (see Add Tests for Existing Code, on page ?) for the changed state. Run tests to get the failure message. Update the assertion to match the actual value, and run tests again.

Repeat this for as many view controllers as you can. Then go back to any view controllers you skipped due to difficulties in either `viewDidLoad()` or the action method. Try to isolate their dependencies using techniques from Chapter 6, Manage Difficult Dependencies, on page ?.

What's Next?

Now you can test user input from button taps. This opens up much of your view controller code to tests. You can also begin testing other user events.

But what do we do when the user taps a button and we want the user to confirm the decision before proceeding with the action? In the next chapter, let's see how to test alerts.