The Case of the Crimson Test Suite

An Introduction to Swift Testing

Daniel H Steinberg

Copyright

ISBN-13: 978-1-944994-07-5

Book Version

This is version 1.0 for Swift 6, Xcode 16 beta 6 or later, macOS Sonoma/Sequoia, and iOS 18 released August 2024.

Code Download

Visit https://github.com/editorscut/ec016swifttesting for all of the code for this book.

Run it in Xcode 16 or higher. All code is written in Swift.

Recommended Settings

The ePub is best viewed in scrolling mode using the original fonts. On smaller devices I also choose landscape. For some reason that I don't understand, scrolling mode is supported by Apple's Books app on the iPad but not on the Mac. If you view this book in Apple's Books app, choose "Let lines break naturally".

Submit Errata

Submit your errata here for the book or for the source code by selecting New Issue. Please provide the book version listed above, chapter, section, and page number in your issue so that I can find it and, if possible, resolve it quickly.

Official Links

Please check http://developer.apple.com for additional resources including videos, sample code, documentation, and forums. You'll also find information on what is required to take advantage of these resources.

Apple has posted videos, slides, and sample code from the Worldwide Developers Conference.

Legal

CHAPTER 2

The Basics Of Swift Testing

Let's get started writing our first actual tests.

We're going to use some code based on my shipping apps. The actual app is both free and open source. You can find the app on the App store and the code on GitHub.

The app allows bread bakers to find the perfect temperature for the water they add to a mix based on other temperatures.

In this chapter we create and test a portion of the app that makes temperature calculations and convert between Celsius and Fahrenheit.

Because we aren't working with anything visual, we'll create and work with a Swift Package.

In this chapter we build and run our first tests. Much of the magic is contained in the attached macro `@Test`.

After we've set up what we need to test, we create an expectation using the `#expect()` macro. In fact, a single test can have more than one expectation.

You'll see that by default the tests run in parallel in an arbitrary order.

If you have experience with XCTest you may be surprised at the lack of ceremony in Swift Testing. You don't have to name the test file in any particular way. We don't need to place our tests in a class that inherits from anything else. The test methods/functions also don't need to be named in any particular way.

We can help the test runner display our tests with nicer names by specifying a wordier display name for a test.

We can also use tags to group tests that are testing related portions of the code. When running the tests in Xcode we can easily run just those tests that share a particular tag.

Note

If at any time you are working in the Terminal and getting feedback that is puzzling, you may need to clean the package. I find I have to do this more frequently than I'd like.

To do this, change to the directory containing *Package.swift* and enter

```
swift package clean
```

Often I find that then I can rerun the tests or build the package and everything behaves as it should.

Multiple Expectations

Our client paused after telling her story. I glanced over and noticed her cup was empty and asked, "more coffee?"

She glanced at me and then at her cup and said, "please."

I poured her a cup of decaf and added a touch of cream and two sugar cubes. I stirred and put the spoon back on the tray and put the cup on the table beside her.

The Agile Detective nodded. It was a small test but the refill had three requirements: decaf, cream, and sugar.

On the plus side, in that small act I'd satisfied three expectations.

On the minus side, it only counted as passing a single test. I also knew that if I'd stumbled on any one of the expectations the entire test would have gone down as a failure.

Continue with our current package or use the package in *Chapter02/03*.

Specifying Fahrenheit

Write a test that creates a new `Temperature` by setting the Fahrenheit value to `140`.

I chose `140` because it's 32 more than a multiple of 9 so I can calculate that the Celsius equivalent is 60 in my head.

You might add a test like this.

*Tests/TemperatureCalculationsTests/**FahrenheitAndCelsius.swift***

```
@Test func createInFahrenheit() {
  let temperature = Temperature(inFahrenheit: 140)
}
```

This is, of course, an error because there is no `init()` that takes a value in Fahrenheit.

Add one. Go ahead and implement this `init()` to convert the Fahrenheit value to Celsius and setting `inCelsius` to this value.

*Sources/TemperatureCalculations/**Temperature.swift***

```
struct Temperature {
  let inCelsius: Double

  init(inCelsius: Double = 50.0) {
    self.inCelsius = inCelsius
  }

  init(inFahrenheit: Double = 122.0) {
    self.inCelsius = (inFahrenheit - 32.0) * 5/9
  }
}
```

I have to say I was feeling pretty good about myself until I returned to the tests and saw this error.

*Tests/TemperatureCalculationsTests/**FahrenheitAndCelsius.swift***

```
@Test func initialValue() {
  let temperature = Temperature()  // error - Ambiguous use of 'init'
  #expect(temperature.inCelsius == 50.0)
}
```

The ambiguity is that using `Temperature()` makes it unclear whether we're initializing `Temperature` in Celsius or Fahrenheit. Let's clean up the inits a bit.

*Sources/TemperatureCalculations/**Temperature.swift***

```
struct Temperature {
  let inCelsius: Double

  init() {
    inCelsius = 50
  }

  init(inCelsius: Double = 50.0) {
    self.inCelsius = inCelsius
  }

  init(inFahrenheit: Double = 122.0) {
    self.inCelsius = (inFahrenheit - 32.0) * 5/9
  }
}
```

Now the ambiguity is removed and everything builds. `Temperature` is clearer. Next let's add expectations to our latest test.

Introducing two expectations

Let's begin by testing that when we use `init(inFahrenheit:)` that the conversion to Celsius is correct and that `inCelsius` is `60`.

*Tests/TemperatureCalculationsTests/**FahrenheitAndCelsius.swift***

```
@Test func createInFahrenheit() {
  let temperature = Temperature(inFahrenheit: 140)
  #expect(temperature.inCelsius == 60.0)
}
```

The tests all pass.

Next, suppose we want to check that we get `140.0` back when we ask for the temperature in Fahrenheit.

We could create a whole new test that looks like this.

*Tests/TemperatureCalculationsTests/**FahrenheitAndCelsius.swift***

```
 // do not add this test
@Test func fahrenheitToFahrenheit() {
  let temperature = Temperature(inFahrenheit: 140)
  #expect(temperature.inFahrenheit == 140.0)
}
```

There's nothing wrong with adding this test, but it does duplicate some of the code unnecessarily.

Instead, add a second expectation to `createInFahrenheit()`.

*Tests/TemperatureCalculationsTests/**FahrenheitAndCelsius.swift***

```
@Test func createInFahrenheit() {
  let temperature = Temperature(inFahrenheit: 140)
  #expect(temperature.inCelsius == 60.0)
  #expect(temperature.inFahrenheit == 140.0)
}
```

The tests all pass. Both expectations in `createInFahrenheit()` are satisfied.

But what happens if they aren't?

Single Failure in Multiple Expectations

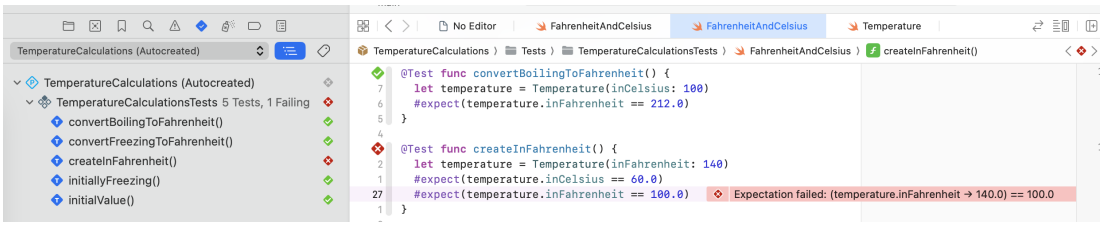
Before leaving this section, I want you to notice that there is no indication in our test report that the `createInFahrenheit()` test had two expectations.

In fact, let's change our test so that one of the expectation fails. Change the expected value of `inFahrenheit` to anything other than `140.0`.

*Tests/TemperatureCalculationsTests/**FahrenheitAndCelsius.swift***

```
@Test func createInFahrenheit() {
  let temperature = Temperature(inFahrenheit: 140)
  #expect(temperature.inCelsius == 60.0)
  #expect(temperature.inFahrenheit == 100.0)
}
```

Let's run the tests in both Xcode and in the Terminal to see what is reported.



In the summary in the Tests Navigator, `createInFahrenheit()` is listed as failing and the rest of the individual tests are marked as passing.

There is no indication that one of the expectations in `createInFahrenheit()` passed. If a single expectation in a test fails, the entire test is marked as failing.

This is consistent. Look at the summary near the top of all of the *TemperatureCalculationTests*. The summary reads "5 Tests, 1 Failing" but there is a red `X` to the right of the report indicating that there is at least one thing failing in that set of tests.

Our goal is for everything to report in green. That's what tells us that all tests are passing.

Run the tests in the Terminal. Here's the important parts of the feedback.

```
Test convertFreezingToFahrenheit() passed.
Test convertBoilingToFahrenheit() passed.
Test initialValue() passed.
Test initiallyFreezing() passed.
Test createInFahrenheit() recorded an issue:
Expectation failed: (temperature.inFahrenheit → 140.0) == 100.0
Test createInFahrenheit() failed with 1 issue.
Test run with 5 tests failed with 1 issue.
```

Next, let's fail both expectations and see what happens.

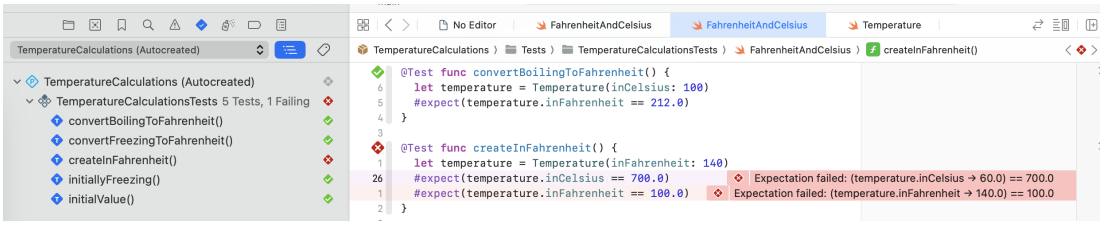
Multiple Failures in Multiple Expectations

Choose any number other than `60.0` for the `inCelsius` expectation so that we fail both expectations.

*Tests/TemperatureCalculationsTests/**FahrenheitAndCelsius.swift***

```
@Test func createInFahrenheit() {
  let temperature = Temperature(inFahrenheit: 140)
  #expect(temperature.inCelsius == 700.0)
  #expect(temperature.inFahrenheit == 100.0)
}
```

Run the tests in Xcode. The feedback in the Tests Navigator is identical. In the editor area we see errors next to each test detailing why they failed.



Run the tests in the Terminal.

This time each issue in `createInFahrenheit()` is reported separately and, although there is still one failing test, the five tests failed because of two issues.

```
Test convertFreezingToFahrenheit() passed.
Test convertBoilingToFahrenheit() passed.
Test initialValue() passed.
Test initiallyFreezing() passed.
Test createInFahrenheit() recorded an issue:
Expectation failed: (temperature.inCelsius → 60.0) == 700.0
Test createInFahrenheit() recorded an issue:
Expectation failed: (temperature.inFahrenheit → 140.0) == 100.0
createInFahrenheit() failed with 2 issues.
```

```
Test run with 5 tests failed with 2 issues.
```

Before moving on, fix our test by restoring the correct values.

*Tests/TemperatureCalculationsTests/**FahrenheitAndCelsius.swift***
```
@Test func createInFahrenheit() {
  let temperature = Temperature(inFahrenheit: 140)
  #expect(temperature.inCelsius == 60.0)
  #expect(temperature.inFahrenheit == 140.0)
}
```

The tests should pass again.

Cleaning up

There is a lot of repeated code in our tests and Swift Testing gives us many ways to combine tests.

You learn to use suites later, but for now let's take advantage of the fact that a single test can contain multiple expectations.

While we're at it, let's do some renaming. The first test was appropriately named `initialValue()` when we created it, but it should probably be renamed `defaultValue()` as that's what it's now testing.

Let's combine the second and third tests, `initiallyFreezing()` and `convertFreezingToFahrenheit()` to a single function named `freezingCreatedInCelsius()` that tests both the creation and converting of the freezing temperature.

I want to make one more change.

In Swift it doesn't matter whether or not we introduce a new line between the annotation and the element it is annotating. For example, it is idiomatic to use a new line after `@Observable` or `@MainActor`. I'm going to introduce a new line after `@Test`. This will become more important in the next section.

Here are my revised tests.

*Tests/TemperatureCalculationsTests/**FahrenheitAndCelsius.swift***

```
import Testing
@testable import TemperatureCalculations

@Test
func defaultValue() {
  let defaultTemperature = Temperature()
  #expect(defaultTemperature.inCelsius == 50.0)
}

@Test
func freezingCreatedInCelsius() {
  let freezing = Temperature(inCelsius: 0)
  #expect(freezing.inCelsius == 0.0)
  #expect(freezing.inFahrenheit == 32.0)
}

@Test
func convertBoilingToFahrenheit() {
  let boiling = Temperature(inCelsius: 100)
  #expect(boiling.inFahrenheit == 212.0)
}

@Test
func createInFahrenheit() {
  let temperature = Temperature(inFahrenheit: 140)
  #expect(temperature.inCelsius == 60.0)
  #expect(temperature.inFahrenheit == 140.0)
}
```

We have fewer tests and more expectations. We could have made other changes but I wanted to keep some of them separate so I could introduce you to names and tags for tests.