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 4

The Finer Points Of Swift Testing

In this final chapter we continue to work with our RPN Calculator project.

One of the many nice things about Swift Testing is that, by default, the tests are run in parallel and in random order.

This means that we won't make errors because we've missed hidden dependencies that might arise if our tests always run in the same order

There are, however, rare times when we want some subset of our tests to run in a specific order.

Rare. Almost never.

In those cases we will explicitly mark a suite so that it runs the tests sequentially from first to last. We'll begin this chapter with an extended example of serialized tests and call out some of the issues that can arise if we're not careful.

We then take a look at testing `async` methods. It's true that our calculator doesn't have a need for `async` methods so I build a side-example that let's us test `async` methods and explore some of the issues we might have with them. Fortunately, Swift Testing is a

modern framework that is built to work with Swift 6 and embrace Swift Concurrency.

We finish implementing our calculator's buttons for unary and binary operations which brings up some of the finer points of any sort of testing (though we are interested in particular in how they manifest in Swift Testing).

We've used a lot of techniques to streamline our tests and test output throughout this book. In the final sections we use parameterized tests to eliminate repeated tests that differ only in their input and `testDescription` to customize the way in which we present results in the Tests Navigator and elsewhere.

Testing Asynchronous Code

Edges looked up, surprised to see me, and said, "Swiftly, mon ami, I thought you were working."

"I am working," I said. I turned to the waiter and asked for a cappuccino.

"Forgive me," said Edges, "it appears as if you aren't doing anything right now."

"I'm waiting," I said. "I have a few phone calls to make. I made them all and I'm waiting for them to call me back."

"And you're going to wait all day?" asked Edges.

"No," I said, "the Agile Detective told me that if they haven't called back by the end of the day I should mark that task as failed and we'll move on to something else tomorrow."

My phone buzzed. I glanced at the screen to see which call was being returned, excused myself from the table, and said, "Hello?".

In this section we write tests for `async` methods.

The calculator will not require any asynchronous code, but I've created some just so we can see how to test it.

Fortunately, Swift Testing works well with the familiar async mechanisms you know from Swift.

Continue with our current project or use the project in *Chapter04/03*.

Awaiting an async method

We aren't going to bother with creating a tag for async code but we will create a new file and suite.

Create an empty file in *RPNCalculatorTests* named *AsyncTests.swift*.

*RPNCalculatorTests/**AsyncTests.swift***
```
import Testing
@testable import RPNCalculator

@Suite("Async tests")
struct AsyncTests {

}
```

You'll find `timesTwo()`, the async function we're going to test, in *Model/Operators/AsyncOperator.swift*.

*RPNCalculator/Model/Operators/**AsyncOperator.swift***
```
func timesTwo(_ input: Double) async -> Double {
  try? await Task.sleep(for: .seconds(2))
  return input * 2
}
```

It sleeps for two seconds and then returns two times whatever number is passed to it.

Next let's write a test that calls `timesTwo()`.

Our call to this `async` method must be labeled `await` to mark a possible suspension point.

*RPNCalculatorTests/**AsyncTests.swift***
```
@Suite("Async tests for fake code")
struct AsyncTests {
  @Test
  func twoTimesANumber() {
    let result = await timesTwo(2.3)  // error
  }
}
```

There is an error because we are making an async call outside of an async context. We have two choices, we can either wrap our async call in a `Task` or we can mark `twoTimesANumber()` as `async`.

The correct solution is to mark `twoTimesANumber()` as `async`, but I want to show you what goes wrong if instead we use `Task`.

Wrongly using Task

To see what's wrong with `Task` let's create an expectation that should fail.

*RPNCalculatorTests/**AsyncTests.swift***
```
@Test
func twoTimesANumber() {
  Task {
    let result = await timesTwo(2.3)
    #expect(result == 17)
  }
}
```

Run the tests for the suite `AsyncTests`. The test reports that it passed.

That's not possible.

We get more of an idea of what's going on by checking out the Console output.

```
Suite "Async tests" started.
Test twoTimesANumber() started.
Test twoTimesANumber() passed after 0.001 seconds.
Suite "Async tests" passed after 0.001 seconds.
Test run with 1 test passed after 0.001 seconds.
```

The test passes after 0.001 seconds but there's sleep for two seconds in the middle of the call to `timesTwo()`.

This is consistent with how `Task` works. We are giving work to be performed asynchronously in a `Task`. This work will be initiated by an executor outside the testing framework. Meanwhile, we immediately resume execution after the close of the `Task` closure and exit the test.

By the time `timesTwo()` completes and returns, you can see that the test function `twoTimesANumber` has returned and our tests have completed. So no errors are reported.

`Task` is the wrong tool for running asynchronous tests.

Test method should be async

If we want to run a test that includes calls to something that is `async`, our test function or method should be `async`.

Remove `Task` and declare `twoTimesANumber()` to be `async`.

*RPNCalculatorTests/**AsyncTests.swift***

```
@Test
func twoTimesANumber() async {
  Task {
    let result = await timesTwo(2.3)
    #expect(result == 17)
  }
}
```

Run `AsyncTests` and this time our test fails.

This is great.

Again, check the Console and the output matches our expectation better. (I've removed the location information to shorten the output.)

```
Suite "Async tests" started.
Test twoTimesANumber() started.
Test twoTimesANumber() recorded an issue:
Expectation failed: (result → 4.6) == (17 → 17.0)
Test twoTimesANumber() failed after 2.055 seconds with 1 issue.
Suite "Async tests" failed after 2.055 seconds with 1 issue.
Test run with 1 test failed after 2.055 seconds with 1 issue.
```

We see that `twoTimesANumber()` failed after a little more than two seconds. The test is sticking around to get the result and then using `result` in the expectation.

Known issues

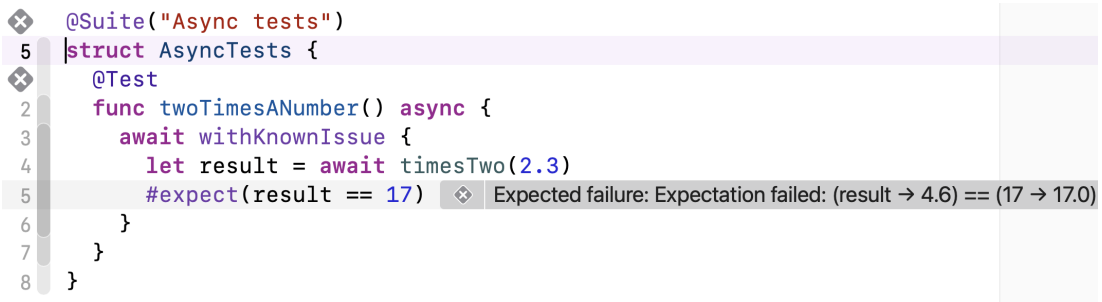
We will fix our test in a minute. For now, suppose we couldn't. We know there's a problem but we can't fix it yet.

We can wrap the problematic code in `withKnownIssue` like this.

*RPNCalculatorTests/**AsyncTests.swift***

```
@Test
func twoTimesANumber() async {
  await withKnownIssue {
    let result = await timesTwo(2.3)
    #expect(result == 17)
  }
}
```

The test is still executed and the reason it isn't passing is displayed in grey.



The issue is also reported in the Console.

```
Test twoTimesANumber() recorded a known issue
Expectation failed: (result → 4.6) == (17 → 17.0)
```

This allows us to keep track of an issue but not have a failing test.

I want to point out two important things about `withKnownIssue`.

First, even though I'm introducing it in the context of testing an async method, we can use `withKnownIssue` in non-async settings as well.

Even cooler, when the issue is resolved, the test will fail.

In other words, suppose we fix this test so that it would ordinarily pass.

*RPNCalculatorTests/**AsyncTests.swift***

```
@Test
func twoTimesANumber() async {
  await withKnownIssue {
    let result = await timesTwo(2.3)
    #expect(result == 4.6)
  }
}
```

This test now fails because there is no longer a known issue.

```
Known issue was not recorded
```

I love this. In a real example we would not be adjusting our test we'd be working on the production code and our test would be able to tell us that there's no longer a known issue and that we can remove this guard rail.

Now that you've seen this, go ahead and remove `withKnownIssue`.

*RPNCalculatorTests/**AsyncTests.swift***

```
@Test
func twoTimesANumber() async {
  await withKnownIssue {
    let result = await timesTwo(2.3)
    #expect(result == 4.6)
  }
}
```

This test now passes. Let's add another test.

A second async test

Instead of first calculating `result` and then using it in the expectation, we can make an asynchronous call inside of the expectation as long as it is made from the left side of the `==`.

*RPNCalculatorTests/**AsyncTests.swift***

```
@Suite("Async tests")
struct AsyncTests {
  @Test
  func twoTimesANumber() async {
    let result = await timesTwo(2.3)
    #expect(result == 4.6)
  }

  @Test
  func twoTimesAnotherNumber() async {
    #expect(await timesTwo(7.9) == 15.8)
  }
}
```

Run the tests in `AsyncTests`. I love what we see.

```
Suite "Async tests" started.
```

```
Test twoTimesAnotherNumber() started.
Test twoTimesANumber() started.
Test twoTimesANumber() passed after 2.012 seconds.
Test twoTimesAnotherNumber() passed after 2.012 seconds.
Suite "Async tests" passed after 2.012 seconds.
Test run with 2 tests passed after 2.012 seconds.
```

Each test takes a little over two seconds to run and yet the entire test suite only takes 2.012 seconds.

We see the benefit of parallel tests.

Time limits and disabled

There is a function in *AsyncOperator.swift* named `longTwoTimes()`. It is the same as `twoTimes()` except that the `Task` sleeps for 90 seconds. Add a test for it to `AsyncTests`. Here's a possible example.

*RPNCalculatorTests/**AsyncTests.swift***

```
@Test
func longOperation() async {
  #expect(await longTimesTwo(4.7) == 9.4)
 }
```

Run `AsyncTests`. No matter which order the tests are run, the first two should pass in a little over two seconds. The third one takes a little more than a minute and a half.

We have a very specific example where the `Task` sleeps for a specific amount of time. Often we're making a network call or performing

some other task which we expect to complete within some time limit. Swift Testing makes it easy for us to set a time limit.

To see it in action, let's specify that `longOperation()` should complete in less than a minute. Timelimits are always specified in minutes.

*RPNCalculatorTests/**AsyncTests.swift***

```
@Test(.timeLimit(.minutes(1)))
func longOperation() async {
  #expect(await longTimesTwo(4.7) == 9.4)
}
```

Run the tests in `AsyncTests` again. After a minute `longOperation()` fails.

The failure is reported as:

```
Time limit was exceeded: 60.000 seconds
```

Perhaps we consider this a bug and want to note that `longOperation()` always exceeds this time limit.

Add the `bug` trait which takes a URL and a description.

*RPNCalculatorTests/**AsyncTests.swift***

```
@Test(.timeLimit(.minutes(1)),
      .bug("http://example.com",
           "longTimesTwo always takes more than 60 seconds"))
func longOperation() async {
  #expect(await longTimesTwo(4.7) == 9.4)
}
```

Using `bug` does not prevent the test from running. We can use the `disabled` trait to do this.

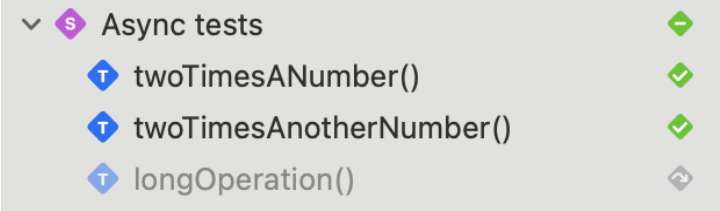
*RPNCalculatorTests/**AsyncTests.swift***

```
@Test(.timeLimit(.minutes(1)),
      .bug("http://example.com",
           "longTimesTwo always takes more than 60 seconds"),
      .disabled("Performance issue with longOperation()"))
func longOperation() async {
  #expect(await longTimesTwo(4.7) == 9.4)
}
```

Run the tests in `AsyncTests`. The test is skipped and the following is reported in the editor and the Console.

```
Test longOperation() skipped: "Performance issue with
longOperation()"
```

We also see `longOperation()` grayed out in the Tests Navigator and a gray arrow indicates the test was skipped.



There are versions of `disabled()` that allow us to specify a condition for which the test or suite is disabled and there is also `enabled()` that works as you would expect.

Closure based async

This section will become less relevant over time as closure-based asynchronous methods are replaced with the modern `async` syntax. I

cover the technique used here more in depth in "The Curious Case of the Async Cafe".

There is a final function in *AsyncOperator.swift* named `closureTimesTwo()` which takes a `Double` and a closure `(Double) -> Void`. It is old school async. Instead of returning a value asynchronously, when `closureTimeTwo()` is ready to announce a value it calls the completion.

Here's the implementation of `closureTimesTwo()`.

*RPNCalculator/Model/Operators/**AsyncOperator.swift***

```
@MainActor
func closureTimesTwo(_ input: Double,
                     completion: @escaping (Double) -> Void) {
  Task {
    let result = await timesTwo(input)
    completion(result)
  }
}
```

We can't call `closureTimeTwo()` with an `await`. Instead, we wrap it in a checked continuation like this.

*RPNCalculatorTests/**AsyncTests.swift***

```
@MainActor
@Test
func closureAsync() async {
  let result = await withCheckedContinuation {continuation in
    closureTimesTwo(5.8, completion: {double in
      continuation.resume(returning: double)})
  }
  #expect(result == 11.6)
}
```

Run `AsyncTests`. All the ones that aren't skipped will pass. Each takes a little more than two seconds and they run in parallel and take a total of a little more than two seconds.

Before moving on, let's disable the entire suite as it just adds unecessary time and tests code we aren't going to use.

*RPNCalculatorTests/**AsyncTests.swift***

```
@Suite("Async tests",
       .disabled())
struct AsyncTests {
```

Run the tests. All four tests in `AsyncTests` are skipped and the total time taken by testing is 0.027 seconds on my machine.

That's our quick look at testing asynchronous code.