CHAPTER 1

Assert Yourself

Every company wants to reduce their costs. In software, making changes is inexpensive: we wiggle our fingers on keyboards. So where do the costs lie? Aside from development time, they lie in errors, and how much time it takes to detect these errors. (They also lie in building the wrong thing, which is beyond the scope of this book.)

To detect problems, mobile developers use all kinds of feedback loops. For example, we keep an eye on crash reports and customer complaints. But that's the longest loop. After making an incorrect change, it takes a long time to get that feedback.

To try to prevent errors from making it all the way to customers, companies use manual testing. The best quality experts apply talent and creativity to do exploratory testing. Let's not waste their time asking them to follow steps in mind-numbing repetition. Besides, the time between making an error and getting feedback from testers is still long.

What if we could do a large amount of testing using computers? In fact, what if the developer's own computer could provide feedback? And what if this feedback were so quick, you could get it on every change you made? This kind of rapid feedback is a game changer. It not only catches problems quickly, it can change the way you code.

This is what unit tests are for. Maybe you haven't done any unit testing in your iOS apps yet. Or maybe you've been able to test some logic, but your tests don't cover the iOS-specific parts. (And those are important parts.) Wherever you are in your unit testing journey, the goal for this book is the same: to reduce your costs.

What Are Unit Tests Anyway?

There's some confusion about what makes a test a *unit test*. Many people try to focus on the "unit" part of the name, thinking it describes testing a unit of production code. I'll continue to use the term because it's widespread, but let's forget about asking "What's a unit?" Instead, here's my definition:

Unit tests are a subset of automated tests where the feedback is quick, consistent, and unambiguous.

Quick: A single unit test should complete in milliseconds. We want thousands of such tests.

Consistent: Given the same code, a unit test should report the same results. The order of test execution shouldn't matter. Global state shouldn't matter.

Unambiguous: A failing unit test should clearly report the problem it detected.

In our first chapter, we'll explore the fundamental tool of unit testing: assertions. You'll learn the most common assertions in the Swift XCTest framework in a hands-on way.

If you're a seasoned unit test writer, you may want to skip ahead to the Key Takeaways, on page 16. But even if you've written some tests, it can be good to go over the fundamentals. What are assertions for? What do they report? Do you know how to choose the right assertion for the right job? This chapter will help you get familiar with these tools, which we're going to be using all the time.

Create a Place to Play with Tests

Assertions give unit tests a way to state their expectations. The tests fail if these expectations aren't met. Let's make a place outside of your actual projects where we can experiment with how they work. Throughout this book, you'll learn new concepts by playing in these safe spaces. Then in the exercises at the end of each chapter, you'll begin applying these concepts to your own code.

When it comes to learning, reading doesn't come close to *doing*. If you take the code from the examples and type them into your computer, your learning will go deeper. So I encourage you to open your IDE of choice and give it a go. (The examples will use Xcode.)

Let's start by making a place where we can play with tests. Xcode playgrounds are tricky to use with XCTest, so we won't do that. Instead, we'll make a new project. In the Xcode menu, select *File* ▶ *New* ▶ *Project...* or press Shift-⌘-N.

It doesn't matter what type of project we make as long as it comes with unit test support. But since we're going to focus on testing iOS apps, we may as well get used to what that feels like. First, create an iOS Single View App.

Next, choose any options you like for your new project. In the examples that follow, we'll use the project name AssertYourself. But make sure to do the following:

- Choose "Swift" as the language.
- Choose "Storyboard" as the user interface. (Don't select "SwiftUI.")
- Select the check box for "Include Unit Tests."

You now have a project set up to run unit tests on an iOS app, which we'll use for our learning experiments.

Select the initial test file that the new project created. Its name will be the project name followed by Tests. So for this project, find AssertYourselfTests.swift.

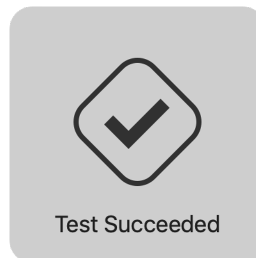
Delete every method in the AssertYourSelfTests class, leaving only an empty shell:

```
class AssertYourselfTests: XCTestCase {
}
```

Make sure your destination is set to an iOS simulator. Any simulator will do.

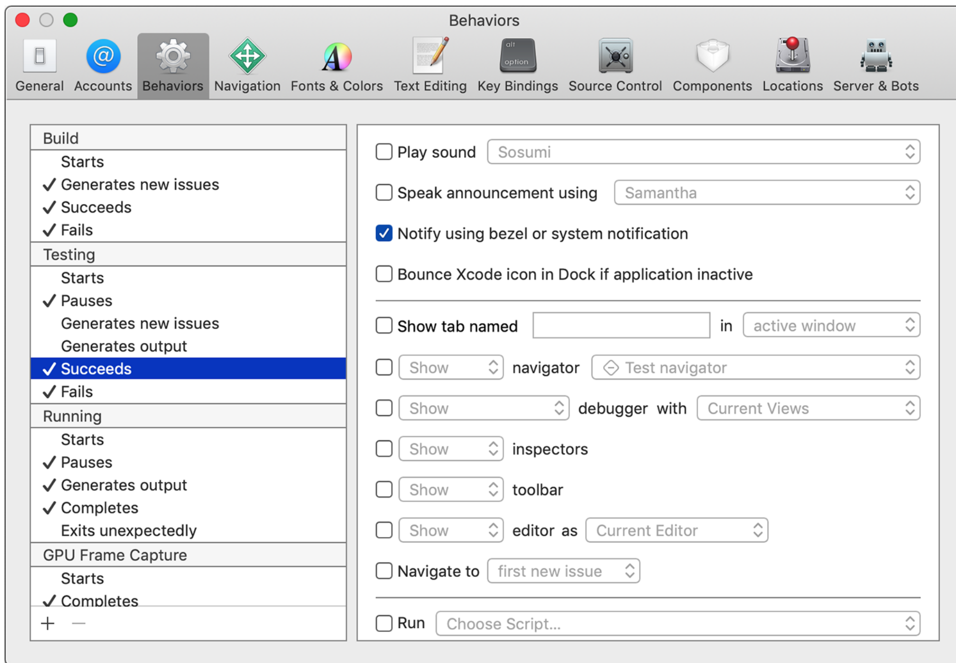
Now in the Xcode menu, select *Product ▶ Test* or press ⌘-U. You might want to learn this keyboard shortcut—you'll be doing this often. Think U for "unit test" to remember it.

This will perform several steps and then run the tests. You won't see any test failures because there are no tests. You may see this image show briefly on your screen:



If you didn't see that image, go to Xcode Preferences and select the Behaviors tab. There you can customize what happens when testing succeeds. To display the image, select the check box "Notify using bezel or system notification," as shown in the image on page 6.

Now we're ready to play. In the following sections, we'll experiment with assertions to learn more about them.



Write Your First Assertion

Now that we have a home for tests, let's go over how to use the testing mechanism. How does a test communicate success or failure? What does Xcode show you when a test fails? What does it show when a test succeeds?

The way a test reports a failure to XCTest is through assertions. Let's start with the simplest assertion. Add the following method to the AssertYourselfTests class:

AssertYourself/AssertYourselfTests/AssertYourselfTests.swift

```
func test_fail() {
    XCTFail()
}
```

First, what makes this function a test?

- It lives within a subclass of XCTestCase.
- It isn't declared private.
- Its name starts with test.
- It takes no parameters.
- It has no return value.

Why the underscore in the test name? This goes against Swift's normal "camel case" naming conventions. But good test names often contain three parts. I like to use underscores to separate these parts and camel case within each part. I'll explain this further when we have a test name describing its inputs and expected output. For now, know that the underscores separate the test name into parts, which we'll look at in Add Tests for Existing Code, on page 40.

This test does nothing but fail. Run it by pressing ⌘-U and observe what happens. First, you may see this image show briefly on your screen:

(If you didn't see that image, go back to the Behaviors tab in Xcode preferences. Only this time, customize what happens when testing fails.)

Looking at the earlier source file within Xcode, you'll see the *Test Status Icon* in the left-hand gutter, like the image to the right.

X marks the spot in two places: the method and the class containing the method. The method is a *test*, also known as a *test case*. The class represents a *test suite*, which is a collection of tests. The X icon shows a failure at both the test level and the suite level. You'll also see that Xcode highlighted the XCTFail() line and added an annotation to its right.

```
XCTFail()                    failed
```

So Xcode has marked the following:

- The class containing a failing test
- The method defining a failing test
- The line with the failed assertion

Now add // before XCTFail() to comment out the assertion. Press ⌘-U to run the tests. You'll see the following:

- The annotation disappears from the assertion line
- The test status icons change from red Xs to green check marks, like the image to the right.

This may look trivial, but it's significant. It means we have a way to fail a test, with Xcode showing us where the test reported the failure. You can also see that when a test finishes without triggering any assertions, the test passes.

> As you progress in your testing ability, you'll even be able to write assertions defining what you *want* the code to do. Then you can change the production code until it passes the tests. We'll return to this topic in Chapter 20, Test-Driven Development Beckons to You, on page 297 at the very end of the book.

Add a Descriptive Message

Seeing the location of a test failure is a good start. But when a test fails, we have to diagnose what went wrong. We can save time for ourselves in the future by having the assertion explain anything we know at the point of failure.

XCTFail() can take a String parameter as an *assertion message*. Let's see how it works. Add the following method to the class:

AssertYourself/AssertYourselfTests/AssertYourselfTests.swift

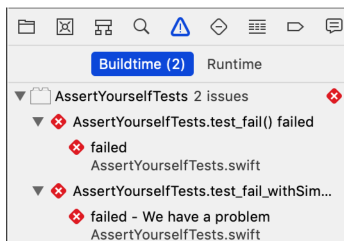
```swift
func test_fail_withSimpleMessage() {
    XCTFail("We have a problem")
}
```

Run the tests. Note how Xcode puts the message in the annotation:

```
XCTFail("We have a problem")        failed - We have a problem
```

Since the annotation is on the same line as the failure, you may ask, "Couldn't we have put a message to ourselves in a code comment?" But this isn't the only place the message appears.

In the Xcode menu, select *View* ▶ *Navigators* ▶ *Show Issue Navigator* (or press ⌘-5). The Navigator column on the left will show any issues, including test failures. You may need to click the Buildtime selector, shown here:



As you can see, the descriptive failure message appears in the Issue Navigator. It also appears in the test logs, which other tools may process—especially on *continuous integration* servers.

Thanks to Swift's string interpolation, XCTFail() can do more than spit out a string literal. Add this to the suite:

AssertYourself/AssertYourselfTests/AssertYourselfTests.swift
```
func test_fail_withInterpolatedMessage() {
    let theAnswer = 42
    XCTFail("The Answer to the Great Question is \(theAnswer)")
}
```

(Strings are italicized in code samples. That's a backslash \ for string interpolation, not a pipe |.)

Run the tests, and you'll see the following:

```
failed - The Answer to the Great Question is 42
```

Avoid Conditionals in Tests

We can report failures and include descriptive messages. Now that you've tasted the power of XCTFail, it's tempting to use it everywhere. All it takes is a little more code in the test, right? That may be true, but "more code" is code that can go wrong. Let's learn how to simplify our test code by introducing more assertions.

For example, it might be tempting to test a Boolean result like this:

AssertYourself/AssertYourselfTests/AssertYourselfTests.swift
```
func test_avoidConditionalCode() {
    let success = false
    if !success {
        XCTFail()
    }
}
```

That would be fine if we didn't have other assertions. But we do. Try adding and running this next test. It achieves the same result but in a more declarative way.

AssertYourself/AssertYourselfTests/AssertYourselfTests.swift
```
func test_assertTrue() {
    let success = false
    XCTAssertTrue(success)
}
```

By using the Boolean assertions XCTAssertTrue() and XCTAssertFalse(), we can avoid many conditionals in our test code.

Eliminating branches from test code makes it easier to understand. I want test code to be extremely simple. In fact, *xUnit Test Patterns* [Mes07] lists *conditional test logic* as a *test smell*.

Let's look at the three types of control flow constructs we use daily:

- Statements in a sequence
- Conditionals
- Loops

These control flows fall into a paradigm called *structured programming*. They've become the building blocks of programming.

If our test code can avoid conditionals and loops, then we're left with one thing: statements executed in sequence. The best test code is dead simple to read. Of course, there are still conditionals inside there somewhere. But by using assertions that have more power, our test code becomes simpler.

Describe Objects upon Failure

Wouldn't it be nice if we had assertions that came with descriptive messages? The assertions we've seen so far can only say that they failed, but they can't tell us why. But there are some assertions that describe objects. We'll also look at how to customize the way objects describe themselves in these messages.

Here's an assertion to confirm that an optional value is `nil`. Add this test and give it a run:

AssertYourself/AssertYourselfTests/AssertYourselfTests.swift

```
func test_assertNil() {
    let optionalValue: Int? = 123
    XCTAssertNil(optionalValue)
}
```

This is the first assertion that gives us more information upon failure:

```
XCTAssertNil failed: "123" -
```

Instead of `nil`, we got `"123"`. But why is it in quotes when the type is an optional integer with value 123? That's the way XCTest reports strings, and assertions ask objects to describe themselves as strings. We can see this better with a `struct` instead of an `Int`:

AssertYourself/AssertYourselfTests/AssertYourselfTests.swift

```
struct SimpleStruct {
    let x: Int
    let y: Int
}
```

```
func test_assertNil_withSimpleStruct() {
    let optionalValue: SimpleStruct? = SimpleStruct(x: 1, y: 2)
    XCTAssertNil(optionalValue)
}
```

Running this test gives us this message:

```
XCTAssertNil failed: "SimpleStruct(x: 1, y: 2)" -
```

That's pretty readable for a simple struct. But some types have complicated descriptions. This can make failure messages hard to read. We can control how a type describes itself by making it conform to CustomStringConvertible.

Here's a structure that is identical to the previous one, but it adds the protocol to give itself a custom description:

AssertYourself/AssertYourselfTests/AssertYourselfTests.swift

```
struct StructWithDescription: CustomStringConvertible {
    let x: Int
    let y: Int

    var description: String { "(\(x), \(y))" }
}

func test_assertNil_withSelfDescribingType() {
    let optionalValue: StructWithDescription? =
            StructWithDescription(x: 1, y: 2)
    XCTAssertNil(optionalValue)
}
```

Running this test gives us the following simplified failure message:

```
XCTAssertNil failed: "(1, 2)" -
```

XCTAssertNil() is one assertion that gives more information. That's because it takes an object instead of a Boolean value. The assertions for equality also give more information, and we'll look at them next.

Even in the cases where we provide our own descriptive messages, it's good to have an option to simplify the output. Keep CustomStringConvertible in your tool belt.

Test for Equality

We've tried a few different assertions so far, including one that gives more output. And we have a way to customize its output. But now we're coming up to the workhorse of assertions, the one you'll use most often.

The most common assertion takes a result and checks if it's equal to an expected value. Try entering and running this test:

AssertYourself/AssertYourselfTests/AssertYourselfTests.swift

```
func test_assertEqual() {
    let actual = "actual"
    XCTAssertEqual(actual, "expected")
}
```

Here's the resulting failure message:

```
XCTAssertEqual failed: ("actual") is not equal to ("expected") -
```

It's worth noting that other unit testing frameworks usually use (expected, actual) for their equality arguments. The order matters because the failure message states which is which in the following format:

```
expected: <"expected"> but was: <"actual">
```

But with XCTAssertEqual(), the argument order doesn't matter. It simply reports ("A") is not equal to ("B"). Since we can put them in any order, there's no need to place the expectation first, as in

```
XCTAssertEqual("expected", actual)
```

But it does change the failure message. I prefer to flip the order, placing the expectation last. Whichever style you prefer, it doesn't matter to XCTest. But to make assertions easier to read, try to be consistent across your project.

Test Equality with Optionals

Let's explore equality further. One of Swift's core features is optional values. When one of the arguments to XCTAssertEqual() is optional, what happens? Enter and run the following test:

AssertYourself/AssertYourselfTests/AssertYourselfTests.swift

```
func test_assertEqual_withOptional() {
    let result: String? = "foo"
    XCTAssertEqual(result, "bar")
}
```

The failure message is

```
XCTAssertEqual failed:
    ("Optional("foo")") is not equal to ("Optional("bar")") -
```

Yet we typed a plain string literal "bar" as the second argument. How did it become optional?

Well, XCTAssertEqual() requires both arguments to be the same type. Swift knows that if a value of type T is being assigned to a variable of type T?, it can wrap it. This promotes the value from non-optional to optional.

All this makes it easier to write equality assertions when optionals are involved. There's no need to balance out both sides of the equation ourselves. This helps make test code more readable.

Why do I emphasize readability for test code? Isn't it good enough to have things pass or fail?

Change happens. The production code will evolve, so test code will need to change with it. To change test code, we need to understand it. And every time we need to understand code, we read it. Making test code readable is an act of kindness to your coworkers, and to yourself.

As *Clean Code: A Handbook of Agile Software Craftsmanship* *[Mar08]* says, "Test code is just as important as production code."

Fudge Equality with Doubles and Floats

We've looked at the equality assertion. We've seen how it continues to work fine with optional values. Now let's see how it works with floating-point numbers. If you're not already aware of what can go wrong, buckle your seat belt.

Enter this next test. But don't run it yet:

AssertYourself/AssertYourselfTests/AssertYourselfTests.swift

```
func test_floatingPointDanger() {
    let result = 0.1 + 0.2
    XCTAssertEqual(result, 0.3)
}
```

Before running the test, try predicting the outcome. Do you have an expected result in your head?

Okay, now run the test. You'll see the following failure message:

```
XCTAssertEqual failed: ("0.30000000000000004") is not equal to ("0.3") -
```

What in the world is going on?

We're used to using ten digits to represent numbers. Can you write ⅓ in decimal notation? No. The sequence 0.3333… goes on forever, so anything you write down is an approximation.

That's just the way math works. Computers face the same problem, but everything boils down to 1s and 0s, so the tricky numbers are different. We

can't write $^1/_{10}$ in binary notation. You can learn more about this at "What Every Programmer Should Know About Floating-Point Arithmetic."[1]

Let's get back to assertions. Since floating-point numbers are approximations, we need a hand-wavy way to assert equality—something that says, "These two numbers should be equal, more or less." Enter the following test:

AssertYourself/AssertYourselfTests/AssertYourselfTests.swift

```
func test_floatingPointFixed() {
    let result = 0.1 + 0.2
    XCTAssertEqual(result, 0.3, accuracy: 0.0001)
}
```

The accuracy parameter gives us a way to express the "more or less" fudge factor. Run this test and you'll see that it passes.

It's hard to predict in advance which floating-point numbers will cause problems. So just use the accuracy parameter whenever you want to use XCTAssertEqual() with Double or Float types.

Avoid Redundant Messages

Let's finish up our examination of the equality assertion by looking at its descriptive message.

As you may have guessed from Add a Descriptive Message, on page 8, each assertion can have an optional message. When you first learn this power, it's easy to get overly excited. But consider the following test:

AssertYourself/AssertYourselfTests/AssertYourselfTests.swift

```
func test_messageOverkill() {
    let actual = "actual"
    XCTAssertEqual(actual, "expected",
                   "Expected \"expected\" but got \"\(actual)\"")
}
```

The resulting failure message is:

```
XCTAssertEqual failed: ("actual") is not equal to ("expected") -
    Expected "expected" but got "actual"
```

The added message may be a little more precise. But if you're consistent with the order you use for actual value versus expected value, it doesn't add much. Getting all that formatting right took extra work for little benefit.

Remember, when XCTAssertEqual() or XCTAssertNil() fail, they provide a fair bit of information. It's usually enough. XCTAssertTrue() and XCTAssertFalse() only say they

1. https://floating-point-gui.de

failed, but that too is often enough. We're going to aim for tests that are so short, we won't need to add any messages of our own.

So for now, resist the temptation: unless you're using XCTFail(), leave the message out. We'll find a use for assertion messages later.

Testing Without Assertions, Back in the Stone Age

Before the invention of unit testing frameworks, it wasn't like programmers didn't run any tests. We wrote little main() functions in our source files, conditionally compiled out. By building a single file with its main() function enabled, we'd make a little program to exercise that one file. It would print() to the console, and we'd read the output to see if it matched what we wanted.

Life is much easier now. Instead of having a human read the output, assertions give us *self-checking tests*. And we have test suites, which let you run a set of tests in one shot.

Choose the Right Assertion

That wraps up our tour of the most common XCTest assertions. With these choices and more, how do you choose which one to use for a particular test? Since all automated tests come down to a true/false decision, it may be tempting to forget the choices and simply use XCTAssertTrue() for all your tests. For example, you may think about writing assertions like these:

```
XCTAssertTrue(a == b)
XCTAssertTrue(optionalValue == nil)
```

These assertions will fail correctly when a is not equal to b, or when optionalValue is not nil. But the failure messages would only say

```
XCTAssertTrue failed -
```

Then we'd have to diagnose what went wrong.

Assertions like these throw away valuable information. As *xUnit Test Patterns [Mes07]* explains, test assertions have two goals:

- Fail the test when something other than the expected outcome occurs.
- Document how the system under test is *supposed* to behave (i.e., *tests as documentation*).

In other words, it's not enough to report that a test failed. What was the actual result? How did it differ from the expected result? These are the questions we should be able to answer from failure messages.

So pick the assertion function that's closest to what you want to say. While XCTest provides sixteen assertion functions,[2] these are the ones you'll use the most:

Assertion	Purpose
XCTAssertEqual(_:_:)	Asserts that two values are equal
XCTAssertEqual(_:_:accuracy:)	Asserts that two floating-point values are equal within a certain accuracy
XCTAssertNil(_:)	Asserts that an optional value is nil
XCTAssertNotNil(_:)	Asserts that an optional value is not nil
XCTAssertTrue(_:)	Asserts that an expression is true
XCTAssertFalse(_:)	Asserts that an expression is false
XCTFail()	Fails the current test. You should always provide a descriptive message.

Key Takeaways

What are the key points from this chapter that you should apply to your coding?

- A test case is a function in a subclass of XCTestCase where the function has the following traits:
  - Its name starts with test
  - It has no parameters, and no return value
  - It isn't private
- Press ⌘-U (think U for "unit test") to run tests.
- An assertion failure marks the test as failing. Otherwise, the test case passes.
- Avoid conditional branches in test code to keep test code simple. You can do this by choosing an assertion that expresses the condition you need.
- When comparing floating-point numbers, use XCTAssertEqual() with an accuracy: argument.
- If your test needs a condition that the built-in assertions don't provide, then put an XCTFail() (with a description message) inside a conditional clause.

2. https://developer.apple.com/documentation/xctest

- Check the failure reporting of your tests. If the description of an object is hard to read, provide a custom description by conforming to the `CustomStringConvertible` protocol.

Activities

Now it's time for you to put this chapter into action. Read through this list, pick one of the activities, and do it. It's only in doing that we actually learn.

1. Read Apple's documentation of test assertions so you know what your other options are.

2. Is there any production code you can begin testing today? Look for low-hanging fruit—functions that use only their input arguments to calculate a return value. This includes failable initializers: write one test checking for a `nil` return value and another for non-`nil`. (Pro tip: Any time you add a new test, make sure you see it fail by temporarily breaking the production code.)

3. If your code already has some unit tests, then do the following:

    a. Read through the tests you have.

    b. Select a simulator, and press ⌘-U to run your own tests. Make sure they all pass. If there are any test failures, delete those tests.

    c. Is each test using the best assertion for the job? Improve any you can.

    d. Check calls to `XCTAssertEqual()` to see if the argument order is consistent. Try to stick to a consistent order for actual/expected.

    e. Look for any `XCTAssertEqual()` assertions that compare floating-point numbers. Do they use the `accuracy` parameter? Add any that are missing.

    f. Consider whether there are any optional assertion messages you can delete because they're redundant.

    g. Add descriptive messages to any `XCTFail()` assertions that are missing them.

    h. If you've changed any assertions or messages, make sure their failure output is helpful. You can check this by introducing temporary errors in either the test code or the production code. Afterward, don't forget to remove these errors, then run the tests to make sure they pass.

What's Next?

We'll talk more about assertions as we put them to use, especially when we make our own test helpers.

You still have plenty of tricks to learn. But you now carry assertions in your tool belt, and you can begin writing simple tests of your own code.

Now that you've written some unit tests, how are these tests run? In the next chapter, we'll clarify common misunderstandings of the life cycle of test cases. You need a mental model that matches what's actually going on. In particular, we'll see how to avoid the most common mistakes that Swift programmers make.