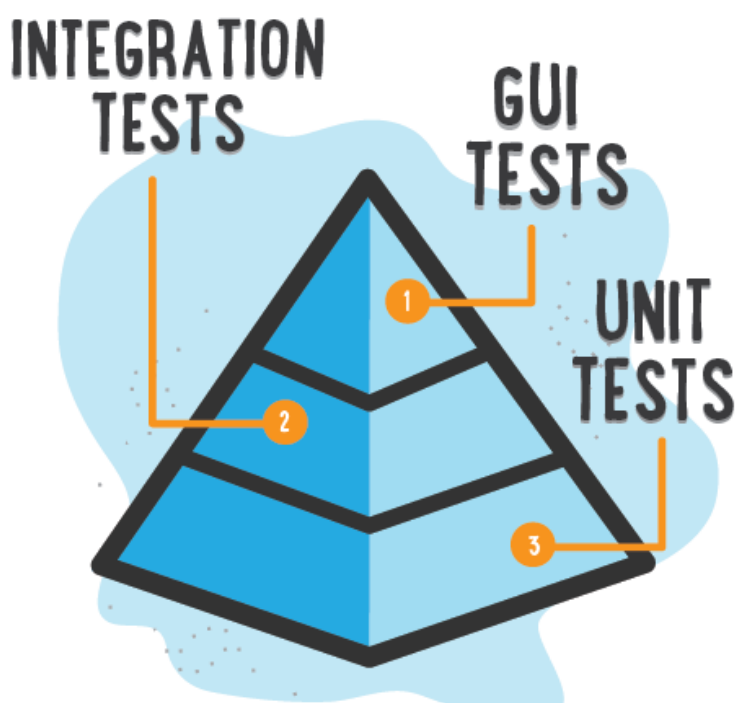
CHAPTER ELEVEN : UNIT TEST OF THE MODEL



I. INTRODUCTION TO SOFTWARE TESTING

In the previous chapter, we have created the timer model of our application. The model is pretty small for now and its functionalities are quite easy to foresee and handle. But let's imaging something big, say for some famous commercial apps with large and complex models, how do we know that the model is well-written before moving on to the next phase of development?

This is the part where software testing comes in handy. Software Test techniques include the process of executing a program or application with the intent of finding software bugs (errors or other defects), and verifying that the software product is fit for use. Software testing involves the execution of a software component or system component to evaluate one or more properties of interest. The quality of software testing procedures is really important because the better they are, the more certainty you'll have in the future development. However, if a piece of code pass the test methods, you can't suppose that it will always work no matter how well-written the tests are.

Unit testing is a software testing method by which individual units of source code, sets of one or more computer program modules together with associated control data, usage procedures, and operating procedures, are tested to determine whether they are fit for use.

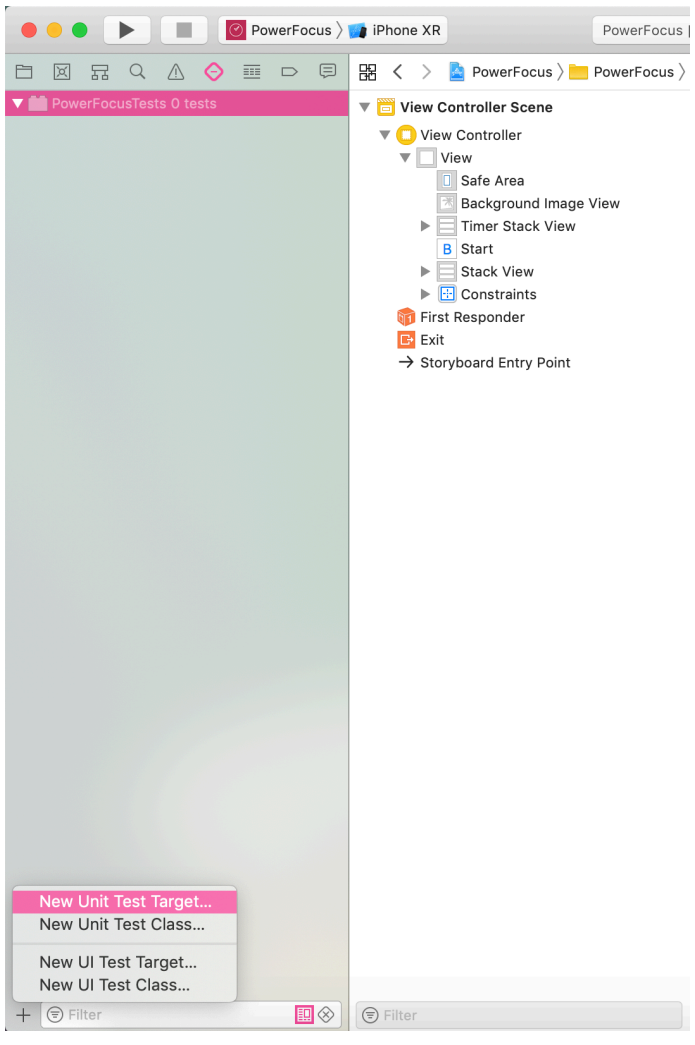
Unit tests are the most basic tests because they focus on the smallest structures and their use cases. They generally follow a bottom up approach in the software hierarchy, which means the general functionalities are tested after all their sub-functionalities are tested (and pass the tests).

There are other types of software testing methods that a software engineer has to know. They are generally more difficult to design and implement than unit tests. In this chapter, we will mainly discuss how to write unit tests, but if you are interested in other types of tests, you can try the exercises at the end of this chapter.

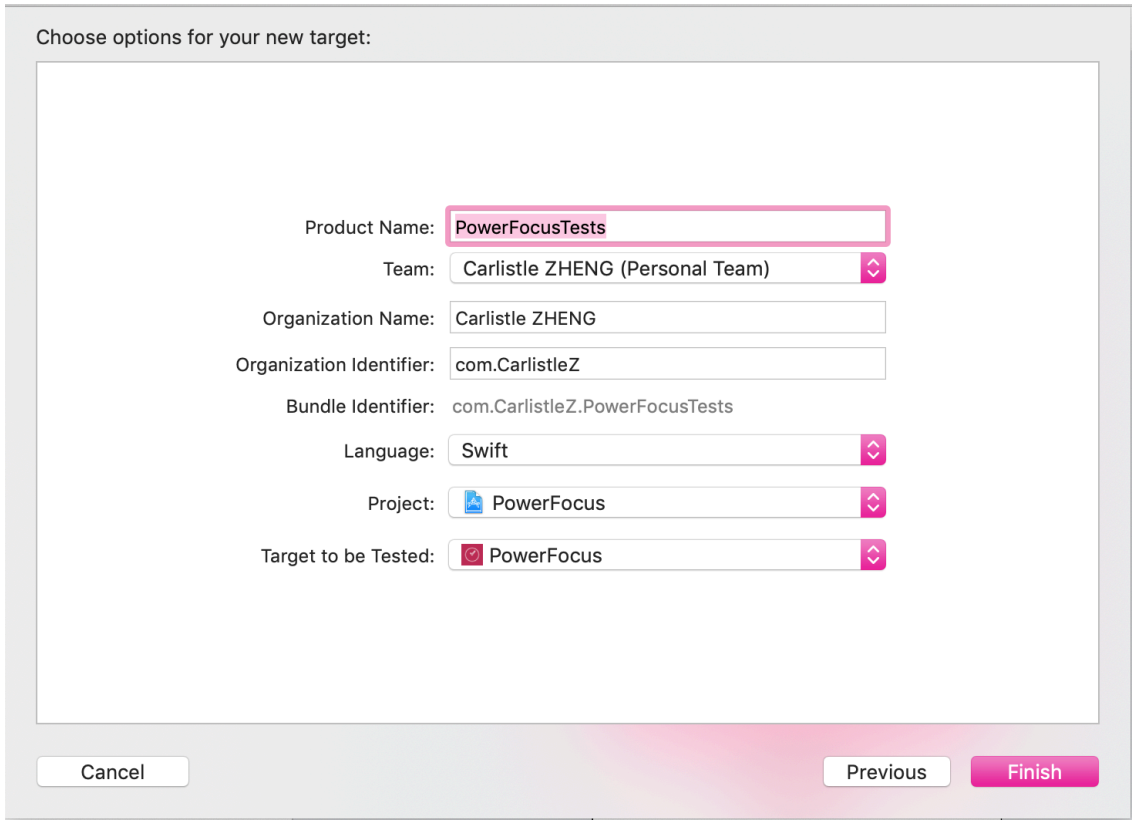
II. CREATE UNIT TESTS IN XCODE

In iOS development, you do not need additional third-party framework to perform unit tests. Xcode and iOS SDK (Software Development Kit) are all you need to write and execute unit tests.

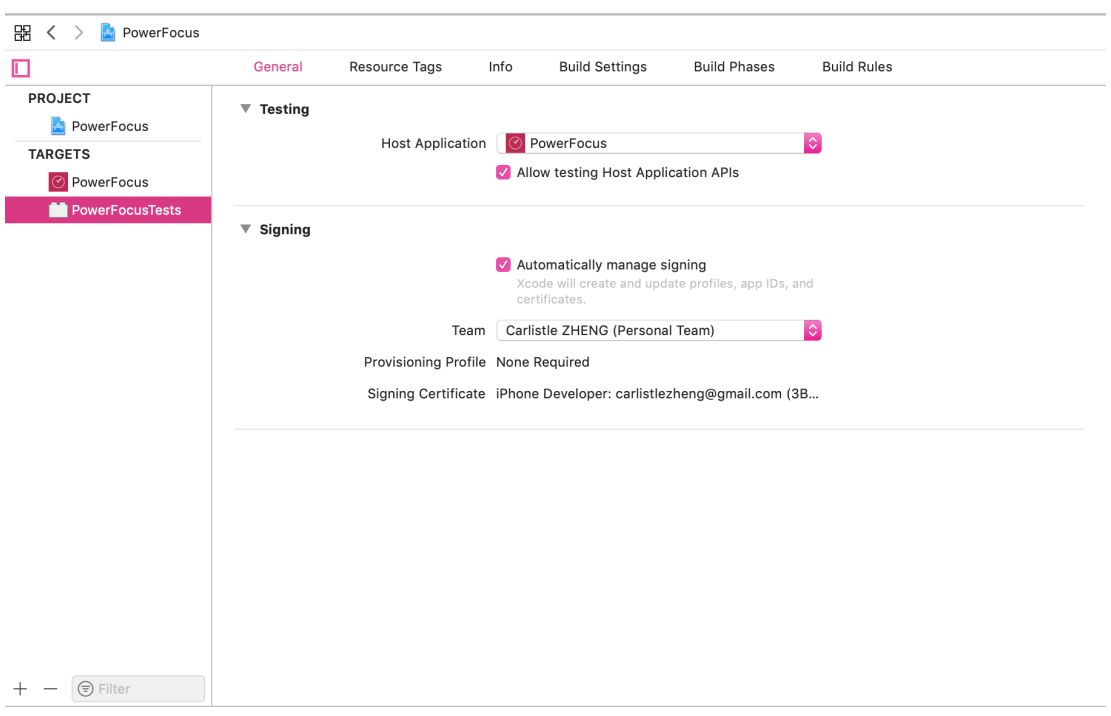
To create a test target, use the **Test Navigator** in the left navigator pane or press **Command(⌘) + 6**. At the bottom of the test navigator, click "**+**" then choose "**New Unit Test Target**".



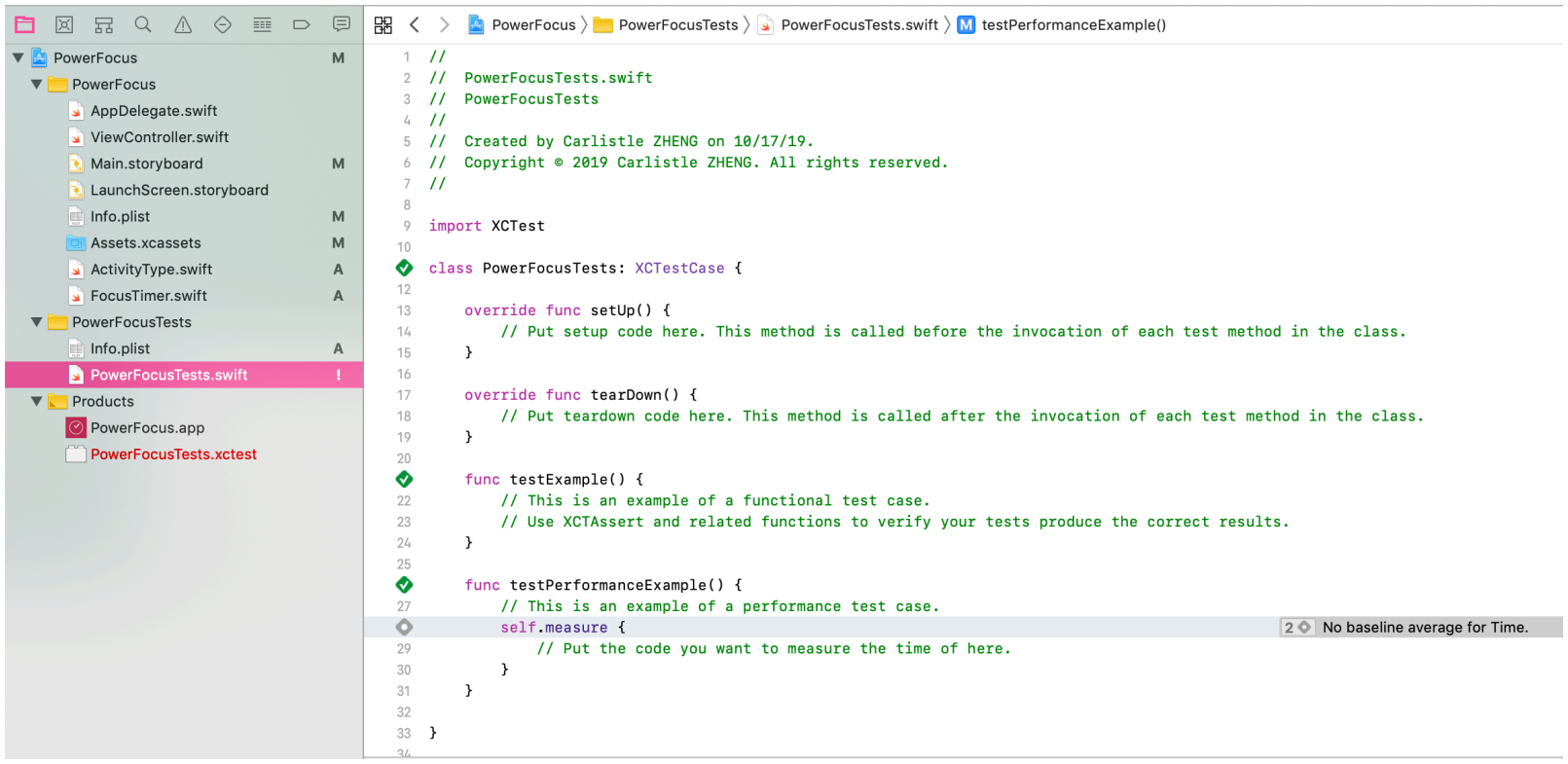
In the next window, there are options to define multiple tests for one target to be tested if it's too big to test all at once. The naming convention here is adding the suffix "**Tests**" to the target name. Simply click Finish since we don't need to modify any default test options here.



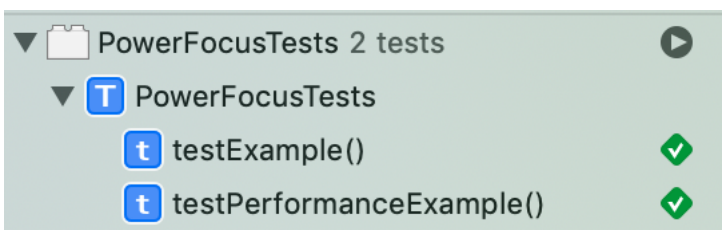
To check if the target is correctly created, click on the project root in the navigator, in the left pane, choose PowerFocusTests as target. Under testing, if the **Host Application** is none, change it to **PowerFocus**, and check "**Allow testing Host Application APIs**".

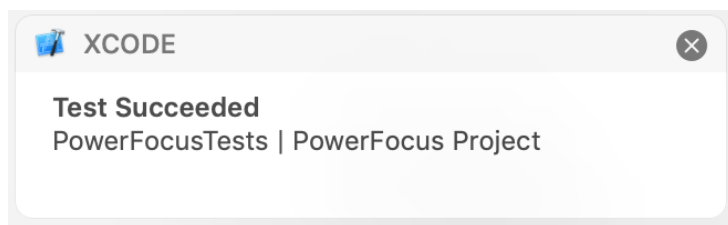


Another thing to check is that the target root is under the PowerFocus project root, not under the build target folder. There is a swift file and an Info.plist file in the folder. Click on PowerFocusTests.swift, you will see the following content:



Click on the check mark on the left of the class declaration or in the test navigator, you will see a notification and the green ticks like this to indicate that it just passed the unit tests under the target:



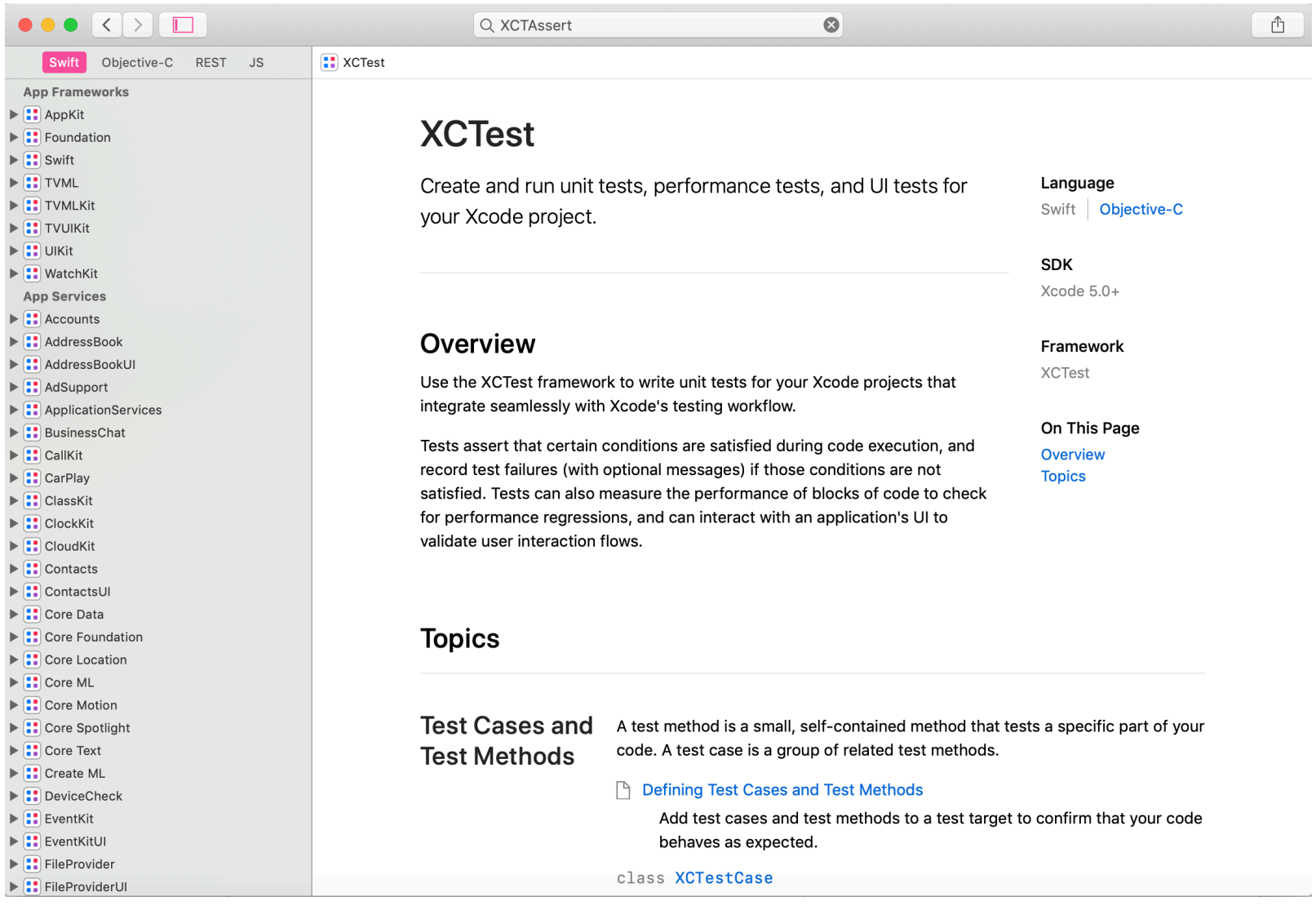


III. BASICS OF UNIT TESTING

The basics of unit testing are really easy to understand. You will need assertions to compare the real results and the desirable results. Go to **Help > Developer Documentation** and search for **XCTest** in the search bar. I highly recommend you to read this entire doc, but if you don't want to dive into too much technical details, you should at least read the section **Test Cases and Test Methods**.

You can also see the online version of the documentation in the following link:

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



For a very simple example, we define a class StringHolder to test like this:

```
// Example class to test
class StringHolder {

    // string to hold
    private var str = ""

    init(_ str: String) {
        self.str = str
    }

    // Default constructor
    init() {
        self.str = "default"
    }

    func setStr(to newVal: String) {
        self.str = newVal
    }

    func getStr() -> String {
        return self.str
    }
}
```

This class has a private attribute str: String that can be accessed with a public getter and setter. It has a default constructor that initializes the string to “default” and a constructor with a string parameter.

To use any method for unit tests, before all the class declarations, put **import XCTest** because XCTests are not automatically imported for efficiency reason. In most cases, you should subclass XCTestCase directly when defining tests in your project. A test class should normally override these important methods:

- func setUp(): Provides an opportunity to reset state before each test method in a test case is called. The setUp() instance method is called once before each test begins. Override this method to reset state for each test method.
- func tearDown(): Provides an opportunity to perform cleanup after each test method in a test case ends. The tearDown() instance method is called once after each test completes. Override this method to perform any per-test cleanup.
- func testSomeMethod(): a test method with some assertions. Normally a test class has several test methods. Each test method should be as "atomic" as possible, which means a test should test one possible case (or one state).

To test the class StringHolder, we need an instance of StringHolder in the test class. This instance should be initiated in the method setUp() and destroyed in the method tearDown(). However, this instance can't be a local variable of the setUp method because it should be visible to all the test methods. So we define the variable var holder like this:

```
class PowerFocusTests: XCTestCase {
    var holder = StringHolder("")
    // test methods..
}
```

For the setUp method, normally we should construct the instance to test. This method is called before **EACH** test method, which means if the instance is changed in a test method, the changes won't persist to other methods. This mechanism makes sure that test methods are independent and they share the same starting point: the setUp method.

In this test class, we only need to instantiate the attribute holder by calling the class StringHolder's constructor:

```
    override func setUp() {
        // Put setup code here. This method is called before the invocation of each test method in
the class.
        holder = StringHolder("Hello Swift!")
    }
```

A tearDown method does the opposite of the setUp method. We don't need to do anything here because the attribute is fully constructed in setUp. So leave the tearDown method like this:

```
    override func tearDown() {
        // Put teardown code here. This method is called after the invocation of each test method
in the class.
    }
```

Each test case should test an atomic case as simple as possible. This is because we want to know exactly where it doesn't pass the test. If there are several assertions in a test method

and the test fails, we don't really know which assertion fails the test. And we don't have any persistence among assertions in the same test method, this can also cause some problems during runtime.

Test methods use test assertions to check for expected values and outcomes. These are some assertion types:

- **Boolean Assertions**: Test a condition that generates a true or false result.
  func XCTAssertTrue(expression: Bool?)
  func XCTAssertFalse(expression: Bool?)
- **Nil and Non-nil Assertions**: Check whether a test condition is nil or non-nil.
  func XCTAssertNil(expression: Any?)
  func XCTAssertNotNil(expression: Any?)
- **Equality and Inequality Assertions**: Check whether two values are equal or unequal.
  func XCTAssertEqual<T>(expression1: Equatable, expression2: Equatable)
  func XCTAssertNotEqual<T>(expression1: Equatable, expression2: Equatable)
- **Comparable Value Assertions**: Compare two values to determine whether one is larger or smaller than the other.
  func XCTAssertGreaterThan<T>(expression1: Comparable, expression2: Comparable)
  func XCTAssertGreaterThanOrEqual<T>(expression1: Comparable, expression2: Comparable)
  func XCTAssertLessThan<T>(expression1: Comparable, expression2: Comparable)
  func XCTAssertLessThanOrEqual<T>(expression1: Comparable, expression2: Comparable)
- **Error Assertions**: Check whether a function call throws (or does not throw) an error.
  func XCTAssertThrowsError<T>(expression: T)
  func XCTAssertNoThrow<T>(expression: T)
- **Failing Unconditionally**: Generate a failure immediately and unconditionally.
  func XCTFail(): void

To test a class that can be instantiated, we should always check if the instance is correctly instantiated. If it has several constructors with different parameters (including a default constructor with no parameter), each constructor should have at lease one test method. Testing all the constructors is really important, but XCTestCase doesn't support multiple setUp methods. So we need to call different constructors ourselves in constructor tests instead of using the result from setUp. We need two tests for constructors, one for default constructor and another for the constructor with parameter.

```
func testInit() {
    // This is an example of a functional test case.
    // Use XCTAssert and related functions to verify your tests produce the correct results.
    XCTAssertEqual("Hello Swift!", holder.getStr())
}

func testDefaultInit() {
    holder = StringHolder()
    XCTAssertEqual("default", holder.getStr())
}
```

We also need to test the getter and the setter of the str attribute like this:

```
func testGet() {
    XCTAssertEqual("Hello Swift!", holder.getStr())
```

```
    }

    func testSet() {
        holder.setStr(to: "Hello Objective C!")
        XCTAssertEqual("Hello Objective C!", holder.getStr())

    }
```

In these methods, we use the XCTAssertEqual method to compare the expected value and the real value. In some complex cases the expected value can be a little hard to imagine, several individual tests should be created to test incrementally.

There are some conventions in unit testing, violating these rules can make the tests less readable and they are often considered as bad practices. Here are some common mistakes in assertion usage:

```
Example 1:
  let a = 1
  let b = 1
  // Good practice: use XCAssertEqual directly
  XCTAssertEqual(a, b)

  // Bad practice: use XCAssertEqual indirectly
  XCTAssertEqual(a - b, 0)

Example 2:
  let bl = true
  // Good practice: use XCTAssertTrue if possible
  XCTAssertTrue(bl)

  // Bad practice: use XCTAssertEquals to true
  XCTAssertEqual(true, bl)

Example 3:
  // Bad practice: several separatable assertions in a test
  var a = 1
  XCTAssertEqual(1, a)
  a += 1
  XCTAssertEqual(2, a)

  // Good practice: one assertion per test
  // testOne
  var a = 1
  XCTAssertEqual(1, a)

  // testTwo
  var a = 1
  a += 1
  XCTAssertEqual(2, a)
```

IV. TEST FOCUSTIMER

Since the test class and the FocusTimer model are not in the same class, we need to import the FocusTimer for the test class. Add the following annotation before the test class declaration:

```
@testable import PowerFocus
```

Inside the test class, add the attribute to test:

```
var focusTimer: FocusTimer!
```

Go to **Project > Build** and build the project or use the shortcut **Command(⌘) + B**, if you have imported the class properly, you will be able to use the class FocusTimer without any error. If Xcode can't find the module, check **PowerFocus > Targets > PowerFocus > Build Settings > Packaging > Product Package Name** is the same as the name you use in @testable. If it's not PowerFocus, change it to PowerFocus then rebuild.

Just like what we did for StringHolder, construct the object in the setUp method.

```
override func setUp() {
    focusTimer = FocusTimer(20, 10, 15, 5, true)
}
```

We need these following tests to check if the attributes are correctly initiated by the constructor's parameters used in the setUp method.

```
func testInitWorkLength() {
    XCTAssertEqual(20, focusTimer.workLength)
}

func testInitBreakLength() {
    XCTAssertEqual(10, focusTimer.breakLength)
}

func testInitLongBreakLength() {
    XCTAssertEqual(15, focusTimer.longBreakLength)
}

func testInitLongBreakFreq() {
    XCTAssertEqual(5, focusTimer.longBreakFreq)
}

func testInitHasLongBreak() {
    XCTAssertTrue(focusTimer.hasLongBreak)
}
```

Since we have a constructor with default parameters, we need a separate test method to test these default values. Unlike the previous tests, we have to use the constructor without indicating the default values in each test.

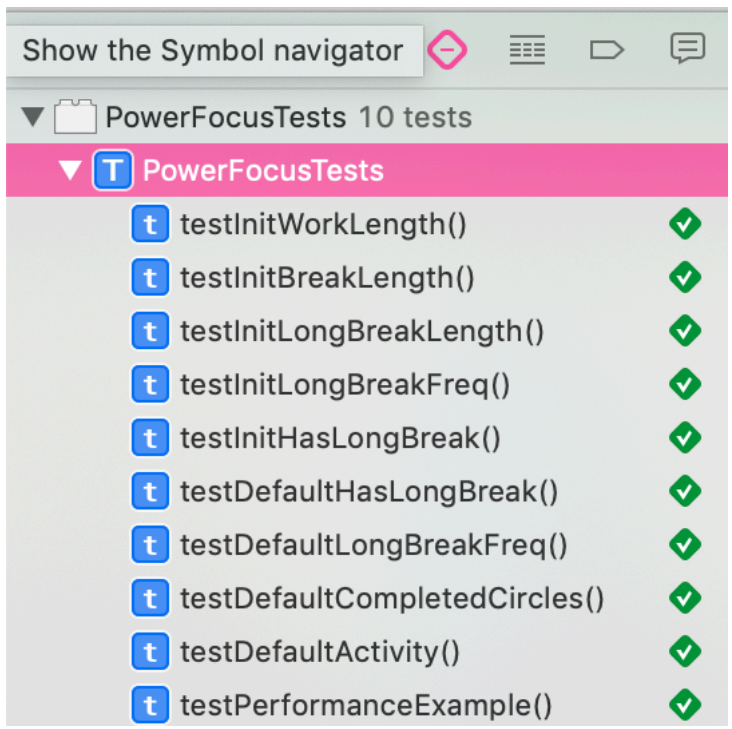
```
func testDefaultHasLongBreak() {
    focusTimer = FocusTimer(25, 5, 15)
    XCTAssertTrue(focusTimer.hasLongBreak)
}

func testDefaultLongBreakFreq() {
    focusTimer = FocusTimer(25, 5, 15)
    XCTAssertEqual(3, focusTimer.longBreakFreq)
}
```

We also have to check that the constructor correctly initializes the completed cycles to zero and the starting activity to work.

```
    func testDefaultCompletedCircles() {
        XCTAssertEqual(0, focusTimer.getCompletedCircles())
    }

    func testDefaultActivity() {
        XCTAssertEqual(ActivityType.work, focusTimer.getActivity())
    }
```

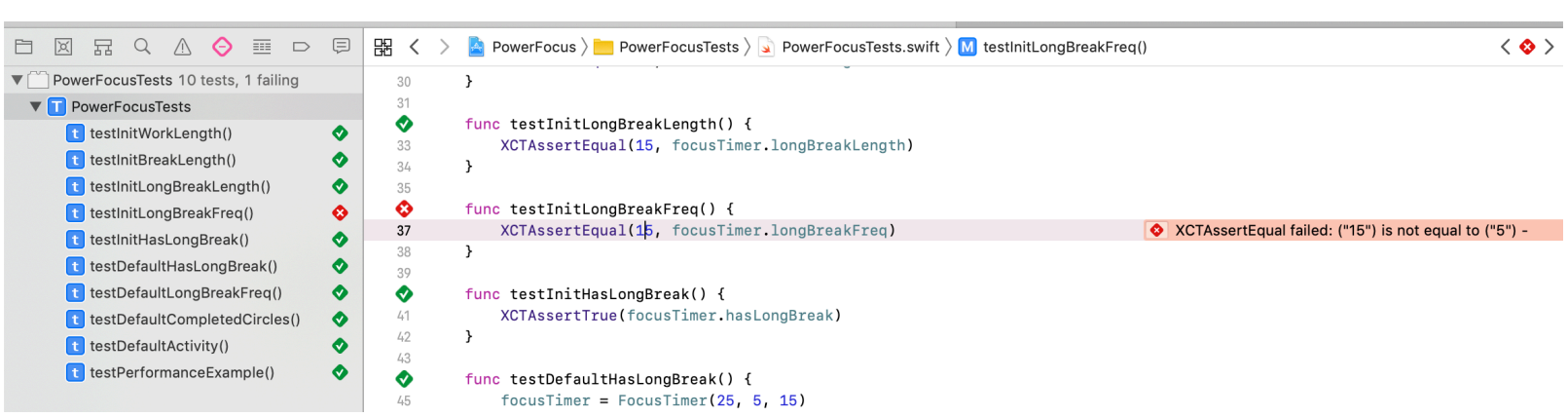
For now, these tests should cover all the possibilities of the model's states. When we add more methods to the FocusTimer, we will write the corresponding tests to test these new methods.

Open the test navigator in the left pane or press **Command(⌘) + 6**. Start the PowerFocusTests class by pressing the start button on the right, you will see all the results of the test methods with assertion.



If your model is correctly written, all tests should pass. If not, go to the model and try to fix the issue. You can check fail details in the popup on the right. For example, change the expected value of long break frequency from 5 to 15. Re-run this test and you will see the following error message: XCAssertEqual failed: ("15") is not equal to ("5").

You can check the value during runtime and compare it to your expected value. You should bear it in mind that when a unit test fails, it doesn't always mean that your model has a bug. In this case, the expected value in test method is incorrect. Change 15 back to 5, the test should pass again.



V. UI TESTS

In iOS development, testing the apps only with unit tests is not enough. Since a lot of modules are UI related, we need to test if the UI is correctly displayed.

The idea of UI tests is very similar to unit tests, but the additional functionality is user interactions in the user interface. To write automatic tests, we need to locate the button. In the following example, we look for a button “Slide” in the app. The method tap simulates the tap gesture of a user while the app is running. By calling this method, you are able to test your code that runs after the tap gesture.

```
let app = XCUIApplication()
app.buttons["Share"].tap()
app.staticTexts[“A line of static text: "].tap()
```

Just like what we did in unit test methods, UI tests check the code with assertions. In the following example, if you want to remove a widget from a view after a button tap, you should use the attribute .exists to check if the widget is correctly removed.

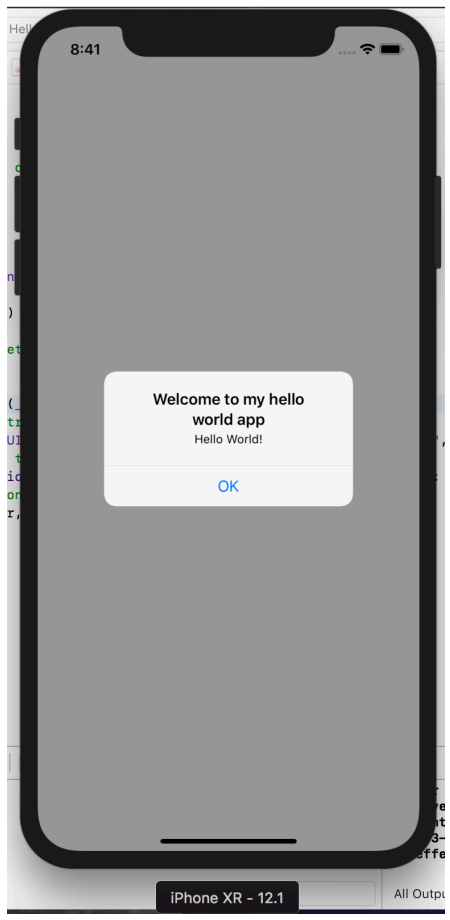
```
// Assertions before and after the tap
if slideButton.isSelected {
  XCTAssertTrue(firstLabel.exists)
  XCTAssertFalse(secondLabel.exists)

  typeButton.tap()
  XCTAssertTrue(secondLabel.exists)
  XCTAssertFalse(firstLabel.exists)
}
```

There are many methods in the iOS SDK that allow developers to write automated tests on user interface and user interaction. In general, these tests are harder to design than the unit test. So you should always try to write small, atomic tests that separate the assertions.

EXERCISE

In Chapter 4, we created a simple program that displays the message “Hello World!” in an alert box. We only tested it out by looking at the visual result. However, this doesn’t always guarantee that the message will always be correctly displayed. So we need an UI test to test it out.

Write two test cases: you should first check if the text exists in the view, then you should check the text content: Hello World!



SUMMARY

In this chapter, we have worked on these topics:

- introduction to software testing
- target for tests
- Unit Test
- UI Test

If any of these is unclear to you, please make sure to go back and read the related part or parts before moving on the the next chapter.

If you can't figure out the exercises on your own, you can check the answers at

https://github.com/CarlistleZ/MyiOSTutorial/blob/master/Chapter_Ten_Exercises.zip

For reference, you can download the complete Xcode project from

https://github.com/CarlistleZ/MyiOSTutorial/blob/master/Chapter_Ten_PowerFocus.zip