Up to date for iOS 13, Xcode 11 & Swift 5.1

iOS Test-Driven Development by Tutorials

FIRST EDITION

Learn Real-World Test-Driven Development

By the raywenderlich Tutorial Team

Joshua Greene & Michael Katz

iOS Test-Driven Development by Tutorials

By Joshua Greene & Michael Katz

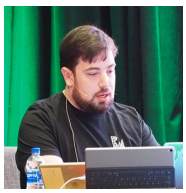
Notice of Rights

Notice of Liability

Trademarks

About the Authors

Joshua Greene is an author of this book. He's an experienced software developer and has created many mobile apps. When he's not slinging code, you can find him wandering the streets of Tokyo. You can reach him on Twitter @jrg_developer.



Michael Katz is a champion baker. ;] Oh, he's also an author of this book, developer, architect, speaker, writer and avid homebrewer. He has contributed to several books on iOS development and is a long-time member of the raywenderlich.com tutorial team. He's currently serving as director of mobile engineering at Viacom. He shares his home state of New York with his family, the world's best bagels and the Yankees. When he's not at his computer, he's out on the trails, in his shop or reading a good book (like this one!).

About the Editors

Darren Ferguson is the final pass editor for this book. He is an experienced software developer and works for M.C. Dean, Inc, a systems integration provider from North Virginia. When he's not coding, you'll find him enjoying EPL Football, traveling as much as possible and spending time with his wife and daughter.

Manda Frederick is the editor of this book. She has been involved in publishing for over ten years through various creative, educational, medical and technical print and digital publications, and is thrilled to bring her experience to the raywenderlich.com family as Managing Editor. In her free time, you can find her at the climbing gym, backpacking in the backcountry, hanging with her dog, working on poems, playing guitar and exploring breweries.

Jeff Rames is a tech editor for this book. He's an enterprise software developer in San Antonio, Texas who's focused on iOS for nearly a decade. He spends his free time with his wife and daughters, except when he abandons them for trips to Cape Canaveral to watch rocket launches. Say hi on Twitter @jefframes!

James Taylor is a tech editor for this book. He's an iOS developer living in San Antonio, Texas with both his wife and daughter. He enjoys bicycle touring around the United States and spending way too much time on YouTube. You can find him on Twitter @jamestaylorios.

About the Artist

Vicki Wenderlich is the designer and artist of the cover of this book. She is Ray's wife and business partner. She is a digital artist who creates illustrations, game art and a lot of other art or design work for the tutorials and books on raywenderlich.com. When she's not making art, she loves hiking, a good glass of wine and attempting to create the perfect cheese plate.

Dedications

"For my girls. I love you very much."

— Joshua Greene

"Dedicated to the memory of my mother-in-law, Barbara Schwartz. Her selflessness and dedication to teaching inspires me to give back to the community and educate others."

— Michael Katz

Table of Contents: Overview

Table of Contents: Extended

Book License

By purchasing *iOS Test-Driven Development by Tutorials*, you have the following license:

- You are allowed to use and/or modify the source code in *iOS Test-Driven Development by Tutorials* in as many apps as you want, with no attribution required.
- You are allowed to use and/or modify all art, images and designs that are included in *iOS Test-Driven Development by Tutorials* in as many apps as you want, but must include this attribution line somewhere inside your app: "Artwork/images/designs: from *iOS Test-Driven Development by Tutorials*, available at www.raywenderlich.com".
- The source code included in *iOS Test-Driven Development by Tutorials* is for your personal use only. You are NOT allowed to distribute or sell the source code in *iOS Test-Driven Development by Tutorials* without prior authorization.
- This book is for your personal use only. You are NOT allowed to sell this book without prior authorization, or distribute it to friends, coworkers or students; they would need to purchase their own copies.

All materials provided with this book are provided on an "as is" basis, without warranty of any kind, express or implied, including but not limited to the warranties of merchantability, fitness for a particular purpose and noninfringement. In no event shall the authors or copyright holders be liable for any claim, damages or other liability, whether in an action or contract, tort or otherwise, arising from, out of or in connection with the software or the use or other dealings in the software.

All trademarks and registered trademarks appearing in this guide are the properties of their respective owners.

Welcome to *iOS Test-Driven Development*!

iOS Test-Driven Development introduces you to a broad range of concepts with regard to not only writing an application from scratch with testing in mind, but also applying these concepts to already written applications which have little or no tests written for their functionality.

This book is for intermediate iOS developers who already know the basics of iOS and Swift development but want to learn how to write code which is both testable and maintainable.

We are pleased to offer you this sample from the full *iOS Test-Driven Development* book that will introduce you to these concepts and give you a chance to practice them in our hands-on By Tutorials style.

This sample includes:

2. **The TDD Cycle:** Introduces the four steps of the **TDD Cycle**, which has four individual steps that are normally called the **Red-Green-Refactor-Cycle**.
3. **TDD App Setup:** Introduces the concept of setting up your application within Xcode to run tests including setting up your test target and writing a few tests.
4. **Test Expressions:** Introduces the concept of how to use the **XCTAssert** functions which are the primary actors of the test infrastructure.

Get the full book here:

- https://store.raywenderlich.com/products/ios-test-driven-development.

— The *iOS Test-Driven Development* Team

B Book Source Code & Forums

If you bought the digital edition

The digital edition of this book comes with the source code for the starter and completed projects for each chapter. These resources are included with the digital edition you downloaded from https://store.raywenderlich.com/products/ios-test-driven-development.

If you bought the print version

You can get the source code for the print edition of the book here:

- https://store.raywenderlich.com/products/ios-test-driven-development-source-code

Forums

We've also set up an official forum for the book at forums.raywenderlich.com.This is a great place to ask questions about the book or to submit any errors you may find.

Digital book editions

We have a digital edition of this book available in both ePUB and PDF, which can be handy if you want a soft copy to take with you, or you want to quickly search for a specific term within the book.

Buying the digital edition version of the book also has a few extra benefits: free updates each time we update the book, access to older versions of the book, and you can download the digital editions from anywhere, at anytime.

Visit our *iOS Test-Driven Development* store page here:

- https://store.raywenderlich.com/products/ios-test-driven-development.

And if you purchased the print version of this book, you're eligible to upgrade to the digital editions at a significant discount! Simply email support@razeware.com with your receipt for the physical copy and we'll get you set up with the discounted digital edition version of the book.

What You Need

To follow along with this book, you'll need the following:

- **Xcode 11 or later.** Xcode is the main development tool for writing code in Swift. You need Xcode 11 at a minimum, since that version includes Swift 5.1. You can download the latest version of Xcode for free from the Mac App Store, here: apple.co/1FLn51R.

If you haven't installed the latest version of Xcode, be sure to do that before continuing with the book. The code covered in this book depends on Swift 5.1 and Xcode 11 — the code may not compile if you try to work with an older version.

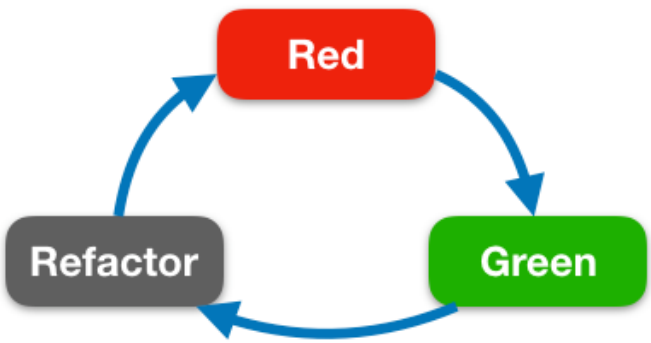
Chapter 2: The TDD Cycle

By Joshua Greene

In the previous chapter, you learned that test-driven development boils down to a simple process called the **TDD Cycle.** It has four steps that are often "color coded" as follows:

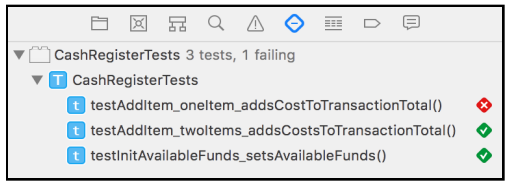
1. **Red**: Write a failing test, before writing any app code.
2. **Green**: Write the bare minimum code to make the test pass.
3. **Refactor**: Clean up both your app and test code.
4. **Repeat**: Do this cycle again until all features are implemented.

This is also called the **Red-Green-Refactor Cycle.**



Why is it color coded? This corresponds to the colors shown in most code editors, including Xcode:

- Failing tests are indicated with a **red X**.
- Passing tests are shown with a **green checkmark**.



This chapter provides an introduction to the TDD Cycle, which you'll use throughout the rest of this book. However, it doesn't go into detail about test expressions (`XCTAssert`, et al.) or how to set up a test target. Rather, these topics are covered in later chapters. For now, focus on learning the TDD Cycle, and you'll learn the rest as you go along.

It's best to learn by doing, so let's jump straight into code!

Getting started

In this chapter, you'll create a simple version of a cash register to learn the TDD Cycle. To keep the focus on TDD instead of Xcode setup, you'll use a playground. Open **CashRegister.playground** in the **starter** directory, then open the **CashRegister** page. You'll see this page has two imports, but otherwise it's empty.

Naturally, you'll begin with the first step in the TDD Cycle: red.

Red: Write a failing test

Before you write *any* production code, you must first write a failing test. To do so, you need to create a test class. Add the following below the import statements:

```
class CashRegisterTests: XCTestCase {

}
```

Above, you declare `CashRegisterTests` as a subclass of `XCTestCase`, which is part of the `XCTest` framework. You'll almost always subclass `XCTestCase` to create your test classes.

Next, add the following at the end of the playground:

```
CashRegisterTests.defaultTestSuite.run()
```

This tells the playground to run the test methods defined within `CashRegisterTests`. However, you haven't actually written any tests yet. Add the following within `CashRegisterTests`, which should cause a compiler error:

```
// 1
func testInit_createsCashRegister() {
  // 2
  XCTAssertNotNil(CashRegister())
}
```

Here's a line-by-line explanation:

1. Tests are named per this convention throughout the book:

- `XCTest`: Requires all test methods begin with `test` to be run.
- `test`: Followed by the name of the method being tested. Here, this is `init`. There's then an underscore to separate it from the next part.
- Optionally, if special set up is required, this comes next. This test *doesn't* include this. If provided, this likewise is followed by an underscore to separate it from the last part.
- Lastly, this is followed by the expected outcome or result. Here this is `createsCashRegister`.

This convention results in test names that are easy to read and provide meaningful context. If a test ever fails, Xcode will tell you the name of the test's class and method. By naming your tests this way, you can quickly determine the problem.

2. You then attempt to instantiate a new instance of `CashRegister`, which you pass into `XCTAssertNil`. This is a test expression that asserts whatever passed to it is *not* `nil`. If it actually is `nil`, the test will be marked as failed.

However, this last line doesn't compile! This is because you haven't created a class for `CashRegister` just yet... how are you suppose to advance the TDD Cycle, then? Fortunately, there's a rule in TDD for this: Compilation failures count as test failures. So, you've completed the red step in the TDD Cycle and can move onto the next step: green.

Green: Make the test pass

You're only allowed to write the *bare minimum code* to make a test pass. If you write more code than this, your tests will fall behind your app code. What's the bare minimum code you can write to fix this compilation error? Define `CashRegister`!

Add the following directly above `class CashRegisterTests`:

```
class CashRegister {

}
```

Press **Play** to execute the playground, and you should see output like this in the console:

```
Test Suite 'CashRegisterTests' started at
  2019-01-02 18:25:57.661
Test Case '-[__lldb_expr_3.CashRegisterTests
  testInit_createsCashRegister]' started.
Test Case '-[__lldb_expr_3.CashRegisterTests
  testInit_createsCashRegister]' passed (0.130 seconds).
Test Suite 'CashRegisterTests' passed at
  2019-01-02 18:25:57.792.
    Executed 1 test, with 0 failures (0 unexpected) in 0.130
    (0.131) seconds
```

Awesome, you've made the test pass! The next step is to refactor your code.

Refactor: Clean up your code

You'll clean up both your app code and test code in the refactor step. By doing so, you constantly maintain and improve your code. Here are a few things you might look to refactor:

- **Duplicate logic**: Can you pull out any properties, methods or classes to eliminate duplication?
- **Comments**: Your comments should explain *why* something is done, not *how* it's done. Try to eliminate comments that explain *how* code works. The how should be conveyed by breaking up large methods into several well-named methods, renaming properties and methods to be more clear or sometimes simply structuring your code better.

- **Code smells**: Sometimes a particular block of code simply seems wrong. Trust your gut and try to eliminate these "code smells." For example, you might have logic that's making too many assumptions, uses hardcoded strings or has other issues. The tricks from above apply here, too: Pulling out methods and classes, renaming and restructuring code can go a long way to fixing these problems.

Right now, `CashRegister` and `CashRegisterTests` don't have much logic in them, and there isn't anything to refactor. So, you're done with this step — that was easy! The most important step in the TDD Cycle happens next: repeat.

Repeat: Do it again

Use TDD throughout your app's development to get the most benefit from it. You'll accomplish a little bit in each TDD Cycle, and you'll build up app code backed by tests. Once you've completed all of your app's features, you'll have a working, well-tested system.

You've completed your first TDD Cycle, and you now have a class that can be instantiated: `CashRegister`. However, there's still more functionality to add for this class to be useful. Here's your to-do list:

- Write an initializer that accepts `availableFunds`.
- Write a method for `addItem` that adds to a transaction.
- Write a method for `acceptPayment`.

You've got this!

TDDing init(availableFunds:)

Just like every TDD cycle, you first need to write a failing test. Add the following below the previous test, which should generate a compiler error:

```
  func testInitAvailableFunds_setsAvailableFunds() {
    // given
    let availableFunds = Decimal(100)

    // when
    let sut = CashRegister(availableFunds: availableFunds)

    // then
```

```
    XCTAssertEqual(sut.availableFunds, availableFunds)
  }
```

This test is more complex than the first, so you've broken it into three parts: **given**, **when** and **then**. It's useful to think of unit tests in this fashion:

- **Given** a certain condition...
- **When** a certain action happens...
- **Then** an expected result occurs.

In this case, you're **given** `availableFunds` of `Decimal(100)`. **When** you create the `sut` via `init(availableFunds:)`, **then** you expect `sut.availableFunds` to equal `availableFunds`.

What's the name `sut` about? `sut` stands for **system under test.** It's a very common name used in TDD that represents whatever you're testing. This name is used throughout this book for this very purpose.

This code doesn't compile yet because you haven't defined `init(availableFunds:)`. Compilation failures are treated as test failures, so you've completed the red step.

You next need to get this to pass. Add the following code inside `CashRegister`:

```
  var availableFunds: Decimal

  init(availableFunds: Decimal = 0) {
    self.availableFunds = availableFunds
  }
```

`CashRegister` can now be initialized with `availableFunds`.

Press **Play** to execute all of the tests, and you should see output like this in the console:

```
  Test Suite 'CashRegisterTests' started at
    2019-01-02 18:29:25.888
  Test Case '-[__lldb_expr_7.CashRegisterTests
    testInit_createsCashRegister]' started.
  Test Case '-[__lldb_expr_7.CashRegisterTests
    testInit_createsCashRegister]' passed (0.129 seconds).
  Test Case '-[__lldb_expr_7.CashRegisterTests
    testInitAvailableFunds_setsAvailableFunds]' started.
  Test Case '-[__lldb_expr_7.CashRegisterTests
    testInitAvailableFunds_setsAvailableFunds]' passed
    (0.004 seconds).
  Test Suite 'CashRegisterTests' passed at
```

```
2019-01-02 18:29:26.022.
  Executed 2 tests, with 0 failures (0 unexpected) in 0.133
   (0.134) seconds
```

This shows both tests pass, so you've completed the green step.

You next need to clean up both your app and test code. First, take a look at the test code.

`testInit_createsCashRegister` is now obsolete: There isn't an `init()` method anymore. Rather, this test is actually calling `init(availableFunds:)` using the default parameter value of `0` for `availableFunds`.

Delete `testInit_createsCashRegister` entirely.

What about the app code? Does it make sense to have a default parameter value of `0` for `availableFunds`? This was useful to get both `testInit` and `testInitAvailableFunds` to compile, but should this class actually have this?

Ultimately, this is a design decision:

- If you choose to keep the default parameter, you might consider adding a test for `testInit_setsDefaultAvailableFunds`, in which you'd verify `availableFunds` is set to the expected default value.
- Alternatively, you might choose to remove the default parameter, if you decide it doesn't make sense to have this.

For this example, assume that it doesn't make sense to have a default parameter. So, **delete** the default parameter value of `0`. Your initializer should then look like this:

```
  init(availableFunds: Decimal) {
```

Press **Play** to execute your remaining test, and you'll see it passes.

The fact that `testInitAvailableFunds` still passes after refactoring `init(availableFunds:)` gives you a sense of security that your changes didn't break existing functionality. This added confidence in refactoring is a major benefit of TDD!

You've now completed the refactor step, and you're ready to move onto the next TDD Cycle.

TDDing addItem

You'll next TDD `addItem` to add an item's cost to a transaction. As always, you first need to write a failing test. Add the following below the previous test, which should generate compiler errors:

```
func testAddItem_oneItem_addsCostToTransactionTotal() {
  // given
  let availableFunds = Decimal(100)
  let sut = CashRegister(availableFunds: availableFunds)

  let itemCost = Decimal(42)

  // when
  sut.addItem(itemCost)

  // then
  XCTAssertEqual(sut.transactionTotal, itemCost)
}
```

This test doesn't compile because you haven't defined `addItem(_:)` or `transactionTotal` yet.

To fix this, add the following property right after `availableFunds` within `CashRegister`:

```
var transactionTotal: Decimal = 0
```

Then, add this code right after `init(availableFunds:)`:

```
func addItem(_ cost: Decimal) {
  transactionTotal = cost
}
```

Here, you set `transactionTotal` to the passed-in `cost`. But wait — that's not exactly right, or is it?

Remember how you're supposed to write the bare minimum code to get a test to pass? In this case, the bare minimum code required to add a single transaction is setting `transactionTotal` to the passed-in `cost` of the item, not adding it! Thereby, this is what you did.

Press **Play**, and you should see console output indicating all tests have passed. This is technically correct — for *one item*. Just because you've completed a single TDD Cycle doesn't mean that you're done. Rather, you must implement *all* of your app's features before you're done!

In this case, the missing "feature" is the ability to add *multiple* items to a transaction. Before you do this, you need to finish the current TDD cycle by refactoring what you've written.

Start by looking over your test code. Is there any duplication? There sure is! Check out these lines:

```
let availableFunds = Decimal(100)
let sut = CashRegister(availableFunds: availableFunds)
```

This code is common to both `testInitAvailableFunds` and `testAddItem`. To eliminate this duplication, you'll create instance variables within `CashRegisterTests`.

Add the following right after the opening curly brace for `CashRegisterTests`:

```
var availableFunds: Decimal!
var sut: CashRegister!
```

Just like production code, you're free to define whatever properties, methods and classes you need to refactor your test code. There's even a pair of special methods to "set up" and "tear down" your tests, conveniently named `setUp()` and `tearDown()`.

`setUp()` is called right before each test method is run, and `tearDown()` is called right after each test method finishes.

These methods are the perfect place to move the duplicated logic. Add the following below your test properties:

```
// 1
override func setUp() {
  super.setUp()
  availableFunds = 100
  sut = CashRegister(availableFunds: availableFunds)
}

// 2
override func tearDown() {
  availableFunds = nil
  sut = nil
  super.tearDown()
}
```

Here's what this does:

1. Within `setup()`, you first call `super.setUp()` to give the superclass a chance to do its setup. You then set `availableFunds` and `sut`.

2. Within `tearDown()`, you do the opposite. You first set `availableFunds` and `sut` to `nil`, and you lastly call `super.tearDown()`.

You should always `nil` any properties within `tearDown()` that you set within `setUp()`. This is due to the way the XCTest framework works: It instantiates *each* `XCTestCase` subclass within your test target, and it doesn't release them until all of the test cases have run. Thereby, if you have a many test cases, and you don't set their properties to `nil` within `tearDown`, you'll hold onto the properties' memory longer than you need. Given enough test cases, this can even cause memory and performance issues when running your tests.

You can now use these instance properties to get rid of the duplicated logic in the test methods. Replace the contents of `testInitAvailableFunds` with the following:

```
XCTAssertEqual(sut.availableFunds, availableFunds)
```

Since there's now a single line in this method, it's very easy to read, and this removes the need for the **given** and **when** comments.

Next, replace the contents of `testAddItem` with the following:

```
// given
let itemCost = Decimal(42)

// when
sut.addItem(itemCost)

// then
XCTAssertEqual(sut.transactionTotal, itemCost)
```

Ah, that's much simpler too! By moving the initialization code into `setup()`, you can clearly see this method is simply exercising `addItem(_:)`. Press **Play** to confirm all tests have passed.

This completes the refactoring work, so you're now ready to move onto the next TDD Cycle.

Adding two items

`testAddItem_oneItem` confirms `addItem()` passes for one item, but it won't pass for two... or will it? A new test can definitively prove this.

Add the following test right after the previous one:

```
func testAddItem_twoItems_addsCostsToTransactionTotal() {
  // given
  let itemCost = Decimal(42)
  let itemCost2 = Decimal(20)
  let expectedTotal = itemCost + itemCost2

  // when
  sut.addItem(itemCost)
  sut.addItem(itemCost2)

  // then
  XCTAssertEqual(sut.transactionTotal, expectedTotal)
}
```

This test calls `addItem()` twice, and it validates whether the `transactionTotal` accumulates.

Press **Play**, and you'll see the console output indicates the test failed:

```
Test Case '-[__lldb_expr_14.CashRegisterTests
  testAddItem_twoItems_addsCostsToTransactionTotal]' started.
CashRegister.playground:89: error:
  -[__lldb_expr_14.CashRegisterTests
  testAddItem_twoItems_addsCostsToTransactionTotal] :
  XCTAssertEqual failed: ("20") is not equal to ("62") -
Test Case '-[__lldb_expr_14.CashRegisterTests
  testAddItem_twoItems_addsCostsToTransactionTotal]'
  failed (0.008 seconds).
...
Test Suite 'CashRegisterTests' failed at
  2019-01-02 18:57:04.208.
    Executed 3 tests, with 1 failure (0 unexpected) in 0.141
    (0.142) seconds
```

You next need to get this test to pass. To do so, replace the contents of `addItem(_:)` with this:

```
transactionTotal += cost
```

Here, you've replaced the = operator with += to add to the `transactionTotal` instead of set it. Press the **Play** button again, and you'll now see that all tests pass.

You lastly need to refactor your code. Notice any duplication? How about the `itemCost` variable used in both `addItem` tests? Yep, you should pull this into an instance property.

Add the following below the instance property for `availableFunds` within `CashRegisterTests`:

```
var itemCost: Decimal!
```

Then, add this line right after setting `availableFunds` within `setUp()`:

```
itemCost = 42
```

Since you set this property within `setUp()`, you also must `nil` it within `tearDown`. Add the following right after setting `availableFunds` to `nil` within `tearDown()`:

```
itemCost = nil
```

Next, delete these two lines from `testAddItem_oneItem`:

```
// given
let itemCost = Decimal(42)
```

Likewise, delete this line from `testAddItem_twoItems`:

```
let itemCost = Decimal(42)
```

When you're done, the only `itemCost` to remain should be the instance property defined on `CashRegisterTests`.

See any other duplication within `CashRegisterTests`? What about this line?

```
sut.addItem(itemCost)
```

This is common to both `testAddItem_oneItem` and `testAddItem_twoItems`. Should you try to eliminate this duplication?

Remember how `setUp()` is called before every test method is run? You already have one test method that doesn't require this call, `testInitAvailableFunds`.

As you continue to TDD `CashRegister`, you'll likely write other methods that *won't* need to call `addItem(_:)`. Consequently, you *shouldn't* move this call into `setUp()`.

When to refactor code to eliminate duplication is more an art than an exact science. Do what you think is best while you're going along, but don't be afraid to change your decision later if needed!

Challenge

`CashRegister` is off to a great start! However, there's still more work to do. Specifically, you need a method to accept payment. To keep it simple, you'll only accept cash payments — no credit cards or IOUs allowed!

Your challenge is to TDD this new method, `acceptCashPayment(_ cash:)`.

Try to solve this yourself first without help. If you get stuck, see below for hints.

For this challenge, you need to create two test methods within `CashRegisterTests`.

First, create a test method called `testAcceptCashPayment_subtractsPaymentFromTransactionTotal`. Within this, do the following:

- Call `sut.addItem(_:)` to set up a "transaction in progress."
- Call `sut.acceptCashPayment(_:)` to accept payment.
- Assert `transactionTotal` has the payment subtracted from it.

Then, implement `acceptCashPayment(_:)` within `CashRegister` to make the test pass, and refactor as needed.

Create a second test method called `testAcceptCashPayment_addsPaymentToAvailableFunds`. Therein, do the following:

- Call `sut.addItem(_:)` to set up a current transaction.
- Call `sut.acceptCashPayment(_:)` to accept payment.
- Assert the `availableFunds` has the payment added to it.

Then, update `acceptCashPayment(_:)` to make this test pass, and refactor as needed.

Key points

You learned about the TDD Cycle in this chapter. This has four steps:

1. **Red**: Write a failing test.
2. **Green**: Make the test pass.
3. **Refactor**: Clean up both your app and test code.
4. **Repeat**: Do it again until all of your features are implemented.

Xcode playgrounds are a great way to learn new concepts, just like you learned the TDD Cycle in this chapter. In real-world development, however, you typically create unit test targets within your iOS projects, instead of using playgrounds. Fortunately, TDD works even better with apps than playgrounds!

Continue onto the next section to learn about using TDD in iOS apps.

3 Chapter 3: TDD App Setup

By Michael Katz

By now, you should be either sold on Test-Driven Development (TDD) or at least curious. Following the TDD methodology helps you write clean, concise and correct code. This chapter will guide you through its fundamentals.

The goal of this chapter is to give you a feel for how Xcode testing works by creating a test target with a few tests. You'll do this while learning the key concepts of TDD.

By the end of the chapter, you'll be able to:

- Create a test target and run unit tests.
- Write unit tests that verify data and state.

About the FitNess app

In this book section, you'll build up a fun step-tracking app: FitNess. FitNess is the premier fitness-coaching app based on the "Loch Ness" workout. Users have to outrun, outswim or outclimb Nessie, the fitness monster. The goal of the app is to motivate user movement by having them outpace Nessie. If they fail, their avatar gets eaten.

Start with the starter project for Chapter 3. This is a shell app. It comes with some things already wired up to save you some busy work. It's mostly bare-bones since the goal is to lead development with writing tests. If you build and run, the app won't do anything.

Your first test

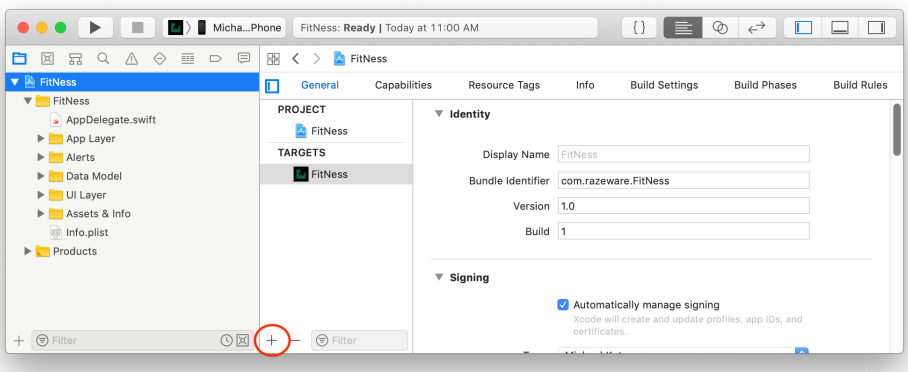
First things first: You can't run any tests without a **test target**. A test target is a binary that contains the test code, and it's executed during the test phase. It's built alongside the app, but is not included in the app bundle.

This means your test code can contain code that doesn't ship to your users. Just because your users don't see this code isn't an excuse to write lower-quality code.

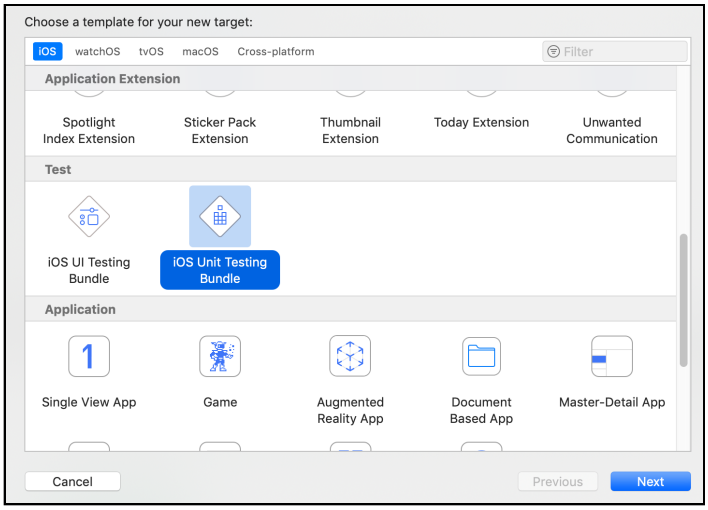
The TDD philosophy treats tests as first-class code, meaning they should fit the same standards as your production code in terms of readability, naming, error handling and coding conventions.

Adding a test target

First, create a test target. Select the **FitNess** project in the Project navigator to show the the project editor. Click the + button at the bottom of the targets list to add a new target.

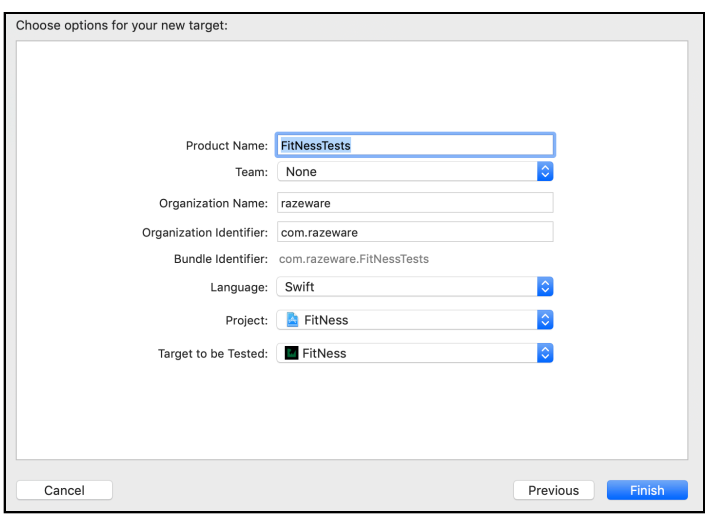


Scroll down to the **Test** section and select **iOS Unit Testing Bundle**. Click **Next**.



Did you notice the other bundle — **iOS UI Testing Bundle**? This is another type of testing. It uses automation scripting to verify views and app state. This type of testing is not necessary for adherence to TDD methodology, and is outside the scope of this book.

On the next screen, double check the **Product Name** is `FitNessTests` and the **Target to be Tested** is `FitNess`. Then click **Finish**.



Voila! You now have a **FitNessTests** target. Xcode will have also added a **FitNessTests** group in the Project navigator with a **FitNessTest.swift** file and an **Info.plist** for the target.

Figuring out what to test

The unit test target template comes with a unit test class: **FitNessTests**. Uselessly, it doesn't actually test anything. Delete the **FitNessTests.swift** file.

Right now, the app does nothing since there is no business logic. There's only one button and users expect tapping **Start** will start the activity. Therefore, you should start with… **Start**.

The TDD process requires writing a test first. This means you have to determine the smallest unit of functionality. This unit is where to start — the smallest thing that does something.

The **App Model** directory contains an `AppState` enum, which, not surprisingly, represents the different states the app can be in. The `AppModel` class holds the knowledge of which state the app is currently in.

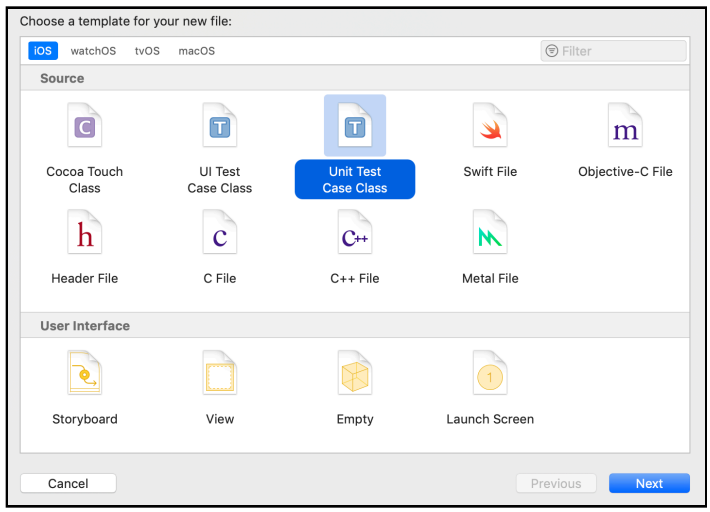
The minimum functionality to start the app is to have the **Start** button put the app into a started, or in-progress, state. There are two statements that can be made to support this goal:

1. The app should start off in the `.notStarted` state. This will allow the UI to render the welcome messaging.
2. When the user taps the **Start** button, the app should move into the `.inProgress` state so the app can start tracking user activity and display updates.

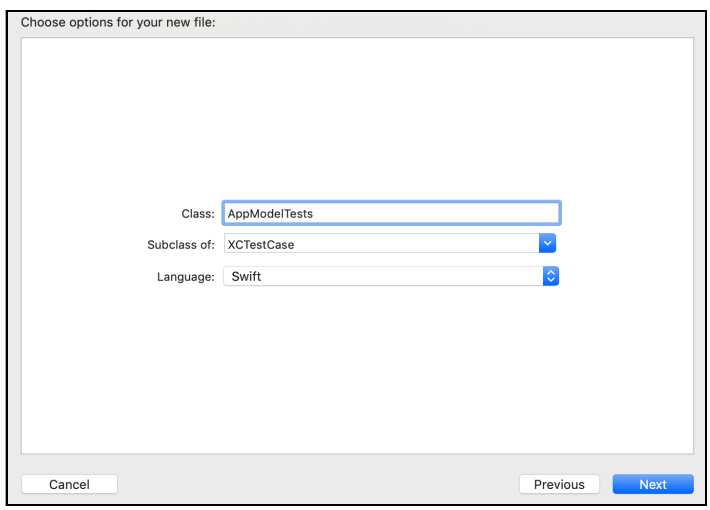
The statements are actually assertions and what you'll use to define test cases.

Adding a test class

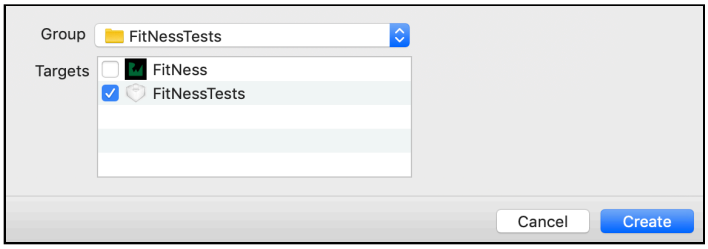
Right-click on **FitNessTests** in the project navigator. Select **New File**. In the **iOS** tab, select **Unit Test Case Class** and click **Next**.



Name the class **AppModelTests**. A good naming convention takes the name of the file or class you're testing and appends the suffix: **Tests**. In this case, you're writing tests for `AppModel`. Click **Next**.



Make sure the group is **FitNessTests** and only the eponymous target is checked. Click **Create**. If Xcode asks to create an Objective-C bridging header, click **Don't Create** — there's no Objective-C in this project.



You now have a fresh test class to start adding test cases. Delete the template methods `testExample()` and `testPerformanceExample()`, and ignore `setUp()` and `tearDown()` for now.

Red-Green-Refactor

The name of the game in TDD is **red, green, refactor**. This means iteratively writing tests in this fashion:

1. Write a test that fails (red).
2. Write the minimum amount of code so the test passes (green).
3. Clean up test(s) and code as needed (refactor).
4. Repeat the process until all the logic cases are covered.

Writing a red test

Add your first failing-to-compile test to the class:

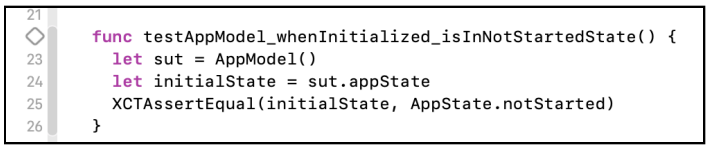
```
func testAppModel_whenInitialized_isInNotStartedState() {
  let sut = AppModel()
  let initialState = sut.appState
  XCTAssertEqual(initialState, AppState.notStarted)
}
```

This method creates an app model and gets its `appState`. The third line of the test actually performs the assertion that the state matches the expected value. More on that in a little bit.

Next, run the test.

Xcode provides several way of running a test:

- You can click the **diamond** next to an individual test in the line number bar. This runs just that test.



- You can click the **diamond** next to the class definition. This runs all the tests in the file.
- You can click the **Play** button at the right of a test or test class in the **Test navigator**. This will run an individual test, a whole test file, or all the tests in a test target.
- You can use the **Product ▸ Test** menu action (**Command + U**). This runs all the tests in the scheme. Right now, there is one test target, so it would just run all the tests in `FitNessTests`.
- You can press **Control + Option + Commanda + U**. This will run the test function if the editor cursor is within a test function, or the whole test file if the cursor is in a test file but outside a specific test function.

That's a lot of ways to run a test! Choose whichever one you prefer to run your one test.

Before the test executes, you should receive two compilation error, which means this is a failing test! Congratulations! A failing test is the first step of TDD! Remember that red is not just good, but *necessary* at this stage. If the test were to pass without any code written, then it's not a worthwhile test.

Making the test green

The first issue with this test is the test code doesn't know what the heck an `AppModel` is. Add this statement to the top of the file:

```
import FitNess
```

In Xcode, although application targets aren't frameworks, they are modules, and test targets have the ability to import them as if it were a framework. Like frameworks, they have to be imported in each Swift file, so the compiler is aware of what the app contains.

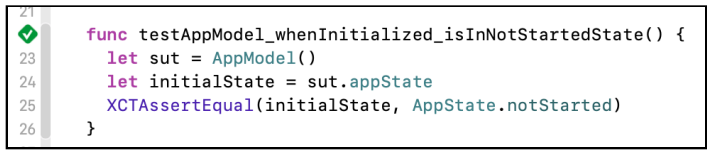
If the compile error `unresolved identifier 'AppModel'` doesn't resolve itself, you can make Xcode rebuild the test target via the **Product ▸ Build For ▸ Testing** menu, or the default keyboard shortcut **Shift + Command + U**.

You're not done fixing compiler errors yet. Now, it should be complaining about `Value of type 'AppModel' has no member 'appState'`.

Go to **AppModel.swift** and add this variable to the class directly above `init()`:

```
public var appState: AppState = .notStarted
```

Run the test again. You'll get a green check mark next to the test since it passes. Notice how the only application code you wrote was to make that one pass.



Congrats, you now have a green test! This is a trivial test: You're testing the default state of an enum variable as the result of an initializer. That means in this case there's nothing to refactor. You're done.

Writing a more interesting test

The previous test asserted the app starts in a not started state. Next, assert the application can go from **not started** to **in-progress**.

Add the following test to the end of your class before the closing bracket:

```
func testAppModel_whenStarted_isInInProgressState() {
  // 1 given app in not started
  let sut = AppModel()

  // 2 when started
  sut.start()

  // 3 then it is in inProgress
  let observedState = sut.appState
  XCTAssertEqual(observedState, AppState.inProgress)
}
```

This test is broken into three parts:

1. The first line creates an `AppModel`. The previous test ensures the model initializes to `.notStarted`.

2. The second line calls a yet-to-be created start method.
3. The last two lines verify the state should then be equal to `.inProgress`.

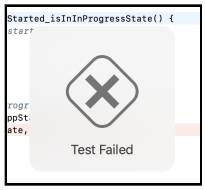
Run the tests. Once again, you have a red test that doesn't compile. Next step is to fix the compiler errors.

Open **AppModel.swift** and add the following method below `init()`:

```
public func start() {

}
```

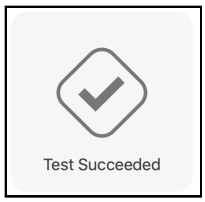
Now, the app should compile. Run the tests.



The test fails! This is obvious since `start()` has no code. Add the minimum code to this method so the test passes:

```
appState = .inProgress
```

Run the tests again, and the test passes!



Note: It's straightforward that an empty `start()` fails the test. TDD is about discipline, and it's good practice to strictly follow the process while learning. With more experience, it's OK to skip the literal build and test step after getting the test to compile. Writing the minimum amount of code so the test passes cannot be skipped, though. It's essential to the TDD process and is what ensures adequate coverage.

Test nomenclature

Some TDD nomenclature and naming best practices were followed for these tests. Take a look again at the second test, line-by-line:

1. `func testAppModel_whenStarted_isInInProgressState() {`

The test function name should describe the test. The test name shows up in the test navigator and in test logs. With a large test suite that runs in a continuous integration rig, you'll be able to just look at the test failures and know what the problem is. Avoid creating tests named `test1`, `test2`, etc.

The naming scheme used here has up to four parts:

1. All tests must begin with `test`.
2. `AppModel` This says an `AppModel` is the system under test (sut).
3. `whenStarted` is the condition or state change that is the catalyst for the test.
4. `isInInProgressState` is the assertion about what the sut's state should be after the `when` happens.

This naming convention also helps keep the test code focused to a specific condition. Any code that doesn't flow naturally from the test name belongs in another test.

2. `let sut = AppModel()`

This makes the **system under test** explicit by naming it `sut`. This test is in the `AppModelTests` test case subclass and this is a test on `AppModel`. It may be slightly redundant, but it's nice and explicit.

3. `sut.start()`

This is the behavior to test. In this case, the test is covering what happens when `start()` is called.

4. `let observedState = sut.appState`

   Define a property that holds the value you observed while executing the application code.

5. `XCTAssertEqual(observedSate, AppState.inProgress)`

The last part is the assertion about what happened to `sut` when it was started. The stated logical assertions correspond directly in `XCTest` to `XCTAssert` functions.

This division of a test method is referred to as **given/when/then**:

- The first part for a test is the things that are **given**. That is the initial state of system.
- The second part is the **when**, which is the action, event, or state change that acts on the system.
- The third part, or **then**, is testing the expected state after the when.

TDD is a process, not a naming convention. This book uses the convention outlined here, but you can still follow TDD on your own using whatever naming you'd like. What's important is your write failing tests, add the code that makes the test pass, and refactor and repeat until the application is complete.

Structure of XCTestCase subclass

XCTest is in the family of test frameworks dervied from XUnit. Like so many good object-oriented things, XUnit comes from Smalltalk (where it was SUnit). It's an architecture for running unit tests. The "X" is a stand-in for the programming language. For example, in Java it's JUnit, and in Objective-C it's OCUnit. In Swift, it's just XCTest.

With XUnit, tests are methods whose name starts with `test` that are part of a **test case class**. Test cases are grouped together into a test suite. Test runner is a program that knows how to find test cases in the suite, run them, and gather and display results. It's Xcode's test runner that is executed when you run the test phase of a scheme.

Each test case class has a `setUp()` and `tearDown()` method that is used to set up global and class state before and after each test method is run. Unlike other XUnit implementations, XCTest does not have lifecycle methods that run just once for a whole test class or the test target.

These methods are important because there a few subtle but extremely important gotchas:

- `XCTestCase` subclass lifecycles are managed outside the test execution, and any class-level state is persisted between test methods.
- The order in which test classes and test methods are run is not explicitly defined and cannot be relied upon.

Therefore, it's important to use `setUp()` and `tearDown()` to clean up and make sure state is in a known position before each test.

Setting up a test

Both tests need an `AppModel()` to test. It's common for test cases to use a common `sut` object.

In **AppModelTests.swift** add the following variable to the top of the class:

```
var sut: AppModel!
```

This sets aside storage for an `AppModel` to use in the tests. It's force-unwrapped in this case because you do not have access to the class initializer. Instead, you have to set up variables at a later time; i.e., in the `setUp()` method.

Next, add the following to `setUp()`:

```
super.setUp()
sut = AppModel()
```

Finally, remove the following:

```
let sut = AppModel()
```

In both `testAppModel_whenInitialized_isInNotStartedState()` and `testAppModel_whenStarted_isInInProgressState()`.

Build and test. The tests should both still pass.

The second test modifies the `appState` of `sut`. Without the set up code, the test ordering could matter, because the first test asserts the initial state of `sut`. But now ordering does not matter, since `sut` is re-instantiated each test.

Tearing down a test

A related gotcha with XCTestCases is it won't be deinitialized until all the tests are complete. That means it's important to clean up a test's state after it's run to control memory usage, clean up the filesystem, or otherwise put things back the way it was found.

Add the following to `tearDown()`:

```
sut = nil
super.tearDown()
```

So far it's a pretty simple test case, and the only persistent state is in `sut`, so clearing it in tearDown is good practice. It helps ensure that new global behavior added in the future won't affect previous tests.

Your next set of tests

You've now added a little bit of application logic. But there is not yet any user-visible functionality. You need to wire up the **Start** button so that it changes app state and it's reflected to the user.

Hold up! This is test-driven development, and that means writing the test first.

Since `StepCountController` contains the logic for the main screen, create a new **Unit Test Case Class** named `StepCountControllerTests` in the **FitNessTests** target.

Test target organization

Take a moment to think about the test target organization. As you continue to add test cases when building out the app, they will become hard to find and maintain in one unorganized list. Unit tests are first class code and should have the same level of scrutiny as app code. That also means keeping them organized.

In this book, you'll use the following organization:

```
Test Target
  | Cases
```

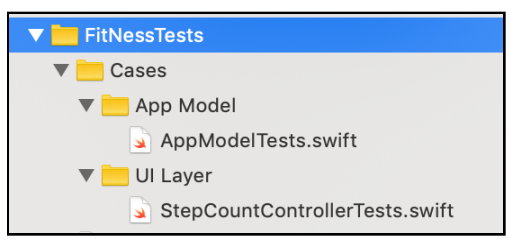
```
      | Group 1
         | Tests 1
         | Tests 2
      | Group 2
         | Tests
   | Mocks
   | Helper Classes
   | Helper Extensions
```

- **Cases**: The group for the test cases, and these are organized in a parallel structure to the app code. This makes it really easy to navigate between the app class and its tests.
- **Mocks**: For code that stands in for functional code, allowing for separating functionality from implementation. For example, network requests are commonly mocked. You'll build these in later chapters.
- **Helper classes and extensions**: For additional code that you'll write to make the test code easier to write, but don't directly test or mock functionality.

Take the two classes already in the target and group them together in a group named **Cases**.

Next, put **AppModelTests.swift** in a `App Model` group. Then put **StepCountControllerTests.swift** in a `UI Layer` group.

When it's all done, your target structure should look like this:



As you add new tests, keep them organized in groups.

Using @testable import

Open **StepCountControllerTests.swift**.

Delete the `testExample()` and `testPerformanceExample()` stubs and delete the comments in `setUp()` and `tearDown()`.

Next, add the following class variable above `setUp()`:

```
var sut: StepCountController!
```

If you build the test class now, you'll see the following error: `use of undeclared type 'StepCountController'`. This is because the class is specified as **internal** because it doesn't explicitly define access control.

There are two ways to fix this error. The first is to declare `StepCountController` as **public**. This will make that class available outside the `FitNess` module and usable by the test class. However, this would violate SOLID principles by making the view controller visible outside of the app.

Fortunately, Xcode provides a way to expose data types for testing without making them available for general use. That's through the `@testable` attribute.

Add the following to the top of the file, under `import XCTest`:

```
@testable import FitNess
```

This makes symbols that are `open`, `public`, *and* `internal` available to the test case. Note that this attribute is only available in test targets, and will not work in application or framework code. Now, the test can successfully build.

Next, update `setUp()` and `tearDown()` as follows:

```
override func setUp() {
  super.setUp()
  sut = StepCountController()
}

override func tearDown() {
  sut = nil
  super.tearDown()
}
```

Testing a state change

Now comes the fun part. There are two things to check when the user taps **Start**: First is that the app state updates, and the second is that the UI updates. Take each one in turn.

Add the following test method below `tearDown()`:

```
func testController_whenStartTapped_appIsInProgress() {
  // when
```

```
    sut.startStopPause(nil)

    // then
    let state = AppModel.instance.appState
    XCTAssertEqual(state, AppState.inProgress)
  }
```

This tests that when the `startStopPause(_:)` action is called, the app state will be `inProgress`.

Build and test, and you'll get a test failure. This is because `startStopPause` is not implemented yet. Remember, test failures at this point are good! Open **StepCountController.swift**, and add the following code to `startStopPause(_:)`:

```
  AppModel.instance.start()
```

Build and test again. Now the test passes!

Testing UI updates

UI testing with UI Automation is a whole separate kind of testing and not covered in this book. However, there plenty of UI aspects that can, and should, be unit tested.

Add the following test case at the bottom of `StepCountControllerTests`:

```
  func testController_whenStartTapped_buttonLabelIsPause() {
    // when
    sut.startStopPause(nil)

    // then
    let text = sut.startButton.title(for: .normal)
    XCTAssertEqual(text, AppState.inProgress.nextStateButtonLabel)
  }
```

Like the previous tests, this performs the `startStopPause(_:)` action, but this time the test checks that the button text updates.

You may have noticed that this test is almost exactly the same as the previous one. It has the same initial conditions and "when" action. The important difference is that this tests a different state change.

TDD best practice is to have one assert per test. With well-named test methods, when the test fails, you'll know exactly where the issue is, because there is no ambiguity between multiple conditions. You'll tackle cleaning up this kind of redundancy in later chapters.

Another good practice illustrated here is the use of `AppState.inProgress.nextStateButtonLabel` instead of hard-coding the string. By using the app's value, the assert is testing behavior and not a specific value. If the string changes or gets localized, the test won't have to change to accommodate that.

Since this is TDD, the test will fail if you run it. Fix the test by adding the appropriate code to the end of `startStopPause(_:)`:

```
let title = AppModel.instance.appState.nextStateButtonLabel
startButton.setTitle(title, for: .normal)
```

Now, build and test again for a green test. You can also build and run to try out the functionality.

Tapping the **Start** button turns it into a **Pause** button.

As you can see from the lack of any other functionality, the app still has a way to go.

Testing initial conditions

The last two tests rely on certain initial conditions for its state. For example in `testController_whenStartTapped_buttonLabelIsPause`, the desire is to test for the transition from `.notStarted` to `.inProgress`. But the test could also pass if the view controller started out already in `.inProgress`.

Part of writing comprehensive unit tests is to make implicit assumptions into explicit assertions. Insert the following code between `tearDown()` and `testController_whenStartTapped_appIsInProgress()`:

```
// MARK: - Initial State

func testController_whenCreated_buttonLabelIsStart() {
  let text = sut.startButton.title(for: .normal)
  XCTAssertEqual(text, AppState.notStarted.nextStateButtonLabel)
}

// MARK: - In Progress
```

This test checks the button's label after it's created to make sure it reflects the `.notStarted` state.

This also adds some `MARK`s to the file to help divide the test case up into sections. As the classes get more complicated, the test files will grow quite large, so it's important to keep them well organized.

Build and test. Hurray, another failure! Go ahead and fix the test.

Open **StepCountController.swift** and add the following at the end of `viewDidLoad()`:

```
let title = AppState.notStarted.nextStateButtonLabel
startButton.setTitle(title, for: .normal)
```

The test is not quite ready yet to pass. Go back to the tests, and add at the top of `testController_whenCreated_buttonLabelIsStart()` the following lines:

```
// given
sut.viewDidLoad()
```

Now, build and test and the tests will pass. The call to `viewDidLoad()` is needed because the `sut` is not actually loaded from the `xib` and put into a view hierarchy, so the view lifecycle methods do not get called. You'll see in Chapter 4, "Test Expressions," how to get a properly loaded view controller for testing.

Refactoring

If you look at **StepCountController.swift**, the code that sets the button text is awfully redundant. When building an app using TDD, after you get all the tests to pass, you can then refactor the code to make it more efficient, readable,

maintainable, etc. You can feel free to modify the both the app code and test code at will, resting easy because you have a complete set of tests to catch any issues if you break it.

Add the following method to the bottom of `StepCountController`:

```
  private func updateButton() {
    let title = AppModel.instance.appState.nextStateButtonLabel
    startButton.setTitle(title, for: .normal)
  }
```

This helper method will be used in multiple places in the file — whenever the button needs to reflect a change in app state. This can be `private` as this is an internal implementation detail of the class. The behavioral methods remain `internal` and can still be tested.

In `viewDidLoad()` and `startStopPause(_:)` replace the two lines that update the title with a call to `updateButton()`.

Build and test. The tests will all still pass. Code was changed, but behavior was kept constant. Hooray refactoring! This type of refactoring is called **Extract Method**. There is a menu item to do it available in the **Editor ▸ Refactor** menu in Xcode.

You're still a long way from a complete app with a full test suite, but you are on your way.

Challenge

There are a few things still to do with the two test classes made already. For example, `AppModel` is **public** when it should really be **internal**. Update its access modifier and use `@testable import` in `AppModelTests`.

And in **StepCountControllerTests.swift** there is a redundancy in the call to `startStopPause(_:)`. Extract that out into a helper **when** method.

Key points

- TDD is about writing tests *before* writing app logic.
- Use logical statements to drive what should be tested.
- Each test should fail upon its first execution. Not compiling counts as a failure.
- Use tests to guide refactoring code for readability and performance.
- Good naming conventions make it easier to navigate and find issues.

Where to go from here?

Test-driven development is pretty simple in its fundamentals: Only write app code in order for a unit test to pass. For the rest of the book, you'll be over and over again following the red-green-refactor model. You'll explore more interesting types of tests, and learn how to test things that aren't obviously unit testable.

For more information specific to how Xcode works with tests and test targets see the developer documentation. For a jam-packed overview on iOS testing try out this free tutorial.

In the next chapter, you'll learn more about `XCTAssert` functions, testing view controllers, code coverage and debugging unit tests.

Chapter 4: Test Expressions

By Michael Katz

The TDD process is straightforward, but writing good tests may not always be. Fortunately, each year, Xcode and Swift have become more capable. This means you have many features at your disposal that help with both writing and running tests.

This chapter covers how to use the `XCTAssert` functions. These are the primary actors of the test infrastructure. Next, you'll learn how to use the **host application** to drive view controller unit testing. Then, you'll go through gathering code coverage to verify the minimum amount of testing. Finally, you'll use the test debugger to find and fix test errors.

In this chapter, you'll learn about:

- XCTAssert functions
- UIViewController testing
- Code Coverage
- Test debugging

> **Note**: Be sure to use the Chapter 4 starter project rather than continuing with the Chapter 3 final project. It has a few new things added to it, including placeholders for the code to add in this tutorial.

Assert methods

In Chapter 3, "Driving TDD," you used `XCTAssertEqual` exclusively. There are several other assert functions in XCTest:

- Equality: `XCTAssertEqual`, `XCTAssertNotEqual`
- Truthiness: `XCTAssertTrue`, `XCTAssertFalse`
- Nullability: `XCTAssertNil`, `XCTAssertNotNil`
- Comparison: `XCTAssertLessThan`, `XCTAssertGreaterThan`, `XCTAssertLessThanOrEqual`, `XCTAssertGreaterThanOrEqual`
- Erroring: `XCTAssertThrowsError`, `XCTAssertNoThrow`

Ultimately, any test case can be boiled down to a conditional: (does it meet an expectation or not) so any test assert can be re-composed into a `XCTAssertTrue`.

> **Note**: With XCTest, a test is marked as passed as long as there are no failures. This means that it does not require a positive `XCTAssert` assertion. A test with no asserts will be marked as success, even though it does not test anything!

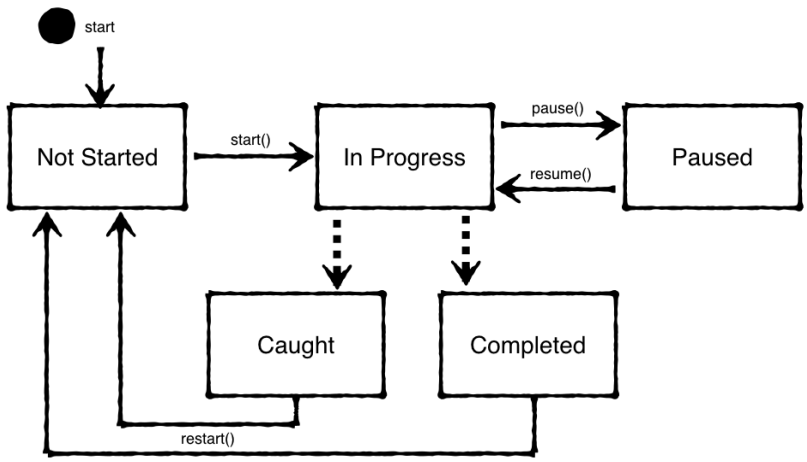
App state

In the previous chapter, you built out the functionality to move the app from a not started state to an in-progress one. Now is a good time to think about about the whole app lifecycle.

Here are the possible app states, as represented by the `AppState` enum:

- `notStarted`: The initial state of the app.
- `inProgress`: The app is actively monitoring the activity of the user and Nessie.
- `paused`: The app was paused by the user. Nessie is put to sleep and the activity tracking stops.
- `completed`: The user has reached their activity goal before Nessie caught up.
- `caught`: Nessie caught up to the user and "ate" them.

The following diagram shows the possible state transitions:



The solid lines represent user action on the UI, and the dotted lines happen automatically due to time or activity events. The user-based transitions will be covered in this chapter project, and the automatic transitions will be covered in Chapter 5: "Test Expectations."

Asserting true and false

To build out the state transitions, you need to add some more information to the app about the user. The `completed` and `caught` states depend on the user activity, the set goal and Nessie's activity. To keep the architecture clean, the app state information will be kept separate from the raw data that is tracking the user.

Add a new unit test case class to the test target, in the **Data Model** group. Name it **DataModelTests**. Once again, and like always, remove `testExample()` and `testPerformanceExample()`.

Add the import to the top of the file:

```
@testable import FitNess
```

Next, add this class variable:

```
var sut: DataModel!
```

Now, you have a red test case class. To fix it, open **DataModel.swift** and add this code, the minimum to get the test to compile:

```
class DataModel {

}
```

This creates a stub class to fix the compiler error. You'll build upon this piece-by-piece.

Next, open **DataModelTests.swift** and replace `setUp()` and `tearDown()` with the following:

```
override func setUp() {
  super.setUp()
  sut = DataModel()
}

override func tearDown() {
  sut = nil
  super.tearDown()
}
```

These create a new `DataModel` for each test, and then cleans it up afterwards.

Add the following code to the end of `DataModelTests`:

```
// MARK: - Goal
func testModel_whenStarted_goalIsNotReached() {
  XCTAssertFalse(sut.goalReached,
      "goalReached should be false when the model is created")
}
```

This test introduces `XCTAssertFalse`, which checks that the expected value is `false`. Each `XCTAssert` function can also take an optional `String` message. This message is displayed in the standard editor and report navigator's error log when the test fails. If you follow the test naming convention and only use one `XCTAssert` per test, then you won't normally need to supply an error message. While test name will usually be descriptive enough to inform you why a failure occurred, it can be useful to add a message if the assertion isn't obvious.

Fix the non-compiling test by adding the following to `DataModel` in **DataModel.swift**:

```
var goalReached: Bool { return false }
```

Build and test, and the test will pass.

The initial state is the boring state. Next build out the business logic. First, open **DataModelTests.swift** and add the following test method below `tearDown()`:

```
func testModel_whenStepsReachGoal_goalIsReached() {
  // given
  sut.goal = 1000

  // when
  sut.steps = 1000

  // then
  XCTAssertTrue(sut.goalReached)
}
```

This tests the logic "the goal is reached when the number of steps equals or exceeds the goal."

Now, you need a goal and steps for it to compile. Open **DataModel.swift** and add the following below `goalReached`:

```
var goal: Int?
var steps: Int = 0
```

`goal` is an optional because it should be set explicitly by the user.

Now, the test will build, but fail.

Next, replace `goalReached` with the following:

```
var goalReached: Bool {
  if let goal = goal,
    steps >= goal {
      return true
  }
  return false
}
```

Run the test again. It's a little tricky on the fingers, but you can use **Product ▸ Perform Action ▸ Test Again** (^⌥⌘G) to re-run the last test from anywhere in Xcode. Now, the test passes, and you've seen `true` and `false` asserts.

Pretty much every assert is just a Boolean test and can be rewritten as such. That means you can write your own helper methods that look like `XCTAssert`'s. These just have to eventually evaluate to a Boolean that is passed to `XCTAssertTrue()`.

Testing Errors

If the optional `goal` property isn't set, it doesn't make sense for the app to enter the `inProgress` state. Therefore starting the app without a goal is an error!

Make it a real **error**. Open **AppModel.swift**, then add the `throws` keyword to the function signature of `start()`:

```
func start() throws {
```

Now, fix the compilation errors. In **StepCountController.swift** replace `startStopPause(_:)` with the following:

```
@IBAction func startStopPause(_ sender: Any?) {
  do {
    try AppModel.instance.start()
  } catch {
    showNeedGoalAlert()
  }

  updateUI()
}
```

Once you're done, tapping the **Start** button without setting a goal will display an alert. Don't worry about writing a test first for this right now.

Next, update `testAppModel_whenStarted_isInInProgressState()` in **AppModelTests.swift**. Add a `try?` to the `sut.start()` line to quiet the error. This test should still pass. You'll come back here after changing the logic in a bit.

Next, add the following test before `testAppModel_whenStarted_isInInProgressState()`:

```
func testModelWithNoGoal_whenStarted_throwsError() {
  XCTAssertThrowsError(try sut.start())
}
```

Using `XCTAssertThrowsError`, you can verify that an error is thrown if the model is started in its initial state without a goal set.

This test fails since there is no error thrown yet. To fix that, open **AppModel.swift** and add the following instance variable:

```
let dataModel = DataModel()
```

The app model will be the container for the data model, since the app's data is a

subset of the app's state. The data model's goal is needed to check for an error.

Add this `guard` statement at the top of `start()`:

```
guard dataModel.goal != nil else {
  throw AppError.goalNotSet
}
```

Now, build and test `testModelWithNoGoal_whenStarted_throwsError`, and the test will pass.

Next, verify that setting a goal means that `start()` will not throw an error. Open **AppModelTests.swift** and add the following under `// MARK: - Given`:

```
func givenGoalSet() {
  sut.dataModel.goal = 1000
}
```

Next, add the following test under `testModelWithNoGoal_whenStarted_throwsError()`:

```
func testStart_withGoalSet_doesNotThrow() {
  // given
  givenGoalSet()

  // then
  XCTAssertNoThrow(try sut.start())
}
```

This test should go right to green, since the app logic was already written. Even though no code had to be added or changed for this test, it's still TDD since the tests are leading the way. This test just completes checking all the cases of the logical flow.

Finally, it's time to fix all the other tests that started failing due to this change.

First, add the following to the top of `testAppModel_whenStarted_isInInProgressState`:

```
// given
givenGoalSet()
```

Next, open **StepCountControllerTests.swift** and add the following under `// MARK: - Given`:

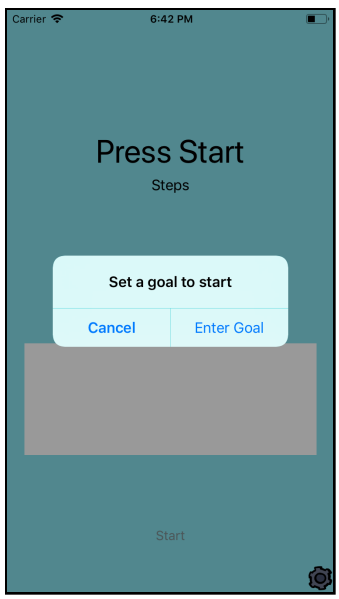
```
func givenGoalSet() {
  AppModel.instance.dataModel.goal = 1000
}
```

Finally, in the two tests under `// MARK: - In Progress`, add the following to the top of each:

```
  // given
  givenGoalSet()
```

Build and run all the tests. They all pass! Changing these existing tests to pass again after changing the app logic is another aspect of the **refactor** phase of the TDD cycle.

If you build and run the app, there will now be an alert when **Start** is tapped and the app won't move into the `inProgress` state. In the next section you will update the app with the ability to save the goal.



View controller testing

Now that the model can have a goal set and the app state checks it, the next feature is to expose that to the user. In the previous chapter, you wrote some unit tests for `StepCountController`. Now build on that with some proper view controller unit testing.

Functional view controller testing

The important thing when testing view controllers is to not test the views and controls directly. This is better done using UI automation tests. Here, the goal is to check the logic and state of the view controller.

Functional testing is done by using separate methods for interacting with the UI (callbacks, delegate methods, etc.) from logic methods (updating state).

> **Note**: If you have experience with other app architectures, using something like MVVM or VIPER makes it cleaner to test this type of logic. Separating a ViewModel from the controller takes the unit-testable logic out of the controller. For the purposes of this section, you'll continue to build the app using the traditional Apple MVC model. This is what's covered in most of the documentation and the traditional place to start developing iOS applications.

First, open **StepCountControllerTests.swift**. Next, add the following test under `// MARK - Goal`:

```
func testDataModel_whenGoalUpdate_updatesToNewGoal() {
  // when
  sut.updateGoal(newGoal: 50)

  // then
  XCTAssertEqual(AppModel.instance.dataModel.goal, 50)
}
```

This test calls `updateGoal(newGoal:)` and verifies the data model has been properly updated.

Be sure to also restore the state by adding the following line to `tearDown()` above `super.tearDown()`:

```
AppModel.instance.dataModel.goal = nil
```

As expected, the test will fail. Let's turn the test green. Open **StepCountController.swift** and replace `updateGoal(newGoal:)` with the following:

```
func updateGoal(newGoal: Int) {
  AppModel.instance.dataModel.goal = newGoal
}
```

Another beautiful green test.

Using the host app

The next requirement for the app is that the central view should show the user's avatar in the running position. The word *should* signifies an assertion, so you'll write one, now. First, open **StepCountControllerTests.swift**. Next, add the following under `// MARK: - Chase View`:

```
func testChaseView_whenLoaded_isNotStarted() {
```

```
    // when loaded, then
    let chaseView = sut.chaseView
    XCTAssertEqual(chaseView?.state, AppState.notStarted)
  }
```

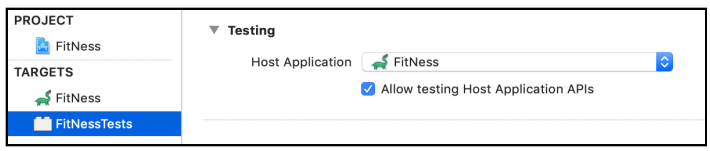
The test builds, but does not pass, because `chaseView` is `nil`. What gives?

Well, there is a cheat in the code to allow the existing tests to pass. Under normal app flow, a `StepCountController` is created and populated by the storyboard. It's already loaded by the time any app code gets to execute.

In this test the `sut` is initialized directly, which means its starting state is not the same as when the app runs. Fortunately, there is a clean way to handle this.

When unit tests are run as part of the **Test** action in an app scheme, Xcode uses a **Host Application** as specified in the target settings.

Open the **General** tab of the Project editor for the **FitNessTests** target. You'll see that **FitNess** is selected as the Host Application.



This means that running the test action, will launch the host app on the specified destination (simulator or device). The test runner waits for the app to load before starting the tests, and the tests are run in the app's context.

As a consequence, you have access to the `UIApplication` object and the whole View hierarchy in the tests.

In the Project navigator, under **FitNessTests** target, add a new group: **Test Classes**. Next, create a new **Swift File**, **ViewControllers.swift**, in that group

Replace the contents of this file with the following:

```
import UIKit
@testable import FitNess

func loadRootViewController() -> RootViewController {
  let window = UIApplication.shared.windows[0]
  return window.rootViewController as! RootViewController
}
```

This function navigates the app's `window` to retrieve the root view controller, which is of type `RootViewController`. This helper function will be used to obtain other view

controllers.

Next, create another new group, **Test Extensions** under **FitNessTests**. In that group, add a new Swift file: **RootViewController+Tests.swift**.

Replace the contents of this file with the following `RootViewController` extension:

```
import UIKit
@testable import FitNess

extension RootViewController {
  var stepController: StepCountController {
    return children.first { $0 is StepCountController }
      as! StepCountController
  }
}
```

Now, you have all the pieces to get the `StepCountController` from the host app.

Fixing the tests

Go back to **StepCountControllerTests.swift**, and replace `setUp()` with the following:

```
override func setUp() {
  super.setUp()
  let rootController = loadRootViewController()
  sut = rootController.stepController
}
```

Remove the call to `viewDidLoad` from `testController_whenCreated_buttonLabelIsStart()`, as this is no longer needed.

Next, add this method under `// MARK: - Given`:

```
func givenInProgress() {
  givenGoalSet()
  sut.startStopPause(nil)
}
```

This sets the app into the `inProgressState`. It's ensured by the test `testController_whenStartTapped_appIsInProgress()`.

Finally, add the following test to the bottom of `StepCountControllerTests`:

```
func testChaseView_whenInProgress_viewIsInProgress() {
```

```
    // given
    givenInProgress()

    // then
    let chaseView = sut.chaseView
    XCTAssertEqual(chaseView?.state, AppState.inProgress)
  }
```

This test will fail since the `chaseView` is not yet updated. Open **StepCountController.swift** and replace `updateChaseView()` at the bottom with the following:

```
private func updateChaseView() {
  chaseView.state = AppModel.instance.appState
}
```

The test testChaseView_whenInProgress_viewIsInProgress will now pass, and no more funny business with loading view controllers.

> **Note**: One alternate way of retrieving and testing a view controller can be done as follows: First, get a reference to the storyboard:
>
> `let storyboard = UIStoryboard(name: "Main", bundle: nil)`
>
> Second, get a reference to the view controller:
>
> ```
> let stepController =
> storyboard.instantiateViewcontroller(withIdentifier:
> "stepController") as! StepCountController
> ```
>
> Finally, if needed, you may load the view as follows:
>
> `stepController.loadViewIfNeeded()`
>
> Following this pattern allows you to instantiate a fresh view controller for each test, and it affords the option to set up and tear down the view controller for each test.

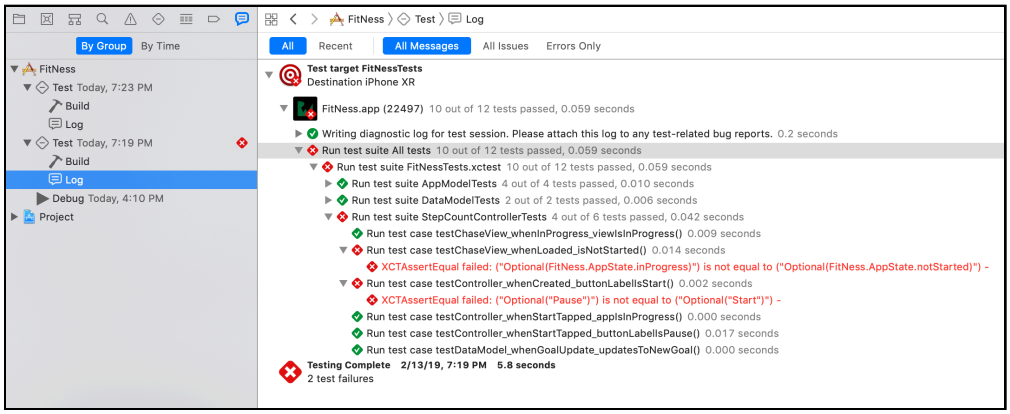
Test ordering matters

Build and test the whole target, and most of the tests should pass, but not `testController_whenCreated_buttonLabelIsStart`. This test fails.

Now, only test `testController_whenCreated_buttonLabelIsStart` and it will pass. Hrm… strange.

Open the report navigator and look at the result for when you last ran all the tests.

Look at the test failure: `XCTAssertEqual failed: ("Optional("Pause")") is not equal to ("Optional("Start")")`.



This message tells you not only that the button text is not what's expected, but specifically that the button text is "Pause." That's what the button should say when the app is `inProgress`. This violates the assumption that the test is starting with a fresh `StepCountController`.

The previous change to using the host app's `StepCountController` meant that a new controller is not created every `setUp()` and the app state is persisted. In order to have clean tests, you need to reset the state in `tearDown()`.

To help with this, you can create a new function on `AppModel` to reset the state. But, first, write the tests.

Open **AppModelTests.swift**. Add the following helper to the `Given` section:

```
func givenInProgress() {
  givenGoalSet()
  try! sut.start()
}
```

This puts the app in an `inProgress` state, allowing for the state restart test to actually test a change.

Next, add the following to the bottom of the test case class:

```
// MARK: - Restart

func testAppModel_whenReset_isInNotStartedState() {
  // given
  givenInProgress()
```

```
    // when
    sut.restart()

    // then
    XCTAssertEqual(sut.appState, .notStarted)
  }
```

This tests that the not-yet-added `restart()` puts the model back into `notStarted`. To get the test to pass open **AppModel.swift** and add the following to `AppModel`:

```
  func restart() {
    appState = .notStarted
  }
```

This function will be used as a test helper for now, but eventually will be part of the whole app's state cycle.

Finally, go back and fix the original issue. Change `tearDown()` in **StepCountControllerTests.swift** to:

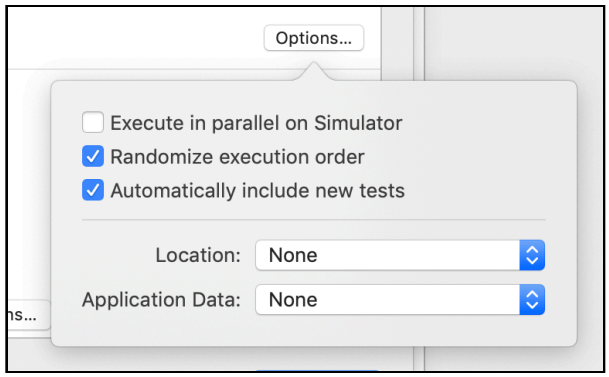
```
  override func tearDown() {
    AppModel.instance.dataModel.goal = nil
    AppModel.instance.restart()
    sut.updateUI()
    super.tearDown()
  }
```

Now, running the whole target's tests will succeed.

Randomized order

There is also an option in the **Test** action of the scheme to randomize the test order.

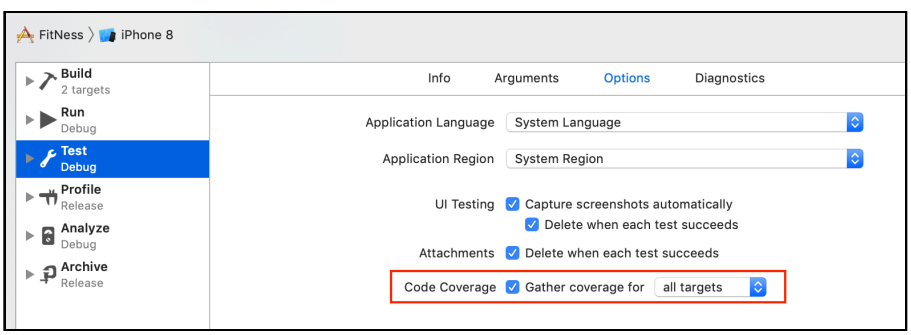
Edit the FitNess scheme. Select the **Test** action. In the center pane, next to **FitNessTests** is an **Options...** button. Click that and, in the pop-up, check **Randomize execution order**. This will cause the tests to run in a random order each time.



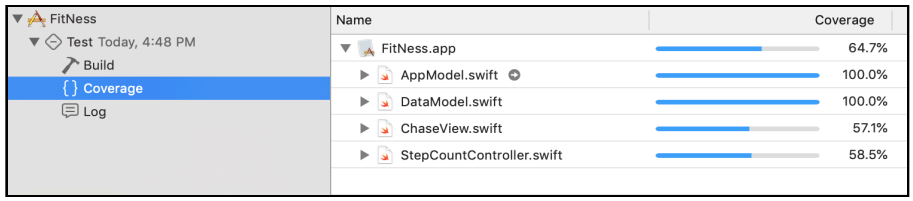
This can expose hidden inter-test dependencies that you wouldn't catch with the default ordering. The downside is that the ordering is not guaranteed, meaning you might have missed the previous issue. Also, if an ordering issue does come up, it might be hard to reproduce if it was very specific. Sporadic and hard-to-diagnose test failures are one symptom that the random ordering uncovered an issue.

Code coverage

While on the subject of the scheme editor, open up the **Test Action** again. This time select the **Options** tab. There is a checkbox for **Code Coverage**. Check it.



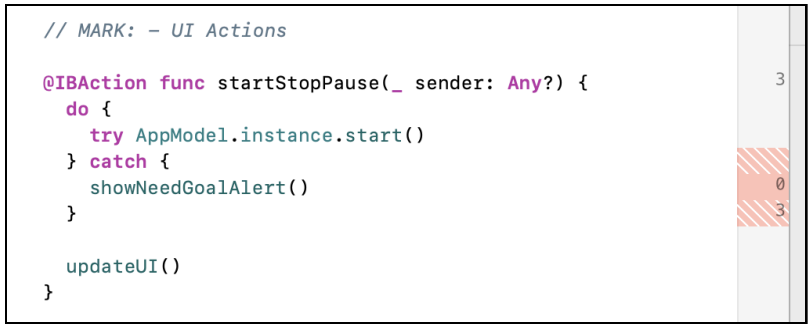
Run the tests again. After the tests succeed, open the Report navigator. Under the latest test, there will be three reports: **Build**, **Coverage** and **Log**. Select **Coverage** to display the coverage report.



Code coverage is the measure of how many lines of app code are executed during tests. There will be a list of each file in the target along with the percentage of the code lines that were executed. Having 100% or close for a file means you're following TDD closely. When the tests are written first, only the code needed to pass the test gets added.

Opening up an individual file will show the coverage on a per-function or closure basis. Double-clicking on a file or function name will open up that file in the editor.

Open **StepCountController.swift** and navigate to `startStopPause(_:)`



You'll see a coverage annotation on the right side of the editor. The number shown represents the number of times that line was executed. Lines with a red coloring or a "0" indicate opportunities to add additional tests.

Lines with a striped red annotation mean that only part of that line was run. Hovering over the stripe in the annotation bar will show you in green which part was run and in red what was not.

In `StepCountController`, it looks like the `startStopPause(_:)` method was never called when `AppModel.start()` throws an error.

The problem with testing that condition is that, when there's an error, an alert controller is shown. You could write a test that checks for that alert controller, but that is really the domain of UI automation testing. You could refactor `StepCountController` so that a variable is set or a callback is called in that error case, but then you would be modifying app code just to add a test. The test would then be testing itself and not app functionality, which does not provide any value.

The goal should be to get as close to 100% as possible. Coverage doesn't mean the code works, but lack of coverage means that it's not tested. For views and view controllers, it's not expected to get to 100% coverage because TDD does not include UI testing. When you combine unit tests with UI automation tests, then you should expect to be able to cover most if not all of these files.

Debugging tests

When it comes to debugging tests, you've already practiced the first line of defense. That is: "Am I testing the right thing?"

Make sure:

- You have the right assumptions in the **given** statements.
- Your **then** statements accurately reflect the desired behavior.

If nothing obvious in the test code appears, next check the test execution order for preserved state. Also use code coverage to make sure the right code paths are taken.

After trying that, you can use some other tools in Xcode's arsenal. To try them out, it's time to think about the other important actor in the app: Nessie.

Using test breakpoints

With Nessie in the picture, the data model gets a little more complicated. Here are the new rules with Nessie:

- When Nessie's distance is greater than or equal to the user's, Nessie wins (the user is caught). The user cannot be caught when the distance is at 0, which is the start condition.
- If the user is caught by Nessie, the goal cannot be reached.

Open **DataModelTests.swift** and add the following test to `DataModelTests`:

```
// MARK: - Nessie
func testModel_whenStarted_userIsNotCaught() {
  XCTAssertFalse(sut.caught)
}
```

This tests that with a fresh `DataModel`, the user is not caught. This test does not yet compile.

Fix the broken test by adding the following to `DataModel` in **DataModel.swift**:

```
// MARK: - Nessie

let nessie = Nessie()
var distance: Double = 0

var caught: Bool {
```

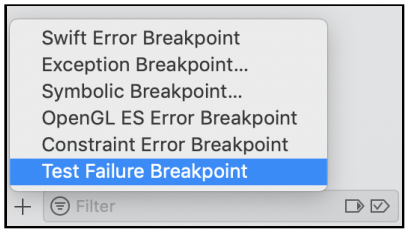
```
    return nessie.distance >= distance
  }
```

This adds a `Nessie` to the data model, a variable to track user distance, and a computed variable to compare the distances. A separate variable for `distance` is used instead of `steps` to keep the calculations cleaner later on.

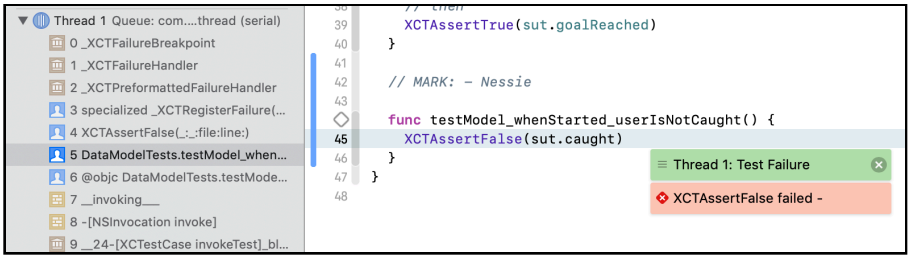
Even with the updated code, the test still fails. There are several ways to go about diagnosing the problem. As you've already seen there are a few things to check:

- The test itself is correct, the **given** is a fresh `DataModel` as created in `startUp()`. The **then** is also correct, `caught` *should* be `false`.
- The `DataModel` code was executed, as shown by the code coverage.

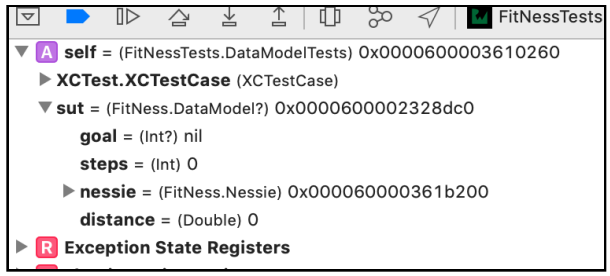
A good next step is to try out the debugger. In the **Breakpoint navigator**, click the + all the way at the bottom. Select **Test Failure Breakpoint**.



This creates a special breakpoint that halts execution when a unit test fails. Run the test again, and the debugger will stop at the test failure.



Open the variables view, and expand **self** and then **sut**.



Here, you'll see that both `distance` and `steps` are `0`. So the app logic is doing the right thing, Nessie is tied with the user, which should be the caught state. However, this is a special case in which the starting condition cannot result in a capture.

To fix this, open **DataModel.swift** and replace `caught` with the following:

```
var caught: Bool {
  return distance > 0 && nessie.distance >= distance
}
```

Now, the test will pass. This might have been an obvious example, but it illustrates that you have all your normal debugging techniques available when running tests.

Completing coverage

If you take a look at the code coverage for **DataModel.swift**, it is no longer 100%. If you look at the file, notice the striped annotation in the updated `caught`. Hovering over the stripe shows that only the `distance > 0` condition was checked. This tells you that there are more conditions to test.

Open **DataModelTests.swift** and add the following test cases to complete `DataModel` coverage:

```
func testModel_whenUserAheadOfNessie_isNotCaught() {
  // given
  sut.distance = 1000
  sut.nessie.distance = 100

  // then
  XCTAssertFalse(sut.caught)
}

func testModel_whenNessieAheadofUser_isCaught() {
  // given
```

```
    sut.nessie.distance = 1000
    sut.distance = 100

    // then
    XCTAssertTrue(sut.caught)
  }
```

Now, test and check out the `DataModel` coverage... 100%

Finishing out the requirements

There is one final piece that hasn't been accounted for yet: The user cannot reach the goal if they have been caught. Add this test to the `Goal` tests section:

```
func testGoal_whenUserCaught_cannotBeReached() {
  //given goal should be reached
  sut.goal = 1000
  sut.steps = 1000

  // when caught by nessie
  sut.distance = 100
  sut.nessie.distance = 100

  // then
  XCTAssertFalse(sut.goalReached)
}
```

Then, to make the test pass, update `goalReached` in **DataModel.swift**:

```
var goalReached: Bool {
  if let goal = goal,
    steps >= goal, !caught {
      return true
  }
  return false
}
```

Test again for success.

Challenge

In `StepCountControllerTests.tearDown()`, there are separate calls to reset the `AppModel` and the `DataModel`. Since the data model is a property of the app model, refactor the data model reset into `AppModel.restart()`, along with the appropriate tests.

For an extra challenge, use some of the other `XCTAssert` functions not yet used, like `XCTAssertNil` or `XCTAssertLessThanOrEqual`.

A second challenge is to add the pause functionality to the app so the user can move back and forth between `.paused` and `.inProgress`. The pause doesn't have to do anything else at this point, since the direct functionality will be covered in later chapters.

Key points

- Test methods require calling a `XCTAssert` function.
- View controller logic can be separated in to data/state functions, which can be unit tested and view setup and response functions, which should be tested by UI automation.
- Test execution order matters.
- The code coverage reports can be used to make sure all branches have a minimum level of testing.
- Test failure breakpoints are a tool on top of regular debugging tools for fixing tests.

Where to go from here?

For more on code coverage, this video tutorial covers that topic. And you can learn everything and more about debugging from the Advanced Apple Debugging and Reverse Engineering book. The tools and techniques taught in that tome are just as applicable to test code as application code.

In the next chapter, you'll learn about testing asynchronous functions using `XCTestExpectation`.

Where to Go From Here?

We hope you enjoyed this sample of *iOS Test-Driven Development by Tutorials*!

If you enjoyed this sample, be sure to check out the full book, which will contain the following chapters:

1. **What Is TDD?:** A quick introduction into test-driven development and the process you should follow.
2. **The TDD Cycle:** Introduces the four steps of the **TDD Cycle**, which has four individual steps that are normally called the **Red-Green-Refactor-Cycle**.
3. **TDD App Setup:** Introduces the concept of setting up your application within Xcode to run tests including setting up your test target and writing a few tests.
4. **Test Expressions:** Introduces the concept of how to use the **XCTAssert** functions which are the primary actors of the test infrastructure.
5. **Test Expectations:** Introduces the concept of the **expectation** and a **waiter** to allow you to test asynchronous code in a test-driven fashion.
6. **Dependency Injection & Mocks:** Introduces the concept of testing code which depends on system or external services without needing to call services — the services may not be available, usable or reliable. These techniques allow you to test error conditions, like a failed save, and to isolate logic from SDKs.

7. **Introducing DogPatch:** A quick introduction to the **DogPatch** application you'll use for writing networking code in a test-driven fashion.
8. **RESTful Networking:** You'll learn how to TDD a RESTful networking client, specifically setting up the networking client, ensure the correct endpoint is called, handle networking errors, valid responses and invalid responses and finally dispatch the results to a response queue.
9. **Using the Network Client:** You'll use the previously created network client in your application by updating the **ListingsViewController** to use the DogPatchClient to actually network.
10. **Image Client:** You'll use TDD to create an ImageClient for handling images. You can use that ImageClient anywhere you need it in the app.
11. **Legacy Problems:** Beginning TDD on a "legacy" project is much different than starting TDD on a new project. For example, the project may have few (if any) unit tests, lack documentation and be slow to build. This chapter will introduce you to strategies for tackling these problems.
12. **Dependency Maps:** Before you can make a change, you must first understand how the system works and which classes relate to one another. This chapter will give you a tool for doing this: dependency maps.
13. **Breaking Up Dependencies:** In this chapter, you'll use the strategies and techniques you learned from the previous chapters to start TDDing changes to the legacy app. To make this possible, however, you'll have to break existing class dependencies. You'll learn how to do this in a safe(r) manner in this chapter.
14. **Modularizing Dependencies:** You'll continue the work from the last chapter, further breaking **MyBiz** into modules so you can reuse the login functionality. You'll learn how to define clean boundaries in the code to create logical units. Through the use of tests, you'll make sure the new architecture works and the app continues to function.
15. **Adding Features to Existing Classes:** You won't always have the time, or it may simply not be possible, to break the dependencies of very large classes. In this chapter, you'll learn strategies to add functionality to existing class, while at the same time, avoid modifying them!

We hope you enjoy the book!

-- Joshua, Michael and the *iOS Test-Driven Development by Tutorials* team

Learn how to test iOS Applications!

iOS Test-Driven Development introduces you to a broad range of concepts with regard to not only writing an application from scratch with testing in mind, but also applying these concepts to already written applications which have little or no tests written for their functionality.

Who This Book Is For

This book is for intermediate iOS developers who already know the basics of iOS and Swift development but want to learn how to write code which is both testable and maintainable.

Topics Covered in iOS Test-Driven Development:

- **The TDD Cycle:** Learn the concepts of Test-Driven Development and how to implement these concepts within an iOS application.
- **Test Expressions and Expectations:** Learn how to test both synchronous code using expressions and asynchronous code using expectations.
- **Test RESTful Networking:** Write tests to verify networking endpoints and the ability to mock the returned results.
- **Test Authentication:** Write tests which run against authenticated endpoints.
- **Legacy Problems:** Explore the problems legacy applications written without any unit tests or without thought of testing the code.
- **Breaking Dependencies into Modules:** Learn how to take dependencies within your code and compartmentalize these into their own modules with their own tests.
- **Refactoring Large Classes:** Learn how to refactor large unweilding classes into smaller more manageable and testable classes / objects.

One thing you can count on: after reading this book, you'll be prepared to write testable applications which you can have confidence in making changes too with the knowledge your tests will catch breaking changes.

About the Tutorial Team

The Tutorial Team is a group of app developers and authors who write tutorials at the popular website **raywenderlich.com.** We take pride in making sure each tutorial we write holds to the highest standards of quality. We want our tutorials to be well written, easy to follow, and fun.

If you've enjoyed the tutorials we've written in the past, you're in for a treat. The tutorials we've written for this book are some of our best yet — and this book contains detailed technical knowledge you simply won't be able to find anywhere else.