

Up to date for iOS 12,
Xcode 10 & Swift 4.2



iOS Test-Driven Development by Tutorials

FIRST EDITION

Learn real-world test-driven development

By the raywenderlich.com Tutorial Team

Joshua Greene & Mike Katz

iOS Test-Driven Development by Tutorials

By Joshua Greene & Michael Katz

Copyright ©2019 Razeware LLC.

Notice of Rights

All rights reserved. No part of this book or corresponding materials (such as text, images, or source code) may be reproduced or distributed by any means without prior written permission of the copyright owner.

Notice of Liability

This book and all corresponding materials (such as source code) 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 action of contract, tort or otherwise, arising from, out of or in connection with the software or the use of other dealing in the software.

Trademarks

All trademarks and registered trademarks appearing in this book are the property of their own respective owners.

Dedications

"For you girls — Madeline and Amelia. I love you very much."

— *Joshua Greene*

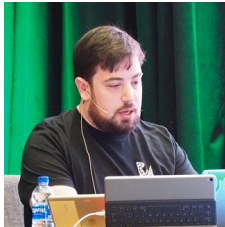
"Dedicated to the memory of my mother-in-law, Barbara Schwartz.
Her selflessness and dedication to teaching inspires me to give back
to the community and educate others."

— *Michael Katz*

About the Authors



Joshua Greene is an author of this book. He's an experienced software developer and has created many mobile apps. He enjoys software development because he likes to see his dreams come to life! When he's not slinging code, he's often watching Netflix, traveling or spending time with his family. He has two wonderful, beautiful daughters whom he loves very much. You can reach him on Twitter [@jrg_developer](https://twitter.com/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 Senior Manager of iOS development 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.

Table of Contents: Overview

Early Access Edition.....	12
Introduction.....	13
What You Need	16
Book License.....	17
Book Source Code & Forums	18
Section I: Hello, TDD!	19
Chapter 1: What Is TDD?	20
Chapter 2: The TDD Cycle	25
Section II: Beginning TDD.....	38
Chapter 3: Driving TDD	39
Chapter 4: Test Expressions	58
Chapter 5: Test Expectations	79
Chapter 6: Dependency Injection & Mocks.....	104
Section III: TDD with Networking	126
Chapter 7: Introducing Dog Patch	127
Chapter 8: RESTful Networking	132
Chapter 9: Authentication Client.....	133
Chapter 10: Authenticated Network Calls	134
Chapter 11: Websockets	135
Section IV: TDD in Legacy Apps	136

Chapter 12: Legacy Problems	138
Chapter 13: Dependency Maps	139
Chapter 14: Breaking Up Dependencies	140
Chapter 15: Adding Features to Existing Classes	141
Chapter 16: Refactoring Large Classes	142

Table of Contents: Extended

Early Access Edition	12
Introduction	13
About this book	14
Section introductions	14
How to read this book	15
What You Need	16
Book License	17
Book Source Code & Forums	18
Section I: Hello, TDD!	19
Chapter 1: What Is TDD?	20
Why should you use TDD?	21
What should you test?	22
But TDD takes too long!	23
When should you use TDD?	23
Key points	24
Chapter 2: The TDD Cycle	25
Getting started	26
Red: Write a failing test	26
Green: Make the test pass	28
Refactor: Clean up your code	28
Repeat: Do it again	29
TDDing init(availableFunds:)	29
TDDing addItem	31
Adding two items	34
Challenge	36
Key points	37

Section II: Beginning TDD	38
Chapter 3: Driving TDD	39
About the FitNess app	39
Your first test	40
Red-Green-Refactor	44
Test nomenclature	48
Structure of XCTestCase subclass	49
Your next set of tests	51
Using @testable import	52
Testing initial conditions	55
Refactoring	56
Challenge	57
Key points	57
Where to go from here?	57
Chapter 4: Test Expressions	58
Assert methods	59
View controller testing	65
Test ordering matters	69
Code coverage	72
Debugging tests	74
Challenge	77
Key points	78
Where to go from here?	78
Chapter 5: Test Expectations	79
Using an expectation	79
Testing for true asynchronicity	82
Waiting for notifications	84
Showing the alert to a user	89
Getting specific about notifications	93
Driving alerts from the data model	95
Using other types of expectations	101
Challenge	102

Key points	103
Where to go from here?.....	103
Chapter 6: Dependency Injection & Mocks	104
What's up with fakes, mocks, and stubs?	104
Understanding CMPedometer.....	105
Mocking	107
Handling error conditions	110
Getting actual data.....	116
Making a functional fake	119
Wiring up the chase view.....	121
Time dependencies	124
Challenge	125
Key points.....	125
Where to go from here?.....	125
Section III: TDD with Networking	126
Chapter 7: Introducing Dog Patch	127
Getting started.....	127
Understanding Dog Patch's architecture	130
Where to go from here?.....	131
Chapter 8: RESTful Networking	132
Chapter 9: Authentication Client	133
Chapter 10: Authenticated Network Calls.....	134
Chapter 11: Websockets.....	135
Section IV: TDD in Legacy Apps	136
Chapter 12: Legacy Problems.....	138
Chapter 13: Dependency Maps	139
Chapter 14: Breaking Up Dependencies.....	140

Chapter 15: Adding Features to Existing Classes ..	141
Chapter 16: Refactoring Large Classes	142

Early Access Edition

You're reading an early access edition of *iOS Test-Driven Development by Tutorials*. This edition contains a sample of the chapters that will be contained in the final release.

We hope you enjoy the preview of this book, and that you'll come back to help us celebrate the full launch of *iOS Test-Driven Development by Tutorials* later in 2019!

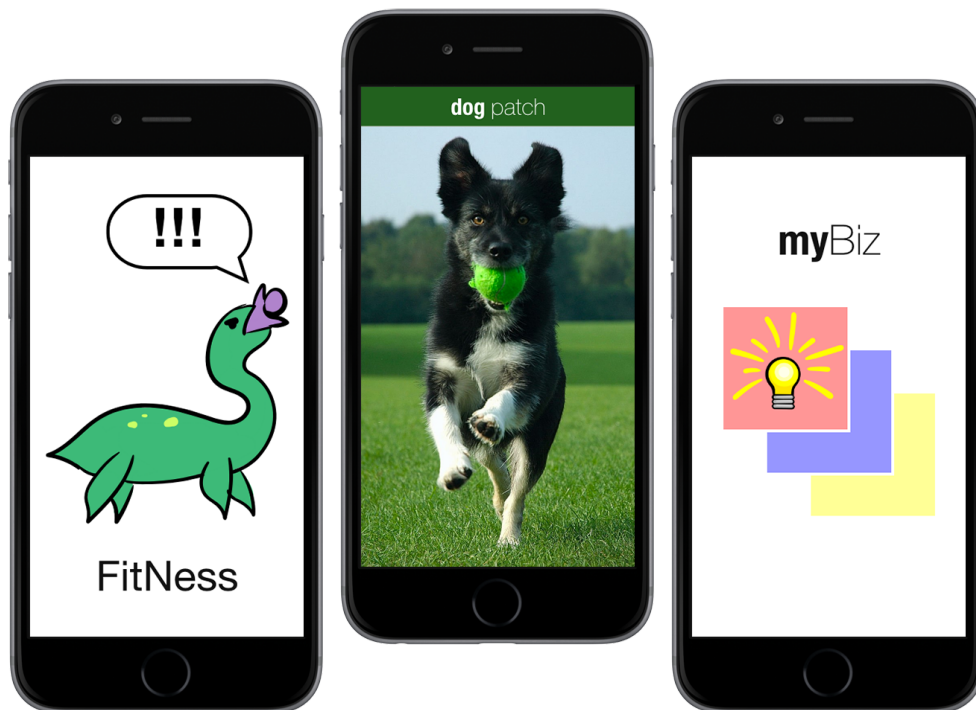
The best way to get update notifications is to sign up for our monthly newsletter. This includes a list of the tutorials that came out on raywenderlich.com that month, any important news like book updates or new books, and a list of our favorite development links for that month. You can sign up here:

- www.raywenderlich.com/newsletter

Introduction

Welcome to *iOS Test-Driven Development by Tutorials*! This book will teach you all about test-driven development (TDD) — the art of turning requirements into tests and tests into production code.

You'll get hands-on TDD experience by creating three real-world apps in this book:



By the end of this book, you'll have a strong understanding of TDD and be able to apply this knowledge to your own apps.

About this book

We wrote this book with beginner-to-intermediate developers in mind. The only requirements for reading this book are a basic understanding of Swift and iOS development.

If you've worked through our classic beginner books — the *Swift Apprentice* <https://store.raywenderlich.com/products/swift-apprentice> and the *iOS Apprentice* <https://store.raywenderlich.com/products/ios-apprentice> — or have similar development experience, you're ready to read this book. You'll also benefit from a working knowledge of design patterns — such as working through *Design Patterns by Tutorials* <https://store.raywenderlich.com/products/design-patterns-by-tutorials> — but this isn't strictly required.

As you work through this book, you'll progress from beginner topics to more advanced concepts.

Section introductions

I. Introduction

This is a high-level introduction to TDD, explaining why it's important and how it will help you.

You'll also be introduced to the **TDD Cycle** in this section. This is the foundation for how TDD works and guiding principles on the best way to apply it.

II. Beginning TDD

You'll learn the basics of TDD in this section, including **XCTest**, **test expressions**, **mocks** and **test expectations**.

The chapters in this section build an example app called **Fitness**. This is the premier fitness-coaching app based on the "Loch Ness" workout: You'll have to outrun, outswim and outclimb Nessie (or get eaten)!

III. TDD with Networking

This section will teach you about TDD and networking, including writing tests for **RESTful networking calls, authentication and even WebSockets**.

You'll create an app called **Dog Patch** throughout this section. Dog Patch lets dog lovers everywhere connect with kind breeders to help get the dog of their dreams. You'll be able to search, view and chat in real time.

IV. TDD in Legacy Apps

This section will teach you how to start TDD in a legacy app that wasn't created with TDD and doesn't have sufficient test coverage.

You'll update an app called **MyBiz** throughout this section. MyBiz is an enterprise resource planning (ERP) app for running a business, including employee management and scheduling, time tracking, payroll and inventory management.

How to read this book

If you're new to unit testing or TDD, you should read this book from cover to cover.

If you already have some experience with TDD, you can skip from chapter to chapter or use this book as a reference. You'll always be provided with a starter project in each chapter to get up and running quickly.

What's the *absolute best way* to read this book? Just start reading wherever makes sense to you!

What You Need

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

- **Xcode 10 or later.** Xcode is the main development tool for writing code in Swift. You need Xcode 10 at a minimum, since that version includes Swift 4.2. 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 4.2 and Xcode 10 — the code may not compile if you try to work with an older version.

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.



Book Source Code & Forums

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

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.

Section I: Hello, TDD!

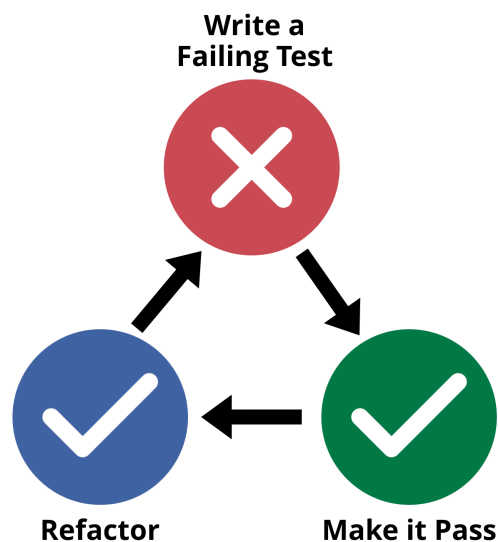
This section is a high-level introduction to test-driven development, how it works and why you should use it. You'll also learn about the TDD cycle in this chapter, and you'll use this throughout the rest of the book.

- **Chapter 1: What Is TDD?:** Test-driven development, or TDD, is an iterative way to develop software by iteratively making many small changes backed by tests.
- **Chapter 2: The TDD Cycle:** In the previous chapter, you learned that test-driven development boils down to a simple process called the TDD Cycle. It has four steps and is also called the **Red-Green-Refactor Cycle**.

Chapter 1: What Is TDD?

By Joshua Greene

Test-driven development, or TDD, is an iterative way to develop software by iteratively making many small changes backed by tests.



It has four steps:

1. Write a failing test
2. Make the test pass
3. Refactor
4. Repeat

This is called the **TDD Cycle**. It ensures you thoroughly and accurately test your code because your development is... driven by testing!

By writing a test followed by the production code to make it pass, you ensure your production code is testable and that it meets all of your requirements during development. As an added bonus, your tests act as documentation for your production code, describing how it works.

On the surface, the TDD process seems pretty simple. Well, I'm sorry to tell you that... wait, it actually *is* really simple!

Sure, there are special circumstances for how to implement this cycle at times, but that's where this book comes in! Once you get the hang of this process, it will become second nature. You'll learn a lot more about this process in the next chapter.

Why should you use TDD?

TDD is the single best way to ensure your software works and continues to work well into the future — well, that's quite a bold claim! Let me explain.

It's hard to argue against testing your code, but you don't have to follow TDD to do this. For example, you could write all of your production code and then write all of your tests. Alternatively, you could skip writing tests altogether and, instead, manually test your code. Why is TDD better than these options?

Good tests ensure your app works as expected. However, not all tests are "good." Writing tests for the sake of having tests isn't a worthwhile exercise. Rather, good tests are failable, repeatable, quick to run and maintainable.

TDD provides methodology that ensures your tests are good:

- The first step is to write a **failing test**. By definition, this proves the test is failable. Tests that can't fail aren't very useful. Rather, they waste valuable CPU time.
- Before you're allowed to write a new test, all other previous tests must pass. This ensures that your tests are **repeatable**: You don't just run the single test you're working on, but rather, you constantly run *all* of the tests.
- By frequently running every test, you're incentivized to make sure tests are **quick** to run. All of your tests should take seconds to run — preferably, one second or less.

A single test that takes a hundred milliseconds is too slow: After only ten tests, your entire test suite will take one second to run. After fifty tests, it takes five seconds. After several seconds, no one runs all of the tests because it takes too long.

- When you refactor, you update both your production and test code. This ensures your tests are **maintained**: You're constantly keeping them up-to-date.
- By iteratively writing production code and tests in parallel, you ensure your code is **testable**. If you were to write tests after completing the code, it's likely the production code would require quite a bit of refactoring to fully unit test.

Nonetheless, the devil's advocate in you may say, "But you *could* write good tests without following TDD." You definitely *could*, but you may struggle to succeed. You can definitely do it in the short term, but it's much more difficult in the long term. You'd need to be disciplined about writing good tests. Before long, you'd likely create some sort of system to ensure that you're writing good tests... you'd likely find yourself doing a variant of TDD!

What should you test?

Better test coverage doesn't always mean your app is better tested. There are things you should test and others you shouldn't. Here are the do's and don'ts:

- **Do** write tests for code that can't be caught in an automated fashion otherwise. This includes code in your classes' methods, custom getters and setters and most anything else you write yourself.
- **Don't** write tests for generated code. For example, it's not worthwhile to write tests for generated getters and setters. Swift does this very well, and you can trust it works.
- **Don't** write tests for issues that can be caught by the compiler. If the tested issue would generate an error or warning, Xcode will catch it for you.
- **Don't** write tests for dependency code, such as first- or third-party frameworks your app uses. The framework authors are responsible for writing those tests. For example, you shouldn't write tests for UIKit classes because UIKit developers are responsible for writing these. However, you should write tests for your custom subclasses thereof: This is your custom code, so you're responsible for writing the tests.

An exception to the above is writing tests in order to determine how a framework works. This can be very useful to do. However, you don't need to keep these tests long term. Rather, you should delete them afterwards.

Another exception is "sanity tests" that prove third-party code works as you expect.

These sort of tests are useful if the library isn't fully stable, or you don't trust it entirely. In either case, you should really scrutinize whether or not you want to use the library at all — is there a better option that's more trustworthy?

But TDD takes too long!

The most common complaint about TDD is that it takes too long — usually followed by exclamation point(s) or sad-face emojis.

Fortunately, TDD gets faster once you get used to doing it. However, the truth is that compared to not writing any tests at all, you're writing more code ultimately. It likely will take a little more time to develop *initially*.

That said, there's a really big hole in this argument: The real time cost of development isn't just writing the initial, first-version production code. It also includes adding new features over time, modifying existing code, fixing bugs and more. In the long run, following TDD takes *much less time* than not following it because it yields more maintainable code with fewer bugs.

There's also another cost to consider: customer impact of bugs in production. The longer an issue goes undiscovered, the more expensive it is. It can result in negative reviews, lost trust and lost revenue.

If an issue is caught during development, it's easier to debug and quicker to fix. If you discovered it weeks later, you'd spend substantially more time getting up to speed on the code and tracking down the root cause. By following TDD, your tests ultimately help safeguard and protect your app against bugs.

When should you use TDD?

TDD can be used during any point in a product's life cycle: new development, legacy apps and everything in between. However, how and where you start TDD does depend on the state of your project. This book will cover how to approach many of these situations!

However, an important question to ask: Should your project use TDD at all?

As a general rule of thumb, if your app is going to last more than a few months, will have multiple releases and/or require complex logic, you're likely better off using TDD than not.

If you're creating an app for a hackathon, test project or something else that's meant to be temporary, you should evaluate whether TDD makes sense. If there's really only going to be one version of the app, you might not follow TDD or might only do TDD for critical or difficult parts.

Ultimately, TDD is a tool, and it's up to you to decide when it's best to use it!

Key points

In this chapter, you learned what TDD is, why you should use it, what to test and when to use it. Here are the key points to remember:

- TDD offers a consistent method to write good tests.
- Good tests are failable, repeatable, quick to run and maintainable.
- Write tests for code that you're responsible for maintaining. Don't test code that's automatically generated or code within dependencies.
- The real cost of development includes initial coding time, adding new features over time, modifying existing code, fixing bugs and more. TDD reduces maintenance costs and quantity of bugs, often making it the most cost effective approach.
- TDD is most useful for long-term projects lasting more than a few months or having multiple releases.

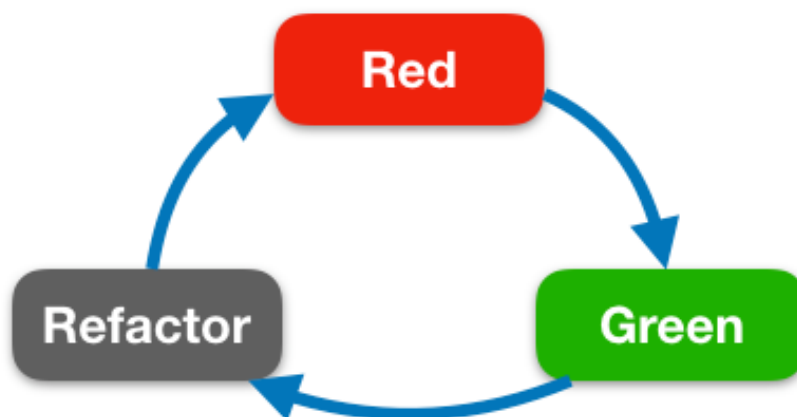
Chapter 2: The TDD Cycle

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:

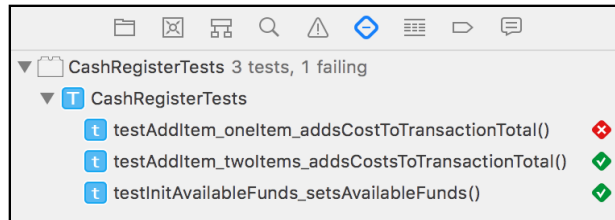
1. **Red**: Write a failing test, before writing any app code.
2. **Green**: Write the bare minimum code to make the test pass.
3. **Refactor**: Clean up both your app and test code.
4. **Repeat**: Do this cycle again until all features are implemented.

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.

Section II: Beginning TDD

This section will teach you the basics of test-driven development. You'll learn about setting up your app for TDD, test expressions, dependency injection, mocks and test expectations.

Along the way, you'll build a fitness app to learn the basics of TDD through hands-on practice.

- **Chapter 3: TDD App Setup:** 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.
- **Chapter 4: Test Expressions:** 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.
- **Chapter 5: Test Expectations:** In the previous chapters you built out the app's state based upon what the user can do with the Start button. The main part of the app relies on responding to changes as the user moves around and records steps. These actions create events outside the program's control. XCTestExpectation is the tool for testing things that happen outside the direct flow.
- **Chapter 6: Dependency Injection & Mocks:** In this chapter you'll learn how to use mocks to test code that depends on system or external services without needing to call services: They 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 like CoreMotion and HealthKit.

Chapter 3: Driving TDD

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

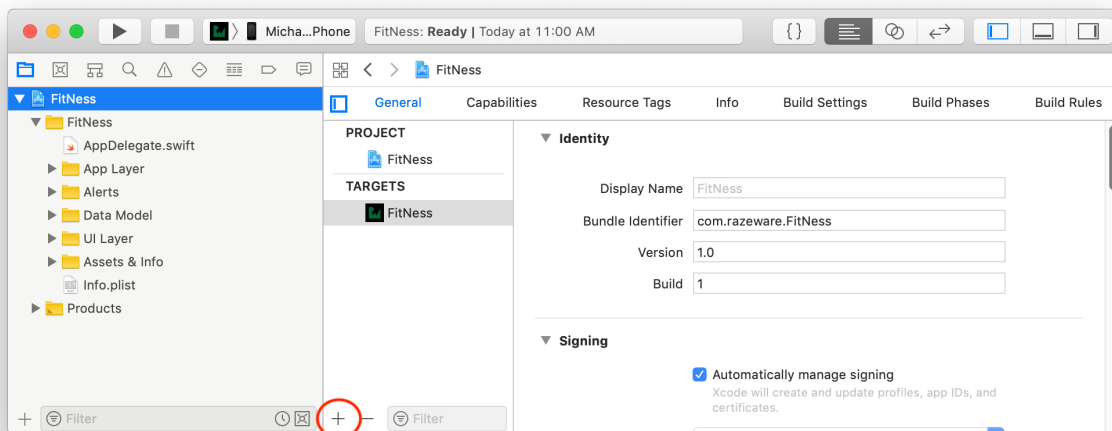
First things first: You can't run any tests without a **test target**. A test target is a binary that contains the test code, and it's executed during the test phase. It's built alongside the app, but is not included in the app bundle.

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.

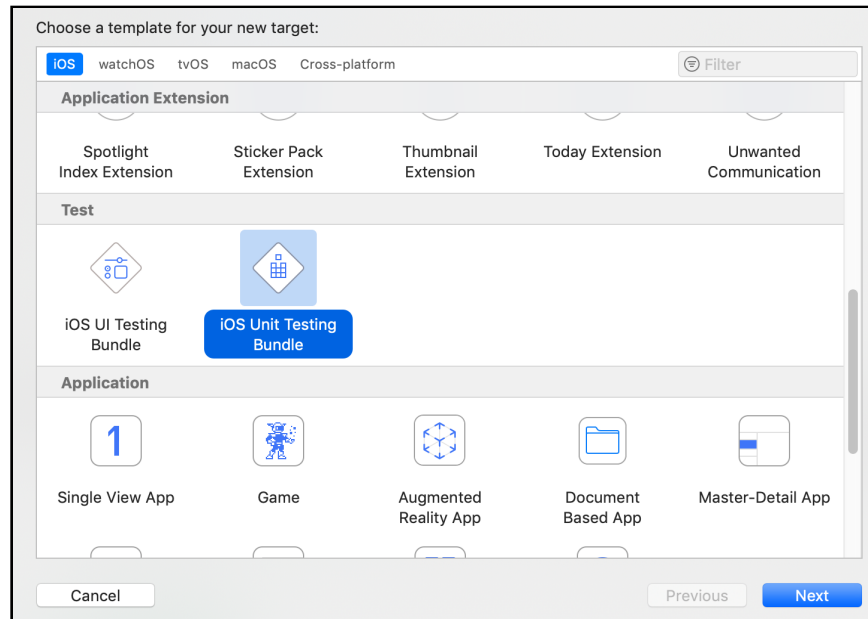
The TDD philosophy treats tests as first-class code, meaning they should fit the same standards as your production code in terms of readability, naming, error handling and coding conventions.

Adding a test target

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

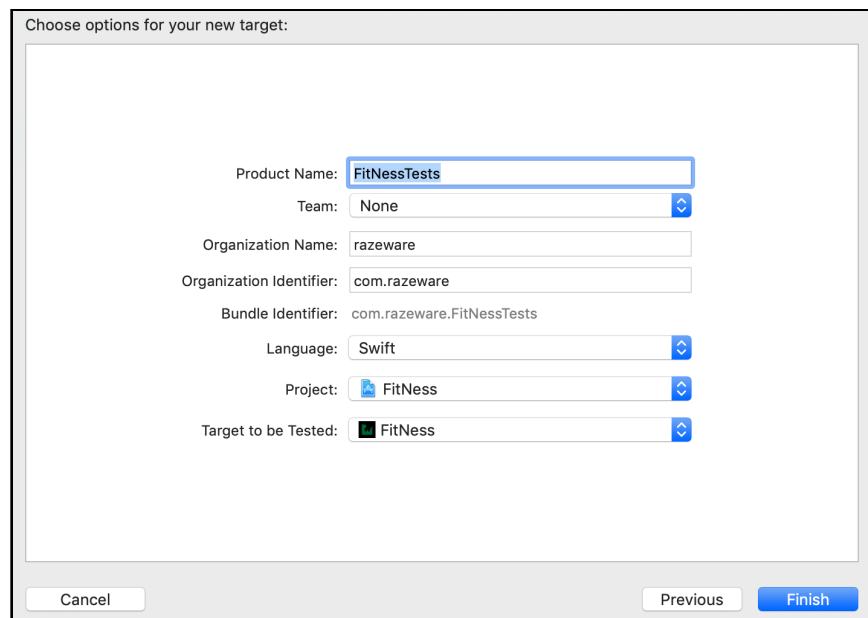


Scroll down to the **Test** section and select **iOS Unit Testing Bundle**. Click **Next**.



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 AppState class holds the knowledge of which state the app is currently in.

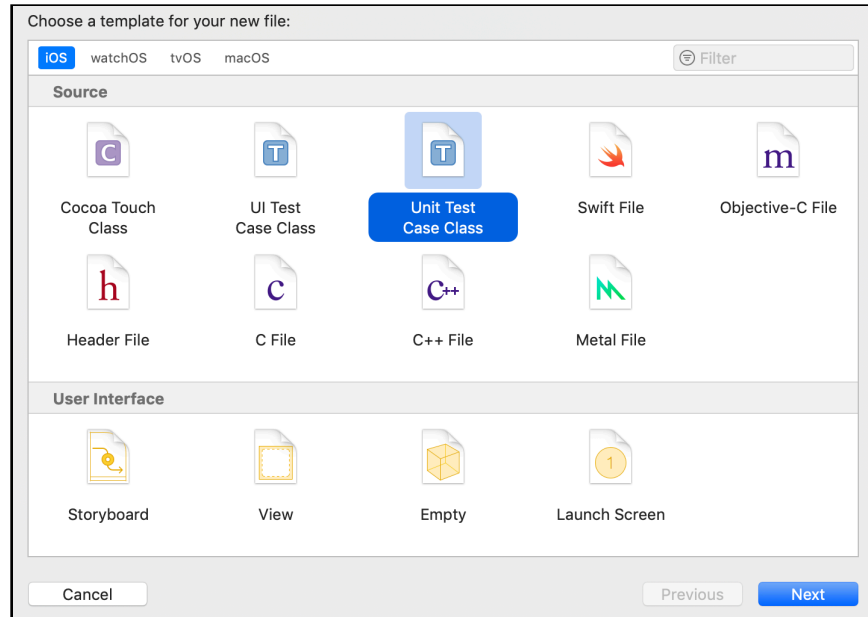
The minimum functionality to start the app is to have the **Start** button put the app into a started, or in-progress, state. There are two statements that can be made to support this goal:

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.

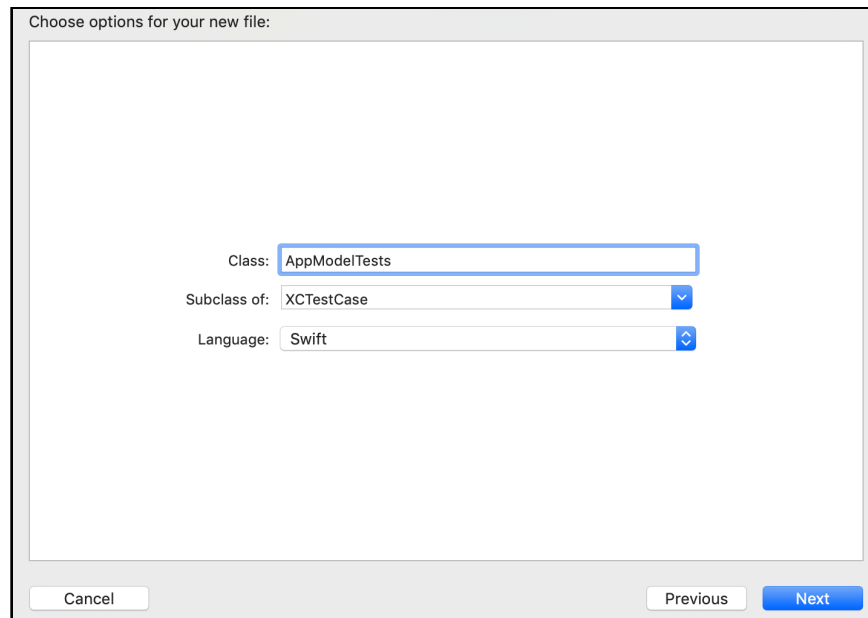
The statements are actually assertions and what you'll use to define test cases.

Adding a test class

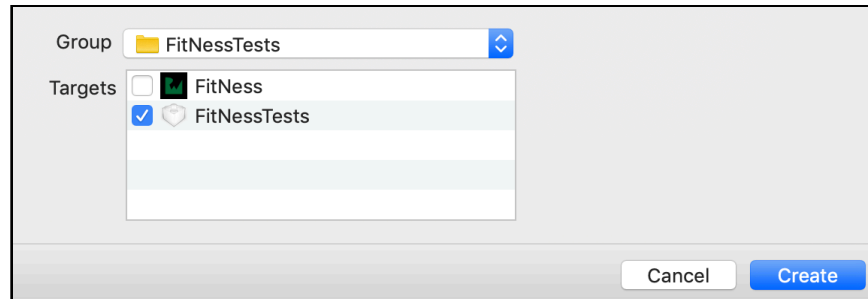
Right-click on **FitNesTests** in the project navigator. Select **New File**. In the **iOS** tab, select **Unit Test Case Class** and click **Next**.



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.

```
21   func testAppModel_whenInitialized_isInNotStartedState() {  
23      let sut = AppModel()  
24      let initialState = sut.appState  
25      XCTAssertEqual(initialState, AppState.notStarted)  
26  }
```

- 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 + Command + 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.

```
21  ✓ func testAppModel_whenInitialized_isInNotStartedState() {  
23      let sut = AppModel()  
24      let initialState = sut.appState  
25      XCTAssertEqual(initialState, AppState.notStarted)  
26  }
```

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_isInProgressState() {  
    // 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() {  
}
```

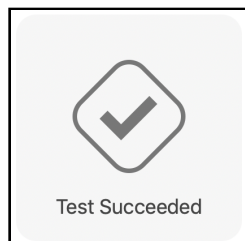
Now, the app should compile. Run the tests.



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_isInProgressState() {
```

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. `isInProgressState` 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(observedState, 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 derived 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 OUnit. 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 are 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 **FitNesseTests** 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
```

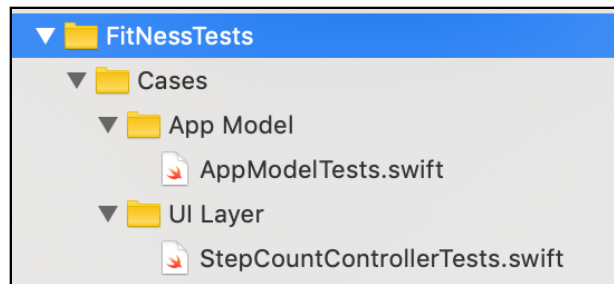
- **Cases:** The group for the test cases, and these are organized in a parallel structure to the app code. This makes it really easy to navigate between the app class and its tests.

- **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 MARKS 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`, `XCTAssertLessThanOrEqualTo`, `XCTAssertGreaterThanOrEqualTo`
- 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!

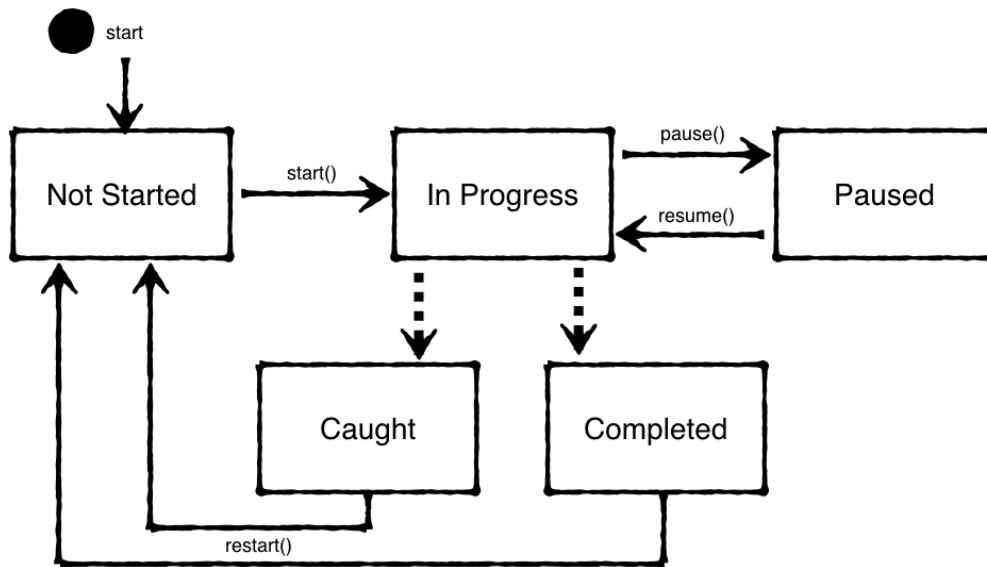
App state

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_isInProgressState()` 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_isInProgressState()`:

```
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`:

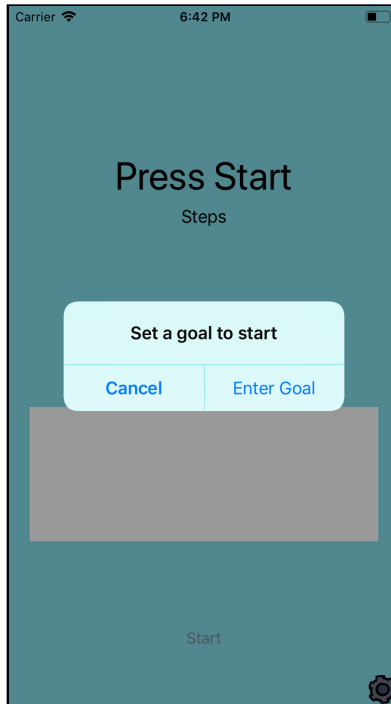
```
func givenGoalSet() {  
    AppModel.instance.dataModel.goal = 1000  
}
```

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)  
}
```

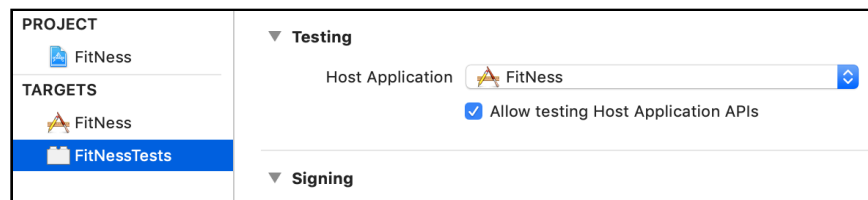
The test builds, but does not pass, because `chaseView` is `nil`. What gives?

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.

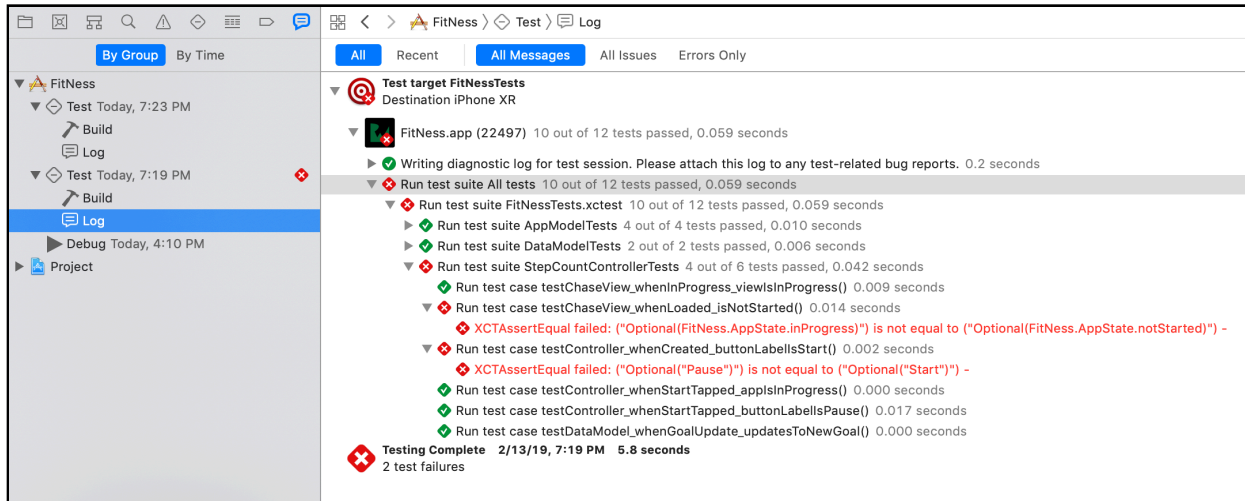
Test ordering matters

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:

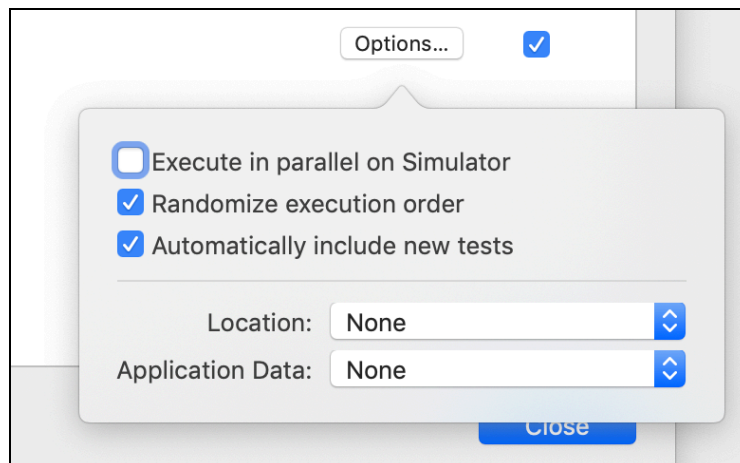
```
override func tearDown() {
    AppModel.instance.dataModel.goal = nil
    AppModel.instance.restart()
    sut.updateUI()
    super.tearDown()
}
```

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.

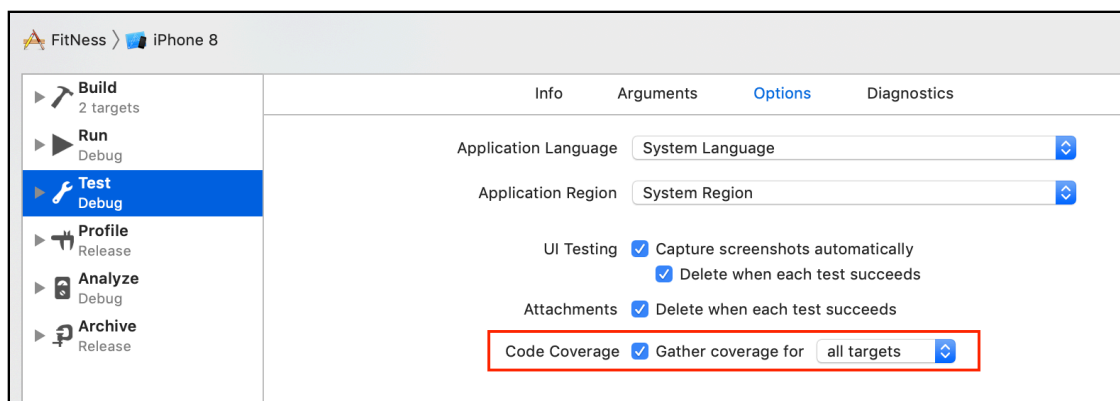
Edit the FitNess scheme. Select the **Test** action. In the center pane, next to **FitNessTests** is an **Options...** button. Click that and, in the pop-up, check **Randomize execution order**. This will cause the tests to run in a random order each time.



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.



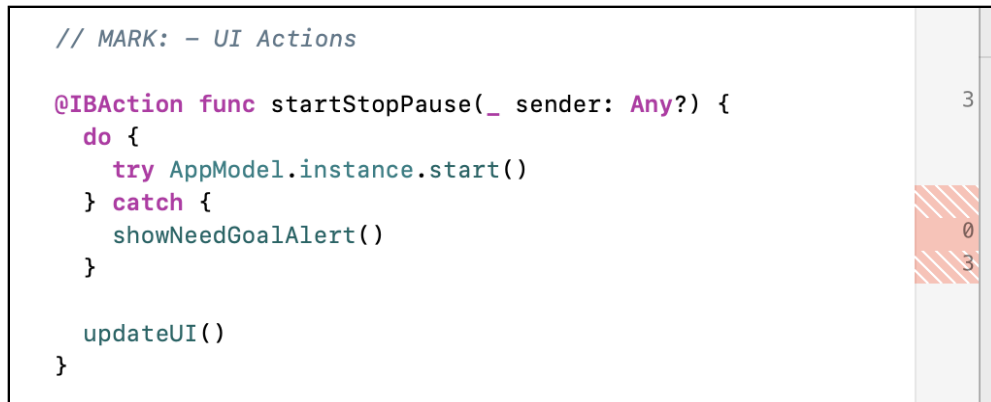
Run the tests again. After the tests succeed, open the Report navigator. Under the latest test, there will be three reports: **Build**, **Coverage** and **Log**. Select **Coverage** to display the coverage report.

FitNess		Name	Coverage
Test Today, 4:48 PM		FitNess.app	64.7%
Build		AppModel.swift	100.0%
Coverage		DataModel.swift	100.0%
Log		ChaseView.swift	57.1%
		StepCountController.swift	58.5%

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

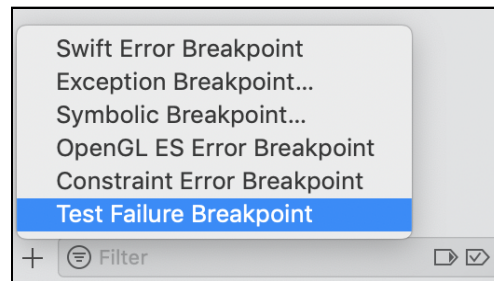
var caught: Bool {
    return nessie.distance >= distance
}
```

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.

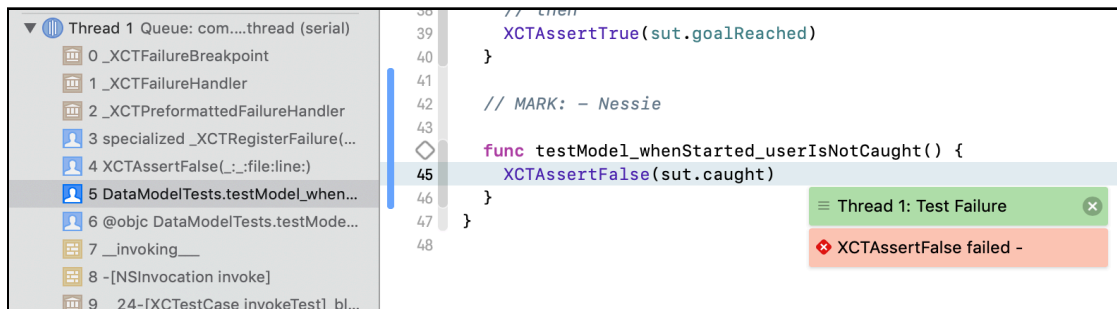
Even with the updated code, the test still fails. There are several ways to go about diagnosing the problem. As you've already seen there are a few things to check:

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

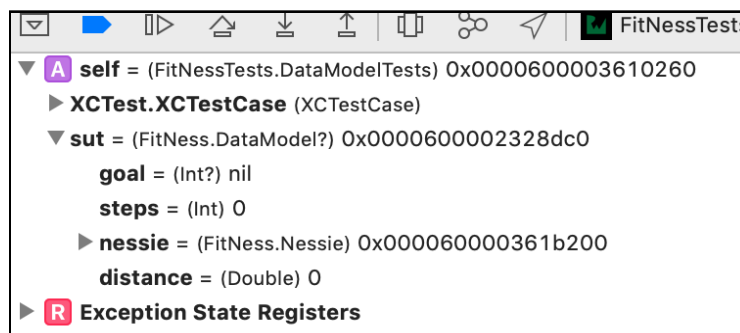
A good next step is to try out the debugger. In the **Breakpoint navigator**, click the + all the way at the bottom. Select **Test Failure Breakpoint**.



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`.

Chapter 5: Test Expectations

By Michael Katz

In the previous chapters you built out the app's state based upon what the user can do with the Start button. The main part of the app relies on responding to changes as the user moves around and records steps. These actions create events outside the program's control. `XCTestExpectation` is the tool for testing things that happen outside the direct flow.

In this chapter you'll learn:

- General test expectations
- Notification expectations

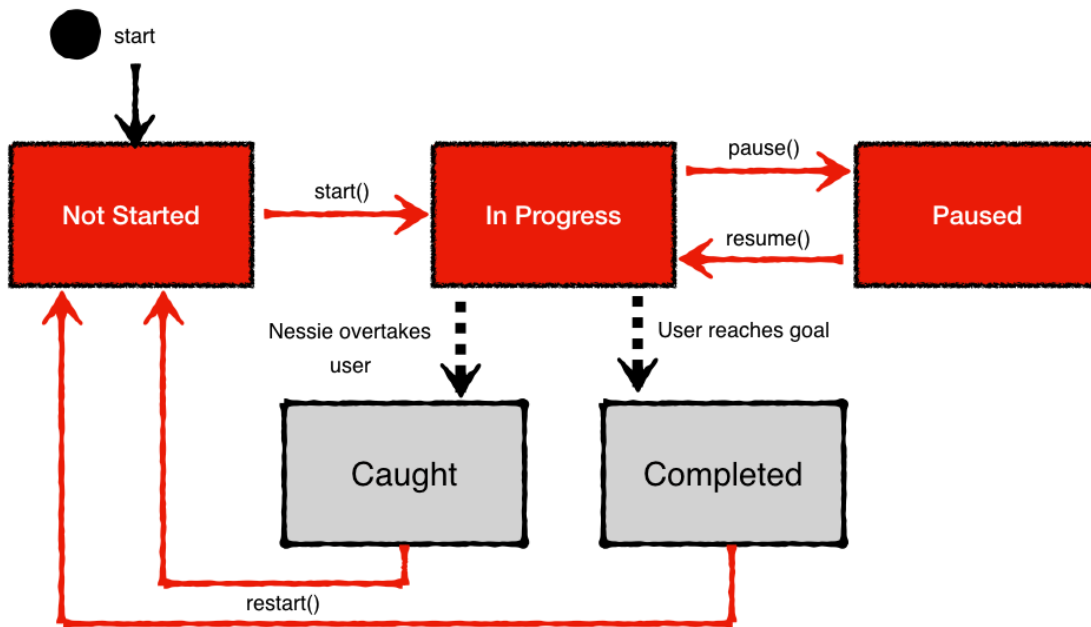
Use this chapter's starter project instead of continuing on from the previous' final, as it has some additions to help you out.

Using an expectation

XCTest expectations have two parts: the **expectation** and a **waiter**. An expectation is an object that you can later fulfill. The `wait` method of `XCTestCase` tells the test execution to wait until the expectation is fulfilled *or* a specified amount of time passes.

In the last chapter you built out the app states corresponding to direct user action: in progress, paused, and not started. In this chapter you'll add support for caught and completed.

These state transitions occur in response to asynchronous events outside the user's control.



The red-shaded states have already been built. You'll be adding the grey states.

Writing an asynchronous test

In order to react to an asynchronous event, the code needs a way to listen for a change. This is commonly done through a closure, a delegate method, or by observing a notification.

To test caught and completed state changes that asynchronously update in `AppModel`, you'll add a callback closure. The first step is to write the test!

Open **AppModelTests.swift** and add the following test under `// MARK: - State Changes`:

```

func testAppModel_whenStateChanges_executesCallback() {
    // given
    givenInProgress()
    var observedState = AppState.notStarted

    // 1
    let expected = expectation(description: "callback happened")
    sut.stateChangedCallback = { model in
        observedState = model.appState
        // 2
        expected.fulfill()
    }
}
  
```



```
// when
sut.pause()

// then
// 3
wait(for: [expected], timeout: 1)
XCTAssertEqual(observedState, .paused)
}
```

This test updates the `appState` using `sut.pause` then checks that `stateChangedCallback` gets triggered and sets `observedState` to the new value. You are using a few new things in this test:

1. `expectation(description:)` is an `XCTestCase` method that creates an `XCTestExpectation` object. The description helps identify a failure in the test logs. You'll see shortly how `expected` is used to track if and when the expectation is fulfilled.
2. `fulfill()` is called on the expectation to indicate it has been fulfilled - specifically, the callback has occurred. Here `stateChangedCallback` will trigger on `sut` when a state change occurs.
3. `wait(for:timeout:)` causes the test runner to pause until all expectations are fulfilled or the timeout time (in seconds) passes. The assertion will not be called until the wait completes.

The test won't compile, because `stateChangedCallback` doesn't yet exist. In **AppModel.swift**, add the following to the class:

```
var stateChangedCallback: ((AppModel) -> ())?
```

Adding this property allows the test to build. Now run it, and you'll see the following failure in the console:

```
Asynchronous wait failed: Exceeded timeout of 1 seconds, with unfulfilled expectations: "callback happened".
```

The expectation never got fulfilled, so the test failed after the 1 second wait timeout.

To fix it, change `appState` in `AppModel` to match the following:

```
private(set) var appState: AppState = .notStarted {
    didSet {
        stateChangedCallback?(self)
    }
}
```

The callback is now triggered each time `AppState` is set.

Back in **AppModelTests.swift**, clean up the callback reference by adding the following to the top of `tearDown`:

```
sut.stateChangedCallback = nil
```

Run the test again, and now it will pass!

Note: It is best practice to always call `fulfill` in the completion block, then test for errors or other negative conditions using `XCTAssert` after the wait. Timeout should not be used to signal a test failure, as it adds significant time to the test.

Testing for true asynchronicity

The last test checks that the callback is called in direct response to an update on the `sut`. Next, you'll tackle a more indirect usage via updates to the view controller. In **StepCountControllerTests.swift** at the end of `// MARK: - Terminal States` add the following two tests:

```
func testController_whenCaught_buttonLabelIsTryAgain() {
    // given
    givenInProgress()
    let exp = expectation(description: "button title change")
    let observer = ButtonObserver()
    observer.observe(sut.startButton, expectation: exp)

    // when
    whenCaught()

    // then
    waitForExpectations(timeout: 1)
    let text = sut.startButton.title(for: .normal)
    XCTAssertEqual(text, AppState.caught.nextStateButtonLabel)
}

func testController_whenComplete_buttonLabelIsStartOver() {
    // given
    givenInProgress()
    let exp = expectation(description: "button title change")
    let observer = ButtonObserver()
    observer.observe(sut.startButton, expectation: exp)

    // when
    whenCompleted()

    // then
    waitForExpectations(timeout: 1)
    let text = sut.startButton.title(for: .normal)
    XCTAssertEqual(text, AppState.completed.nextStateButtonLabel)
}
```

These tests observe the `startButton` title to confirm it properly updates after model state changes.

`observe(_:expectation:)` will fulfill the passed expectation (`exp`) when the `titleLabel` of `sut.startButton` is updated. This requires the `ButtonObserver` helper class, which you're about to create!

Add a new **Swift File** to the **Test Classes** group and name it **ButtonObserver.swift**. Place the following in the file:

```
import XCTest

class ButtonObserver: NSObject {

    var expectation: XCTestExpectation?
    weak var button: UIButton?

    func observe(_ button: UIButton,
                 expectation: XCTestExpectation) {
        self.expectation = expectation
        self.button = button

        button.addObserver(self, forKeyPath: "titleLabel.text",
                           options: [.new], context: nil)
    }

    override func observeValue(
        forKeyPath keyPath: String?,
        of object: Any?,
        change: [NSKeyValueChangeKey : Any]?,
        context: UnsafeMutableRawPointer?) {
        expectation?.fulfill()
    }

    deinit {
        button?.removeObserver(self, forKeyPath: "titleLabel.text")
    }
}
```

`ButtonObserver` observes a `UIButton` for changes to its `titleLabel`'s text by using **Key-Value Observing**. When the text changes, a callback is made to `observeValue(forKeyPath:of:change:context:)`. This object holds on to the supplied `XCTestExpectation` and fulfills it in that callback.

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

```
func whenCaught() {
    AppModel.instance.setToCaught()
}
```

```
func whenCompleted() {  
    AppModel.instance.setToComplete()  
}
```

Build and run the `StepCountControllerTests` tests, and you'll see a couple failures in the console:

```
XCTAssertEqual failed: ("Optional("Pause")") is not equal to  
("Optional("Try Again")")  
  
XCTAssertEqual failed: ("Optional("Pause")") is not equal to  
("Optional("Start Over")")
```

The button titles aren't updating when `whenCaught()` and `whenCompleted()` are called in your test, because there aren't yet any hooks in the production code to do this. Fix that by adding the following to `viewDidLoad` in **StepCountController.swift**:

```
AppModel.instance.stateChangedCallback = { model in  
    DispatchQueue.main.async {  
        self.updateUI()  
    }  
}
```

`stateChangedCallback` is now used to update the UI when `appState` is updated in the model. Now the tests will pass and you're ready to move on.

Note: Stopping execution in the debugger doesn't pause the wait timeout. You just added a bunch of code, and if there was a mistake you might go back and debug the problem. This is common when writing tests, especially when they do not behave as expected. When the debugger pauses at a breakpoint and you explore for the logic error, be mindful that the test will probably fail due to timeout. Simply disable or remove the breakpoint and re-run once the issue is corrected.

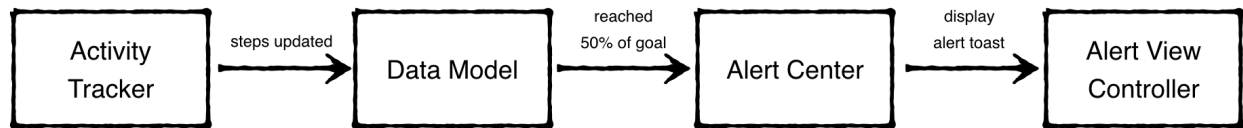
Waiting for notifications

In the next phase of app building, you'll add a feature to visually notify the users when an event happens, such as meeting a milestone goal or when Nessie catches up.

In addition to fulfilling expectations in arbitrary callbacks, there is also a feature that allows the test to wait for User Notifications.

Building the alert center

One important feature for an activity app or game is to update the user when important events happen. In **FitNess** these updates are managed by an **AlertCenter**. When something interesting happens, the code will post Alerts to the AlertCenter. The alert center is responsible for managing a stack of messages to display to the user.



AlertCenter uses Notifications to communicate with the view controllers which handle the alerts on screen. Because this happens asynchronously, it's a good case to test using XCTestExpectation.

A stub implementation of AlertCenter and AlertCenterTests have been added to the project to speed things up.

To test out the notification behavior add the following test in **AlertCenterTests.swift**:

```
func testPostOne_generatesANotification() {
    // given
    let exp = expectation(forNotification: AlertNotification.name,
                          object: sut,
                          handler: nil)

    let alert = Alert("this is an alert")

    // when
    sut.postAlert(alert: alert)

    // then
    wait(for: [exp], timeout: 1)
}
```

`expectation(forNotification:object:handler:)` creates an expectation that fulfills when a notification posts. In this case, when `AlertNotification.name` is posted to `sut`, the expectation is fulfilled. The test then posts a new `Alert` and waits for that notification to be sent.

Note that it's not generally a good idea to use a `wait` as the test assertion. It's better to use an explicit `assert` call. `wait` only tests that an expectation was fulfilled and does not make any claims about the app's logic. You'll test the contents of the notification a little later in this chapter.

Build and test, and this test will fail. If you look at the error in the console, you'll see a timeout failure:

```
Asynchronous wait failed: Exceeded timeout of 1 seconds, with unfulfilled expectations: "Expect notification 'Alert' from FitNess.AlertCenter".
```

Time to implement the application code to fix this! In **AlertCenter.swift**, replace the stub implementation of `postAlert(alert:)` with the following:

```
func postAlert(alert: Alert) {
    let notification = Notification(name: AlertNotification.name,
                                    object: self)
    notificationCenter.post(notification)
}
```

This creates and posts the Notification your test is listening for. Note that the passed alert isn't used currently, but you'll circle back to this later.

Build and test. And the test will pass! :]

Waiting for multiple events

Next, try testing if posting two alerts sends two notifications. Add the following to the end of `AlertCenterTests`:

```
func testPostingTwoAlerts_generatesTwoNotifications() {
    //given
    let exp1 = expectation(
        forNotification: AlertNotification.name,
        object: sut,
        handler: nil)
    let exp2 = expectation(
        forNotification: AlertNotification.name,
        object: sut,
        handler: nil)
    let alert1 = Alert("this is the first alert")
    let alert2 = Alert("this is the second alert")

    // when
    sut.postAlert(alert: alert1)
    sut.postAlert(alert: alert2)

    // then
    wait(for: [exp1, exp2], timeout: 1)
}
```

This creates two expectations waiting for `AlertNotification.name`, posts two different alerts, and waits for both alerts to notify.

Build and test, and it will pass. However, this test is a little naïve. To see how, delete this line:

```
sut.postAlert(alert: alert2)
```

Now you're only posting one of the two alerts tied to expectations the wait requires.

Test again, and it will still pass! This is because the two expectations are expecting the same thing. They run in parallel—they don't stack. So as soon as one alert is posted, both expectations are fulfilled.

To solve this conundrum, you can use notification expectation's `expectedFulfillmentCount` property refine the fulfillment condition. Replace `testPostingTwoAlerts_generatesTwoNotifications()` with the following:

```
func testPostingTwoAlerts_generatesTwoNotifications() {  
    //given  
    let exp = expectation(forNotification: AlertNotification.name,  
                          object: sut,  
                          handler: nil)  
    exp.expectedFulfillmentCount = 2  
    let alert1 = Alert("this is the first alert")  
    let alert2 = Alert("this is the second alert")  
  
    // when  
    sut.postAlert(alert: alert1)  
  
    // then  
    wait(for: [exp], timeout: 1)  
}
```

Setting `expectedFulfillmentCount` to two means the expectation won't be met until `fulfill()` has been called twice before the timeout.

Run the test, and you'll see it fails because you only called `postAlert` once. This is good proof your test is working as expected!

In the `when` section, add back the second `postAlert` under `sut.postAlert(alert: alert1)`:

```
sut.postAlert(alert: alert2)
```

Run the test again, and you'll see it pass.

Expecting something not to happen

Good test suites not only test when things happen according to plan, but also check that certain side effects do not occur. One of things the app should not do is spam the user with alerts. Therefore, if a specific alert is posted twice, it should only generate one notification.

And of course, you can test for this scenario. Add the following test:

```
func testPostDouble_generatesOnlyOneNotification() {  
    //given  
    let exp = expectation(forNotification: AlertNotification.name,  
                          object: sut,  
                          handler: nil)  
    exp.expectedFulfillmentCount = 2  
    exp.isInverted = true  
    let alert = Alert("this is an alert")  
  
    // when  
    sut.postAlert(alert: alert)  
    sut.postAlert(alert: alert)  
  
    // then  
    wait(for: [exp], timeout: 1)  
}
```

This is almost exactly like the last one, except for this line:

```
exp.isInverted = true
```

When an expectation is **inverted** it indicates this test fails if the expectation is fulfilled and succeeds if the wait times out. Put another way, this test will fail if two notifications are triggered by the two alerts.

Right now, the test fails because the application code currently allows multiple alerts to post.

Open **AlertCenter.swift**. Add the following instance variable:

```
private var alertQueue: [Alert] = []
```

The alertQueue will be an important part of AlertCenter. It will help manage a potentially large stack of messages for the user, as they can accumulate in the background.

Next add the following statements to the top of postAlert(alert:):

```
guard !alertQueue.contains(alert) else { return }  
alertQueue.append(alert)
```


If the same alert is passed to `postAlert(alert:)` twice, the second one will be ignored.

Build and test again. All green!

Be sure to run all the tests from time to time to make sure fixes for one test don't break another.

Showing the alert to a user

In the app's architecture, the `RootViewController` is responsible for showing alerts to the user via its `alertContainer` view.

Create a new **Unit Test Case Class** file in the **App Layer** folder, under **Cases**. Name it **`RootViewControllerTests.swift`**.

Add the following import:

```
@testable import FitNess
```

Next, replace the test boilerplate in the class with:

```
var sut: RootViewController!

override func setUp() {
    super.setUp()
    sut = loadRootViewController()
}

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

Finally, add a test for the base condition: that is, when the view controller is loaded, there are no alerts showing:

```
// MARK: - Alert Container

func testWhenLoaded_noAlertsAreShown() {
    XCTAssertTrue(sut.alertContainer.isHidden)
}
```

Run this and confirm it passes.

Next, add the following to test that the alert container is shown when there is an alert:

```
func testWhenAlertsPosted_alertContainerIsShown() {
    // given
    let exp = expectation(forNotification: AlertNotification.name,
```

```

                                object: nil, handler: nil)
    let alert = Alert("show the container")

    // when
    AlertCenter.instance.postAlert(alert: alert)

    // then
    wait(for: [exp], timeout: 1)
    XCTAssertFalse(sut.alertContainer.isHidden)
}

```

An expectation will be fulfilled by `AlertNotification.name` and `postAlert(alert:)` is called to ultimately trigger the notification. After waiting for the expectation, `XCTAssertFalse` checks the `alertContainer` is visible.

Now it's time to get the test to pass by adding the code to show the alert. Go back to **RootViewController.swift** and add the following at the bottom of `viewDidLoad`:

```

AlertCenter.listenForAlerts { center in
    self.alertContainer.isHidden = false
}

```

`AlertCenter.listenForAlerts(_:)` is a helper method that you'll create to register for alert notifications, and run the passed closure. The closure will unhide the `alertContainer` when triggered.

In **AlertCenter.swift**, in the "class helpers" extension add:

```

class func listenForAlerts(
    _ callback: @escaping (AlertCenter) -> ()) {

    instance.notificationCenter
        .addObserver(forName: AlertNotification.name,
                      object: instance, queue: .main) { _ in
        callback(instance)
    }
}

```

`listenForAlerts(_:)` adds `AlertCenter` as an observer for the `AlertNotification.name` notification that triggers the callback. This will result in `alertContainer` displaying in `RootViewController`.

Build and run your new test and it should now pass.

Continuous refactoring

When you only run `testWhenLoaded_noAlertsAreShown()`, it will pass. If you run all the tests in `RootViewControllerTests`, then `testWhenLoaded_noAlertsAreShown()` may fail.

That is because the sut state is tied to the running UIApplication and is preserved between runs. If `testWhenAlertsPosted_alertContainerIsShown()` runs first and displays the alert, it will still be there when `testWhenLoaded_noAlertsAreShown()` checks if any are displayed.

To resolve this issue, you'll refactor the code and build a way to clear out all the alerts and reset the view between tests.

First, you need an interface to the state of `AlertCenter`. Add the following test to **AlertCenterTests.swift**:

```
// MARK: - Alert Count
func testWhenInitialized_AlertCountIsZero() {
    XCTAssertEqual(sut.alertCount, 0)
}
```

This means that `AlertCenter` needs an `alertCount` variable for the test to compile. Add the following property to the class in **AlertCenter.swift**:

```
var alertCount: Int {
    return alertQueue.count
}
```

Build and test `testWhenInitialized_AlertCountIsZero()` and you'll see it now passes.

When adding new functionality, it's important to cover the basic conditions as well. Add the following to **AlertCenterTests.swift**:

```
func testWhenAlertPosted_CountIsIncreased() {
    // given
    let alert = Alert("An alert")

    // when
    sut.postAlert(alert: alert)

    // then
    XCTAssertEqual(sut.alertCount, 1)
}

func testWhenCleared_CountIsZero() {
    // given
    let alert = Alert("An alert")
    sut.postAlert(alert: alert)

    // when
    sut.clearAlerts()

    // then
    XCTAssertEqual(sut.alertCount, 0)
}
```

`testWhenAlertPosted_CountIsIncreased()` tests that posting an alert increases the `alertCount` you added for the prior test.

`testWhenCleared_CountIsZero()` tests a new method, `clearAlerts()`, which you need to create. First, you'll want to run it in `tearDown()`, by adding the following to the top of the method:

```
AlertCenter.instance.clearAlerts()
```

Because `AppModelTests` indirectly mess with `DatModel` state, they can also trigger alerts that need to be cleared. Back in **`AppModelTests.swift`**, add the following to the top of `tearDown`:

```
AlertCenter.instance.clearAlerts()
```

This ensures the state of `AlertCenter` is reset after each test that modifies it. Back in **`AlertCenter.swift`**, add the following to `AlertCenter`:

```
// MARK: - Alert Handling

func clearAlerts() {
    alertQueue.removeAll()
}
```

This allows you to remove all alerts from `alertQueue`, which can be used to solve your issues with persisted alerts between tests. But first, there is one more place you need to use your new `alertCount`.

Go back to **`RootViewController.swift`** and change the `listenForAlerts` callback block in `viewDidLoad` to:

```
self.alertContainer.isHidden = center.alertCount == 0
```

Now when an alert is triggered, you display `alertContainer` only if more than one alert is currently present. Are you dizzy yet? With TDD, adding functionality requires looping back and forth between the application and tests code.

Finally, you can fix the broken `testWhenLoaded_noAlertsAreShown` by adding to the top of `tearDown` in **`RootViewControllerTests.swift`**:

```
AlertCenter.instance.clearAlerts()
```

Now `alertQueue` will clear after each test, preventing tests that modify the queue from impacting each other.

With the count reset, you just need to clear any existing alerts at the start of each test to avoid the persistence issue you observed in `testWhenLoaded_noAlertsAreShown()`.

Add the following to the bottom of `setUp`:

```
sut.reset()
```

Now all the tests will pass, regardless of execution order.

If you want to see the alert view in practice, temporarily replace `startStopPause(_:)` in **StepCountController.swift** with the following:

```
@IBAction func startStopPause(_ sender: Any?) {  
    let alert = Alert("Test Alert")  
    AlertCenter.instance.postAlert(alert: alert)  
}
```

Now it'll display an alert for any state change. Build and *run*. When the app loads tap **Start**.



For now undo those changes and move on for more expectation testing.

Getting specific about notifications

To make sure the UI is updated effectively, it will be useful to add additional information to the alert notification beyond the name.

In particular, it will be useful to add the associated `Alert` to the notification's `userInfo`.

Open **AlertCenterTests.swift** and add the following to **AlertCenterTests**:

```
// MARK: - Notification Contents

func testNotification_whenPosted_containsAlertObject() {
    // given
    let alert = Alert("test contents")
    let exp = expectation(forNotification: AlertNotification.name,
                          object: sut,
                          handler: nil)

    var postedAlert: Alert?
    sut.notificationCenter.addObserver(
        forName: AlertNotification.name,
        object: sut,
        queue: nil) { notification in
        let info = notification.userInfo
        postedAlert = info?[AlertNotification.Keys.alert] as? Alert
    }

    // when
    sut.postAlert(alert: alert)

    // then
    wait(for: [exp], timeout: 1)
    XCTAssertNotNil(postedAlert, "should have sent an alert")
    XCTAssertEqual(alert,
                    postedAlert,
                    "should have sent the original alert")
}
```

In addition to using a notification expectation, this test also sets up an additional listener for an **AlertNotification**. In the observation closure, the **Alert** that is expected to be in the **userInfo** is stored so it can be compared in the test assert.

Note: While you should strive for a single assert per test, it's OK to have more than one if they both confirm the same truth. In this case, you're trying to validate that the notification contains the same **Alert** object that was posted. Checking that the notification's alert isn't nil is part of that validation, as is comparing it to the posted alert.

To get this test to pass, you have to add the alert object to the notification. In **AlertCenter.swift** change the `let notification = ...` line in `postAlert(alert:)` to:

```
let notification = Notification(
    name: AlertNotification.name,
    object: self,
    userInfo: [AlertNotification.Keys.alert: alert])
```

This adds the posted alert object to the notification so it can be observed in the test's closure. Now run `testNotification_whenPosted_containsAlertObject()` and you should see another green test.

Driving alerts from the data model

In order to drive engagement and give the user a sense of fulfillment as they near their goal, it's important to present messages to the user as they reach certain milestones.

To start off on a positive note, encourage the user by giving them alerts at certain milestones. When they reach 25%, 50%, and 75% of the goal, they should see an encouragement alert, and at 100% a congratulations alert.

There are already some hard coded values for these in an `Alert` extension.

Before writing the next set of tests, create a new helper file. Under the **Test Extensions** group add a new group, **Alerts**. Then add a new Swift file named **Notification+Tests.swift**.

Add the following code to the new file, below the `Foundation` import:

```
@testable import FitNess

extension Notification {
    var alert: Alert? {
        return userInfo?[AlertNotification.Keys.alert] as? Alert
    }
}
```

This helper extension will make it easier to get the `Alert` object out of the notification. You can be fairly confident this works because `testNotification_whenPosted_containsAlertObject()` tested similarly built `userInfo`. You could also go back and update that test to use this new helper. TDD For The Win!

Now you can start writing tests to check that milestone notifications are generated.

In **DataModelTests.swift** add the following test to the end of `DataModelTests`:

```
// MARK: - Alerts

func testWhenStepsHit25Percent_milestoneNotificationGenerated() {
    // given
    sut.goal = 400
    let exp = expectation(forNotification: AlertNotification.name,
                          object: nil) { notification -> Bool in
        return notification.alert == Alert.milestone25Percent
    }
}
```

```
// when
sut.steps = 100

// then
wait(for: [exp], timeout: 1)
}
```

In this test, the optional handler closure is used when setting up the expectation. The closure takes the `Notification` as input and returns a `Bool` indicating whether or not the expectation should be fulfilled. Here you only fulfill the expectation when the alert is a `.milestone25Percent`. With the goal set to 400, setting steps to 100 should trigger that alert and fulfill your expectation.

To make this pass, you'll need to update `DataModel` to trigger the 25 percent alert when appropriate.

First open **`DataModel.swift`**. Next, replace the `steps` var with the following:

```
var steps: Int = 0 {
    didSet {
        updateForSteps()
    }
}
```

Now changes in the step count will trigger `updateForSteps()`, which will post necessary milestone alerts.

Finally, add the following method below `restart()`:

```
// MARK: - Updates due to distance
func updateForSteps() {
    guard let goal = goal else { return }
    if Double(steps) >= Double(goal) * 0.25 {
        AlertCenter.instance.postAlert(alert: Alert.milestone25Percent)
    }
}
```

Now when steps hit 25% of the goal, you post `Alert.milestone25Percent`. Build and test `testWhenStepsHit25Percent_milestoneNotificationGenerated()` and it will pass when the alert is generated.

Previous tests let you know that because the alert is generated it will be shown to the user. You'll have to wait for the next chapter to see the actual step counter in action.

On your own, add three more tests: one each for 50%, 75%, and 100% of completion with a goal of 400:

- 50%: Use `Alert.milestone50Percent` and `steps = 200` for the when condition.
- 75%: Use `Alert.milestone75Percent` and `steps = 300` for the when condition.

- 100%: Use `Alert.goalComplete` and `steps = 400` for the when condition.

Duplicate the `if` statement in `updateForSteps` for each of these conditions to get the tests to pass. With these separate `if` statements, `updateForSteps` will post all alerts up to the current threshold when triggered; you shouldn't address that issue yet. You'll also need to add `AlertCenter.instance.clearAlerts()` to the test's `tearDown` to flush out the alert queue each time.

Testing for multiple expectations

Your new milestone notification tests all seem pretty similar. This is an indicator that you should refactor them to reduce repeated code.

Still in **DataModelTests.swift**, add a new method under `// MARK: – Given`:

```
func givenExpectationForNotification(
    alert: Alert) -> XCTestExpectation {

    let exp = expectation(forNotification: AlertNotification.name,
                          object: nil) { notification -> Bool in
        return notification.alert == alert
    }
    return exp
}
```

This helper method creates an expectation that waits for a notification containing the passed alert. Next, refactor `testWhenStepsHit25Percent_milestoneNotificationGenerated()` to use this helper. Replace the expectation definition with the following:

```
let exp = givenExpectationForNotification(alert: .milestone25Percent)
```

Do the same for the other three milestone tests.

Now you can write a test that checks that all of these alerts are generated, each in order.

Add the following test to **DataModelTests**:

```
func testWhenGoalReached_allMilestoneNotificationsSent() {
    // given
    sut.goal = 400
    let expectations = [
        givenExpectationForNotification(alert: .milestone25Percent),
        givenExpectationForNotification(alert: .milestone50Percent),
        givenExpectationForNotification(alert: .milestone75Percent),
        givenExpectationForNotification(alert: .goalComplete)
    ]

    // when
    sut.steps = 400
}
```

```
// then
wait(for: expectations, timeout: 1, enforceOrder: true)
}
```

So far you've been using `wait(for:timeout:)` with an array of just one expectation. Here you can see why accepting an array is useful. It allows you to provide multiple expectations and wait for all of them to be fulfilled.

Also shown here is the optional `enforceOrder` parameter. This makes sure not only that all the expectations are fulfilled but that those fulfillments happen in the order specified by the input array.

The ordering check allows for sophisticated tests. For example, you could use this when writing a test for a multi-step process like image filtering or a network login that requires multiple API calls (like OAuth or SAML). These tests not only ensure all the steps happen in the necessary order in production code, but also validate that your test code isn't going through a different flow than expected.

Refining Requirements

The previous set of unit tests have one flaw when it comes to validating the app. They test a snapshot of the app's state and do not consider that the app is dynamic.

When in progress, the app will continually update the step count, and it's important to not spam the user at each step, but instead only alert them when a threshold is first crossed. In addition, the user has the option to clear the alerts, so the guard added to `postAlert(alert:)` won't prevent a repeat alert if an earlier alert was cleared by the user.

Always testing first, open **AlertCenterTests.swift** and add this to the bottom of `AlertCenterTests`:

```
// MARK: - Clearing Individual Alerts

func testWhenCleared_alertIsRemoved() {
    // given
    let alert = Alert("to be cleared")
    sut.postAlert(alert: alert)

    // when
    sut.clear(alert: alert)

    // then
    XCTAssertEqual(sut.alertCount, 0)
}
```

This tests that if an alert is added and then cleared, there are no alerts left in the `AlertCenter`.

To pass the test, add the following method to the "Alert Handling" section of **AlertCenter.swift**:

```
func clear(alert: Alert) {
    if let index = alertQueue.firstIndex(of: alert) {
        alertQueue.remove(at: index)
    }
}
```

This removes the passed alert from the alertQueue. Run your tests and they should all pass again.

Next, open **DataModelTests.swift** and add the following:

```
func testWhenStepsIncreased_onlyOneMilestoneNotificationSent() {
    // given
    sut.goal = 10
    let expectations = [
        givenExpectationForNotification(alert: .milestone25Percent),
        givenExpectationForNotification(alert: .milestone50Percent),
        givenExpectationForNotification(alert: .milestone75Percent),
        givenExpectationForNotification(alert: .goalComplete)
    ]

    // clear out the alerts to simulate user interaction
    let alertObserver = AlertCenter.instance.notificationCenter
        .addObserver(forName: AlertNotification.name,
                     object: nil,
                     queue: .main) { notification in
        if let alert = notification.alert {
            AlertCenter.instance.clear(alert: alert)
        }
    }

    // when
    for step in 1...10 {
        self.sut.steps = step
        sleep(1)
    }

    // then
    wait(for: expectations, timeout: 20, enforceOrder: true)
    AlertCenter.instance.notificationCenter
        .removeObserver(alertObserver)
}
```

This is your busiest test yet, and it has a few parts:

- The *given* section sets up a sequence of milestone alert expectations.
- A separate observer watches for alerts and clears them from the AlertCenter. This ensures that repeated notifications don't get ignored because they haven't yet been dismissed by the user.

- The *when* section increments steps to generate the alerts by crossing a series of the milestones individually. Using `sleep` or equivalent in tests should only be done sparingly as this drastically increases the test time. It's necessary here to give time for the notifications to post and be cleared.
- The *then* section uses `wait` to test that the expectations are fulfilled as expected. At the end of the test, you remove `alertObserver` to prevent it from impacting other tests.

Right now the test will pass, which violates the TDD step of writing a failing test first. That's because right now it's not enforcing that there should be a single notification per milestone. That has to be done in the expectation itself.

Still in **DataModelTests.swift**, replace `givenExpectationForNotification(alert:)` with the following:

```
func givenExpectationForNotification(
    alert: Alert) -> XCTestExpectation {

    let exp = XCTNSNotificationExpectation(
        name: AlertNotification.name,
        object: AlertCenter.instance,
        notificationCenter: AlertCenter.instance.notificationCenter)
    exp.handler = { notification -> Bool in
        return notification.alert == alert
    }
    exp.expectedFulfillmentCount = 1
    exp.assertForOverFulfill = true
    return exp
}
```

This ditches the convenience method in order to create an `XCTNSNotificationExpectation`, which is a `XCTestExpectation` with more notification specific features. You set the `expectedFulfillmentCount` and `assertForOverFulfill` which will generate an assertion if the expectation is fulfilled more than the count.

Now the test will fail as a single alert is repeated for multiple steps. To get the test to pass, `DataModel` has to be modified to keep track of sent alerts.

Open **DataModel.swift** and add the following to the top of the class:

```
// MARK: - Alerts
var sentAlerts: [Alert] = []
```

Next, replace `updateForSteps()` with the following:

```
private func checkThreshold(percent: Double, alert: Alert) {
    guard !sentAlerts.contains(alert),
        let goal = goal else {
```

```

        return
    }
    if Double(steps) >= Double(goal) * percent {
        AlertCenter.instance.postAlert(alert: alert)
        sentAlerts.append(alert)
    }
}

func updateForSteps() {
    checkThreshold(percent: 0.25, alert: .milestone25Percent)
    checkThreshold(percent: 0.50, alert: .milestone50Percent)
    checkThreshold(percent: 0.75, alert: .milestone75Percent)
    checkThreshold(percent: 1.00, alert: .goalComplete)
}

```

This cleans up the code a little bit and now checks not just that the threshold was crossed but also that an alert wasn't already sent. This way if a user crosses a threshold and dismisses the alert, they won't see that same alert again.

Finally, add the following to the end of `restart()`:

```
sentAlerts.removeAll()
```

This ensures that a restart clears out your alerts. Build and run, and the tests should all pass!

Using other types of expectations

The bulk of the time you're testing asynchronous processes, you'll use a regular `XCTestExpectation`. `XCTNSNotificationExpectation` covers most other needs. For specific uses, there are two other stock expectations: `XCTKV0Expectation` and `XCTNSPredicateExpectation`.

These look for their eponymous conditions: KVO expectations observe changes to a `keyPath` and predicate expectations wait for their predicate to be true.

There's one place where you've already used KVO for an expectation, and that's with the `ButtonObserver` found in **`StepCountControllerTests.swift`**. You can replace that helper class completely using a KVO based `XCTestExpectation`. Rather than using the more fully featured `XCTKV0Expectation`, you'll use a special `XCTestExpectation` initializer that provides KVO capabilities.

Delete **`ButtonObserver.swift`**. Next, open **`StepCountControllerTests.swift`** and add this method in the *given* section:

```

func expectTextChange() -> XCTestExpectation {
    return keyValueObservingExpectation(

```

```
    for: sut.startButton,  
    keyPath: "titleLabel.text")  
}
```

This helper creates an expectation on `startButton` that observes the `keyPath` `titleLabel.text`. The same `keyPath` was used in the old `ButtonObserver`. This method accepts an optional handler block where you would check the observation to see if it meets the expectation. For these tests, only the first change needs to be observed, so you don't supply the handler to filter fulfillment.

Next, in `testController_whenCaught_buttonLabelIsTryAgain()` and `testController_whenComplete_buttonLabelIsStartOver()` replace the `let exp = ...` and two observer lines with the following:

```
let exp = expectTextChange()
```

And change the `waitForExpectations` lines to:

```
wait(for: [exp], timeout: 1)
```

Build and test and the tests will pass as if nothing happened!

Challenge

This tutorial only scratched the surface of testing asynchronous functions. Here are some things to add to the app with test coverage:

- Add `AlertCenter` tests addressing edge cases for clearing alerts such as clearing an empty queue and clearing the same alert multiple times.
- Create tests for `AlertViewController`. Test that the text used for `alertLabel`'s updates to reflect a new alert, and that it uses the proper color for the given severity. This requires adding the ability to get the first alert out of the `AlertCenter`, and updating tests around that as well.
- It wouldn't be fair to the user if they didn't get a warning of Nessie's progress. Add tests in `DataModelTests` for Nessie catching up to 50% and then to 90%.

Key points

- Use `XCTestExpectation` and its subclasses to make tests wait for asynchronous process completion.
- Test expectations help test properties of the asynchronicity, like order and number of occurrences, but `XCTAssert` functions should still be used to test state.

Where to go from here?

So much app code is asynchronous by nature—disk and network access, UI events, system callbacks, and so on. It's important to understand how to test that code, and this chapter gives you a good start. Many popular 3rd party testing frameworks also have functions that make writing these types of tests easier. For example [Quick+Nimble](#) allows you to write an assert, expectation and wait in one line:

```
expect(alerts).toEventually(contain(alert1, alert2))
```

Alternatively if your app uses a framework like [RxSwift](#) then you can use their `RxBlocking` and `RxTest` frameworks. See [this tutorial](#) for more information.

Chapter 6: Dependency Injection & Mocks

By Michael Katz

So far, you've built and tested a fair amount of the app. There is one gigantic hole that you may have noticed... this "step-counting app" doesn't yet count any steps!

In this chapter, you'll learn how to use mocks to test code that 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, like Core Motion and HealthKit.

Don't have an iPhone handy? Don't worry; you'll dip into functional testing using the Simulator to handle mock data.

What's up with fakes, mocks, and stubs?

When writing tests, it's important to isolate the SUT from other parts of the code so your tests have high confidence that they're testing the system as described. Tests focused on edge cases or error conditions can be very difficult to write, as they often involve specific state external to the SUT. It's also difficult to diagnose and debug tests that fail due to intermittent or inconsistent issues outside the SUT.

The way to isolate the SUT and circumvent these issues is to use **test doubles**: objects that stands in for real code. There are several variants of test doubles:

- **Stub**: Stubs stand in for the original object and provide canned responses. These are often used to implement one method of a protocol and have empty or nil returning implementations for the others.

- **Fake:** Fakes often have logic, but instead of providing real or production data, they provide test data. For example, a fake network manager might read/write from local JSON files instead of connecting over a network.
- **Mock:** Mocks are used to verify behavior, that is they should have an expectation that a certain method of the mock gets called or that its state was set to an expected value. Mocks are generally expected to provide test values or behaviors.
- **Partial mock:** While a regular mock is a complete substitution for a production object, a partial mock uses the production code and only overrides part of it to test the expectations. Partial mocks are usually a subclass or provide a proxy to the production object.

Understanding CMPedometer

There are a few ways of gathering activity data from the user, but the CMPedometer API in Core Motion is by far the easiest.

Using a CMPedometer is easy as:

1. Check that the pedometer is available and the user has granted permission.
2. Start listening for updates.
3. Gather step and distance updates until the user pauses, completes the goal or loses to Nessie.

The pedometer object is supplied a CMPedometerHandler, which has a single callback that receives CMPedometerData (or an error). This data object has the step count and distance travelled.

Here's the thing... you're using TDD so using a CMPedometer is tricky, even if you have the host app run on a physical device. CMPedometer depends on the device state, which is too variable for consistent unit tests.

Give it a try. First, open **PedometerTests.swift** which has been added to the DataModel test case group. Next add the following below `tearDown()`:

```
func testCMPedometer_whenQueries_loadsHistoricalData() {  
    // given  
    var error: Error?  
    var data: CMPedometerData?  
    let exp = expectation(description: "pedometer query returns")  
  
    // when
```

```

let now = Date()
let then = now.addingTimeInterval(-1000)
sut.queryPedometerData(from: then, to: now) {
    pedometerData, pedometerError in
    error = pedometerError
    data = pedometerData
    exp.fulfill()
}

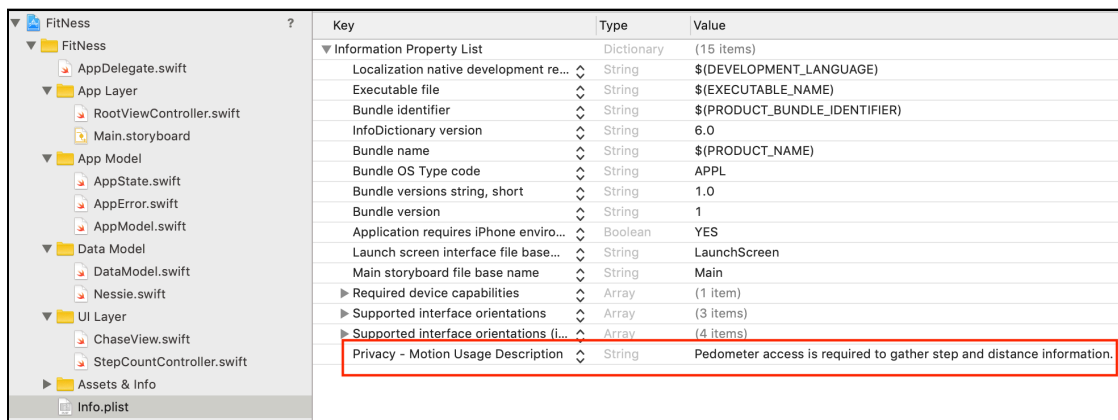
// then
wait(for: [exp], timeout: 1)
XCTAssertNil(error)
XCTAssertNotNil(data)
if let steps = data?.numberOfSteps {
    XCTAssertGreaterThan(steps.intValue, 0)
} else {
    XCTFail("no step data")
}
}

```

This test creates an expectation for a returned pedometer query, calls `queryPedometerData(from:to:)` to query the data and fulfill the expectation. It then asserts that the data contains at least one step.

Although this test compiles, it crashes on launch. Apple requires permission to use Core Motion. Strike #1 against using a real `CMPedometer` object in the tests. In order to ask for permission, a usage description is required. Open the app's **Info.plist**.

Add a new row, use the key **Privacy - Motion Usage Description** and set the value to "Pedometer access is required to gather step and distance information."



Build and test, and it may fail depending on if you run the app on device or Simulator, and if you've accepted the permission pop-up or not. The unpredictability caused by lack of control over `CMPedometer` makes this a pretty poor test. This sounds like a job for a mock!

Delete the **PedometerTests.swift** test file; you're about do much better.

Mocking

Restating the problem

Open **AppModelTests.swift**, and add the following test beneath the "Pedometer" mark:

```
func testAppModel_whenStarted_startsPedometer() {
    //given
    givenGoalSet()
    let exp = expectation(for: NSPredicate(block:
    { thing, _ -> Bool in
        return (thing as! AppModel).pedometerStarted
    }), evaluatedWith: sut, handler: nil)

    // when
    try! sut.start()

    // then
    wait(for: [exp], timeout: 1)
    XCTAssertTrue(sut.pedometerStarted)
}
```

This test intends to verify that starting the app model will also start the pedometer. If you read the previous chapter, you'll recognize the elusive XCTNSPredicateExpectation used to wait for the status change.

This test is subtly different from the previous one: It doesn't test the pedometer object directly. Instead, the test verifies the behavior of the SUT by measuring the effect on the pedometer (as exposed through pedometerStarted).

To get this compiling, you'll need to modify AppModel. Open **AppModel.swift**, add the following two vars:

```
let pedometer = CMPedometer()
private(set) var pedometerStarted = false
```

This adds a little state to keep track of the pedometer.

Next, add the following to the bottom of start():

```
startPedometer()
```

Finally, add the following extension to the bottom of the file:

```
// MARK: - Pedometer
extension AppModel {
    func startPedometer() {
        pedometer.startEventUpdates { event, error in
            if error == nil {
                self.pedometerStarted = true
            }
        }
    }
}
```

```

    }
  }
}

```

This uses the pedometer event handler callback to determine if the pedometer has started. With a `CMPedometer`, you can't write a simple test to check if it's started as that state isn't exposed in the API. However, this callback will be called soon after starting event updates. If step counting is available, then there won't be an error, and you'll know it's started.

Build and test, and this will pass if you run it on a device and have granted permission to motion data. If you run on Simulator or device without this permission granted, it'll fail.

Mocking the pedometer

To move past this impasse, it's time to create the mock pedometer. In order to swap `CMPedometer` for its mock object, you'll first need to separate the pedometer's **interface** from its **implementation**.

To do that, you'll make use of two classic patterns: **Facade** and **Bridge**.

First, create a new group in the app, named **Pedometer**. In that group, create a new Swift file, **Pedometer.swift**.

For now, just add the following code:

```

protocol Pedometer {
    func start()
}

```

This is the start of the Bridge protocol that will allow you to substitute any pedometer implementation for the real one.

In order to do that, you'll have to declare conformance for `CMPedometer`. Create another Swift file in the group: **CMPedometer+Pedometer.swift** and replace its contents with the following:

```

import CoreMotion

extension CMPedometer: Pedometer {
    func start() {
        startEventUpdates { event, error in
            // do nothing here for now
        }
    }
}

```

This declares conformance to the new protocol and migrates the start behavior you implemented in `startPedometer`. It doesn't do anything much yet, but will soon.

Next, open **AppModel.swift** and decouple `AppModel` from the specific implementation of `CMPedometer`:

1. Change the pedometer declaration to: `let pedometer: Pedometer`.
2. Remove the `pedometerStarted` property.
3. Add the following initializer:

```
init(pedometer: Pedometer = CMPedometer()) {  
    self.pedometer = pedometer  
}
```

4. Change `startPedometer` to:

```
func startPedometer() {  
    pedometer.start()  
}
```

The optional `init` parameter is where you'll be able to replace the default `CMPedometer` with the mock object. The reduction of code in `startPedometer` is the advantage of using a **Facade**: You can hide the specific complexity of the `CMPedometer` behind a simplified interface.

Now, it's time to create the mock!

Create a new Swift file in the **Mocks** group in **FitNessTests** named **MockPedometer.swift** and replace its contents with the following:

```
import CoreMotion  
@testable import FitNess  
  
class MockPedometer: Pedometer {  
    private(set) var started: Bool = false  
  
    func start() {  
        started = true  
    }  
}
```

This creates a very different implementation of `Pedometer`. Its `start` method instead of making `CoreMotion` calls just sets a `Bool` that can be checked in a test. Here's another value of mocking — you can spy or inspect the mock to check that the right methods were called or that its state was set appropriately.

Now, go back to **AppModelTests.swift** and add the following property up top and update `setUp`:

```
var mockPedometer: MockPedometer!

override func setUp() {
    super.setUp()
    mockPedometer = MockPedometer()
    sut = AppModel(pedometer: mockPedometer)
}
```

This creates a mock pedometer and uses it when creating the sut.

Now, go back to `testAppModel_whenStarted_startsPedometer` and replace it with the following:

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

    // when
    try! sut.start()

    // then
    XCTAssertTrue(mockPedometer.started)
}
```

This simplified test now tests the side effect of `start` on the mock object. In addition to being a simpler test, it's guaranteed to pass regardless of the device state. Build and test, and you'll see that it passes.

Handling error conditions

Mocks make it easy to test error conditions. If you've been following along so far using both Simulator and a device, you may have encountered one or both of these error states:

- Step counting is not available on a device, such as the Simulator.
- The user may deny permission for motion recording on device.

Dealing with no pedometer

To handle the first case, you'll have to add functionality to detect that the pedometer is not available and to inform the user.

First, add this test in `AppModelTests` under the "Pedometer" mark:

```
func testPedometerNotAvailable_whenStarted_doesNotStart() {  
    // given  
    givenGoalSet()  
    mockPedometer.pedometerAvailable = false  
  
    // when  
    try! sut.start()  
  
    // then  
    XCTAssertEqual(sut.appState, .notStarted)  
}
```

This simple check just makes sure the app state doesn't proceed to `inProgress` when the pedometer isn't available.

Next, open **Pedometer.swift** and add the following to the protocol definition:

```
var pedometerAvailable: Bool { get }
```

This creates a var to read the availability state.

Next, open **MockPedometer.swift** and update `MockPedometer` by adding the following:

```
var pedometerAvailable: Bool = true
```

And for the real implementation — to be used by your app code — open **CMPPedometer+Pedometer.swift** and add the following:

```
var pedometerAvailable: Bool {  
    return CMPPedometer.isStepCountingAvailable() &&  
        CMPPedometer.isDistanceAvailable() &&  
        CMPPedometer.authorizationStatus() != .restricted  
}
```

You can see that the "real" implementation is a lot more interesting, but not controllable.

Now the test compiles, and it's time to get it to pass.

Open **AppModel.swift**, find `start()` and add the following before `appState = .inProgress`:

```
guard pedometer.pedometerAvailable else {  
    AlertCenter.instance.postAlert(alert: .noPedometer)  
    return  
}
```

Unlike the other guard statement, this condition doesn't raise an exception; instead, it uses the new `AlertCenter` way of communicating with the user. The resulting error handling, where `start()` is called, will be a little different, and refactoring it is out of scope of this chapter.

Build and test, and it will pass now, as the new guard prevents the `appState` from progressing to `InProgress` when the pedometer isn't available. Note that, if you run the entire suite, some other tests will now fail — you'll circle back to those in a moment.

It's a good idea to test the alert, as well.

Open, **AppModelTests.swift** and add the following below `testPedometerNotAvailable_whenStarted_doesNotStart()`:

```
func testPedometerNotAvailable_whenStarted_generatesAlert() {
    // given
    givenGoalSet()
    mockPedometer.pedometerAvailable = false
    let exp = expectation(forNotification: AlertNotification.name,
                          object: nil,
                          handler: alertHandler(.noPedometer))

    // when
    try! sut.start()

    // then
    wait(for: [exp], timeout: 1)
}
```

This sets `pedometerAvailable` to `false` and waits for the corresponding alert. The test will pass out of the gate due to the code previously added to `AppModel` for displaying this alert.

Injecting dependencies

Re-run all the tests, and you will see failures in `StepCountControllerTests`. That's because this new `pedometerAvailable` guard in `AppModel` is still dependent on the production `CMPedometer` in other tests.

One way to fix that this to make the pedometer into a variable so it can be modified for testing.

Open **AppModel.swift** and change the `let` to a `var`:

```
var pedometer: Pedometer
```


Next, open **ViewControllers.swift** and add the following to the top of `loadRootViewController()`:

```
AppModel.instance.pedometer = MockPedometer()
```

This sets the mock pedometer when the root view controller is fetched for tests, which means any view controller test will get a mock pedometer.

Build and run all the tests, and they will now pass.

Dealing with no permission

The other error state that needs to be handled is when the user declines the permission pop-up.

Open, **AppModelTests.swift** and add the following to the end of the class:

```
func testPedometerNotAuthorized_whenStarted_doesNotStart() {
    // given
    givenGoalSet()
    mockPedometer.permissionDeclined = true

    // when
    try! sut.start()

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

func testPedometerNotAuthorized_whenStarted_generatesAlert() {
    // given
    givenGoalSet()
    mockPedometer.permissionDeclined = true
    let exp = expectation(forNotification: AlertNotification.name,
                           object: nil,
                           handler: alertHandler(.notAuthorized))

    // when
    try! sut.start()

    // then
    wait(for: [exp], timeout: 1)
}
```

These test handling of a `permissionDeclined` error. The first test checks that the app state stays in `.notStarted` and the second checks for a user alert.

To get them to work, you need to add `permissionDeclined` in a few places:

First, open **Pedometer.swift**, and add the following to the protocol definition:

```
var permissionDeclined: Bool { get }
```

Next, open **MockPedometer.swift** and add the following to the mock implementation:

```
var permissionDeclined: Bool = false
```

Next, open **CMPedometer+Pedometer.swift** and add the following to the real implementation:

```
var permissionDeclined: Bool {  
    return CMPedometer.authorizationStatus() == .denied  
}
```

Finally, open **AppModel.swift**, and add another guard statement to start:

```
guard !pedometer.permissionDeclined else {  
    AlertCenter.instance.postAlert(alert: .notAuthorized)  
    return  
}
```

With permissionDeclined handled, the tests will now pass.

Mocking a callback

There is another important error situation to handle. This occurs the very first time the user taps **Start** on a pedometer-capable device. In that case, the start flow goes ahead, but the user can decline in the permission pop-up. If the user declines, there is an error in the eventUpdates callback.

Lets test that condition. Open **AppModelTests.swift** and add the following to the end of the class definition:

```
func testAppModel_whenDeniedAuthAfterStart_generatesAlert() {  
    // given  
    givenGoalSet()  
    mockPedometer.error = MockPedometer.notAuthorizedError  
    let exp = expectation(forNotification: AlertNotification.name,  
                          object: nil,  
                          handler: alertHandler(.notAuthorized))  
  
    // when  
    try! sut.start()  
  
    // then  
    wait(for: [exp], timeout: 1)  
}
```

Unlike the previous tests, this doesn't explicitly set permissionDeclined, so the model can attempt to start the pedometer. Instead, the test relies on passing an error to the mock to generate the alert while the pedometer is starting.

The next step is to build a way to get that error back to the SUT.

Open **Pedometer.swift**, change the definition of `start()` to the following:

```
func start(completion: @escaping (Error?) -> Void)
```

This allows for a completion callback for error handling.

Next, update **CM_Pedometer+Pedometer.swift** by replacing `start` with:

```
func start(completion: @escaping (Error?) -> Void) {  
    startEventUpdates { event, error in  
        completion(error)  
    }  
}
```

This forwards the error on to the completion.

Next add the error handling in **AppModel.swift**, by replacing `startPedometer` with the following:

```
func startPedometer() {  
    pedometer.start { error in  
        if let error = error {  
            let alert = error.is(CMErrorMotionActivityNotAuthorized)  
                ? .notAuthorized : Alert(error.localizedDescription)  
            AlertCenter.instance.postAlert(alert: alert)  
        }  
    }  
}
```

The closure checks if an error was returned when starting the pedometer. If it's a `CMErrorMotionActivityNotAuthorized`, then it posts a `notAuthorized` alert; otherwise, a generic alert with the error's message is posted.

This takes care of the production code, but you also need to update the `MockPedometer`.

Open **MockPedometer.swift** and replace `start()` with the following:

```
var error: Error?  
  
func start(completion: @escaping (Error?) -> Void) {  
    started = true  
    DispatchQueue.global(qos: .default).async {  
        completion(self.error)  
    }  
}  
  
static let notAuthorizedError =  
    NSError(domain: CMErrorDomain,  
            code: Int(CMErrorMotionActivityNotAuthorized.rawValue),  
            userInfo: nil)
```

This update will call the completion, passing its error property. For convenience, the static `notAuthorizedError` creates an error object that matches what is returned by Core Motion when unauthorized. This is what you used in `testAppModel_whenDeniedAuthAfterStart_generatesAlert`.

Build and test again, and your tests should pass.

Getting actual data

It's time move on to handling data updates. The incoming data is the most important part of the app, and it's crucial to have it properly mocked. The actual step and distance count are provided by `CMPedometer` through the aptly named `CMPedometerData` object. This too should be abstracted between the app and Core Motion.

Open **Pedometer.swift** and add the following protocol:

```
protocol PedometerData {
    var steps: Int { get }
    var distanceTravelled: Double { get }
}
```

This adds an abstraction around `CMPedometerData` so that the step and distance data can be mocked. Do that by creating a new `.swift` file in the **Mocks** group of the test target: **MockData.swift** and replacing its contents with the following:

```
@testable import FitNess

struct MockData: PedometerData {
    let steps: Int
    let distanceTravelled: Double
}
```

With this in place, open **AppModelTests.swift** and add the following test at the end of the class definition:

```
func testModel_whenPedometerUpdates_updatesDataModel() {
    // given
    givenInProgress()
    let data = MockData(steps: 100, distanceTravelled: 10)

    // when
    mockPedometer.sendData(data)

    // then
    XCTAssertEqual(sut.dataModel.steps, 100)
    XCTAssertEqual(sut.dataModel.distance, 10)
}
```

The test verifies that the supplied data is applied to the data model. This requires an update to `MockPedometer` to pass the data. First, think about how that data will eventually be passed to `AppModel`.

Open **Pedometer.swift**. In the `Pedometer` protocol, change the signature of `start(completion:)` to the following:

```
func start(
    dataUpdates: @escaping (PedometerData?, Error?) -> Void,
    eventUpdates: @escaping (Error?) -> Void)
```

The `dataUpdates` block will provide a means of returning `PedometerData` from the pedometer. `eventUpdates` will return events, as the old `completion` block did.

In `MockPedometer`, create two new variables to hold these callback blocks:

```
var updateBlock: ((Error?) -> Void)?
var dataBlock: ((PedometerData?, Error?) -> Void)?
```

Next, replace `start(completion:)` with the following:

```
func start(
    dataUpdates: @escaping (PedometerData?, Error?) -> Void,
    eventUpdates: @escaping (Error?) -> Void) {

    started = true
    updateBlock = eventUpdates
    dataBlock = dataUpdates
    DispatchQueue.global(qos: .default).async {
        self.updateBlock?(self.error)
    }
}

func sendData(_ data: PedometerData?) {
    dataBlock?(data, error)
}
```

The two blocks are saved for later use, but the `updateBlock` is still called as part of this method, as `completion` was previously. You won't have to update any previous tests for this one, as the behavior is the same. Also added is `sendData(_:)`, which is used by the test to call the `dataBlock` with the mock data.

You also need to update the `CMPedometer` extension for this new logic. Open **CMPedometer+Pedometer.swift** and change `start(completion:)` to the following:

```
func start(
    dataUpdates: @escaping (PedometerData?, Error?) -> Void,
    eventUpdates: @escaping (Error?) -> Void) {

    startEventUpdates { event, error in
        eventUpdates(error)
    }
```

```

    }

    startUpdates(from: Date()) { data, error in
        dataUpdates(data, error)
    }
}

```

This preserves the previous `startEventUpdates` behavior, plus adds a new call to `startUpdates` to forward the data updates.

You also need to wrap `CMPedometerData` with the new `PedometerData` protocol. Add the following extension to bottom of the file:

```

extension CMPedometerData: PedometerData {

    var steps: Int {
        return numberOfSteps.intValue
    }

    var distanceTravelled: Double {
        return distance?.doubleValue ?? 0
    }
}

```

This forwards the `CMPedometerData` values as `PedometerData` variables.

Finally, open **AppModel.swift**, and replace `startPedometer()` with the following:

```

func startPedometer() {
    pedometer.start(dataUpdates: handleData,
                    eventUpdates: handleEvents)
}

func handleData(data: PedometerData?, error: Error?) {
    if let data = data {
        dataModel.steps += data.steps
        dataModel.distance += data.distanceTravelled
    }
}

func handleEvents(error: Error?) {
    if let error = error {
        let alert = error.is(CMErrorMotionActivityNotAuthorized)
            ? .notAuthorized : Alert(error.localizedDescription)
        AlertCenter.instance.postAlert(alert: alert)
    }
}

```

This moves the previous event handling to its own method and creates a new one to update `dataModel` when there is new data. You'll notice that data update errors are not handled here. That's left as a Challenge for you after this chapter is complete!

Build and test, and watch that green grow!

Making a functional fake

At this point it sure would be nice to see the app in action. The unit tests are useful for verifying *logic* but are bad at verifying you're building a good *user experience*. One way to do that is to build and run on a device, but that will require you to walk around to complete the goal. That's very time and calorie consuming. There has got to be a better way!

Enter the fake pedometer: You've already done the work to abstract the app from a real `CMPedometer`, so it's straightforward to build a fake pedometer that speeds up time or makes up movement.

Create a new `.swift` file in the pedometer group: **SimulatorPedometer.swift**. Replace its contents with the following:

```
import Foundation

class SimulatorPedometer: Pedometer {

    struct Data: PedometerData {
        let steps: Int
        let distanceTravelled: Double
    }

    var pedometerAvailable: Bool = true
    var permissionDeclined: Bool = false

    var timer: Timer?
    var distance = 0.0

    var updateBlock: ((Error?) -> Void)?
    var dataBlock: ((PedometerData?, Error?) -> Void)?

    func start(
        dataUpdates: @escaping (PedometerData?, Error?) -> Void,
        eventUpdates: @escaping (Error?) -> Void) {

        updateBlock = eventUpdates
        dataBlock = dataUpdates

        timer = Timer(timeInterval: 1, repeats: true,
                      block: { timer in
                        self.distance += 1
                        print("updated distance: \(self.distance)")
                        let data = Data(steps: 10,
                                       distanceTravelled: self.distance)
                        self.dataBlock?(data, nil)
                    })
        RunLoop.main.add(timer!, forMode: .defaultRunLoopMode)
        updateBlock?(nil)
    }
}
```

```
func stop() {
    timer?.invalidate()
    updateBlock?(nil)
    updateBlock = nil
    dataBlock = nil
}
```

This giant block of code implements the `Pedometer` and `PedometerData` protocols. It sets up a `Timer` object that, once `start` is called, adds ten steps every second. Each time it updates, it calls `dataBlock` with the new data.

You've also added a `stop` method that stops the timer and cleans up. This will be used when you add the ability to pause the pedometer by tapping the **Pause** button.

To use the simulated pedometer in the app, open **AppModel.swift**, and add the following static var:

```
static var pedometerFactory: (() -> Pedometer) = {
    #if targetEnvironment(simulator)
    return SimulatorPedometer()
    #else
    return CMPedometer()
    #endif
}
```

This method creates either a `SimulatorPedometer()` or a `CMPedometer()` depending on the app's target environment.

Next, replace `init` with the following:

```
init(pedometer: Pedometer = pedometerFactory()) {
    self.pedometer = pedometer
}
```

Now build and run in Simulator. Tap the settings **cog** in the lower-right and enter a goal of 100 steps.

Tap **Start**, and you'll see alert notifications coming in!



Wiring up the chase view

Looking at the app now, that white box in the middle is a little disappointing. This is the **chase view** (it illustrates Nessie's chase of the user), and hasn't yet been wired up.

In order to test that it will accurately reflect the user's state, you can use a **partial mock**. By partially mocking the chase view, you can add a little extra test functionality without interrupting its main logic. This is instead of a full mock, which replaces all functionality.

Create a new file in the **Mocks** group called **ChaseViewPartialMock.swift** and replace its contents with the following:

```
@testable import FitNess

class ChaseViewPartialMock: ChaseView {
    var updateStateCalled = false
    var lastRunner: Double?
    var lastNessie: Double?

    override func updateState(runner: Double, nessie: Double) {
        updateStateCalled = true
        lastRunner = runner
        lastNessie = nessie
        super.updateState(runner: runner, nessie: nessie)
    }
}
```

```
}  
}
```

This partial mock overrides `updateState(runner:nessie:)` so that the values sent to it can be recorded and verified in tests. `updateStateCalled` can be used by tests to track that the method has been called — a common mock validation.

This class is used by `StepCountController`.

First open **StepCountControllerTests.swift** and add the following variable:

```
var mockChaseView: ChaseViewPartialMock!
```

Next, add the following lines to the bottom of `setUp()`:

```
mockChaseView = ChaseViewPartialMock()  
sut.chaseView = mockChaseView
```

Finally, add a test that verifies that the view gets updated:

```
func testChaseView_whenDataSent_isUpdated() {  
    // given  
    givenInProgress()  
  
    // when  
    let data = MockData(steps:500, distanceTravelled:10)  
    (AppModel.instance.pedometer as! MockPedometer).sendData(data)  
  
    // then  
    XCTAssertTrue(mockChaseView.updateStateCalled)  
    XCTAssertEqual(mockChaseView.lastRunner, 0.5)  
}
```

This uses the mocked pedometer to send data and verifies the state on the partial mock chase view. The value for Nessie's position isn't checked since the code for Nessie isn't part of the project yet.

Build and test, and you'll see neither assert passes, because the chase view isn't yet being updated.

Open, **StepCountController.swift**, and add the following to `viewDidLoad()` to kick off this update:

```
NotificationCenter.default  
    .addObserver(forName: DataModel.UpdateNotification,  
                 object: nil,  
                 queue: nil) { _ in  
    self.updateUI()  
}
```

This listens for data model updates and calls `updateUI` when there is a data update.

`updateUI` calls `updateChaseView`, which needs to calculate the location of Nessie and the runner, then update them in the view. Replace `updateChaseView` with the following:

```
private func updateChaseView() {
    chaseView.state = AppModel.instance.appState
    let dataModel = AppModel.instance.dataModel
    let runner =
        Double(dataModel.steps) / Double(dataModel.goal ?? 10_000)
    let nessie = dataModel.nessie.distance > 0 ?
        dataModel.distance / dataModel.nessie.distance : 0
    chaseView.updateState(runner: runner, nessie: nessie)
}
```

This gathers the distance of the user and Nessie from the data model, computes a percent completion, and presents it to the chase view so that the avatars can be placed accordingly.

Build and test to see the test pass! Build and run to see the view in action:



Time dependencies

The final major piece missing is Nessie. She should be chasing after the user while the app is in progress. Her progress will be measured at a constant velocity. Measuring something over time? Sounds like a `Timer` is the answer.

Timers are notoriously hard to test: They require using expectations along with having a potentially long wait. There are few common solutions:

1. During tests, use a *very* short timer (e.g., one millisecond instead of one second).
2. Swap the timer for a mock that executes the callback immediately.
3. Use the callback directly, and save the timing for app or user-acceptance testing.

Any of these are reasonable solutions, but you're going to go with option #3. In **NessieTests.swift**, add this test:

```
func testNessie_whenUpdated_incrementsDistance() {  
    // when  
    sut.incrementDistance()  
  
    // then  
    XCTAssertEqual(sut.distance, sut.velocity)  
}
```

This calls `incrementDistance` directly, just as the `Timer` callback does in the `Nessie` class. It asserts that after distance increments it is equal to the `velocity`.

The test doesn't yet pass, because `incrementDistance` is stubbed out. Open **Nessie.swift**, and add the following line to `incrementDistance()`:

```
distance += velocity
```

The distance now increments, and the test will pass.

Challenge

You've reached the end of the chapter, but not the end of the app. You should be able to take the testing tools you've learned and finish the app. Your challenge is to add the following tests and features to complete the app:

- Complete the **Pause** functionality to be able to pause and resume the pedometer.
- Wire up Nessie to app state so it can start, pause and reset appropriately. You'll also have to give the user a little bit of a head start since both the user and Nessie will start at 0.
- Complete the handling of data errors from the pedometer (use the Alert Center).

Key points

- **Test doubles** let you test code in isolation from other systems, especially those that are part of system SDKs, rely on networking or timers.
- **Mocks** let you swap in a test implementation of a class, and **partial mocks** let you just substitute part of a class.
- **Fakes** let you supply data for testing or use in Simulator.

Where to go from here?

That's it. Over the past few chapters, you've built an app from the ground up following TDD principles.

This chapter covered using mocks to separate the test subjects from external code and events. This just scratches the surface of what's possible. The next section will be all about using external services like network requests.

If you want to learn more about the use and history of doubles, read this excellent Martin Fowler article, "Mocks Aren't Stubs": <https://martinfowler.com/articles/mocksArentStubs.html>.

Section III: TDD with Networking

This section will teach you test-driven development with networking. You'll learn how to do TDD for RESTful network calls, an authentication client, authenticated calls and web sockets.

Throughout this section, you'll build onto a puppy-buying app that interacts with a backend service to give you hands-on experience doing TDD with networking!

- **Chapter 7: Introducing DogPatch:** You'll complete a puppy-adoption app called Dog Patch throughout this section. This app connects dog lovers with kind, professional breeders to find the puppy of their dreams.
- **Chapter 8: RESTful Networking:** You'll learn how to start TDD for RESTful networking in this chapter. By the end of this chapter, you'll have created a networking client and will be able to make unauthenticated calls.
- **Chapter 9: Authentication Client:** You'll use TDD to create an authentication client in this chapter that will trigger sign-up and sign-in flows.
- **Chapter 10: Authenticated Network Calls:** You'll TDD connecting the authentication and networking clients together that you created in the previous chapters to make authenticated networking calls in this chapter.
- **Chapter 11: WebSockets:** You'll learn about websockets and how you can use TDD to create a real-time chat component in this chapter.

Chapter 7: Introducing Dog Patch

By Joshua Greene

You've learned the basics of TDD, and you should be starting to feel comfortable with it. However, you haven't learned how to do TDD for a very critical part of most apps: networking!

Over the next several chapters, you'll learn the ins-and-outs of writing networking code in a test-driven fashion. The goal of this chapter is to introduce you to this section's sample project and highlight what work remains to be completed.

Getting started

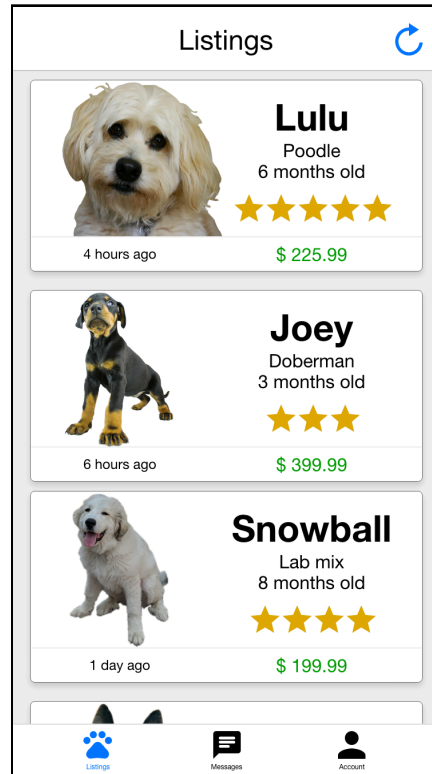
You'll complete a puppy-adoption app called **Dog Patch** throughout this section. This app connects dog lovers with kind, professional breeders to find the puppy of their dreams.

A prospective owner first browses available puppy listings within the app. When they find an adorable match, they can contact and chat directly with the breeder for more information. The puppy purchase and delivery happens outside the app.

Let's go over what needs to be done to make this possible.

Making unauthenticated network calls

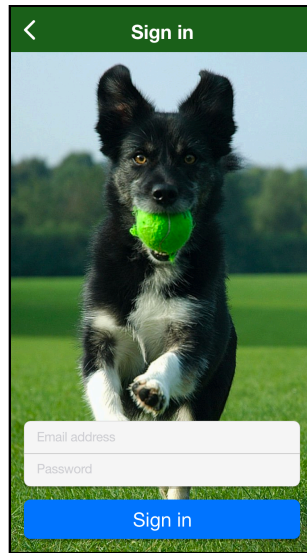
In Chapter 8, you'll do TDD for making unauthenticated network calls. Specifically, you'll complete `ListingsViewController`, which displays available puppy listings:



You'll make a GET request to fetch `SaleListing` models from a remote server, and you'll then convert these to `SaleViewModels`. `ListingViewController` already has logic to display these directly on screen.

Creating an authentication client

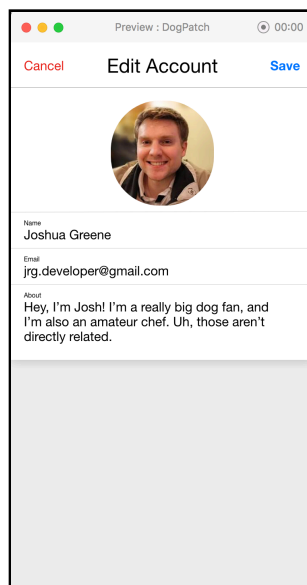
In Chapter 9, you'll follow TDD to create an `AuthenticationClient`. This will allow existing users to sign into and new users to register on the app. You'll complete both `SignInViewController` and `RegisterViewController` as part of this:



You'll use two forms of authentication in Chapter 9 as well. You'll first pass an email and password via basic authentication to sign in. In response, you'll receive a JSON Web Token (JWT), which you'll persist to later use as bearer authentication.

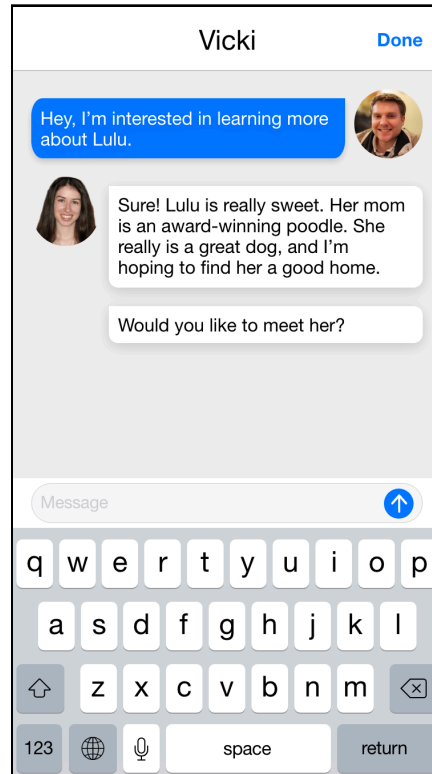
Making authenticated network calls

In Chapter 10, you'll use your previously obtained JWT to make authenticated network calls. Specifically, you'll follow TDD to write a PUT request to update the User, including the profile image via a multipart form requesting. You'll ultimately complete the `EditAccountViewController` in this chapter:



Making calls over web sockets

In Chapter 11, you'll use TDD to send messages over web sockets. You'll use this to complete both `ChatListViewController` and `ChatsViewController`:



Understanding Dog Patch's architecture

You'll use **Model-View-Controller-Networking (MVC-N)** for this app's architecture. If you've done any work in iOS before, you're very likely familiar with the **Model-View-Controller (MVC)** architecture, wherein you separate objects into three types. These are aptly named **models**, **views** and **controllers**, of course.

MVC-N is a spin-off architecture of MVC. Instead of just three types, however, it separates out a fourth type for **networking**.

Especially for networking-heavy apps like Dog Patch, it makes sense to separate networking into its own type. If you didn't do this, where would the networking code go, after all? In pure MVC-architecture apps, developers tend to lump networking into each view controller.

The problem here is that *a lot* of networking code is interrelated. For example, URL and content serialization, authentication headers and more require *exactly the same* logic. If networking code is directly in each view controller, you quickly wind up with a lot of duplication. This quickly becomes an unmanageable mess as a result.

Fortunately, MVC-N allows you to avoid this issue altogether by putting your networking code into a **networking client**. This client is then passed into whatever view controllers need it, and this effectively eliminates the duplication across view controllers.

It's OK if you haven't heard of MVC-N before. You'll learn all about it over the course of the next few chapters!

Where to go from here?

This chapter introduced Dog Patch and what you'll be doing over the next few chapters. Continue onto the next chapter to dive into the code!

Chapter 8: RESTful Networking

You'll learn how to start TDD for RESTful networking in this chapter. By the end of this chapter, you'll have created a networking client and will be able to make unauthenticated calls.

This is an early access release of this book. Stay tuned for this chapter in a future release!

Chapter 9: Authentication Client

You'll use TDD to create an authentication client in this chapter that will trigger sign-up and sign-in flows.

This is an early access release of this book. Stay tuned for this chapter in a future release!

Chapter 10: Authenticated Network Calls

You'll TDD connecting the authentication and networking clients together that you created in the previous chapters to make authenticated networking calls in this chapter.

This is an early access release of this book. Stay tuned for this chapter in a future release!

Chapter 11: Websockets

You'll learn about websockets and how you can use TDD to create a real-time chat component in this chapter.

This is an early access release of this book. Stay tuned for this chapter in a future release!

Section IV: TDD in Legacy Apps

This section will show you how to start test-driven development in a legacy app that lacks sufficient unit tests. You'll learn strategies for introducing TDD into existing apps, methods for visualizing and splitting up dependencies, ways to add features safely alongside existing code and how to refactor large classes.

Throughout this section, you'll introduce TDD into an app for managing a business. The app is feature-rich with spaghetti code and ready for a TDD clean up!

Several techniques and concepts in this section were inspired by Michael Feather's book *Working Effectively with Legacy Code*. Reading that book isn't a strict requirement for working through these chapters. However, you'll likely benefit by having some familiarity with the topics herein if you already have read it!

- **Chapter 12: 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 to start tackling these problems.
- **Chapter 13: 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.
- **Chapter 14: 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.

- **Chapter 15: Adding Features to Existing Classes:** You won't always have the time, or it may simply not be possible, to break 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!
- **Chapter 16: Refactoring Large Classes:** As you continue to modify your legacy code base using TDD, you'll naturally create "test islands" throughout your code that are easier to test and change. Classes that were once "too big" to break up will start to become possible now. In this chapter, you'll learn strategies specifically for breaking up these very large classes.

Chapter 12: 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 to start tackling these problems.

This is an early access release of this book. Stay tuned for this chapter in a future release!

Chapter 13: 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.

This is an early access release of this book. Stay tuned for this chapter in a future release!

Chapter 14: 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.

This is an early access release of this book. Stay tuned for this chapter in a future release!

Chapter 15: Adding Features to Existing Classes

You won't always have the time, or it may simply not be possible, to break 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!

This is an early access release of this book. Stay tuned for this chapter in a future release!

Chapter 16: Refactoring Large Classes

As you continue to modify your legacy code base using TDD, you'll naturally create "test islands" throughout your code that are easier to test and change. Classes that were once "too big" to break up will start to become possible now. In this chapter, you'll learn strategies specifically for breaking up these very large classes.

This is an early access release of this book. Stay tuned for this chapter in a future release!

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 unweildng 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.