



Modularizing Legacy Projects Using TDD

Test-Driven Development with
XCTest for iOS

—
Khaled El-Morabea
Hassaan El-Garem

Apress®

Modularizing Legacy Projects Using TDD

Test-Driven Development
with XCTest for iOS

Khaled El-Morabea
Hassaan El-Garem

Apress®

Modularizing Legacy Projects Using TDD: Test-Driven Development with XCTest for iOS

Khaled El-Morabea
Giza, Egypt

Hassaan El-Garem
Cairo, Egypt

ISBN-13 (pbk): 978-1-4842-7427-9
<https://doi.org/10.1007/978-1-4842-7428-6>

ISBN-13 (electronic): 978-1-4842-7428-6

Copyright © 2021 by Khaled El-Morabea and Hassaan El-Garem

This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.

Trademarked names, logos, and images may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, logo, or image we use the names, logos, and images only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

The use in this publication of trade names, trademarks, service marks, and similar terms, even if they are not identified as such, is not to be taken as an expression of opinion as to whether or not they are subject to proprietary rights.

While the advice and information in this book are believed to be true and accurate at the date of publication, neither the authors nor the editors nor the publisher can accept any legal responsibility for any errors or omissions that may be made. The publisher makes no warranty, express or implied, with respect to the material contained herein.

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Aaron Black
Development Editor: James Markham
Coordinating Editor: Jessica Vakili

Distributed to the book trade worldwide by Springer Science+Business Media New York, 1 NY Plaza, New York, NY 10014. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail booktranslations@springernature.com; for reprint, paperback, or audio rights, please e-mail bookpermissions@springernature.com.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at <http://www.apress.com/bulk-sales>.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/978-1-4842-7427-9. For more detailed information, please visit <http://www.apress.com/source-code>.

Printed on acid-free paper

When I started writing this book, it was in the middle of the pandemic, and we had a new arrival to our family, Noah. It was a tough period. Imagine raising a new baby during the pandemic and with all these restrictions to stay at home and you need to focus on writing your first book. So I would like to dedicate this book to my wife, Yasmina—without her help and support, it wouldn't have been possible. And to my parents, Huda and Mohamed—without their continuous support and love, I wouldn't be where I am now.

—Khaled El-Morabea

To my sister, Rana, for pushing me to take on this challenging yet fulfilling project. And to my parents, Sahar and Saleh, for their unending love and their much-needed emotional support. And to Aya, without her love and support, this book would have never seen the light of day.

—Hassaan El-Garem

Table of Contents

- About the Authors.....xiii**
- About the Technical Reviewerxv**
- Acknowledgments xvii**

- Chapter 1: TDD Basics 1**
 - Types of Testing 2
 - Trouble with Automated Testing..... 3
 - TDD in a Nutshell 3
 - Why Use TDD?..... 5
 - External and Internal Quality..... 6
 - When to Use TDD?..... 7
 - When Not to Use TDD? 8
 - Refactoring..... 8
 - Modularization 8
 - Test Structure..... 9
 - Let's TDD 9
 - Maximum Out of TDD 13
 - Exercise 16
 - Summary..... 17

TABLE OF CONTENTS

Chapter 2: Unit Tests	19
Your First Test	19
What Do We Want to Test?	20
Creating a Unit Test Target.....	21
Adding a Test Case Class.....	23
Assert Methods	25
Assert Method Types	28
Expectations	33
Expectation Types.....	35
Test Ordering.....	36
Randomized Ordering.....	38
Code Coverage	39
Exercise	41
Summary.....	42
Chapter 3: UI Tests	45
Your First Test	45
XCUITest Components	49
Our Chapter Goal.....	50
First Test Case	50
Launching the App	50
Querying the UI	51
Relationships.....	51
Interacting with the UI.....	54
UI Events.....	56
Assertions	56
Value Assertion.....	57

Accessibility	57
Accessibility Tips	59
Putting It All Together	61
Improve UI Tests.....	62
Exercise	63
Summary.....	63
Chapter 4: Testing Pyramid	65
Our App	66
UI Tests.....	67
Integration Tests.....	70
Unit Tests.....	77
Summary.....	81
Chapter 5: TDD Deep Dive	85
CoffeePot.....	85
Eye on the Big Picture.....	86
Requirements.....	87
Testing Pyramid	89
First Story	90
Architecture	91
MVP	92
First Integration Test	93
Unit Tests.....	96
CoffeeDrinksDataSource	96
CoffeeDrinksModelTests	98
CoffeeDrinksPresenterTests	102

TABLE OF CONTENTS

Test Health Check	107
Second Story.....	109
Architecture	111
Exercise	116
Summary.....	116
Chapter 6: Modularization for the Win	119
Why Bother with Modularization?	119
What Is a Module?	122
Modularizing Your App	126
Introducing Books	128
Modularization Process.....	131
Initial Module Map	132
Choose a Class as a Starting Point.....	133
Identify the Class’s Responsibilities	134
Refactor Responsibilities	135
Refactor the Rest of the Responsibilities	152
Next Starting Point	153
Exercise	153
Summary.....	153
Chapter 7: Dependency Injection and Mocks	157
Stubbing.....	157
Mocking	160
Test Doubles Creation	163
Creation Using Inheritance	163
Creation Using Protocols	165

Dependency Injection.....	168
Initializer Injection	169
Property Injection	170
Stubbing the Network in UI Tests.....	171
Summary.....	180
Chapter 8: Avoiding Multithreading Nightmares	183
What Is Concurrency?	183
GCD.....	184
Queues	184
Serial vs. Concurrent	185
Sync vs. Async.....	186
Cost of Concurrency.....	188
Reader-Writer Problem	190
Singleton Classes	190
Identifying a Race Condition.....	191
Applying TDD to the Problem.....	193
Thread Sanitizer	201
Make It Pass	203
Fixing Threading Issues in Books.....	205
Applying TDD	207
Summary.....	210
Chapter 9: Testing Your Network	213
Networking ABCs	213
HTTP Requests	214
HTTP Responses.....	214
URL	215

TABLE OF CONTENTS

Networking in iOS	216
NSURLSession.....	217
NSURLSessionConfiguration	217
URLRequest	218
NSURLSessionTask	218
Networking in Books.....	219
Process Overview	219
Identify the Class’s Responsibilities	220
Design Overview.....	220
Kickoff	222
Verification Tests	222
Make a Network Request	222
RequestProtocol	224
Execute Request.....	227
Showcasing Test Value	233
Handle a Failing Request.....	233
Putting It All Together	237
Exercise	243
Summary.....	243
Chapter 10: Taming Core Data	245
The Core Data Stack	246
Managed Object Model.....	247
Persistent Store Coordinator	248
Persistent Store	248
Managed Object Context	249
Persistent Container	249

Core Data in Books.....	250
Testing Stack	250
CoreDataManager.....	251
CoreDataStack.....	253
Inject the Stack into CoreDataManager.....	262
TestEntity.....	264
Creation	265
Fetching.....	270
Updating	272
Advanced Fetching	274
Next Steps	277
Putting It All Together	283
Exercise	285
Summary.....	285
Chapter 11: Adding Features to a Legacy App.....	287
Legacy Code Disclaimer.....	288
A/B Testing.....	288
New Feature.....	289
Kickoff.....	290
UI Tests	291
Integration Tests.....	294
Unit Tests and Actual Implementation	297
Final Steps.....	310
Summary.....	310

TABLE OF CONTENTS

Chapter 12: Handling Production Issues	311
Our Tool	311
Integration	312
Production Bug	313
Debugging	313
UI Test	314
Unit Tests	315
Production Crash.....	319
Debugging	320
UI Test	323
Handle A/B Testing.....	325
Fixing Our Test.....	326
Summary.....	327
Index.....	329

About the Authors

Khaled El-Morabea is an engineering manager at Instabug. He has been an iOS developer for more than 8 years and leading the iOS team for more than 3 years. In that time he has worked on several projects. During his time at Instabug, he has worked on multiple integral products, both as a developer and as a strategic engineering manager.

Hassan El-Garem has been involved in the field of iOS development for 5 years, during which he worked on multiple apps and projects. He has a passion for testing and for working on complex projects while maintaining the highest level of quality. Following his passion for testing has led him to create a closed-source testing framework used for randomized stress testing.

About the Technical Reviewer

Vishwesh Ravi Shrimali graduated in 2018 from BITS Pilani, where he studied mechanical engineering. Since then, he has worked with Big Vision LLC on deep learning and computer vision and was involved in creating official OpenCV AI courses. Currently, he is working at Mercedes-Benz Research and Development India Pvt. Ltd. He has a keen interest in programming and AI and has applied that interest in mechanical engineering projects. He has also written multiple blogs on OpenCV and deep learning on LearnOpenCV, a leading blog on computer vision. He has also coauthored *Machine Learning for OpenCV 4* (Second Edition) by Packt. When he is not writing blogs or working on projects, he likes to go on long walks or play his acoustic guitar.

Acknowledgments

We would like to thank the many people who helped us write this book. We are very grateful to Vishwesh Shrimali—his constant and thorough feedback was integral in making this book. We would also like to thank Moataz Soliman, Ahmed AbouElhamayed, Mahmoud Othman, Aprille Muscara, and Anwar El-Wakil—without their feedback, this book would've been in a much worse state. We also thank Aaron Black, for believing in this project from day one. And, finally, we especially thank our editor, Jessica Vakili, who was extremely supportive throughout the whole process of writing this book.

CHAPTER 1

TDD Basics

A developer is a craftsman, a skilled individual driven by passion. Most developers enjoy what they do for a living, to the extent that a lot of developers choose coding as their secondary hobby in their free time. They are proud of what they develop and set high quality standards for their work. Nothing feels better than releasing new code that works well and meets users' expectations. The user here could be the customer or the developer who developed the code themselves. This is important to realize. This sets the intention of the developer as someone who wants to produce high-quality results.

It is no secret that everyone out there including you wants their software projects to be of the highest quality. Yet achieving such a standard isn't particularly easy, and maintaining it can be even harder. Let's say you have worked on an MVP (Minimal Viable Product) and it got released. In most cases this is not the end of the story. You'll probably keep on adding features to it. At some point you'll even realize you need to rewrite a big part of your code or swipe out a dependency for another. These constant changes will eventually compromise your project's quality. Even bug fixes can make a dent at your quality. It's very common to fix one bug and have it cause another more serious bug someplace else. So how can we reach a high quality standard and maintain it? We need to have constant feedback that tells us if our changes introduce any issues. And how can we get such feedback? The answer is simple: testing.

Types of Testing

There is more than one type of testing you can utilize to address these problems. The first solution we will discuss is manual testing. Manual testing is a type of testing in which test cases are executed manually either by a tester or directly by the developer. Manual testing in many cases is considered to be an imperative part of the software cycle. Good testers often have a knack of thinking of highly irregular scenarios, which ultimately leads to identifying hidden bugs.

Humans are amazing creatures. However, for a system of any size, solely depending on manual testing is highly impractical for a variety of reasons. Due to the limited speed of humans, depending on manual testing will ultimately slow down the release process as well as hinder the ability to scale your system. Also, no matter how good a tester is, they are still susceptible to human errors. Switching out the number “0” with the letter “O,” for example, in some contexts can be the sign of a major bug, but many humans might miss this. And last but certainly not least, if you depend only on manual testing, your testing budget will cost you an arm and a leg.

Since we can't solely depend on manual testing, we need to introduce automated testing into our process. Automated tests address all the problems with manual testing. It's fast; a machine can run a test in milliseconds. It's accurate; a machine will not make humanlike mistakes, unless the human who wrote the test makes a mistake. It's inexpensive in the long run. Only the creation of tests is expensive, but running tests after that costs close to nothing. Generally, a combination of manual and automated testing yields the best results. But in many cases, where the project is small enough, we can actually depend solely on automated testing.

Trouble with Automated Testing

The introduction of automated tests gives you and the testers more confidence in your project. It provides an immediate validation that all the basic requirements are being met and leaves the testers to focus on identifying those hidden bugs. However, writing automated tests is considered by many developers a boring activity. We saw many cases where developers started off the project with the intention of writing tests, but they ditched adding tests once the ball actually started rolling. And the main reason for that was that they just didn't like adding tests.

Even if you were able to bite the bullet and commit to writing tests or if you are one of the minority that finds testing fun, you can still be writing bad or unnecessary tests. To be able to see a direct positive effect on quality, we need to ensure the quality of our tests themselves. Yes, tests have quality. Just like we can have bad code, we can have bad tests. After all, tests are also code. Another point to consider is how relevant our tests are. A higher test coverage does not mean that our code is properly tested. We could be adding lots of tests that are useless. For example, we could be adding tests for unused code or multiple tests that test the same thing or even tests that can never fail, like testing getters and setters.

We need to be writing the right tests with good quality and for the right components. This is where **Test-Driven Development** (TDD) comes in. It helps us in achieving just that and more.

TDD in a Nutshell

TDD in its essence is a very simple programming process. It consists merely of four steps (Figure 1-1).

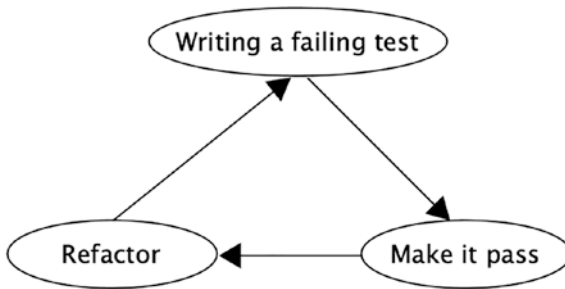


Figure 1-1. *The TDD cycle*

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

This cycle is called the TDD cycle. This process is arguably the best way to ensure high quality of any project. This is because it ensures that your code is fully covered by tests, because writing of the code is **test-driven**.

The cycle is often color-coded:

1. Red: Write a failing test. Since you haven't written anything before that, it's only natural that this test will fail.
2. Green: Write the minimum amount of code that gets your test to pass.
3. Refactor: Clean up your test and code to get it up to standards if needed.
4. Repeat: Do this cycle again. This is what makes it a cycle. We only stop when all requirements are implemented.

The cycle is color-coded as the colors correspond to how most editors (including Xcode) display test results:

- Failing tests are shown with red color.
- Passing tests are shown with green color.

Why Use TDD?

We've mentioned a few troubling problems with writing tests. The most popular problem among developers is how boring and demotivating writing tests can be. Many can't wrap their heads around writing a test for code they themselves wrote. And even if they are able to look past this and see the value of having tests, they can end up writing bad tests. And we can't blame them. It's normal to not perform well when you're not enjoying what you're doing.

This is where TDD changes the game. TDD transforms testing from a boring practice to a design activity. By writing **tests** before writing code, TDD redefines how we look at testing. We no longer use tests to merely validate that the code we just wrote works (while knowing in the back of our minds it probably works since we just wrote it). In TDD, we use them to think about what we want the code to do and how we'll implement it.

As we mentioned before, not all tests are good. TDD helps in ensuring the quality of our tests. For us we consider a test to be good when it follows the FIRST rules that are defined by Uncle Bob Martin in his well-known book *Clean Code*. FIRST is an acronym with each letter referring to a rule:

- **Fast:** Tests need to be fast. With TDD we always run our tests with every step, which pushes us to have fast tests. If we have slow tests, say 1 second each, and we keep adding tests as we go, this will eventually discourage us from running the test suite. If we end in this state, it means this is no longer TDD. Therefore, to keep using TDD, we're always encouraged to keep tests fast.

- **Independent:** Tests should not depend on each other. In TDD, we always run all our tests to make sure everything is passing before we proceed. This makes sure that a test passes even when run with other tests.
- **Repeatable:** Tests should be repeatable in any environment. As we just mentioned, in TDD, we constantly run all our tests with every step. This ensures that our tests are always passing and forces us to keep them unaffected by any external factors.
- **Self-validating:** Tests should have a Boolean output, either pass or fail. The first step in TDD is to write a failing test. And then we add code to make it pass. This proves that the test can fail and pass. Having a test that always passes is counterintuitive and is just a waste of time.
- **Timely:** Tests should be written right before writing code. Which is basically Uncle Bob telling you: “Use TDD!”

In addition to being a good test, a test needs to be relevant and add value. With TDD we write our tests **before** writing the code. Since each test we write directly corresponds to an acceptance criteria for a part in our code, this gives us confidence that the tests we’re adding actually have value.

External and Internal Quality

Our project’s quality is divided into two sections, the external quality and the internal quality. External quality is how well the system meets customer expectations. With external quality we care about our app being functional and providing the expected experience for our end user. We also

care whether or not our app is reliable, responsive, etc. Internal quality, on the other hand, is how well the system meets developer expectations. With internal quality we care about how our internal components behave in different situations. We also care about how easy our code is to understand, change, scale, etc.

When using TDD, you always think of the requirement first and write a test for it and then think about the implementation. This gives us high confidence that our test correctly validates our end requirement. In other words, it upholds and maintains the external quality. When it comes to internal quality, every step in TDD helps us gather feedback both on our design and actual implementation. As you'll see in future chapters, we always cover each component completely with tests. This ensures that each internal component performs as expected. It also upholds the quality of the code itself, since developing using TDD forces you to constantly rethink your design at every step. Having to write a failing test at first encourages us to write loosely coupled code so that it can be easily tested. So thinking test-first directly contributes to the quality of our design, and in each cycle it pushes us to write a better-structured code if needed.

When to Use TDD?

You can use TDD at any point in the lifetime of a project. You can use it with projects from the get-go or on outdated legacy projects. We strongly encourage using TDD whenever possible. Best-case scenario is using TDD on a brand-new project and sticking with it. Then you would truly feel the blessing of having a completely comprehensive test suite, and you will reap the full rewards of TDD. We can also use TDD to guide the process of adding new features to legacy apps. We can even use TDD to guide the refactoring of parts inside a legacy app.

When Not to Use TDD?

The answer to that question is subjective. In almost all cases, it will make sense to use TDD. However, some use cases do not warrant the use of TDD. The benefits of TDD are most evident in long-term projects. So if you're working on a small project that will be done in a short time and you won't revisit it again, then it might make sense to skip TDD in favor of speed and just add tests after or even don't add tests. It all depends on the nature of the project. At the end of the day, TDD is a tool, and it's up to you to use it when you think it is needed.

Refactoring

We've mentioned refactoring a couple times now. It is the third step in the TDD cycle. So what is refactoring? Refactoring is the process of changing how internal code is structured/written without changing its behavior. Refactoring is always done in small iterative steps. Each step should enhance the structure of our code and be small enough at the same time so that it's understandable. An example of a small meaningful refactor is moving a block of code to a new helper function or extracting it into a new class. Though it might not seem like much, when numerous small refactorings are performed, we eventually start to see an impact on our code. With each change applied, we can make sure that it doesn't break anything by running our tests, that is, of course if we had been using TDD.

Modularization

The term "modularization" refers to the division of a system into a number of relatively independent and interchangeable modules with well-defined interfaces. Each one is tiny enough and simple enough to be well understood and extensively tested; each one contains everything required to carry out

the intended functionality. We can go for a modularized approach when designing our system, and using TDD will encourage us to do so. However, if we have a non-modularized system, we can still modularize it through the use of refactoring. A non-modularized app, by its nature, will contain lots of code smells, will not be testable, and will be harder to maintain. You'll learn more about the process of modularizing a legacy app by using TDD in future chapters. For the remainder of this chapter, let's look at some examples of TDD in action beginning with test structure.

Test Structure

Before we start writing tests using TDD, let's talk about how we'll structure our tests. A good structure for all your tests is this one:

1. Set up the test data.
2. Call your method under test.
3. Assert that the expected results are returned.

An easier way to remember this pattern is the “given,” “when,” and “then” triad, which is inspired from Behavior-Driven Development (BDD), where *given* reflects the setup, *when* the method call, and *then* the assertion part.

This pattern ensures that your tests remain consistent and easy to read. On top of that, tests written with this structure in mind tend to be shorter and more verbose. We will be using this structure throughout this book in all our tests.

Let's TDD

Now let's take an example and try to implement it using TDD. Go ahead and open up this chapter's starter project. You can find it in the chapter's resources. We want to create a tax calculator that calculates the net salary

out of an original salary after subtracting 30% taxes. Let's start with the first step, writing the test. Our first test can be something like this:

```
func testExample() throws {
    // Given
    let calculator = TaxCalculator()
}
```

Ultimately a test represents a requirement, and the preceding test details our most basic requirement: that we have a class named `TaxCalculator`. Since this one line won't even compile, you might think we are heading in the wrong direction, but we're now actually done with our first step; we wrote a failing test.

On to step 2, let's make this test pass using the minimum amount of code. To do so we need to add the following:

```
class TaxCalculator: NSObject {
}
```

Now if we run our test, it will pass, meaning we're done with step 2. Now for step 3, we check if there's anything to refactor. Right now there's none, since we only wrote two lines of code.

Since we're done with our three steps, what we do now is repeat our TDD cycle. Let's start by adding a new requirement to our test that will make it fail. The next requirement is that we have a function in `TaxCalculator` that takes salary and calculates net salary. When we translate this requirement, our test will look like this:

```
func testExample() throws {
    // Given
    let calculator = TaxCalculator()

    // When
    let netSalary = calculator.calculate(100)
}
```

Now let's fix this test by modifying `TaxCalculator` but again using minimum code. So basically all we need to do is this:

```
class TaxCalculator: NSObject {
    public func calculate(salary: Int) -> Int {
        return 0
    }
}
```

Since now the test is passing and there's no need for refactoring, let's repeat our cycle one more time. Now we'll add the requirement for the output of our function:

```
func testExample() throws {
    // Given
    let calculator = TaxCalculator()

    // When
    let netSalary = calculator.calculate(100)

    // Then
    XCTAssertEqual(netSalary, 70,
                    "Net salary failed")
}
```

If you run this test, it will fail, which is what we're expecting. But before going to fix the test, we need to test something essential. If you see this message "Net salary failed" while working on your project, do you think you will know your project's current problem or you will need to debug? If the answer is no, you will need to write a descriptive message to help whoever is working on this project (possibly your future self) to know what they just broke:

```
func testExample() throws {
    // Given
    let calculator = TaxCalculator()

    // When
    let netSalary = calculator.calculate(salary: 100)

    // Then
    XCTAssertEqual(netSalary, 70,
                   "Net salary should be 70$ when you
                   subtract 30% taxes from 100$")
}
```

If you saw "Net salary should be 70\$ when you subtract 30% taxes from 100\$", you will precisely know what the problem is and which method you need to check.

Now we need to write the code that makes the test pass. After adding the code, it should be something like this:

```
class TaxCalculator: NSObject {
    public func calculate(salary: Int) -> Int {
        return salary - ((salary * 30)/100);
    }
}
```

After running the test, the test is green now, and we still don't need refactoring (Figure 1-2).

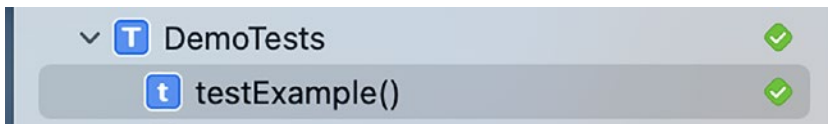


Figure 1-2. *testExample* passed

Maximum Out of TDD

What we did in the introduction is a quick brief about TDD. But to make TDD improve your quality significantly, you need to change your way of thinking about test cases. Test cases are not just happy scenarios, they should also cover corner cases. Most of the time, you will write code that fulfills all happy scenarios. Let's try to improve our test cases. You need to think about how to break it. What if someone passes a fraction? What if someone passes a negative value? What if someone passes zero?

We're gonna handle these using the exact same steps. Let's take the fraction scenario and write a test for it:

```
func testPassingFractionNumber() throws {
    // Given
    let calculator = TaxCalculator()

    // When
    let netSalary = try calculator.calculate(salary: 0.5)

    // Then
    XCTAssertEqual(netSalary, 0.35,
                    "Net salary should be 0.35$ when you
                    subtract 30% taxes from 0.5$")
}
```

Now to fix the test we'll have to do the following:

```
class TaxCalculator: NSObject {
    public func calculate(salary: Double) throws -> Double {
        return salary - (salary * 0.3);
    }
}
```

Still no need for refactoring, so we'll repeat once again. Now let's consider the cases for zero and negative. We'll probably need to throw an error in these cases. Which is exactly what we're going to reflect in our tests:

```
func testPassingNegativeNumber() throws {
    // Given
    let calculator = TaxCalculator()

    // When
    do {
        _ = try calculator.calculate(salary: -1)
    } catch let caughtError as TaxCalculatorError {
        // Then
        XCTAssertEqual(caughtError, .negativeSalaryError,
            "Should throw error when passing a negative
            salary.")
    }
}

func testPassingZero() throws {
    // Given
    let calculator = TaxCalculator()

    // When
    do {
        _ = try calculator.calculate(salary: 0)
    } catch let caughtError as TaxCalculatorError {
        // Then
        XCTAssertEqual(caughtError, .zeroSalaryError,
            "Should throw error when passing a zero salary.")
    }
}
```

After applying step 2, our class would look like this:

```
enum TaxCalculatorError: Error {
    case negativeSalaryError
    case zeroSalaryError
}

class TaxCalculator: NSObject {
    public func calculate(salary: Double) throws -> Double {
        if salary < 0 {
            throw TaxCalculatorError.negativeSalaryError
        }

        if salary == 0 {
            throw TaxCalculatorError.zeroSalaryError
        }

        return salary - (salary * 0.3);
    }
}
```

Now that all tests are passing, we can move on to step 3. We can probably refactor error handling into a helper function:

```
enum TaxCalculatorError: Error {
    case negativeSalaryError
    case zeroSalaryError
}

class TaxCalculator: NSObject {
    public func calculate(salary: Double) throws -> Double {
        try handleErrors(salary: salary)
        return salary - (salary * 0.3);
    }
}
```

```

private func handleErrors(salary: Double) throws {
    if salary < 0 {
        throw TaxCalculatorError.negativeSalaryError
    }

    if salary == 0 {
        throw TaxCalculatorError.zeroSalaryError
    }
}
}

```

And after any refactor, we can simply run our tests to make sure we did not break any functionality in the process (Figure 1-3).

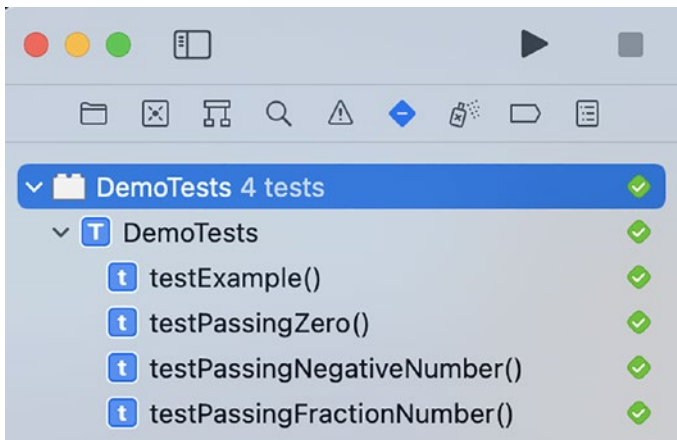


Figure 1-3. All tests passed

Exercise

TaxCalculator now calculates the salary after removing a constant percentage, which is always 30%. For your exercise, try making this percentage dynamic, meaning that we can pass the calculate function the salary and a custom percentage. We also want to keep the 30% as the default.

Summary

Untested code is basically a ticking time bomb that can explode at any second in the form of bugs and crashes. Even the tiniest of changes can introduce regressions. And these regressions can only be caught by testing. We found that we can't solely depend on manual testing and need to make use of automated testing.

Though writing tests has tremendous value and directly contributes to a project's quality, most developers don't do it. This is because for many developers, writing tests is quite boring. They'd rather be writing actual code instead of writing tests for code they just wrote. However, there is one way of developing that completely revolutionized how we look at tests, which is Test-Driven Development (TDD).

TDD is the process of writing tests first before writing code. Doing so leads to us having a project that is highly covered by tests. TDD transforms the process of writing tests from a boring ordeal to a fun design activity. By having to write tests before we write code, tests now become a way of defining our requirements and help us think how to achieve these requirements.

TDD has a direct and substantial effect on the number of tests in our projects. We add a test before writing any code, which means all our code will be covered by tests. Working on a code base that has high test coverage can be life changing. It efficiently catches regressions and gives confidence for the developer whenever a change is made through a very fast feedback loop.

TDD doesn't only affect our test coverage, which helps in maintaining our external quality. It also directly affects the quality of code. Writing tests before code makes us clearly think about what the code should do and how it will do it. And having to write tests first also forces us to write testable code, which in turn translates to loosely coupled code with good design.

We can use TDD in various settings. We can use it on new projects from the start or when adding new features to old legacy projects. We can also apply TDD when attempting to refactor parts of old code or even when attempting to modularize a legacy app.

The TDD process is a very simple one. We have just three steps. First, we write a failing test. In order to write a failing test, we need to think about what the code should do and translate this requirement into a test. The second step is to write as little code as possible to make this test pass. And finally when we have a passing test, we start to think if we can improve our code in any way, be it a design change or an implementation change. When we finish step 3 and we're sure that our change (if any) didn't make any of our tests break, we loop back again to step 1 and find a new requirement that we can translate into a failing test.

CHAPTER 2

Unit Tests

As you know by now, TDD is a process in which you write a test first before writing actual code. But before jumping into TDD, you need to understand the basics of testing in iOS. Luckily, every year, Xcode and Swift are becoming more and more powerful when it comes to testing. And the testing framework “XCTest” is also evolving with them.

This chapter covers how to use XCTAssert functions to write functional tests. These are the main components of XCTest. You’ll also learn how to use expectations to test async code. Next, you’ll go through best practices when it comes to organizing your test suite and tests. Then you’ll use the debugger to find and fix errors in your tests. Finally, you’ll go through gathering code coverage to make sure the tests you’ve written are sufficient.

Your First Test

Let’s forget about TDD for a while and just focus on testing basics. Go ahead and download and open the starter project **Calc**, which you can find in this chapter’s resources. Calc is a framework (Figure 2-1) that provides some basic mathematical operations as well as some special operations. Calc also logs and saves the output from each operation.

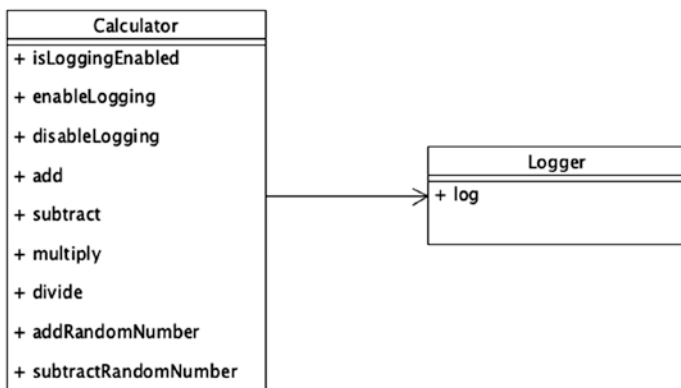


Figure 2-1. *Calculator framework class diagram*

Calc has two components: Calculator and Logger. Calculator has the following functions: add, subtract, multiply, divide, addRandomNumber, and subtractRandomNumber. It also has a function to check if logging is enabled and functions to enable/disable logging. As for Logger, it has one function that takes a number and logs it. If the number is within the limit, it saves it. Else, it throws an error. Calculator uses Logger to log the output of each operation.

If you look around the project, you'll find that there are no tests added at all. And that's what we'll fix while walking you through the basics of XCTest.

What Do We Want to Test?

- Logging is enabled by default,
- The disableLogging function correctly disables logging.
- The enableLogging function correctly enables logging.
- The Logger instance inside Calculator is initialized by default.

- The Logger instance is cleared when logging is disabled.
- All operations are working as expected.
- Logger's log function saves the provided number if it's less than the limit.
- Logger's log function throws an error if it's greater than the limit.

Creating a Unit Test Target

In order to run tests, first, we need a unit test target. A unit test target is a separate executable with a single purpose, running your unit tests. When you ship your app to the App Store or distribute your framework, this test target is not included.

Open the Test navigator by pressing **Command+6**.

Click the + button in the lower-left corner. Then select **New Unit Test Target...** from the menu (Figure 2-2).

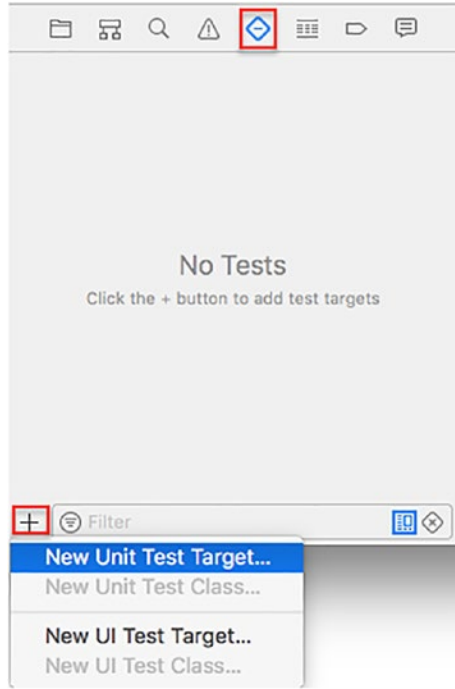


Figure 2-2. Add a unit test target

Accept all the default values and click **Finish**.

You should now see the newly added test target in the Test navigator (Figure 2-3).

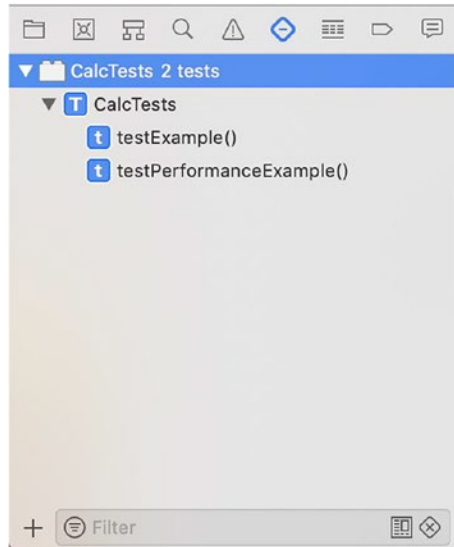


Figure 2-3. Test navigator

Xcode automatically generates a test case file named **CalcTests.swift**. We will not be needing that, so go ahead and delete it.

Adding a Test Case Class

We will start by writing tests for Calculator, and the first step would be to create a test case class to include these tests.

Go to the Test navigator and select our now empty test target **CalcTests**. Then click the + button in the lower-left corner. Then select **New Unit Test Class...** from the menu. In the **Class** field, enter “CalculatorTests” and then press Next and then Create.

The default template (Figure 2-4) imports the testing framework, XCTest, and defines a CalculatorTests subclass of XCTestCase, with setUpWithError(), tearDownWithError(), and example test methods.

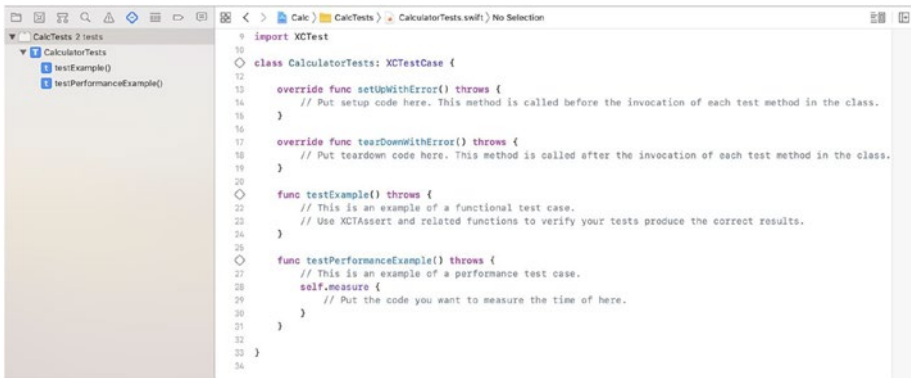


Figure 2-4. *CalculatorTests*

Go ahead and remove the example test methods. Also remove the setup and teardown methods as we will not be needing them for now.

Now it's time to add the very first test in this project. We want to test that logging is enabled by default. We can check whether logging is enabled or not using the public function `isLoggingEnabled()`.

First add this new line to the beginning of the file to import our framework:

```
import Calc
```

Then add the following inside `CalculatorTests`:

```
func testIsLoggingEnabledByDefault() {
    // Given
    let calc = Calculator()

    // When
    let isEnabled = calc.isLoggingEnabled()

    // Then
    XCTAssertTrue(isEnabled)
}
```

Here we create a new instance of `Calculator`, and then we call `isLoggingEnabled` and save the outcome in the variable `isEnabled`. And in the **Then** section, we assert that `isEnabled` is true.

Run the test by clicking the diamond next to it or from the Test navigator. The test should pass (Figure 2-5).

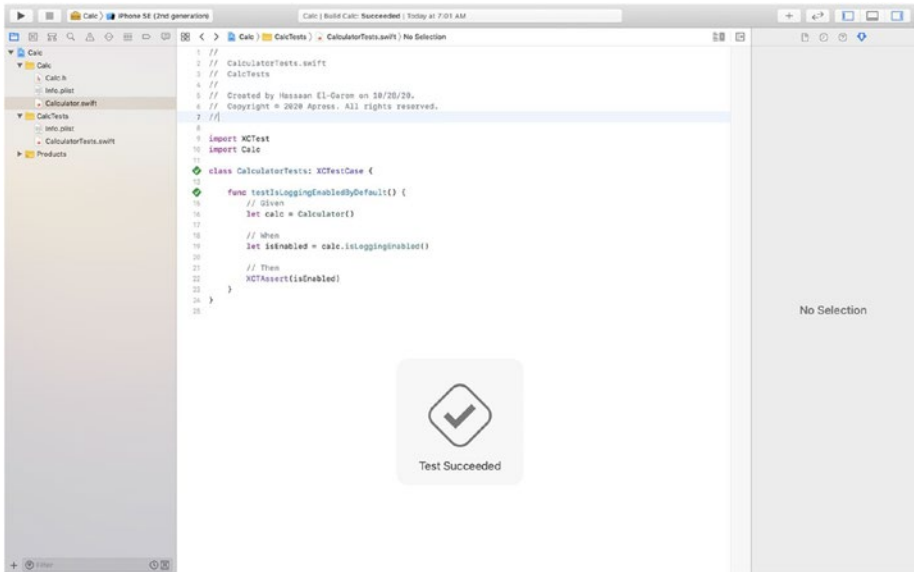


Figure 2-5. *Your first test!*

You've just written and run your first test!

Assert Methods

In the first test we wrote, we used `XCTAssertTrue`, which asserted that the given expression evaluated to true. However, our method has another possible outcome that returns false. If `disableLogging()` is called,

CHAPTER 2 UNIT TESTS

`isLoggingEnabled()` should return `false`. Let's go ahead and write that test:

```
func testDisableLogging() {
    // Given
    let calc = Calculator()

    // When
    calc.disableLogging()
    let isEnabled = calc.isLoggingEnabled()

    // Then
}
```

Now we want to assert that `isEnabled` is `false`. Your first instinct might be to do something like this:

```
XCTAssertTrue(!isEnabled)
```

Let's go ahead and add it and run the test.

The test passes!

As you can see, we can assert on anything we want using `XCTAssertTrue` alone, equality, nullability, comparison, or anything. However, this introduces two problems: bad test readability and bad test output readability. Let's take a look at the following test for instance:

```
func testExample() {
    let x = "foo"
    let y = "bar"
    let z = foo == bar
    XCTAssertTrue(z)
}
```

By glancing at this test, we can see that we're asserting that `z` is true, but in order to understand what we're actually testing, we need to go back and check what "`z`" is. You might think that this is not too big of a problem, but when tests get more complex and elaborate, this problem will be very evident.

The second and more important problem is the test result. This is the test result error when running the preceding test:

```
XCTAssertTrue failed
```

As you can see, it's a bit uninformative and tells us nothing about what went wrong or the values of `x` and `y`.

Now that we've identified the problems with using `XCTAssertTrue` only, where should we go from here? Fortunately, `XCTest` has got us covered. As we've mentioned before, `XCTest` is a very powerful testing framework, and one of those core powers is its versatile suite of assertion methods. One of those methods is `XCTAssertEqual`.

We can refactor the previous test case to use `XCTAssertEqual`, and it would look like this:

```
func testExample() {
    let x = "foo"
    let y = "bar"
    XCTAssertEqual(x, y)
}
```

This makes the test a little bit more verbose and easier to understand. And if we run this test, the test result error is far more descriptive:

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

Assert Method Types

XCTest has many assertion methods, and they can be categorized into five categories:

1. Truthfulness
2. Equality
3. Nullability
4. Comparison
5. Errors

Truthfulness Asserts

- `XCTAssertTrue`

Asserts that the given expression evaluates to true

- `XCTAssertFalse`

Asserts that the given expression evaluates to false

- `XCTAssert`

An alias for `XCTAssertTrue`

So far, we have been using `XCTAssertTrue` exclusively. However, now we can refactor `testDisableLogging` to use `XCTAssertFalse`. Go ahead and replace the last line in the test with this:

```
XCTAssertFalse(isEnabled)
```

Equality Asserts

- `XCTAssertEqual`

Asserts that the given two expressions are equal to each other

- `XCTAssertNotEqual`

Asserts that the given two expressions are not equal to each other

For all equality assertions, the passed expressions need to be of the same type, and that type should conform to `Equatable` or `FloatingPoint`.

Let's add a test for `add(firstArgument: FloatingPoint, secondArgument: FloatingPoint)`:

```
func testAdd() {
    // Given
    let calc = Calculator()

    // When
    let output = calc.add(firstArgument: 1, secondArgument: 2)

    // Then
    XCTAssertEqual(output, 3)
}
```

Here we simply assert that the output of the function is equal to the expected output, which is “3.”

Nullability Asserts

- `XCTAssertNil`

Asserts that the given expression is `nil`

- `XCTAssertNotNil`

Asserts that the given expression is not `nil`

CHAPTER 2 UNIT TESTS

When a `Calculator` instance is initialized, a new `Logger` object is created and saved as a variable inside `Calculator`. And when `disableLogging()` is called, this variable is set to `nil`. Let's add tests to cover this part:

```
func testLoggerIsInitializedByDefault() {
    // Given
    let calc = Calculator()

    // Then
    XCTAssertNotNil(calc.logger)
}
```

```
func testDisableLoggingResetsLogger() {
    // Given
    let calc = Calculator()

    // When
    calc.disableLogging()

    // Then
    XCTAssertNil(calc.logger)
}
```

When you add these tests, you'll face a build error that looks like this:

```
'logger' is inaccessible due to 'internal' protection level
```

To fix this, replace

```
import Calc
```

with

```
@testable import Calc
```

When we add the `@testable` attribute to an import statement for a module compiled with testing enabled, we activate elevated access for that module in that scope. Classes and class members marked as `public` behave as if they were marked `open`. Other entities marked as `internal` act as if they were declared `public`.

Comparison Asserts

- `XCTAssertGreaterThan`
- `XCTAssertGreaterThanOrEqual`
- `XCTAssertLessThan`
- `XCTAssertLessThanOrEqual`

For all comparison assertions, the passed expressions need to be of the same type, and that type should conform to `Comparable`.

Let's make use of the comparison asserts and write a test for `addRandomNumber`. Here we want to assert that the output is greater than the passed argument:

```
func testAddRandomNumber() {
    // Given
    let calc = Calculator()

    // When
    let output = calc.addRandomNumber(argument: 1)

    // Then
    XCTAssertGreaterThan(output, 1)
}
```

Errors Asserts

- XCTAssertThrowsError
- XCTAssertNoThrow

These assert methods are used to assert functions that throw errors.

Our `Logger` throws an error if we try to log a number greater than **1000**. Let's use these assert methods to cover this part.

First, we'll need to add a new test case class to include the `Logger` test. Create it the same way as before and name it "`LoggerTests`." First, add the `@testable` import statement. Then remove the autogenerated code and replace it with this:

```
func testAddLogShouldThrowIfExceedsLimit() {
    // Given
    let logger = Logger()
    let number: Double = 2000

    // Then
    XCTAssertThrowsError(try logger.log(number))
}

func testAddLogShouldNotThrowIfUnderLimit() {
    // Given
    let logger = Logger()
    let number: Double = 500

    // Then
    XCTAssertNoThrow(try logger.log(number))
}
```

These two tests cover the two scenarios where the logger throws an error and where it doesn't.

Expectations

Now that you are familiar with the assertion functions, we'll kick it up a notch. Let's try testing async code. First of all, what is async code? When you execute something synchronously, you wait for it to finish before moving on to another task. When you execute something asynchronously, you can move on to another task before it finishes.

Our `Logger.log(_ number: Double, completion: LogCompletion)` function adds the log asynchronously. And it accepts a completion handler and calls it when it's done executing.

Let's try writing a test for it:

```
func testAddingLog() throws {
    // Given
    let logger = Logger()
    let number: Double = 1

    // When
    try logger.log(number) {
        // Then
        XCTAssertEqual(logger.logs.count, 0)
    }
}
```

If you examine the function and test we just wrote closely, you'll find out that the assertion should fail, as the logs count is expected to be 1, not 0. But when we run this test, it passes and only fails occasionally. This is because `log` is async, and what basically happens is that the test execution scope finishes before the function finishes execution or calls the completion handler. So our assert is never actually called. This is where `XCTAssertTrue` alone falls short, async code.

We can fix this test by forcing it to wait until the log finishes. There are many ways we can do that: wait for a specific time or use `DispatchGroup`. But these could be somewhat of an overkill and/or unnecessary, because as you have guessed it, XCTest has got our back again, this time with `XCTestExpectation`.

`XCTestExpectation` is an object that describes something we are expecting to happen in the future, and we want to wait until it happens.

We can create an expectation this way:

```
let exp = expectation(description: "Log added")
```

Go ahead and add this line at the start of our test.

And to wait for an expectation, we need to add this line:

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

Let's fix our assert statement as well. Now the test should look like this:

```
func testAddingLog() throws {
    let exp = expectation(description: "Log added")
    // Given
    let logger = Logger()
    let number: Double = 1

    // When
    try logger.log(number) {
        // Then
        XCTAssertEqual(logger.logs.count, 1)
    }
    wait(for: [exp], timeout: 1)
}
```

Now run the test. The test should still be failing but now with a different error:

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

This here means that the timeout has passed without our expectation being fulfilled, which makes sense since we never defined when the expectation is fulfilled. This here shows the beauty of `XCTestExpectations`. They don't just help us wait for async tasks to finish; they also act as assertion that the expectation is fulfilled in the given time, and if not they report an error.

Let's fix our test by defining when the expectation is fulfilled. Add this line right after the `XCTAssertEqual` line:

```
exp.fulfill()
```

Now the test passes when we run it!

Expectation Types

Just like `XCTAssertTrue`, `XCTestExpectation` is our base expectation, and we can use it to wait and test any async code. But we also have other types of expectations that make it easier to wait for specific events:

1. Normal
2. Key-Value Observing (KVO)
3. Notification
4. Predicate

We covered the normal expectation type, and we'll cover the rest in later chapters.

Test Ordering

We are done with adding tests for now. Open up the checkpoint version of the project, which can be found in the chapter's resources. Run all the tests by pressing **Command+U**.

You should find that one test is failing, which is `testIsLoggingEnabledByDefault`. A helpful tip for debugging failing tests is to use breakpoints. Xcode has a special breakpoint called **Test Failure Breakpoint**, which pauses execution automatically whenever an assertion or expectation failure occurs. You can then make use of Xcode's debugger to examine the current state of your variables.

To add **Test Failure Breakpoint**, open the Breakpoint navigator by pressing **Command+8**.

Click the + button in the lower-left corner. Then select Test Failure Breakpoint from the menu (Figure 2-6).

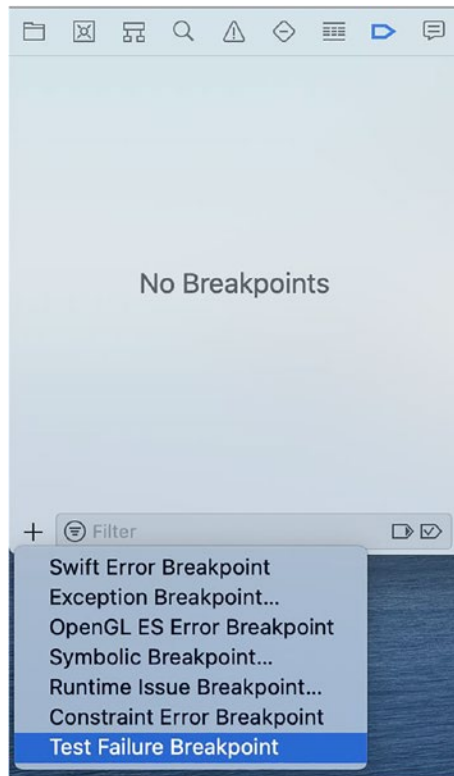


Figure 2-6. *Test Failure Breakpoint*

One interesting thing you might have noticed is that this test was passing before and the only changes we made were adding more tests. This means that our tests are not correctly encapsulated and that some tests affect other tests. Therefore, we need to reset the state of the shared Calculator instance before every test. This can be done by overriding the `setUp()` function. Before each test begins, XCTest calls `setUpWithError()`, followed by `setUp()`. If state preparation might throw errors, we should override `setUpWithError()`. Since we won't be calling any throwing functions, `setUp()` will be enough. Sometimes we might need to perform some sort of cleanup after each test. Then we could use `tearDown()` or `tearDownWithError()`.

Add this inside `CalculatorTests` and before the tests:

```
override func setUp() {
    UserDefaults.standard.removeObject(forKey: Calculator.
kLoggingEnabledDefaultsKey)
}
```

This resets the value of logging enabled as if it was a clean run. Now run all tests again. They should pass again.

Randomized Ordering

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

Edit the **Calc** scheme (**Command+Shift+,**). Select the **Test** action. In the center pane, next to **CalcTests** is an **Options...** button (Figure 2-7).

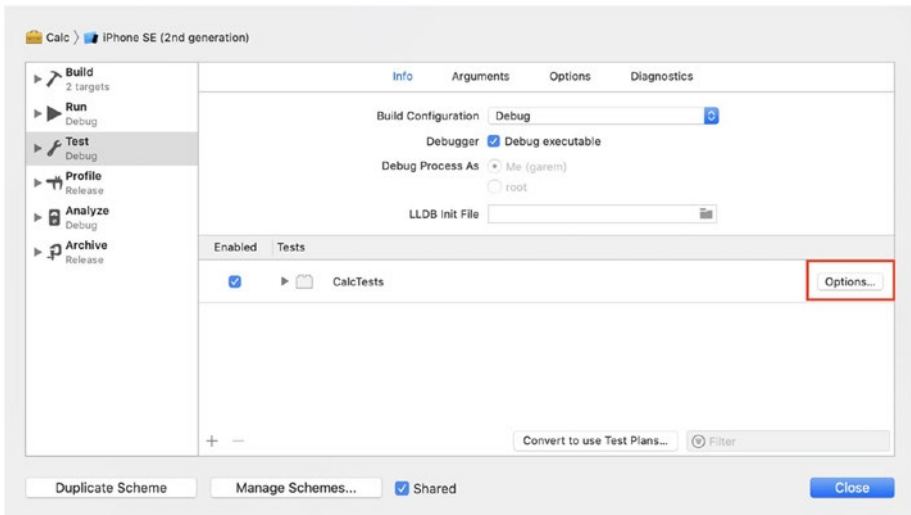


Figure 2-7. Randomize the test order

Click that and, in the pop-up, check **Randomize execution order** (Figure 2-8). This will cause the tests to run in a random order each time.

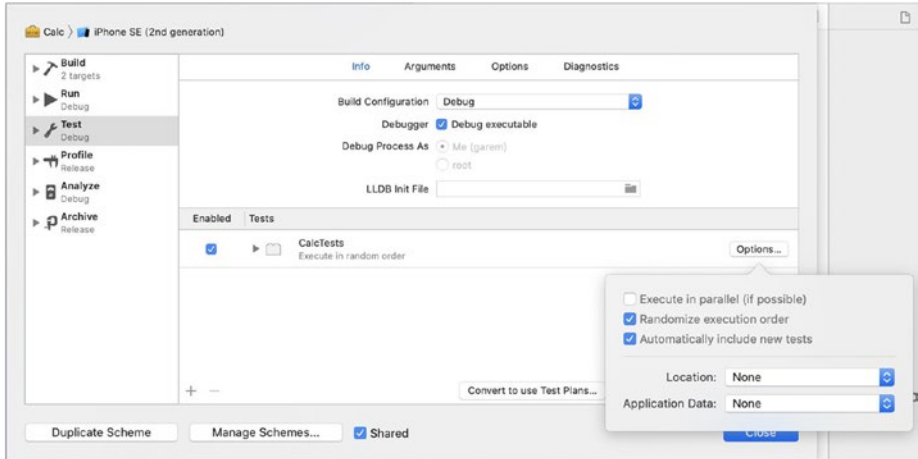


Figure 2-8. *Randomize execution order*

This can help you discover more bugs and expose dependencies between tests that you wouldn't catch using normal static ordering. The downside, however, is that ordering issues are hard to reproduce if they are too specific.

Code Coverage

Since we were just editing our scheme, let's open it up again to enable code coverage. Code coverage enables you to visualize and measure how much of your code is being exercised by tests.

To enable code coverage, open up the Test action again. This time select the Options tab. There is a checkbox for Code Coverage. Check it (Figure 2-9).

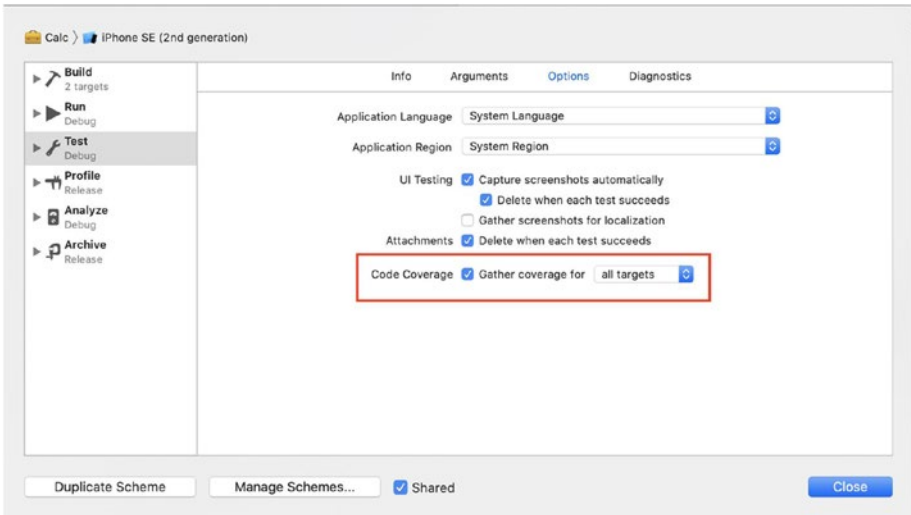


Figure 2-9. Code coverage

Now run the tests again. After the tests pass, open up the Report navigator by pressing **Command+9**. Choose to display the reports **By Group**. And under the latest test, you should find the coverage report, which you can select to display it (Figure 2-10).

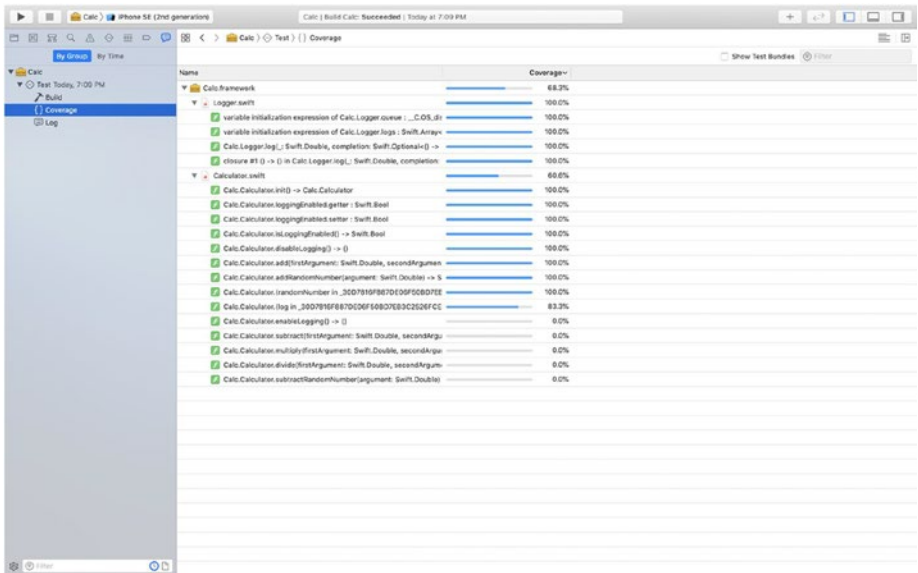


Figure 2-10. Code coverage results

There will be a list of each file in the target along with the percentage of the code lines that were executed. You should always aim for the highest coverage percentage possible.

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

You should note that having a high coverage percentage doesn't necessarily mean that you have added all the required tests.

Exercise

Open the final version of the project from the chapter's resources. Now that you have enabled code coverage, try adding tests till you reach at least 90% coverage. You should make use of the list under the "What Do We Want to Test?" section.

Summary

In this chapter, you got introduced to the basics of unit testing in iOS and to all the powerful functionalities that come with the native testing framework **XCTest**. We learned of the function of test targets and test case classes. We created a test target to be able to add tests for our Calculator project. And then we proceeded to add test case classes for each of our components to be able to add tests inside them.

We then started exploring all the different types of assertion that XCTest has to offer. We have our **Truthfulness** assertions, which basically verify that the expression we provide is either true or false. We then have our **Equality** assertions, which we use to verify that two expressions are either equal or not equal. We also have **Nullability** assertions to verify that the expression we provide is either null or not null. We have our **Comparison** assertions, which can be used to compare two expressions and make sure that one is greater than the other or vice versa. And finally, we have our **Errors** assertions, which we use to verify that a certain expression throws an error or that it doesn't.

It's important to point out that we can perform all the needed assertions using the plain `XCTAssert` or `XCTAssertTrue`. However, using other types of assertions when applicable makes our tests more readable and also makes the error messages that Xcode outputs when an assertion fails more readable and much more useful.

Other than the various assertions that XCTest offers, there are also **expectations**, which make testing asynchronous code rather seamless. We basically create an expectation object, and then we mark it as fulfilled whenever our asynchronous task is finished. And to make our test wait for our async task, we add a single line that tells our test to wait till the expectation is fulfilled.

When running our whole test suite, we might run into the situation where one test causes another test to fail. This happens when tests share the same environment and our tests make changes to that environment,

which then leak to other tests. So when running tests in a specific order, tests might start failing due to them not running in a clean environment. For that, we learned how to use XCTestCase's setup and teardown functions to make any common setup between all our tests and to make any necessary cleanup after each test is done.

Finally we explored some of Xcode's hidden features. We added **Test Failure Breakpoint** to make debugging a failing test easier using Xcode's debugger. With this breakpoint enabled, Xcode will pause whenever an assertion fails, and then you could inspect the state of your variables at the moment of failure. We also enabled **randomized test ordering**, which tells Xcode to run your tests in a different order every time. This can help in spotting even more bugs. And finally, we enabled **code coverage** to get a sense of how much of our code is covered by tests. When this feature is enabled, Xcode generates a report after each test run, which can help in identifying areas with poor coverage that need more tests.

CHAPTER 3

UI Tests

UI tests are your first line of defense, which will tell you whether your application works or not; they interact with the application precisely like what your user does. The XCTest framework will help you query UI elements inside your app and do interactions and then validate UI properties and states. UI tests access your app using the iOS accessibility system. Accessibility is a technology that gives disabled people the same experience on our applications that all our users receive. It offers rich semantic data about the UI, so that voice-over can be used to guide users through the application.

Your First Test

This chapter aims to explore the UI tests in Xcode since we are going to depend on them heavily moving forward. We are going to be writing UI tests for a simple app that displays a list of cities. You can find the starter project for this app in the chapter's resources. The app (Figure 3-1) contains the main screen, which shows a list of cities. Once you select one of them, it will open another screen in which the title will match the chosen city name. You will find a button; once you tap this button, it will show a welcome alert.

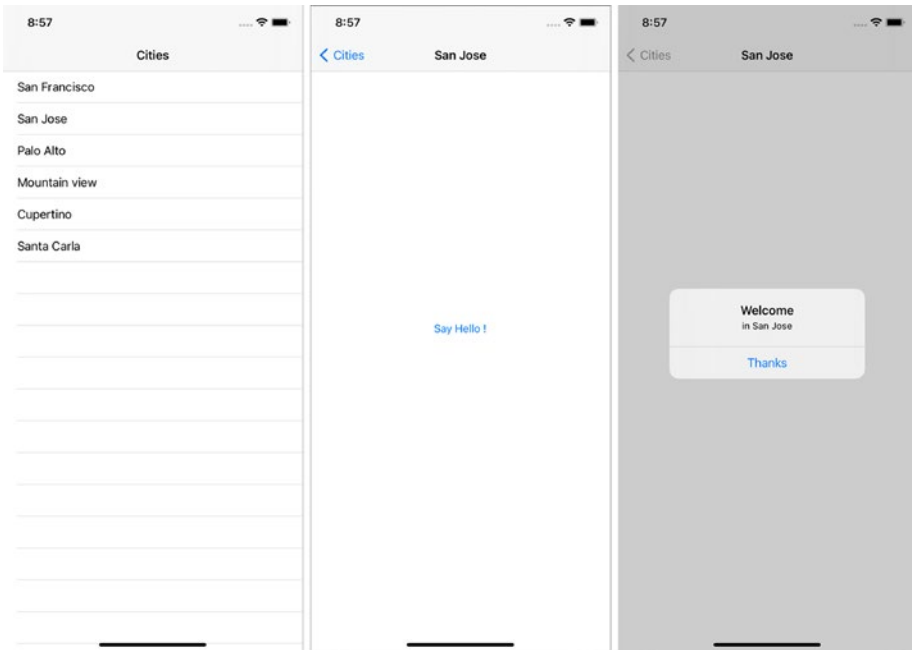


Figure 3-1. *App to be tested*

If we open up the demo app, we'll see that there is no UI test target for our app. So you need to create a new target for UI tests. A UI test target is a separate executable with a single purpose, running your UI tests. When you ship your app to the App Store or distribute your framework, this test target is not included.

Open the Test navigator by pressing **Command+6**.

Click the + button in the lower-left corner. Then select **New UI Test Target...** from the menu (Figure 3-2).



Figure 3-2. *New UI Test Target*

Once you create the UI test target, it will create a new folder that contains your first UI test class that inherits from XCTestCase (Figure 3-3).

```

1 //
2 // DemoUITests.swift
3 // DemoUITests
4 //
5 // Created by khaled mohamed el morabea on 10/11/20.
6 //
7
8 import XCTest
9
10 class DemoUITests: XCTestCase {
11
12     override func setUpWithError() throws {
13         // Put setup code here. This method is called before the invocation of each test method in the class.
14
15         // In UI tests it is usually best to stop immediately when a failure occurs.
16         continueAfterFailure = false
17
18         // In UI tests it's important to set the initial state - such as interface orientation - required for your
19         // tests before they run. The setUp method is a good place to do this.
20     }
21
22     override func tearDownWithError() throws {
23         // Put teardown code here. This method is called after the invocation of each test method in the class.
24     }
25
26     func testExample() throws {
27         // UI tests must launch the application that they test.
28         let app = XCUIApplication()
29         app.launch()
30
31         // Use recording to get started writing UI tests.
32         // Use XCTAssert and related functions to verify your tests produce the correct results.
33     }
34
35     func testLaunchPerformance() throws {
36         if #available(macOS 10.15, iOS 13.0, tvOS 13.0, *) {
37             // This measures how long it takes to launch your application.
38             measure(metrics: [XCTApplicationLaunchMetric()]) {
39                 XCUIApplication().launch()
40             }
41         }
42     }
43 }

```

Figure 3-3. Boilerplate tests

Requirements:

- iOS 9 is the minimum version that supports UI tests.
- UI tests' minimum iOS version should match the version of the application to be tested.

You need to click the diamond button beside testExample. You've just run your first test! (Figure 3-4)

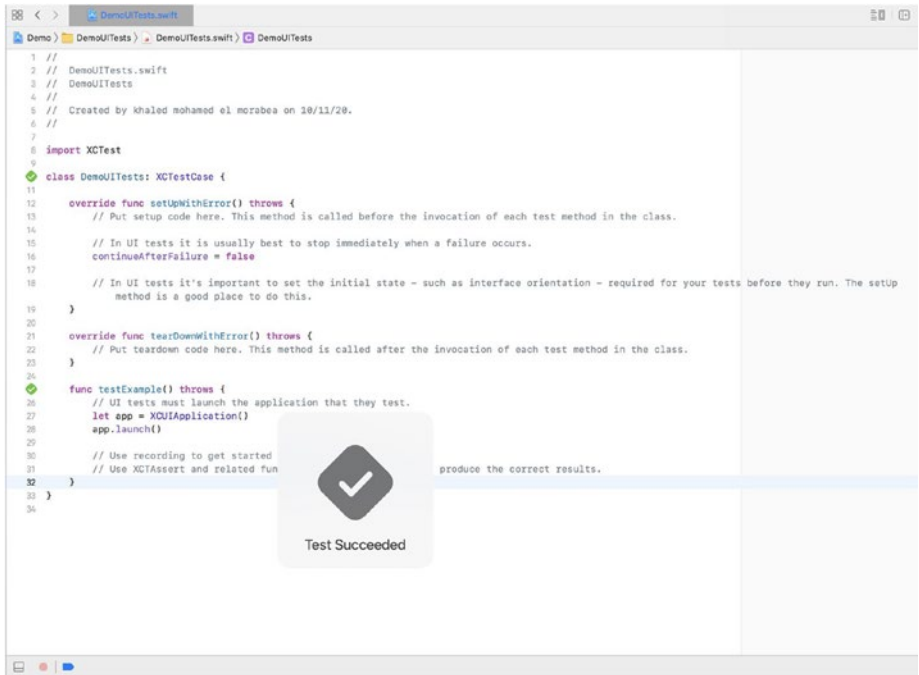


Figure 3-4. Running dummy test

XCUITest Components

The XCUITest framework consists of three main components. We will cover them on the go. These components are

- XCUIApplication
- XCUIElementQuery
- XCUIElements

Our Chapter Goal

As we mentioned earlier, UI tests interact with the application exactly as our user. So we want to interact with our application as our user will do and validate if everything is working as expected or not.

First Test Case

- As a user, I should see six cities in a table view; when I tap on San Francisco city, the app should navigate into another view, and the title should match the selected city. Upon navigating into another view, I should be able to see a "Say Hello !" button, and when I tap it, it should show a welcome message.

If we converted the test case into actions, it would be as follows:

1. Launch the app.
2. Count all cities inside the table view.
3. Select "San Francisco" city.
4. Make sure that the title in the details view is San Francisco.
5. Tap the "Say Hello !" button.
6. Make sure that you see a welcome alert.

Launching the App

To run your tests, you must launch the app. So XCUITest provides `XCUIApplication`, which is a proxy for your application, so that you can launch, terminate, and activate your application. Inside each test you must have a single instance from `XCUIApplication` and call `app.launch()`.

After using the launch API, our first test will be

```
func testExample() throws {
    // UI tests must launch the application that they test.
    let app = XCUIApplication()
    app.launch()
}
```

`XCUIApplication` contains a potent API; we will use it heavily later, which is in `launchArguments`. It helps you send a launch argument to the app to make specific customization. We will use this API heavily in the book. Before every UI test, you must launch your application, whether with launch arguments or not, which will clear the application's previously existing instance.

Querying the UI

We need to have access to the table view to count the cells inside. But how can we do this? `XCUIElementQuery` provides a class to do this. `XCUIElementQuery` is a query to locate `UIElements` so that I can assert on `UIElement` or do an interaction. Let's dig deep into how `XCUIElementQuery` works.

`XCUIElementQuery` does two main functions, relationships and filtering.

Relationships

- **Descendants:** Which will get all descendant elements under a specific `UIElement`.

For example: `View's` descendants contain all elements under `View`: `view.descendants` (Figure 3-5).

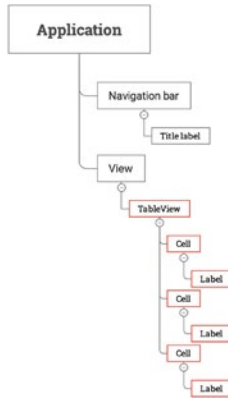


Figure 3-5. *Descendants relationship*

- Children: Which will get all elements directly below a specific UIElement. For example: TableView’s children contain all elements directly below TableView, which are cells (Figure 3-6).

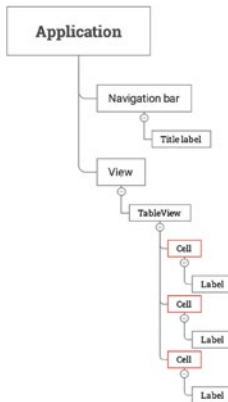


Figure 3-6. *Children relationship*

- **Containment:** Which will be helpful if `UIElement` is not unique, but it contains a unique element. For example: `cells.containing(NSPredicate(format: "label CONTAINS %@", "San Francisco"))` (Figure 3-7).

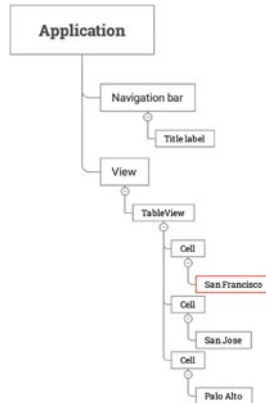


Figure 3-7. Containment relationship

Filtering

We can combine filters and relationships to be more assertive. We can filter the descendants to get only the labels under a specific `UIElement`.

`tableView.descendants(matching: .button)` will return all descendant elements under `TableView` filtered by type `button`. This is also equivalent to the following query: `tableView.buttons`. We can combine queries to build up more complex queries, for example, `app.tables.staticTexts` will get all labels under `TableView` (Figure 3-8).

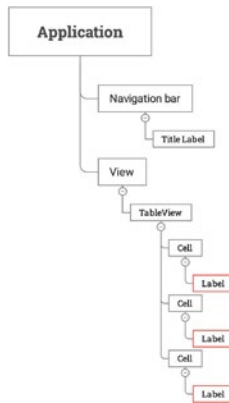


Figure 3-8. Combining relationships

You can use the query itself to be the end of the assertions, so you can check the count of cells after adding a new cell: `let query = app.tables.cells` and then `query.count`. But be careful every time you call this query; it will be evaluated again and get the most updated query result.

After using the query API, our first test will be

```

func testExample() throws {
    // UI tests must launch the application that they test.
    let app = XCUIApplication()
    app.launch()
    XCTAssertEqual(app.tables.cells.count, 6)
}
  
```

Interacting with the UI

We should use the power of `XCUIElementQuery` to find the "San Francisco" cell. First, you need to fetch all table view's descendants and to return labels only, which will be something like this: `app.tables.staticTexts`. This query will return all labels inside the table view. The

next step now is to find the label that contains "San Francisco". The query will return an array of `XCUIElements`.

`XCUIElement` is a proxy for `UIElements` in the application. Elements have types like `button`, `cell`, `staticText`, etc. They also have identifiers, which we get from an accessibility system, an accessibility identifier, or an accessibility label or title. Most of the time, we will find `UIElement` with a combination of type and identifier; for example, `let button = app.buttons["Edit"]`. We find a `UIElement` of type `button` with identifier `Edit`. Another way to query elements is to query based on the element's content. If we know that a label should display a specific text for example we can search for that label by querying its content. We can use this to find the "San Francisco" label. Also, there is another important property, which you can use to check if the `UIElement` exists or not: `element.exists`.

After asserting on the San Francisco label, our first test will be

```
func testExample() throws {
    // UI tests must launch the application that they test.
    let app = XCUIApplication()
    app.launch()
    XCTAssertEqual(app.tables.cells.count, 6)

    let cell = app.tables.staticTexts["San Francisco"]
    XCTAssertTrue(cell.exists)
}
```

Note It's very risky to depend on content when this content is dynamic or can differ from one run to another. In these cases you should always depend on accessibility identifiers.

UI Events

Once you find your element, you need to simulate user interactions. XCUIElements provides some APIs you can use to interact with UIElement:

- tap()
- doubleTap()
- press(forDuration: , thenDragTo:)
- twoFingerTap()
- swipeUp(), swipeDown(), swipeLeft(), swipeRight()
- typeText("")

After using tap API, our first test will be

```
func testExample() throws {
    // UI tests must launch the application that they test.
    let app = XCUIApplication()
    app.launch()
    XCTAssertEqual(app.tables.cells.count, 6)

    let cell = app.tables.staticTexts["San Francisco"]
    cell.tap()
}
```

Assertions

Like what we did in step 3, we need to fetch all navigation bar's descendants and to return labels only and then assert if it contains the "San Francisco" label.

After asserting on the title label, our first test will be

```

func testExample() throws {
    // UI tests must launch the application that they test.
    let app = XCUIApplication()
    app.launch()
    XCTAssertEqual(app.tables.cells.count, 6)

    let cell = app.tables.staticTexts["San Francisco"]
    cell.tap()

    let titleLabel = app.navigationBars.staticTexts["San
Francisco"]
    XCTAssertTrue(titleLabel.exists)
}

```

Value Assertion

You can assert on the value of `UIElement` using the `value` property, which varies based on the element's type. If the `UIElement` is `UISwitch`, it will be its state:

```
let genderSwitch = app.tables.switches["Gender"].value
```

Here if the switch is turned off, the value will be a string with the value "0," and the value will be "1" if the switch is turned on.

Accessibility

Application is the root of a tree of elements. All these are elements that you can access using types and identifiers. To make your life easy when UI testing, you need to make each `UIElement` unique. In a way we will repeat what we did in step 4, but we will use the accessibility identifier to get the "Say Hello !" button. Let's recall the elements hierarchy of the app.

You can add accessibility identifiers using Storyboard from the Identity Inspector by checking if Accessibility is enabled and adding an identifier (Figure 3-9) or using API `view.isAccessibilityElement = true` and `view.accessibilityIdentifier = "Hello"`.

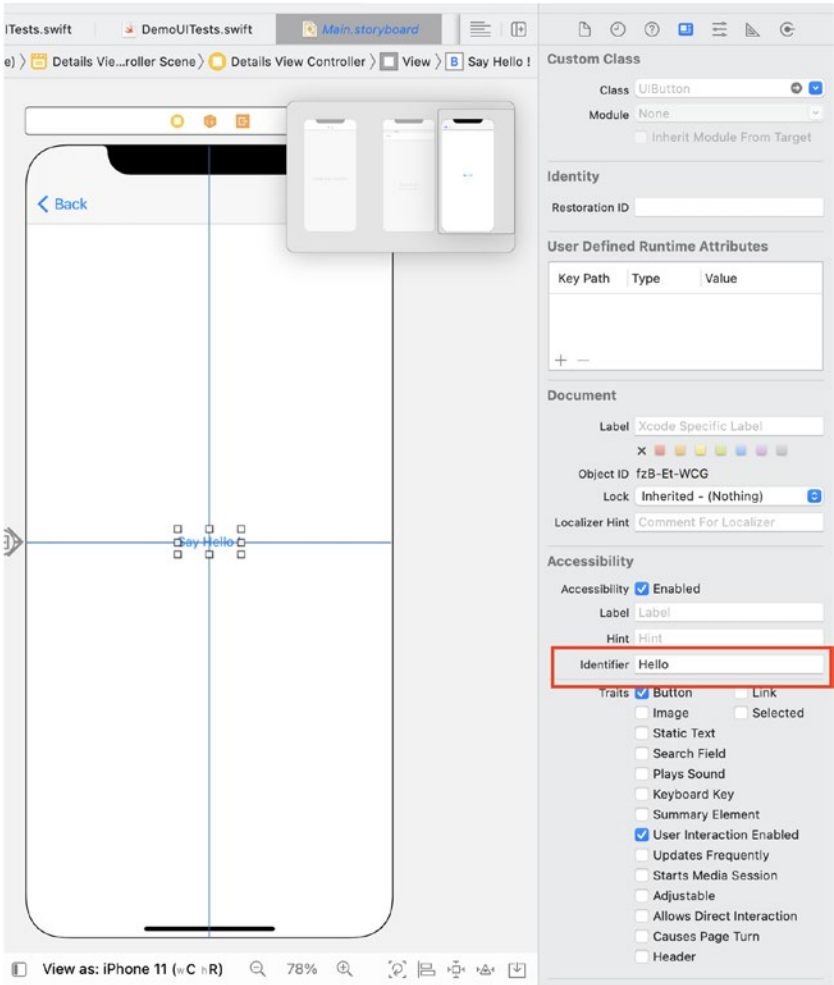


Figure 3-9. Adding an identifier

After using Accessibility to find the Hello button, our first test will be

```
func testExample() throws {
    // UI tests must launch the application that they test.
    let app = XCUIApplication()
    app.launch()
    XCTAssertEqual(app.tables.cells.count, 6)

    let cell = app.tables.staticTexts["San Francisco"]
    cell.tap()

    let titleLabel = app.navigationBars.staticTexts["San
Francisco"]
    XCTAssertTrue(titleLabel.exists)

    let helloButton = app.buttons["Hello"]
    helloButton.tap()
}
```

Accessibility Tips

- Add breakpoints inside tests (Figure 3-10) and print the description of a UIElement inside LLDB using this command: `p print(helloButton.debugDescription)`.

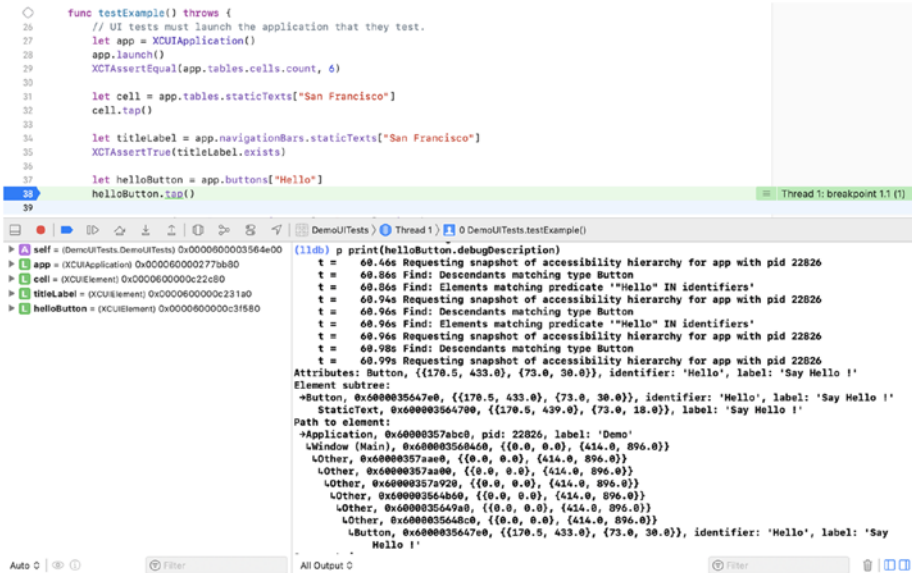


Figure 3-10. Debugging accessibility

- When you launch the Accessibility Inspector (Figure 3-11), you can touch UIElements inside the simulator to check the accessibility system’s output (Figure 3-12).

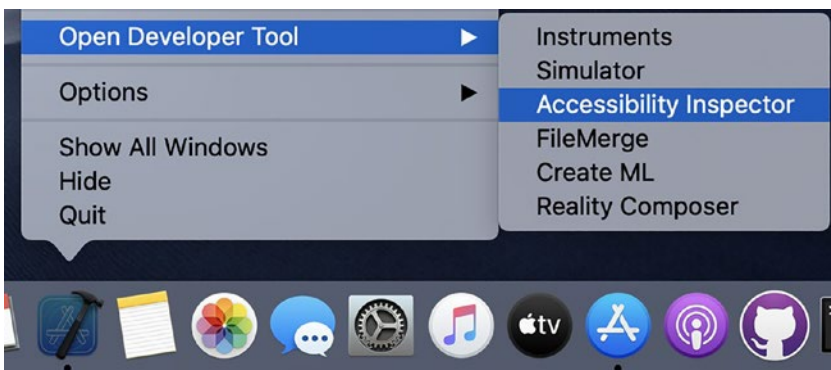


Figure 3-11. Opening the Accessibility Inspector

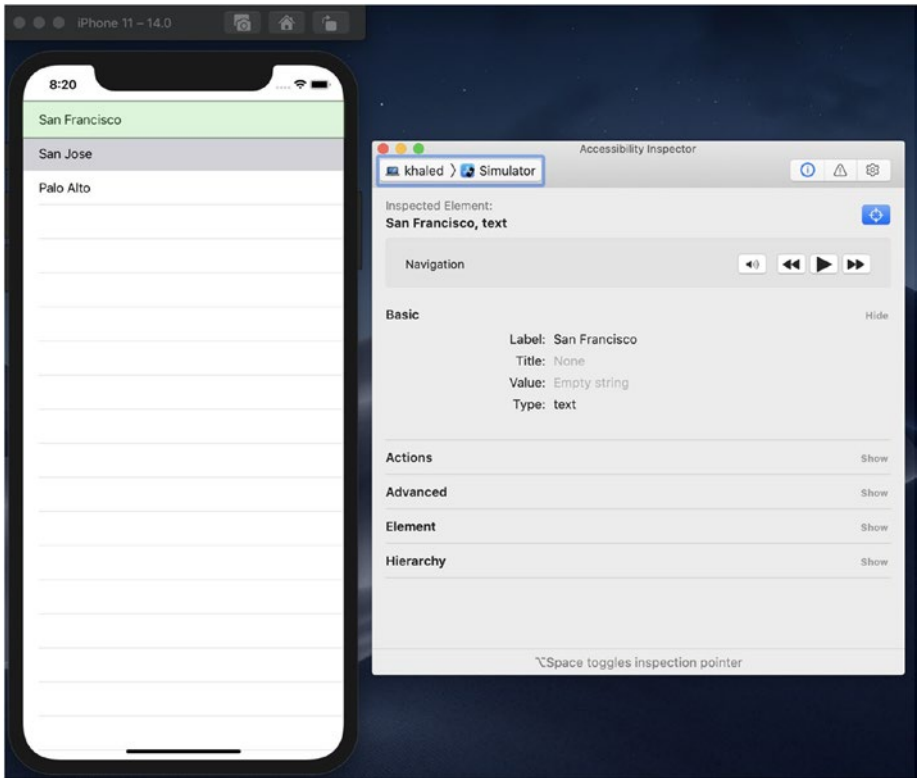


Figure 3-12. Debugging using the Accessibility Inspector

Putting It All Together

After asserting on alert content, our first test will be

```
func testExample() throws {
    // UI tests must launch the application that they test.
    let app = XCUIApplication()
    app.launch()
    XCTAssertEqual(app.tables.cells.count, 6)

    let cell = app.tables.staticTexts["San Francisco"]
    cell.tap()
}
```

```
let titleLabel = app.navigationBars.staticTexts["San
Francisco"]
XCTAssertTrue(titleLabel.exists)

let helloButton = app.buttons["Hello"]
helloButton.tap()

XCTAssertTrue(app.alerts.staticTexts["Welcome"].exists)
XCTAssertTrue(app.alerts.staticTexts["in San
Francisco"].exists)
}
```

Improve UI Tests

UI tests are much slower than normal unit tests. This is due to their nature as they directly interact with the UI the same as a normal user would. However, there are a few things to keep in mind in order to make your UI tests efficient:

- **Waiting times:** Do not use `sleep` inside your tests to wait for a specific operation because it makes your tests slower and still can make them flaky; you need to use `.waitForExistence(timeout:)`.
- **Parallel UI tests execution** starting from Xcode 10, but it's more stable on Xcode 11 (Figure 3-13).

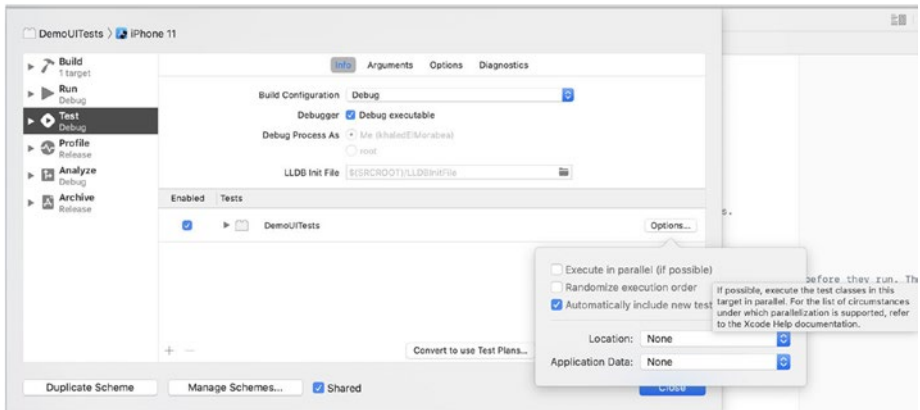


Figure 3-13. *Parallelize test execution*

Exercise

We are done with the first UI test, which navigates to the first city and taps the “Say Hello !” button. Try writing another UI test that navigates to the first city and taps the “Say Hello !” button and then goes back and navigates to the second one and taps the “Say Hello !” button.

Summary

In UI tests we interact with the app exactly as an actual user would. In this chapter we explored the basics of UI testing in iOS and how we can use the XCUITest framework to write all tests by searching for UI elements on-screen, interacting with them, and verifying the expected UI state of the app.

We used XCUIApplication to create a proxy for our app and used that proxy to launch our app. We can also use that proxy to terminate our app. After the app is launched, in order to start interacting with it, we need to access the UI elements on the screen. To search and find a certain UI element, we use the powerful XCUIElementQuery to search inside our app’s view. And by combining multiple queries together, we can reach the element we need.

When we have an element, we can either assert on its state using normal `XCTAsserts` discussed in the previous chapter, or we can interact with this element. There are multiple user interactions that we can simulate using `XCUITest`. We can tap or double-tap, we can press and hold, we can swipe in any direction, and we can even type text if applicable.

UI testing in iOS and accessibility features work hand in hand. Adding accessibility identifiers, labels, and values to your views will not only make your app accessible to people with vision, motor, learning, or hearing disabilities but will also make writing UI tests much easier. When you make your views accessible, you enable your tests to query these elements using accessibility identifiers or labels and can check on the value to verify correct behavior.

CHAPTER 4

Testing Pyramid

Now that we know how to use XCTest and XCUITest to write tests in iOS, we need to know the types of tests we should be writing, as well as the quantity of each type of testing. And this is where the “Testing Pyramid” comes in (Figure 4-1). It is a concept that helps in answering both of these questions. Mike Cohn came up with this concept in his book *Succeeding with Agile*. It’s a great visual metaphor telling you to think about different layers of testing. It also tells you how much testing to do on each layer.

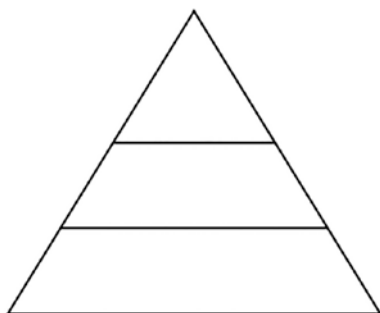


Figure 4-1. *Empty Testing Pyramid*

Instead of showing the conclusion Mike Cohn reached, we will try to deduce it by going through a few examples. In this chapter, you will be introduced to three types of testing, and we’ll implement some tests for each type. And by the end of the chapter, we will try to deduce the position of each type in the Testing Pyramid.

Our App

Let's take a look at our demo app for this chapter, **TestingPyramid**. You can find the starter project for this app in the chapter's resources. It's an extremely simple app with just two screens (Figure 4-2). The initial screen is the login screen, where the user is asked to enter their email and password. If successful, they are routed to our second screen, which is the statistics page. The statistics page shows the number of successful and failed logins since the app was installed.

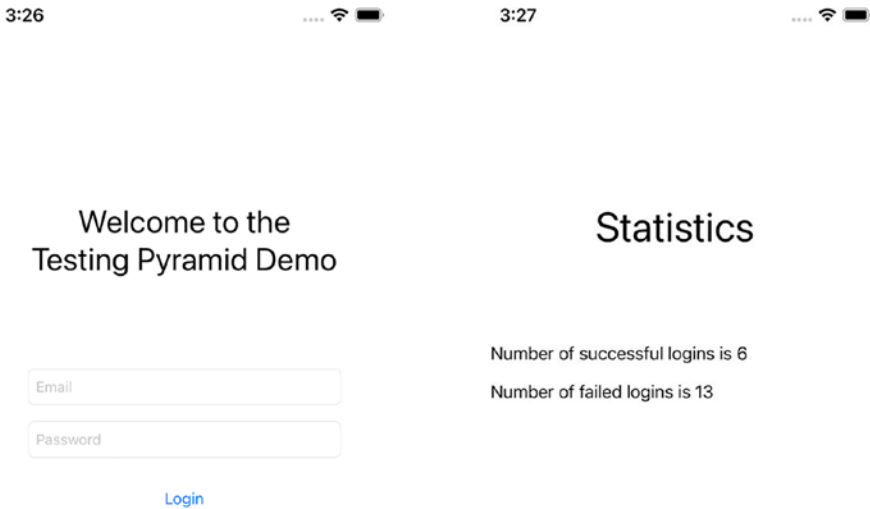


Figure 4-2. App screens

The app internally has the following components:

- **Validator:** Validates email and password
- **DatabaseManager:** Queries the local database to check if the login attempt is valid
- **PersistenceManager:** Saves the number of failed logins and successful logins in user defaults
- **LoginManager:** Responsible for executing the whole path of logging in, which is validating the entered credentials, making the network request, and updating the saved statistics based on the result

UI Tests

The first type we will explore is UI testing. When writing UI tests, we want to test two interconnected things, that our UI is displayed correctly and that the app functionalities are working as expected. When UI testing, we're seeing the app exactly how users do. We also interact with the app the same way users do. We have no access to any internal code, we can't check on network requests, and we can't query our persistence layer or write to an internal variable.

Let's take a look at some UI tests we can write for our app:

```
func testInvalidLogin() throws {
    // Initial state
    let title = app.staticTexts[AccessibilityIdentifiers.
kLoginWelcomeLabelIdentifier]
    let emailTextField = app.textFields[Accessibility
Identifiers.kLoginEmailTextFieldIdentifier]
    let passwordTextField = app.textFields[Accessibility
Identifiers.kLoginPasswordTextFieldIdentifier]
```

```

let loginButton = app.buttons[AccessibilityIdentifiers.
kLoginButtonIdentifier]

XCTAssertTrue(title.exists)
XCTAssertTrue(emailTextField.exists)
XCTAssertTrue(passwordTextField.exists)
XCTAssertTrue(loginButton.exists)

// Invalid login

loginButton.tap()

// Then
let alert = app.alerts.element
let alertExists = alert.waitForExistence(timeout: 5)
XCTAssertTrue(alertExists)
XCTAssertEqual(alert.label, "Login Error")
XCTAssertTrue(alert.staticTexts["Email can not be
empty"].exists)
}

func testValidLogin() throws {
    // Initial state
    let title = app.staticTexts[AccessibilityIdentifiers.
kLoginWelcomeLabelIdentifier]
    let emailTextField = app.textFields[Accessibility
Identifiers.kLoginEmailTextFieldIdentifier]
    let passwordTextField = app.textFields[Accessibility
Identifiers.kLoginPasswordTextFieldIdentifier]
    let loginButton = app.buttons[AccessibilityIdentifiers.
kLoginButtonIdentifier]

    XCTAssertTrue(title.exists)
    XCTAssertTrue(emailTextField.exists)

```

```

XCTAssertTrue(passwordTextField.exists)
XCTAssertTrue(loginButton.exists)

// Valid login
emailTextField.tap()
emailTextField.typeText("valid@valid.com")
passwordTextField.tap()
passwordTextField.typeText("Password!")
loginButton.tap()

// Then
let statisticsTitle = app.staticTexts[Accessibility
Identifiers.kStatisticsTitleLabelIdentifier]
let failedLabel = app.staticTexts[Accessibility
Identifiers.kFailedCountLabelIdentifier]
let successfulLabel = app.staticTexts[Accessibility
Identifiers.kSuccessfulCountLabelIdentifier]
XCTAssertTrue(statisticsTitle.exists)
XCTAssertTrue(failedLabel.exists)
XCTAssertTrue(successfulLabel.exists)
}

```

Here we test two scenarios, one where the login is successful and the other where the login is unsuccessful because the email and password are empty. And in both scenarios, we assert on the expected behavior. However, when it comes to unsuccessful login, we know that this is not the only scenario that leads to an unsuccessful login. One option is to add a UI test for each scenario where the login fails, but since UI tests are expensive in terms of execution time, it doesn't make sense to add multiple new tests, which are all almost identical. So what we will do is try to cover these scenarios within a different level in our Testing Pyramid. Another aspect in which UI tests fall short is asserting on internal changes. For example,

we would want to assert that login counts are updated when an attempted login occurs. But since we have no access to our internal code, we will need to cover this within a different level as well.

Integration Tests

For each component, we can describe it based on its level of isolation. We'll call completely isolated components (components that depend on no other components) **solitary components**. And we'll call components that depend on/integrate with other components **sociable components**. Just like in human beings, some sociable components can be more sociable than other sociable components.

Integration tests are targeted toward highly sociable components, components that integrate other smaller components together. Normally, the numbers of these components are relatively small.

There is no one rule on which components should be the subject of integration tests. You will have to use your judgement here when it comes to this. However, there are some things to consider when making this judgment. No integration tests are needed for solitary components, since they don't integrate with anything. Highly sociable components will most likely fall under the integration testing level. Components in between don't always have to be subject of integration tests. Yes, we can add integration tests for all sociable components. However, for components that are closer to being solitary than to being sociable, adding integration tests will probably not add much value and will slow up our integration test suite. Adding unit tests for these components will probably be enough. But this will ultimately depend on your judgement.

When it comes to our demo app, `LoginManager` is a highly sociable component, as it interacts with our other three components. Let's take a look at some integration tests we can write for `LoginManager`:

```
func testInvalidCredentialsLogin() {
    // Given
    let databaseManager = TestDatabaseManager() // #1
    let persistenceManager = PersistenceManager.shared
    let manager = LoginManager(databaseManager: databaseManager)

    // That
    let expectation = self.expectation(description: "Login
    finished")

    // When
    manager.login(email: "invalid", password: "invalid") {
        (success, error) in
            // Then
            XCTAssertFalse(success, "Login should not be
            successful") // #2
            XCTAssertEqual(error, ValidationError.invalidEmail.
            message, "Wrong error returned from login") // #3
            expectation.fulfill()
        }

    // Then
    self.wait(for: [expectation], timeout: 2)
    XCTAssertEqual(persistenceManager.failedLoginsCount, 1,
    "Failed login counts should be incremented") // #4
    XCTAssertEqual(persistenceManager.successfulLoginsCount, 0,
    "Successful login counts should not be incremented") // #5
    XCTAssertEqual(databaseManager.queriesCount, 0, "Database
    should not be queried") // #6
}
```

Here we wrote a test to verify that `LoginManager` interacts correctly with its dependencies. In the case of invalid credentials, we make the following assertions:

1. We create an instance of `TestDatabaseManager` which behaves the same as the normal `DatabaseManager` except it keeps record of how many queries to the database are made.
2. We assert that the login function returns a false success flag.
3. We assert that the error returned is a validation error of type “invalidEmail.”
4. We assert that the login manager asks the persistence manager to increment failed login count.
5. We assert that the login manager does not ask the persistence manager to increment successful login count.
6. We assert that the login manager does not query the database.

```
func testIncorrectCredentialsLogin() {
    // Given
    let databaseManager = TestDatabaseManager(databaseFilename:
    "testAccounts")
    let persistenceManager = PersistenceManager.shared
    let manager = LoginManager(databaseManager:
    databaseManager)

    // That
    let expectation = self.expectation(description: "Login
    finished")
```

```

// When
manager.login(email: "test@test.com", password:
"Incorrect!") { (success, error) in
    // Then
    XCTAssertFalse(success, "Login should not be
successful") // #1
    XCTAssertEqual(error, DatabaseError.credentialMismatch.
message, "Wrong error returned from login") // #2
    expectation.fulfill()
}

// Then
self.wait(for: [expectation], timeout: 2)
XCTAssertEqual(persistenceManager.failedLoginsCount, 1,
"Failed login counts should be incremented") // #3
XCTAssertEqual(persistenceManager.successfulLoginsCount, 0,
"Successful login counts should not be incremented") // #4
XCTAssertEqual(databaseManager.queriesCount, 1, "Database
should be queried") // #5
}

```

For the second test, we look at the case of incorrect credentials, and we make the following assertions:

1. We assert that the login function returns a false success flag.
2. We assert that the error returned is a database error of type “credentialMismatch.”
3. We assert that the login manager asks the persistence manager to increment failed login count.
4. We assert that the login manager does not ask the persistence manager to increment successful login count.

5. We assert that the login manager queries the database exactly once.

```
func testSuccessfulLogin() {
    // Given
    let databaseManager = TestDatabaseManager(databaseFilename:
        "testAccounts")
    let persistenceManager = PersistenceManager.shared
    let manager = LoginManager(databaseManager: databaseManager)

    // That
    let expectation = self.expectation(description: "Login
        finished")

    // When
    manager.login(email: "test@test.com", password: "!2345678")
    { (success, error) in
        // Then
        XCTAssertTrue(success, "Login should be successful")
        XCTAssertNil(error, "No error should be returned from
            login")
        expectation.fulfill()
    }

    // Then
    self.wait(for: [expectation], timeout: 2)
    XCTAssertEqual(persistenceManager.failedLoginsCount, 0,
        "Failed login counts should not be incremented")
    XCTAssertEqual(persistenceManager.successfulLoginsCount, 1,
        "Successful login counts should be incremented")
    XCTAssertEqual(databaseManager.queriesCount, 1, "Database
        should be queried")
}
```


Finally, for the successful login case, we make the following assertions:

1. We assert that the login function returns a true success flag.
2. We assert that no error is returned.
3. We assert that the login manager does not ask the persistence manager to increment failed login count.
4. We assert that the login manager asks the persistence manager to increment successful login count.
5. We assert that the login manager queries the database exactly once.

Now the question we need to answer is: Should we add more tests for `LoginManager`? We probably can, since, for example, we haven't covered all cases in which the initial validation will fail. So if we take a look at the first test we wrote, `testInvalidCredentialsLogin`, we could probably add multiple similar tests, each having a different validation fault and assert on the matching error. This would be an example for a new test:

```
func testInvalidCredentialsLoginEmptyEmail() {
    // Given
    let databaseManager = TestDatabaseManager()
    let persistenceManager = PersistenceManager.shared
    let manager = LoginManager(databaseManager:
        databaseManager)

    // That
    let expectation = self.expectation(description: "Login
        finished")
```

```

// When
manager.login(email: "", password: "invalid") { (success,
error) in
    // Then
    XCTAssertFalse(success, "Login should not be
    successful") // #1
    XCTAssertEqual(error, ValidationError.emptyEmail.
    message, "Wrong error returned from login") // #2
    expectation.fulfill()
}

// Then
self.wait(for: [expectation], timeout: 2)
XCTAssertEqual(persistenceManager.failedLoginsCount, 1,
"Failed login counts should be incremented") // #3
XCTAssertEqual(persistenceManager.successfulLoginsCount, 0,
"Successful login counts should not be incremented") // #4
XCTAssertEqual(databaseManager.queriesCount, 0, "Database
should not be queried") // #5
}

```

If you look closely at the preceding test, you will find that it's almost a duplicate of our first test. And we could also add five more duplicate tests, each with a different error. But the problem with these tests is that they will all be performing the exact same checks (checks related to `PersistenceManager` and `DatabaseManager`) over and over again, meaning that if one of the duplicates passes or fails when it comes to `PersistenceManager` or `DatabaseManager` checks, all other tests will for sure behave the same way. So the only value from them is testing the `Validator`, since it's the only variable among them. Once we spot this problem, we can safely deduce that these tests should not be here in the integration test level, which brings us to our third and final level: unit tests.

Unit Tests

Before we talk about unit tests, we need to first define what a unit is. This is not a fairly easy thing to answer. However, there has been a general consensus that when it comes to object-oriented languages (Swift), every class is considered a “unit.”

When testing a specific class, we should at least test the public interface of the class. Unit tests should cover the happy scenarios as well as edge cases.

Unit tests run in a high degree of isolation, meaning each unit should be tested to ensure that it’s working properly on its own. This means that if a unit depends on another component, this component needs to be stubbed. We will talk about stubbing and mocking in detail later in Chapter 7. Due to this high degree of isolation, unit tests are the fastest type of tests we will write.

When it comes to our demo app, we will need to add unit tests for `Validator`, `PersistenceManager`, and `DatabaseManager`. Let’s take a look at unit tests that we can write for `Validator`. In our tests for `LoginManager`, we went through scenarios in which the validation failed and asserted that the error returned from the login function is equal to expected validation error. And we also went through scenarios where the validation passed. But for `Validator` tests, we will cover all possible scenarios when it comes to validating our credentials:

```
// Test validating a valid credential
func testValidCredentials() {
    // Given
    let validator = Validator()
    let credentials = Credentials(email: "valid@valid.com",
    password: "Password!")
```

CHAPTER 4 TESTING PYRAMID

```
// When
let result = validator.validateCredentials(credentials)

// Then
XCTAssertTrue(result.success)
XCTAssertNil(result.error)
}

// Test validating an invalid credential with empty email
func testEmptyEmail() {
    // Given
    let validator = Validator()
    let credentials = Credentials(email: "", password:
    "Password!")

    // When
    let result = validator.validateCredentials(credentials)

    // Then
    XCTAssertFalse(result.success)
    XCTAssertEqual(result.error, .emptyEmail)
}

// Test validating an invalid credential with invalid email
func testInvalidEmail() {
    // Given
    let validator = Validator()
    let credentials = Credentials(email: "invalid", password:
    "Password!")

    // When
    let result = validator.validateCredentials(credentials)

    // Then
    XCTAssertFalse(result.success)
}
```

```
    XCTAssertEqual(result.error, .invalidEmail)
}
// Test validating an invalid credential with long email
func testTooLongEmail() {
    // Given
    let validator = Validator()
    let email = randomString(100) + "@valid.com"
    let password = "Password!"
    let credentials = Credentials(email: email, password:
password)

    // When
    let result = validator.validateCredentials(credentials)

    // Then
    XCTAssertFalse(result.success)
    XCTAssertEqual(result.error, .tooLongEmail)
}

// Test validating an invalid credential with empty password
func testEmptyPassword() {
    // Given
    let validator = Validator()
    let credentials = Credentials(email: "valid@valid.com",
password: "")

    // When
    let result = validator.validateCredentials(credentials)

    // Then
    XCTAssertFalse(result.success)
    XCTAssertEqual(result.error, .emptyPassword)
}
```

CHAPTER 4 TESTING PYRAMID

// Test validating an invalid credential with short password

```
func testShortPassword() {  
    // Given  
    let validator = Validator()  
    let credentials = Credentials(email: "valid@valid.com",  
    password: "1234!")  
  
    // When  
    let result = validator.validateCredentials(credentials)  
  
    // Then  
    XCTAssertFalse(result.success)  
    XCTAssertEqual(result.error, .tooShortPassword)  
}
```

// Test validating an invalid credential with long password

```
func testLongPassword() {  
    // Given  
    let validator = Validator()  
    let email = "valid@valid.com"  
    let password = randomString(41)  
    let credentials = Credentials(email: email, password:  
    password)  
  
    // When  
    let result = validator.validateCredentials(credentials)  
  
    // Then  
    XCTAssertFalse(result.success)  
    XCTAssertEqual(result.error, .tooLongPassword)  
}
```

```

// Test validating an invalid credential with password having
no special
//characters
func testNoSpecialCharacterPassword() {
    // Given
    let validator = Validator()
    let credentials = Credentials(email: "valid@valid.com",
password: "12345678")

    // When
    let result = validator.validateCredentials(credentials)

    // Then
    XCTAssertFalse(result.success)
    XCTAssertEqual(result.error, .noSpecialCharacters)
}

```

For our Validator component, we cover all the possible scenarios when it comes to validation logic. We have a high degree of freedom when adding unit tests, since unit tests are the least expensive type of tests. So all the scenarios we chose not to cover with UI or integration tests, we can cover them in this level.

Summary

In the unit tests, we tested the isolated functionality of the validator, network, and persistence. In the integration tests, we tested a special component (LoginManager) that basically integrates all our components together, and by these tests, we made sure that our units integrate correctly with each other. And in our UI tests, we also tested the integration of all our components, as well as testing that our UI is working properly.

Your unit tests make sure that a certain component works as intended. When testing a component, we carry out the test with complete isolation from other components. The number of unit tests in your test suite will largely outnumber any other type of test, and thankfully they are also the fastest type of testing.

Integration tests are targeted toward components that link and integrate other smaller components together. They make sure that smaller components work together as expected. Without integration tests, many bugs can be left unnoticed even if you have very high coverage with your unit tests. And when it comes to speed, integration tests are slower than unit tests.

Your UI tests make sure that the UI behaves correctly and that the app's main functionalities are working. When it comes to UI testing, we can actually say that it's a very high-level type of integration testing. You're seeing the app exactly how users do—there's no special internal knowledge of how your code is structured as we get with unit and integration tests, and you can't add mocks or stubs to isolate specific functionality. Without UI tests, you will have no guarantee that your app works as expected, as this tests the app the same way your user does. Because this type of testing deals with the UI, it is the slowest type of testing.

And this brings us to our Testing Pyramid (Figure 4-3), now populated with three equally important levels of testing.

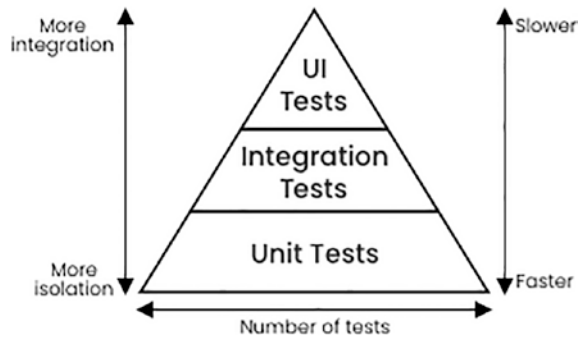


Figure 4-3. *Final Testing Pyramid*

The Testing Pyramid serves as a good rule of thumb when it comes to establishing your own test suite. Stick to the pyramid shape to come up with a healthy, fast, and maintainable test suite: write lots of small and fast unit tests. Write some more integration tests for your sociable components and very few high-level tests that test your application from end to end.

CHAPTER 5

TDD Deep Dive

So far you have been introduced to the basics of testing and TDD. We have also utilized TDD to implement somewhat simple examples. This chapter aims to take this to an even further step. One of our goals in writing this book was to show you the whole experience of test-driven software development. We want to show you how TDD fits in different types of projects and not just simple examples. We will start implementing this project from scratch and keep adding a small piece of code incrementally and safely using TDD, until we finish the project together.

CoffeePot

Have you ever found yourself standing in line at your favorite coffee shop struggling to understand the difference between the vast variety of options and then just ending up ordering the one coffee order you know by heart? Even if you are a coffee buff now, there must have been a time when you were still a coffee newbie. Here comes CoffeePot. It is all coffee newbies' best friend. **CoffeePot** is an app aimed at helping you understand the differences between all types of coffee, as well as different ways of preparing coffee. You can think of it as an ultimate coffee guide. By the end of this chapter, we will have **CoffeePot** up and running, ready to assist with any coffee order. This app is heavily inspired by [this article](#) from **Taste of Home**, and it's where we got our data.

Eye on the Big Picture

The golden rule for tackling any project/problem is granularity; you can't complete a project in one go. You have to break it into tiny chunks and finish them one by one. The key is how to add a tiny chunk and make sure that it is being integrated correctly and does not break the previous features. Each chunk should be significant and concrete enough that you can tell when it's done and small enough to be focused on one concept and achievable quickly. Dividing our work into small, coherent chunks also helps to manage the development risk.

Granularity (Figure 5-1) is powerful, but you need to keep your eyes on the big goal or get lost, which is finishing the project. So, when we start implementing a new feature, we start with acceptance tests, which exercise the functionality we want to build end to end; when the acceptance test fails, it's an indication that we are not done yet. When it passes, we are done. When implementing a new feature, the test loop is a measure of our progress, and the growing test suite of tests protects us against regression of failures when we change the system moving forward. Also, we need to keep the code as simple as possible, making it easier to understand and modify. Always remember: developers spend more time reading code than writing it. So that's what we should optimize for. Inside TDD, we can continuously refactor our code to simplify and improve the design. The test suites in the feedback loop protect us from mistakes.

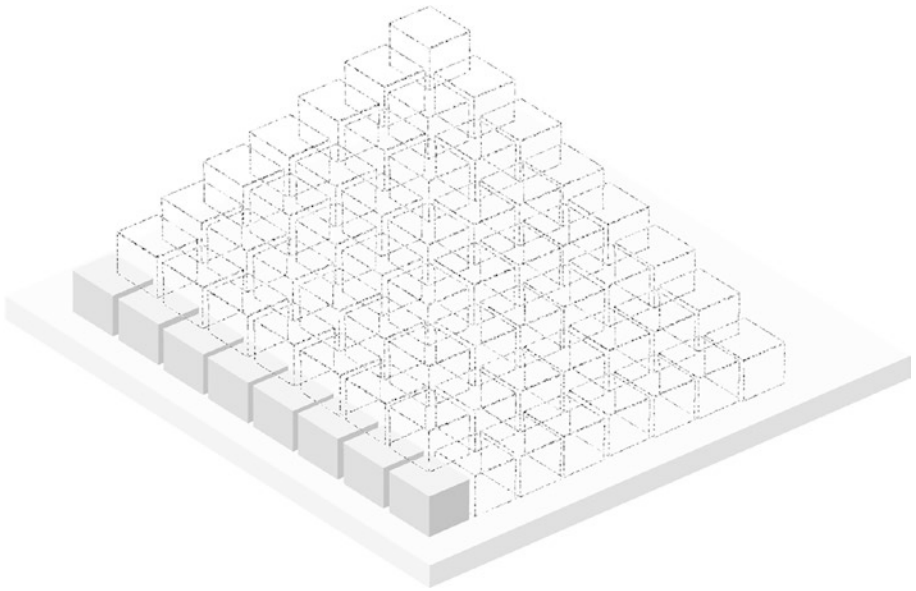


Figure 5-1. Granularity visualized

Requirements

Let's begin with user stories:

If you are not familiar with user stories, a user story is a general explanation of a software feature written from the perspective of the end user.

1. As a user, I want to know all types of hot and cold coffee drinks, including a picture of the coffee drink.
2. As a user, I want to tap any coffee drink type to show more details about this drink, including a picture of the coffee and a brief description of ingredients.
3. As a user, I want to know all types of coffee machines, including a picture of the coffee machine.

4. As a user, I want to tap any coffee maker type to show more details about this machine, including a picture of the machine and a brief description of how to use it.

Note All required data are inside the start project as a plist file.

Project wireframes (Figure 5-2):



Figure 5-2. Wireframes

Testing Pyramid

As mentioned in the previous chapter, we have three types of testing; each one is doing a specific task or answering a particular question. In the unit tests, we test the isolated functionality of each class; do our objects do the right thing? In the integration tests, we test components that integrate a group of other components; do our objects work with each other correctly? And in our UI tests, we test the system end to end; does the whole system work? We will use all three testing levels while implementing this project. And we'll see how we can combine the Testing Pyramid concept with a TDD implementation approach.

A user story is the smallest feature that can add value to a user on its own. We will work on user stories one by one. Although the user story is minimal, we cannot implement it in one go; we need to break it into tiny chunks and finish these chunks one by one. Our strategy (Figure 5-3) to finish each user story is writing a failing end-to-end test, and then we will design our user story using a set of integration tests. Integration tests will define how our objects will communicate with each other; after that, we will go through each object and write a failing unit test that will describe how this object will do its job. Our integration tests will pass by making all failing unit tests pass.

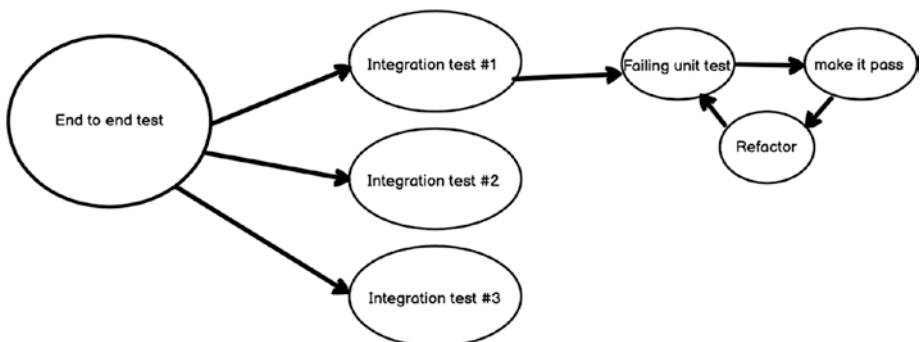


Figure 5-3. Testing plan diagram

First Story

“As a user, I want to know all types of hot and cold coffee drinks, including a picture of the coffee drink.”

Let’s open up the starter project, which can be found in this chapter’s resources. Firstly, we will need to write a failing end-to-end test that validates that the coffee drinks view shows all coffee drinks (Figure 5-4). When this end-to-end test passes, this will indicate that we finished this story.

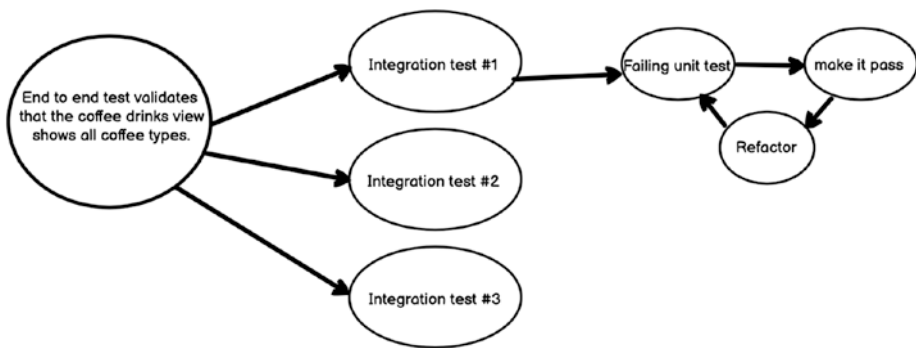


Figure 5-4. Testing plan diagram (end-to-end test added)

Let’s write our first test, which will be a UI test:

```

func testShowsAllCoffeeDrinks() {
  let app = XCUIApplication()
  app.launchEnvironment = ["coffee_drinks_stubbed": "coffee_
  drinks_stub"]
  app.launch()

  let coffeeCollectionView = app.collectionViews

```

```

XCTAssertTrue(coffeeCollectionView.cells["coffee1"].exists,
"Failed to show the first coffee item in plist")
XCTAssertTrue(coffeeCollectionView.cells["coffee2"].exists,
"Failed to show the second coffee item in plist")
}

```

Here we wrote our first end-to-end test. We set up our app using launch arguments, which we will discuss in detail later on in this chapter (see section “[CoffeeDrinksDataSource](#)”). And then we assert that the data is displayed correctly inside our collection view.

Architecture

First, let’s talk about object-oriented design before making the end-to-end test pass. Object-oriented design focuses more on the communication between modules and communication between objects inside these modules rather than the object itself. An object communicates by messages: it receives messages from other objects and reacts by sending messages to other objects, returning a value to the original sender. An object must do a single task. This lets us change the system’s behavior by changing objects’ composition—adding and removing instances, plugging different modules together.

We now need to design how our objects will interact under the hood to deliver the required story. There are multiple patterns we can apply, patterns like MVC, MVP, MVVM, and many more. All these design patterns help in developing applications that are loosely combined and easy to test and maintain. These patterns always aim to divide the application into distinct component groups, each group carrying a specific aspect of the application. In this project, we will use simple MVP.

MVP

The Model View Presenter (MVP) architecture pattern (Figure 5-5) separates the data model from a view through a presenter.

1. **The view**

A view component in MVP contains a visual part of the application.

It contains only the UI, and it does not contain any logic or knowledge of the data displayed. It also handles any interaction a user may have with the screen and directs it to the presenter.

2. **The presenter**

The presenter is a layer that connects models and views. It triggers the business logic and tells the view when to update. It interacts with the model and fetches and formats data from the model to update the view.

3. **The model**

This contains a data provider, the code to fetch and update, the data and the business logic. Usually, this data is fetched from the network or a local database.

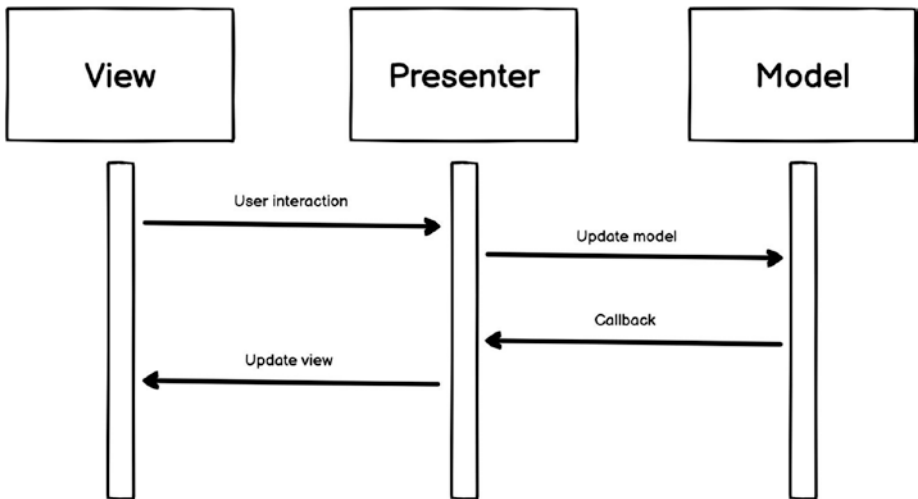


Figure 5-5. MVP design pattern

First Integration Test

Integration tests are mainly responsible for how our objects integrate and communicate with each other. The integration test allows us to think about the design first and how all objects will do their job and interact inside the system. As mentioned in Chapter 4, we write integration tests for highly sociable components. By applying the MVP design pattern on the logic we need in the first story, we'll find that our presenter is considered a sociable component. Our design can look something like Figure 5-6.

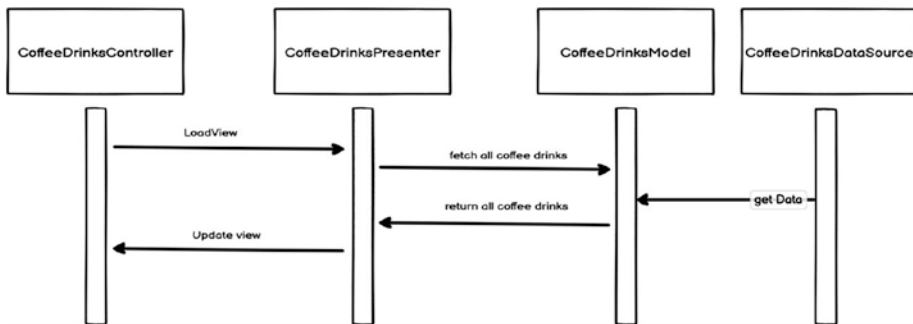


Figure 5-6. MVP applied

If we convert the diagram into code, once `CoffeeDrinksController` is loaded, we will initialize `CoffeeDrinksPresenter`, which will take `CoffeeDrinksModel` inside the constructor. `CoffeeDrinksPresenter` will contain a method that will fetch all coffee drinks and abstract the communication to `CoffeeDrinksModel` under the hood; then, the model will return the drinks. Last, `CoffeeDrinksPresenter` will update the view. Converting this to a test will be something like the following:

```

func testFetchingAllCoffeeDrinks() {
  //Given
  let expectedDrinks = ""
  [
    {
      "name": "coffee1",
      "image_name": "black",
      "desc": "desc1"
    },
    {
      "name": "coffee2",
      "image_name": "black",
      "desc": "desc2"
    }
  ]
}

```

```

"""
let coffeeDrinksDataSource = CoffeeDrinksDataSourceStub(stu
bbledDataJSON:expectedDrinks)
let coffeeDrinksModel = CoffeeDrinksModel(source:
coffeeDrinksDataSource)
let coffeeDrinksPresenter = CoffeeDrinksPresenter(model:cof
feeDrinksModel)

// when & then
XCTAssertEqual( coffeeDrinksPresenter.getDrinksCount(), 2)

XCTAssertEqual(coffeeDrinksPresenter.getDrinkName(index:0),
"coffee1")
XCTAssertEqual(coffeeDrinksPresenter.
getDrinkImageName(index:0), "black")

XCTAssertEqual(coffeeDrinksPresenter.getDrinkName(index:1),
"coffee2")
XCTAssertEqual(coffeeDrinksPresenter.
getDrinkImageName(index:1), "black")
}

```

The chart status now will be something like Figure 5-7.

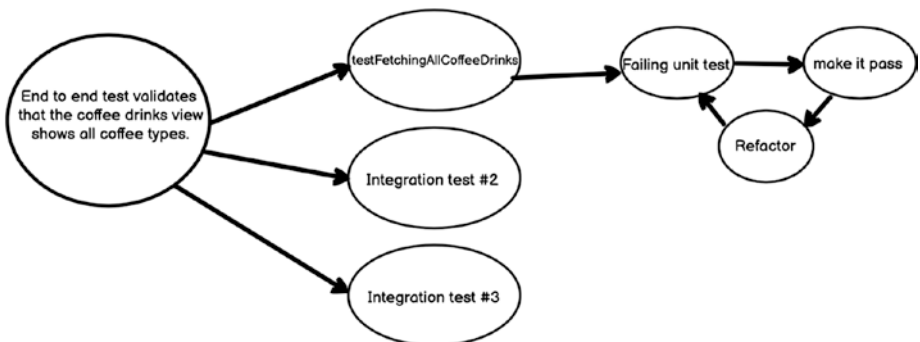


Figure 5-7. Testing plan diagram (integration #1 test added)

Unit Tests

If you run the integration test, it will definitely fail. We need to go through each object and start implementing it using unit tests until we make the integration test pass. We will write a failing unit test, then make it pass, and then refactor it; check Figure 5-8.

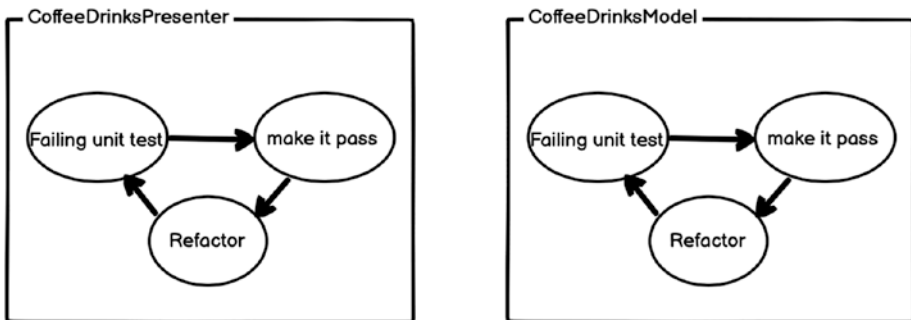


Figure 5-8. TDD cycle on units

CoffeeDrinksDataSource

We'll start with our smallest unit, which is `CoffeeDrinksDataSource`. It's an object that has the sole responsibility of reading a plist file from disk and returning it in the form of `Data`. Due to its nature, we'll find that writing a test for it will be basically duplicating the implementation code. This is an example of the very rare cases where we encounter a class that doesn't need to be tested. But at the same time, we can't inject this logic into another class because we'll need it to facilitate other tests. (More on that later in this chapter.)

Now let's write our class:

```
class CoffeeDrinksDataSource {
    func plistDataSourcePath() -> String? {
        var fileName = "coffee_drinks"
```

```

// UITests
if let stubbedFileName = ProcessInfo.processInfo.
environment["coffee_drinks_stubbed"] {
    fileName = stubbedFileName
}

return Bundle.main.path(forResource: fileName, ofType:
"plist")
}

public func getData() -> Data? {
    let dataPath = plistDataSourcePath()
    guard let path = dataPath, let dataArray =
NSArray(contentsOfFile:path) else {
        return nil
    }

    var data:Data?
    do {
        data = try JSONSerialization.data(withJSONObject:
dataArray)
    }catch {
        print("JSON serialization failed: \(error)")
    }

    return data
}
}
}

```

CoffeeDrinksDataSource is implemented to read data from the plist file and return data from this plist file. There is some extra logic we need to add used only for UI tests. There are times when we need our UI test code to pass some information to our mobile app, not by typing it into a text field or a user interaction but by sending it as a command-line argument

or as a launch environment/arguments. If you remember the first UI test we wrote, we needed to stub the data inside the coffee drinks view instead of depending on the actual data that may change over the application's life and can cause our test to be unreliable. Here we add the ability to stub the data returned by the data source through environment variables. We access the environment variables using `ProcessInfo.processInfo.environment`.

CoffeeDrinksModelTests

Since `CoffeeDrinksModel` depends on `CoffeeDrinksDataSource`, if we need to test it precisely, we need to exclude all these objects that `CoffeeDrinksModel` depends on and make it return expected data and assert on all public methods inside `CoffeeDrinksModel`. This is called **stubbing**.

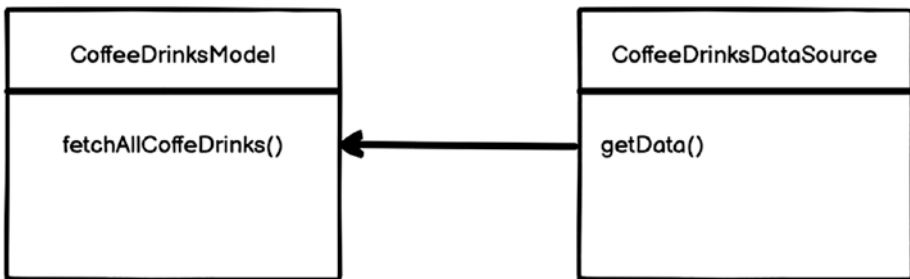


Figure 5-9. *CoffeeDrinksModel* dependency on *CoffeeDrinksDataSource*

Stubbing means creating a fake version of an object that can stand in for the real one, helping your tests run more quickly and more reliably. We will need to stub some components from here on out. We won't dive deep in this topic as we will be covering it later on in Chapter 7.

In Figure 5-9 the `CoffeeDrinksModel` class uses `CoffeeDrinksDataSource` to fetch all coffee drinks from the plist file. Testing `CoffeeDrinksModel` without stubbing `CoffeeDrinksDataSource` will be challenging and will not be reliable; in other words, if we change the data inside the plist file, this test will fail. The purpose of stubbing (Figure 5-10) is to isolate and focus on the code being tested and not on external dependencies' behavior or state. The external dependency here is `CoffeeDrinksDataSource`, which provides the data from the plist file.

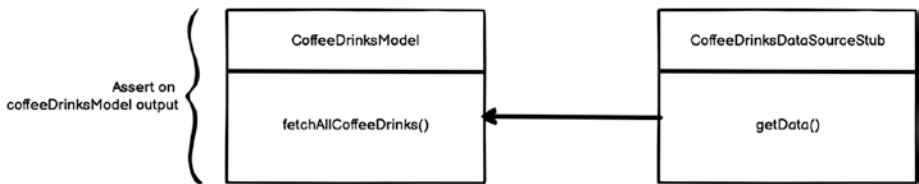


Figure 5-10. Replace the dependency with a stub object

Now let's write our stub object:

```
class CoffeeDrinksDataSourceStub: CoffeeDrinksDataSource {
    var stubbedDataJSON: String?

    init(stubbedDataJSON: String){
        self.stubbedDataJSON = stubbedDataJSON
    }

    public override func getData() -> Data? {
        let jsonData = self.stubbedDataJSON?.data(using: .utf8)
        return jsonData
    }
}
```

`CoffeeDrinksDataSourceStub` will take the expected data in its constructor and return it as data inside the `getData()` function. So no logic, and we can test `CoffeeDrinksModel` separately.

Let's now write tests for `CoffeeDrinksModel` using the newly created stub object:

```
func testFetchingAllCoffeeDrinks() {
    //Given
    let expectedDrinks = ""
    [
        {
            "name": "coffee1",
            "image_name": "black",
            "desc": "desc1"
        },
        {
            "name": "coffee2",
            "image_name": "black",
            "desc": "desc2"
        }
    ]
    ""
    let coffeeDrinksDataSource = CoffeeDrinksDataSourceStub(stu
bbedDataJSON:expectedDrinks)
    let coffeeDrinksModel = CoffeeDrinksModel(source:
coffeeDrinksDataSource)

    // when
    let actualDrinks = coffeeDrinksModel.fetchAllCoffeDrinks()

    // then
    let coffeeDrink1 = actualDrinks![0]
    XCTAssertEqual(coffeeDrink1.name, "coffee1")
    XCTAssertEqual(coffeeDrink1.imageName, "black")
    XCTAssertEqual(coffeeDrink1.description, "desc1")
}
```

```

let coffeeDrink2 = actualDrinks![1]
XCTAssertEqual(coffeeDrink2.name, "coffee2")
XCTAssertEqual(coffeeDrink2.imageName, "black")
XCTAssertEqual(coffeeDrink2.description, "desc2")
}

```

After applying the TDD cycle as we slowly build our test case till we reach the preceding comprehensive test, we will end up with the following two components:

```

struct CoffeeDrink: Codable, Equatable {
    let name:String?
    let imageName: String?
    let description: String?

    private enum CodingKeys : String, CodingKey {
        case name = "name"
        case imageName = "image_name"
        case description = "desc"
    }
}

class CoffeeDrinksModel {
    private var dataSource:CoffeeDrinksDataSource?

    init(source:CoffeeDrinksDataSource?) {
        self.dataSource = source
    }

    public func fetchAllCoffeDrinks() ->[CoffeeDrink]? {
        guard let data = self.dataSource?.getData() else {
            return []
        }
    }
}

```

```

var drinks:[CoffeeDrink]?
do {
    drinks = try JSONDecoder().decode([CoffeeDrink].
        self, from: data)
} catch {
}

return drinks
}
}

```

Let's comment out the previous test inside `CoffeeDrinksIntegrationTests` and run `CoffeeDrinksModelTests`. It should pass now ✓. This will be our current status (Figure 5-11).

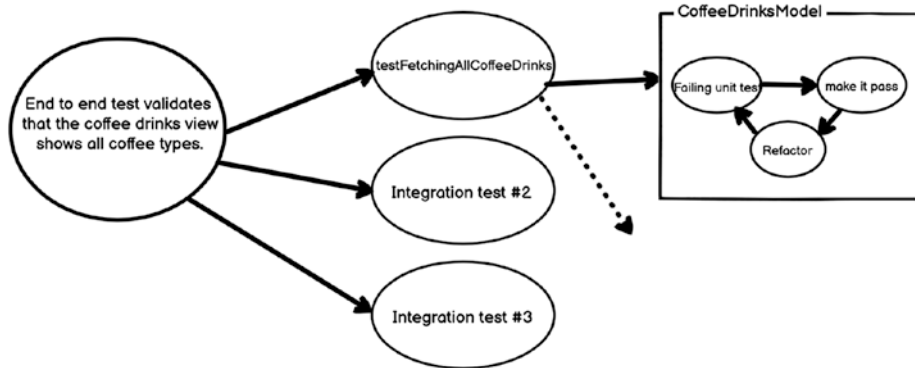


Figure 5-11. Testing plan diagram (first unit added)

CoffeeDrinksPresenterTests

We already added an integration test for our presenter, so you might think we don't need unit tests for it. But that's never the case. Integration tests can never be a substitute for unit tests. As we covered in Chapter 4, each

type serves a different purpose. We already wrote a test to validate that our presenter integrates correctly with other components. Now we need to write tests for it but in isolation.

Again `CoffeeDrinksPresenter` is dependent on `CoffeeDrinksModel`. If we need to test it, we need to stub all these objects that `CoffeeDrinksPresenter` depends on and return expected data. Here we write a stub for `CoffeeDrinksModel`, which takes the expected data in its constructor and returns it as data inside the `fetchAllCoffeDrinks()` function:

```
class CoffeeDrinksModelStub: CoffeeDrinksModel {
    var stubbedDrinks:[CoffeeDrink]?

    init(stubbedDrinks:[CoffeeDrink]) {
        super.init(source: nil)
        self.stubbedDrinks = stubbedDrinks
    }

    public override func fetchAllCoffeDrinks() ->[CoffeeDrink]? {
        return self.stubbedDrinks
    }
}
```

Now let's start writing our tests one by one:

```
func testFetchingDrinksCount() {
    //Given
    let drinks = [CoffeeDrink(name: "coffee1",imageName:
    "black",description: "desc1"),
                  CoffeeDrink(name: "coffee2",imageName:
                  "black",description: "desc2")]
    let coffeeDrinksModel = CoffeeDrinksModelStub(stubbedDrin
    ks: drinks)
    let coffeeDrinksPresenter = CoffeeDrinksPresenter(model:cof
    feeDrinksModel)
```

```

    // when & then
    XCTAssertEqual( coffeeDrinksPresenter.getDrinksCount(), 2)
}

func testFetchingDrinksName() {
    //Given
    let drinks = [CoffeeDrink(name: "coffee1",imageName:
    "black",description: "desc1"),
                  CoffeeDrink(name: "coffee2",imageName:
                  "black",description: "desc2")]
    let coffeeDrinksModel = CoffeeDrinksModelStub(stubbedDrin
    ks: drinks)
    let coffeeDrinksPresenter = CoffeeDrinksPresenter(model:cof
    feeDrinksModel)

    // when & then
    XCTAssertEqual(coffeeDrinksPresenter.getDrinkName(index:0),
    "coffee1")
    XCTAssertEqual(coffeeDrinksPresenter.getDrinkName(index:1),
    "coffee2")
}

func testFetchingDrinksImagesName() {
    //Given
    let drinks = [CoffeeDrink(name: "coffee1",imageName:
    "black",description: "desc1"),
                  CoffeeDrink(name: "coffee2",imageName:
                  "black",description: "desc2")]
    let coffeeDrinksModel = CoffeeDrinksModelStub(stubbedDrin
    ks: drinks)
    let coffeeDrinksPresenter = CoffeeDrinksPresenter(model:cof
    feeDrinksModel)

```

```

// when & then
XCTAssertEqual(coffeeDrinksPresenter.
  getDrinkImageName(index:0), "black")
XCTAssertEqual(coffeeDrinksPresenter.
  getDrinkImageName(index:1), "black")
}

```

As you know by now, after writing each test, we go and apply the TDD cycle over and over again. And after writing all the preceding tests and making all of them pass one after the other, we will end up with the following class:

```

class CoffeeDrinksPresenter {
  private var model:CoffeeDrinksModel?
  var drinks:[CoffeeDrink]?

  init(model:CoffeeDrinksModel?) {
    self.model = model
    self.drinks = self.model?.fetchAllCoffeDrinks()
  }

  public func getDrinksCount() -> Int {
    self.drinks?.count ?? 0
  }

  public func getDrinkName(index:Int) -> String? {
    guard let drink = self.drinks?[index] else {
      return nil
    }

    return drink.name
  }
}

```

```

public func getDrinkImageName(index:Int) -> String? {
    guard let drink = self.drinks?[index] else {
        return nil
    }

    return drink.imageName
}
}

```

Now we can run `CoffeeDrinksPresenterTests`, and it should pass ✓.

And we can uncomment `CoffeeDrinksIntegrationTests` and run; it should pass too. Now, the current status (Figure 5-12) is that every object is working well separately as well as working well when integrated together.

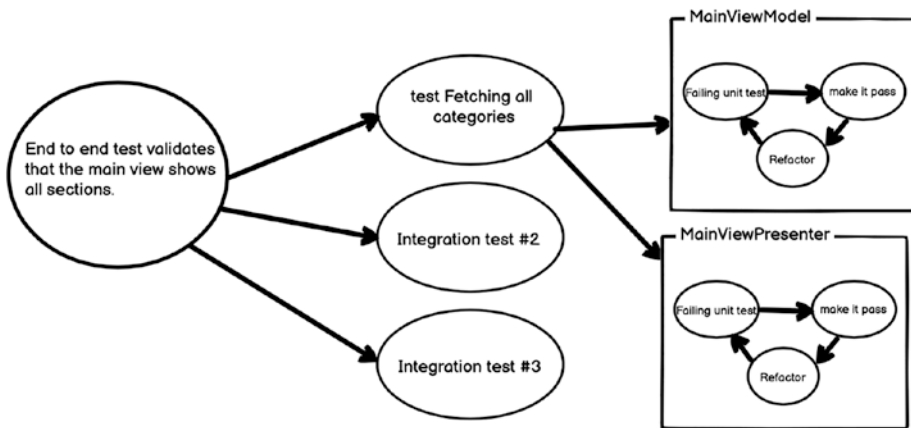


Figure 5-12. Testing plan diagram (second unit added)

Let’s now implement the last part of our feature to populate the data inside the view. After that, we need to run our end-to-end test to ensure everything is working fine. Once we see Figure 5-13, we are done with our first user story. This feature seems to be simple. We will implement the same process for the rest of the stories.

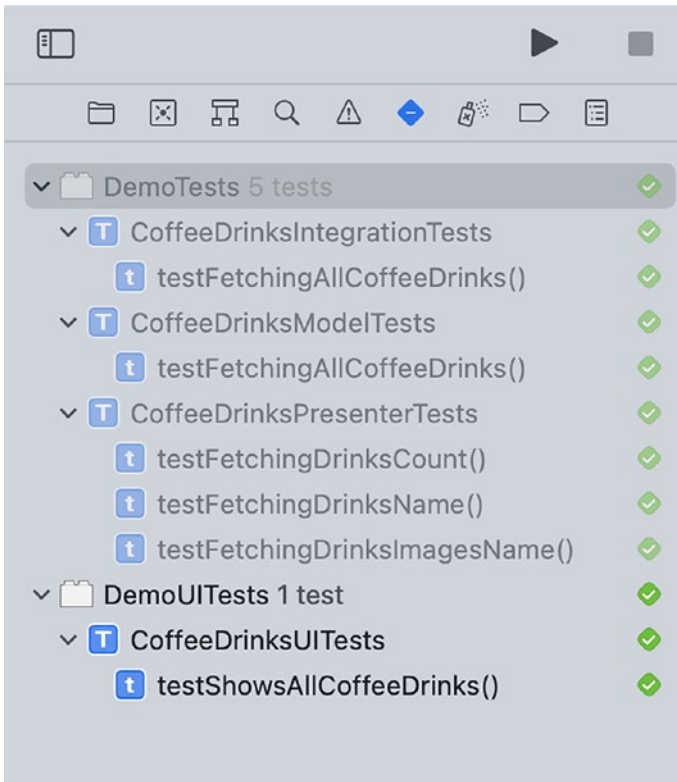


Figure 5-13. All added tests

We did not go into details on implementing the UI element of this feature, but you'll find the code in this chapter's resources.

Test Health Check

We need to validate that when our tests pass, it indicates that everything is working fine, and when they fail, we have a problem, and the problem is identified from the tests. In Figure 5-14 are all possible locations for bugs. So let's try to introduce a bug intentionally and see if our tests are able to catch it or not.

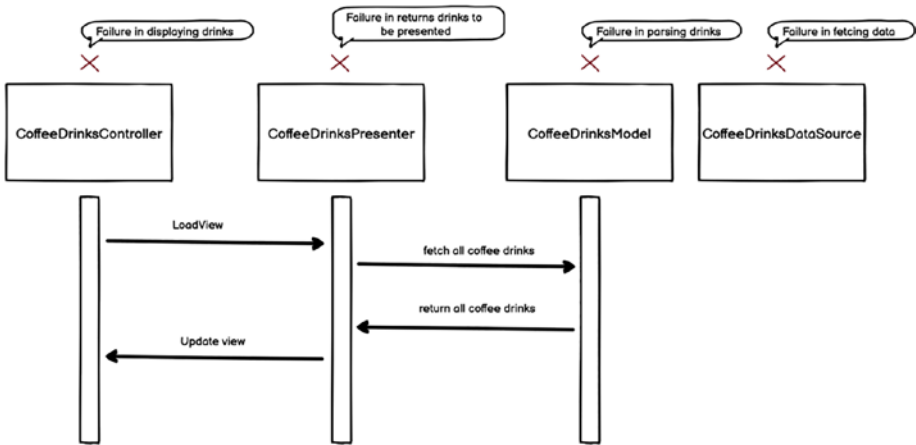


Figure 5-14. Possible bugs

Example: Let’s try to change `getDrinkName` inside `CoffeeDrinksPresenter` and make it return `imageName` instead of `name` (Figure 5-15) and run our tests.

```

7
8 import UIKit
9
10 class CoffeeDrinksPresenter {
11     private var model:CoffeeDrinksModel?
12     var drinks:[CoffeeDrink]?
13
14     init(model:CoffeeDrinksModel?) {
15         self.model = model
16         self.drinks = self.model?.fetchAllCoffeeDrinks()
17     }
18
19     public func getDrinksCount() -> Int {
20         self.drinks?.count ?? 0
21     }
22
23     public func getDrinkName(index:Int) -> String? {
24         guard let drink = self.drinks?[index] else {
25             return nil
26         }
27         return drink.imageName
28     }
29
30     public func getDrinkImageName(index:Int) -> String? {
31         guard let drink = self.drinks?[index] else {
32             return nil
33         }
34         return drink.imageName
35     }
36 }
37
38
    
```

Figure 5-15. Faulty code change

Now let's run our tests (Figure 5-16).

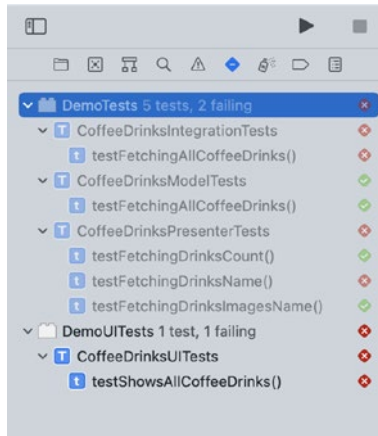


Figure 5-16. *Failing tests*

The tests were able to catch the bug successfully.

Second Story

“As a user, I want to tap any coffee drink type to show more details about this drink, including a picture of the coffee and a brief description of ingredients.”

We need to write a failing end-to-end test that validates that pressing on any coffee drink type will show details about this drink. (Figure 5-17)

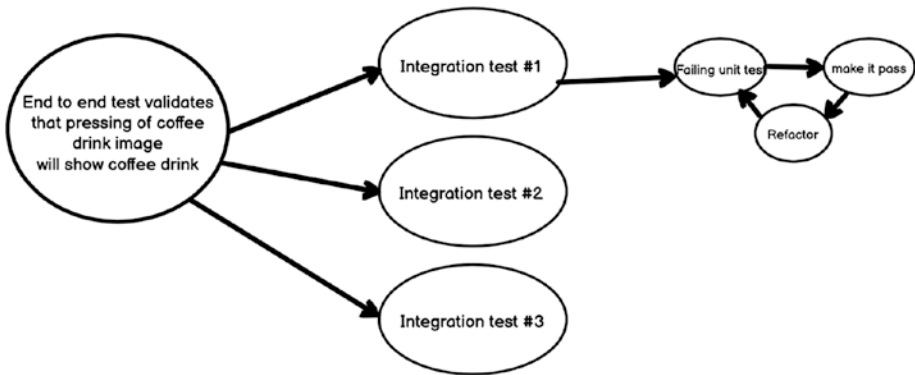


Figure 5-17. Testing plan diagram

Let's write the first test for this story:

```

func testDetailedCoffeeView() {
  let app = XCUIApplication()
  app.launchEnvironment = ["coffee_drinks_stubbed": "coffee_
  drinks_stub"]
  app.launch()

  let coffeeCollectionView = app.collectionViews
  coffeeCollectionView.cells["coffee1"].tap()

  XCTAssertTrue(app.navigationBars["coffee1"].exists)
  XCTAssertEqual(app.textViews["desc"].value as? String,
  "description1")
}

```

Here we wrote our end-to-end test for this story. We set up our app using launch arguments. Then we navigate to a specific drink page and assert that its details are correctly displayed.

Architecture

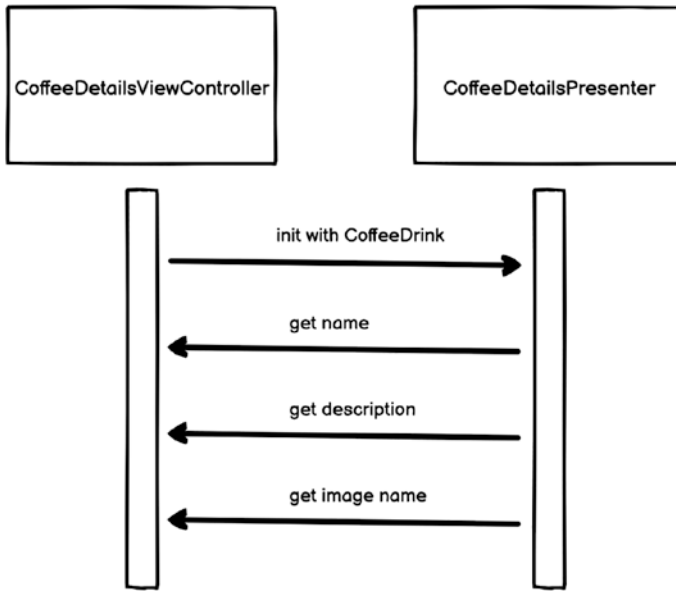


Figure 5-18. MVP applied

As we can see from Figure 5-18, there aren't too many objects integrated to deliver this story. Which basically means there aren't any sociable components to write integration tests for. So it's enough to write only unit tests for `CoffeeDetailsPresenter`.

Let's start writing our tests one by one:

```

func testFetchingDrinkName() {
  //Given
  let coffeeDetailsPresenter = CoffeeDetailsPresenter(drink:
  CoffeeDrink(name: "coffee1", imageName: "black", description:
  "desc1"))
}
  
```

```

    // when & then
    XCTAssertEqual(coffeeDetailsPresenter.getName(), "coffee1")
}

func testFetchingDrinkDescription() {
    //Given
    let coffeeDetailsPresenter = CoffeeDetailsPresenter(drink:
    CoffeeDrink(name: "coffee1",imageName: "black",description:
    "desc1"))

    // when & then
    XCTAssertEqual(coffeeDetailsPresenter.getDescription(),
    "desc1")
}

func testFetchingDrinkImageName() {
    //Given
    let coffeeDetailsPresenter = CoffeeDetailsPresenter(drink:
    CoffeeDrink(name: "coffee1",imageName: "black",description:
    "desc1"))

    // when & then
    XCTAssertEqual(coffeeDetailsPresenter.getImageName(),
    "black")
}

```

After writing all the preceding tests and making all of them pass one after the other using TDD, we will end up with the following class:

```

class CoffeeDetailsPresenter {
    private var drink:CoffeeDrink?

    init(drink:CoffeeDrink?) {
        self.drink = drink
    }
}

```

```
public func getName() -> String? {
    guard let drink = self.drink else {
        return nil
    }
    return drink.name
}

public func getImageName() -> String? {
    guard let drink = self.drink else {
        return nil
    }
    return drink.imageName
}

public func getDescription() -> String? {
    guard let drink = self.drink else {
        return nil
    }
    return drink.description
}
}
```

After adding all our tests, this is how our test suite should look like (Figure 5-19) as well as our app (Figure 5-20):

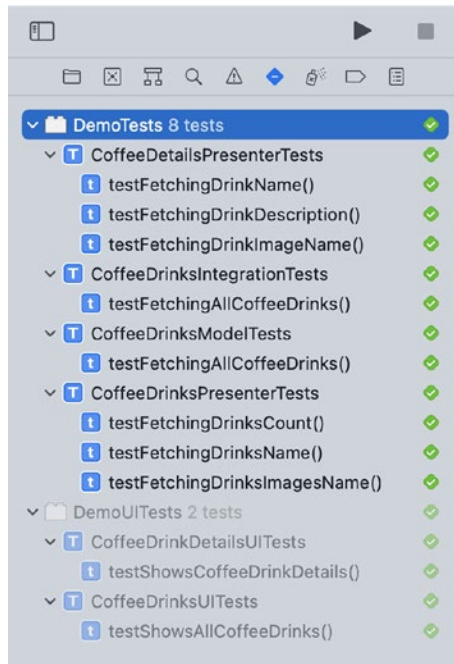


Figure 5-19. *Final test suite*

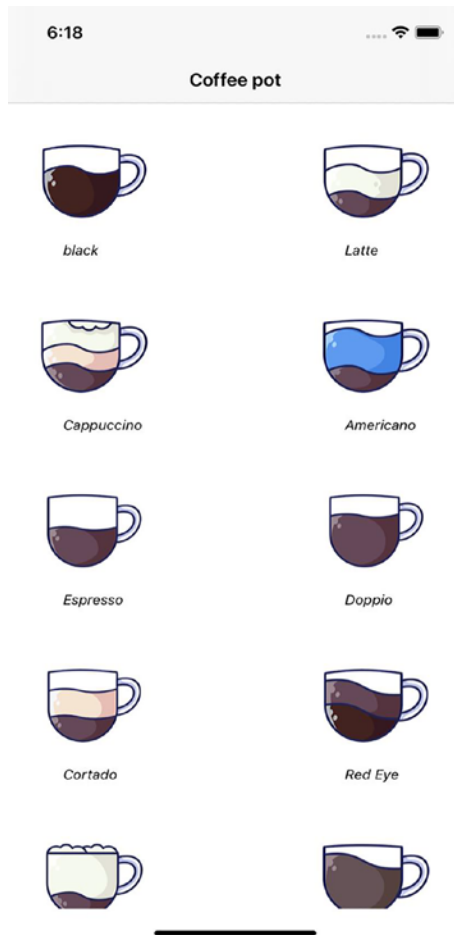


Figure 5-20. App main screen

The same as the first feature, we did not go into details on implementing the UI, but you'll find the code in this chapter's resources.

Exercise

We are done with the first and second stories. But there are still two more stories to go. You should be able to apply the same process we equipped in this chapter and implement these two stories. You can find the final project, with the first two stories implemented, in the chapter's resources.

Summary

In this chapter, we took a look at how TDD can be employed on a slightly complex project, which is a challenge similar to what you'll encounter in your day-to-day life. You got introduced to **CoffeePot**, which is an app that helps users understand the differences between different types of coffee. The app has two views: one is a view that lists all types of coffee, and the other is a detailed view for a single type of coffee.

When working on such a project, we can't aim to complete it in one go. This is both unrealistic and will have us ending up with poorly written code. The key here is granularity, where we break up our project into smaller chunks of logic and finish them one by one. TDD helps us to think in a granular manner. Since we need to always start with one failing test, which is basically a single requirement, in this case this requirement is our small chunk. And by applying the TDD cycle, we finish this chunk before thinking about the next chunk.

In order to break up our project into smaller chunks, the first step is to properly define and think thoroughly about all the project requirements. Then we take these requirements and translate them to tests. The first requirement acts as our first test, which kicks off the TDD cycle. We keep going through this cycle until we've fulfilled all the requirements we have defined.

We took our first requirement—which is viewing all the types of coffee including a picture for each type—and we wrote a UI test for that before

we even started thinking of how we would implement it. Normally this test failed since we hadn't added any code. As mentioned many times before, TDD forces us to think clearly about our design and architecture. In this case we went with a popular design pattern called Model View Presenter (MVP), which gave us a good idea of the components we'd add and how they'd interact. Since we knew how we'd design our code, we then went down a level and added an integration test. Finally we went down another level and started adding unit tests, and we just looped through the TDD cycle until all tests passed, including integration and UI tests we added at the very beginning (Figure 5-21). Our end-to-end test passing was an indication that we're done with this feature. We then took another requirement and did the same test-driven process all over again.

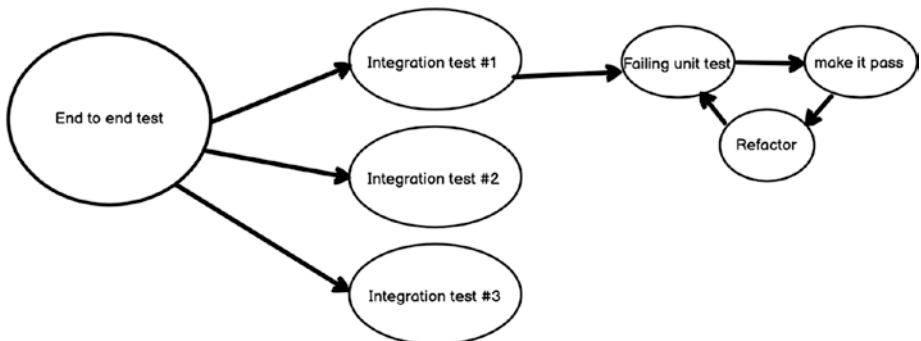


Figure 5-21. Test plan diagram used for TDD

CHAPTER 6

Modularization for the Win

Modularization is dividing the system into a number of relatively independent and interchangeable modules with well-defined interfaces, such that each one contains everything necessary to execute the desired functionality. Each one is small enough and simple enough to be thoroughly understood and well tested.

Though an extremely important design aspect, modularization typically is one of the first things that developers sacrifice when their code base grows. They may still have modules by name, but they all depend on each other and they end up with a **big ball of mud**. Which is a term used to describe software systems that lack a perceivable architecture.

Why Bother with Modularization?

From that brief definition of what modularization is, it might already seem that it's a nice-to-have characteristic in your app. But do we need it? Do we need to put that extra effort while designing the app's architecture to make sure it's properly modularized? And do we need to put that even greater effort into modularizing an existing app?

Well, one way to answer these questions is to look at how a modularized and a non-modularized app would handle the challenge

of scaling. Scaling is a process that any successful app goes through, and it basically means an increase in number of users, an increase in size of the app and number of features and functionalities inside the app, more frequent releases, and in most cases larger teams.

Let's talk about how our two types of apps can handle the scaling of their features and functionality. If we take a look at a non-modularized app and try to figure out how its components depend on and communicate with one another, we will end up with a dependency diagram that might look like the diagram in Figure 6-1. It's a dummy diagram, but it's quite realistic for a non-modularized app. A diagram like this would probably represent a simple, feature-poor app. So if you already think this diagram looks complex, then if we try to scale this said app, the diagram would most definitely turn into a chaotic mesh of nodes and edges. Sadly, the readability of an app's dependency diagram is not our only problem in this situation. If our only concern is that our diagram is not pretty, then we can just avoid looking at it. However, our real problem lies in what the diagram represents: dependencies. The more dependencies we have, the more unpredictable our app becomes.

This unpredictability becomes evident when we start adding new features in one place and end up introducing a bug or a crash in a completely different place in our app. So basically due to our complex unmanaged dependencies, when introducing a change, we would never be able to know the extent of this change's impact on our app. On the other hand, doing the same thing in a modularized app is vastly different (Figure 6-2). Due to the complete separation in our code, implementing a change means only impacting the module that we are changing. Another aspect to think about is dealing with bugs. It's definitely easier to track down a bug in an organized, structured app like our modularized app in Figure 6-2 than the one in Figure 6-1. Probably by just reading the description of the bug, we can identify which module to look at. However, in a non-modularized app, debugging bugs will be much more tedious.

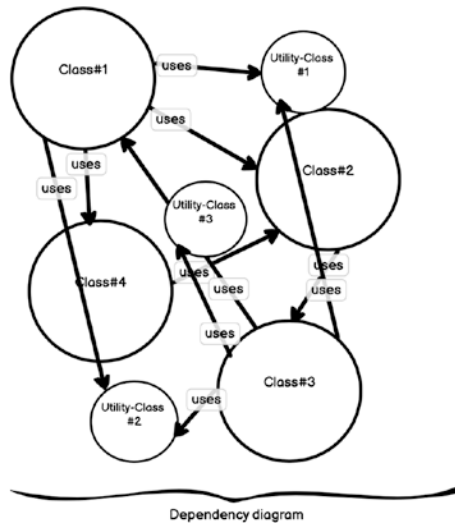


Figure 6-1. *Non-modularized app*

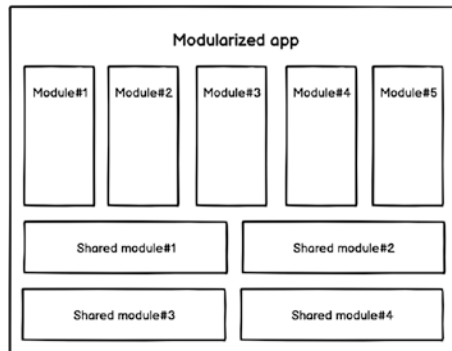


Figure 6-2. *Modularized app*

Other than the size of the app, the size of the team that manages and maintains the app can also scale. This will introduce a couple of challenges; one of them is onboarding new members. The more readable your code base is, the smoother the onboarding. Attempting to understand the code base of a non-modularized app with lots of interconnected components can be very confusing. That's why a modularized app, with

its separated design, makes it much more readable. Trying to find a part in the code responsible for a specific feature in a modularized app is as simple as finding the related module and just looking there, instead of looking through the whole code base.

Another challenge that arises with large teams is how teams collaborate with each other. In a modularized app, you can have multiple members working on different features at the same time without having to communicate with each other; that's of course given that each one is working on a different module. This simultaneous work on different modules will also rarely result in conflicts due to the separation of changes. That definitely doesn't apply to non-modularized apps, where attempting the same simultaneity would require a lot of extra effort to communicate changes across team members and solve conflicts. Another thing that's made possible by modularization is assigning code ownership. It's much easier to assign ownership of modules to certain team members or subteams.

The advantages and disadvantages of a modularized app and non-modularized app, respectively, do not only apply to applications of large scale. The advantages and disadvantages apply on apps of all sizes. However, the larger the application, the more amplified they are. The takeaway from this is that you don't need to wait for your app to scale to start thinking about modularization. You will reap a lot of benefits even if your app is of small scale. And you will set yourself for exponentially increasing benefits as your app scales in the future.

What Is a Module?

We've mentioned the word module ten times by now during this chapter, but we still haven't properly defined what a module actually is. By now you probably have an idea in mind, and you're probably not far out. But let's

agree on a proper definition. Generally speaking, a module is a standalone piece of code that provides specific and tightly coupled functionality.

While that definition makes sense, let's take a look at a real-life example to see what a module can actually look like. If you've owned an iOS device, then you've definitely used the **App Store** before. Let's take a close look at the App Store iOS application (Figure 6-3) and try to divide it into modules.

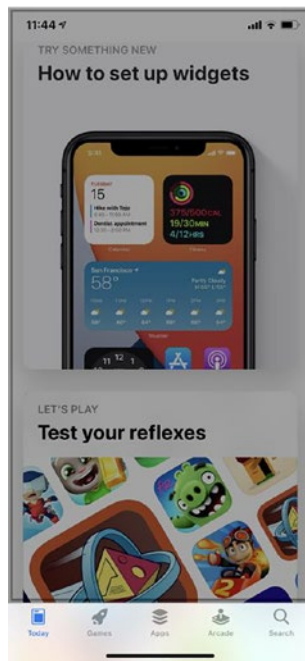


Figure 6-3. App Store app

We can split the main app into five modules; each module represents a tab inside the bottom tab bar. And we can split these main modules into way more sub-modules. So in this case a module is a group of features that provides a coupled functionality for the end user:

1. Today module
2. Games module
3. Apps module
4. Arcade module
5. Search module

Besides main modules, we will need to separate shared code into modules to be easily used across different modules. If we explore the app a little bit, we'll find that the app view in Figure 6-4 can be accessed from all our five main modules. This means that this functionality belongs to a sub-module that the five main modules use.

If we for some reason decide not to have this sub-module, then we'd have to do one of two things. Either duplicate the app page functionality in all our five main modules, which is a really bad code smell. If we do that, then whenever we need to make a change in our app page functionality, we'll need to update it in five places. And this is just the tip of the iceberg when it comes to problems with duplicating code. The other option is to implement this common functionality in one of the five modules, the Today module, for example, and have the other four modules depend on the Today module. This kind of design decisions will soon lead us to a situation much like in Figure 6-1, where we have modules with dependencies that they don't need, and might eventually lead to dependency cycles. So it's always best to separate unrelated code completely.

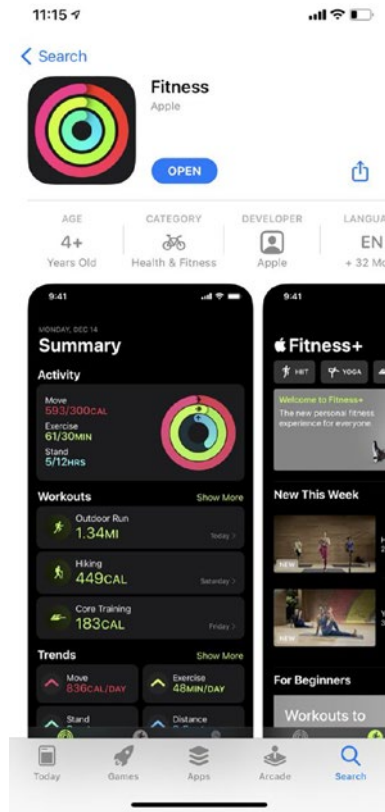


Figure 6-4. App page in App Store

Modules are not only made up of coupled features, like our five main modules or the app module; we can create modules for low-level functionalities as well, given that they are coupled together. For the App Store, we can have a module for networking, a module for analytics, and many more. The beauty of these low-level modules is that if they are written well enough, they can be reused across different apps.

So if we modularize the App Store app, it will be something like Figure 6-5.

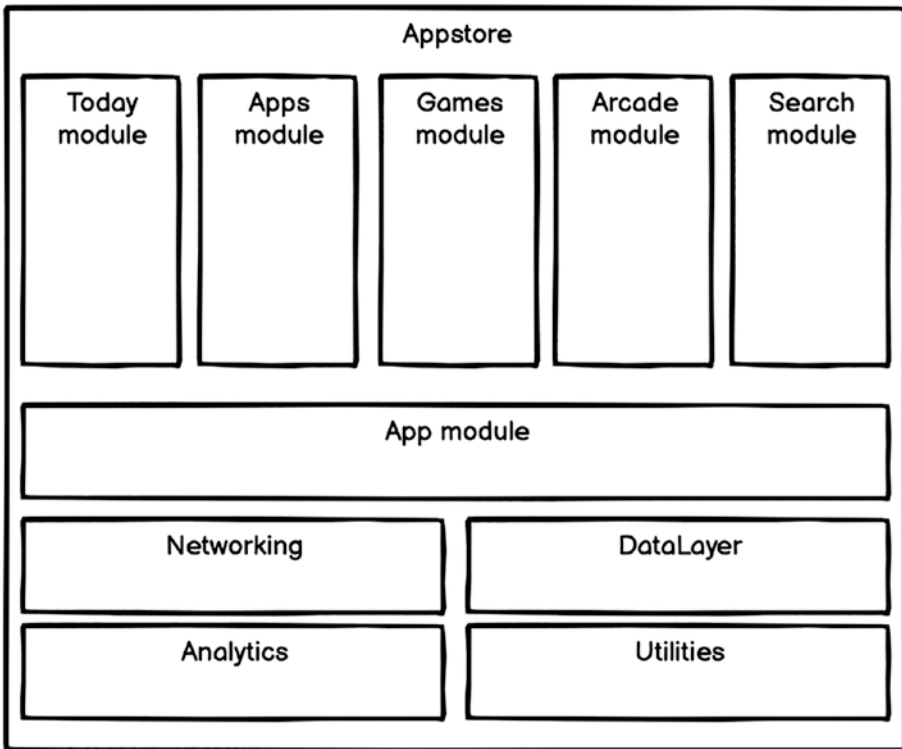


Figure 6-5. App Store module map

We should always try avoiding dependency cycles between modules, meaning we can't have module A depending on module B and module B depending on module A. Having such a cycle indicates a code smell, and we should attempt to break it by refactoring.

Modularizing Your App

When working on a brand-new app from scratch, it's always best to adopt a modularized approach while designing it. Transforming a non-modularized app to a modularized one is a costly process. And it's always

better to avoid a problem before it happens. However, if you find yourself in that position, solving this problem is not impossible. The rest of this chapter will walk you through how to tackle such a process.

When you find yourself with a non-modularized app and you want to modularize it, you have one of two options: rewrite the whole app, or refactor the app.

Rewriting the app is simple. You would basically throw most of what you have and start from scratch. This is a very aggressive approach and will require a huge and sudden investment. And due to that high level of investment needed, it comes with high risk. The rewrite time will probably end up being more than predicted, which could cause many problems. But as with most things in life, when you put in high investment and accept the high risks and all goes well, you will end up with high reward. If you go with this approach, you will start feeling the impact right away. Another thing about rewriting an app is that you will have to pause all work on new features until the rewrite is done. The alternative to pausing is duplicating the effort, as you'll have to implement new features once in an old app and once in a new rewritten app, which is quite expensive.

Though the vigorous rewriting approach has some perks, it has some pretty major drawbacks, and in most cases it's unfeasible to go that route. Luckily, we have another option, which is gradual refactoring. Contrary to the rewriting approach, it's a low-investment, low-risk, and low-impact approach. It allows us to modularize our app at our own pace without blocking the release of new features. And since the changes are of low impact, this means that so are the risks. One drawback is the slow speed of modularization, but that's completely in our hands as we can speed up or slow down based on many deciding factors.

The biggest drawback, however, is that taking this approach requires skill and following a thought-out process. Otherwise, our refactoring might lead to introducing regressions on our app. To avoid that, we need to make sure, through the use of tests, that the part of code we're refactoring is working correctly before and after refactoring. But this is not the only thing we'll use tests for. It's important to have your refactor be driven by tests just as you would while writing new code. And it's always recommended to take a step-by-step approach and not take too big steps, to avoid making breaking changes.

Introducing Books

Books is a simple app that displays the latest bestselling books (Figure 6-6). We will be working on maintaining and improving **Books** in this chapter and in following chapters as well. You can find this project in this chapter's resources. Though it might seem simple, it will showcase many issues you may encounter while working on a legacy app. For us a legacy app is an app with no tests; it's an app that can easily be broken by introducing simple changes. In the upcoming chapters, we will transform **Books** from an easy-to-break legacy app to a scalable and maintainable app.

Books depends on making requests to the New York Times API. For the app to function properly you'll need a valid API key. You can find steps on how to obtain one in the project's README. Make sure to replace all instances of "YOUR_API_KEY" in the project with the actual API key. Also make sure to replace all instances in any future snippets you will add throughout the coming chapters.

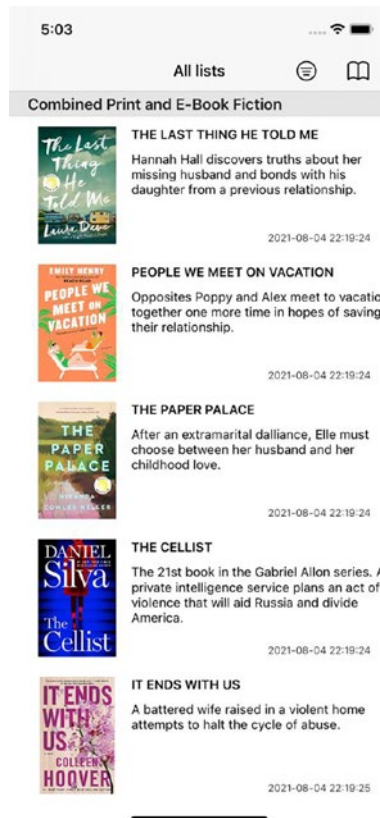


Figure 6-6. Legacy Books app

One challenge working with Books is that it does not use modern architecture. Instead, a lot of the business logic, network calls, and persistence logic exist in monolithic view controllers. For the time being, it works, as all legacy code. But as we interact with it more, you'll see just how hard and risky it is to add new things.

Our goal for this chapter is to convert this legacy monolithic app, which contains many features (Figure 6-7), into a modularized app with separated modules for each set of related features or functionalities.

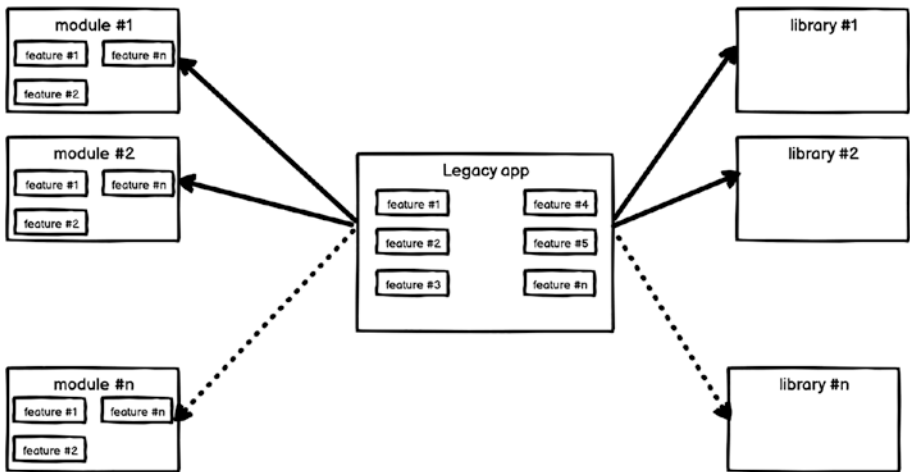


Figure 6-7. Legacy app module map

And the final result should be something like Figure 6-8.

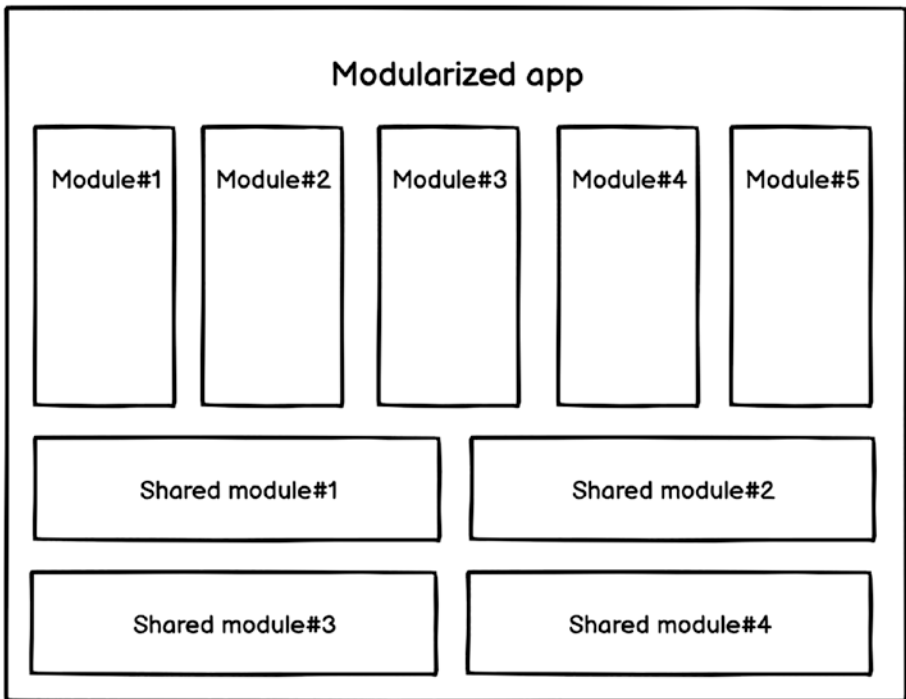


Figure 6-8. Modularized app module map

Modularization Process

- Step 0: Create initial module map
- Step 1: Choose class as a starting point
- Step 2: Identify class's responsibilities
- Step 3: For each responsibility:
 - Step 1: Add verification tests
 - Step 2: Refactor related code
 - Step 3: Rerun verification tests
- Step 4: Repeat from step 1 by choosing a new starting point

Figure 6-9. Modularization process

The preceding diagram (Figure 6-9) illustrates the process we will apply to modularize our project. It might look a bit complicated, but once we go through it step by step, you'll get the hang of it.

Initial Module Map

Step 0: Create initial module map

Step 1: Choose class as a starting point

Step 2: Identify class's responsibilities

Step 3: For each responsibility:

Step 1: Add verification tests

Step 2: Refactor related code

Step 3: Rerun verification tests

Step 4: Repeat from step 1 by choosing a new starting point

Figure 6-10. Step 0

Before we start modularizing **Books**, we will perform an exercise first. The goal of this exercise is to come up with a module map similar to the one we created for the **App Store** app (Figure 6-5). This is a one-time exercise that we'll only perform before kicking off our modularization process (Figure 6-10). We will create this map without looking at our code. Instead, we'll just start navigating our app with fresh eyes and try to group related features and functionalities together into modules. This module map will act as a guide and as a blurry goal that we're actively trying to reach through our process of modularization. However, this module map is not binding; it only acts as an initial proposed design. While we're actually in the process of modularizing the app, we might make decisions to add new modules or merge two modules together, and that's totally fine.

If we create an initial module map based on the available features and functionalities in **Books**, it will be something like that in Figure 6-11.

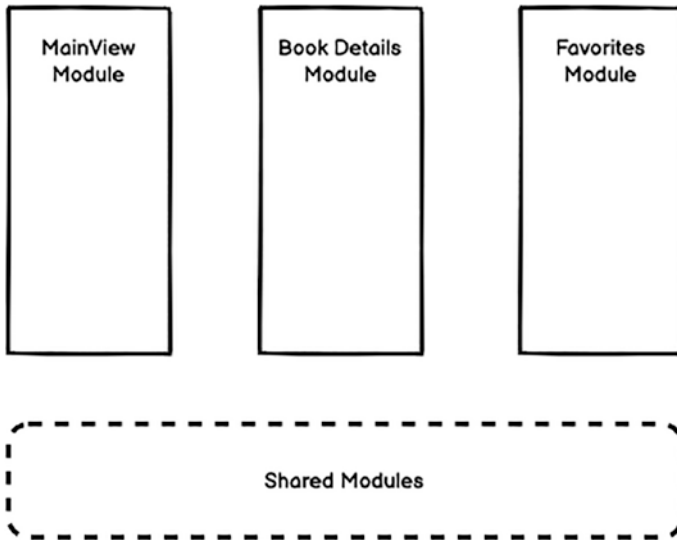


Figure 6-11. Books module map

Choose a Class as a Starting Point

- Step 0: Create initial module map
- Step 1: Choose class as a starting point**
- Step 2: Identify class's responsibilities
- Step 3: For each responsibility:
 - Step 1: Add verification tests
 - Step 2: Refactor related code
 - Step 3: Rerun verification tests
- Step 4: Repeat from step 1 by choosing a new starting point

Figure 6-12. Step 1

First thing we need to do is to pick a class to act as our starting point for the rest of the coming steps (Figure 6-12). This is a pretty trivial step, and there's really no right or wrong here. However, one thing to take into consideration is that it's better to try to look for classes with a bloated amount of responsibilities, as these tend to have higher impact when

refactored into modules. And in legacy apps like **Books** that don't follow any real design patterns, you'll find that the best starting points are usually our **ViewControllers**.

As mentioned before, the module map in Figure 6-8 can help guide us during our process. It can also help us choose our starting point. From the module map, we'll choose one module; in this case, we'll choose the **MainView Module**. And then we'll start looking for a starting point that has the most responsibilities related to that module. The best starting point in our case is **MainViewController**.

Identify the Class's Responsibilities

- Step 0: Create initial module map
- Step 1: Choose class as a starting point
- Step 2: Identify class's responsibilities**
- Step 3: For each responsibility:
 - Step 1: Add verification tests
 - Step 2: Refactor related code
 - Step 3: Rerun verification tests
- Step 4: Repeat from step 1 by choosing a new starting point

Figure 6-13. Step 2

Now that we have our starting point, we need to actually start. What we'll do next is we'll identify all the key features and functionalities that our starting point is responsible for (Figure 6-13). We do that by basically traversing the code of said class and understanding what it does. If the code is too complex and hard to understand, then we can focus on a few entry points to our code in order to make it easier to grasp the scope of responsibilities of this particular class. We need to look at all public functions, at all functions triggered when the object is created (init), and at all functions triggered either by user interactions (taps, gestures, view lifecycle events, etc.) or something else (notifications, KVO, etc.).

If you take a deep dive into what the code inside **MainViewController** does, you'll find out it can be simply represented by the diagram in Figure 6-14.

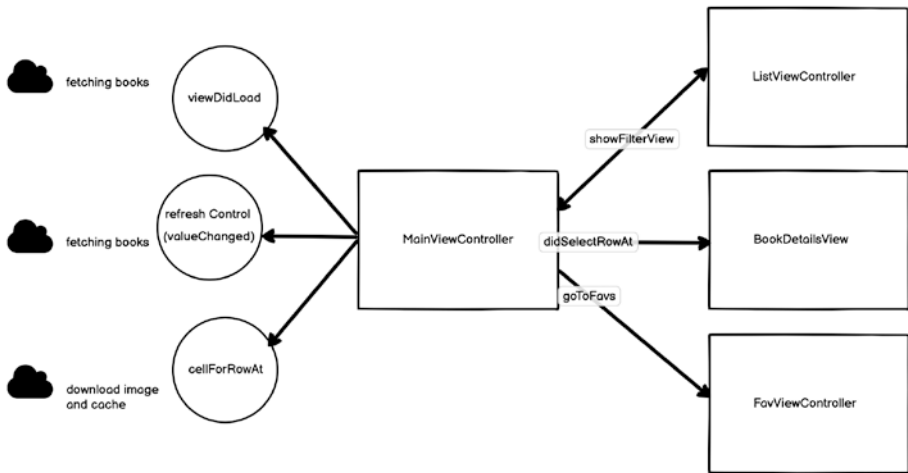


Figure 6-14. *MainViewController responsibilities diagram*

Let's formally define the key responsibilities of **MainViewController**:

1. Fetch latest books on startup.
2. Display each book in a separate cell.
3. Fetch latest books when the table is pulled down.
4. User can filter the books.
5. User can view a specific book's details.
6. User can view their favorites.

Refactor Responsibilities

Now that we've identified the responsibilities of our class, it's time to start refactoring. For each responsibility, we're going to do the following steps:

1. Add verification tests.
2. Refactor related code.
3. Rerun verification tests.

Verification Tests

Step 0: Create initial module map
 Step 1: Choose class as a starting point
 Step 2: Identify class's responsibilities
Step 3: For each responsibility:
 Step 1: Add verification tests
 Step 2: Refactor related code
 Step 3: Rerun verification tests
 Step 4: Repeat from step 1 by choosing a new starting point

Figure 6-15. Step 3.1

Let's start with our first responsibility, which is "Fetch latest books on startup." Before we start refactoring the related code, we first need to add verification tests (Figure 6-15). Verification tests are high-level tests that verify that the feature or functionality that we're refactoring is working fine. For user-facing features like the one we're trying to refactor now, a verification can be in the form of a UI test, as that's the highest level of testing we have. If the part we're refactoring is not user-facing, then integration tests can be used. Verification tests are an integral part of our process, as they help in avoiding regressions due to our refactor.

Let's write a verification test for our feature:

```
func testShowingBestSellerBooks() throws {
    // Given
    let app = XCUIApplication()
    app.launch()
}
```

```

// When
let booksTableView = app.tables
let cells = booksTableView.cells
_ = cells.firstMatch.waitForExistence(timeout: 1.0)

// Then
XCTAssertGreaterThan(cells.count, 0)
}

```

Our verification test simply makes sure that the list table view contains at least one cell. For the scope of this feature, we only care about the table view being populated on startup, and we don't care about the content of the cells yet.

The preceding test highly depends on the back end and could easily fail if the back-end server fails. This dependency is not optimum at all, and we'll talk about how we can remove it in Chapter 8. However, for now, this test will do as it is.

Refactoring

- Step 0: Create initial module map
- Step 1: Choose class as a starting point
- Step 2: Identify class's responsibilities
- Step 3: For each responsibility:**
 - Step 1: Add verification tests
 - Step 2: Refactor related code**
 - Step 3: Rerun verification tests
- Step 4: Repeat from step 1 by choosing a new starting point

Figure 6-16. Step 3.2

Now that we have our verification test, we can safely start refactoring (Figure 6-16). To refactor this feature, we need to ask ourselves a few questions. Is the code responsible for this feature in the right place, or should it be moved to a new component or even a new module? And after we move that code to its right place, does it need to be refactored?

If we take a look at the code responsible for our feature, we'll find that the function we need to address is `fetchBooks()`. So the first question is, is it in the right place? Since there is no specific design pattern or an architecture inside the app, we will try to apply a design pattern while refactoring. We are going to use MVP as we did in Chapter 5. And from MVP we know that view controllers should not contain any business logic and should only be responsible for handling the UI. Therefore, we know that we need to move `fetchBooks()` somewhere else, but where? We already know that it will be included in the **MainView Module**, but what component? For that question, we'll try to understand more what `fetchBooks()` does. `fetchBooks()` makes a network request and parses the responses in order to extract the lists of books and then uses that to update the data source of the table. We will apply the **MVP** design pattern on the logic we want to implement as if we'll implement it from scratch. We should think about how the new objects will interact with each other without looking at the current code in order not to be affected by the current implementation. By doing that, we will end up with the following design in Figure 6-17.

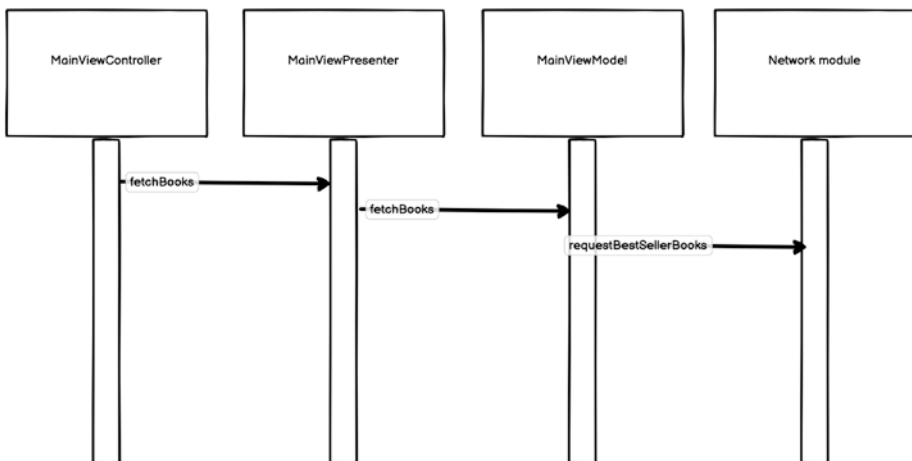


Figure 6-17. MVP design pattern

Now it's time to bust out our TDD skills. If you look at Figure 6-18, you'll probably remember it from Chapter 5. For the end-to-end test, we already have that covered by our verification test. And since we already know how the objects will interact with each other, we are ready to start writing some integration tests.

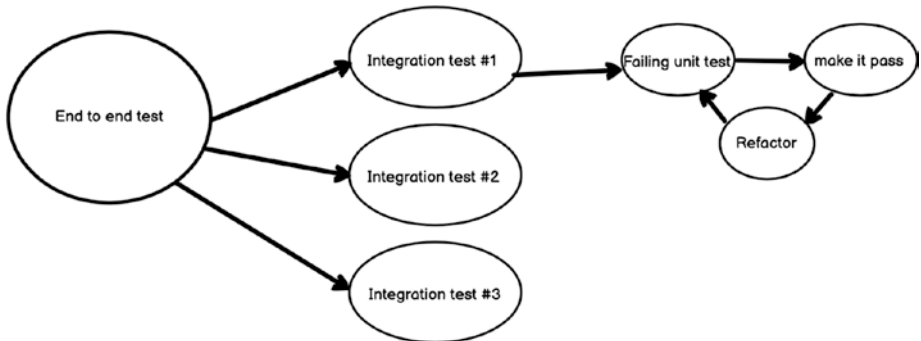


Figure 6-18. Testing plan diagram

Integration Test

Let's create a new class called `MainViewIntegrationTests`, which will include all integration tests related to this module. It's very useful to group the same types of tests together so that you have the flexibility to run a specific type of tests easily.

From Figure 6-17 we know that once `MainViewController` is loaded, we will initialize `MainViewPresenter`, which will take `MainViewModel` inside the constructor. `MainViewPresenter` will contain a method that will fetch all books and abstract the communication to `MainViewModel` under the hood; then, the model will return the books. Lastly, `MainViewPresenter` will update the view. Now let's convert this to a test:

```

func testFetchBestSellerBooksReturnsList() throws {
    // Given
    let testBundle = Bundle(for: type(of: self))
    let booksJSONURL = testBundle.url(forResource:
        "BestSellerBooksStub", withExtension: "json")
    let booksJSON = try Data(contentsOf: booksJSONURL!)

    let expectedLists: [List] = stubbedlists()
    var actualLists: [List] = []

    let networkLayer = NetworkLayerStub(stubbedData: booksJSON)
    let mainViewModel = MainViewModel(networkLayer: networkLayer)
    let mainViewPresenter = MainViewPresenter(mainViewModel:
        mainViewModel)

    // when & then
    let waitForBooks = XCTestExpectation(description: "Wait to
        fetch books")
    mainViewPresenter.fetchBestSellerBooks { lists in
        actualLists = lists ?? []
        waitForBooks.fulfill()
    }

    self.wait(for: [waitForBooks], timeout: 0.1)
    XCTAssertEqual(actualLists, expectedLists, "Fetched books
        does not match the expected")
}

func stubbedlists() -> [List] {
    let firstBook = BookModel(title: "THE LAST THING HE TOLD
        ME", contributor: "by Laura Dave", author: "Laura Dave",
        createDate: "2021-05-26 22:10:24")
    let secondBook = BookModel(title: "SOOLEY", contributor:
        "by John Grisham", author: "John Grisham", createDate:
        "2021-05-26 22:10:24")
}

```



```

    let firstList = List(listID: 704, listName: "Combined Print
and E-Book Fiction", displayName: "Combined Print & E-Book
Fiction", books: [firstBook,secondBook])
    return [firstList]
}

```

The test will not even build because we still haven't added any of the components that it's testing, and that's normal.

It makes sense to allow our network layer to stub API requests to return the expected JSON so that we can assert on values and prevent our tests from depending on network calls, which will make it flaky. We will talk more on stubbing in Chapter 7.

NetworkLayer

Now that we have our integration test, it's time to go down a level to unit tests. We will start with the **Network Module**. But since testing the network layer can be quite tricky, we'll skip its tests for now. But don't worry. We will go deep on how we can test our network layer later in **Chapter 9**. What we'll do is add our network layer class in its separate module. It's a real simple class. It will only execute a single request and return data:

```

class NetworkLayer {
    let host = "api.nytimes.com"
    let API_KEY = "YOUR_API_KEY"
    let bestSellerBooks = "/svc/books/v3/lists/overview.json"

    public func executeNetworkRequest(callBack: @escaping
(_ data:Data?) -> Void) {
        var components = URLComponents()
        components.scheme = "https"
        components.host = host
    }
}

```

```

components.path = bestSellerBooks
components.queryItems = [URLQueryItem(name: "api-key",
value: API_KEY), URLQueryItem(name: "offset", value: "20")]

guard let url = components.url else {
    callback(nil)
    preconditionFailure("Failed to construct URL")
}

let task = URLSession.shared.dataTask(with: url) {
    data, response, error in

    guard let data = data else {
        callback(nil)
        return
    }

    callback(data)
}

task.resume()
}
}

```

MainViewModel

Let's jump to the next class, which will be `MainViewModel`. It's part of the **MainView Module** and will be responsible for creating a `NetworkLayer` object and performing network requests and then parsing the response data and returning the parsed data through a callback. As usual we will start with `MainViewModelTests`. We will write all tests to make sure that this class is working fine and as expected:

```

func testFetchingAndParsingBestSellerBooks() throws {
    // Given
    let testBundle = Bundle(for: type(of: self))
    let booksJSONURL = testBundle.url(forResource:
        "BestSellerBooksStub", withExtension: "json")
    let booksJSON = try Data(contentsOf: booksJSONURL!)

    let expectedLists: [List] = stubbedlists()
    var actualLists: [List] = []

    let networkLayer = NetworkLayerStub(stubbedData: booksJSON)
    let mainViewModel = MainViewModel(networkLayer: networkLayer)

    // when & then
    let waitForBooks = XCTestExpectation(description: "Wait to
        fetch books")
    mainViewModel.fetchBestSellerBooks { lists in
        actualLists = lists ?? []
        waitForBooks.fulfill()
    }

    self.wait(for: [waitForBooks], timeout: 0.1)
    XCTAssertEqual(actualLists, expectedLists, "Fetched books
        does not match the expected")
}

func stubbedlists() -> [List] {
    let firstBook = BookModel(title: "THE LAST THING HE TOLD
        ME", contributor: "by Laura Dave", author: "Laura Dave",
        createDate: "2021-05-26 22:10:24")
    let secondBook = BookModel(title: "SOOLEY", contributor:
        "by John Grisham", author: "John Grisham", createDate:
        "2021-05-26 22:10:24")

```

```

    let firstList = List(listID: 704, listName: "Combined Print
    and E-Book Fiction", displayName: "Combined Print & E-Book
    Fiction", books: [firstBook,secondBook])
    return [firstList]
}

```

The preceding test first sets up an instance of `MainViewModel` by initializing it using a `NetworkLayer` instance. We then call the function that we're trying to test, which fetches the data from the server, and then we wait till it's done. And finally we assert on the returned data.

In order to test `MainViewModel`, we need to stub `NetworkLayer` to return specific JSON so that we can assert on the output of `MainViewModel`. We need to create a new class that will stub the network, as our just added test is not building because of that. `NetworkLayerStub` will look like this:

```

class NetworkLayerStub: NetworkLayer {
    var stubbedData:Data?

    init(stubbedData:Data) {
        self.stubbedData = stubbedData
    }

    public override func executeNetworkRequest(callback:
    @escaping (_ data:Data?) -> Void){
        let jsonData = self.stubbedData
        callback(jsonData)
    }
}

```

We solved one build error by adding `NetworkLayerStub`, but the test is still not building. Now it's time to write code to make `MainViewModelTests` pass. For that to happen, we need to create `MainViewModel`, and it should look like this:

```

class MainViewModel: NSObject {
    private var networkLayer:NetworkLayer?

    init(networkLayer:NetworkLayer?) {
        self.networkLayer = networkLayer
    }

    public func fetchBestSellerBooks(callBack: @escaping
    (_ data:[List]?) -> Void) {
        self.networkLayer?.executeNetworkRequest(callBack:
        { data in
            guard let data = data else {
                callBack(nil)
                return
            }

            var response:Response?
            do {
                response = try JSONDecoder().decode(
                Response.self, from: data)
            } catch {
                print(error.localizedDescription)
            }

            if let lists = response?.results.lists {
                callBack(lists)
                return;
            }

            callBack(nil)
        })
    }
}

```

Here we simply implement the function we need, which is `fetchBestSellerBooks`. The function is passed a callback block as a parameter, which should be called with the fetched books when done. We use the instance of `NetworkLayer` to make the request, and we decode the response and then return it in the callback.

Now if we run `MainViewModelTests` (Figure 6-19), it should pass ✓.



Figure 6-19. *MainViewModelTests passing*

MainViewPresenter

Next, it's time to write unit tests for `MainViewPresenter`. First we'll create a new class to act as a stub for the `MainViewModel`:

```
@testable import Books

class MainViewModelStub: MainViewModel {
    var stubbedLists:[List]?

    init(stubbedLists:[List]) {
        self.stubbedLists = stubbedLists
        super.init(networkLayer: nil)
    }

    public override func fetchBestSellerBooks(callBack:
    @escaping (_ lists:[List]?) -> Void) {
        callBack(self.stubbedLists)
    }
}
```

And now we can write our test:

```
func testFetchingBestSellerBooksReturnsLists() throws {
    // Given
    let expectedLists: [List] = stubbedlists()
    var actualLists: [List] = []

    let mainViewModel = MainViewModelStub(stubbedLists:
    expectedLists)
    let mainViewPresenter = MainViewPresenter(mainViewModel:
    mainViewModel)

    // when & then
    let waitForBooks = XCTestExpectation(description: "Wait
    to fetch books")
    mainViewPresenter.fetchBestSellerBooks { lists in
        actualLists = lists ?? []
        waitForBooks.fulfill()
    }

    self.wait(for: [waitForBooks], timeout: 0.1)
    XCTAssertEqual(actualLists, expectedLists, "Fetched
    books does not match the expected")
}

func stubbedlists() -> [List] {
    let firstBook = BookModel(title: "THE LAST THING HE
    TOLD ME", contributor: "by Laura Dave", author: "Laura
    Dave", createdAt: "2021-05-26 22:10:24")
    let secondBook = BookModel(title: "SOOLEY",
    contributor: "by John Grisham", author: "John Grisham",
    createdAt: "2021-05-26 22:10:24")
}
```

```

        let firstList = List(listID: 704, listName: "Combined
        Print and E-Book Fiction", displayName: "Combined Print
        & E-Book Fiction", books: [firstBook,secondBook])
        return [firstList]
    }

```

The preceding test is a bit similar to the test we just wrote for `MainViewModel`. We set up an instance of our presenter using a stub object. We then call our function and wait for it to finish fetching the bestseller books. And finally we assert on the returned books.

We can now write code to make `MainViewPresenterTests` pass:

```

class MainViewPresenter: NSObject {

    private var mainViewModel:MainViewModel?

    init(mainViewModel:MainViewModel?) {
        self.mainViewModel = mainViewModel
    }

    public func fetchBestSellerBooks(callback: @escaping
    (_ data:[List]?) -> Void) {
        self.mainViewModel?.fetchBestSellerBooks(callback: {
            lists in
                callback(lists)
        })
    }
}

```

The presenter implementation is quite straightforward. It implements a function that fetches the best-seller books. And the implementation of this function is basically calling the corresponding function inside `MainViewModel`. You might think that we don't need the presenter and

that it just acts as a wrapper, but that’s only for now. The separation of logic is extremely important, and as we keep refactoring more code, this importance will become more prominent.

Now if we run `MainViewPresenterTests` (Figure 6-20), it should pass ✓.

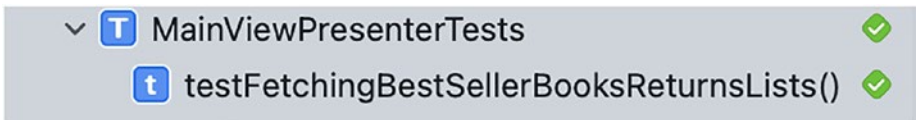


Figure 6-20. *MainViewPresenterTests* passing

Last Touches

All our unit tests are passing now. But not only that, now if we run the integration test, it should pass as well. Last thing we need to do is replace the old implementation of `fetchBooks()` with the new one that makes use of the newly added components. Let’s replace the existing `fetchBooks()` func with the following:

```
func fetchBooks() {
    self.mainViewPresenter?.fetchBestSellerBooks(callback:
    { lists in
        if let lists = lists {
            self.lists = lists
            DispatchQueue.main.async {
                self.refreshControl.endRefreshing()
                self.tableView?.reloadData()
            }
        }
    })
}
```

This here marks the end of **step 3.2**. We have now completely refactored the logic related to our feature.

Test Value

Before we jump to the next step, let's try to do something that might showcase the value of all the tests that we've been adding. Inside `MainViewModel` let's replace the `fetchBestSellerBooks` method with the following code. We simply remove the return function after `callback(lists)`, and as a result of this, the callback will be called twice. This is a ticking time bomb, as this misbehavior is not causing bugs now but can cause problems in the future. If you run the app now, it will work as expected because we have a guard on `nil` inside `MainViewController`. But if we remove that guard one day or reuse that code somewhere else, bugs will start showing. However, if we just run our tests now, we'll see that they'll catch this (Figure 6-21).

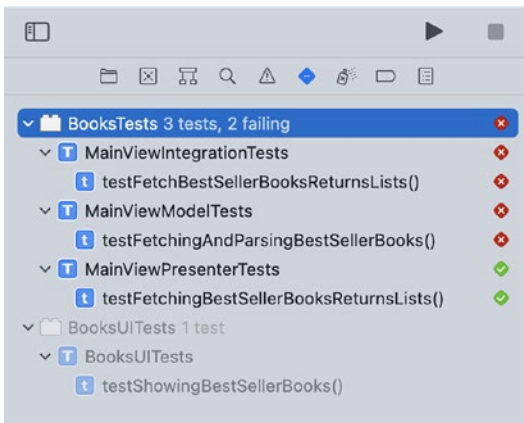


Figure 6-21. *Failing unit tests*

```
public func fetchBestSellerBooks(callback: @escaping
(_ lists:[List]?) -> Void) {
    self.networkLayer?.executeNetworkRequest(callback: {
    data in
        guard let data = data else {
```

```

        callBack(nil)
        return
    }

    var response:Response?
    do {
        response = try JSONDecoder().decode(Response.
            self, from: data)
    } catch {
        print(error.localizedDescription)
    }

    if let lists = response?.results.lists {
        callBack(lists)
    }

    callBack(nil)
})
}

```

Rerun Verification Tests

- Step 0: Create initial module map
- Step 1: Choose class as a starting point
- Step 2: Identify class's responsibilities
- Step 3: For each responsibility:**
 - Step 1: Add verification tests
 - Step 2: Refactor related code
 - Step 3: Rerun verification tests**
- Step 4: Repeat from step 1 by choosing a new starting point

Figure 6-22. Step 3.3

All this time we’ve been working on refactoring one responsibility of `MainViewController`, which is “Fetch latest books on startup.” Before we can say we’re done with this responsibility, we need to run the verification test we added in **step 3.1** to verify that everything is running as expected (Figure 6-22). If we try to run `testShowingBestSellerBooks()` (Figure 6-23), it should pass ✓.

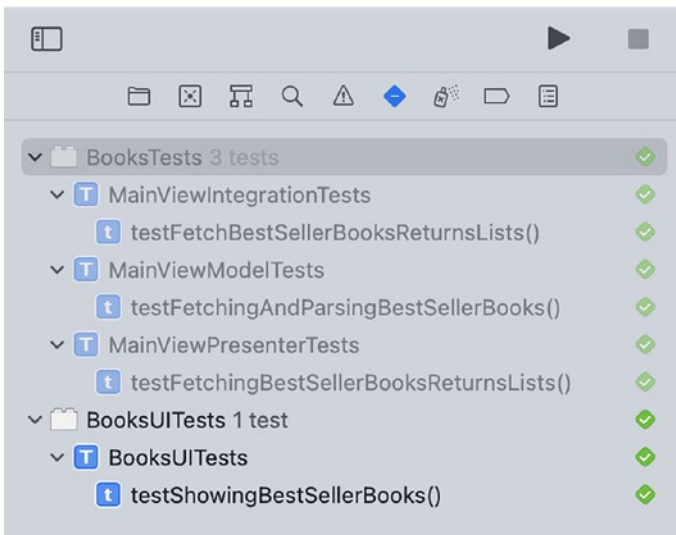


Figure 6-23. Test passing

Refactor the Rest of the Responsibilities

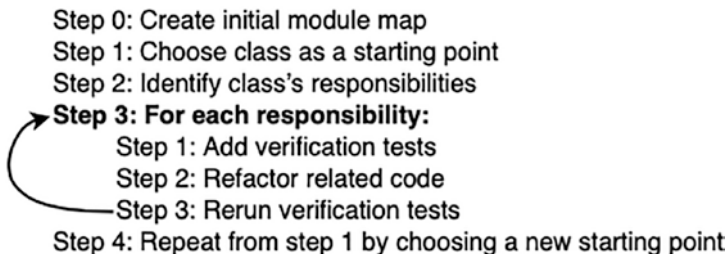


Figure 6-24. Repeats step 3 and its substeps

Back in **step 2** we identified six responsibilities of **MainViewController**. We have just finished refactoring the first responsibility. Now we should carry out the same steps for the rest of the responsibilities one by one (Figure 6-24).

Next Starting Point

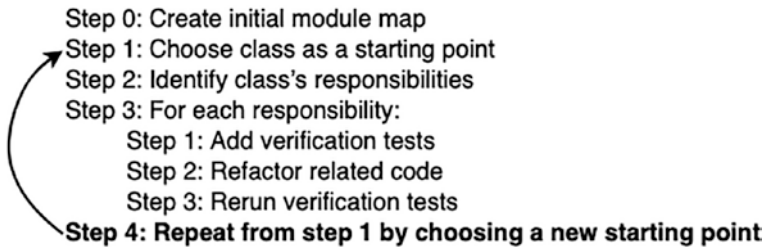


Figure 6-25. Step 4

Once we're done refactoring all the responsibilities in **MainViewController**, we will basically go back to step 1 and repeat the process all over again (Figure 6-25). So we will pick a new starting point and refactor it completely as we did for **MainViewController**.

Exercise

We are done with the first responsibility of **MainViewController**. For your exercise, try refactoring the rest of the responsibilities using the same process we followed in this chapter.

Summary

In this chapter, we talked about the concept of modularization, which is the idea of breaking up a system into multiple modules that are relatively independent and interchangeable. Generally, a module is a standalone

piece of code that provides specific and tightly coupled functionality. When we ignore modularization completely, we tend to end up with a messy architecture, which is commonly referred to as a **big ball of mud** architecture.

Having a modularized app has many benefits over non-modularized apps. Non-modularized apps tend to be unpredictable whenever change is introduced into the project. This is due to interconnected dependencies between components. On the other hand, in a modularized app, when we're introducing a change in a module, we are certain that we will only affect this module. This is thanks to the strong separation in our code. This code separation provides another very important benefit, which is code readability. We've mentioned before that developers spend more time reading code than writing it, and having a modularized app makes it much easier to read and understand how it works. In a properly modularized app, one developer can actively work on a module without understanding or touching other modules in the app.

Given these many benefits, it's probably best to take a modularized approach when working on a new app from scratch. However, if we have a legacy app that's not modularized, we can still transform it. There are two ways we can do that: First, we can rewrite our whole app. Rewriting as a concept is pretty straightforward; we basically throw all what we have and start with a clean (modularized) slate. However, this approach is pretty aggressive and requires a huge investment in a short time. The other approach is refactoring, which is a more granular approach where we modularize our app one step at a time. This approach is much slower, but it allows us to continue working on our app and add features while we actively transform it at the same time.

Modularizing an app through gradual refactoring is not an easy task. However, there's a systematic process (Figure 6-26) that we can follow. First, we create a projection of what our app would look like internally if we would divide it into modules to give us an idea of our end goal. After that

we pick a class as a starting point and list down this class's responsibilities. Then for each responsibility we write a verification test to make sure that our following changes will not introduce any regressions. Then we proceed with refactoring this responsibility if needed. We could move it to a different class or even a different module or even create a whole new module for it. Once we refactor all responsibilities for that class, we would just loop over our process again by choosing a new starting point. And we'll keep going through that loop until we run out of starting points. When we reach this point, this means that we no longer have un-modularized code.

- Step 0: Create initial module map
- Step 1: Choose class as a starting point
- Step 2: Identify class's responsibilities
- Step 3: For each responsibility:
 - Step 1: Add verification tests
 - Step 2: Refactor related code
 - Step 3: Rerun verification tests
- Step 4: Repeat from step 1 by choosing a new starting point

Figure 6-26. *Modularization process*

CHAPTER 7

Dependency Injection and Mocks

Writing tests for a component can be a tedious task if this component depends on another component that has an unpredictable behavior. To test such a component, we need to be able to control this unpredictable behavior. We can do that with the help of a **test double**. The term test double was first introduced in Gerard Meszaros's book *XUnit Test Patterns*. Test double is a generic term for any kind of pretend object used in place of a real object for testing purposes. Another situation in which it can be challenging to write tests is if we have a component that communicates with another component and we want to verify something related to this interaction. In this case, a test double is also the best course of action. Test doubles are an imperative tool in any programmer's arsenal. And using them is essential for having an application that's highly covered with tests and makes our tests more stable.

Stubbing

One type of test doubles is stubs. A **stub** is an object that holds predefined data and provides these data during tests. It is used when we don't want to use real data and to have a more consistent data source. A test doesn't really care if the function is called or not on a stub, as long as the test object

(or system under test) gets the data it needs from the stub and does the right thing. And if the stub is passed a value, the test doesn't care about that value. Also, regardless of the input, the stub always outputs the same predefined data. Due to its nature, a stub is considered a fairly lightweight test double.

An example of when we need stubs is when we have an object that depends on making a network call to a server. Making an actual network request will lead to our test being both slow and unpredictable as we can't control what the server will return each time.

Let's say that we have an object A that has a Boolean variable `status` whose value depends on the data returned from the server. So if the server returns success, then `status` will be `true`; and if the server returns failure, then `status` will be `false` (Figure 7-1). To be able to test both these scenarios with confidence, we will need to use a stub.

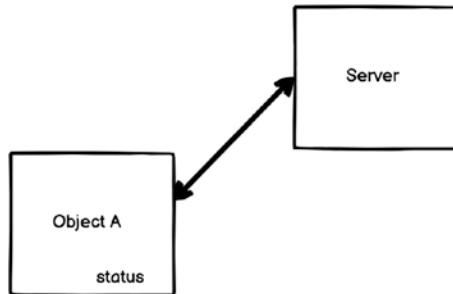


Figure 7-1. *Dependency example*

We will create a new object called `ServerStub`, and we will use it in place of the real `Server` object as seen in Figure 7-2. Our stub has two methods to control the kind of data it should return. We will use these methods to set up our tests.

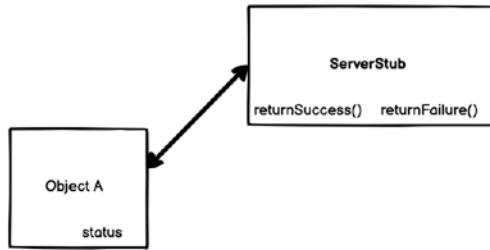


Figure 7-2. *Stubbing a dependency*

When we write tests for our two scenarios, they will look like this:

```
func testObjectASuccessStatus() {
    // Given
    let server = ServerStub()
    server.returnSuccess()

    // When
    let objectA = ObjectA(server)

    // Then
    XCTAssertTrue(objectA.status)
}

func testObjectAFailureStatus() {
    // Given
    let server = ServerStub()
    server.returnFailure()

    // When
    let objectA = ObjectA(server)

    // Then
    XCTAssertFalse(objectA.status)
}
```

In each test we create an instance of our stub and then set it up using either `returnSuccess()` or `returnFailure()`. Then we pass the stub to our test object and do our assertion on status. We will talk about injecting stubs into our test objects later on in this chapter.

Mocking

Another type of test doubles is mocks. A mock is slightly more complex than a stub. It could return some fake data just like a stub and can also verify whether a particular method was called. Mocks register calls they receive, and in our tests, we can verify that all expected actions were performed on a specific mock. We use mocks when we don't want to invoke production code or when there is no easy way to verify that intended code was executed.

Let's say we have three objects: objects A, B, and C. Object A has a method that takes an input, and based on that input, it decides to either call object B or object C (Figure 7-3). If we pass true to our test object, it should call object B, and if we pass it false, it should call object C. To be able to verify both these scenarios, we will need to use a mock.

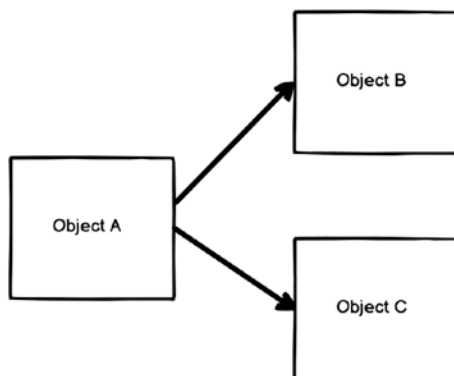


Figure 7-3. *Dependency example*

We create two new objects that will act as our mocks (Figure 7-4). `ObjectBMock` and `ObjectCMock` will both do the same simple task, which is register if they are called and save this info in the public property `isCalled`.

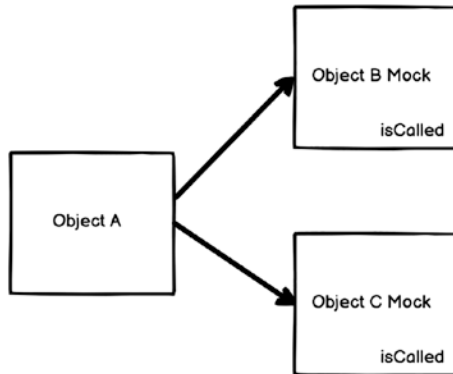


Figure 7-4. *Mocking dependencies*

Now we can write our tests like this:

```

func testObjectALogic1 () {
    // Given
    let objectB = ObjectBMock()
    let objectC = ObjectCMock()
    let objectA = ObjectA(objectB, objectC)

    // When
    objectA.doLogic(true)

    // Then
    XCTAssertTrue(objectB.isCalled)
    XCTAssertFalse(objectC.isCalled)
}

func testObjectALogic2 () {
    // Given
    let objectB = ObjectBMock()
  
```

```

    let objectC = ObjectCMock()
    let objectA = ObjectA(objectB, objectC)

    // When
    objectA.doLogic(false)

    // Then
    XCTAssertFalse(objectB.isCalled)
    XCTAssertTrue(objectC.isCalled)
}

```

As well as registering if they are called, mocks can also register the values they are passed with each call. And in our tests, we can verify that the values passed to our mocks are correct.

So, for our example, we can modify our two mocks to save the values they are passed. And then we can modify our tests to be like this:

```

func testObjectALogic1 () {
    // Given
    let objectB = ObjectBMock()
    let objectC = ObjectCMock()
    let objectA = ObjectA(objectB, objectC)

    // When
    objectA.doLogic(true)

    // Then
    XCTAssertTrue(objectB.isCalled)
    XCTAssertEqual(objectB.value, "Test")
    XCTAssertFalse(objectC.isCalled)
}

func testObjectALogic2 () {
    // Given
    let objectB = ObjectBMock()

```

```

let objectC = ObjectCMock()
let objectA = ObjectA(objectB, objectC)

// When
objectA.doLogic(false)

// Then
XCTAssertFalse(objectB.isCalled)
XCTAssertTrue(objectC.isCalled)
XCTAssertEqual(objectC.value, "Test")
}

```

Test Doubles Creation

We talked about different types of test doubles: mocks and stubs. But we did not talk about how we can create them. Doubles by their definition are objects that can be used in place of real objects. So a double has to be somewhat related to the original object so that we can seamlessly swap in our double in our tests. There are multiple ways of creating doubles. In this chapter, we'll talk about creation using inheritance and creation using protocols.

Creation Using Inheritance

We used this approach a lot in the previous chapters. The inheritance concept in general is a mechanism where you can derive a class from another class. It is one of the core concepts of Object-Oriented Programming (OOP). When we inherit from a class, we inherit all characteristics of the parent class. And this is the essence of this approach. We inherit all properties and functions of the object to be mocked or stubbed, and we change the behavior of the part we want to mock or stub through overriding (Figure 7-5). The good thing about this approach is that

we have access to the original implementation, so we can either change the implementation or if needed we can just extend it, keeping the old logic as it is and just adding new logic that's specific to testing.

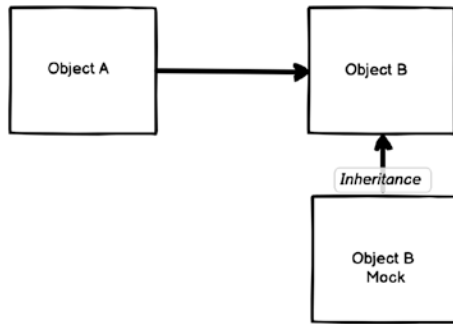


Figure 7-5. *Creation by inheritance*

If you recall, we used this approach in Chapter 6 when we were writing tests for `MainViewModel`. `MainViewModel` depended on `NetworkLayer`, so we created `NetworkLayerStub` using inheritance. And it looked like this:

```

class NetworkLayerStub: NetworkLayer {
    var stubbedData:Data?

    init(stubbedData:Data) {
        self.stubbedData = stubbedData
    }

    public override func executeNetworkRequest(callback: @
    escaping (_ data:Data?) -> Void){
        let jsonData = self.stubbedData!
        callback(jsonData)
    }
}
  
```

We inherit from `NetworkLayer`, and we override `executeNetworkRequest` and make it return `stubbedData` instead of actually making a network request. We set `stubbedData` from our tests as needed.

Creation Using Protocols

Creation using protocols is a bit similar to creation using inheritance, and it goes hand in hand with Protocol-Oriented Programming (POP). A protocol acts as a blueprint to what we expect from the type (class, struct, or enum) that conforms to it. In the protocol-oriented approach, we start designing our system by defining protocols. So if we need to create a component that will be doing some logic, we will abstract this logic to APIs and define it inside a protocol. Then we create our component by conforming to that protocol and implementing the required functions.

We can use the same protocol-oriented approach when creating test doubles. If we have a dependency that we need to exchange with a double, we add a protocol describing this dependency. Now our original object will conform to this dependency, and we can now say that our test object depends on a component that conforms to this protocol. In our tests, we can now add a new component that conforms to the protocol and inject it into our test object, and this will be our test double (Figure 7-6).

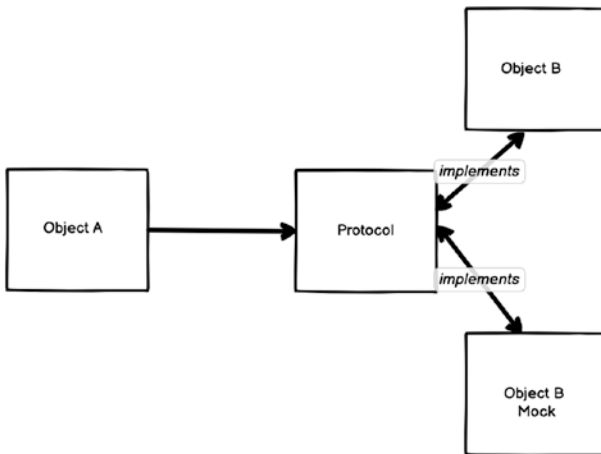


Figure 7-6. *Creation by protocol conformance*

Let's try to rewrite the `MainViewModel` example using a protocol. `MainViewModel` needs to depend on the protocol instead of the `NetworkLayer` object. Our protocol will look like this:

```
protocol NetworkProtocol {
    func executeNetworkRequest(callback: @escaping (_
        data:Data?) -> Void)
}
```

And now we will modify `MainViewModel` so that it now depends on `NetworkProtocol` instead of `NetworkLayer`:

```
class MainViewModel: NSObject {
    private var networkLayer:NetworkProtocol?

    init(networkLayer:NetworkProtocol) {
        self.networkLayer = networkLayer
    }
}
```

```

public func fetchBestSellerBooks(callback: @escaping (_
lists:[List]?) -> Void) {
    self.networkLayer?.executeNetworkRequest(callback: {
        data in
            guard let data = data else {
                callback(nil)
                return
            }

            var response:Response?
            do {
                response = try JSONDecoder().decode(Response.
                self, from: data)
            } catch {
                print(error.localizedDescription)
            }

            if let lists = response?.results.lists {
                callback(lists)
                return;
            }

            callback(nil)
        })
    }
}

```

Finally, we will create our test double by creating a new class that conforms to `NetworkProtocol`:

```

class NetworkLayerStub: NetworkProtocol {
    var stubbedData:Data?
}

```

```

    init(stubbedData:Data) {
        self.stubbedData = stubbedData
    }

    func executeNetworkRequest(callBack: @escaping (_
data:Data?) -> Void){
        let jsonData = self.stubbedData!
        callBack(jsonData)
    }
}

```

Dependency Injection

We talked about mocks and stubs, and we talked about how we can create these helpful test doubles. But we are still to learn how we can inject these test doubles into our code. There are multiple ways to inject our test dependencies. We will talk about property injection and initializer injection.

We have the following class `Example` that we would like to write tests for. `Example` depends on `Network.shared`, which is a singleton instance. However, we need to mock `Network` in order to verify that our request is made:

```

class Example {
    func doWork() {
        Network.shared.makeRequest()
    }
}

```

So let's refactor our class so that we can easily inject our mock from our tests.

Initializer Injection

We used this approach a lot in the previous chapters. In this approach our entry for injecting a dependency is our initializer. We pass the dependency to our object whenever we create a new instance. We save a reference to this dependency in our object, and we use that reference whenever we need to access our dependency. So in tests, when we're creating an instance of our object, we simply pass our test double in the initializer instead of the real thing (Figure 7-7).

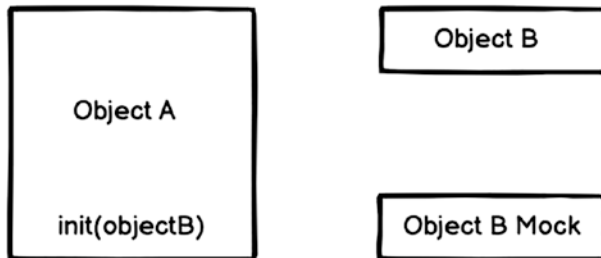


Figure 7-7. *Initializer injection*

When in our code we always pass the same dependency to our object and only need to pass something different in tests, then it's a good idea to use default arguments in Swift. Here we tell our initializer that the default for the dependency is this object, but we can override it when we need. This is useful as it makes our code neater and more readable.

When we refactor our class, it should look like this:

```
class Example {
    private var network:Network?
    init(network:Network = Network.shared) {
        self.network = network
    }
}
```

```

func doWork() {
    self.network.makeRequest()
}
}

```

And to inject a test double, now we can simply do this:

```

let networkMock = NetworkMock()
let testObject = Example(network: networkMock)

```

Property Injection

Injection using a property is the easiest way to inject, but it will not be applicable in most of the cases. Let's imagine that we have object A that uses object B to perform a specific task. If object A has a public property that holds object B, then we can use this to inject our mock in place of the original object B inside our tests (Figure 7-8). But we need to be careful not to expose properties only for tests as this will break the abstraction of our objects and it will lead to a lot of code smells.

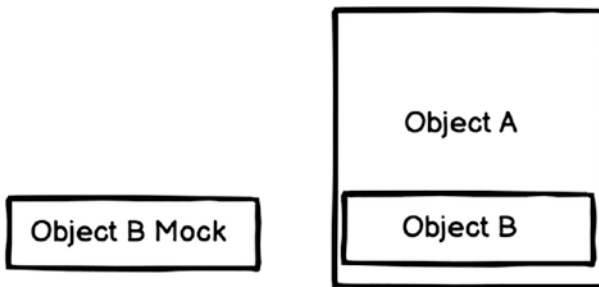


Figure 7-8. *Property injection*

When we refactor our class to use property injection, it should look like this:

```
class Example {
    public var network:Network?
    init() {
        self.network = Network.shared
    }

    func doWork() {
        self.network.makeRequest()
    }
}
```

And to inject a test double, now we can simply do this:

```
let networkMock = NetworkMock()
let testObject = Example()
testObject.network = networkMock
```

Stubbing the Network in UI Tests

All previous approaches can be implemented inside unit and integration tests. It's not recommended to use these approaches inside UI tests because UI tests should test your app as a black box exactly like what your customer will use. It does not make sense to test a mock object inside an end-to-end test and not the actual code. However, in some cases, we'll need to stub a certain behavior, and we can do that with a higher level of stubbing.

First, let's open up the starter project from this chapter's resources. This is a version of **Books**, the app we've worked on in the previous chapter. Let's take a look at the end-to-end test implemented in Chapter 6, step 3.2:

```

func testShowingBestSellerBooks() throws {
    // Given
    let app = XCUIApplication()
    app.launch()

    // When
    let booksTableView = app.tables
    let cells = booksTableView.cells
    _ = cells.firstMatch.waitForExistence(timeout: 1.0)

    // Then
    XCTAssertGreaterThan(cells.count, 0)
}

```

This test is not useful at all. First, it's depending on the network request so it's slow, and we are not asserting on the data shown inside the table. The app may show the wrong data, and the test will pass.

In order to fix this test, we are going to stub the network request and return specific data (Figure 7-9), and the test should make sure that the data is rendered correctly inside the app.

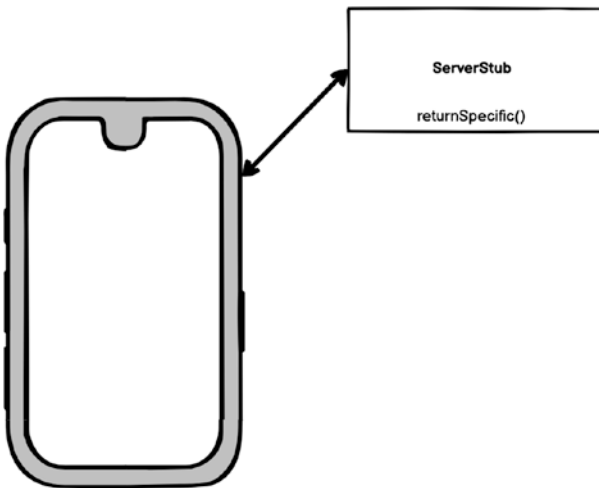


Figure 7-9. Network stubbing

We are going to use a third-party library called **Swifter** to mock network requests. We can achieve this using different other libraries or even manually. But for this example, we will be going with this lightweight third-party dependency.

First, we need to integrate Swifter. We will use Swift Package Manager (SPM) to install the dependency (Figure 7-10). We need to make sure to add it to the BooksUITests target, not the app (Figure 7-11).

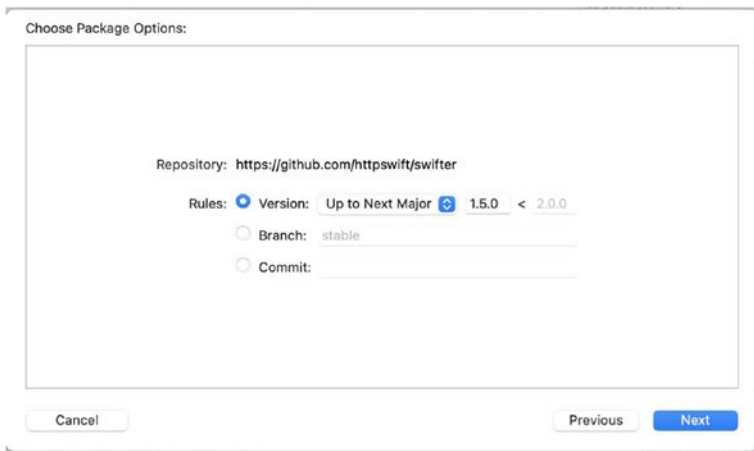


Figure 7-10. *Integrate a third party using SPM (Step 1)*

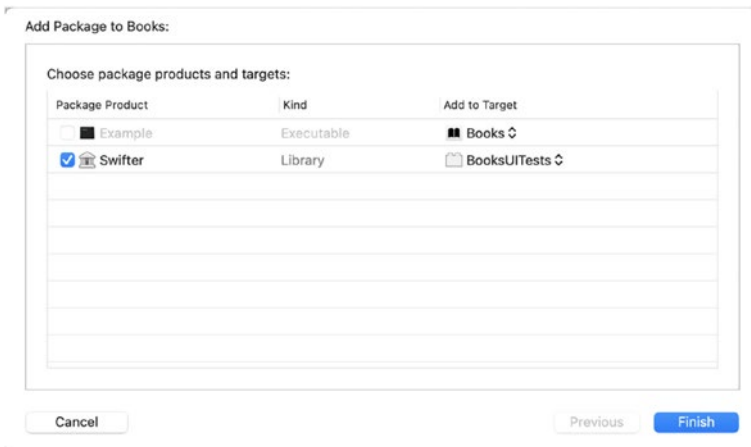


Figure 7-11. Integrate a third party using SPM (Step 2)

Now that we have **Swifter** installed, we need to make a minor change inside our network layer to allow **Swifter** to stub the network requests. We need to check on the launch argument inside `ProcessInfo`, and in case it contains `TESTING`, we need to change the domain to `localhost`, change `HTTPS` to `HTTP`, and add `8080` to port:

```
func getHost() -> String {
    if ProcessInfo.processInfo.arguments.
        contains("TESTING") {
        return "localhost"
    } else {
        return "api.nytimes.com"
    }
}

func getScheme() -> String {
    if ProcessInfo.processInfo.arguments.
        contains("TESTING") {
        return "http"
    } else {
```

```

        return "https"
    }
}

public fun executeNetworkRequest(callBack: @escaping (_
data:Data?) -> Void) {
    var components = URLComponents()
    components.scheme = getScheme()
    components.host = getHost()
    components.port = 8080
    components.path = bestSellerBooks
    components.queryItems = [URLQueryItem(name: "api-key",
value: API_KEY), URLQueryItem(name: "offset", value:
"20")]

    guard let url = components.url else {
        callBack(nil)
        preconditionFailure("Failed to construct URL")
    }

    let task = URLSession.shared.dataTask(with: url) {
        data, response, error in

        guard let data = data else {
            callBack(nil)
            return
        }

        callBack(data)
    }

    task.resume()
}

```

What we need to do in the setup is start the server. If the server fails to start, it will throw an error, which will fail our test. This makes sense as the test will be useless if our stub server is not running:

```
class BooksUITests: XCTestCase {
    var server = HttpServer()

    override fun setUpWithError() throws {
        continueAfterFailure = false
        try server.start()
    }

    override fun tearDownWithError() throws {
        server.stop()
    }
}
```

We are going to use the same `BestSellerBooksStub.json`, so we will make sure to include it in both targets (Figure 7-12).

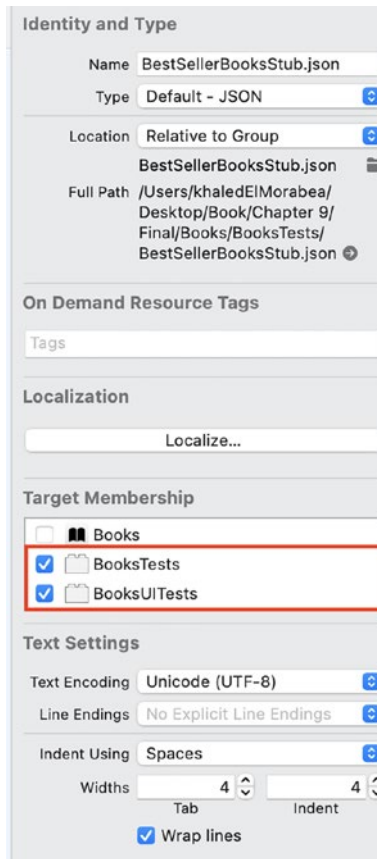


Figure 7-12. Setting target membership for *BestSellerBooksStub.json*

Also we need to allow only the local host domain to use HTTP instead of HTTPS. This will prevent the system from blocking our requests due to security reasons. We can do that by modifying the Info.plist (Figure 7-13).

Key	Type	Value
Information Property List	Dictionary	(17 items)
Localization native development region	String	\$(DEVELOPMENT_LANGUAGE)
Executable file	String	\$(EXECUTABLE_NAME)
Bundle identifier	String	\$(PRODUCT_BUNDLE_IDENTIFIER)
InfoDictionary version	String	6.0
Bundle name	String	\$(PRODUCT_NAME)
Bundle OS Type code	String	\$(PRODUCT_BUNDLE_PACKAGE_TYPE)
Bundle version string (short)	String	1.0
Bundle version	String	1
Application requires iPhone environment	Boolean	YES
Application Scene Manifest	Dictionary	(2 items)
Application supports indirect input events	Boolean	YES
Launch screen interface file base name	String	LaunchScreen
Main storyboard file base name	String	Main
Required device capabilities	Array	(1 item)
Supported interface orientations	Array	(3 items)
Supported interface orientations (iPad)	Array	(4 items)
App Transport Security Settings	Dictionary	(1 item)
Exception Domains	Dictionary	(1 item)
localhost	Dictionary	(1 item)
NSExceptionAllowsInsecureHTTPLoads	Boolean	1

Figure 7-13. Enabling HTTP for localhost

Now it's time to actually stub the network and update our test:

```
func testShowingBestSellerBooks() throws {
    // Given
    let testBundle = Bundle(for: type(of: self))
    let booksJSONURL = testBundle.url(forResource:
        "BestSellerBooksStub", withExtension: "json")
    let booksJSON = try String(contentsOf: booksJSONURL!)
    server.GET["/svc/books/v3/lists/overview.json"] = {_ in
        HttpResponse.ok(.text(booksJSON))}

    let app = XCUIApplication()
    app.launchArguments += ["TESTING"]
    app.launch()

    // When
    let booksTableView = app.tables
```

```

let cells = booksTableView.cells
_ = cells.firstMatch.waitForExistence(timeout: 1.0)

// Then
XCTAssertTrue(cells.staticTexts["book_title_0"].label
== "THE LAST THING HE TOLD ME")
XCTAssertTrue(cells.staticTexts["book_desc_0"].label
== "Hannah Hall discovers truths about her missing
husband and bonds with his daughter from a previous
relationship.")
XCTAssertTrue(cells.staticTexts["book_date_0"].label ==
"2021-05-26 22:10:24")

XCTAssertTrue(cells.staticTexts["book_title_1"].label
== "SOOLEY")
XCTAssertTrue(cells.staticTexts["book_desc_1"].label
== "Samuel Sooleymon receives a basketball scholarship
to North Carolina Central and determines to bring his
family over from a civil war-ravaged South Sudan.")
XCTAssertTrue(cells.staticTexts["book_date_1"].label ==
"2021-05-26 22:10:24")
}

```

In our test, first, we tell Swifter to stub our path and return the expected JSON so that we can assert on it inside the UI presented. We then launch our app with extra launch arguments to indicate that we're testing. Then we assert on the existence of the expected cells and assert on the data displayed as well.

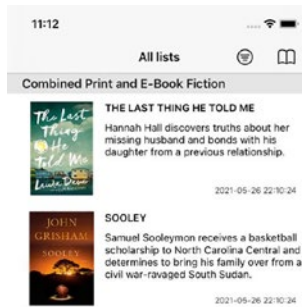


Figure 7-14. *Stubbed app*

This updated test should pass. But the important thing is that now, thanks to our network stubbing, we are able to assert on actual data in our UI (Figure 7-14). Later on, if we display something wrong, for example, this test will catch it.

Summary

When writing tests, we often find ourselves in a position where we need to assert on something that we don't have access to, and sometimes we need to control a certain behavior to avoid unpredictability. In these situations, our solution to all our problems is test doubles. A test double is any kind of fake object that we use in place of the real object, and they have many

forms and uses. In this chapter we talked about different types of test doubles. We also talked about how to create and inject doubles into our code being tested.

Stubs are one type of test doubles. A stub holds some predefined data and returns it instead of returning real data. This is useful in tests to improve speed and eliminate unpredictability. Another type of test doubles is mock objects. Mocks can also return fake data, but their main function is that they register calls made to them. And they can also register the values passed to them through function calls. This allows us to assert if a specific interaction between the object we're testing and our mock happened or not.

There are multiple ways to create our test doubles. We can create them using inheritance, where we would subclass the original class and then override and change the functions we want to stub or mock. Another way to create doubles is by using protocols. If our test object depends on a certain protocol, then we can create our double by creating a new component that conforms to that protocol and implementing the protocol's requirement.

As for injecting our doubles into our code to be tested, this is a fairly simple task. We can either inject it through the initializer of the object being tested; this is called initializer injection. Or we can use property injection, where we would create our object normally and then inject our double by accessing its property and assigning our double to it.

Finally, we explored a specific but highly important type of stubbing, which is network stubbing in the UI test layer. We used a third-party library to stub network requests. And that allowed us to write more comprehensive UI tests. At the same time, this increased the test's stability and speed.

CHAPTER 8

Avoiding Multithreading Nightmares

Concurrency and multithreading are a core part of iOS development. Understanding them and understanding how to properly leverage them is a key part of developing a high-quality app. Lack of concurrency usually leads to having nonresponsive apps that freeze up once a heavy operation is being performed.

What Is Concurrency?

The concept of concurrency is that two or more tasks can be defined independently and each task can be executed regardless if the other tasks are executing or not. This means that two or more tasks can be executed at the same time, in other words, executing concurrently.

Concurrency can be achieved in one of two ways, either by context switching (time slicing) or by parallelism. Which way is used depends on the type of processor. With a single-core processor, context switching is used, in which the system switches between threads quick enough that it virtually seems that both tasks are running at the same time. With a

multi-core processor, however, concurrency is achieved through actually running each thread on a separate core in parallel.

GCD

So far we have talked about threads and how it's possible to execute two or more tasks on separate threads at the same time. But threads are a low-level tool, and managing threads manually to achieve concurrency is a fairly complicated task

Grand Central Dispatch (GCD) was created by Apple and has been available since iOS 4. GCD basically abstracts the manual handling of threads away from the developer. It helps developers leverage the multithreading features of the system without actually having to create or manage threads themselves. Instead of creating threads, you use GCD to schedule tasks, and the system will execute these tasks in the most efficient way possible.

Queues

As mentioned before, GCD abstracts the handling of threads. So after this abstraction, what do you deal with? You deal with **dispatch queues**. You can deduce its functionality from its name. You submit tasks to a queue, and GCD will execute them in FIFO order (First In, First Out). Depending on the available resources, the type of queue used, and the dispatching function (function used to submit a task), GCD will decide when and on what thread this task will be executed.

We've been saying how great GCD is, and rightfully so. However, just using GCD does not guarantee bug-free code. The key is choosing the right type of dispatch queue and the right dispatching function.

Serial vs. Concurrent

Queues have two types, serial and concurrent:

A **serial queue** (Figure 8-1) guarantees that all tasks submitted to it run one after the other, meaning that first task has to finish in order for the second task to start. This means that a serial queue will not run on more than one thread.

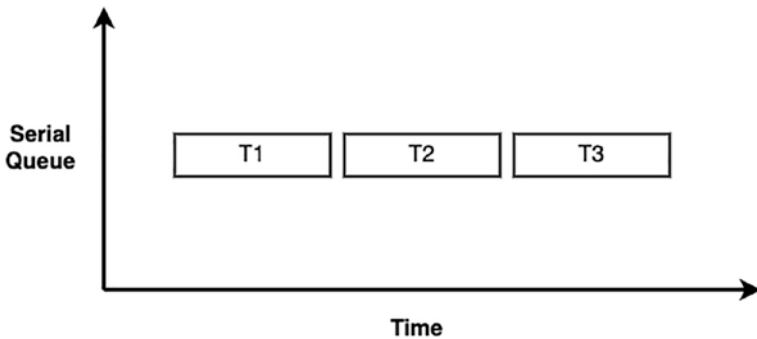


Figure 8-1. Serial queue tasks illustration

A **concurrent queue** (Figure 8-2) can run on more than one thread, meaning that the tasks submitted to it can run simultaneously. A very important distinction between a concurrent and serial queue is that a concurrent queue only guarantees FIFO order when it comes to starting the task. However, because the queue doesn't wait for tasks to finish before starting a new task, FIFO order is not guaranteed for the finishing of the tasks.

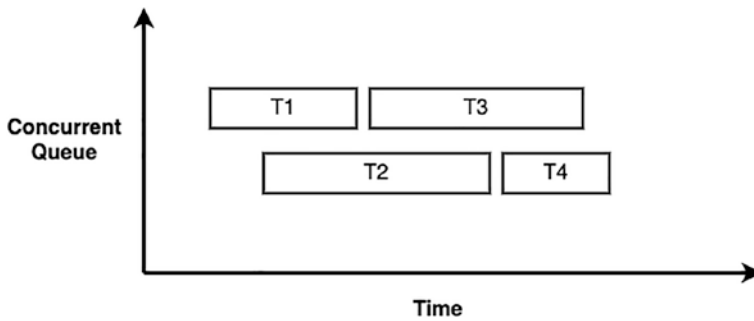


Figure 8-2. Concurrent queue tasks illustration

Sync vs. Async

When dispatching a task onto a queue, you can either dispatch it synchronously or asynchronously. Your choice of serial vs. concurrent affects the **destination**—the queue on which the task is submitted to run. This is contrary to sync vs. async, where your choice affects the **source**—the queue from which you submit the task.

When you use a **sync** statement (Figure 8-3), it will block the current queue (source) until the block is executed and finished. When it finishes, it returns control back to the caller, and the source queue can resume.

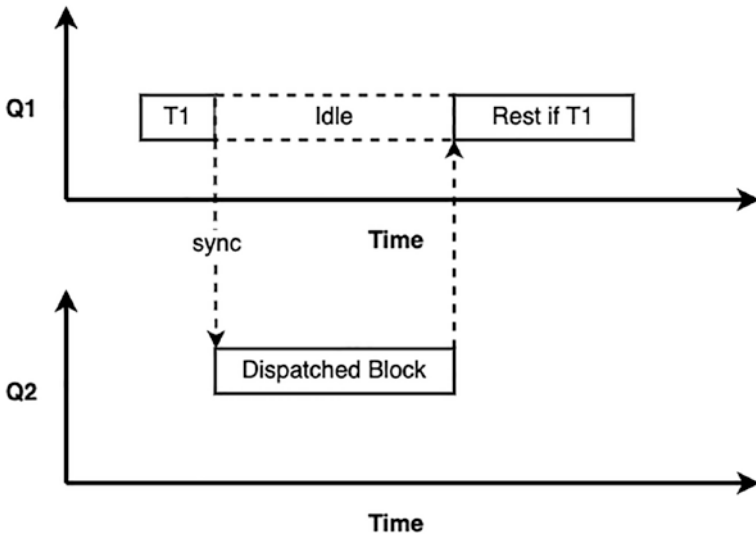


Figure 8-3. Sync task illustration

On the other hand, an **async** statement (Figure 8-4) gets executed asynchronously with respect to the current queue (source). Control is returned immediately to the caller, and the source queue is never blocked. And there's also no guarantee as to when exactly the block gets executed.

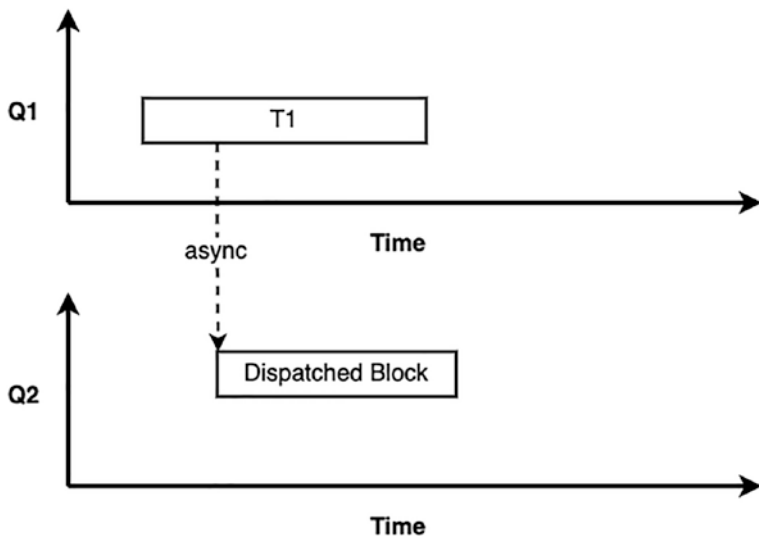


Figure 8-4. Async task illustration

Cost of Concurrency

GCD is meant to simplify the use of threads and add concurrency to the tasks performed by our app. And concurrency is meant to improve the performance of our app and ultimately lead to a highly responsive app even when performing heavy operations. But sadly there is a negative cost to concurrency, which means we can't just apply it whenever and wherever.

Concurrency is used to enhance the app's performance, but misusing it might actually lead to the exact opposite. Imagine having a very low-impact operation that we want to perform 10,000 times. You might think we have to use GCD to improve performance in this case. But if we create 10,000 tasks and submit them all to a queue, this will actually result in extremely high memory consumption and will negatively impact the allocation and deallocation of operation blocks. So in this case, while trying to enhance our performance, we actually end up degrading it. GCD is not a magical technology that enhances the performance regardless of any other factors. Just like any technology, it has its limitations. So it all

comes down to how GCD is used. It's up to you to use it in a way that is effective.

Other than introducing overhead on the system resources, using GCD also introduces some serious risks. One risk in particular is the risk of encountering a **deadlock**. In simple terms, a deadlock is a state where two threads are waiting on each other to finish so that they can resume. In the following figure, thread A is waiting on thread B to finish so that it can resume, and thread B is waiting on thread A to finish so it can resume. Since this means that neither can finish, then neither can resume. Which causes these two threads to be suspended indefinitely (Figure 8-5). This is a very common risk when working with multithreaded programming, and in turn it's very common when using GCD.

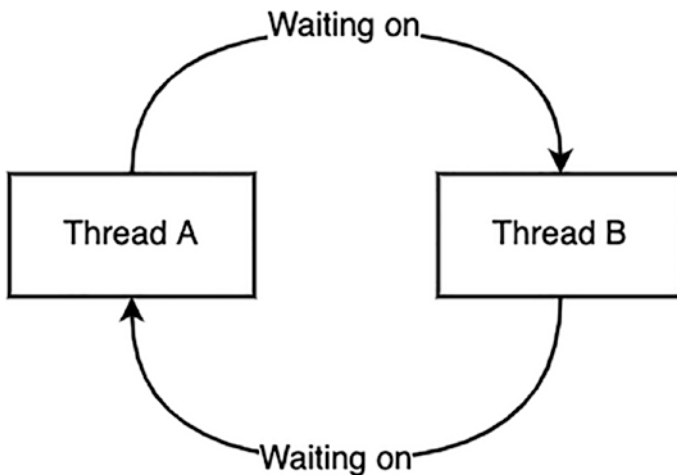


Figure 8-5. Deadlock

Other risks when working with GCD also include **race conditions**. Race conditions occur when two threads are trying to access or modify the same resource at the exact same time. The problem with race conditions is that it solely depends on when threads are scheduled to perform certain tasks, which, by the nature of GCD, is completely unpredictable. Which

makes identifying, debugging, and reproducing them really tricky. This is a synchronization problem and can be fixed using serial queues or dispatch barriers if we're using a concurrent queue. There are other ways to achieve synchronization, but we won't be discussing them in our book.

Reader-Writer Problem

There are many problems that can lead to race conditions. One of these is the reader-writer problem. It is one of the more common problems that we might find ourselves facing. This problem occurs when there is a shared resource and one thread is trying to **read** the shared resource and another thread is trying to **write** to it.

Let's talk about how we can identify that our code introduces this kind of problem. Our keyword here is **shared resource**. Once we have a shared resource that we're not handling properly, it's highly likely to cause a race condition. The first place to look for shared resources in any app is our infamous **singleton** classes.

Singleton Classes

Singleton classes are classes that can only have one instance. One instance is created and usually held statically in the class and then shared everywhere this object is needed. Creating a singleton in Swift is as simple as adding an empty private `init` to our class or struct, which makes sure our singleton can't be initialized from outside the class. And then we just add a static variable that holds the only created instance of this class. Here is an example of a singleton class:

```
struct TestStruct {
    static let shared = TestStruct()
    private init() { }
}
```


Due to their nature, singleton classes can be easily accessed from two threads at the same time because a single object serves our whole application. However, normal classes can have a shared resource between two threads as well. It all depends on how the objects of these classes are being handled and used and how each object handles its resources. Once we find a resource that we suspect, we need to ask ourselves, is this resource accessible from multiple threads? And can this resource be accessed (read) and modified (write)? If the answer to both these questions is yes, then we have found a potential race condition.

Identifying a Race Condition

First, let's take a look at the project **ReaderWriter**, which you can find in this chapter's resources. This is an empty project that has only one class `Database`, which was written using TDD:

```
public class Database {
    // MARK:- Singleton
    public static let shared = Database.shared

    // MARK:- Initializer
    private init() {}

    // MARK:- Private Variables
    private var dictionary: [String:Any] = [:]

    // MARK:- Public Functions
    public func addObject(_ object: Any, for key: String) {
        dictionary[key] = object
    }
}
```

```

public func removeObject(for key: String) {
    dictionary.removeValue(forKey: key)
}

public func object(for key: String) -> Any? {
    return dictionary[key]
}

public func recordsCount() -> Int {
    return dictionary.count
}

public func reset() {
    dictionary = [:]
}
}

```

This is a singleton class that acts as a very primitive database. It stores key/value pairs inside an internal dictionary. And it has some public APIs to interact with the database. There's an API to add a new record, an API to delete a record, an API to retrieve a record, and an API to get the current number of records.

This class has an internal dictionary. Could this resource cause a reader/writer data race? To answer this, let's ask our two questions for this resource:

1. Is this resource accessible from multiple threads?

Since Database is a singleton class, then it's highly possible for any of its public APIs to be called from multiple threads. Therefore, **yes**.

2. Can this resource be accessed (read) and modified (write)?

By looking at the public functions, we have `object(for key: String)` and `recordsCount()`, and both access our resource. We also have `addObject(_ object: Any, for key: String)` and `removeObject(for key: String)`, and both modify our resources. Therefore, **yes**.

The answers of these two questions tell us that this class is not thread-safe and could cause a race condition.

Applying TDD to the Problem

By now, once you read “TDD,” you should immediately think of the following cycle (Figure 8-6).

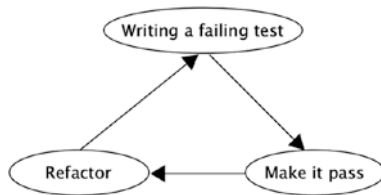


Figure 8-6. *The TDD cycle*

As always we will start with the first step, writing a failing test. We know that our code can cause a race condition when we try to read and write at the same time. So now our goal is to write a test that fails due to this problem.

Now let’s write our test. First, we’ll start by setting up our test. We need to create a new `Database` object and add to it a record that we’ll attempt retrieving later on in the test. Our **Given** should look like this:

```
// Given
let database = Database.shared
database.addObject("InitialValue", for: "InitialKey")
```

Next, we'll attempt to write to our database and read from it, in hopes that this will cause a race condition. Our **When** should look like this:

```
// When
database.addObject("Test", for: "Key1")
let _ = database.object(for: "InitialKey")
```

Finally, in our **Then** section of the test, we usually assert that the expected behavior actually happened. In our case, we actually have two assertions. We have an explicit assertion that the record was actually added. We don't actually care much about that assertion. What we care more about is our implicit assertion. If the test runs normally, this means that no race condition occurred, but if a race condition occurs, the test will crash and fail. This here acts as our implicit assertion. Our **Then** should look like this:

```
// Then
let count = database.recordsCount()
XCTAssertEqual(count, 2)
```

If we run the test we just wrote, it will actually pass. But why did it not cause a race condition and fail? Let's take a deeper look into the test we just wrote:

```
func testReadWriteDataRace() {
    // Given
    let database = Database.shared
    database.addObject("InitialValue", for: "InitialKey")

    // When
    database.addObject("Test", for: "Key1") // #1
    let _ = database.object(for: "InitialKey") // #2

    // Then
    let count = database.recordsCount()
```

```

    XCTAssertEqual(count, 2)
}

```

We know that this whole test is a single block. And we know that in the context of a block each line is executed one after the other (serially). So this means that the call to `addObject (#1)` is executed and finished and then the call to `object (#2)` is executed. Which means that the read and write operations never execute concurrently (Figure 8-7).

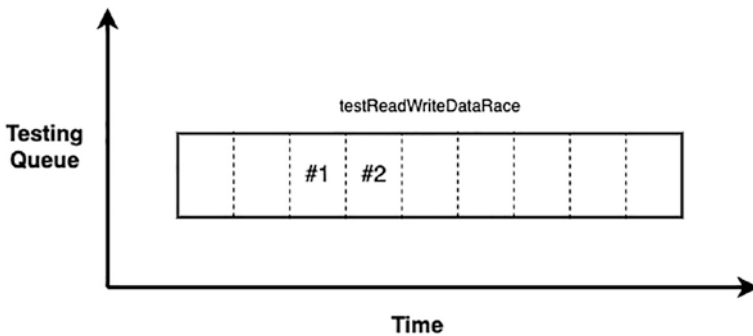


Figure 8-7. Test illustration

Now what we need to do is add concurrency between our two operations. We'll do that by using a concurrent queue. After modifying our test, it should look like this:

```

func testReadWriteDataRace() {
    // Given
    let queue = DispatchQueue(label: "com.ReaderWriterTests.
DatabaseTests", attributes: .concurrent)
    let database = Database.shared
    database.addObject("InitialValue", for: "InitialKey")

    // When
    queue.async { // #1
        database.addObject("Test", for: "Key1") // #2
    }
}

```

```

    }
    queue.async { // #3
        let _ = database.object(for: "InitialKey") // #4
    }

    // Then
    let count = database.recordsCount()
    XCTAssertEqual(count, 2) // #5
}

```

Here we create a new concurrent queue and give it a label. And in the `When` section, we dispatch both our operations asynchronously onto our concurrent queue. We use `async` not `sync` because if we use `sync`, as we mentioned before, this will affect the source, which is the thread the test is running on, meaning our test will be paused at the first `sync` call until it's finished and then we'll dispatch the second operation onto our concurrent queue. In this case, our two operations will never exist on the queue at the same time, which defeats the purpose.

If we try running our test now, it will fail. At first glance this is a good thing because that's what we were trying to reach. But when we actually look at the cause of failure, we'll find that our `XCTAssertEqual` fails. If you recall, we don't really care about this assertion as it doesn't indicate a race condition and it should pass in all cases. This means that the call to `addObject` was not executed, which means there's an issue with our test.

Let's take a look at what happens when we run our test (Figure 8-8). Because we dispatch our two operations using `async`, this means that the source thread is not blocked. And because it's not blocked, the test will immediately resume after we dispatch our operation onto our queue. If we look at our test, this means that it will immediately execute our **Then** section. And that's why the test fails. We execute our assertion before our operations are even executed.

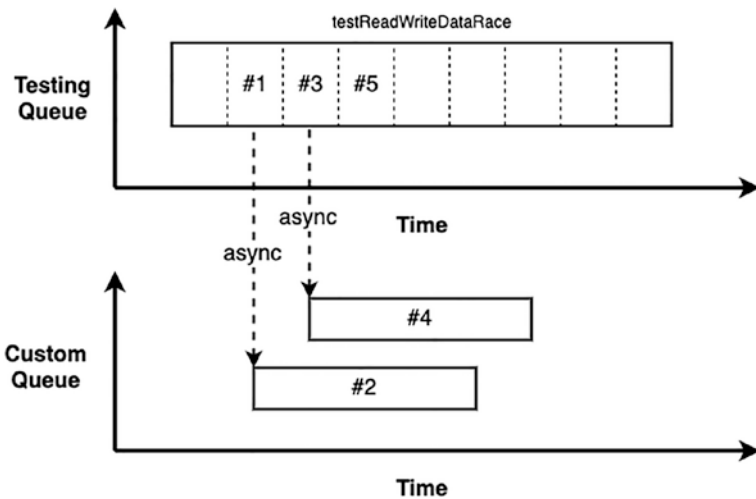


Figure 8-8. Test illustration

To fix our test, we'll need to block the test until our operations are done. We can't use `sync` as mentioned before. Instead, we need our test to wait after dispatching our two tasks. We can achieve this using `XCTestExpectation`. We'll create an expectation for each operation and fulfill them inside the `async` block. Then we'll wait for both expectations right before our assertion. Our test should look like this:

```
func testReadWriteDataRace() {
    // Given
    let queue = DispatchQueue(label: "com.ReaderWriterTests.
    DatabaseTests", attributes: .concurrent)
    let database = Database.shared
    database.addObject("InitialValue", for: "InitialKey")

    // When
    let exp1 = expectation(description: "Adding Key1 done")
    let exp2 = expectation(description: "Adding Key2 done")
```

```
queue.async {
    database.addObject("Test", for: "Key1")
    exp1.fulfill()
}
queue.async {
    let _ = database.object(for: "InitialKey")
    exp2.fulfill()
}
wait(for: [exp1, exp2], timeout: 1)

// Then
let count = database.recordsCount()
XCTAssertEqual(count, 2)
}
```

Now that we fixed the test and made it wait for the operations to be performed (Figure 8-9), let's run it again while looking out for the race condition we're looking for. When we run our test, it will (most probably) pass. This is both good and bad news. It's good news because it means the expectations we added actually did their job. But bad news because we now have two operations running on a concurrent queue, but they are still not being performed at the same time.

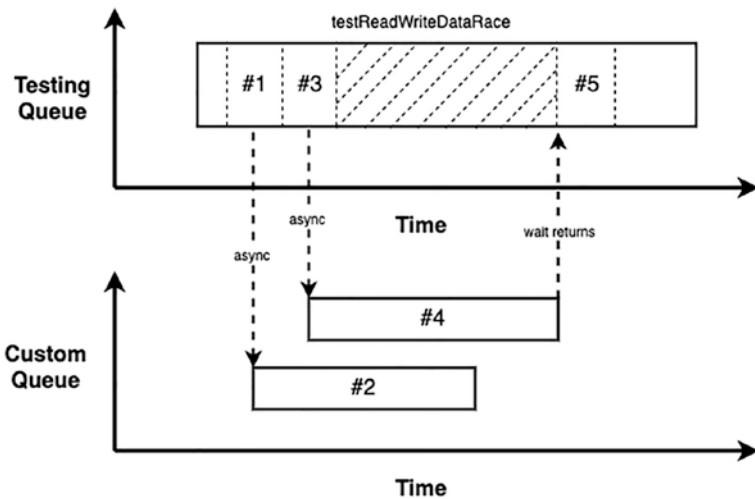


Figure 8-9. Test illustration

Where is our race condition? Does this mean that Database is thread-safe?

Actually the answer to these questions is hidden in the previous paragraph. If you take a look at it, you'll find that it's mentioned that the test will **most probably** pass. There's a reason why we're not 100% certain if the test will pass. This is because there's a very small chance that the race condition we're looking for actually happens. Since the queue we're using is concurrent, then GCD might decide to run each operation on a separate thread, leading to the race condition. But as mentioned before, the probability of this happening is extremely low. The reason for this is that the two operations in question are light operations and there are also only two tasks dispatched on the queue. And it's highly unlikely for GCD to decide to allocate an extra thread for our queue. And even if GCD allocates an extra thread, because of the nature of our operations and how light they are and how quick they take to finish, the probability of them being executed at the exact same time is really low as well.

So now what? We know that there's a very low probability that our test will fail (close to zero). Sadly this means that our test has little to no value. If the test doesn't fail even though it should, then there's no reason to even have that test. Luckily, there's still something we can do. We know that the probability of a race condition is extremely low because there's only two tasks on our queue and they're both light. Which means if we add more tasks to our queue, we will increase this probability. We can even increase this probability significantly till we reach a point where we're certain that a race condition will happen. Let's take a look at how we can do that:

```
func testReadWriteDataRace() {
    // Given
    let queue = DispatchQueue(label: "com.ReaderWriterTests.
    DatabaseTests", attributes: .concurrent)
    let database = Database.shared
    database.addObject("InitialValue", for: "InitialKey")

    // When
    var expectations:[XCTestExpectation] = []
    for i in 0..<500 {
        let key = "Key\(i+1)"
        let exp = expectation(description: "Adding \(key) done")
        queue.async {
            database.addObject("Test", for: key)
            exp.fulfill()
        }
        expectations.append(exp)
    }
    for i in 0..<500 {
        let key = "Key\(i+1)"
        let exp = expectation(description: "Adding \(key) done")
```

```

queue.async {
    let _ = database.object(for: "InitialKey")
    exp.fulfill()
}
expectations.append(exp)
}
wait(for: expectations, timeout: 10)

// Then
let count = database.recordsCount()
XCTAssertEqual(count, 501)
}

```

We simply modified our test so we would be performing 500 read operations and 500 write operations. By overloading our queue with this extremely high number of tasks, we're basically forcing GCD to allocate more than one thread for this queue, and due to the high number of reads and writes, it is almost certain that two of these operations are executed at the same time.

If we try running our test now, it will finally fail due to a race condition. 🍷

Thread Sanitizer

However, now we have a different problem. Our test now takes too much time. And if we try to reduce the number of iterations, we will reduce its accuracy in catching threading issues. Luckily, we have access to a tool in Xcode that can help us with this, the **Thread Sanitizer**.

The Thread Sanitizer, commonly referred to as TSan, is a tool Apple provides as part of the LLVM compiler. It helps in auditing threading issues in your Swift and C language written code. This sanitizer is able to detect when multiple threads attempt to access the same resource and at least

one of these accesses is a write operation. It's able to do that by rebuilding the whole app and adding checks around each memory access in your code. These checks record that a memory access occurred along with when it occurred and from which thread. And from that information, it's able to add a breakpoint whenever an illegal memory access occurs.

The beauty of the Thread Sanitizer is that it's able to detect the silent data races. In many cases, the same resource can be accessed and modified from different threads, but the threads miss collision by microseconds. Without the sanitizer, this scenario will go unnoticed as it won't cause misbehavior or a crash. However in other times, they may collide. This randomness is what makes threading issues so hard to debug. But with the Thread Sanitizer enabled, catching threading issues becomes far more likely to happen.

To enable the Thread Sanitizer, we need to go into our scheme configuration (Figure 8-10).

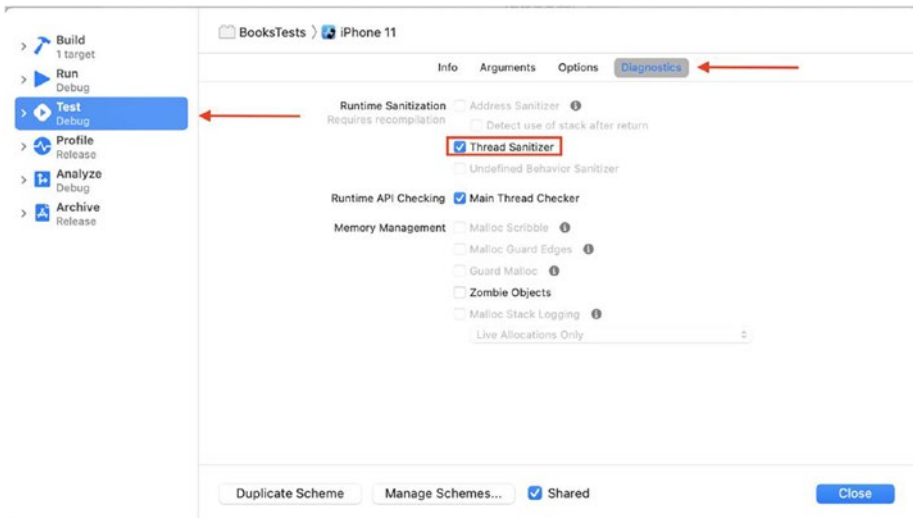


Figure 8-10. Enabling Thread Sanitizer

Let's enable the sanitizer for our Run and Test configurations.

Now in order to make Xcode pause whenever a data race is detected, we need to add **Runtime Issue Breakpoint**. We can add that from the Breakpoint navigator (Figure 8-11).

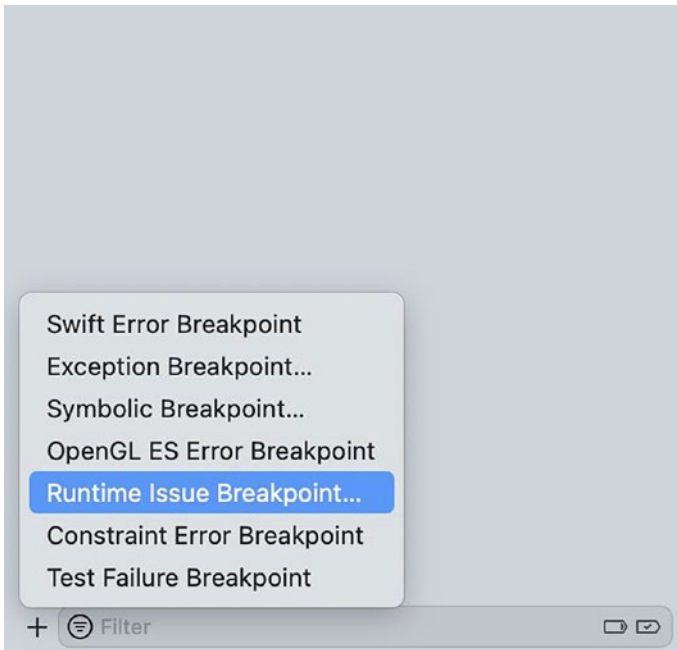


Figure 8-11. Adding Runtime Issue Breakpoint

Since we now have the Thread Sanitizer enabled, we can actually reduce the number of iterations a bit if we want.

Make It Pass

Now that we're finally done with the first step in the TDD cycle, it's time to fix the test whose failure we just celebrated. As mentioned before, race conditions are a problem of synchronization and can be fixed using many ways. Let's try fixing it using a serial queue. The goal is to leverage a serial

queue in order to achieve synchronization between all the operations performed by our database object. We need to make sure that only one operation is performed at any given time.

When we add the serial queue, our Database class should look like this:

```
public class Database {
    // MARK:- Singleton
    public static let shared = Database.shared

    // MARK:- Private Variables
    private var dictionary: [String:Any] = [:]
    private let queue = DispatchQueue(label: "com.ReaderWriter.
    Database")

    // MARK:- Public Functions
    public func addObject(_ object: Any, for key: String) {
        queue.sync {
            dictionary[key] = object
        }
    }

    public func removeObject(for key: String) {
        queue.sync {
            _ = dictionary.removeValue(forKey: key)
        }
    }

    public func object(for key: String) -> Any? {
        queue.sync {
            return dictionary[key]
        }
    }
}
```

```

public func recordsCount() -> Int {
    queue.sync {
        return dictionary.count
    }
}
public func reset() -> Int {
    queue.sync {
        dictionary = [:]
    }
}
}

```

Now let's try running our test once again. It should be passing ✓. Let's run the rest of our tests as well to make sure our change did not cause any regressions. Since all tests are passing and there's nothing that needs refactoring, this means we're done fixing this issue. So there you have it. We identified a problem in our code that had to do with multithreading, and we successfully applied TDD to fix this issue.

Fixing Threading Issues in Books

Books is the project introduced in Chapter 6. Currently, Books is a modularized app, but that doesn't mean it's bug-free. Luckily for us, when **Books** was being written, concurrency issues weren't a top priority. So we now have a chance to see a threading issue in a real app and attempt to fix it using TDD like we just did for our reader-writer problem in the Database class.

Let's open up the project, which can be found in this chapter's resources, and start looking for potential threading issues. If we look carefully, we'll find that there's one place in our code that could potentially be not thread-safe. That part is our extension on UIImageView that handles image caching:

```

extension UIImageView {
    static var dictionaryImageCache = [String:UIImage]()

    func load(url: URL) {
        DispatchQueue.global().async { [weak self] in
            if (UIImageView.dictionaryImageCache[url.path] !=
                nil) {
                DispatchQueue.main.async {
                    self?.image = UIImageView.
                        dictionaryImageCache[url.path]
                }
                return
            }

            if let data = try? Data(contentsOf: url) {
                if let image = UIImage(data: data) {
                    UIImageView.dictionaryImageCache[url.path]
                        = image
                    DispatchQueue.main.async {
                        self?.image = image
                    }
                }
            }
        }
    }
}

```

How that extension works is that it has a static dictionary that we store images in and that dictionary is accessible for all instances of `UIImageView`. This here indicates that data race could occur on the shared dictionary. To be certain, let's ask our two questions:

1. **Is this resource accessible from multiple threads?**

The extension by its nature is global and applies to all UIImageView instances, which means that we can call `load(url: URL)` from more than one thread easily.

2. **Can this resource be accessed (read) and modified (write)?**

By looking at the `load` function, what it does is that it accesses the dictionary to check if the image is available in cache to return it; if not, it loads the image and then modifies the dictionary to save the newly loaded image.

The answers of these two questions tell us that this extension is not thread-safe and could cause a race condition.

Applying TDD

The first step to TDD is to write a failing test. This test will be very similar to the one we ended up with in the previous example:

```
func testLoadImageMultiThreading() {
    // Given
    let queue = DispatchQueue(label: "com.
    ReaderWriterTests.DatabaseTests", attributes:
    .concurrent)
    let image = UIImageView()

    // When
    var expectations:[XCTestExpectation] = []
    for i in 0..<500 {
        let key = "Key\(i+1)"
```

```

    let exp = expectation(description: "Adding \(key)
    done")
    queue.async {
        image.load(url: URL(string: "https://
        storage.googleapis.com/du-prd/books/
        images/9781501171345.jpg")!)
        exp.fulfill()
    }
    expectations.append(exp)
}

for i in 0..<500 {
    let key = "Key\(i+1)"
    let exp = expectation(description: "Adding \(key)
    done")
    queue.async {
        image.load(url: URL(string: "https://
        storage.googleapis.com/du-prd/books/
        images/9781501171345.jpg")!)
        exp.fulfill()
    }
    expectations.append(exp)
}

wait(for: expectations, timeout: 10)
}

```

Now that we have a failing test that showcases that our code is not thread-safe, now it's time to fix our code. We can fix our extension using a serial queue like we did in the database example, but let's try something new. We'll use locks this time, which are a common method for ensuring synchronization:

```

extension UIImageView {
    // MARK:- Variables
    static var dictionaryImageCache = [String:UIImage]()
    static var lock = NSRecursiveLock()

    // MARK:- Functions
    func load(url: URL) {
        DispatchQueue.global().async { [weak self] in
            Self.lock.lock()
            if (Self.dictionaryImageCache[url.path] != nil) {
                DispatchQueue.main.async {
                    self?.image = Self.
                        dictionaryImageCache[url.path]
                }
                Self.lock.unlock()
                return
            }
            Self.lock.unlock()

            if let data = try? Data(contentsOf: url) {
                if let image = UIImage(data: data) {
                    Self.lock.lock()
                    Self.dictionaryImageCache[url.path] = image
                    Self.lock.unlock()
                    DispatchQueue.main.async {
                        self?.image = image
                    }
                }
            }
        }
    }
}

```

What we do here is that before accessing or modifying, we first acquire the lock by calling `lock()`. This makes sure that whenever any other thread tries to acquire the same lock, it will be forced to wait until the thread holding the lock lets go of it. We release the lock by calling `unlock()`.

Now if we run our test again (Figure 8-12), it will pass ✓.



Figure 8-12. Multithreading test passing

Summary

In this chapter you learned about some of the main concepts in multithreading programming. First of all was the concept of concurrency, which is that two or more tasks can be executed simultaneously on different threads. Concurrency is achieved in iOS by the use of Grand Central Dispatch (GCD). GCD abstracts the manual handling of threads away from the developer. Instead of creating threads, you create your tasks and dispatch on a queue, and GCD handles the low-level execution of these tasks on multiple threads if needed.

Dispatch queues are at the core of how GCD operates. When tasks are submitted to a queue, GCD will execute these tasks in First In, First Out order. However, we have two types of queues, serial and concurrent. Serial queues make sure that only one task in that queue is running at any given time, and when a task finishes, the next task in line starts. Concurrent queues, on the other hand, are able to run more than one task at the same time.

When submitting a task to a queue, we can submit it using `sync` or `async`. This distinction affects the **source** queue (queue that performs the dispatch), not the destination queue (queue to which the task is

submitted). When using `sync`, this blocks the calling queue until the task is completed. While using `async`, this calling queue continues normally.

Concurrency has many benefits especially when it comes to performance. Since we're able to perform multiple tasks at the same time and not block one task by another heavy task, this naturally leads to enhancement in our app's performance. But this is actually not always the case, as overusing concurrency can degrade the performance due to unnecessary high memory consumption.

Apart from negative performance impact, concurrency has some serious drawbacks. When not used correctly, concurrency can result in bugs and crashes. When two operations depend on the same shared resource and get executed concurrently relative to each other, this can cause an array of problems. We might encounter a deadlock or even a data race condition.

Race conditions occur when one thread is modifying a resource and another thread is either trying to read the same resource or trying to modify it as well. Race conditions tend to lead to unexpected behavior that's extremely hard to debug and in some cases might cause crashes. In this chapter we looked at how we can apply TDD to fix our data races. We apply the first step in TDD by adding a special type of test that tests our code in a multithreaded environment (plus the use of the `Thread Sanitizer`). And then we go about the second and third steps in the TDD cycle as we would normally do.

CHAPTER 9

Testing Your Network

Most apps these days will communicate with the Internet at some point. Purely local apps are great, but communicating with a web service can help transform your app to a truly extraordinary app. There is a huge collection of diverse public web services that your app can make use of. And you can also hook into your own private web service to provide an expanded set of features to your users that you just can't provide if your app is purely local.

Networking ABCs

When one thinks of the Internet, many things might come to mind. One of them is “**www**.” This stands for the **World Wide Web**, which is the information system that we're able to access through the Internet. From its name, this system is worldwide. For something this big to evolve to what we know today, it had to be governed by some agreements so that it can be accessed by all the machines worldwide. So for two machines to communicate, they do so using defined protocols. A protocol is basically a contract between two parties with an established set of rules that dictate how data is transferred between different devices.

HTTP Requests

Hypertext Transfer Protocol (HTTP) is a protocol that allows transfer of resources between two parties.

An HTTP request normally contains the following:

- URL: Which identifies the resource we want.
- HTTP method: Which states the type of action that can be performed.
- Headers (optional): These are key/value pairs that allow us to pass additional information to the server.
- Data (optional): This can be in multiple forms, for example, JSON. Often referred to as the body of the request.

There are various options for HTTP methods:

- GET: For fetching a resource
- POST: For creating or updating a resource
- PATCH: For modifying a resource
- PUT: For replacing a resource
- DELETE: For deleting a resource

HTTP Responses

When you make an HTTP request to a server, the server returns a response.

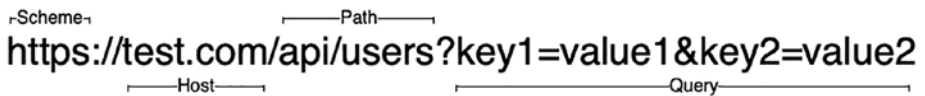
A response usually contains the following:

- Status code: This is a number that tells you whether your request succeeded or failed due to some error.

- **Headers:** This is similar to headers in the request. They carry additional information about the response.
- **Data:** This is also similar to the data in the request. This carries the data you requested if any. It can be in many forms, but most servers return data in JSON format.

URL

A Uniform Resource Locator (URL) is basically an address to a unique resource. This resource can be an HTML page, an image, JSON data, etc.



`https://test.com/api/users?key1=value1&key2=value2`

Figure 9-1. URL components

A basic URL has different components (Figure 9-1):

- **Scheme:** Which indicates the protocol the client must use to access the resource.
- **Host:** Which is usually a domain name (but an IP address can also be used), which indicates the server we are communicating with.
- **Path:** Which identifies the specific resource we are requesting from the server.
- **Query (optional):** With this, we can add extra parameters that the server might use to further process the resource before returning it.

There are other possible parts, but they are not relevant for basic usage. So we won't be discussing them.

Networking in iOS

As mentioned before, almost every app that's worth its salt will make a network request at some point. Which makes performing network requests a skill that any iOS developer must master. Many developers nowadays rely on third-party libraries to handle their network calls. Some of these libraries are quite powerful, but in many cases they can be an overkill, and by using them, you will just be adding an external dependency, which is always a risk. Instead, we can use the native **iOS URL Loading System**, with the main component being `NSURLSession`. `NSURLSession` is a part of a collection of classes that work together to handle network requests.

Let's talk about the most important components in the **iOS URL Loading System** in detail (Figure 9-2).

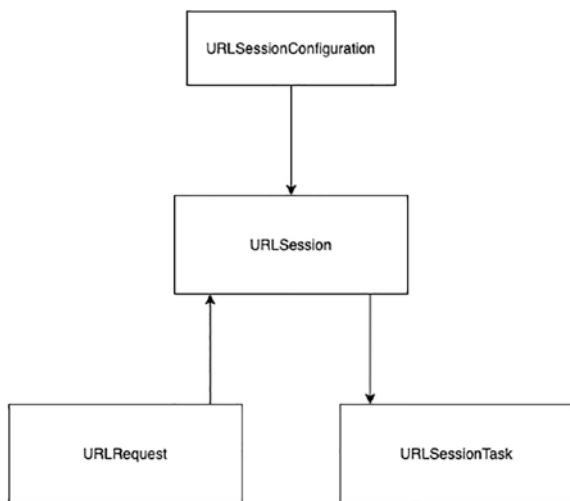


Figure 9-2. *iOS URL Loading System*

URLSession

The session is a core concept of HTTP. You can think of a session as an open tab or window of your web browser, through which you make multiple network requests. Loading a single web page can be fetching many resources through multiple requests under the hood, to be able to render the page. These requests are made using a single session as they share multiple things. `URLSession`, from the name, is used to manage an HTTP session. Making multiple requests using the same `URLSession` allows us to share configurations and cached data among requests.

URLSessionConfiguration

Since we just mentioned that requests made with the same session share the same configuration, let's talk about the object that holds these configurations. A `URLSessionConfiguration` defines the behavior and policies used when making requests using a `URLSession`. We can use it to set the timeout values, caching policies, connection requirements, etc. There are three types of `URLSessionConfiguration`:

- **Default:** A session configuration that uses disk-persistent storage for caches, cookies, or credentials.
- **Ephemeral:** A session configuration that uses no persistent storage for caches, cookies, or credentials.
- **Background:** A session configuration that allows app uploads or downloads to be performed in the background, even when the app itself is suspended or terminated.

URLRequest

We've already discussed the components that make up an HTTP request: URL, HTTP method, headers, and data. `URLRequest` is a structure that encapsulates all these components that describe a single request.

URLSessionTask

This is what actually performs the request. Normally we don't directly use `URLSessionTask`, but we use one of its subclasses. There are four native types of tasks:

- **Data task:** This type of task is able to send and receive data. It's the most common type of task and is used when sending or requesting JSON, for example.
- **Upload task:** This type of task is similar to a data task, but it also supports uploading data in the background.
- **Download task:** This type of task is able to download data from a server and directly write it to a file on disk. You can also track the download progress and can pause and resume the download.
- **Stream task:** This type of task provides a stream of data by establishing a connection with the server.

We don't create the tasks directly. Instead, we use one of the functions inside `URLSession` to create a new task. Once we create a task, we start it by calling the `resume()` function.

Networking in Books

The main premise of our app is that we fetch a list of bestseller books and display it for our users. A substantial part of our functionality depends on networking. And we might need to add new features in the future that depend on networking. However, when we take a look at our network layer, we'll find that it's highly coupled with a specific request to the single resource we're requesting from our server. We'll also find out that our network layer is not properly covered by tests.

Luckily, this is fixable, and this will be our aim during the rest of this chapter. Our goal will be to create a generic enough network layer (Figure 9-3), so that it becomes easy to reuse for different requests. And as always, we will implement this refactor using a test-driven approach. You can find the project in this chapter's resources.

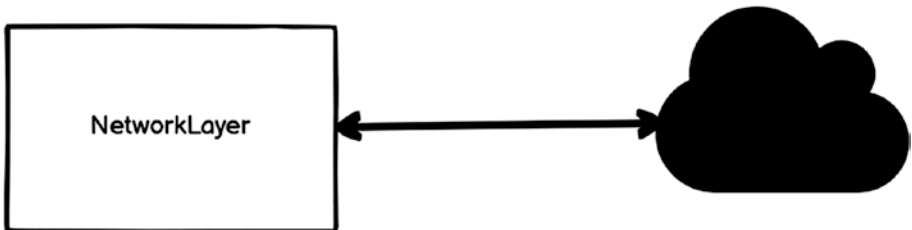


Figure 9-3. Network Module

Process Overview

Our network layer is already separated from the rest of our code. What we're attempting to do here is just to refactor it. Even though what we're attempting is not considered modularization, we will apply the same principles and almost the same process outlined in Chapter 6 (Figure 9-4). Let's take a look at this process and see what we can apply in our case.

Modularization process

- Step 1: Choose class as a starting point
- Step 2: Identify class's responsibilities
- Step 3: For each responsibility:
 - Step 1: Add verification tests
 - Step 2: Refactor related code
 - Step 3: Rerun verification tests
- Step 4: Repeat from step 1 by choosing a new starting point

Figure 9-4. Modularization process

First of all, we don't really have to choose a starting point, because we only have one point, which is the class `NetworkLayer`. Which makes step 1 and step 4 redundant.

Identify the Class's Responsibilities

The class in question is the `NetworkLayer` class. If we scan this class, we'll find that it has only one responsibility, which is executing a request to fetch books from our server. However, as we already mentioned before, we need to tweak this responsibility a bit and make it more generic. We want to make this class able to make any request, not just fetch our books from the server. We'll attempt to achieve this in the next steps.

Design Overview

Now before we start refactoring, let's take a closer look at what `NetworkLayer` does internally. We can see that it's doing a lot of things. First, it's storing a lot of static information related to the environment like host, API key, etc. Second, it contains information about creating the request itself, which is messy, and we already know we want to make it more generic. Third, it creates a `URLSessionTask` and executes the request. A good idea would be to separate all these coherent tasks.

NetworkLayer Tasks to Be Refactored

1. Storing static information
2. Creating a URL request
3. Creating a URLSessionTask and executing the request

NetworkLayer New Design

We should think about how the new objects will interact with each other without looking at the current code in order not to be affected by the current implementation. Our design can be something like Figure 9-5. NetworkLayer will use APIEnvironment to get all static data. RequestProtocol is a protocol that encapsulates creation of URLRequest. So, whenever we want to make a network call, we need to implement RequestProtocol. NetworkLayer will now be able to create a URLSessionTask and execute the request.

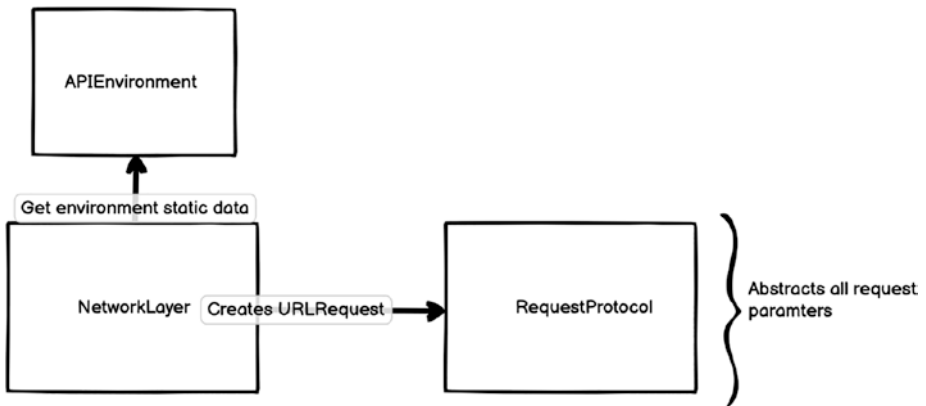


Figure 9-5. Network layer design

Kickoff

Since we've identified the responsibilities of our `NetworkLayer` class and also identified how these responsibilities should be modified, it's time to start refactoring.

From our modularization process, we know we need to follow these steps when refactoring:

1. Add verification tests.
2. Refactor related code.
3. Rerun verification tests.

Verification Tests

So let's start with our verification test. We need to verify that the functionality of `MainViewModel` is exactly the same after the refactor. `MainViewModel` fetches the list of books by consuming `NetworkLayer`, which will make a network request call and return the list of books to `MainViewModel` through a callback. We need our verification test to verify that this flow is working fine after refactor. By looking at our current test suite, we can see that `testFetchBestSellerBooks` inside `MainViewIntegrationTests` can act as our verification test. Also if we go up a level, we'll find that we have UI tests covering this as well. Now if we break anything in integration between `MainViewModel` and `NetworkLayer`, our tests will fail. We can now refactor the `NetworkLayer` with confidence.

Make a Network Request

Let's start by writing a test for this. We'll create a new test case class and call it `NetworkLayerTests` and add this to it:

```

func testExecutingSuccessfulRequest() {
    // Given
    let network = NetworkLayer()
    let request = TestRequest()
    let env = APIEnvironment.production

    // When
    let expectation = XCTestExpectation(description: "Request
    is done")

    network.executeRequest(request, callback: {
        expectation.fulfill()
    })

    self.wait(for: [expectation], timeout: 0.1)

    // Then
    // Missing assertion
}

```

To fix the build errors, let's add some code. First, let's add the new API in `NetworkLayer` but leave the implementation empty for now:

```

public func executeRequest<T: RequestProtocol>(_ request: T,
    callback: @escaping NetworkCompletion) {
    // We'll leave it empty for now
}

```

And add this type alias at the top of the file outside the scope of the class:

```

typealias NetworkCompletion = () -> Void

```


RequestProtocol

Now in order to use the new API, we'll always need to pass an instance that conforms to our request protocol. This instance will carry all the info we need to make a request. It's time to define this protocol. Let's create a new file and add this inside:

```
enum HTTPMethod: String {
    case GET
    case POST
    case PATCH
    case PUT
    case DELETE
}

protocol RequestProtocol {
    var method: HTTPMethod { get }
    var body: Data? { get }
    var path: String { get }
    var queryItems: [URLQueryItem]? { get }
}
```

Right now our test is still not building. This is because we need to create our `TestRequest` struct inside our test target and make it conform to `RequestProtocol`:

```
import Foundation
@testable import Books

struct TestRequest: RequestProtocol {
    var method: HTTPMethod {
        return .GET
    }
}
```

```

var body: Data? {
    return "Request Data".data(using: .utf8)
}

var path: String {
    return "/api/mock"
}

var queryItems: [URLQueryItem]? {
    return [URLQueryItem(name: "offset", value: "20")]
}
}

```

We can add a very useful function to `RequestProtocol` through an extension, which is creating a URL describing the request. This will keep our code clean. The URL creation still needs the host and scheme to be defined. This won't change between requests; however, it will change between production and testing environments. We'll extract this info to be saved into a new component.

Let's start by writing a test for URL request creation and see how it goes. We'll add a new test case class and name it `RequestProtocolTests`, and we'll add this test to it:

```

func testCreateURLRequest() {
    // Given
    let environment = APIEnvironment(scheme: "http", host:
    "test.com", port: 433, API_KEY: "KEY")
    let request = TestRequest()

    // When
    let urlRequest = request.createURLRequest(with:
    environment)
}

```

```

// Then
XCTAssertEqual(urlRequest?.url?.absoluteString, "http://
test.com:433/api/mock?offset=20")
XCTAssertEqual(urlRequest?.httpMethod, "GET")
XCTAssertEqual(urlRequest?.httpBody, "Request Data".
data(using: .utf8))
}

```

For this test to build, we'll need to create `APIEnvironment`. We'll skip the detailed TDD steps for this component as it is very simple. But after multiple TDD cycles, we should end up with this struct:

```

struct APIEnvironment {
    let scheme: String
    let host: String
    let port: Int?
    let API_KEY: String

    static let production: APIEnvironment = .init(scheme:
"https", host: "api.nytimes.com", port: nil, API_KEY:
"YOUR_API_KEY")
    static let testing: APIEnvironment = .init(scheme: "http",
host: "localhost", port: 8080, API_KEY: "KEY")
}

```

This just encapsulates the scheme and host of the environment. And also we included the API key. And we added two preset instances as static variables.

To have our create URL request test pass, we'll need to add this extension:

```

extension RequestProtocol {
    func createURLRequest(with environment: APIEnvironment) ->
        URLRequest? {
        guard let url = createURL(with: environment) else {
            return nil
        }
        var request = URLRequest(url: url)
        request.httpMethod = method.rawValue
        request.httpBody = body
        return request
    }

    private func createURL(with environment: APIEnvironment) ->
        URL? {
        var components = URLComponents()
        components.scheme = environment.scheme
        components.host = environment.host
        components.port = environment.port
        components.path = path
        components.queryItems = queryItems
        return components.url
    }
}

```

We can add this anywhere, but it makes sense to keep it in the same file as the protocol.

Execute Request

After this detour to add the create URL function to `RequestProtocol`, let's get back on track. If we run `testExecutingSuccessfulRequest`, we'll find that it's failing due to the expectation never being fulfilled. Let's fix this by simply calling the completion handler inside `executeRequest`:

```
public fun executeRequest<T: RequestProtocol>(_ request: T,
    callBack: @escaping NetworkCompletion) {
    callBack()
}
```

Now our test will pass ✓✓.

Mocking URLSession

We need to update our test, since now it's passing even though we don't even make any network requests. To be able to test what `NetworkLayer` does internally, we need to insert a mock of a `URLSession` and use it to assert that the request is performed. Let's create our mock:

```
class URLSessionMock: URLSession {
    typealias CompletionHandler = (Data?, URLResponse?, Error?)
    -> Void

    public var stubbedData: Data?
    public var request: URLRequest?

    override fun dataTask(with request: URLRequest,
        completionHandler: @escaping CompletionHandler) ->
        URLSessionDataTask {
        let data = self.stubbedData
        self.request = request

        return URLSessionDataTaskMock {
            completionHandler(data, nil, nil)
        }
    }
}
```

```

class URLSessionDataTaskMock: URLSessionDataTask {
    private let closure: () -> Void

    init(closure: @escaping () -> Void) {
        self.closure = closure
    }

    override func resume() {
        closure()
    }
}

```

Here we create a test double to be able to mock a `URLSession`. We create this mock by subclassing `URLSession` and overriding the function that creates a data task. We override it so that it does two things: first, it saves the inputs that are passed to it, and second, it returns an instance of `URLSessionDataTaskMock`, which is a test double for `URLSessionDataTask`. Instead of making the network request, this mock executes a block that's passed to it. We use this block to identify if the task was run or not.

Now that we've created our test double, it's time to inject it into our `NetworkLayer` instance in our test:

```

func testExecutingSuccessfulRequest() {
    // Given
    let session = URLSessionMock()
    let network = NetworkLayer(session: session)
    let request = TestRequest()
    let env = APIEnvironment.production

    // When
    let expectation = XCTestExpectation(description: "Request
    is done")

```

```

    network.executeRequest(request, callback: {
        expectation.fulfill()
    })

    self.wait(for: [expectation], timeout: 0.1)

    // Then
    // Missing assertion
}

```

Now we need to update our class to accept this injection of a `URLSession`. We'll add this variable to our class:

```
let session: URLSession
```

And add a new initializer:

```

init(session:URLSession = .shared) {
    self.session = session
}

```

Here we pass a session in the initializer and save it in a local variable so we can use it to make requests. If no custom session is passed, we default to the shared session.

Now that we have successfully injected our mock, we can now update our test to actually assert on the creation and running of a data task:

```

func testExecutingSuccessfulRequest() {
    // Given
    let expectedData = "Sample Data".data(using: .utf8)
    let session = URLSessionMock()
    session.stubbedData = expectedData // #1
    let network = NetworkLayer(session: session)
    let request = TestRequest()
    let env = APIEnvironment.production
}

```

```

// When
let expectation = XCTestExpectation(description: "Request
  is done")
var actualData: Data?
var actualError: APIError?

network.executeRequest(request, callback: { data, error in
// #2
  actualData = data // #
  actualError = error
  expectation.fulfill()
})

self.wait(for: [expectation], timeout: 0.1)

// Then
XCTAssertNotNil(session.request)
XCTAssertEqual(session.request?.httpMethod, "GET")
XCTAssertEqual(session.request?.httpBody, "Request Data".
data(using: .utf8))
XCTAssertEqual(session.request?.url, request.
createURLRequest(with: env)?.url)
XCTAssertEqual(expectedData, actualData)
XCTAssertNil(actualError)
}

```

In this test, we do a couple of things:

1. Here we tell our session mock what data to return when a request is made.
2. We change our block since we now expect our function to return data.
3. Here we save the returned data and error so we can assert on them later on.

And in our **Then** section, we assert that the request that's passed to the session is created correctly and has the correct method, body, and URL. We also assert that the data returned is the data expected. We also assert that there is no error returned.

Using URLSession

To fix this test, we need to utilize the URLSession and actually make the request. First, we'll need to add the following so that NetworkLayer can provide the base URL for our request:

```
public static var environment: APIEnvironment {
    return isTesting() ? .testing : .production
}
```

Here we make use of our two APIEnvironment instances we already created. And we return one of them based on the current environment.

We'll need to update our NetworkCompletion type alias and also add our error enum:

```
typealias NetworkCompletion = (Data?, APIError?) -> Void
enum APIError: Error {
    // We'll leave it empty for now
}
```

Then we'll update our function to actually make the request:

```
public func executeRequest<T: RequestProtocol>(_ request: T,
    callBack: @escaping NetworkCompletion) {
    guard let urlRequest = request.createURLRequest(with: Self.
        environment) else {
        return
    }
}
```

```

let task = self.session.dataTask(with: urlRequest) { data,
response, error in
    guard let data = data else {
        return
    }

    callback(data, nil)
}

task.resume()
}

```

After these changes, our test will pass ✓.

Showcasing Test Value

To showcase the value of our test, if for any reason we don't call `task.resume()`, which can happen if we refactor our code moving forward, our test will fail. We can simulate this by commenting out the call to `task.resume()` and running our test. It will fail as you see in Figure 9-6.

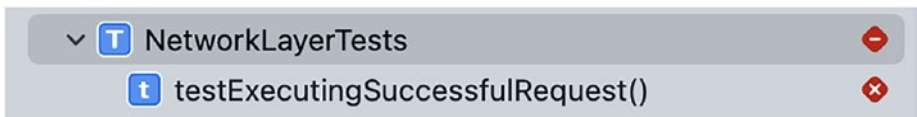


Figure 9-6. *Failing test*

Handle a Failing Request

Now that we've covered making a successful request with a test case, let's write test cases for failure scenarios. In this chapter, we'll cover two situations where our request might fail. First is if the server doesn't return data at all. The second is a client-side failure, which happens if we don't provide a valid URL to perform the request on. So let's write these two tests.

Let's add this test that simulates the server returning no data and an error:

```
func testExecutingFailedRequest() {
    // Given
    let session = URLSessionMock()
    session.stubbedData = nil
    let network = NetworkLayer(session: session)
    let request = TestRequest()
    let env = APIEnvironment.production

    // When
    let expectation = XCTestExpectation(description: "Request
    is done")
    var actualData: Data?
    var actualError: APIError?

    network.executeRequest(request, callback: { data, error in
        actualData = data
        actualError = error
        expectation.fulfill()
    })

    self.wait(for: [expectation], timeout: 0.1)

    // Then
    XCTAssertNotNil(session.request)
    XCTAssertEqual(session.request?.httpMethod, "GET")
    XCTAssertEqual(session.request?.httpBody, "Request Data".
    data(using: .utf8))
    XCTAssertEqual(session.request?.url, request.
    createURLRequest(with: env)?.url)
    XCTAssertNil(actualData)
    XCTAssertEqual(actualError, .requestFailed)
}
```

Here we tell our mock session to not return any data, and then we assert that `NetworkLayer` correctly handles this scenario by checking the value of the returned error. This test will fail for now. Let's add the second test and then we'll fix both.

We want to add another test that simulates the scenario where we attempt to make an invalid request. To create this test, we need to create a new struct that conforms to `RequestProtocol`. This struct should describe our invalid request:

```
struct InvalidRequest: RequestProtocol {
    var body: Data? {
        return nil
    }

    var path: String {
        return "INVALID PATH"
    }

    var queryItems: [URLQueryItem]? {
        return nil
    }

    var method: HTTPMethod {
        return .GET
    }
}
```

Now let's add our test:

```
func testExecutingRequestWithInvalidURL() {
    // Given
    let session = URLSessionMock()
    session.stubbedData = "Sample Data".data(using: .utf8)
    let network = NetworkLayer(session: session)
    let request = InvalidRequest()
```

CHAPTER 9 TESTING YOUR NETWORK

```
// When
let expectation = XCTestExpectation(description: "Request
is done")
var actualData: Data?
var actualError: APIError?

network.executeRequest(request, callback: { data, error in
    actualData = data
    actualError = error
    expectation.fulfill()
})

self.wait(for: [expectation], timeout: 0.1)

// Then
XCTAssertNil(session.request)
XCTAssertNil(actualData)
XCTAssertEqual(actualError, .invalidRequest)
}
```

To make our tests pass, we'll have to modify the `executeRequest` function to handle these two scenarios.

First, we'll need to add two cases to `APIError`:

```
enum APIError: Error {
    case requestFailed
    case invalidRequest
}
```

And we'll need to change the implementation of `executeRequest` to this:

```

public func executeRequest<T: RequestProtocol>(_ request: T,
callback: @escaping NetworkCompletion) {
    guard let urlRequest = request.createURLRequest(with: Self.
environment) else {
        callback(nil, .invalidRequest)
        return
    }

    let task = self.session.dataTask(with: urlRequest) { data,
response, error in
        guard let data = data else {
            callback(nil, .requestFailed)
            return
        }

        callback(data, nil)
    }

    task.resume()
}

```

When we detect that the request is invalid or the request failed, we call our completion handler with the appropriate value of type `APIError`.

Putting It All Together

Since all our tests are passing, then it's time to use the new API in our app. Let's take a look at a high-level overview of how our design should look like (Figure 9-7).

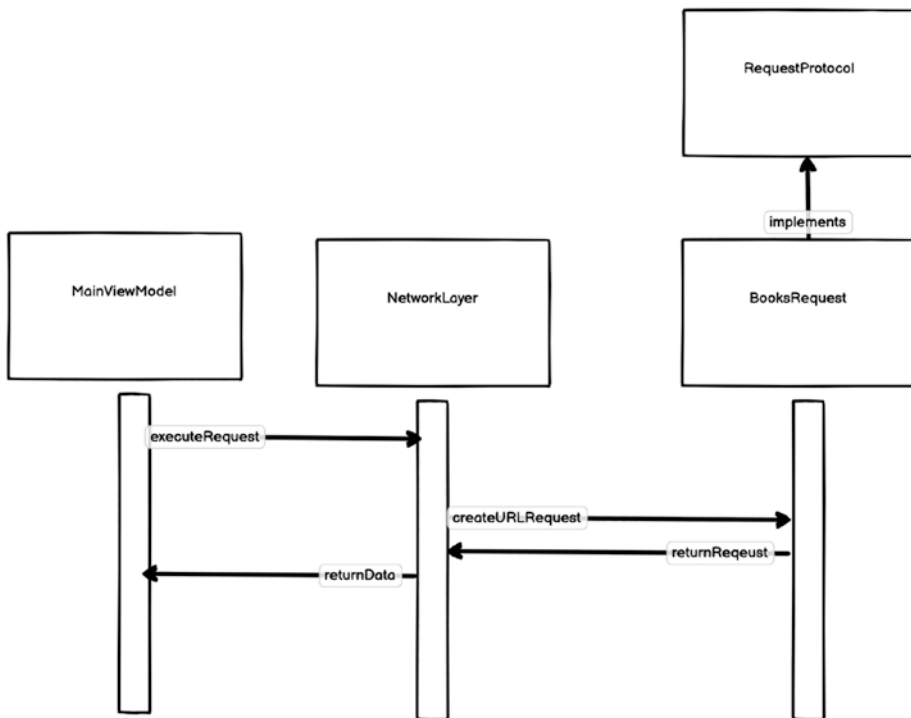


Figure 9-7. Integrating new network layer

Our `MainViewModel` will use the public API of `NetworkLayer` that is `executeRequest` and will pass it a request. This request will be of type `BookRequest`. Then `NetworkLayer` will use the passed books request to retrieve the required information and then execute the request. When the request is done, `NetworkLayer` will return the response back to the view model through the callback.

Given this overview, we know we need to create a new component, which is `BookRequest`. Let's create a test case class for it and name it `BookRequestTests`. And we'll add these tests in it:

```

func testBookRequestHTTPMethod() {
    //Given
    let bookRequest = BooksRequest()
}

```

```

    //When & Then
    XCTAssertEqual(bookRequest.method, .GET)
}

func testBookRequestURL() {
    //Given
    let bookRequest = BooksRequest()
    let env = APIEnvironment(scheme: "http", host: "test.com",
        port: 433, API_KEY: "")

    // When
    let urlRequest = bookRequest.createURLRequest(with: env)

    //When & Then
    XCTAssertEqual(urlRequest?.url?.absoluteString,
        "http://test.com:433/svc/books/v3/lists/overview.
        json?offset=20&api-key=\(APIEnvironment.production.API_
        KEY)")
}

func testBookRequestBody() {
    //Given
    let bookRequest = BooksRequest()

    //When & Then
    XCTAssertNil(bookRequest.body)
}

```

To make these tests pass, we'll have to actually add `BookRequest` and have it conform to `RequestProtocol`:

```

struct BooksRequest: RequestProtocol {
    var path: String {
        return "/svc/books/v3/lists/overview.json"
    }
}

```



```

var queryItems: [URLQueryItem]? {
    return [URLQueryItem(name: "offset", value: "20"),
            URLQueryItem(name: "api-key", value: NetworkLayer.
                environment.API_KEY)]
}

var method:HTTPMethod {return .GET}

var body: Data? {return nil}
}

```

Now let's remove our old code from `NetworkLayer` and fix the build error that will arise by using the new API along with the newly created `BookRequest`. Our `NetworkLayer` class should finally look like this:

```

typealias NetworkCompletion = (Data?, APIError?) -> Void

enum APIError: Error {
    case requestFailed
    case invalidRequest
}

class NetworkLayer {

    // MARK:- Variables
    let session: URLSession
    static var environment: APIEnvironment {
        return isTesting() ? .testing : .production
    }

    // MARK:- Initializer

    init(session:URLSession = .shared) {
        self.session = session
    }
}

```

```

// MARK:- Public Functions

public func executeRequest<T: RequestProtocol>(_ request:
T, callback: @escaping NetworkCompletion) {
    guard let url = request.createURL(with: Self.
environment) else {
        callback(nil, ..invalidRequest)
        return
    }

    var urlRequest = URLRequest(url: url)
    urlRequest.httpMethod = request.method.rawValue
    urlRequest.httpBody = request.body

    let task = self.session.dataTask(with: urlRequest) {
data, response, error in
        guard let data = data else {
            callback(nil, .requestFailed)
            return
        }

        callback(data, nil)
    }

    task.resume()
}

// MARK:- Helper Functions

static func isTesting() -> Bool {
    return ProcessInfo.processInfo.arguments.
contains("TESTING")
}
}

```

Now we'll have to make two changes. First, we'll modify our code in `MainViewModel` to use the new API.

We'll replace this line

```
self.networkLayer?.executeNetworkRequest(callback: { data in
```

by this line:

```
self.networkLayer?.executeRequest(BooksRequest(), callback: {
(data, error) in
```

We'll also need to update `NetworkLayerStub` as it's causing a build error as well:

```
class NetworkLayerStub: NetworkLayer {
    var stubbedData:Data?

    init(stubbedData:Data) {
        self.stubbedData = stubbedData
    }

    override fun executeRequest<T>(_ request: T, callback:
@escaping NetworkCompletion) where T : RequestProtocol {
        let jsonData = self.stubbedData!
        callback(jsonData, nil)
    }
}
```

Now we are all done! If we run our whole test suite including our verification tests (Figure 9-8), everything will pass. 🎉



Figure 9-8. Test suite passing

Exercise

We have an extension on UIImageView that we use for downloading images. This extension uses a native API to load the image from a given URL. Your exercise is to change the implementation of this extension to instead use our newly created NetworkLayer.

Summary

Networking is a requirement for almost every app out there. It allows us to take our app to the next level. Being able to request resources from any web service opens the door to countless ways we can improve our apps.

The iOS URL Loading System is made up of a number of classes and structs that are provided natively within iOS's Foundation framework. We use this system to communicate with servers using Internet protocols.

The main class in this system is `URLSession`, which mimics a session in an open tab or window in your web browser. Requests made within the same session share the same configurations and caching. We use an instance of `URLSession` to create instances of `URLSessionTask`. These tasks can fetch data from a server, download/upload files, or open a stream with a server. We use `URLSessionConfiguration` to configure how a session behaves.

When our app is performing network calls, it's extremely important to cover our networking code with tests. Any problem in the network layer can easily cause critical bugs in any app. And it can also cause performance issues, if we're doing unnecessary network calls, for example. Writing tests for a network layer can sometimes be challenging, but it's integral for maintaining the app's quality.

In this chapter we rewrote our network layer by following a test-driven approach. We separated that environment-specific code and covered that with tests. We also made use of protocols to be able to easily create new requests with different endpoints and parameters. This was also covered by tests. Finally, when it came to actually performing the network calls, we were able to cover this as well with tests by the use of test doubles. We injected a mock `URLSession` and used that to be able to assert on the requests going out. Because the part of our app that consumed our network code was already covered by tests, we were able to make this change with confidence. And after we were done, we were able to verify that our changes were functional and didn't break anything.

CHAPTER 10

Taming Core Data

Core Data is one of the most famous frameworks known to iOS developers. Core Data has been available since the release of iOS 3, and it has evolved a lot since then. A common misconception is that Core Data is a database or a database wrapper. Though persisting data is one of its features, Core Data is much more than that. The essence of Core Data is that it manages our application's object graph. An object graph is basically a collection of objects connected with relationships (Figure 10-1). Core Data manages the objects in these graphs, and we can use Core Data to persist the graph on disk if we want. In addition to that, the framework has multiple other features such as data validation and undo/redo management.

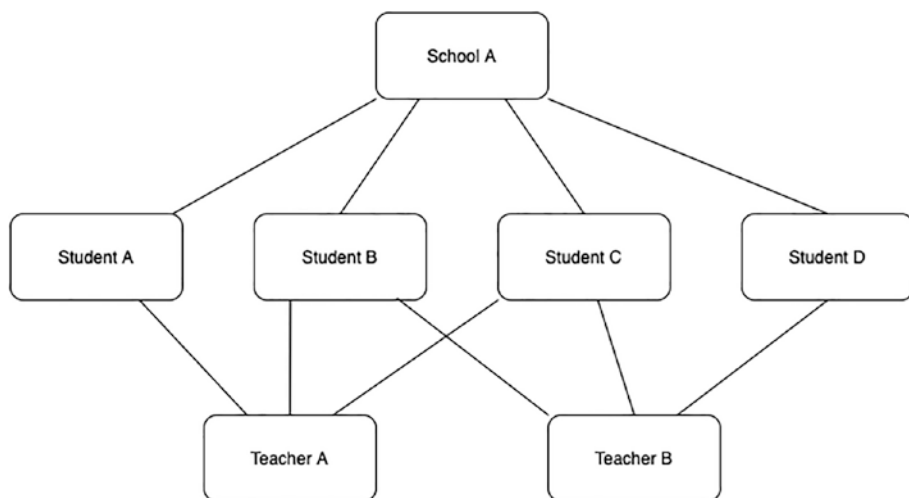


Figure 10-1. Object graph

As famous as Core Data is, many developers suffer when using it. This is largely caused by two things. First, many developers dive headfirst into using Core Data without fully grasping how it operates internally and even externally. Core Data is known to have many building blocks, and not fully understanding what each block is responsible for increases the likelihood of misusing it. Second, and just as important, is the lack of testing. Core Data is one of the most challenging parts in any application to write tests for. Therefore, many developers opt out of covering their Core Data layer with tests. We've talked about the importance of testing over and over again in past chapters, and this importance is even magnified when it comes to Core Data.

The Core Data Stack

Now that we know what Core Data is and what it's capable of, it's time to explore how it functions internally. Core Data has many building blocks that interact together (Figure 10-2). Understanding the function of each building block is crucial in fully grasping this framework and being able to use it properly.

The main building blocks are

1. Managed object model
2. Persistent store coordinator
3. Managed object context

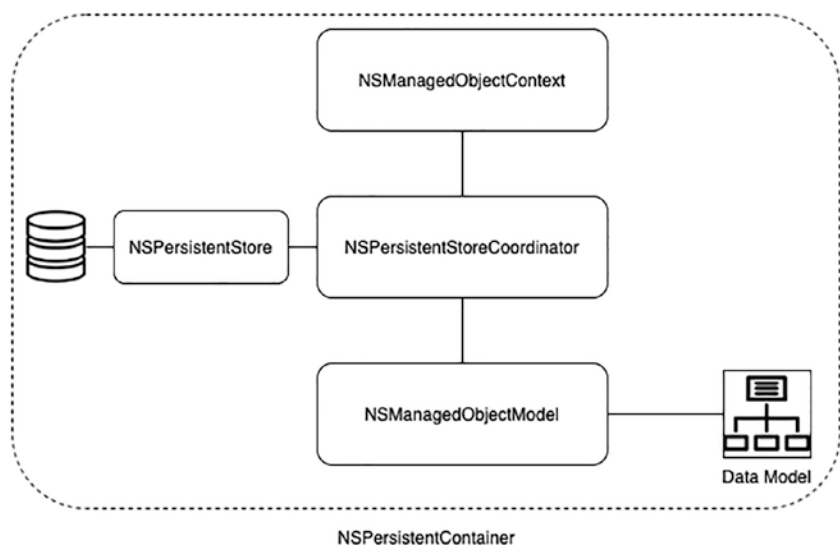


Figure 10-2. The Core Data stack

Managed Object Model

A managed object model is a description of the object graph to be managed. This description is basically the schema for our model. It is the entities with their properties as well as relationships to other entities. A data schema is represented by an `.xcdatamodeld` file, and Xcode comes with a powerful editor that makes it easy to edit our schema file. We can easily create entities, create relationships, version our schema, and prepare migration, all from within Xcode's editor. Core Data does not interact with files; it interacts with instances of `NSManagedObjectContext`. This class provides a programmatic representation of the `.xcdatamodeld` file describing our schema, which Core Data can understand and use. While a typical Core Data implementation has one instance of the `NSManagedObjectContext` class, it's possible to have multiple.

Persistent Store Coordinator

A persistent store coordinator is represented by an instance of the `NSPersistentStoreCoordinator` class, and it plays a key role in the functionality of Core Data. From its name, a persistent store coordinator coordinates between managed object contexts and persistent stores. It takes care of loading, caching, and persisting data. Despite it being one of the most important members of the Core Data stack, you will rarely interact with it directly.

Persistent Store

A persistent store represents where your data actually lives. We've mentioned that Core Data manages an object graph, but in order for the framework to be useful, the persistent store coordinator needs to be connected to at least one persistent store. This allows the coordinator to load data into contexts and push new data into the store making the new change a permanent part of the object graph's state.

Core Data provides four different types of persistent store built in:

1. **SQLite:** This store is backed by a SQLite database, and it's the most widely used store type.
2. **XML:** This store is backed by an XML file.
3. **Binary:** This store is backed by a binary data file.
4. **In-memory:** This store utilizes the app's memory for storage. It's only partially persistent as data is lost when the app is terminated for any reason.

And you can also create your own store types by subclassing `NSAtomicStore` or `NSIncrementalStore`,

Managed Object Context

A managed object context is an object that is responsible for managing a collection of managed objects. It is represented by an instance of the `NSManagedObjectContext` class. A Core Data application can have one or more managed object contexts. Each context is connected to a persistent store coordinator. You can think of a context as a scratch pad where you can make any changes you want to the objects inside the context. You can fetch objects into your context from the persistent store coordinator. You can also insert new objects, make changes to existing objects, or undo/redo changes. Any changes you make to objects inside a context remain local to that context and only saved in-memory, which means that changes are not propagated to the persistent store coordinator. Changes remain local to the context until you manually commit these changes by telling the context to save its changes. You can think of this as if you're writing on a board using a nonpermanent marker, which gives you the ability to clear all your writing at any time. And when you're ready to commit what you wrote, you then **save** your writing by going over them by a permanent marker.

Persistent Container

Before iOS 10, we used to have to manually set up the preceding three components to have a functioning Core Data stack. But from iOS 10, Apple introduced `NSPersistentContainer`, which was a real game changer as it completely simplified the process of setting up Core Data. It is a container that encapsulates the Core Data stack and takes care of the creation and management of the managed object model, persistent store coordinator, and managed object context.

Core Data in Books

If we take a look at **Books**, we will find that it uses Core Data. However, this part of our app is still not modularized. Actually, if we look closely, we'll find that our Core Data code is all over the place. We will attempt to fix this during this chapter. You will find the Books project in this chapter's resources.

Our goal for this following section is to do the following:

- Create a generic Core Data interface.
- Create a component that consumes this interface to provide needed functionality for our app.
- Use this new component instead of the old implementation.

We will go through these incremental steps while following a test-driven approach.

Testing Stack

We want our new Core Data layer to operate using a SQLite persistent store, as it's the store type that makes most sense in our case. This store type persists on disk and at the same time has low performance overhead and low memory footprint.

However, when it comes to testing, this store type causes a few problems. Since data is persisted on disk in a database, this makes data persistent between tests. Persisting data between tests might lead to one test failing due to a change in environment caused by a previous test. This can be fixed by deleting and then recreating the database after each test, but this makes our tests slow, and we need our unit tests to be fast.

You might think that we've hit a dead end here. Well, think again. We've already mentioned that there are other store types. One of them is

the in-memory store type, and it's exactly what we need. This solves our problem because with this store, data isn't persisted to disk; it stays in memory. So with each test, the in-memory store releases its data.

So this means we need to use different stacks for testing and production. We need to use the SQLite store in our production code and in-memory store in our tests. We'll keep that in mind moving forward.

CoreDataManager

Now that we know how Core Data operates, let's kick off the implementation.

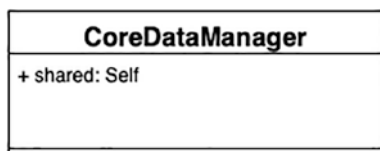


Figure 10-3. *Current UML*

We'll start with our first goal, which is creating a generic interface for Core Data (Figure 10-3). This interface should provide **CRUD** (Create, Read, Update, Delete) operations and should operate on generic models. We know that we'll need to create a new object to be our interface, and this object should be accessible from anywhere in the app. And it makes sense to have only one instance of it. So let's translate this into a test. First, let's add a new test case class and call it `CoreDataManagerTests`. And we'll add this test to it:

```

func testSharedInstance() {
    // When
    let manager = CoreDataManager.shared

    // Then
    XCTAssertNotNil(manager)
}
  
```

Normally there's a build error as we haven't created the class yet. Let's go ahead and fix our test by adding this new class to our app:

```
class CoreDataManager {
    // MARK:- Singleton
    public static let shared = CoreDataManager()
}
```

The test should now be passing.

Make sure to add “@testable import Books” at the beginning of all your test files.

Now on to the next test. We know that CoreDataManager should provide an interface for CRUD operations. So we should now start adding these tests.

All our tests will require initializing an instance of CoreDataManager. This can be added inside the common setup function:

```
// MARK:- Variables
var manager: CoreDataManager!

// MARK:- Setup
override func setUp() {
    super.setUp()
    self.manager = CoreDataManager()
}

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

Now if you remember, we decided on using different stacks for production and testing. This means that we need to create a stack for testing and inject it into our manager. Creating a custom stack for testing should look like this:

```
let stack = CoreDataStack(name: "TestModel", storageType:
.inMemory)
```

CoreDataStack

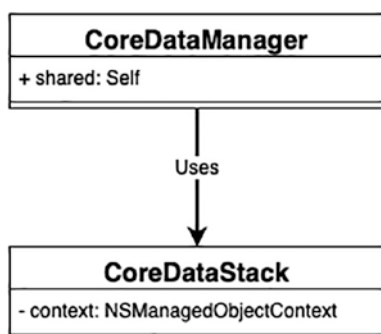


Figure 10-4. Current UML

CoreDataStack is the class responsible for initializing the Core Data stack. Since we haven't created it yet, adding the preceding line will cause a build error. So we'll pause working on CoreDataManagerTests for now and move our focus on creating CoreDataStack (Figure 10-4). Once we're done, we'll circle back to it.

So what exactly is CoreDataStack responsible for? It should initialize a persistent container, and by default the underlying managed model should be the app's model. So let's translate this into a test. First, we'll add a new test case class named CoreDataStackTests, and then add this test inside it:

```

func testDefaultStoreName() {
    // Given
    let stack = CoreDataStack()

    // When
    let container = stack.storeContainer

    // Then
    XCTAssertEqual(container.name, "Books")
}

```

To fix this test, we'll need to create the new class `CoreDataStack` as in the following:

```

import CoreData

class CoreDataStack {
    // MARK:- Lazy Variables
    lazy var storeContainer: NSPersistentContainer = {
        let container = NSPersistentContainer(name: "Books")
        container.loadPersistentStores { _, error in
            if let error = error as NSError? {
                print("Unresolved error \(error), \(error.
                    userInfo)")
            }
        }
        return container
    }()
}

```

Now we need to be able to customize our stack so that it uses custom models, not just the default. We'll heavily depend on this in our tests. So let's add a test for this:

```

func testCustomStoreName() {
    // Given
    let stack = CoreDataStack(name: "TestModel")

    // When
    let container = stack.storeContainer

    // Then
    XCTAssertEqual(container.name, "TestModel")
}

```

To make this pass, we'll need to do two things. First, update our class to handle custom model names:

```

class CoreDataStack {
    // MARK:- Variables
    private let modelName: String

    // MARK:- Lazy Variables
    lazy var storeContainer: NSPersistentContainer = {
        let container = NSPersistentContainer(name: self.
            modelName)
        container.loadPersistentStores { _, error in
            if let error = error as NSError? {
                print("Unresolved error \(error), \(error.
                    userInfo)")
            }
        }
        return container
    }()
}

```



```

// MARK:- Initializer
public init(name: String = "Books") {
    self.modelName = name
}
}

```

Second, we need to add the new data model. To do this, we'll add a new data model file (Figure 10-5) and name it "TestModel."

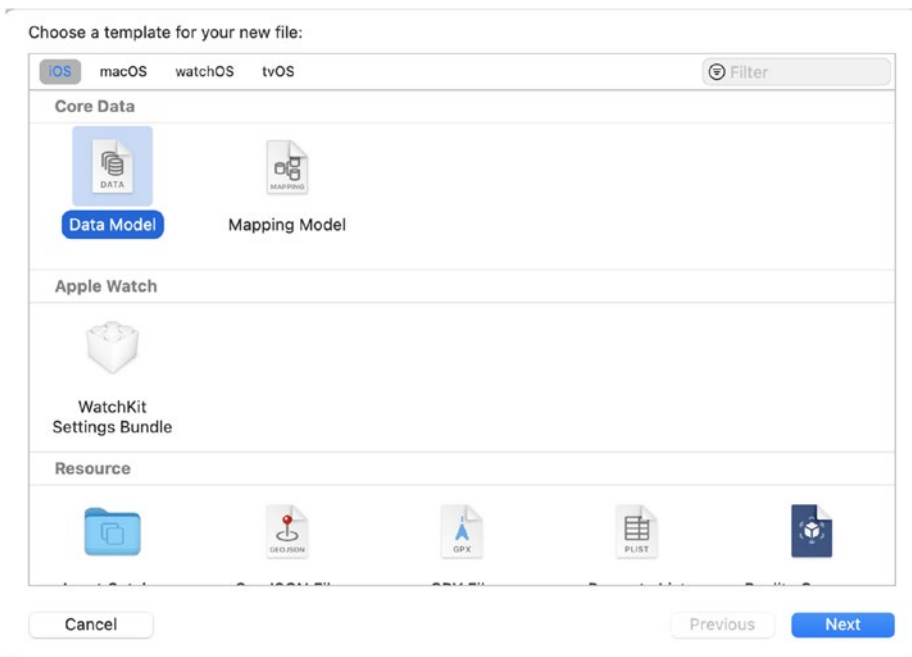


Figure 10-5. Adding a new data model file

After adding the model, head to the project file. Open the test target and under **Build Phases** make sure that the data model file is **NOT** in **Compile Sources** and is included under **Copy Bundle Resources** (Figure 10-6). This will prevent build errors that might happen later on when we create entities.

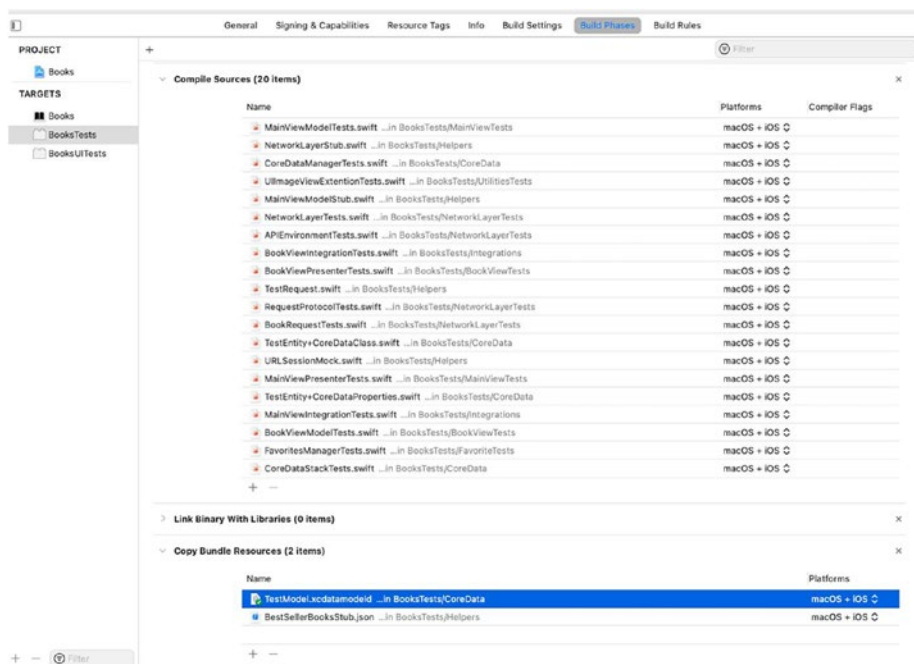


Figure 10-6. Setting up the test data model correctly

Sadly, our test will still be failing after adding the new model. This is because the persistent container searches for the data model file by default inside the app's main bundle. However, our test data model is inside our tests bundle. To do this we need to manually pass the object model for our data model to the Core DataTests stack. Let's update our test to this:

```
func testCustomStoreName() {
    // Given
    let testBundle = Bundle(for: type(of: self))
    let modelUrl = testBundle.url(forResource: "TestModel",
    withExtension: "momd")!
    let objectModel = NSManagedObjectModel(contentsOf:
    modelUrl)
```

```

let stack = CoreDataStack(name: "TestModel", objectModel:
objectModel)

// When
let container = stack.storeContainer

// Then
XCTAssertEqual(container.name, "TestModel")
}

```

And update our class to this:

```

class CoreDataStack {

// MARK:- Variables
private let modelName: String
private let objectModel: NSManagedObjectModel?

// MARK:- Lazy Variables
lazy var storeContainer: NSPersistentContainer = {
    var container: NSPersistentContainer
    if let objectModel = self.objectModel {
        container = NSPersistentContainer(name: self.
modelName, managedObjectModel: objectModel)
    }
    else {
        container = NSPersistentContainer(name: self.
modelName)
    }
    container.loadPersistentStores { _, error in
        if let error = error as NSError? {
            print("Unresolved error \(error), \(error.
userInfo)")
        }
    }
}
}

```

```

        return container
    }()

    // MARK:- Initializer
    public init(name: String = "Books", objectModel:
    NSManagedObjectModel? = nil) {
        self.modelName = name
        self.objectModel = objectModel
    }
}

```

Here we add the ability to inject a custom object model. And when initializing our container, we check if a custom model is passed. If so, we use it to create the container. If not, then we create the container normally.

Now if you recall, we need to be able to create Core Data stacks that utilize in-memory stores. Right now our stack can only be set up using the default store, which is the SQLite store. So let's add tests for this:

```

func testPersistentStoreType() {
    // Given
    let stack = CoreDataStack(storageType: .persistent)

    // When
    let container = stack.storeContainer

    // Then
    XCTAssertEqual(container.persistentStoreDescriptions[0].
    type, NSSQLiteStoreType)
}

func testInMemoryStoreType() {
    // Given
    let stack = CoreDataStack(storageType: .inMemory)

```

```

    // When
    let container = stack.storeContainer

    // Then
    XCTAssertEqual(container.persistentStoreDescriptions[0].
type, NSInMemoryStoreType)
}

```

To fix our tests, we need to add an enum that represents store type. We can add it in a separate file or inside the CoreDataStack file. The enum should look like this:

```

enum StorageType {
    case persistent, inMemory
}

```

Then to fix our tests, we need to change our class to this:

```

class CoreDataStack {

    // MARK:- Variables
    private let modelName: String
    private let objectModel: NSManagedObjectModel?
    private let storageType: StorageType

    // MARK:- Lazy Variables
    lazy var storeContainer: NSPersistentContainer = {
        var container: NSPersistentContainer
        if let objectModel = self.objectModel {
            container = NSPersistentContainer(name: self.
modelName, managedObjectModel: objectModel)
        }
        else {
            container = NSPersistentContainer(name: self.
modelName)
        }
    }
}

```

```

    }
    if self.storageType == .inMemory {
        let description = NSPersistentStoreDescription()
        description.type = NSInMemoryStoreType
        container.persistentStoreDescriptions =
            [description]
    }
    container.loadPersistentStores { _, error in
        if let error = error as NSError? {
            print("Unresolved error \(error), \(error.
                userInfo)")
        }
    }
    return container
}()

// MARK:- Initializer
public init(name: String = "Books", objectModel:
    NSManagedObjectModel? = nil, storageType: StorageType =
    .persistent) {
    self.modelName = name
    self.objectModel = objectModel
    self.storageType = storageType
}
}

```

Here we add a new variable to hold our storage type. And we add a new parameter in the `init` to be able to set the storage type. We also set the default value to `.persistent`. And finally we check if the storage type is in-memory; if so, we override the store type. Else, we leave the default store type, which is SQLite. After these changes, our tests should be passing now.

Lastly, we need our stack to provide us with a context. This context should be used on the main thread. In our app, all of our usage of Core Data is lightweight and will reflect on our app's UI. Which means we don't need background contexts.

Let's add a test for this:

```
func testContext() {
    // Given
    let stack = CoreDataStack(storageType: .inMemory)

    // When
    let context = stack.context

    // Then
    XCTAssertNotNil(context)
    XCTAssertEqual(context.concurrencyType,
.mainQueueConcurrencyType)
}
```

To fix the test, we need to add the following inside CoreDataStack:

```
public lazy var context: NSManagedObjectContext = {
    return storeContainer.viewContext
}()
```

Inject the Stack into CoreDataManager

Now that we have our CoreDataStack ready, let's go back to CoreDataManagerTests, which triggered all this. Now we can create a custom stack and pass it to the manager:

```
// MARK:- Variables
var manager: CoreDataManager!
var stack: CoreDataStack!
```

```

// MARK:- Setup
override func setUp() {
    super.setUp()
    let testBundle = Bundle(for: type(of: self))
    let modelUrl = testBundle.url(forResource: "TestModel",
    withExtension: "momd")!
    let objectModel = NSManagedObjectModel(contentsOf:
    modelUrl)
    self.stack = CoreDataStack(name: "TestModel", objectModel:
    objectModel, storageType: .inMemory)
    self.manager = CoreDataManager(coreDataStack: stack)
}

override func tearDown() {
    super.tearDown()
    self.manager = nil
    self.stack = nil
}

```

This will cause a build error. To fix this, we need to update `CoreDataManager` to accept a `CoreDataStack` as a dependency:

```

// MARK:- Variables
private var stack: CoreDataStack

// MARK:- Singleton
public static let shared = CoreDataManager(coreDataStack:
CoreDataStack())

// MARK:- Initializer
public init(coreDataStack: CoreDataStack) {
    self.stack = coreDataStack
}

```


TestEntity

Before we start writing tests for the CRUD operations, it makes sense to create an entity to be able to perform operations on. We'll head to `TestModel.xcdatamodel` and add a new entity from Xcode's editor and call it `TestEntity`. And we'll add the two attributes in Figure 10-7.

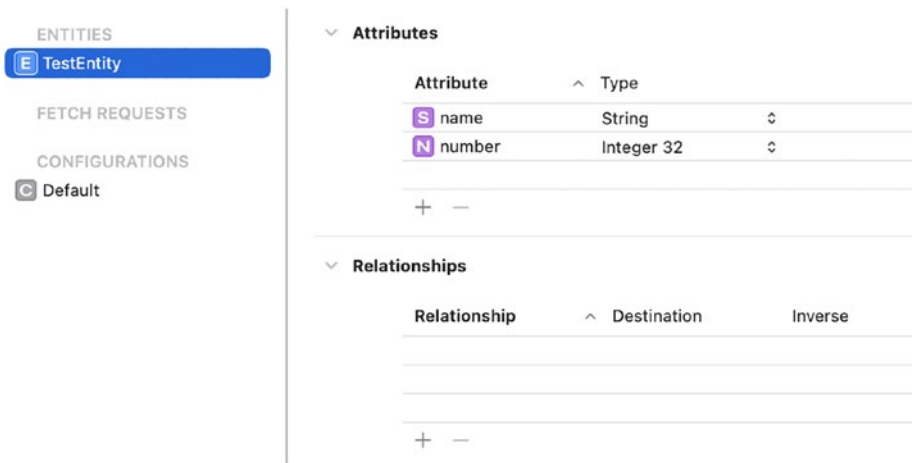


Figure 10-7. Adding *TestEntity* to a data model file

Now to finish off the setup of our new entity, we need to add code representation for it. We'll follow the conventions here and add two files (both in the test target). First, the file `TestEntity+CoreDataClass` should contain this:

```
import CoreData

@objc(TestEntity)
public final class TestEntity: NSObject {
}
```

And the second file `TestEntity+CoreDataProperties` should contain this:

```
import CoreData

extension TestEntity {

    @nonobjc public class func fetchRequest() ->
        NSFetchedRequest<TestEntity> {
        return NSFetchedRequest<TestEntity>(entityName:
            String(describing: TestEntity.self))
        }

    @NSManaged public var name: String?
    @NSManaged public var number: Int32
}
```

Creation

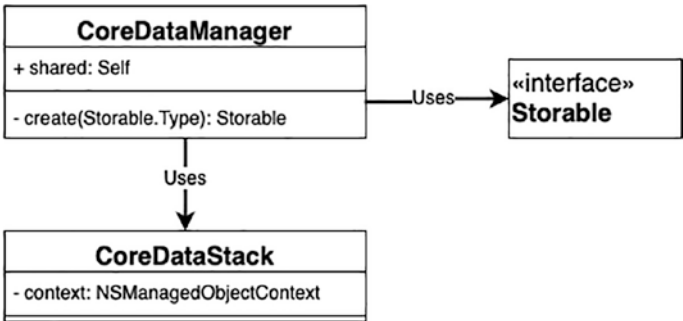


Figure 10-8. Current UML

Now let's start with the first CRUD operation, which is create, and let's write a test for it:

```
func testCreateEntity() {
    // When
    let testModel = manager.create(TestEntity.self)

    // Then
    XCTAssertNotNil(testModel)
    XCTAssertEqual(stack.context.insertedObjects.count, 0)
}
```

Here we create a new object and assert that it's not nil and that it actually gets saved in the persistent store, not just in the context.

This will result in a build error, because there is no create function. So let's add it:

```
public func create<T: Storable>(_ entity: T.Type) -> T? {
    return nil
}
```

Introducing Storable

Storable is a protocol that describes a class that can be stored using our Core Data manager (Figure 10-8). And any Storable needs to be an `NSManagedObject`. Let's add this protocol:

```
import CoreData

public protocol Storable: NSManagedObject {
}
```

Now the test is still not building because `TestEntity` does not conform to `Storable`.

We fix this by simply conforming to the protocol like so:

```
extension TestEntity: Storable {
}
```

Now that our test is building, if we try to run it, it will fail.

Creation Implementation

Let's fix this by actually creating a new entity:

```
public func create<T: Storable>(_ entityType: T.Type) -> T? {
    guard let entityDescription = NSEntityDescription.
        entity(forEntityName: entityType.entityName, in: stack.
            context) else {
        return nil
    }
    let entity = NSManagedObject(entity: entityDescription,
        insertInto: stack.context)
    return entity as? T
}
```

We'll need to add this to Storable:

```
public protocol Storable: NSManagedObject {
    static var entityName: String {get}
}
```

And update TestEntity's implementation to this:

```
extension TestEntity: Storable {
    public static var entityName: String {
        String(describing: Self.self)
    }
}
```

If we run our test, we'll find that the second assertion is still failing. This means we need to save our changes.

Saving Changes

Inside the create function, we'll add a line to save our context right before we return. Our function should now look like this:

```
public func create<T: Storable>(_ entityType: T.Type) -> T? {
    guard let entityDescription = NSEntityDescription.
        entity(forEntityName: entityType.entityName, in: stack.
            context) else {
        return nil
    }
    let entity = NSManagedObject(entity: entityDescription,
        insertInto: stack.context)
    stack.saveContextIfNeeded()
    return entity as? T
}
```

This will lead to a build error. To fix it, we need to add a new function to CoreDataStack (Figure 10-9):

```
public func saveContextIfNeeded() {
}
```

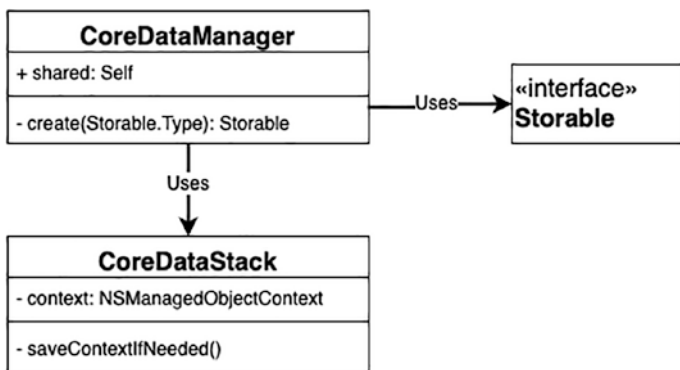


Figure 10-9. Current UML

Now let's add a test for saving a context inside `CoreDataStackTests`:

```
func testSavingContextIfNeeded() {
    // Given
    let testBundle = Bundle(for: type(of: self))
    let modelUrl = testBundle.url(forResource: "TestModel",
    withExtension: "momd")!
    let objectModel = NSManagedObjectModel(contentsOf:
    modelUrl)
    let stack = CoreDataStack(name: "TestModel", objectModel:
    objectModel, storageType: .inMemory)
    let context = stack.context
    let _ = TestEntity(context: context)

    // Expected
    expectation(forNotification:
    .NSManagedObjectContextDidSave, object: context,
    handler: nil)

    // When
    stack.saveContextIfNeeded()

    // Then
    waitForExpectations(timeout: 1.0, handler: nil)
}
```

To fix our test, we'll update `saveContextIfNeeded` to this:

```
public func saveContextIfNeeded() {
    if context.hasChanges {
        do {
            try context.save()
        }
    }
}
```

```

    catch let error as NSError {
        print("Unresolved error \(error), \(error.
            userInfo)")
    }
}
}
}

```

Now all our tests are passing, `testSavingContextIfNeeded` and `testCreateEntity`. This means we're done with our first CRUD operation.

Fetching

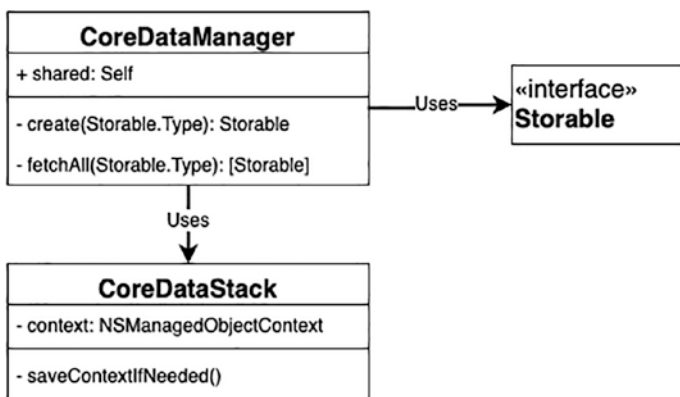


Figure 10-10. Current UML

Now let's move on to the second CRUD operation, which is fetching data (Figure 10-10).

We'll start by adding a test that creates a new entity and then fetches all entities and checks that the returned value is correct. This test will look like this:

```

func testFetchEntities() {
    // Given
    let testModel = manager.create(TestEntity.self)
}

```

```

// When
let models = manager.fetchAll(TestEntity.self)

// Then
XCTAssertNotNil(models)
XCTAssertEqual(models?.count, 1)
XCTAssertEqual(models?[0].objectID, testModel?.objectID)
}

```

To fix this test, we'll go and implement the fetch function.

Add this func:

```

public func fetchAll<T: Storable>(_ entityType: T.Type) -> [T]?
{
    let request: NSFetchRequest<T> = T.fetchRequest()
    do {
        let results = try stack.context.fetch(request)
        return results
    } catch let error as NSError {
        print("Unresolved error \(error), \(error.userInfo)")
    }
    return nil
}

```

We'll need to update `Storable` because we require each `Storable` to provide its own fetch request, and we need this fetch request to be able to fetch our objects. It should now look like this:

```

public protocol Storable: NSManagedObject {
    static var entityName: String {get}
    static func fetchRequest() -> NSFetchRequest<Self>
}

```


Updating

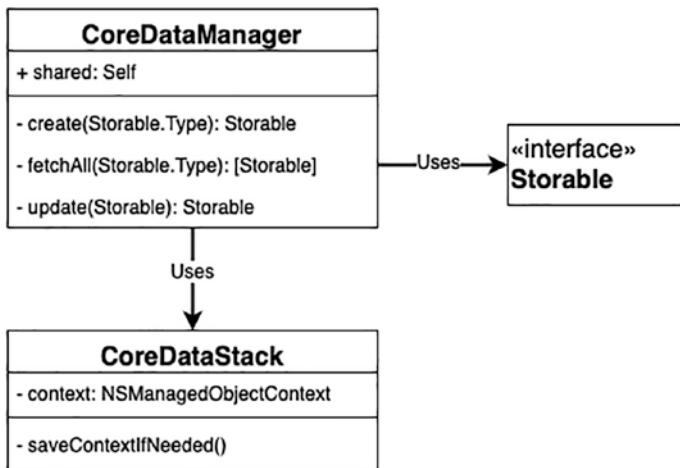


Figure 10-11. Current UML

Now let's move to a new CRUD operation (Figure 10-11). Let's add a test for updating:

```

func testUpdateEntity() {
    // Given
    let testModel = manager.create(TestEntity.self)
    testModel?.name = "Test"
    testModel?.number = 123

    // When
    manager.update(testModel)
    stack.context.rollback()

    // Then
    let updatedModel = manager.fetchAll(TestEntity.self)?[0]
    XCTAssertNotNil(updatedModel)
    XCTAssertEqual(updatedModel?.name, "Test")
    XCTAssertEqual(updatedModel?.number, 123)
}
  
```

You might have noticed that we call `rollback()` on our context inside our test. So what does this do? First, let's look at what we're attempting to do in our test. We insert a new object using `create`, and then we make some changes to it. We call our update function, and we're expecting it to persist these changes. Given the nature of Core Data, we know that the changes we make will be applied only locally to the current context we're in. And since we use the same context for creating and updating as we do for fetching, then even if we don't persist our changes in the store, the fetch will return the updated data. To showcase this, let's add implementation for `update` that doesn't actually save the changes:

```
@discardableResult
public func update<T: Storable>(_ entity: T?) -> T? {
    return entity
}
```

The test now fails because we do not save. To showcase the importance of `rollback()`, comment out the line where we call `rollback()` and rerun the test. We'll find that our test passes. So in this test the use of `rollback` is essential to clear all unsaved data and assert only on the saved data.

To fix the test, we need to change our implementation so that we actually save the changes:

```
@discardableResult
public func update<T: Storable>(_ entity: T?) -> T? {
    stack.saveContextIfNeeded()
    return entity
}
```

Advanced Fetching

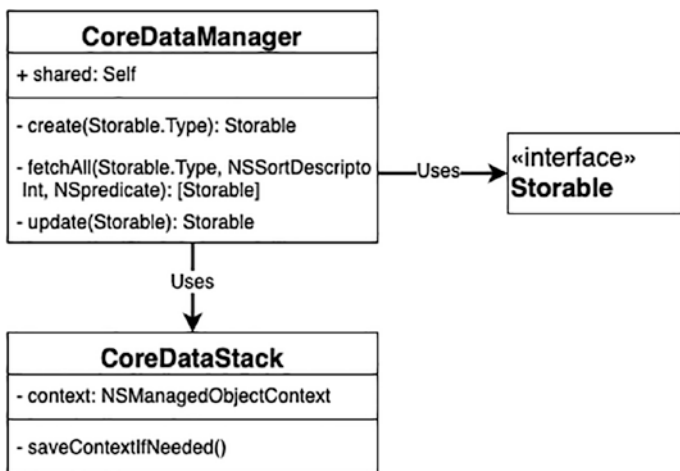


Figure 10-12. Current UML

Since tests are now passing, let's add a new functionality. We need to be able to sort our fetched results (Figure 10-12). Sorting them while fetching is much more optimized than sorting them in memory after fetching. In addition to sorting, we need to be able to filter our fetch results. We also need to set a limit for fetch results. Let's add tests for these two functionalities:

```

func testFetchSorted() {
  // Given
  for i in 1...10 {
    let testModelOne = manager.create(TestEntity.self)
    testModelOne?.number = Int32(i)
    manager.update(testModelOne)
  }

  // When
  let sort = NSSortDescriptor(key: "number", ascending:
false)
  let models = manager.fetchAll(TestEntity.self, sort: sort)
}
  
```

```

// Then
XCTAssertNotNil(models)
XCTAssertEqual(models?.count, 10)
XCTAssertEqual(models?[0].number, 10)
XCTAssertEqual(models?[9].number, 1)
}

func testFetchWithLimit() {
    // Given
    for i in 1...10 {
        let testModelOne = manager.create(TestEntity.self)
        testModelOne?.number = Int32(i)
        manager.update(testModelOne)
    }

    // When
    let models = manager.fetchAll(TestEntity.self, limit: 5)

    // Then
    XCTAssertNotNil(models)
    XCTAssertEqual(models?.count, 5)
}

func testFetchWithPredicate() {
    // Given
    for i in 1...10 {
        let testModelOne = manager.create(TestEntity.self)
        testModelOne?.number = Int32(i)
        manager.update(testModelOne)
    }

    // When
    let predicate = NSPredicate(format: "number > 5")
    let models = manager.fetchAll(TestEntity.self, predicate:
predicate)

```

```

// Then
XCTAssertNotNil(models)
XCTAssertEqual(models?.count, 5)
}

```

To fix these tests, update `fetchAll` to this:

```

public func fetchAll<T: Storable>(_ entityType: T.Type,
sort: NSSortDescriptor? = nil, limit: Int = 0, predicate:
NSPredicate? = nil) -> [T]? {
    let request: NSFetchedRequest<T> = T.fetchRequest()
    if let sort = sort {
        request.sortDescriptors = [sort]
    }
    if let predicate = predicate {
        request.predicate = predicate
    }
    request.fetchLimit = limit
    do {
        let results = try stack.context.fetch(request)
        return results
    } catch let error as NSError {
        print("Unresolved error \(error), \(error.userInfo)")
    }
    return nil
}

```

Next Steps

Let's recall our goal we stated earlier. It was divided into three subgoals:

- Create a generic Core Data interface ✓.
- Create a component that consumes this interface to provide needed functionality for our app.
- Use this new component instead of the old implementation.

Since we're done with the creation of the generic Core Data interface (CoreDataManager), let's move now to our second subgoal. We will create a new component called FavoritesManager, which will be responsible for adding, deleting, and fetching our favorites (Figure 10-13).

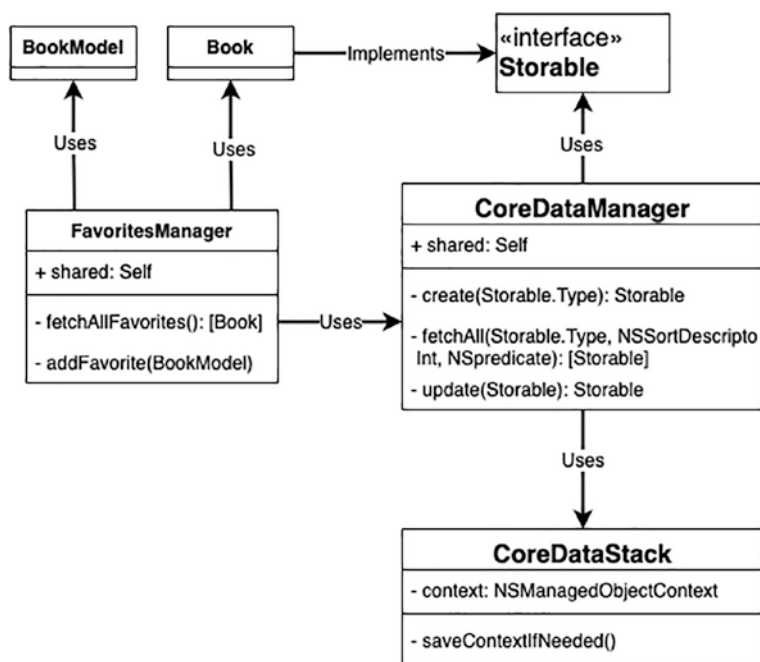


Figure 10-13. Current UML

We'll start as usual with tests. Let's add a new test case class named `FavoritesManagerTests`, which will house our tests. Next, let's set up our manager. We need to set it up using a custom in-memory Core Data manager that consumes our app's data model. Our file should look like this:

```
import XCTest
@testable import Books
import CoreData

class FavoritesManagerTests: XCTestCase {

    // MARK:- Variables
    var favoritesManager: FavoritesManager!
    var coredataManager: CoreDataManager!
    var stack: CoreDataStack!

    // MARK:- Setup
    override func setUp() {
        super.setUp()
        self.stack = CoreDataStack(storageType: .inMemory)
        self.coredataManager = CoreDataManager(coreDataStack:
        stack)
        self.favoritesManager = FavoritesManager(coredataManage
        r: coredataManager)
    }

    override func tearDown() {
        super.tearDown()
        self.favoritesManager = nil
        self.coredataManager = nil
        self.stack = nil
    }
}
```

This will cause build errors. To fix these, we'll need to add a new class `FavoritesManager` that has an internal dependency on `CoreDataManager`. Our class should look like this:

```
class FavoritesManager {
    // MARK:- Variables
    private var CoreDataManager: CoreDataManager

    // MARK:- Singleton
    public static let shared = FavoritesManager()

    // MARK:- Initializer
    init(CoreDataManager: CoreDataManager = .shared) {
        self.CoreDataManager = CoreDataManager
    }
}
```

Now we know from before that interacting with `CoreDataManager` requires our models to conform to the `Storable` protocol. Let's go ahead and get this out of the way. We'll do the same thing we did with `TestEntity` for our two managed object classes: `Book` and `BuyLink`. We need to also make sure that both classes are marked with the `final` keyword to avoid build errors.

Now let's write tests for the operations `FavoritesManager` is responsible for. Normally we would tackle this in a normal TDD fashion. However, we won't go through this part step by step to avoid repetitiveness. After going through multiple TDD cycles, we will end up with this set of new tests:

```
func testAddingBook() {
    // Given
    let buyLink = BuyLinkModel(name: .amazon, url: "URL")
```



```

var book = BookModel(title: "BookTitle", contributor:
"Contributor", author: "Author", createdAt: "2021-05-26
22:10:24")
book.amazonProductURL = "Amazon"
book.bookImage = "Image"
book.bookDescription = "Description"
book.publisher = "Publisher"
book.buyLinks = [buyLink]

// When
favoritesManager.addFavorite(book)

// Then
let books = CoreDataManager.fetchAll(Book.self)
XCTAssertNotNil(books)
XCTAssertEqual(books?.count, 1)
let retrievedBook = books![0]
XCTAssertEqual(retrievedBook.title, book.title)
XCTAssertEqual(retrievedBook.contributor, book.contributor)
XCTAssertEqual(retrievedBook.author, book.author)
XCTAssertEqual(retrievedBook.created_date, book.
createdAt)
XCTAssertEqual(retrievedBook.amazon_product_url, book.
amazonProductURL)
XCTAssertEqual(retrievedBook.book_image, book.bookImage)
XCTAssertEqual(retrievedBook.desc, book.bookDescription)
XCTAssertEqual(retrievedBook.publisher, book.publisher)
XCTAssertEqual(retrievedBook.buyLinks?.count, 1)
let link = retrievedBook.buyLinks?.allObjects[0] as?
BuyLink
XCTAssertEqual(link?.name, buyLink.name.rawValue)
XCTAssertEqual(link?.url, buyLink.url)
}

```

```

func testFetchingFavoritesSorted() {
    // Given
    let book1 = BookModel(title: "Book1", contributor:
    "Contributor", author: "Author", createdAt: "2021-05-01
    22:00:00")
    let book2 = BookModel(title: "Book2", contributor:
    "Contributor", author: "Author", createdAt: "2021-05-02
    22:00:00")
    let book3 = BookModel(title: "Book3", contributor:
    "Contributor", author: "Author", createdAt: "2021-05-03
    22:00:00")

    favoritesManager.addFavorite(book1)
    favoritesManager.addFavorite(book3)
    favoritesManager.addFavorite(book2)

    // When
    let favorites = favoritesManager.fetchAllFavorites()

    // Then
    XCTAssertEqual(favorites.count, 3)
    XCTAssertEqual(favorites[0].title, "Book3")
    XCTAssertEqual(favorites[1].title, "Book2")
    XCTAssertEqual(favorites[2].title, "Book1")
}

```

And the code that makes these tests pass is this:

```

// MARK:- Public Functions
func fetchAllFavorites() -> [Book] {
    let sort = NSSortDescriptor(key: "created_date", ascending:
    false)
    return CoreDataManager.fetchAll(Book.self, sort: sort) ?? []
}

```

```

func addFavorite(_ model: BookModel) {
    guard let book = CoreDataManager.create(Book.self) else {
        return
    }
    book.title = model.title
    book.amazon_product_url = model.amazonProductURL
    book.author = model.author
    book.book_image = model.bookImage
    book.contributor = model.contributor
    book.created_date = model.createdDate
    book.desc = model.bookDescription
    book.publisher = model.publisher

    let links:NSMutableSet? = []
    guard let buyLinks = model.buyLinks else {
        return
    }

    for buyLink in buyLinks {
        if let link = CoreDataManager.create(BuyLink.self) {
            link.url = buyLink.url
            link.name = buyLink.name.rawValue
            link.book = book
            links?.add(link)
        }
    }

    book.buyLinks = links
    CoreDataManager.update(book)
}

// MARK:- Private Helpers
func getBook(from model: BookModel) -> Book? {

```

```

let predicate = NSPredicate(format: "title == %@", model.
title ?? "")
let results = coreDataManager.fetchAll(Book.self, limit: 1,
predicate: predicate)
guard let books = results, books.count == 1 else {
    return nil
}
return books[0]
}

```

- `fetchAllFavorites` uses the `fetchAll` function of `CoreDataManager` to fetch all objects of type `Book` and returns them sorted by date.
- `addFavorite` takes a `BookModel` object. It inserts a new `Book` object into our store and then populates this book with data from the passed model.

Putting It All Together

So far we have not changed any of our old code. We've only added new code, but we haven't used it anywhere in our app yet. Which brings us to the last goal we had when we started this Core Data-themed journey: we want to use the new code we've written instead of the old implementation.

Making this change will directly affect our app. So as with any step we take in TDD, we need to start it with tests. We need to make sure that we have verification tests in place covering all the logic that will be affected. The logic that's going to be affected is everything related to favorites handling in our app. Luckily, if we take a look at our UI test suite, we'll find that we have tests for all our scenarios. This means we can now switch out the implementations with confidence.

Easiest way to guide this change is to remove the old Core Data code from its root. So let's head to our AppDelegate and remove all code related to Core Data. This will result in an array of build errors in our app. Now we just go over the errors one by one and replace the old code by calls to FavoritesManager.

In FavViewController the function loadSavedData will now look like this:

```
func loadSavedData() {
    let results = FavoritesManager.shared.fetchAllFavorites()
    for book in results {
        books.append(convertToBookModel(book: book))
    }
    self.tableView?.reloadData()
}
```

And the saveBookAsFavorite implementation in both BookViewControllerA and BookViewControllerB will look like this:

```
func saveBookAsFavorite(withBook bookModel:BookModel) {
    FavoritesManager.shared.addFavorite(bookModel)
    let alert = UIAlertController(title: "Saved", message:
    "Your book saved to favorites", preferredStyle: .alert)
    alert.addAction(UIAlertAction(title: "Ok", style: .default,
    handler: nil))
    self.present(alert, animated: true, completion: nil)
}
```

Now that we're done with our changes, we need to rerun our verification tests to make sure our changes did not break anything. When we run our tests, everything passes, which means we've successfully written a testable Core Data layer and integrated it swiftly in our app!

Exercise

We have one last operation to add, which is the deletion operation. Your exercise is to add a new delete API to `CoreDataManager`. And use that new API to implement deleting a book inside `FavoritesManager`. Then you'll update `FavViewController` to use the new API in `FavoritesManager`.

After adding the delete API, the final design should look like Figure 10-14.

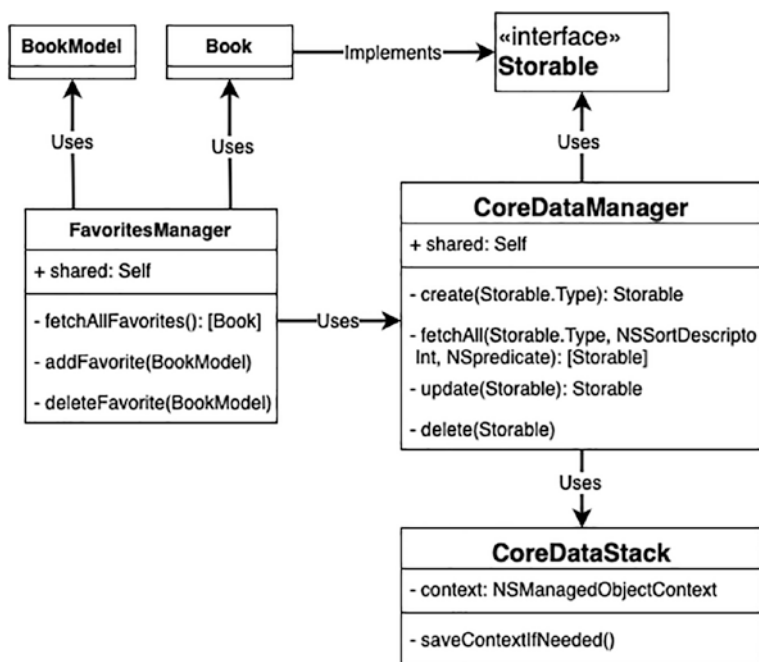


Figure 10-14. Final UML

Summary

Core Data is indeed a powerful framework, and it's used by many developers due to its vast array of features. But as mentioned before, using Core Data can sometimes be a tedious task. This is largely due to two

things. First is that many developers use Core Data without fully grasping what this framework really is and how it functions. Second is that many developers find it challenging to write tests for their Core Data code, which ultimately leads to hard-to-spot regressions. In this chapter, we've attempted to address these two issues.

We talked about what Core Data really is and what it's not. Core Data is not a database. Though it's capable of persisting data on disk, Core Data is much more than that. In its essence, Core Data manages object graphs, meaning it manages the lifecycle of our objects. Internally Core Data depends on multiple objects to function, each having a specific responsibility. There is the managed object model, which is a programmatic representation of our object schema. And there's the managed object context, which acts as a scratch pad for us to apply the changes we want, and then we can either discard them or persist them. Finally, there's the persistent store coordinator, which acts as a middleman between our contexts and the persistent store, which is responsible for actually saving the data. These represent the main building blocks of the Core Data stack. And then finally there's the persistent container, which encapsulates the Core Data stack and simplifies its creation and management.

We then went on and debunked the myth that Core Data is not testable. Yes, testing Core Data can be challenging, but once you get the hang of it, it becomes a piece of cake. We created a generic Core Data layer and used it in our app (Books) instead of the old Core Data implementation. And we did all that using TDD. We saw how using a SQLite store in testing can cause issues in our tests. And we were able to overcome this by using an in-memory store. We also saw how we can write tests that are completely isolated from our app's data model by adding a separate data model just for testing and initializing our Core Data stack in tests using this new model.

CHAPTER 11

Adding Features to a Legacy App

If you recall in Chapter 1, we talked about the various situations where we can use TDD. TDD can basically be applied at any stage in a project's lifetime. The most obvious option is start using TDD from the very beginning. This is what we always recommend. However, what if you only just recently heard about TDD and you already have a project you're working on? Well, TDD is still for you. TDD can help guide the refactor of old legacy code, and also we can use TDD to properly modularize and decouple our code base. We've already put this to action extensively in previous chapters.

We can also use TDD when adding new functionalities to our existing legacy code. This is what we will discuss in this chapter. Let's assume we have a legacy app that is on its way to being fully refactored and modularized using TDD. But in the middle of that process, we got a request for a new feature. Given the fast-paced world we're living in, in most cases we won't have the luxury of pausing all new advancements of a project until a big refactor is done. We need to continue developing new features while simultaneously enhancing/refactoring our legacy code.

So how can we do this? First, we'll examine our feature to determine if the feature is coupled with an old feature or a completely new feature that doesn't depend on any old code. And if the feature is coupled with old code, is this code legacy or refactored and modularized? But don't get this

wrong. We're not doing this examination to determine if we need TDD or not. We'll be using TDD in all cases. This will only affect the complexity of our task. Working on a completely separated feature will be relatively simple, since we are almost writing code from scratch, so we will spend less time refactoring code.

On the other hand, working on a feature coupled with old legacy code is a bit challenging because we can't first refactor our code and then add the feature. In these scenarios, you might be tempted to ditch TDD and testing altogether because the parent code is not tested. However, there's a rule of thumb we should always try to follow from Uncle Bob's clean code, which states "Always leave the code better than you found it." That's why we should try not to give in to that temptation.

Legacy Code Disclaimer

We will be adding a new feature to **Books**. Let's open up the starter project, which can be found in the chapter's resources. If we take a look at our code, we'll find that we have two view controllers for the detailed book view: `BookViewControllerA` and `BookViewControllerB`. This exists because we're running an A/B testing experiment. Even though the two view controllers have a lot of common functionalities, the code is duplicated between them, which is something expected from legacy code. This is a huge code smell, and if we have time we should definitely fix this, but sadly we don't.

A/B Testing

A/B testing is essentially an experiment where two or more variants of a page are shown to users at random, and statistical analysis is used to determine which variation performs better for a given conversion goal. So the motivation of having an A/B test here was to determine which design for `BookViewController` gives a better conversion for purchasing books.

New Feature

The feature we want to add is to display reviews for each book, which can help our users in making a decision whether they want to read this book or not (Figure 11-1).

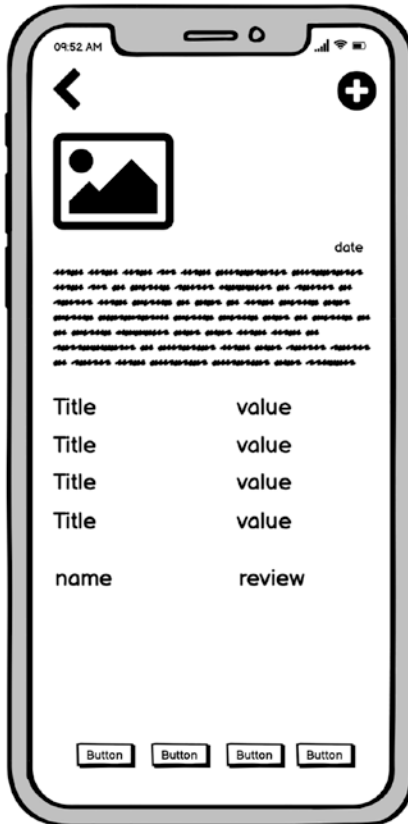


Figure 11-1. *Reviews wireframe*

As we already pointed out, there are two view controllers for the book view: `BookViewControllerA` and `BookViewControllerB`. And we need to make this change in both view controllers. Knowing that the

change is common between both controllers, we can follow the current implementation and add the change in both controllers. However, this way we won't be following Uncle Bob's rule we mentioned earlier. Let's start implementing the feature and see how we can address this problem without having to refactor the entirety of the code.

Kickoff

We'll start by listing the possible scenarios a user can go through:

1. When a user opens a book view that contains no review, they should be able to see an indication that there are no reviews.
2. When a user opens a book view that contains reviews, they should be able to see the first review.

We will follow our approach in implementing this feature as you can see in Figure 11-2.

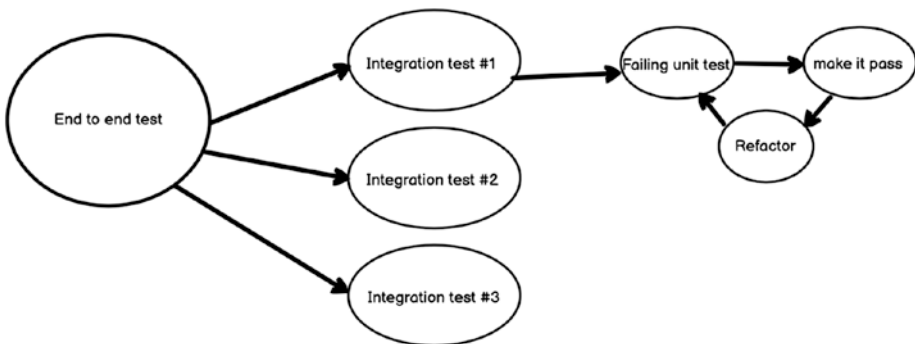


Figure 11-2. Testing plan diagram

Now let's transform these scenarios into UI tests. These tests are our end goal. Once these tests pass, we then know that we're done with our new feature ✓.

UI Tests

Let's transform our first scenario to a test. We'll open up `BooksUITests` and add a new test called `testShowingBookViewWithNoReviews`. Let's take a look at the "Given" section of our test:

```
// Given
let testBundle = Bundle(for: type(of: self))
let booksJSONURL = testBundle.url(forResource:
"BestSellerBooksStub", withExtension: "json")
let booksJSON = try! String(contentsOf: booksJSONURL!)
let booksNoReviewsJSONURL = testBundle.url(forResource:
"booksNoReview", withExtension: "json")
let booksNoReviewsJSON = try! String(contentsOf:
booksNoReviewsJSONURL!)
server.GET[ "/svc/books/v3/lists/overview.json" ] = { _ in
  HttpResponse.ok(.text(booksJSON))
}
server.GET[ "/svc/books/v3/reviews.json?title=THE+LAST+THING+HE+
TOLD+ME" ] = { _ in HttpResponse.ok(.text(booksNoReviewsJSON)) }

let app = XCUIApplication()
app.launchArguments += ["TESTING"]
app.launch()
```

This is almost identical to how we set up the already existing tests. The only difference is that we now need to stub one more request, which is the reviews request. Here we stub it and return a response having no reviews.

Now on to the "When" section:

```
// When
let booksTableView = app.tables
let cells = booksTableView.cells
let firstCell = cells.firstMatch
_ = firstCell.waitForExistence(timeout: 1.0)
firstCell.tap()
```

Here we just tap on a book to go to the book details view.

And now for the “Then” section:

```
let reviewsCell = cells.staticTexts["book_review"]
_ = reviewsCell.waitForExistence(timeout: 1.0)
XCAssertTrue(cells.staticTexts["book_review"].label == "No
Reviews Available")
```

Here we make sure that the text “No Reviews Available” is shown.

Now that we’ve added a UI test for our first scenario, let’s add a test for the second scenario:

```
func testShowingBookViewWithReveiw () {
    // Given
    let testBundle = Bundle(for: type(of: self))
    let booksJSONURL = testBundle.url(forResource:
        "BestSellerBooksStub", withExtension: "json")
    let booksJSON = try! String(contentsOf: booksJSONURL!)
    let booksReveiwJSONURL = testBundle.url(forResource:
        "booksReview", withExtension: "json")
    let booksReveiwJSON = try! String(contentsOf:
        booksReveiwJSONURL!)
    server.GET["/svc/books/v3/lists/overview.json"] = {_ in
        HttpResponse.ok(.text(booksJSON))}
    server.GET["/svc/books/v3/reviews.json?title=THE+
        LAST+THING+HE+TOLD+ME"] = {_ in HttpResponse.ok(.
        text(booksReveiwJSON))}

    let app = XCUIApplication()
    app.launchArguments += ["TESTING"]
    app.launch()
}
```

```

    // When
    let booksTableView = app.tables
    let cells = booksTableView.cells
    let firstCell = cells.firstMatch
    _ = firstCell.waitForExistence(timeout: 1.0)
    firstCell.tap()

    // Then
    let reviewsCell = cells.staticTexts["book_review"]
    _ = reviewsCell.waitForExistence(timeout: 1.0)
    XCTAssertTrue(cells.staticTexts["book_review"].label ==
    "The book is interesting")
}

```

This is almost identical to the first test we've added. The only changes are that we stub the request using a different response. And we check that the review returned in the response is displayed.

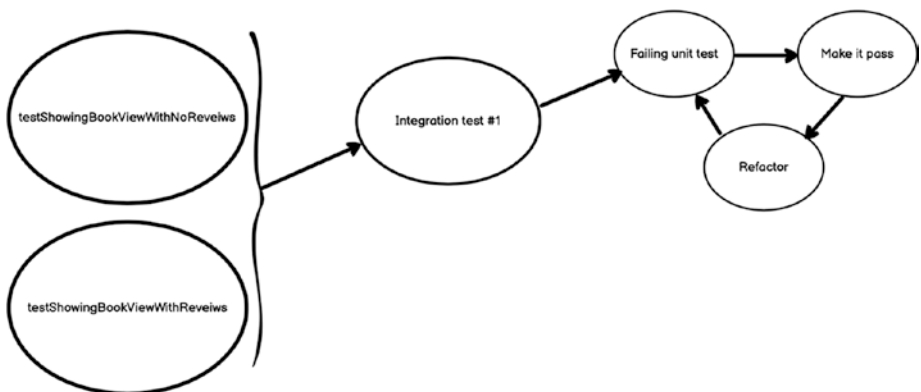


Figure 11-3. Testing plan diagram (end-to-end tests added)

Now that we're done with our UI tests (Figure 11-3), let's go down a level.

Integration Tests

Now, we can design how the feature will work using integration tests. We will use MVP again as we did before in all previous chapters. However, `BookViewControllerA` and `BookViewControllerB` contain a lot of spaghetti code. They both make a network request and save data into the database. They also share a lot of duplicated code between them. Unfortunately as we said before, we don't have time to refactor this whole mess. So we need to design this feature to be added into our spaghetti code so that the added code is loosely coupled with each other and well tested and enhances the already implemented code without refactoring the whole class.

We will implement/inject this feature inside `BookViewControllerA` and `BookViewControllerB` as if these viewControllers don't do anything else. So this feature will be implemented using the MVP design pattern (Figure 11-4), and the old feature will remain the same with no change.

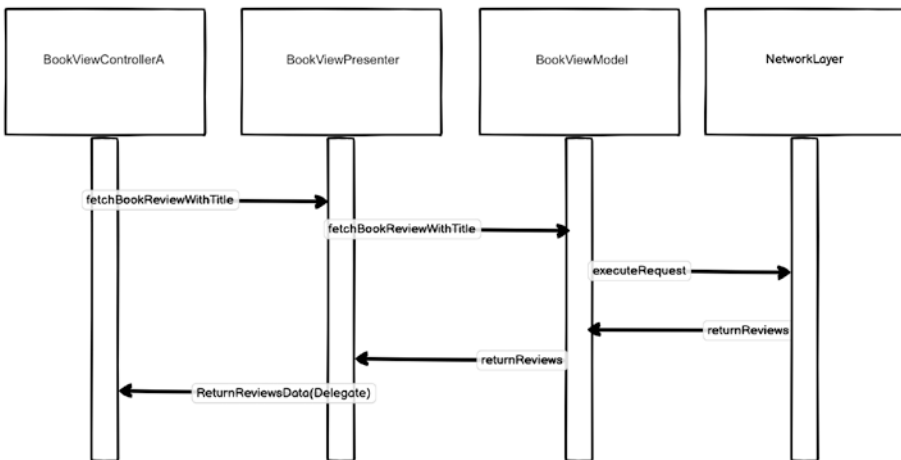


Figure 11-4. MVP Design

As we can see in the diagram, `BookViewControllerA` depends on the presenter to return data that will be displayed inside the `TableView`. `BookViewPresenter` depends on `BookViewModel` in order to return the reviews array. `BookViewModel` depends on `NetworkLayer` to make the request.

Now let's convert this to a test. Let's create a new test case class named `BookViewIntegrationTests` and add this test to it:

```
func testFetchingBooksReturnsAReviewInPresenterDelegate () {
    // Given
    let testBundle = Bundle(for: type(of: self))
    let booksReveiwJSONURL = testBundle.url(forResource:
        "booksReview", withExtension: "json")
    let booksReveiwJSON = try! Data(contentsOf:
        booksReveiwJSONURL!)
    let networkLayer = NetworkLayerStub(stubbedData:
        booksReveiwJSON)

    let bookViewModel = BookViewModel(network: networkLayer)
    let bookViewPresenter = BookViewPresenter(bookViewModel:
        bookViewModel)
    let delegateMock = BookViewPresenterDelegateMock()
    bookViewPresenter.delegate = delegateMock

    let expectation = XCTKVOExpectation(keyPath: "review",
        object: delegateMock)

    // When
    bookViewPresenter.fetchBookReviews(title: "THE LAST THING
    HE TOLD ME")
}
```



```

// Then
self.wait(for: [expectation], timeout: 0.1)
XCTAssertTrue(delegateMock.review == "The book is
interesting", "Fetched fetch and view expected reviews")
}

```

This is a slightly complicated test, so let's break it down:

- Given section:
 - We create an instance of `NetworkLayerStub` that returns the content of `booksReview.json`.
 - We create an instance of `BookViewModel` that depends on `NetworkLayerStub`.
 - We create an instance of `BookViewPresenter` that depends on `BookViewModel`.
 - We create an instance of `BookViewPresenterDelegateMock` and set it as the delegate of our presenter.
- When section: We call `fetchBookReviews` with the expectation that the presenter will eventually call its delegate and pass it the book reviews.
- Then section: We wait for the expectation and assert on the value of the reviews.

Since almost all the classes we used in this test are still not created, we'll have to comment out this test until we're done with our unit test phase. This is to allow our tests to build. Then we'll come back and run it to make sure we're done.

Unit Tests and Actual Implementation

So far we've just been adding tests as you can see in Figure 11-5. But since we've reached this level of testing, it means we're close to actually adding code. We'll need to implement each component in our design (Figure 11-5).

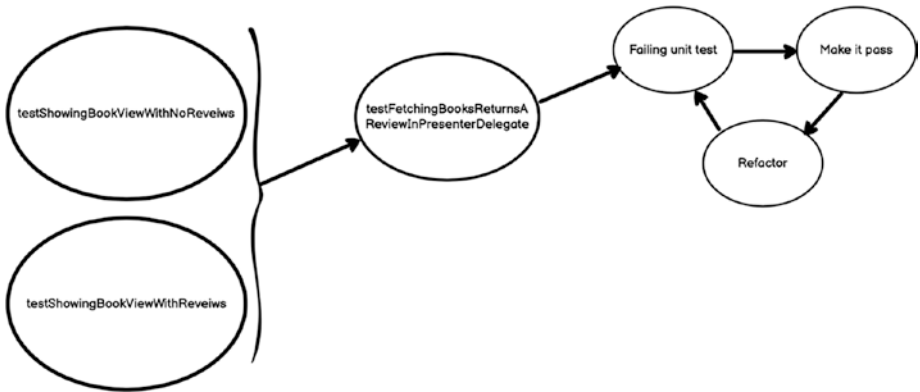


Figure 11-5. Testing plan diagram (integration test added)

BookViewModel

We'll start by implementing `BookViewModel`. To do that we'll create a new test case class and name it `BookViewModelTests` (Figure 11-6). And add this to it:

```

func testFetchingBookReveiw[s]() throws {
    // Given
    let expectedReveiw[s]: [Review] = stubbedReveiw[s]()
    let testBundle = Bundle(for: type(of: self))
    let booksReveiw[s]JSONURL = testBundle.url(forResource:
        "booksReveiw[s]", withExtension: "json")
    let booksReveiw[s]JSON = try Data(contentsOf:
        booksReveiw[s]JSONURL!)
    let networkLayer = NetworkLayerStub(stubbedData:
        booksReveiw[s]JSON)
  
```

```

let bookViewModel = BookViewModel(network: networkLayer)

// When
var actualReviews: [Review]?
let waitForBookReviews = XCTestExpectation(description:
"Wait to fetch book reviews")
bookViewModel.fetchBookReviews(with: "Title", callback: {
reviews in
    actualReviews = reviews
    waitForBookReviews.fulfill()
})

// Then
self.wait(for: [waitForBookReviews], timeout: 0.1)
XCTAssertEqual(actualReviews, expectedReviews, "Fetched
books does not match the expected")
}

func stubbedReviews() -> [Review]{
    return [Review(byLine:"ERROL MORRIS", summary:"The book is
interesting")]
}

```

Let's break it down.

- Given section:
 - We create an array of stubbed reviews that returns the content of `booksReview.json`.
 - We create an instance of `NetworkLayerStub` that returns the content of `booksReview.json`.
 - We create an instance of `BookViewModel` that depends on `NetworkLayerStub`.

- When section: We call `fetchBookReviews` with the expectation that the model will return the actual reviews.
- Then section: We wait for the expectation and assert on the value of the reviews.

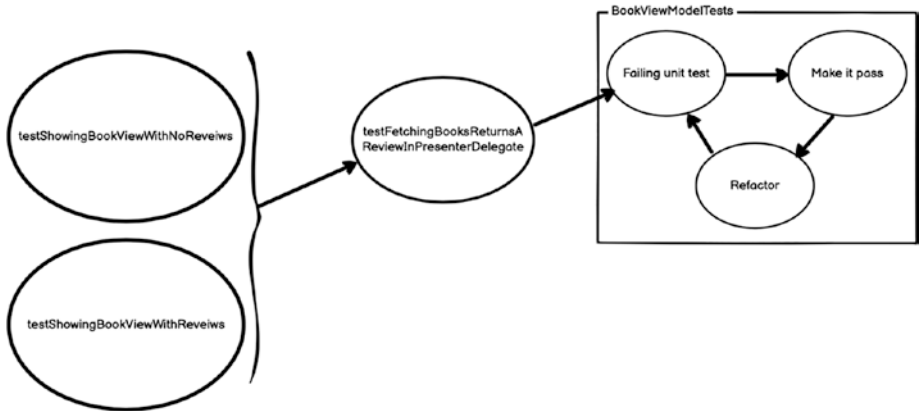


Figure 11-6. Testing plan diagram (unit test added)

For our test to build, we need to do a couple of things. First, we need to create the `Review` object and make sure it implements `Codable` and `Equatable`:

```
// MARK: - ReviewsResponse
struct ReviewsResponse: Codable {
    let status, copyright: String
    let numResults: Int
    let results: [Review]

    enum CodingKeys: String, CodingKey {
        case status, copyright
        case numResults = "num_results"
        case results
    }
}
```

```
// MARK: - Review
struct Review: Codable, Equatable {
    var byLine: String?
    var summary: String?

    init(byLine:String, summary:String) {
        self.byLine = byLine
        self.summary = summary
    }

    enum CodingKeys: String, CodingKey {
        case byLine = "byline"
        case summary = "summary"
    }

    static func == (lhs: Review, rhs: Review) -> Bool {
        lhs.byLine == rhs.byLine &&
        lhs.summary == rhs.summary
    }
}
```

Now let's update the `BookViewModel` class. The class should have a dependency on `NetworkLayer` and should have the `fetchBookReviews` public function:

```
class BookViewModel {
    private var favoritesManager:FavoritesManager?
    private var networkLayer: NetworkLayer?

    init(networkLayer: NetworkLayer? = .init(), favoritesManager:
FavoritesManager? = .shared) {
        self.networkLayer = networkLayer
        self.favoritesManager = favoritesManager
    }
}
```

```

public func addFavorite(_ model: BookModel) {
    self.favoritesManager?.addFavorite(model)
}

public func fetchBookReviews(with title:String, callBack:
@escaping (_ reviews:[Review]?) -> Void) {
    callBack(nil)
}
}

```

Here we create our class and add the needed function with empty implementation.

If we run our test, it will fail, which is expected. Now we need to actually implement `fetchBookReviews`. To do so we need to make a network request. This means we need to create a new struct conforming to `RequestProtocol` that describes the request we need to make.

Let's create a new test case class and name it `ReviewsRequestTests`. And we'll add these tests to it:

```

func testReviewsRequestHTTPMethod() {
    //Given
    let reviewsRequest = ReviewsRequest(title: "title")

    //When & Then
    XCTAssertEqual(reviewsRequest.method, .GET)
}

func testReviewsRequestURL() {
    //Given
    let bookRequest = ReviewsRequest(title: "title")
    let env = APIEnvironment(scheme: "http", host: "test.com",
port: 433, API_KEY: "")

    // When
    let urlRequest = bookRequest.createURLRequest(with: env)
}

```

```

    //When & Then
    XCTAssertEqual(urlRequest?.url?.absoluteString, "http://
test.com:433/svc/books/v3/reviews.json?title=title&api-
key=\(APIEnvironment.production.API_KEY)")
}

```

```

func testReviewsRequestBody() {
    //Given
    let reviewsRequest = ReviewsRequest(title: "title")

    //When & Then
    XCTAssertNil(reviewsRequest.body)
}

```

Now to get these tests to pass, we'll need to create `ReviewsRequest` like so:

```

struct ReviewsRequest: RequestProtocol {
    var title:String
    var path: String {
        return "/svc/books/v3/reviews.json"
    }
    var queryItems: [URLQueryItem]? {
        return [URLQueryItem(name: "title", value: self.title),
        URLQueryItem(name: "api-key", value: NetworkLayer.
        environment.API_KEY)]
    }
    var method:HTTPMethod {return .GET}
    var body: Data? {return nil}
}

```

Now if we run `ReviewsRequestTests` (Figure 11-7), they will pass ✓.



Figure 11-7. *ReviewsRequest tests passing*

Now that we have `ReviewsRequest` ready, we can implement `fetchBookReviews` properly:

```
public func fetchBookReviews(with title:String, callback:
@escaping (_ reviews:[Review]?) -> Void) {      self.
network?.executeRequest(ReviewsRequest(title: title), callback:
{ data, Error in
    guard let data = data else {
        callback(nil)
        return
    }
    var response:ReviewsResponse?
    do {
        response = try JSONDecoder().
        decode(ReviewsResponse.self, from: data)
    } catch {
        print(error.localizedDescription)
    }
    if let reviews = response?.results {
        callback(reviews)
        return
    }
}
```



```

        callback(nil)
    })
}

```

Here we make our network request, and we then parse the response to `Review` objects and return it in the callback. And if any error occurs, we return `nil`.

Now if we run the test in `BookViewModelTests`, it should pass ✓.

BookViewPresenter

Now let's jump to our presenter. As usual, we'll start by creating a test case class and name it `BookViewPresenterTests`. And we'll add this test to it:

```

func testFetchingBookReveiwInDelegate() {
    // Given
    let bookViewModel = BookViewModelStub(stubbedReviews:
stubbedReviews())
    let bookViewPresenter = BookViewPresenter(bookViewModel:
bookViewModel)
    let delegateMock = BookViewPresenterDelegateMock()
    bookViewPresenter.delegate = delegateMock
}

func stubbedReviews() -> [Review]{
    return [Review(byLine:"ERROL MORRIS", summary:"The book is
interesting")]
}

```

Here we create an instance of `BookViewPresenter` injected with a stub for our view model. And we set its delegate to an instance of `BookViewPresenterDelegateMock`. Since all these classes don't exist, we'll need to create them so that our test can build.

We'll start with `BookViewPresenter`:

```
protocol BookViewPresenterDelegate: AnyObject {
    func reviewDidFinish(_ review: String?)
}

class BookViewPresenter {
    private var bookViewModel: BookViewModel
    weak var delegate: BookViewPresenterDelegate?

    init(bookViewModel: BookViewModel) {
        self.bookViewModel = bookViewModel
    }
}
```

Here we define the protocol for our delegate. And we create our class that has a dependency on `BookViewModel`.

Now let's create `BookViewModelStub`:

```
class BookViewModelStub: BookViewModel {
    var stubbedReviews:[Review]?

    init(stubbedReviews:[Review]) {
        self.stubbedReviews = stubbedReviews
        super.init(network: nil)
    }

    override public func fetchBookReviews(with title:String,
        callback: @escaping (_ reviews:[Review]?) -> Void) {
        callback(self.stubbedReviews!)
    }
}
```

This simply takes an array of reviews as the stubbed data and returns it whenever `fetchBookReviews` is called.

Finally we need to create `BookViewPresenterDelegateMock`:

```
class BookViewPresenterDelegateMock: BookViewPresenterDelegate
{
    public var review:String?

    func reviewDidFinish(_ review: String?) {
        self.review = review
    }
}
```

This here simply conforms to `BookViewPresenterDelegate` and saves the passed value in a variable.

Since our test is building now, it's time to write the rest of it:

```
func testFetchingBookReveiwInDelegate() throws {
    // Given
    let bookViewModel = BookViewModelStub(stubbedReviews:
    stubbedReviews())
    let bookViewPresenter = BookViewPresenter(bookViewModel:
    bookViewModel)
    let delegateMock = BookViewPresenterDelegateMock()
    bookViewPresenter.delegate = delegateMock

    // When
    let expectation = XCTKVOExpectation(keyPath: "review",
    object: delegateMock)
    bookViewPresenter.fetchBookReviews(title: "Title")

    // Then
    self.wait(for: [expectation], timeout: 0.1)
    XCTAssertEqual(delegateMock.review, "The book is
    interesting")
}
```

Here we call `fetchBookReviews` and expect that our delegate will be called. We then assert on the value passed to our delegate.

We use a KVO expectation in our test. And in order to make this expectation work, we need to make `BookViewPresenterDelegateMock` inherit from `NSObject` and annotate the variable we are listening on with `@objc` and `dynamic`:

```
class BookViewPresenterDelegateMock: NSObject,
BookViewPresenterDelegate {
    @objc dynamic var review:String = ""

    func reviewDidFinish(_ review: String?) {
        self.review = review ?? ""
    }
}
```

Now to make our test build and pass, we need to implement `fetchBookReviews`:

```
func fetchBookReviews(title:String) {
    self.bookViewModel?.fetchBookReviews(with:title, callBack:
{ reviews in
    var dataToBeDisplayed: String?
    if let reviews = reviews, reviews.count > 0 {
        let firstReview = reviews[0]
        dataToBeDisplayed = firstReview.summary
    }

    DispatchQueue.main.async {
        self.delegate?.reviewDidFinish(dataToBeDisplayed)
    }
    })
}
```

Here we use the view model to fetch the reviews, and we get the summary of the first review and pass it to our delegate.

Now if we run our test (Figure 11-8), it should pass ✓.



Figure 11-8. *Presenter test passing*

Let's add a new test to handle the scenario where the view model does not return reviews. It will be almost identical to our first test:

```
func testFetchingBookReveiwReturnsNoResultsInDelegate() throws
{
    // Given
    let bookViewModel = BookViewModelStub(stubbedReviews: [])
    let bookViewPresenter = BookViewPresenter(bookViewModel:
    bookViewModel)
    let delegateMock = BookViewPresenterDelegateMock()
    bookViewPresenter.delegate = delegateMock

    // when
    let expectation = XCTKVOExpectation(keyPath: "review",
    object: delegateMock)
    bookViewPresenter.fetchBookReviews(title: "Title")
    self.wait(for: [expectation], timeout: 0.1)

    // Then
    XCTAssertEqual(delegateMock.review, "No Reviews Available")
}
```

Here we just pass an empty array to our view model stub, and we assert that the value passed to our delegate is the expected empty state text.

This test will fail. To fix it, we need to handle this case in our code. It will be as simple as adding a default value like so:

```
func fetchBookReviews(title:String) {
    self.bookViewModel?.fetchBookReviews(with:title, callBack:
{ reviews in
    var dataToBeDisplayed: String?
    if let reviews = reviews, reviews.count > 0 {
        let firstReview = reviews[0]
        dataToBeDisplayed = firstReview.summary
    }

    DispatchQueue.main.async {
        self.delegate?.reviewDidFinish(dataToBeDisplayed ??
        "No Reviews Available")
    }
    })
}
```

Since the value passed back to the `BookViewPresenterDelegate` is no longer optional, we can update our delegate to this:

```
protocol BookViewPresenterDelegate: AnyObject {
    func reviewDidFinish(_ review: String)
}
```

If we run our presenter tests (Figure 11-9), they should pass ✓.

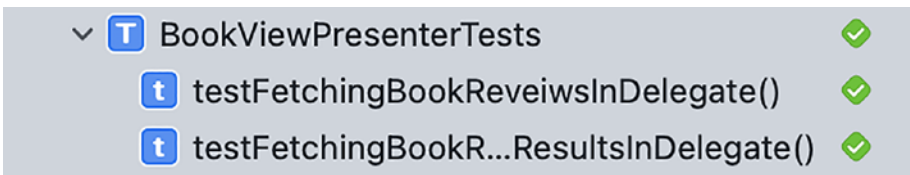


Figure 11-9. Presenter tests passing

Now that all our unit tests are passing, let's uncomment our integration test inside `BookViewIntegrationTests` and try to run it. It should pass now as well ✓.

Final Steps

We will not be doing this step in detail in this chapter as it's a bit trivial. However, what we need to do is to make use of our new presenter inside our view controllers. The view controllers need to conform to `BookViewPresenterDelegate`, and we need to call `fetchBookReviews` in `viewDidLoad`. When the presenter calls `reviewDidFinish`, we should use the data passed and populate our view. When we do this, our UI tests should all pass.

We can go the extra mile and create a new class called `BookViewControllerBase` and implement this functionality inside it. And then we'll have our two view controllers inherit from it.

Summary

Using TDD on legacy code can be a bit challenging. Developers normally tend to avoid using TDD and best practices when working on legacy code. However, we should always try to leave any code we work on better than we've found it. And this applies to adding new features to legacy apps. Even if we don't have the time to refactor the whole app, the code we add needs to be well designed, tested, and maintainable. And this actually sets the path to transforming the legacy code to well-designed code.

In this chapter we worked on adding a new feature to our legacy app. If we had followed the standards the old code followed, we would've ended up with more duplicated code that is impossible to test. Instead we applied TDD and ended up with a new feature that works perfectly with the old code and is well designed and highly covered by tests at the same time.

CHAPTER 12

Handling Production Issues

App quality has been a prominent topic in this book. We talked about our external quality and internal quality. We also talked about how using TDD can significantly enhance our quality. But quality is something we need to be always working toward. Even the biggest companies are constantly working toward enhancing their quality. As we said, having a well-tested project helps in avoiding setbacks in our quality. But it doesn't eliminate them. Even if we follow TDD in everything we do, we might still miss a few corner cases. In the end we're only human.

Our Tool

To be able to proactively work toward better quality, we need to be able to track two things: bugs and crashes. We need to be able to track the crashes our users encounter. This is an extremely hard thing to implement ourselves. However, thankfully there are many third-party tools that can provide us with this. We also need to provide our users with a way to communicate with us any bugs they encounter while using our app. We can manually implement this in a very basic way. However, there are also tools that can provide us with this functionality along with a collection of

added functionalities (network logs, console logs, user steps, device state with every bug reported).

In this chapter, we are going to use Instabug for bug reporting and crash reporting. It's perfectly fitting what we exactly need to keep track of our bugs/crashes. We will show you how you can use it to be able to reproduce your bugs/crashes so that you can write tests to fix them.

Integration

First, let's open up our starter project from this chapter's resources. Now in order to integrate our tool, we'll need to go to their [website](#) and sign up. When we sign up, we'll be provided with a token, which is what we'll need to link our account to our app.

Next, we will add Instabug's SDK to our app using Swift Package Manager (Figure 12-1). Their package lives in this repo: <https://github.com/Instabug/Instabug-SP>.

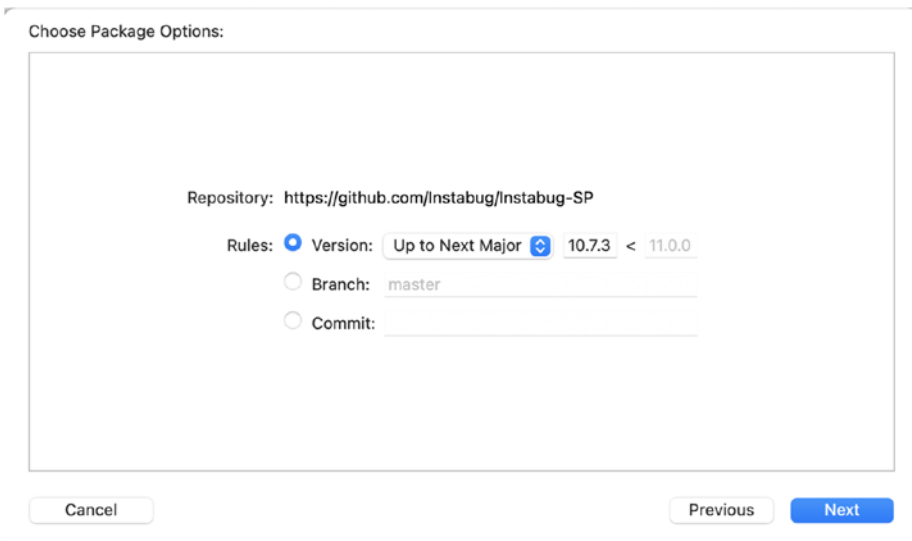


Figure 12-1. Adding a third-party library using SPM

Finally we will add this line to the `AppDelegate.swift`, and you are ready to go 🚀.

```
Instabug.start(withToken: "TOKEN", invocationEvents: .shake)
```

Production Bug

We just received our first bug (Figure 12-2). A user is complaining that they can't find any books.

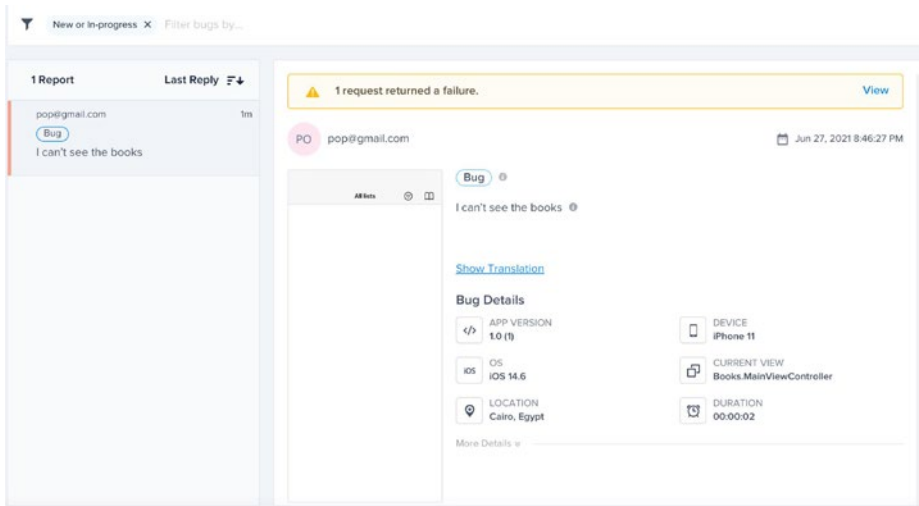


Figure 12-2. User submitted a bug report

Debugging

From the attached screenshot (Figure 12-2), we can see that the `MainViewController` is empty. So this means that their complaint is valid.

After checking the bug report and looking at the network logs (Figure 12-3), we can see that the books request failed. So this behavior is expected. However, we don't show any error messages at all.

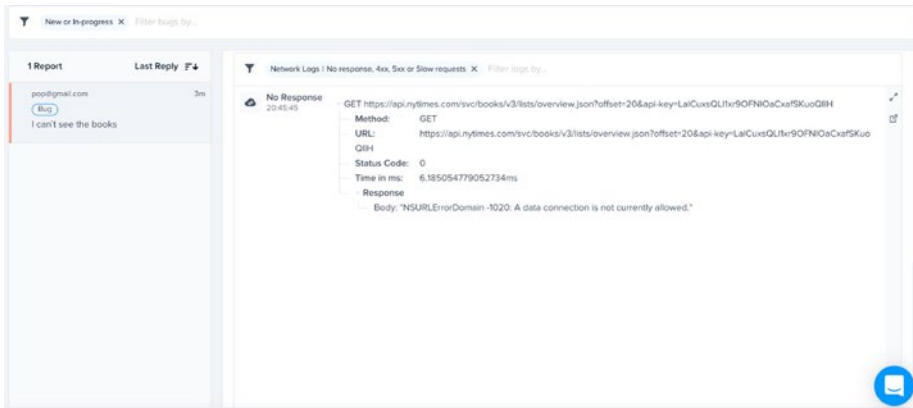


Figure 12-3. Network logs from bug report

What we need to do here is once the books request failed, we need to make sure to show this message: “Failed to load best seller books”.

Fixing this should be simple. But even if it was as simple as adding one letter, we still need to use TDD. The rule of TDD is that we can’t write any code without having a failing test. And given that we’ve shipped this bug to production, then this means that we don’t have tests covering this scenario.

UI Test

Let’s open `BooksUITests` and add a new test to simulate this bug. The test should look like this:

```
func testShowingErrorMessageWhenFailedToFetchBooksRequest() {
    // Given
    server.GET["/svc/books/v3/lists/overview.json"] = {_ in
        HttpResponseMessage.notFound}
}
```

```

let app = XCUIApplication()
app.launchArguments += ["TESTING"]
app.launch()

// When
let booksTableView = app.tables

// Then
let failureMessage = booksTableView.staticTexts["Failed to
fetch best seller books"]
_ = failureMessage.waitForExistence(timeout:10)
}

```

Here we stub our request as we used to do, but now we return a failed response. Then we assert that the error message is displayed.

Unit Tests

The `MainViewPresenter` is the class that should be responsible for returning an error message based on the list returned from the `MainViewModel`.

`fetchBestSellerBooks` returns a list only. We need to extend this method to return a `Boolean` to indicate if the presenter succeeded in fetching the request or not and an error message to be displayed to our user. Let's add a new test in `MainViewPresenterTests`:

```

func testFailureToFetchBooks() throws {
    // Given
    let mainViewModel = MainViewModelStub(stubbedLists: [])
    let mainViewPresenter = MainViewPresenter(mainViewModel:
mainViewModel)
    var status:Bool?
    var message:String?

```

```

var actualLists: [List] = []

// when & then
let waitForBooks = XCTestExpectation(description: "Wait to
fetch books")
mainViewPresenter.fetchBestSellerBooks { lists, success,
errorMessage in
    actualLists = lists ?? []
    status = success
    message = errorMessage
    waitForBooks.fulfill()
}

self.wait(for: [waitForBooks], timeout: 0.1)
XCTAssertEqual(actualLists, [])
XCTAssertEqual(status, false)
XCTAssertEqual(message, "Failed to fetch best seller books")
}

```

Here we tell our stub to return an empty array. And then we call `fetchBestSellerBooks` that now returns a Boolean indicating success and an error message in case of failure. Then we assert on the values returned in the callback.

To fix this test, we need to update `fetchBestSellerBooks` to handle this case:

```

public func fetchBestSellerBooks(callBack: @escaping (_
data:[List]?, _ success:Bool, _ errorMessage:String?) -> Void)
{
    self.mainViewModel?.fetchBestSellerBooks(callBack: { lists in
        if let lists = lists, lists.count > 0 {
            callBack(lists, true, nil)
        } else {

```

```

        callBack([], false, "Failed to fetch best seller
        books")
    }
})
}

```

This will cause multiple build errors in our code and tests since we've changed the signature of the function. We just need to pass through every build error and update the signature.

After fixing all build errors, if we run our new test in `MainViewPresenterTests` (Figure 12-4), it should now pass ✓.

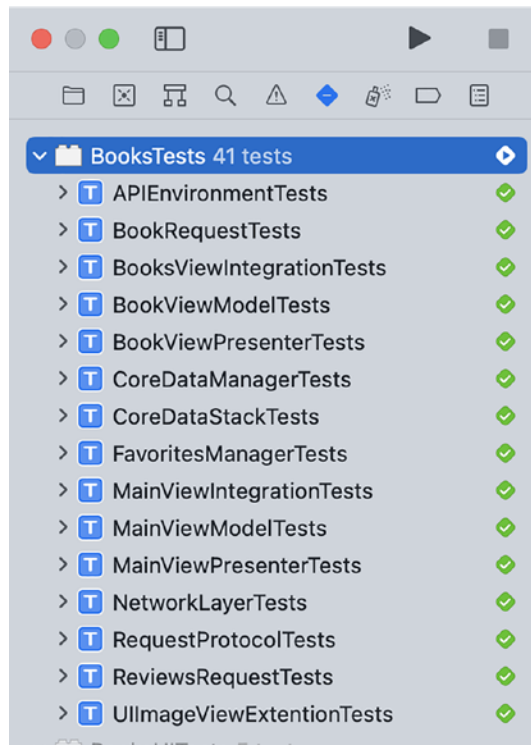


Figure 12-4. All tests passing

Now we just need to update our view controller to display the error:

```
func fetchBooks() {
    self.mainViewPresenter?.fetchBestSellerBooks(callback: {
        lists, success, errorMessage in
        if success {
            if let lists = lists {
                self.lists = lists
                DispatchQueue.main.async {
                    self.refreshControl.endRefreshing()
                    self.tableView?.reloadData()
                }
            }
        } else {
            self.lists = lists
            DispatchQueue.main.async {
                self.refreshControl.endRefreshing()
                self.tableView?.reloadData()
                self.showErrorMessage(errorMessage:
                    errorMessage)
            }
        }
    })
}

func showErrorMessage(errorMessage:String?) {
    let label = UILabel(frame: CGRect(x: 0, y: 0, width: 100,
        height: 40))
    label.translatesAutoresizingMaskIntoConstraints = false
    label.text = errorMessage
    label.sizeToFit()
    self.tableView?.addSubview(label)
}
```

```

label.centerXAnchor.constraint(equalTo: (self.tableView?.
centerXAnchor!).isActive = true
label.centerYAnchor.constraint(equalTo: (self.tableView?.
centerYAnchor!).isActive = true
}

```

Now if we run the UI test (Figure 12-5), it should also pass ✓.

```

110 ✓
111 func testShowingErrorMessageWhenFailedToFetchBooksRequest() throws {
112     // Given
113     server.GET["/svc/books/v3/lists/overview.json"] = {_ in HttpResponse.notFound}
114
115     let app = XCUIApplication()
116     app.launchArguments += ["TESTING"]
117     app.launch()
118
119     // When
120     let booksTableView = app.tables
121
122     // Then
123     let failureMessage = booksTableView.staticTexts["Failed to fetch best seller books"]
124     _ = failureMessage.waitForExistence(timeout: 1.0)
125 }

```

Figure 12-5. UI test passing

Production Crash

We just received our first crash with number of occurrences 3 (Figure 12-6).

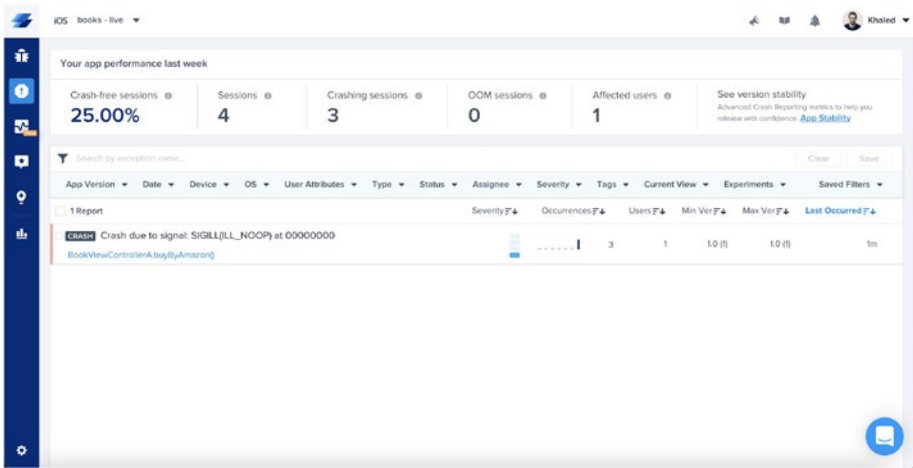


Figure 12-6. Crash report

Debugging

If we look at the crash stack trace (Figure 12-7), we’ll find that it happens when someone tries to buy a book using Amazon.

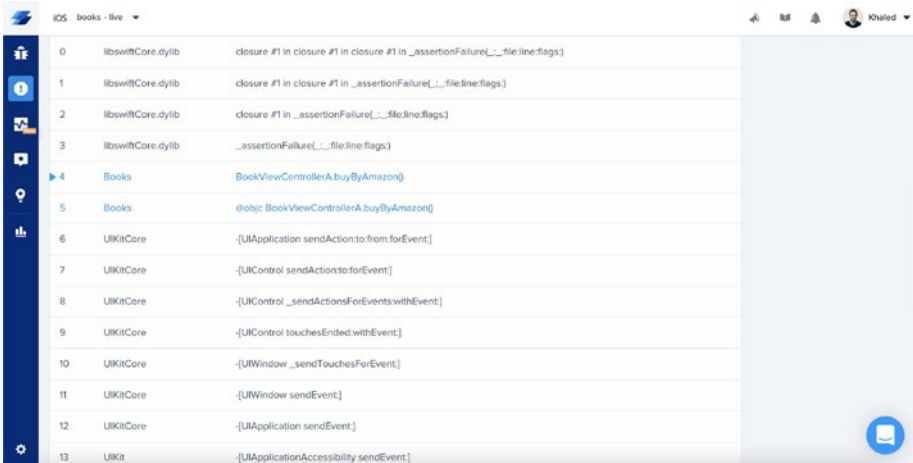


Figure 12-7. Crash stack trace

First though, it could be that it's an issue from the web service we're using. It's possible that the book returned does not contain an Amazon link or something. But if we check the network logs (Figure 12-8), we'll find that the web service returned a correct response.

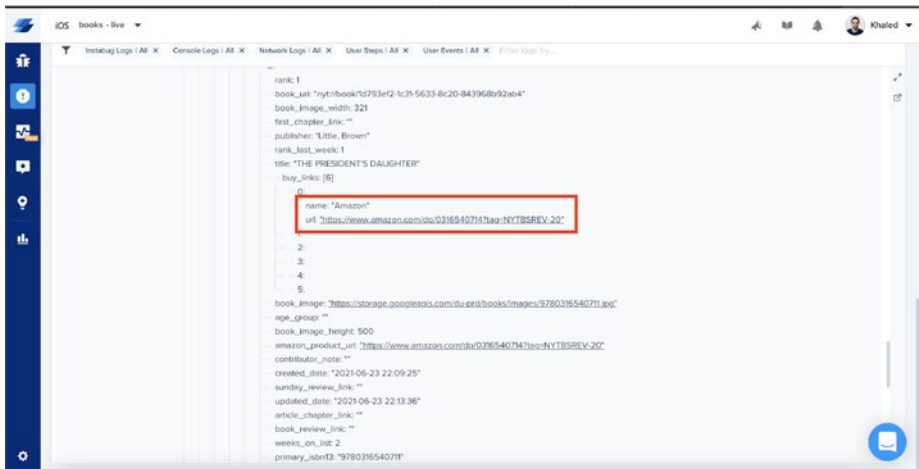


Figure 12-8. Network logs

If we debug our crash further and look at the user steps for all three occurrences (Figure 12-9), we can reach the conclusion that all crashes happened inside `BookViewControllerA`. And they always happened after going to the background and coming back to foreground.

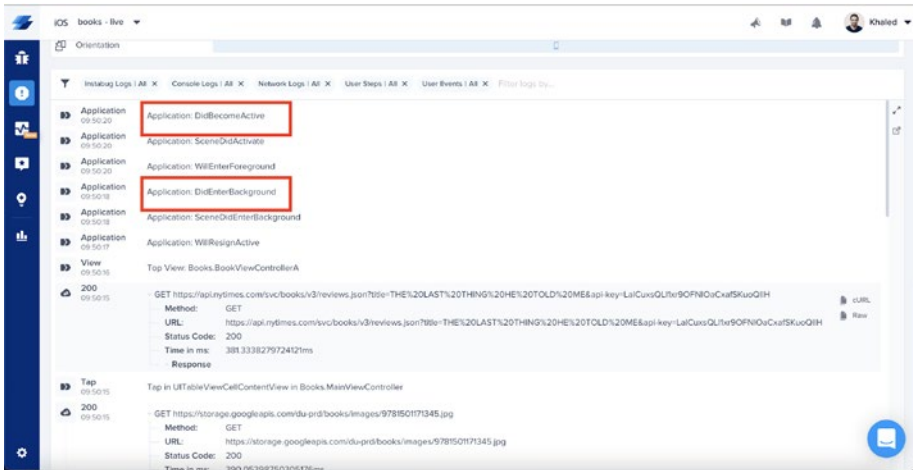


Figure 12-9. Steps prior to crash

If we check the code inside `BookViewControllerA`, we'll find the culprit:

```
NotificationCenter.default.addObserver(self, selector:
#selector(didEnterBackground), name: UIApplication.
didEnterBackgroundNotification, object: nil)
```

We listen on the `didEnterBackground` notification, and when it's fired we do this:

```
@objc func didEnterBackground() {
    self.book = nil
}
```

And when a user taps on the Amazon button

```
@IBAction func buyByAmazon() {
    for buyLink in self.book!.buyLinks! {
        if buyLink.name == .amazon {
            if let url = URL(string: buyLink.url) {
                UIApplication.shared.open(url)
            }
        }
    }
}
```

```

    }
  }
}

```

we force-unwrap our book instance to be able to use it.

Gotcha!!!

So now that we have our root cause, we will do the same thing we did with our bug. We'll apply TDD.

UI Test

Let's open `BooksUITests` and add a new test named `testShowingBookViewAfterEnterBackground` to simulate the scenario that causes the crash.

The Given section of the test should look like this:

```

// Given
let testBundle = Bundle(for: type(of: self))
let booksJSONURL = testBundle.url(forResource:
"BestSellerBooksStub", withExtension: "json")
let booksJSON = try! String(contentsOf: booksJSONURL!)
let booksNoReveiwJSONURL = testBundle.url(forResource:
"booksNoReview", withExtension: "json")
let booksNoReveiwJSON = try! String(contentsOf:
booksNoReveiwJSONURL!)
server.GET["/svc/books/v3/lists/overview.json"] = {_ in
  HttpResponse.ok(.text(booksJSON))}
server.GET["/svc/books/v3/reviews.json?title=THE+LAST+THING+HE+
TOLD+ME"] = {_ in HttpResponse.ok(.text(booksNoReveiwJSON))}

```

```
let app = XCUIApplication()
app.launchArguments += ["TESTING"]
app.launch()
```

Here we just set up our test by stubbing our two requests and then launching the app.

Now on to the “When” section:

```
// When

// Go to book
let booksTableView = app.tables
let cells = booksTableView.cells
let firstCell = cells.firstMatch
_ = firstCell.waitForExistence(timeout: 1.0)
firstCell.tap()

// Move to background
XCUIDevice.shared.press(.home)

// Move back to foreground
app.activate()
```

Here we navigate to a book details page. And then we go to the background and then back to the foreground.

Finally our “Then” section:

```
// Then
let amazonButton = app.buttons["amazon"]
_ = amazonButton.waitForExistence(timeout: 1.0)
amazonButton.tap()
```

Here we should tap on the Amazon button. Normally in the Then section we do some assertions. However, for this test, our assertion is that the app doesn’t crash.

Handle A/B Testing

Now we have a problem: every time the test runs, it may open `BookViewControllerA` or `BookViewControllerB`. This is because of our A/B testing experiment that chooses a view controller by random. So if it chooses to go to `BookViewControllerB`, our test will pass even though it should fail. For our test to be effective, we need it to fail consistently.

So we need to add another launch argument inside our UI test to force our app to use the first experiment:

```
let app = XCUIApplication()
app.launchArguments += ["TESTING", "detailsA"]
app.launch()
```

We need to adjust the `AppDelegate` to force a specific experiment:

```
if ProcessInfo.processInfo.arguments.contains("TESTING"){
    if ProcessInfo.processInfo.arguments.
        contains("detailsA") {
        UserDefaults.standard.set(true, forKey: "detailsA")
    } else {
        UserDefaults.standard.set(false, forKey: "detailsA")
    }
} else {
    let randomBool = Bool.random()
    if randomBool {
        Instabug.addExperiments(["detailsA"])
    } else {
        Instabug.addExperiments(["detailsB"])
    }
    UserDefaults.standard.set(randomBool, forKey: "detailsA")
}
```

Here we check if a launch argument is passed. If it is, we use the value passed; if not, we fall back to our normal implementation, which is choosing a view randomly.

Now, if we run our test, it should crash (Figure 12-10).

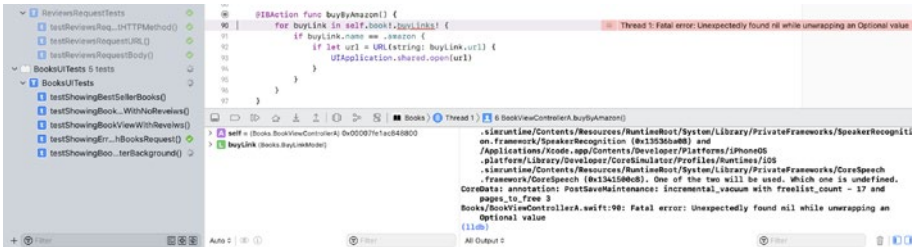


Figure 12-10. Crash reproduced

Fixing Our Test

Fixing our test, and in turn our production issue, is pretty simple. We just need to remove the force casting inside and replace our implementation with this:

```

@IBAction func buyByAmazon() {
    guard let buyLinks = self.book?.buyLinks else {
        return
    }

    for buyLink in buyLinks {
        if buyLink.name == .amazon {
            if let url = URL(string: buyLink.url) {
                UIApplication.shared.open(url)
            }
        }
    }
}

```

Here we use a guard to check if the book exists or not.

We should always avoid using force casting as it's extremely unsafe.

Most crashes happening on iOS are caused by force casting.

We can also remove the code that listens on the `DidEnterBackground` notification altogether as we don't seem to need it.

Now if we run our test (Figure 12-11), it should pass ✓.

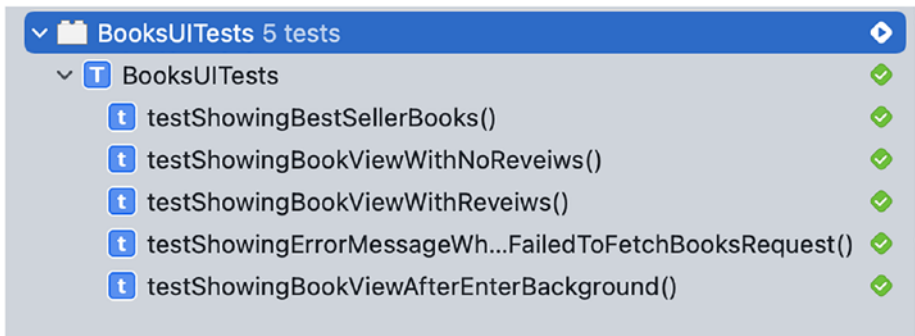


Figure 12-11. UI test passing

Summary

Our goal is to continuously improve our app quality. Sometimes it's possible to miss a certain scenario and not have it handled. We can't always predict how our users will interact with our app. That's why it's always best to have a way to track the fatal crashes happening to our production users and to also provide our users with a way to report faulty behaviors in our app.

In this chapter we talked about how to use third-party tools to keep track of bugs and crashes on production. When encountering a production issue, fixing it should also be test-driven. We used TDD when adding features by transforming our requirements to tests. With production bugs and crashes, it's the exact same thing, and our requirement is simply for the issue to not happen. When we do this, we will be preventing this specific issue from ever happening again.

Index

A

- A/B testing, 288, 325
- Accessibility identifier, 55, 57, 64
- Accessibility inspector, 60, 61
- addRandomNumber, 31
- App Store app, 123, 125, 132
- App Store module map, 126
- Assertion methods
 - comparison asserts, 31
 - equality asserts, 28
 - errors asserts, 32
 - nullability asserts, 29
 - truthfulness asserts, 28
- Async task illustration, 188
- Automated testing, 2, 3, 17

B

- Behavior-driven
 - development (BDD), 9
- Boilerplate tests, 48
- Books
 - fixing threading issues, 205–208, 210
 - networking (*see* Networking)
- BooksUITests target, 173
- BookViewModel, 297, 299–304
- BookViewPresenter, 304–309

- BookViewPresenterDelegate, 306
- Bug reporting, 312

C

- CalcTests.swift, 23
- CalculatorTests, 24, 38
- Code coverage, 39, 41
- CoffeeDrinksDataSource, 96–98
- CoffeeDrinksDataSourceStub, 99
- CoffeeDrinksModel, 100
- CoffeeDrinksModelTests, 98–102
- CoffeeDrinksPresenter, 103
- CoffeeDrinksPresenterTests, 102, 104–108
- Comparison asserts, 31
- Concurrency
 - cost, 188–190
 - definition, 183
 - GCD, 184
 - queues, 184
 - serial *vs.* concurrent queues, 185
 - sync *vs.* async, 186–188
- Core data
 - advanced fetching, 274–276
 - CoreDataManager, 251–253
 - CoreDataManagerTests, 262, 263

INDEX

Core data (*cont.*)

- CoreDataStack, 253–262
- creation
 - implementation, 267
 - saving changes, 268, 270
 - storable, 266
- fetching, 270, 271
- object graph, 245
- TestEntity, 264, 265
- testing stack, 250, 251
- updating, 272, 273

CoreDataManagerTests, 251, 253, 262, 263

- Core data stack, 253–260
 - managed object context, 249
 - managed object model, 247
 - persistent container, 249
 - persistent store, 248
 - persistent store coordinator, 248

Cost of concurrency, 188–190

Crash report, 312, 320

Crash stack trace, 320

D

DatabaseManager, 67

Deadlock, 189, 211

Debugging

- accessibility, 60
- accessibility inspector, 61
- production bug, 313
- production crash, 320, 321, 323

Dependency injection

- initializer injection, 169, 170

- property injection, 170, 171

Descendants relationship, 52

Double creation

- by inheritance, 163, 164
- protocols, 165–168

E

Equality asserts, 28

Errors asserts, 32, 42

Expectations

- creation, 34
- types, 35
- XCTest, 34
- XCTAssertTrue, 33

Explicit assertion, 194

F

Failing unit tests, 89, 96, 150

FavoritesManager, 277

FavoritesManager class, 279

fetchBestSellerBooks, 316

fetchBookReviews, 301, 303, 307

fetchBookReviews public
function, 300

fetchBooks(), 138, 149

G

getData() function, 99

Grand central dispatch (GCD),
184, 210

Granularity, 86, 87, 116

H

Health check testing
 bugs, 108
 failing tests, 109
 faulty code change, 108

Hypertext Transfer
 Protocol (HTTP)
 methods, 214
 requests, 214
 responses, 214

I, J, K

Implicit assertion, 194
 Inheritance, 163, 164
 Initializer injection, 169, 170
 Instabug, 312
 Integration tests, 70–76, 93–95
 iOS URL Loading System, 216, 243
 isLoggingEnabled(), 24, 26

L

Legacy App
 feature, 289
 implementation
 BookViewModel, 297–304
 BookViewPresenter, 304–309
 integration tests, 294, 296
 testing plan diagram, 290
 UI tests, 291–293
 Legacy app module map, 130
 Legacy Books app, 129

Legacy code disclaimer, A/B
 testing, 288
 LoginManager, 67, 72, 77

M

MainViewController, 134, 135
 MainViewController
 responsibilities
 diagram, 135
 MainViewIntegrationTests,
 139–141, 222
 MainViewModel, 142–146, 222
 MainViewModelTests, 144, 146
 MainViewPresenter, 146, 148,
 149, 315
 MainViewPresenterTests, 148
 Managed object context, 249
 Minimal viable product (MVP), 1
 Mocking, 160–163
 Mocking URLSession, 228–230, 232
 Model View Presenter (MVP),
 92, 117
 Modularization process
 class as starting point, 133
 class's responsibilities, 134, 135
 definition, 119
 initial module map, 132, 133
 refactor responsibilities (*see*
 Refactor responsibilities)
 Modularized app, 121
 advantages and
 disadvantages, 122
 module map, 131

INDEX

Module, [122](#), [124](#), [125](#)
MVP design pattern, [138](#)

N

Networking

- class's responsibilities, [220](#)
- design overview, [220](#), [221](#)
- execute request
 - mocking URLSession, [228–230](#), [232](#)
 - URLSession, [232](#), [233](#)
- failing request handling, [233](#), [235–237](#)
- Kickoff, [222](#)
- make request, [222](#)
- module, [219](#)
- process overview, [219](#)
- RequestProtocol, [224–227](#)
- showcasing test value, [233](#)
- verification tests, [222](#)

Networking ABCs

- HTTP requests, [214](#)
- HTTP responses, [214](#)
- in iOS
 - URLRequest, [218](#)
 - URLSession, [217](#)
 - URLSessionConfiguration, [217](#)
 - URLSessionTask, [218](#)
- URL, [215](#)

Non-modularized app, [121](#)
NSManagedObject class, [249](#)
NSManagedObjectModel, [247](#)

NSPersistentContainer, [249](#)
NSURLSessionTask., [244](#)
Nullability asserts, [29](#)

O

Object-oriented design, [91](#)
Object-oriented programming (OOP), [163](#)

P

Parallelize test execution, [63](#)
Persistent store

- coordinator, [240](#), [246](#), [248](#), [286](#)

Presenter test passing, [308](#)

Production bug

- debugging, [313](#)
- UI test, [314](#), [315](#)
- unit tests, [315–319](#)

Production crash

- A/B testing handle, [325](#), [326](#)
- debugging, [320](#), [321](#), [323](#)
- fixing our test, [326](#), [327](#)
- report, [320](#)
- UI test, [323](#), [324](#)

Property injection, [170](#), [171](#)

Protocol-oriented programming (POP), [165](#)

Q

Queues, [184](#), [185](#), [190](#), [210](#)

R

Race conditions, [189](#), [190](#)
 Randomized ordering, [38](#), [39](#)
 Randomize execution order, [39](#)
 Reader-writer problem
 race condition identifying,
 [191–193](#)
 singleton classes, [190](#), [191](#)
 TDD
 cycle, [193](#)
 failing test, writing, [193–197](#)
 make it pass, [203–205](#)
 race condition, [199](#)
 XCTestExpectation, [197](#)
 thread sanitizer, [201–203](#)
 Refactoring, [8](#), [14](#)
 Refactor responsibilities
 fetchBooks() func, [149](#)
 integration test, [139–141](#)
 MainViewModel, [142–146](#)
 MainViewPresenter,
 [146](#), [148](#), [149](#)
 NetworkLayer, [141](#), [142](#)
 rerun verification tests, [152](#)
 test value, [150](#), [151](#)
 verification tests, [136](#), [137](#)
 RequestProtocol, [224–227](#), [235](#)
 RequestProtocolTests, [225](#)
 returnFailure(), [160](#)
 returnSuccess(), [160](#)
 ReviewsRequestTests, [301](#)
 ReviewsRequest tests passing, [303](#)
 Runtime issue breakpoint, [203](#)

S

Serial queue, [185](#), [190](#), [203](#), [208](#)
 Serial queue task illustration, [185](#)
 setUp() function, [37](#)
 setUpWithError(), [23](#), [37](#)
 Shared resource, [190](#)
 Singleton classes, [190](#), [191](#)
 Sociable components, [70](#)
 Solitary components, [70](#)
 Stubbing, [160](#)
 app, [180](#)
 definition, [157](#)
 dependency, [158](#), [159](#)
 network, UI tests
 BestSellerBooksStub.json, [176](#)
 target membership,
 BestSellerBooksStub.json,
 [177](#)
 enabling HTTP for
 localhost, [178](#)
 implementation, [178](#)
 SPM, [173](#), [174](#)
 swifter, [173](#)
 Swift Package Manager (SPM),
 [173](#), [312](#)
 Sync task illustration, [187](#)

T

tearDownWithError(), [23](#)
 Test double
 creation
 by inheritance, [163](#), [164](#)

INDEX

Test double (*cont.*)

using protocols, 165–168

mocks, 160–163

stubs, 157–160

Test-driven development (TDD), 3

benefits, 8

cycle, 4, 5

external and internal quality, 6

fraction scenario, 13

implementing, 9, 11, 12

in Nutshell, 3

refactor error handling, 15

TaxCalculator, 16

test cases, 13

testing, 5

TestEntity, 264, 265

Test failure breakpoint, 36, 37

Testing pyramid

app screens, 66

integration tests, 70–76

plan diagram, 89

UI tests, 67–70

unit tests, 77–81

testInvalidCredentialsLogin, 75

testShowingBestSellerBooks(), 152

testShowingBookViewWithNo

Reviews, 291

Thread sanitizer, 201–203

Tool, integration, 312

Truthfulness asserts, 28

U

UIImageView, 205, 206

UITests

accessibility, 57–59

assertions, 56

boilerplate tests, 48

children relationship, 52

combining relationships, 54

containment relationship, 53

debugging accessibility, 60

descendants relationship, 52

improvement, 62, 63

interaction, UIElement, 56

launchArguments, 51

network stubbing, 171–180

querying, 51

target, 46, 47

test case, 50

testing pyramid, 67–70

Xcode, 45

XCUITest components, 49

Uniform resource locator (URL), 215

Unit tests

CoffeeDrinksDataSource, 96–97

CoffeeDrinksModelTests,

98–102

CoffeeDrinksPresenterTests,

102, 104–108

MainViewPresenter, 146–149

PersistenceManager/

DatabaseManager, 77

production bug, 315–319

validating, 77–80

URLRequest, 218

URLSession, 216, 217, 232, 233, 244

URLSessionConfiguration, 217

URLSessionDataTask, [229](#)
URLSessionDataTaskMock, [229](#)
URLSessionTask, [218](#), [220](#)

V

Validator component, [81](#)
Value assertion, [57](#)
ViewControllers, [134](#)

W

Wireframes, [88](#)

X, Y, Z

Xcode, [19](#), [23](#), [36](#)
XCTAssertEqual, [196](#)
XCTAssertEqual line, [35](#)
XCTAssert functions, [19](#)
XCTAssertTrue, [27](#)
XCTest, [19](#), [27](#)
XCTestExpectation,
 [34](#), [35](#), [197](#)
XCUIElement, [55](#)
XCUIElementQuery, [51](#)
XCUIElementTest components, [49](#)