CHAPTER 10

Introduction to Behavior-Driven Development

Behavior-Driven Development (BDD) is an approach to software development that was built with the aim of formalizing the best practices followed by Test-Driven Development practitioners. BDD as we know it today is the result of the efforts of Dan North and numerous others over the years. To read a detailed introduction to BDD, visit Dan North's website at https://dannorth.net/introducing-bdd/. This chapter will introduce you to BDD concepts and techniques.

What Is Behavior Driven Development

One of the key issues faced by people who are new to TDD is deciding what to test. Unfortunately TDD leaves this aspect open to the practitioner to decide. While experienced TDD practitioners know from experience what to test (and what not to), newcomers to TDD often do not know and, in some cases, give up on TDD altogether.

Behavior-driven development is about testing the behavior of a system, and not the implementation details. A system could be either an individual class or a group of classes that make up an aggregate unit of functionality.

As an example, consider the bank account project discussed in Chapter 4 with three key classes – BankAccount, AccountOwner, and Transaction. In terms of relationships, a BankAccount can have up to two AccountOwners and a variable number of Transactions (Figure 10-1).

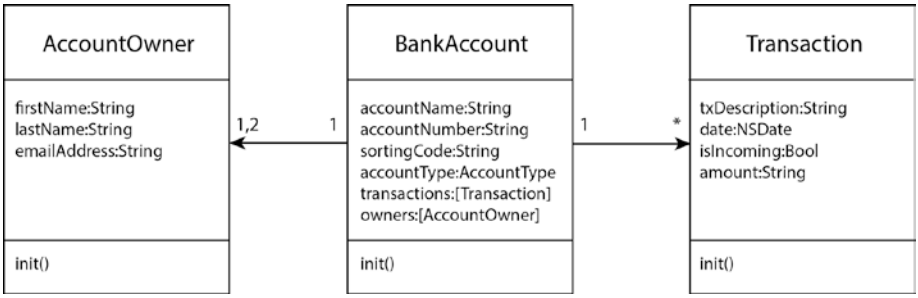

Figure 10-1. *Relationship Between Model Layer Objects*

A. Mishra, *iOS Code Testing*, DOI 10.1007/978-1-4842-2689-6_10

These model objects in isolation are not very useful from a business perspective. We have followed a rigid test-driven approach to developing these components in Chapter 4. The tests that we wrote verified that a number of validator objects worked as expected, and that creating a Model layer object makes calls to a number of validator objects. These tests, however, are of little value to a product owner as they don't directly tell him whether a business requirement is met.

The business requirement could, for instance, be something like this: As a joint account customer, I want to be able to withdraw money from my account if there is money in the account, so that I can use the cash to make a purchase.

To put it another way, the tests that we have written while following a test-driven approach are too detailed to be useful for a product owner to be able to verify that the developers have built the system that was asked of them.

The Difference between BDD and TDD

The key difference between behavior-driven development and test-driven development is that BDD tests are written at a different level of detail than TDD tests.

BDD-style tests system behavior is where the acceptable behavior of the system is defined by a set of scenarios, which are, in turn, derived from business requirements.

BDD-style tests are generally more descriptive and meaningful to the business. They are described in a language called Domain Specific Language (DSL) that contains terms and concepts encountered in the business domain.

BDD-style tests could, in theory, be written using the existing XCTest framework with cleverly thought of method names, and a fair bit of mocking and stubbing. In practice, BDD-style tests are written using a special framework. One such framework for iOS developers using Swift is called Quick.

Business Requirements and User Scenarios

The best way to understand how BDD works is to examine a concrete example. Let us assume that your company has been contracted to build a new banking system for retail operations, and after a few weeks of analysis, the business analyst has documented the following two requirements:

> **As a** [customer]
>
> **I want to** [deposit money in my savings bank account]
>
> **So that** [I can reach my savings goals]
>
> **As a** [customer].
>
> **I want to** [withdraw money from my savings bank account].
>
> **So that** [I can meet a financial obligation].

This is obviously an oversimplification of a real-world scenario where the business analyst has probably documented a few hundred requirements, but it serves to illustrate how a team practicing BDD would approach this problem.

A developer would then sit with the business analyst and a member of the QA team to agree on a set of user scenarios. Let us assume the team has been able to come up with the following two scenarios (again an oversimplification; in real life each requirement would expand into multiple scenarios):

Given [A joint savings account has a credit balance of $100]

When [An account holder withdraws $50 from the account]

Then [The account should have a credit balance of $50]

Given [A joint savings account has a credit balance of $100]

When [An account holder deposits $50 into the account]

Then [The account should have a credit balance of $150]

Once a set of user scenarios has been mutually agreed upon, the QA team will proceed to write QA scripts to test the scenarios when the system is testable using either automated testing techniques or manual testing techniques.

From User Scenarios to BDD Tests

The developer will then create a Swift class in the test target and write BDD-style tests using Quick. The name of the class will have the word "Specification" (or Spec) in it, as BDD tests are written to a specification provided by the business. Listing 10-1 presents a BDD-style test class called BankAccountSpecification.swift.

Listing 10-1. BankAccountSpecification.swift

```
import Foundation
import Quick
import Nimble

class BankAccountSpecification : QuickSpec {

    override func spec() {
        var mary:AccountOwner?
        var phil:AccountOwner?
        var maryAndPhil:[AccountOwner] = [AccountOwner]()
        var jointSavingsAccount:BankAccount?

        beforeEach {

            mary = AccountOwner(firstName: "Mary",
                                lastName: "Daniels",
                                emailAddress: "mdaniels@domain.com")

            phil = AccountOwner(firstName: "Phil",
                                lastName: "Burlington",
                                emailAddress: "p.burlington@domain.com")
```

```
            maryAndPhil.removeAll()
            maryAndPhil.append(mary!)
            maryAndPhil.append(phil!)

            jointSavingsAccount =
            BankAccount(accountName: "Savings Account",
            accountNumber: "87548390",
            sortingCode: "498711",
            accountType: .savingsAccount,
            owners: maryAndPhil)
        }

        describe("A joint savings account has a credit balance of $100") {
            context("An account holder withdraws $50 from the account") {
                it("The account should have a credit balance of $50") {

                 jointSavingsAccount?.setOpeningBalance(100)
                 jointSavingsAccount?.withdraw(50, mary)
                 expect(jointSavingsAccount!.accountBalance).to(equal(50))
                }
            }
        }

        describe("A joint savings account has a credit balance of $100") {
            context("An account holder deposits $50 into the account") {
                it("The account should have a credit balance of $150") {

                  jointSavingsAccount?.setOpeningBalance(100)
                  jointSavingsAccount?.deposit(50, mary)
                  expect(jointSavingsAccount!.accountBalance).to(equal(150))
                }
            }
        }

    }
}
```

The test case file starts out by importing the Quick and Nimble frameworks:

```
import Foundation
import Quick
import Nimble
```

Quick is a framework that allows you to write BDD-style tests in Swift. Nimble is a framework that lets you create assertions that are more verbose than the standard XCTAssert macros provide by Xcode.

Anatomy of a Quick Test Case

A Quick test case class is always a subclass of QuickSpec, and must have a method called spec in it. Tests for all user scenarios that define the specification are placed within the body of the spec() method:

```
class BankAccountSpecification : QuickSpec {

    override func spec() {

        // All test code goes here.

    }
}
```

Inside the spec() method, you will find call to a function called `beforeEach` with a single closure as the function argument:

```
class BankAccountSpecification : QuickSpec {

    override func spec() {

        beforeEach {

            // Setup code goes here

        }
    }
}
```

The beforeEach method of a Quick test case is equivalent to the setUp() method of an XCTestCase. Quick test cases can also have an afterEach method that would be the equivalent of the teardown() method of a unit test.

After the call to the beforeEach method (and before the call to the afterEach method if the test class has one), a number of BDD-style tests are written using nested calls to three functions: describe(), context(), it():

```
override func spec() {

    beforeEach {
    }

    describe(/* the "Given" part of a scenario statement*/) {
        context(/* the "When" part of a scenario statement*/){
            it(/* the "Then" part of a scenario statement */) {
                    // test logic goes here
            }
        }
    }
}
```

The describe() function takes a string argument that corresponds to the "Given" part of the scenario that you are testing and a trailing closure that contains statements to be executed by Quick when testing the scenario.

The context() function takes a string argument that corresponds to the "When" part of the scenario you are testing and a trailing closure that contains statements to be executed by Quick when testing the scenario.

The it() function also takes a string argument that corresponds to the "Then" part of the scenario you are testing and a trailing closure that contains the actual statements that will test your production code.

There is a one-to-one correspondence between a user scenario and a Quick BDD test. To make things easier to understand, Listing 10-2 presents a user scenario and its corresponding BDD test, written using Quick.

> **Given** [A joint savings account has a credit balance of $100]
>
> **When** [An account holder withdraws $50 from the account]
>
> **Then** [The account should have a credit balance of $50]

Listing 10-2. User Scenario and Corresponding Quick BDD Test

```
describe("A joint savings account has a credit balance of $100") {
  context("An account holder withdraws $50 from the account") {
     it("The account should have a credit balance of $50") {

          jointSavingsAccount?.setOpeningBalance(100)
          jointSavingsAccount?.withdraw(50, mary)
          expect(jointSavingsAccount!.accountBalance).to(equal(50))
     }
   }
}
```

From a business perspective, if this test passes it means that some tangible unit of functionality has been built – something that a customer can relate to.

Your test statements go in the it() block of a Quick BDD test. In the case of Listing 10-1, the test statements are the following:

```
jointSavingsAccount?.setOpeningBalance(100)
jointSavingsAccount?.withdraw(50, mary)
expect(jointSavingsAccount!.accountBalance).to(equal(50))
```

These tests are built assuming that a BankAccount object has methods called setOpeningBalance, withdraw(), and a computed property called accountBalance, which will behave in a manner consistent with the scenario being described.

To ensure that the BankAccount class behaves as expected, a test expectation statement is used:

```
expect(jointSavingsAccount!.accountBalance).to(equal(50))
```

The expectation statement is expressed using constructs available in the Nimble framework. Nimble is included with Quick and provides a more verbose method of creating an expectation.

However, there is nothing stopping you from using XCTest assert macros to make these expectations; the equivalent statement using the XCTAssertEqual macro would be this:

```
XCTAssertEqual(jointSavingsAccount!.accountBalance, 50)
```

Whether you choose to use Nimble over XCTest assert macros is a matter of personal preference. If you would like more information on Nimble assertions, visit the following URL:

https://github.com/Quick/Nimble

If you compare the BDD-style test with TDD-style tests, you should see that BDD style tests are more verbose, and focus on the what and not the how. There is nothing in these BDD tests that focuses on the details of the underlying implementation of the BankAccount class, just how it should behave in different scenarios.

The BankAccount class as developed in chapter 4 does not contain methods called setOpeningBalance(), withdraw(), deposit() or a computed property called accountBalance. Therefore, as with any test code, these tests will not compile just yet.

To get these tests to compile, the BankAccount class will have to be modified to resemble Listing 10-3.

Listing 10-3. Modified BankAccount.swift

```
import Foundation

enum AccountType {
    case currentAccount
    case savingsAccount
}

class BankAccount: NSObject {

    var accountName:String
    var accountNumber:String
    var sortingCode:String
    var accountType:AccountType
    var transactions:[Transaction]
    var owners:[AccountOwner]

    var accountBalance:Float {
        get {
            var balance:Float = 0.0
            for transaction in self.transactions {
                if let amount = Float(transaction.amount) {
```

```
                if transaction.isIncoming {
                    balance += amount
                } else {
                    balance -= amount
                }
            }
        }
        return balance
    }
}

init?(accountName:String,
      accountNumber:String,
      sortingCode:String,
      accountType:AccountType,
      owners:[AccountOwner],
      accountNameValidator:AccountNameValidator? = nil,
      accountNumberValidator:AccountNumberValidator? = nil,
      sortingCodeValidator:SortingCodeValidator? = nil) {

    let validator1 = accountNameValidator ?? AccountNameValidator()
    if validator1.validate(accountName) == false {
        return nil
    }

    let validator2 = accountNumberValidator ?? AccountNumberValidator()
    if validator2.validate(accountNumber) == false {
        return nil
    }

    let validator3 = sortingCodeValidator ?? SortingCodeValidator()
    if validator3.validate(sortingCode) == false {
        return nil
    }

    if (owners.count == 0 || owners.count > 2) {
        return nil
    }

    self.accountName = accountName
    self.accountNumber = accountNumber
    self.sortingCode = sortingCode
    self.accountType = accountType
    self.owners = owners
    self.transactions = [Transaction]()
}
```

```
    func setOpeningBalance(_ amount:Float) -> Void {
        if let openingBalanceTransaction =
            Transaction(txDescription: "Opening Balance",
            date: NSDate(),
            isIncoming: true,
            amount: "100.0") {
                self.transactions.removeAll()
                self.transactions.append(openingBalanceTransaction)
        }
    }

    func withdraw(_ amount:Float, _ person:AccountOwner?) -> Void {
        if let newTransaction =
            Transaction(txDescription: "ATM Withdrawal",
            date: NSDate(),
            isIncoming: false,
            amount: "\(amount)") {
                self.transactions.append(newTransaction)
        }
    }

    func deposit(_ amount:Float, _ person:AccountOwner?) -> Void {
        if let newTransaction =
            Transaction(txDescription: "Cash Deposit",
            date: NSDate(),
            isIncoming: true,
            amount: "\(amount)") {
                self.transactions.append(newTransaction)
        }
    }

}
```

You can execute Quick BDD-style tests just as you do any other test, using the Product ➤ Test menu item. After executing the tests, if you were to look at the test navigator for a test report, you would see that BDD style tests appear alongside regular unit tests, but are more human readable (Figure 10-2).

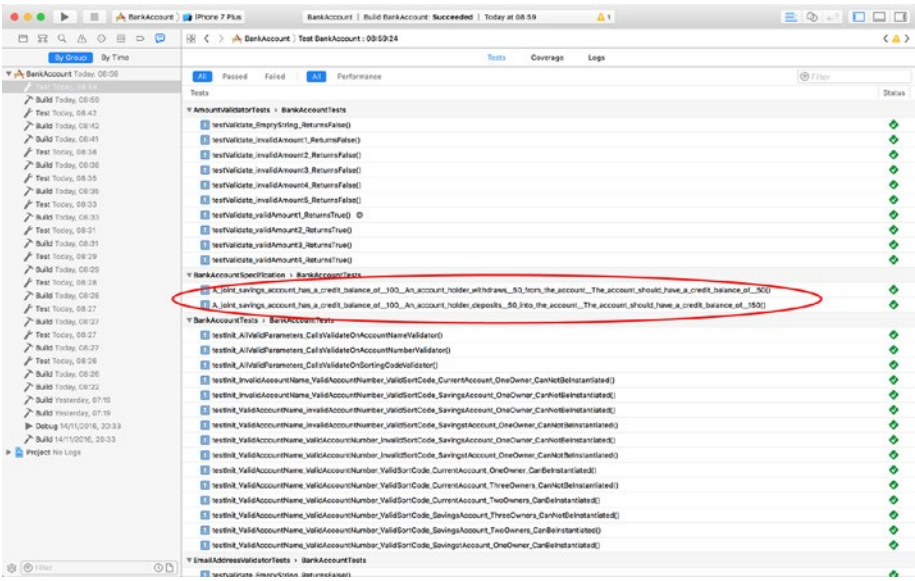

Figure 10-2. *BDD Tests Have More Verbose Names Than TDD Tests*

In the next two chapters, you will learn to integrate Quick and Nimble into a Swift project and try out a few Quick tests.

Advantages and Disadvantages of BDD

After having being introduced to behavior-driven development, you might be wondering whether BDD is a replacement for TDD. Both TDD and BDD have their own uses: test-driven development focuses on how your code is structured and operates at a lower level than BDD. Behavior-driven development helps ensure that the code you are writing fulfills business objectives.

As with any technique, behavior-driven development has its own advantages and disadvantages. Some of the advantages of BDD over TDD are the following:

- Tests are more verbose.
- Each passing test proves that the product is closer to what the customer wants.
- BDD tests are useful to business analysts and product owners as well as developers.
- BDD tests are not as fragile as TDD tests. If you change the manner in which a scenario is implemented, BDD tests are less likely to break.

Some of the disadvantages of BDD are the following:

- BDD requires product owners, testers, and business analysts to buy into the process. All too often teams start out with good intentions, but after a few weeks the business loses interest in writing specifications, and it becomes the responsibility to the developer to write the scenarios as well as the code to make those scenarios pass.
- User requirements change as the project evolves, and BDD requires that user requirements are documented in a usable format before the developers start developing. For this to work, the project has a well-defined road map of upcoming features. Having up-front, well-defined requirements before development does not necessarily mean that the project has to follow the waterfall model. BDD can be used in Agile Scrum projects, but the business will need to commit to making sure those requirements for all the stories that are picked up in a sprint are well defined before the sprint begins.
- BDD works best in a team where iterative development is practiced.
- BDD requires collaboration between the business and the development team. The business has to factor the technical constraints of the current system before creating new user stories. All too often the business analysts and product owners work in their own camps and hand over their requirements to the developers in a sprint planning session.

Summary

In this chapter you have learned about the core concepts involved in Behavior-Driven Development. Using a hypothetical example of a development team that has been contracted to build a simple banking solution, you have examined the process of business requirement analysis and user story creation.

You have also been introduced to two popular open source frameworks called Quick and Nimble. These frameworks are commonly used to develop BDD-style tests for iOS projects.