Test-Driven iOS Development with Swift

This is an easy-to-follow, example-driven tutorial, packed with lots of tips and tricks that explore Test-Driven Development (TDD) bit by bit in the process of making an iOS application.

TDD is a proven way to find software bugs early. Writing tests before your code improves the structure and maintainability of your app.

Test-Driven iOS Development with Swift will help you understand the process of TDD and how it impacts your applications written in Swift. Through practical, real-world examples, you'll start seeing how to implement TDD in context. We will begin with an overview of your TDD workflow and then deep-dive into unit testing concepts and code cycles. We will showcase the workings of functional tests, which will help you improve the user interface. Finally, you will learn about automating deployments and continuous integration to run an environment.

Who this book is written for
If debugging iOS apps is a nerve-racking task for you and you are looking for a fix, this book is for you.

What you will learn from this book
- Implement TDD in Swift application development
- Get to know the fundamentals, life cycle, and benefits of TDD
- Explore the tools and frameworks to effectively use TDD
- Develop models and controllers driven by tests
- Construct the network layer using stubs
- Use functional tests to ensure the app works as planned
- Automate and streamline the building, analysing, testing, and archiving of your iOS apps

Test-Driven iOS Development with Swift

Create fully-featured and highly functional iOS apps by writing tests first

Dr. Dominik Hauser

Who this book is written for
If debugging iOS apps is a nerve-racking task for you and you are looking for a fix, this book is for you.
In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 2 'Planning and Structuring Your Test-Driven iOS App'
- A synopsis of the book’s content
- More information on Test-Driven iOS Development with Swift
About the Author

**Dr. Dominik Hauser** completed his PhD in physics at Heidelberg University, Germany. While working as a university professor, he started iOS development in his spare time. His first app on physics has been an astounding success worldwide. Since then, he’s turned himself into a full-time iOS developer, crediting a number of successful apps to his name. He has been a Swift developer since day one and runs a blog on iOS development at [http://swiftandpainless.com/](http://swiftandpainless.com/).
Preface

iOS projects have become bigger and more complex. Many projects have already surpassed desktop applications in their complexity. One important strategy to manage this complexity is through the use of unit tests. By writing tests, a developer can point out the intention of the code and provide a safety net against the introduction of bugs.

By writing the tests first (Test-Driven Development), the developer focuses on the problem. This way, they are forced to think about the domain and rephrase a feature request using their own understanding by writing the test. In addition to this, applications written using Test-Driven Development (TDD) only contain code that is needed to solve the problem.

As a result, the code is clearer, and the developer gains more confidence that the code actually works.

In this book, you will develop an entire iOS app using TDD. You will experience different strategies of writing tests for models, View Controller, and networking code.

What this book covers

Chapter 1, Your First Unit Tests, walks you through your first unit tests using Xcode and discusses the benefits of using TDD.

Chapter 2, Planning and Structuring Your Test-Driven iOS App, introduces the app you are going to write through the course of this book and how to set up a project in Xcode.

Chapter 3, A Test-Driven Data Model, discusses the TDD of a data model.

Chapter 4, A Test-Driven View Controller, shows you how to write tests for View Controller, and describes how to use fake objects to isolate micro features for the test.
Preface

Chapter 5, *Testing Network Code*, teaches you to test network code using stubs to fake a server component before it is developed.

Chapter 6, *Putting It All Together*, walks you through the integration of all the different parts developed in previous chapters and shows the use of functional tests.

Chapter 7, *Code Coverage and Continuous Integration*, shows you how to measure the code coverage of your tests using Xcode and introduces you to continuous integration.

Chapter 8, *Where to Go from Here*, wraps up and shows you the possible next steps to improve your acquired testing skills.
In the previous chapter, we learned how to write unit tests, and we saw an easy example of TDD. When starting TDD, writing unit tests is easy for most people. The hard part is to transfer the knowledge from *writing the test* to *driving the development*. What can be assumed? What should be done before one writes the first test? What should be tested to end up with a complete app?

As a developer, you are used to thinking in terms of code. When you see a feature on the requirement list for an app, your brain already starts to layout the code for this feature. And for recurring problems in iOS development (such as building table views), you most probably have already developed your own best practices.

In TDD, you should not think about the code while working on the test. The tests have to describe what the unit under test should do and not how it should do it. It should be possible to change the implementation without breaking the tests.

To practice this approach of development, we will develop a simple to-do list app in the remainder of this book. It is, on purpose, not a very sophisticated app. We want to concentrate on the TDD workflow, not complex implementations.

This chapter introduces the app we are going to build, and it shows the views that the finished app will have.
We will cover the following topics in this chapter:

- Task list view
- Task detail view
- Task input view
- Structure of an app
- Getting started with Xcode
- Setting useful Xcode behaviors for testing

**Task list view**

When starting the app, the user sees a list of to-do items. The items in the list consist of a title, an optional location, and the due date. New items can be added to the list by an add (➕) button, which is shown in the navigation bar of the view. The task list view will look like this:

<table>
<thead>
<tr>
<th>Carrier</th>
<th>11:04 AM</th>
<th>ToDo</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>🗑</td>
</tr>
<tr>
<td>Write book about TDD</td>
<td>01/01/2016</td>
<td>➔</td>
</tr>
<tr>
<td>Shopping</td>
<td>01/03/2016</td>
<td>➔</td>
</tr>
<tr>
<td>Laundry</td>
<td>01/11/2016</td>
<td>➔</td>
</tr>
<tr>
<td>Call Mitch</td>
<td></td>
<td>➔</td>
</tr>
<tr>
<td>Talk to boss</td>
<td>02/01/2016</td>
<td>➔</td>
</tr>
<tr>
<td>Write about UIAlertController m...</td>
<td></td>
<td>➔</td>
</tr>
</tbody>
</table>
User stories:

- As a user, I want to see the list of to-do items when I open the app
- As a user, I want to add to-do items to the list

In a to-do list app, the user will obviously need to be able to check items when they are finished. The checked items are shown below the unchecked items, and it is possible to uncheck them again. The app uses the delete button in the UI of **UITableView** to check and uncheck items. Checked items will be put at the end of the list in a section with the **Finished** header. The user can also delete all the items from the list by tapping the trash button. The UI for the to-do item list will look like this:

```
| Shopping  | 01/03/2016 |
| Laundry   | 01/11/2016 |
| Talk to boss Office | 02/01/2016 |
| Write about UIAlertController m... | |
| Finished  |
| Write book about TDD |
```

User stories:

- As a user, I want to check a to-do item to mark it as finished
- As a user, I want to see all the checked items below the unchecked items
- As a user, I want to uncheck a to-do item
- As a user, I want to delete all the to-do items

When the user taps an entry, the details of this entry is shown in the task detail view.
Task detail view
The tasks detail view shows all the information that’s stored for a to-do item. The information consists of a title, due date, location (name and address), and a description. If an address is given, a map with an address is shown. The detail view also allows checking the item as finished. The detail view looks like this:

User stories:

- As a user, given that I have tapped a to-do item in the list, I want to see its details
- As a user, I want to check a to-do item from its details view

Task input view
When the user selects the add (+) button in the list view, the task input view is shown. The user can add information for the task. Only the title is required. The Save button can only be selected when a title is given. It is not possible to add a task that is already in the list. The Cancel button dismisses the view. The task input view will look like this:
User stories:

- As a user, given that I have tapped the add (+) button in the item list, I want to see a form to put in the details (title, optional date, optional location name, optional address, and optional description) of a to-do item
- As a user, I want to add a to-do item to the list of to-do items by tapping on the **Save** button

We will not implement the editing and deletion of tasks. But when you have worked through this book completely, it will be easy for you to add this feature yourself by writing the tests first.

Keep in mind that we will not test the look and design of the app. Unit tests cannot figure out if an app looks like it was intended. Unit tests can test features, and these are independent of their presentation. In principle, it would be possible to write unit tests for the position and color of UI elements. But such things are very likely to change a lot in the early stages of development. We do not want to have failing tests only because a button has moved 10 points.

However, we will test whether the UI elements are present on the view. If your user cannot see the information for the tasks, or if it is not possible to add all the information of a task, then the app does not meet the requirements.
Structure of the app
The following diagram shows the structure of the app:

The Table View Controller, the delegate and the data source
In iOS apps, data is often presented using a table view. Table views are highly optimized for performance; they are easy to use and to implement. We will use a table view for the list of to-do items.

A table view is usually represented by UITableViewViewController, which is also the data source and delegate for the table view. This often leads to a massive table View Controller because it is doing too much: presenting the view, navigating to other view controllers, and managing the presentation of the data in the table view.
It is a good practice to split up the responsibility into several classes. Therefore, we will use a helper class to act as the data source and delegate for the table view. The communication between the Table View Controller and the helper class will be defined using a protocol. Protocols define what the interface of a class looks like. This has a great benefit: if we need to replace an implementation with a better version (maybe because we have learned how to implement the feature in a better way), we only need to develop against the clear interface. The inner workings of the other classes do not matter.

**Table view cells**
As you can see in the preceding screenshots, the to-do list items have a title and, optionally, they can have a due date and a location name. The table view cells should only show the set data. We will accomplish this by implementing our own custom table view cell.

**A model**
The model of the application consists of the to-do item, the location, and an item manager, which allows the addition and removal of items and is also responsible for managing the items. Therefore, the controller will ask the item manager for the items to present. The item manager will also be responsible for storing the items on disc.

Beginners often tend to manage the model objects within the controller. Then, the controller has a reference to a collection of items, and the addition and removal of items is directly done by the controller. This is not recommended because if we decide to change the storage of the items (for example, by using Core Data), their addition and removal would have to be changed within the controller. It is difficult to keep an overview of such a class, and because of this reason, it is a source of bugs.

It is much easier to have a clear interface between the controller and the model objects because if we need to change how the model objects are managed, the controller can stay the same. We could even replace the complete model layer if we just keep the interface the same. Later in the chapter, we will see that this decoupling also helps to make testing easier.
Other view controllers
The application will have two more view controllers, a task detail View Controller, and a View Controller for the input of the task.

When the user taps a to-do item in the list, the details of the item are presented in the task detail View Controller. From the Details screen, the user will be able to check an item.

New to-do items will be added to the list of items using the view presented by the input View Controller.

Development strategy
In this book, we will build the app from inside out. We will start with the model, and then build the controllers and networking. At the end of the book, we will put everything together.

Of course, this is not the only way to build apps. But by separating on the basis of layers instead of features, it is easier to follow and keep an overview of what is happening. When you later need to refresh your memory, the relevant information you need is easier to find.

Getting started with Xcode
Now, let's start our journey by creating a project that we will implement using TDD.

Open Xcode and create a new iOS project using the Single View Application template. In the options window, add Todo as the product name, select Swift as language, choose iPhone in the Devices option, and check the box next to Include Unit Tests. Let the Use Core Data and Include UI Tests boxes stay unchecked.

Xcode creates a small iOS project with two targets: one for the implementation code and the other for the unit tests. The template contains code that presents a single view on screen. We could have chosen to start with the master-detail application template because the app will show a master and a detail view. However, we have chosen the Single View Application template because it comes with hardly any code, and in TDD, we want to have all the implementation code demanded by failing tests.
To take a look at how the application target and test target fit together, select the project in Project Navigator, and then select the ToDoTests target. In the General tab, you'll find a setting for the host application that the test target should be able to test. It should look like this:

Xcode has already set up the test target correctly to allow the testing of the implementations that we will write in the application target.

Xcode has also set up a scheme to build the app and run the tests. Click on the Scheme selector next to the stop button in the toolbar, and select Edit Scheme... In the Test action, all the test bundles of the project will be listed. In our case, only one test bundle is shown: ToDoTests. On the right-hand side of the shown window is a column named Test, with a checked checkbox. This means that if we run the tests while this scheme is selected in Xcode, all the tests in the selected test suite will be run.

### Setting useful Xcode behaviors for testing

Xcode has a feature called behaviors. With the use of behaviors and tabs, Xcode can show useful information depending on its state.

Open the Behaviors window by going to Xcode | Behaviors | Edit Behaviors. On the left-hand side are the different stages for which you can add behaviors (Build, Testing, Running, and so on). The following behaviors are useful when doing TDD.

The behaviors shown here are those that I find useful. Play around with the settings to find the ones most useful for you. Overall, I recommend using behaviors because I think they speed up development.
Useful build behaviors

When building starts, Xcode compiles the files and links them together. To see what is going on, you can activate the build log when the building starts. It is recommended that you open the build log in a new tab because this allows us to switch back to the code editor when no error occurs during the build. Select the Starts stage and check Show tab named. Put in the name Log and select in active window. Check the Show navigator setting and select Issue Navigator. At the bottom of the window, check Navigate to and select current log. After you have made these changes, the settings window should look like this:

Build and run to see what the behavior looks like.
Testing behaviors

To write code, I have an Xcode tab called Coding. Usually, in this tab, the test is open on the left-hand side, and in the Assistant Editor on the right-hand side is the code to be tested (or in the case of TDD, the code to be written). It looks like this:

![Xcode tab](image)

When the test starts, we want to see the code editor again. So, we add a behavior to show the Coding tab. In addition to this, we want to see the Test Navigator and debugger with the console view.

When the test succeeds, Xcode should show a bezel to notify us that all tests have passed. Go to the Testing | Succeeds stage, and check the Notify using bezel or system notification setting. In addition to this, it should hide the navigator and the debugger because we want to concentrate on refactoring or writing the next test.
In case testing fails (which happens a lot in TDD), Xcode should show a bezel again. I like to hide the debugger because usually it is not the best place to figure out what is going on in the case of a failing test. And in TDD, in most cases, we already know what the problem is. But we want to see the failing test. Therefore, check **Show navigator** and select **Issue navigator**. At the bottom of the window, check **Navigate to** and select **first new issue**.

You can even make your Mac speak the announcements. Check **Speak announcements using** and select the voice you like. But be careful not to annoy your coworkers. You might need their help in the future.

Now, the project and Xcode are set up, and we can start our TDD journey.

**Summary**

In this chapter, we took a look at the app we are going to build throughout the course of this book. We took a look at how the screens of the app will look when we are finished. We created the project that we will use later on, and we learned about Xcode behaviors.

In the next chapter, we will develop the data model of the app using TDD. We will use structs for the model wherever we can because models are best represented in Swift by value types. We will add some conformance to the Equatable protocol to make the comparison of the model instances easier.
Where to buy this book

You can buy Test-Driven iOS Development with Swift from the Packt Publishing website.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.