Unit Testing versus TDD

So far we've been focused on unit testing our Swift apps. But unit testing is not necessarily TDD. Test Driven Development involves writing the unit test before writing the code, whereas unit tests don't mandate when you write tests. Without TDD, more often than not unit tests are written at the end of a coding cycle to improve code coverage metrics. So you can do unit testing with TDD, but you can't do TDD without unit testing. Once you start TDD, you should find that it is less painful than classic unit testing.

Value of TDD

We know that unit testing and testing in general helps catch mistakes, but why would we want to use TDD? There are several fundamental reasons for this. TDD pushes the developer to only implement what is minimally needed to implement a feature, so it can help us shape our design to actual or real use and avoids any gold plating in our implementation. We call this process YAGNI, or You Ain't Going to Need It. It leads to much simpler implementations as the focus is on what is required, not necessarily what you might be able to do so saving money and reducing complexity.

In these days of faster mobile startups, YAGNI also encourages getting a minimum viable product or MVP out the door as quickly as possible. The business owners choose the bare minimum of features needed to launch an app in the app store. This minimum feature list is then split into manageable chunks that feed your developers' TDD process.

Unit testing without TDD can get you a regression test suite that will help you from introducing any defects as your code. But because you're writing unit tests before you write any code, your understanding of what the code is trying to do is naturally going to be much fresher than writing unit tests weeks or even months later. The TDD regression test suite is probably going to have more coverage and be much more comprehensive than unit testing without TDD.

Also because of the ongoing refactoring, the code becomes more maintainable and much leaner, leading to a longer life for your codebase. It is very easy to write horrible, untestable code in Swift. Refactoring will encourage you to write single-line methods that are easily tested rather than monolithic View Controllers.

Finally, the process of coding in this continuous red/green/refactor cycle helps kill procrastination, as the focus is on small discrete steps and the app gradually emerges from the bottom up as one feature after another is implemented.

Writing an App Using TDD

Before we get started, we're going to need some basic requirements for our horoscope app.

- Display each star sign
- Display information about each star sign
- Display a daily horoscope for star sign

There are lots of other things that we could add, but we're practicing YAGNI so we're going to go with the minimum of features for our MVP horoscope app.

We're going to create a simple horoscope app that displays the list of Zodiac signs, displays some information about each star sign, and then shows the daily horoscope.

Feature 1

The initial feature requires that we display a list of star signs, which we will do using a Table View, as shown in Figure 6-1.

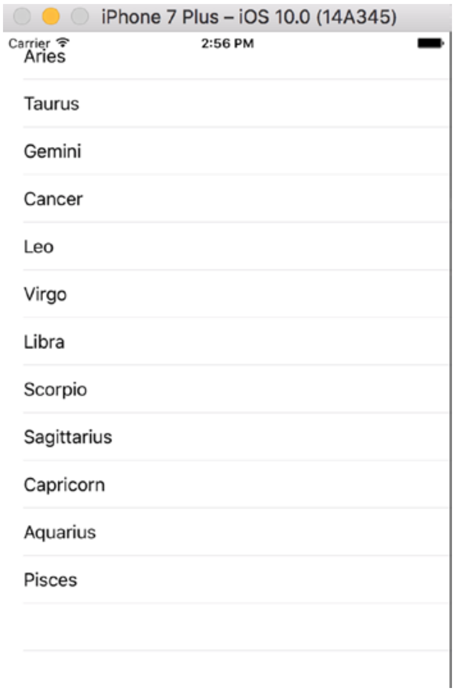


Figure 6-1. *Horoscope app feature 1*

Getting Started

For this feature, we need to create a new project. Close any other Xcode projects that you have open. Because of the nature of the mobile apps we not only need to create a Swift file with our horoscope information, but we also need to set up the interface to display the signs.

Note that while this will end up being a Master-Detail application, the TDD process tells us that we only need the minimum amount of information to create our feature. A single view application meets our immediate needs (see Figure 6-2).

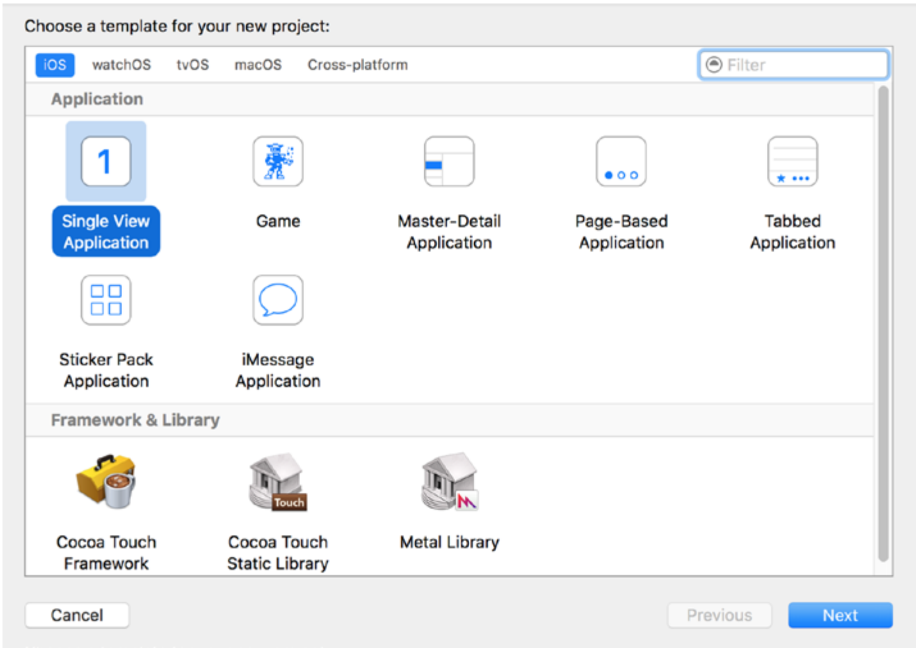
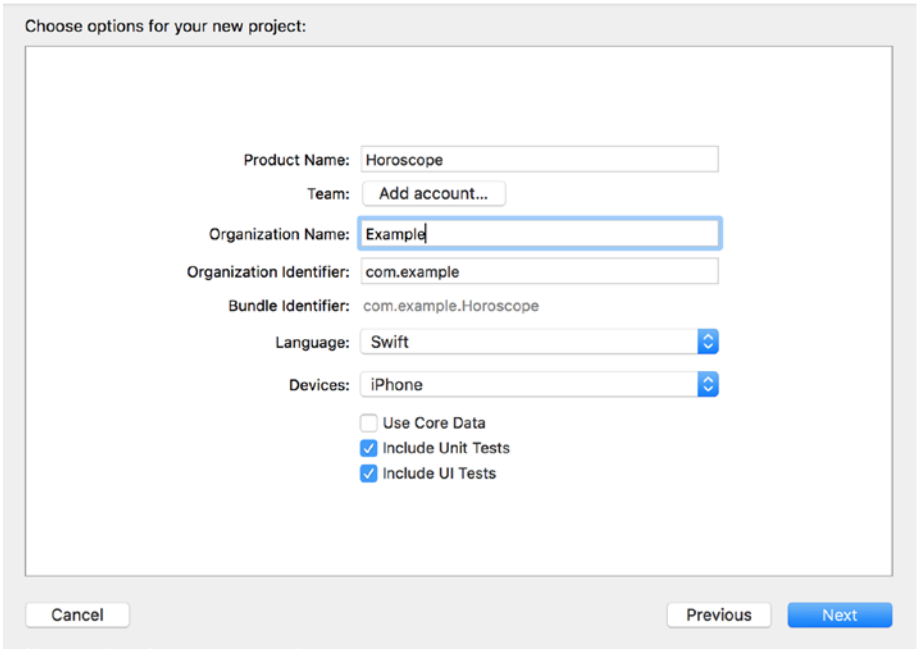


Figure 6-2. *Single view iOS application*

1. In Xcode, go to File ➤ New ➤ Project.
2. User iOS and Application, choose the Single View Application template, as shown in Figure 6-2.
3. Click next and enter the following options (see Figure 6-3):
   - Product Name: Horoscope
   - Organization Name: Example
   - Organization Identifier: com.example
   - Language: Swift
   - Devices: iPhone
   - Include Unit Tests: Checked
4. Click Next.
5. Navigate to Desktop ➤ Agile Swift ➤ Chapter 6.
6. Check Create Git Repository on My Mac.
7. Click Create.


Choose options for your new project:

Product Name: Horoscope
Team: Add account...
Organization Name: Example
Organization Identifier: com.example
Bundle Identifier: com.example.Horoscope
Language: Swift
Devices: iPhone
☐ Use Core Data
☑ Include Unit Tests
☑ Include UI Tests

Figure 6-3. *Use these project options*

Writing the Test First

Open the `HoroscopesTests.Swift` file in the `HoroscopesTest` folder. It will be the same template we've seen in the past, with `setUp`, `tearDown`, `testExample`, and `testPerformanceExample` methods. Our first feature is to display the horoscope star signs, not the first page. Create the unit tests in Listing 6-1 to start the process.

Listing 6-1. Feature 1 Unit Tests

```
let horoscopeModel = HoroscopeData.horoscopes

func testNumHoroscopeSigns() {
        XCTAssertEqual(horoscopeModel.count, 12)
}

func testFirstHoroscopeSignAries() {
    XCTAssertEqual(horoscopeModel[0].name, "Aries")
}
```

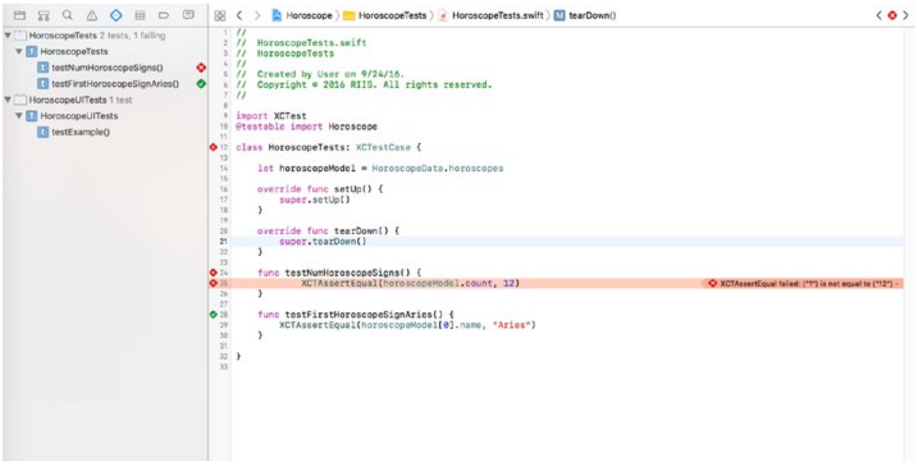
A couple of things to note. Because we haven't created the horoscope data this isn't going to compile. So we're going to need to create a `struct` or a class to store the horoscope data. Secondly we're going to need to create the user interface too. Technically we don't need to do that to make the test pass but then we won't be displaying the signs. So I always recommend creating the user interface along with your tests. This seems strange in the world of TDD, but there really isn't any way around it for mobile apps.

Listing 6-2 shows the code in `HoroscopeData.swift`. We can use a `struct`, as the data isn't going to change and nobody is going to be adding, editing, or deleting any of the elements of our horoscope data once we have it then way we want.

Listing 6-2. Horoscope Data

```
import Foundation

struct Horoscope {
    var name: String
}

struct HoroscopeData {

    static let horoscopes = [
        Horoscope(name: "Aries")
    ]

}
```

We've done enough for the tests to compile and run. Run the tests by clicking on the test icon ✓ in the `HoroscopeTests` class. The `testNumHoroscopeSigns` should fail, as shown in Figure 6-4.



```
//
//  HoroscopeTests.swift
//  HoroscopeTests
//
//  Created by User on 9/24/16.
//  Copyright © 2016 RIIS. All rights reserved.
//

import XCTest
@testable import Horoscope

class HoroscopeTests: XCTestCase {

    let horoscopeModel = HoroscopeData.horoscopes

    override func setUp() {
        super.setUp()
    }

    override func tearDown() {
        super.tearDown()
    }

    func testNumHoroscopeSigns() {
            XCTAssertEqual(horoscopeModel.count, 12)
    }

    func testFirstHoroscopeSignAries() {
        XCTAssertEqual(horoscopeModel[0].name, "Aries")
    }

}
```

```
XCTAssertEqual failed: ("1") is not equal to ("12") -
```

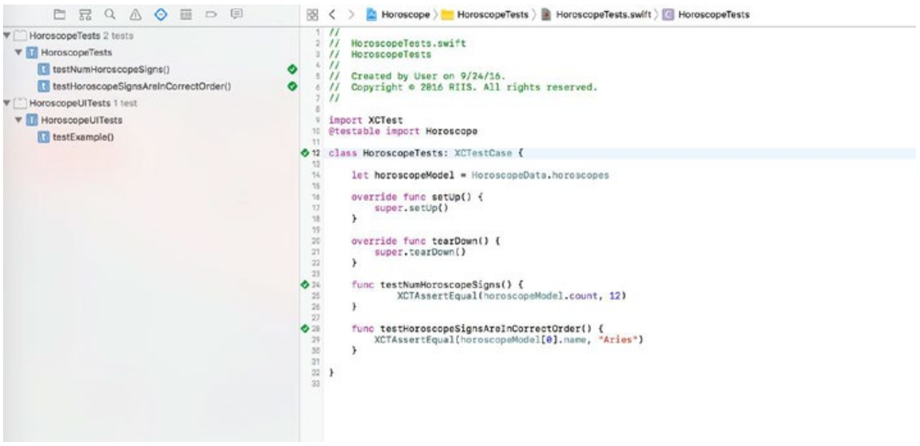
Figure 6-4. *Failing tests feature 1*

To get the tests to pass, add the rest of the horoscope names to the `HoroscopeData.swift` file, see Listing 6-3.

Listing 6-3. Complete List of Horoscope Signs

```
import Foundation

struct Horoscope {
    var name: String
}

struct HoroscopeData {

    static let horoscopes = [
        Horoscope(name: "Aries"),
        Horoscope(name: "Taurus"),
        Horoscope(name: "Gemini"),
        Horoscope(name: "Cancer"),
        Horoscope(name: "Leo"),
        Horoscope(name: "Virgo"),
        Horoscope(name: "Libra"),
        Horoscope(name: "Scorpio"),
        Horoscope(name: "Sagittarius"),
        Horoscope(name: "Capricorn"),
        Horoscope(name: "Aquarius"),
        Horoscope(name: "Pisces")
    ]

}
```

Run the tests again and this time they should pass, as shown in Figure 6-5.



```
//
//  HoroscopeTests.swift
//  HoroscopeTests
//
//  Created by User on 9/24/16.
//  Copyright © 2016 RIIS. All rights reserved.
//

import XCTest
@testable import Horoscope

class HoroscopeTests: XCTestCase {

    let horoscopeModel = HoroscopeData.horoscopes

    override func setUp() {
        super.setUp()
    }

    override func tearDown() {
        super.tearDown()
    }

    func testNumHoroscopeSigns() {
            XCTAssertEqual(horoscopeModel.count, 12)
    }

    func testHoroscopeSignsAreInCorrectOrder() {
        XCTAssertEqual(horoscopeModel[0].name, "Aries")
    }

}
```

Figure 6-5. *Passing tests feature 1*

We've created the data for our app, but it's not much use without the user interface. So we're going to create that next.

Creating a Table View App

1. In the Project Navigator, notice that there are a number of files created automatically. Xcode created a View Controller as well as an `AppDelegate.swift` file when we chose the single view application.
2. Click on `Main.storyboard`.
3. Click on the View Controller in the Document Outline. If it's not visible, then click the Show Document Outline button at the bottom left of the Editor area.
4. Click on the View Controller and delete it.
5. Delete `ViewController.swift`.
6. Go to the Object Library at the bottom right of the Xcode screen.
7. In the search window, search for the Table View Controller in the object window.
8. Drag and drop a Table View Controller onto the Editor.
9. Click on the Table View Controller in the Document Outline window.
10. Next, click on the Attributes Inspector in the Inspector area.
11. Check on the Is Initial View Controller to make this the Storyboard Entry point when the application is started.

Adding a Label

1. In the Document Outline, expand Table View Controller.
2. Click on Table View.
3. In the Utilities area on the right, go to the Attributes Inspector.
4. Under Table View, from the Content menu, choose Dynamic Prototype.
5. In the Document Outline, expand the Table View.
6. Expand the Table View Section.

7. Click the remaining table view cell to select it.
8. In the Attributes Inspector, set the Style menu to custom.
9. Enter signCell in the Identifier just below the Style
10. In the Document Outline, expand Table View Cell and click on Content View.
11. Go to the Object Library at the bottom right of the Xcode screen.
12. In the search window, search for the Label in the object window.
13. Drag and drop the Label on the Table View Cell in the Storyboard, see Figure 6-6.

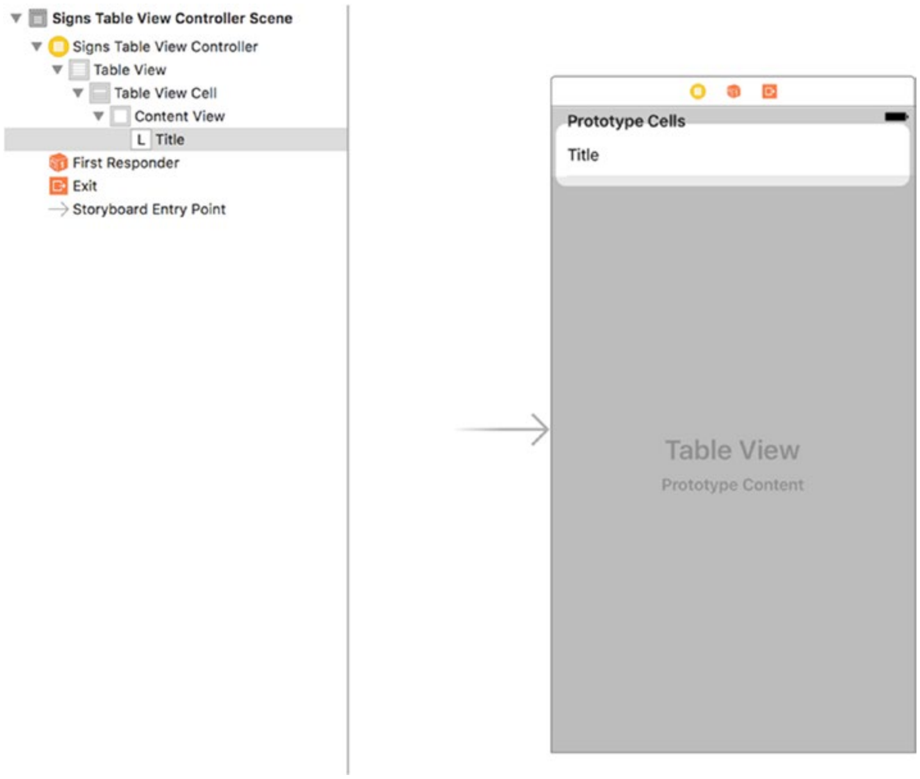
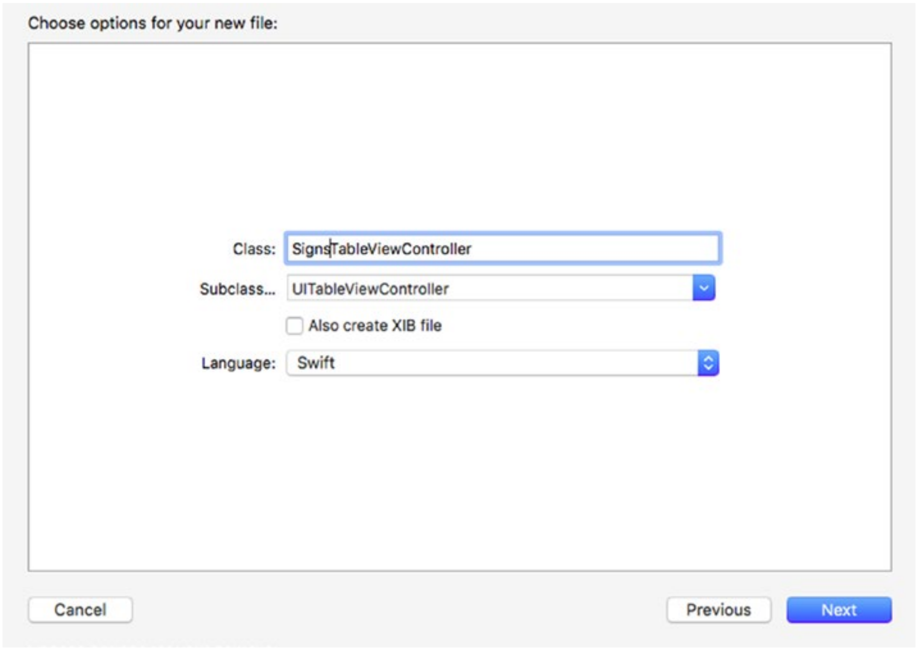


Figure 6-6. *Adding a Label to the Signs Table's Table View Cell*

Creating a Table View Class

1. Go to File ➤ New ➤ File.
2. On the left, under iOS, make sure Source is selected.
3. Choose Cocoa Touch Class.
4. From the Subclass of menu, choose UITableViewController.
5. Edit the name of the Class to be `SignsTableViewController` (see Figure 6-7).
6. Click Next.
7. You should already be in the `Horoscope` folder, so click Create.
8. Notice that `SignsTableViewController.swift` has been added to the Project Navigator.


Choose options for your new file:

Class: SignsTableViewController

Subclass...: UITableViewController

[ ] Also create XIB file

Language: Swift

Figure 6-7. *Create Table View Controller SignsTableViewController*

By making our class a subclass of the `UITableViewController`, we will have a `SignsTableViewController.swift` class with template methods for all the functionality we need for our table object.

Connecting the Table Class

1. In the Project Navigator, click on `Main.storyboard`.
2. In the Document Outline, click Table View.
3. In the Utilities area on the right, click on the Connections inspector tab.
4. Notice there are two outlets that are assigned to the Table View: dataSource and delegate.
5. Next we need to connect the view we have in the Storyboard to our new class. In the Document Outline, click Table View Controller.
6. In the Utilities area on the right, click the Identity Inspector tab.
7. Next to Class, type `SignsTableViewController` or choose it from the dropdown.
8. Click Return to apply the change.
9. In the Document Outline, click Table View.
10. In the Utilities area on the right, click the Connections Inspector tab again.
11. Notice that dataSource and delegate are now connected to the Signs Table View Controller (see Figure 6-8).

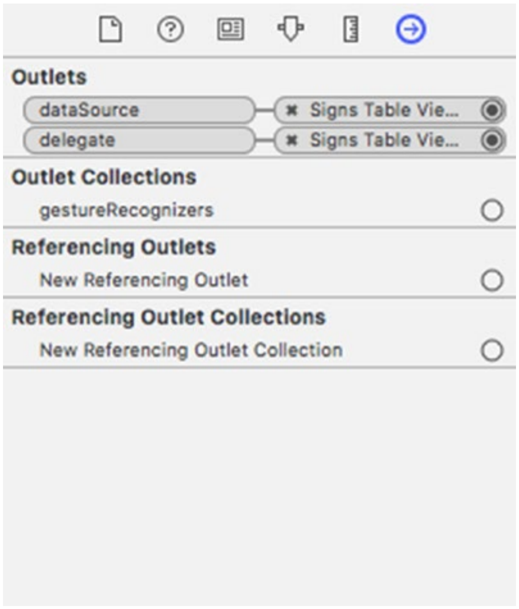


Figure 6-8. *dataSource and delegate outlets*

Now we can start adding the code we need to populate the table cells.
We need to make the following changes to `SignsTableViewController.swift`:

- Add a reference to the horoscope data by adding `let horoscopeModel = HoroscopeData.horoscopes`
- Edit the `numberOfSectionsInTableView`, set to `return 1` as there is only one section table
- Edit the `numberOfRowsInSectionsInTableView` to `return horoscopeModel.count` i.e. 12 signs
- Configure the cell to display the horoscope data
- Change

```
       let cell = tableView.dequeueReusableCell(withIdentifier:
"reuseIdentifier", for: indexPath)
```

to

```
let cell = tableView.dequeueReusableCell(withIdentifier: "signCell", for:
indexPath)
```

And add the cell name so it can be displayed.

```
let horoscopeDetail = horoscopeModel[indexPath.row]
cell.textLabel?.text = horoscopeDetail.name
```

The cleaned up file is shown in Listing 6-4.

Listing 6-4. Table View Code

```
import UIKit

class SignsTableViewController: UITableViewController {

    let horoscopeModel = HoroscopeData.horoscopes

    override func viewDidLoad() {
        super.viewDidLoad()
    }

    override func didReceiveMemoryWarning() {
        super.didReceiveMemoryWarning()
    }

    override func numberOfSections(in tableView: UITableView) -> Int {
        return 1
    }

    override func tableView(_ tableView: UITableView, numberOfRowsInSection
    section: Int) -> Int {
        return horoscopeModel.count
    }

    override func tableView(_ tableView: UITableView, cellForRowAt
    indexPath: IndexPath) -> UITableViewCell {
        let cell = tableView.dequeueReusableCell(withIdentifier: "signCell",
        for: indexPath)

        // Configure the cell...
        let horoscopeDetail = horoscopeModel[indexPath.row]
        cell.textLabel?.text = horoscopeDetail.name
        return cell
    }
    /*
    // MARK: - Navigation
    // In a storyboard-based application, you will often want to do a little
       preparation before navigation override func prepare(for segue:
       UIStoryboardSegue, sender: Any?) {
        // Get the new view controller using segue.
           destinationViewController.
        // Pass the selected object to the new view controller.
    }
    */

}
```

We need to leave the template prepare function in our code as we're going to need it in the next feature. Run the app and the table displays with the 12 signs.

Refactor

The final stage of testing is to refactor. Begin by deleting the `ViewController.swift` file. Right click and choose delete and then choose 'Move to Trash.' The remaining code is so simple we don't need to do much refactoring with the code. We should, however, add more tests to make sure all the remaining signs are being displayed, as shown in Listing 6-5.

Listing 6-5. Adding more Tests

```
func testHoroscopeSignsAreInCorrectOrder() {
    XCTAssertEqual(horoscopeModel[0].name, "Aries")
    XCTAssertEqual(horoscopeModel[1].name, "Taurus")
    XCTAssertEqual(horoscopeModel[2].name, "Gemini")
    XCTAssertEqual(horoscopeModel[3].name, "Cancer")
    XCTAssertEqual(horoscopeModel[4].name, "Leo")
    XCTAssertEqual(horoscopeModel[5].name, "Virgo")
    XCTAssertEqual(horoscopeModel[6].name, "Libra")
    XCTAssertEqual(horoscopeModel[7].name, "Scorpio")
    XCTAssertEqual(horoscopeModel[8].name, "Sagittarius")
    XCTAssertEqual(horoscopeModel[9].name, "Capricorn")
    XCTAssertEqual(horoscopeModel[10].name, "Aquarius")
    XCTAssertEqual(horoscopeModel[11].name, "Pisces")

}
```

Feature 2

In feature 2, we want to display information about each sign. Let's keep it basic so we can choose to display the following information:

- Name
- Description
- Symbol
- Month

We'll display the information on a second view when the user clicks on the relevant table cell.

We could store the information in a SQLite database, but that's not a requirement, so we'll take the YAGNI route and instead store the sign information in our `struct`. It's neat, clean, and meets the requirement.

Writing the Test

Start with the tests. We can quickly test for the Description, Symbols, and Month in the `HoroscopeTests.swift` file, as shown in Listing 6-6.

Listing 6-6. Testing Description, Symbol, and Month

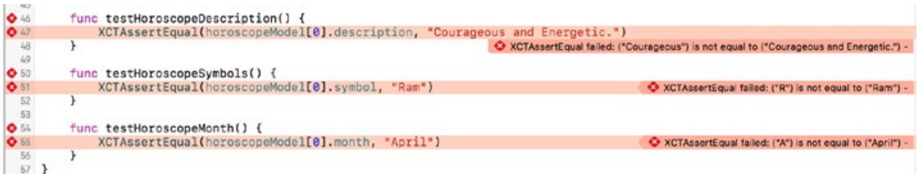
```
func testHoroscopeDescription() {
    XCTAssertEqual(horoscopeModel[0].description, "Courageous and
    Energetic.")
}

func testHoroscopeSymbols() {
    XCTAssertEqual(horoscopeModel[0].symbol, "Ram")
}

func testHoroscopeMonth() {
    XCTAssertEqual(horoscopeModel[0].month, "April")
}
```

To get the tests to run, we need to add the description, symbol, and month to the horoscope struct in `HoroscopeData.swift` (see Listing 6-7).

Listing 6-7. Updated Horoscope Struct

```
struct Horoscope {
    var name: String
    var description: String
    var symbol: String
    var month: String
}
```

Run the tests. As expected, seeing as we're in the red part of the red/green/refactor TDD cycle, the unit tests all fail (see Figure 6-9).



```
    func testHoroscopeDescription() {
        XCTAssertEqual(horoscopeModel[0].description, "Courageous and Energetic.")
    }                                     XCTAssertEqual failed: ("Courageous") is not equal to ("Courageous and Energetic.") -

    func testHoroscopeSymbols() {
        XCTAssertEqual(horoscopeModel[0].symbol, "Ram")      XCTAssertEqual failed: ("R") is not equal to ("Ram") -
    }

    func testHoroscopeMonth() {
        XCTAssertEqual(horoscopeModel[0].month, "April")     XCTAssertEqual failed: ("A") is not equal to ("April") -
    }
}
```

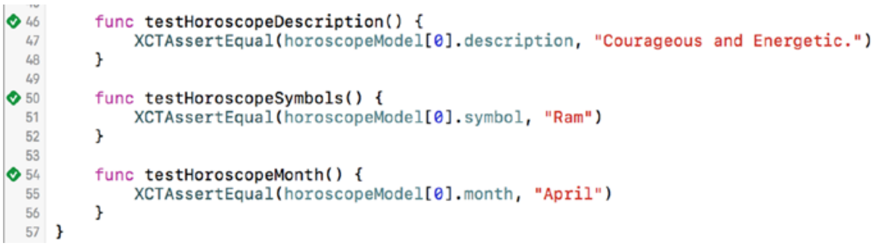
Figure 6-9. *Failing tests*

Complete the description of the Aries horoscope data shown in Listing 6-8.

Listing 6-8. Completed Horoscope Details for Aries

```
struct HoroscopeData {

    static let horoscopes = [
        Horoscope(name: "Aries",
        description: "Courageous and Energetic.",
        symbol: "Ram",
        month: "April")
    ]
}
```

Run the tests again and they turn green, as shown in Figure 6-10.



```
    func testHoroscopeDescription() {
        XCTAssertEqual(horoscopeModel[0].description, "Courageous and Energetic.")
    }

    func testHoroscopeSymbols() {
        XCTAssertEqual(horoscopeModel[0].symbol, "Ram")
    }

    func testHoroscopeMonth() {
        XCTAssertEqual(horoscopeModel[0].month, "April")
    }
}
```

Figure 6-10. *Passing tests*

Creating a User Interface

We're not done yet, as we need to display the information to the user. We're going to create a detailed view of the horoscope sign. When the user clicks on one of the signs it will take them to a detail view of the information for that sign. To add this functionality to our app, we'll first add a Navigation Controller.

Adding the Navigation Controller

1. In the Project Navigator, click on `Main.storyboard`.
2. Hide the Document Outline button by clicking the ▯ at the bottom left of the Editor area.
3. Go to the Object Library ⊙ in the Utilities area on the bottom right.
4. Find the Navigation Controller.
5. Drag it onto the Storyboard.

6. The Navigation Controller comes with two scenes:
   - Navigation Controller: Manages the relationships and transitions between our views
   - Root View Controller: The first controller instantiated by the Navigation Controller
7. Delete the provided Root View Controller so the Table View Controller can be the first controller instantiated.
8. Click onto the top bar of the Root View Controller so that it is outlined in blue and delete it.

The Editor should now look similar to what you see in Figure 6-11.

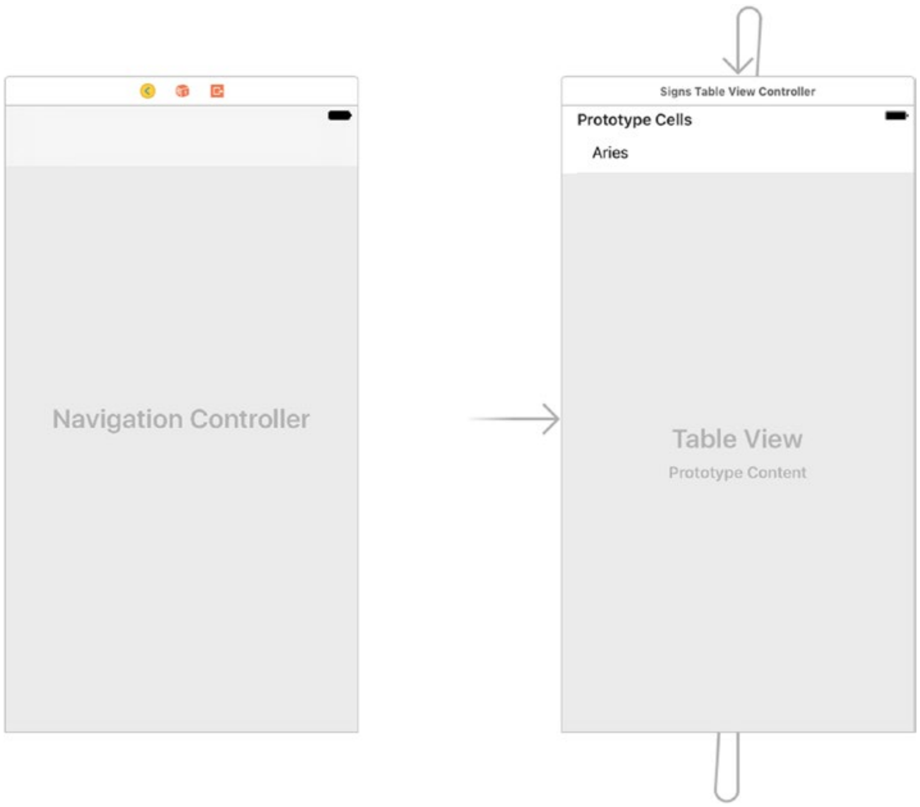


***Figure 6-11.** Adding a Navigation Controller*

Setting the Initial View Controller

We can see the gray arrow is pointing to our Signs Table View Controller. We need to change that so that the Navigation Controller is our initial View Controller.

1. Open the Document Outline button by clicking the ▯ at the bottom left of the Editor area.
2. Click on the Navigation Controller to select it.
3. In the Utilities area on the right, click on the Attributes inspector tab ⇩.
4. In the View Controller section, check on Is Initial View Controller.
5. In the Editor area, the gray arrow should now be pointing to the Navigation Controller, as shown in Figure 6-12.

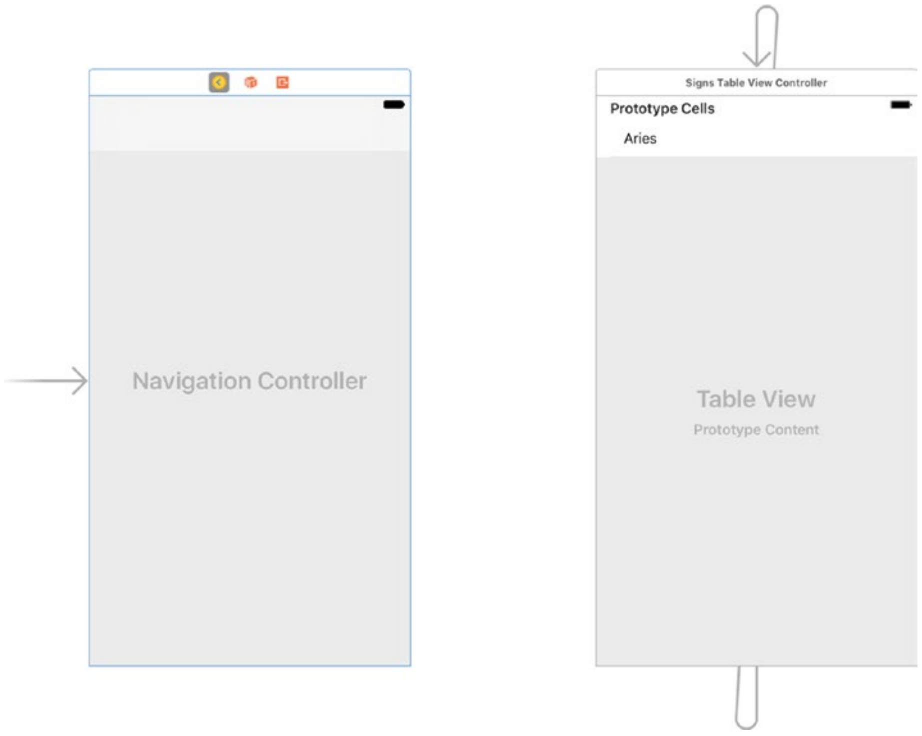


Figure 6-12. *Initial View Navigation Controller*

Setting the Root View Controller

Next we need to set the Signs Table View Controller as the root View Controller so it is the first controller that users see. We set this in the Connections Inspector.

1. Select the Navigation Controller in the Editor.
2. In the Utilities area on the right, click on the Connections Inspector tab ⊖.
3. In the Triggered Segues section in the Connections Inspector tab, click on the root View Controller.
4. Hold Ctrl and drag from the + circle beside root View Controller to the Signs Table View Controller.
5. In the Connections Inspector tab ⊖ the root View Controller is now connected to the Signs Table View Controller.
6. In the Editor section, the Signs Table View Controller is now the root View Controller for the Navigation Controller, as shown in Figure 6-13.

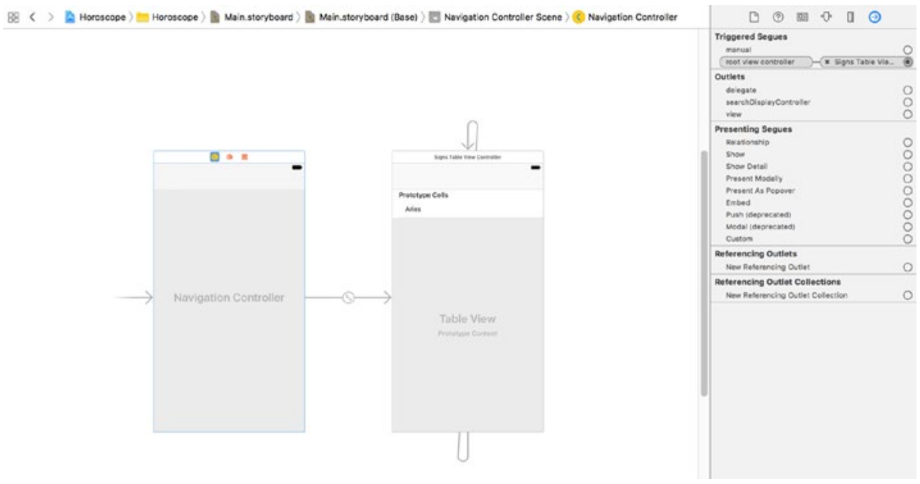


Figure 6-13. *Setting the root View Controller*

This now creates a connection between the Navigation Controller and the Table View Controller which is now the Root View Controller for the Navigation Controller.

Adding the Detail View Controller

Next we create a View Controller to display the sign information and then create a connection between it and the cell in our Signs Table View Controller. Therefore, when users tap on the cell, they are taken to the new View Controller.

1. We need to add a View Controller that will list the details about our star sign. In the Object Library, ⊡ find the View Controller.
2. Drag a View Controller and drop it to the right of Signs Table View Controller in the Editor.
3. Hold Ctrl and drag from the Label 'Aries' cell to the View Controller on the right.
4. A Segue menu will open, as shown in Figure 6-14. Under Selection Segue, click Show.

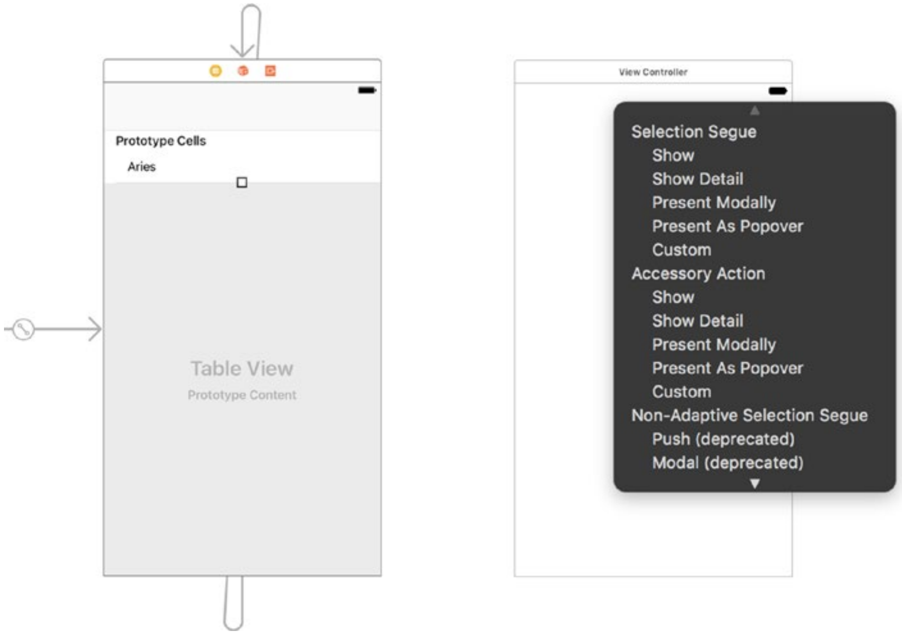


Figure 6-14. *Creating the segue between the Table View and View Controller*

5. In the Document Outline, expand Signs Table View Controller if it isn't already expanded.
6. Click on Navigation Item to select it.
7. In the Utilities area on the right, click the Attributes Inspector tab ⊖.
8. Enter Signs as the Title.
9. The Signs Table View Controller title bar has been updated.
10. Click the Run button and you should be able to click on a Sign and get a blank detail page.

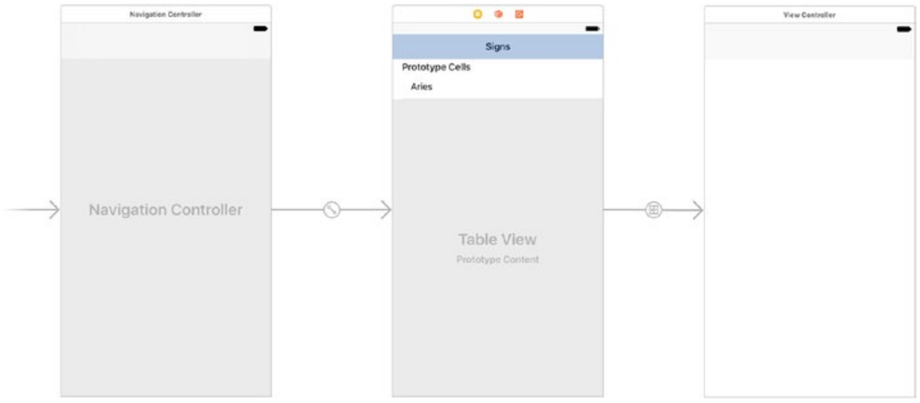


Figure 6-15. *Adding the Detail View Controller*

Adding a Name, Description, Symbol, and Month

We can quickly create the layout for the View Controller that appears when you tap on a horoscope sign. This view will list the extra information we created earlier.

1. Go to the Document Outline. If it's not open, click on ▯.
2. Under View Controller Scene in the Document Outline, click View Controller to select it.
3. In the Object Library area on the right, ⊙ search for Label.
4. Add a Label for Name, Symbol, Month, and Description, and then order them as shown in Figure 6-16.

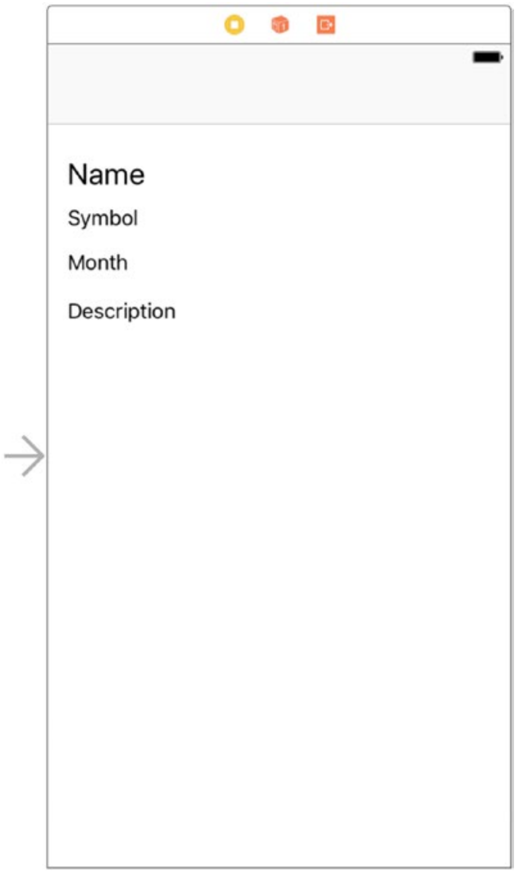


Figure 6-16. *Adding info labels to the Detail View Controller*

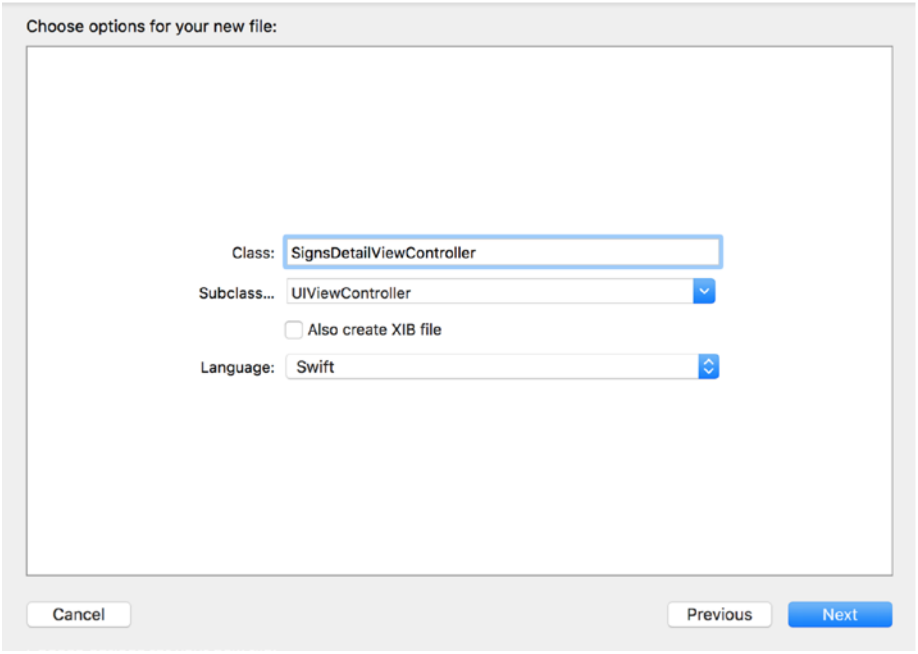
Creating SignsDetailViewController Class

1. Go to File ➤ New ➤ File in the Project Navigator.
2. Make sure on the left, under iOS, that Source is selected.
3. Double-click Cocoa Touch Class to choose it.
4. From the Subclass of menu, choose UIViewController.
5. Choose `SignsDetailViewController` as the name of the class, see Figure 6-17.
6. Click Next and Create.
7. In the Project Navigator, click on `Main.storyboard`.
8. In the Document Outline, select Signs Detail View Controller.

9. In the Utilities area on the right, click on the Identity Inspector tab ▤.

10. For the class, select SignsDetailViewController from the drop-down then press Return. SignsDetailViewController is now connected to the new class.

■ **Note** Just like with `UITableViewController`, by subclassing `UIViewController`, our new class will have template code for the view functionality we're going to need.


Choose options for your new file:

Class: SignsDetailViewController

Subclass... UIViewController

Also create XIB file ☐

Language: Swift

Figure 6-17. *Creating SignsDetailViewController.swift*

Creating a Segue

Add a variable for the current sign at the start of the detail view, as shown in Listing 6-9.

Listing 6-9. Adding the currentSignDetail Variable

```
class SignsTableViewController: UITableViewController {
    var currentSignDetail:Int?
```

Uncomment the prepare method under `// MARK: - Navigation`; see Listing 6-10.

Listing 6-10. Preparing for Segue Code

```
// MARK: - Navigation

// In a storyboard-based application, you will often want to do a little
preparation before navigation
override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
    // Get the new view controller using segue.destinationViewController.
    // Pass the selected object to the new view controller.
}
```

Segues allow us to pass data from one page to the next. In this case we're going to send the ID of the table cell that's been clicked so that we can pick this up and display the appropriate horoscope sign Details View.

Replace the code with the code in Listing 6-11.

Listing 6-11. Updated prepareForSegue

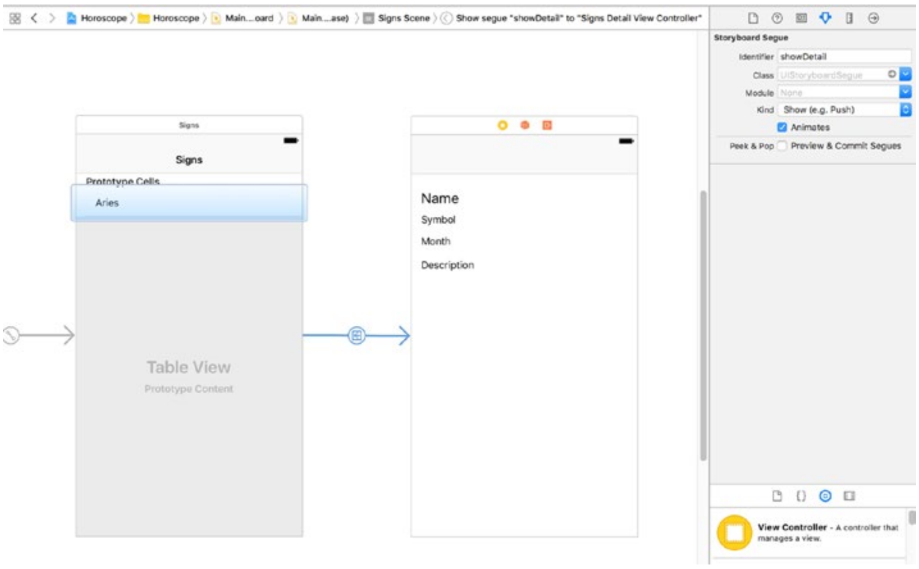
```
    override func prepare(for segue: UIStoryboardSegue, sender: Any?) {

        if (segue.identifier == "showDetail") {
            if let indexPath = tableView.indexPathForSelectedRow {
                    let signsDetailViewController:SignsDetailViewController =
                            segue.destination as! SignsDetailViewController
                let signNumber = indexPath.row
                signsDetailViewController.currentSignDetail = signNumber

            }
        }
    }
```

We're preparing for the segue by first referencing the `showDetail` segue, and then we're taking the `indexPath.row` or the ID of the clicked row and using the `signNumber` to pass the ID to the segue destination, which we'll set up next.

Creating the showDetail Segue

Let's connect the two views so `showDetail` knows where to go.

1. Click on `Main.storyboard`.
2. In the Editor, click on the segue between the Table View and Detail View Controllers (see Figure 6-18).
3. Click the Attributes Inspector tab.
4. Enter `showDetail` in the Identifier field so that the `prepare` method knows where to send the data.


Horoscope 〉 Horoscope 〉 Main...oard 〉 Main...ase) 〉 Signs Scene 〉 Show segue "showDetail" to "Signs Detail View Controller"

Storyboard Segue
Identifier: showDetail
Class: UIStoryboardSegue
Module: None
Kind: Show (e.g. Push)
[✓] Animates
Peek & Pop [ ] Preview & Commit Segues

View Controller - A controller that manages a view.

Figure 6-18. *showDetail Segue*

Connecting Sign Detail Outlets

1. Click on `Main.storyboard` in the Project Navigator.
2. Hide the Document Outline by clicking .
3. Click the top bar of the Signs Detail View Controller in the Editor area.
4. Click on the Assistant Editor button to view the Storyboard next to `SignsDetailViewController.swift`.
5. Hold Ctrl and drag the Name label to the `SignsDetailViewController.swift` file.
6. In the menu that pops up, set the following:
   - Connection: Outlet
   - Name: signName
   - Type: UILabel
   - Storage: Weak
7. Click Connect.
8. Repeat this process for `signSymbol`, `signMonth`, and `signDescription`.

Receiving the Segue Data and Updating the Detail View

Finally we need to do two things—receive the data that's been passed from the Sign Table View and update the labels with the horoscope information for that sign.

First, let's create the `currentSignDetail` variable to receive the segue data, as shown in Listing 6-12.

Listing 6-12. currentSignDetail Variable for Receiving Segue Data

```
class SignsDetailViewController: UIViewController {

    var currentSignDetail:Int?
```

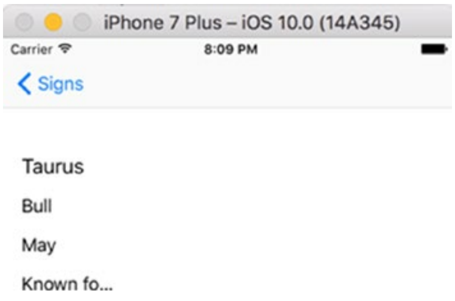
Secondly, we update the `viewDidLoad` method to change the values of the labels and display the horoscope information from our `struct`, as shown in Listing 6-13.

Listing 6-13. Update Labels with the currentSignDetailValue

```
override func viewDidLoad() {
    super.viewDidLoad()

    if let currentSignDetailValue =  currentSignDetail
    {
        signName.text = HoroscopeData.horoscopes[currentSignDetailValue].
        name
        signSymbol.text = HoroscopeData.horoscopes[currentSignDetailValue].
        symbol
        signMonth.text = HoroscopeData.horoscopes[currentSignDetailValue].
        month
        signDescription.text = HoroscopeData.horoscopes[currentSignDetailVa
        lue].description
    }
}
```

Refactor

When we run the app the Description is getting truncated, see Figure 6-19.



Taurus

Bull

May

Known fo...

Figure 6-19. *Truncated Sign Description*

We can fix this as follows.

1. Click on the Main.storyboard
2. Click on Description field on the Signs Detail View Controller
3. In the Utility Area open the Attributes Inspector
4. Set Lines = 0
5. Go to Line Break and choose Word Wrap.
6. Open the Size Inspector
7. Set X = 15, Y = 200, Width = 300 and Height = 75

Run the simulator again and the Description is no longer truncated.

You can also remove the unused template methods in the views as well as add more tests similar to what was shown in the last step. Removing the unused code will make the remaining code more obvious when you return in the future and you can find the template code again if needed.

Feature 3

Feature 3 says we should display the horoscope for each star sign. Once again, let's start with the testing. The requirement is that it must be free and available in XML or JSON (Java Script Object Notation). We can create our own simple API or use one of the many free APIs. In this case, we're going to use the daily horoscope from `http://www.findyourfate.com/rss/dailyhoroscope-feed.asp`.

The Aries output can be seen in Listing 6-14.

Listing 6-14. Horoscope XML

```
<rss version="2.0">
  <channel>
        <title>Daily Horoscope</title>
        <description>Daily Horoscopes by FindYourFate.com</description>
        <link>http://www.findyourfate.com</link>
        <item>
                <title>Aries Horoscope for Thursday, September 29, 2016</
title>
                <description>
                        You may feel that you are in a crucial place in your
                        life today, but try not to let your emotions run
                        away with you. It is easy to blame others, but you
                        know that will not be the answer. Try not to hurt
                        someone`s feelings today when they try to give you a
                        compliment. Flattery is not your thing, but speaking
                        out loud about what you are thinking may not be
                        the best way to handle this. Don`t discard those
                        invitations just yet. Such events can be fun and you
                        could finally meet that special someone.
                </description>
                <link>
                        http://horoscope.findyourfate.com/
                        ariesdailyhoroscope.html
                </link>
        </item>
  </channel>
</rss>
```

We know from our unit testing chapter that we're not going to test any network communication. It is not something we want to do during our unit testing. But we should be testing our own methods that call `findyourfate`. Listing 6-15 shows a test to see if we appended the URL correctly.

Listing 6-15. testAppendURL

```
func testAppendURL() {
    let compURL = HoroscopeService.sharedInstance.appendURL(sign: "Aries")
    XCTAssertEqual(compURL, "http://www.findyourfate.com/rss/dailyhoroscope-
    feed.asp?sign=Aries&id=45")
}
```

The `appendURL` code is shown in Listing 6-16.

Listing 6-16. AppendURL

```
func testAppendURL() {
    let compURL = HoroscopeService.sharedInstance.appendURL(sign: "Aries")
    XCTAssertEqual(compURL, "http://www.findyourfate.com/rss/dailyhoroscope-
    feed.asp?sign=Aries&id=45")
}
```

Run the code and the test passes.

Creating the User Interface

We want to display the horoscope on our Detail View so that it can take the `FindYourFate` daily horoscope and display it to our users.

Updating the Detail View

1. Click on the `Main.storyboard`.
2. Go to the Object Library ⊡ and find the Label object.

3. Drag and drop the label onto the Signs Detail View Controller and put it under the description, as shown in Figure 6-20.

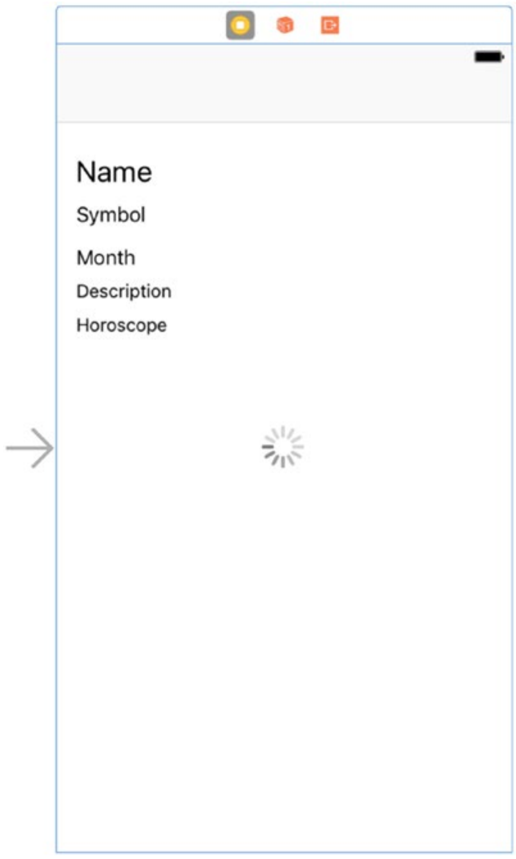


Figure 6-20. *Completed signs detail view*

Creating the Horoscope Service

1. Open File ➤ New ➤ File.
2. Create a `.swift`.
3. Cut and paste the code in Listing 6-17. `appendURL` is the code we tested and `callDailyhoroscopeApi` is template code for calling an URL asynchronously.

Listing 6-17. HoroscopeService.swift

```
import Foundation

class HoroscopeService
{
    static var horoscopeUrl:String = ""

    class var sharedInstance :HoroscopeService
    {
        struct Singleton
        {
            static let instance = HoroscopeService()
        }

        return Singleton.instance
    }

    func appendURL (sign:String) -> String {
       let baseURL = "http://www.findyourfate.com/rss/dailyhoroscope-feed.asp"
        let findYourFateID = "45"
        let completedURL = baseURL + "?sign=" + sign + "&id=" +
        findYourFateID
        return completedURL
    }

    func callDailyhoroscopeApi(sign:String,parameters: AnyObject?,
                         success: (( _ resp: Data) -> Void)?,
                         failure: (( _ error: NSError? ) -> Void)?)
    {

        let urlString = appendURL(sign: sign)
        let url = URL(string: urlString)
        let request = NSMutableURLRequest(url: url!)
        let session = URLSession.shared
        request.httpMethod = "GET"
        request.addValue("application/xml", forHTTPHeaderField: "Content-
        Type")
        request.addValue("application/xml", forHTTPHeaderField: "Accept")

        let task = session.dataTask(with: request as URLRequest,
        completionHandler:
            {
                data, response, error -> Void in
                let httpResponse = response as! HTTPURLResponse
                let strData = NSString(data: data!, encoding: String.
                Encoding.utf8.rawValue)
                print("Body: \(strData)")
                if httpResponse.statusCode == 200
                {
                    DispatchQueue.main.async(execute: { () -> Void in
                        success!(data!)
                    })
                }
                else
                {
                    DispatchQueue.main.async(execute: { () -> Void in

                        failure!(error as NSError?)
                })
            }
        })
        task.resume()
    }
}
```

Updating SignsDetailViewController.swift

There are three things we need to do to update the Horoscope field.

- Call `HoroscopeService`'s `callDailyhoroscopeApi` method to get the horoscope
- Parse the returned XML to pull out the daily horoscope field, Description
- Finally, update the Horoscope field with the daily horoscope

The complete code is shown in Listing 6-18.

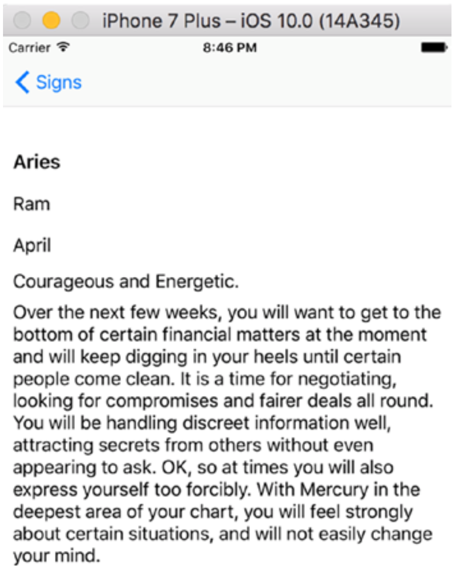
Listing 6-18. SignsDetailViewController

```
import UIKit

class SignsDetailViewController: UIViewController, XMLParserDelegate {

    var currentSignDetail:Int?

    @IBOutlet weak var signName: UILabel!
    @IBOutlet weak var signSymbol: UILabel!
    @IBOutlet weak var signMonth: UILabel!
    @IBOutlet weak var signDescription: UILabel!
    @IBOutlet weak var signHoroscope: UILabel!
    @IBOutlet weak var spinner: UIActivityIndicatorView!
    var parser = XMLParser()
    var element = NSString()

    var horoscopeDescription = String()
    var insideAnItem = false

    // MARK: - View life cycle

    override func viewDidLoad() {
        super.viewDidLoad()

        if let currentSignDetailValue =  currentSignDetail
        {
            signName.text = HoroscopeData.horoscopes[currentSignDetail
            Value].name
            signSymbol.text = HoroscopeData.horoscopes[currentSignDetail
            Value].symbol
            signMonth.text = HoroscopeData.horoscopes[currentSignDetail
            Value].month
            signDescription.text = HoroscopeData.horoscopes[currentSign
            DetailValue].description

            self.callDailyhoroscopeApi(sign: HoroscopeData.horoscopes
            [currentSignDetailValue].name)
        }
    }

    override func didReceiveMemoryWarning()
    {
        super.didReceiveMemoryWarning()
        // Dispose of any resources that can be recreated.
    }

    // MARK: - API Call
    func callDailyhoroscopeApi(sign:String)
    {
        self.spinner.startAnimating()
        HoroscopeService.sharedInstance.callDailyhoroscopeApi(sign: sign,
        parameters: nil, success: {
            (data) in

            self.spinner.stopAnimating()

            self.parseXmlData(data: data)

            }) { (error) in
                self.spinner.stopAnimating()
        }
    }
```

```
// MARK: - XMLParser
func parseXmlData(data:Data)
{
    //create xml parser
    self.parser = XMLParser(data: data)
    self.parser.delegate =  self
    self.parser.parse()

    let success:Bool = self.parser.parse()
    if success {
        print(success)
    }
}

// MARK: - XMLParser Delegate methods

// didStartElement
func parser(_ parser: XMLParser, didStartElement elementName: String,
namespaceURI: String?, qualifiedName qName: String?, attributes
attributeDict: [String : String])
{

    element = elementName as NSString
    if (elementName as NSString).isEqual(to: "item")
    {
        insideAnItem = true
        print(attributeDict)
    }
}

// foundCharacters
func parser(_ parser: XMLParser, foundCharacters string: String)
{

    if element.isEqual(to: "description") && insideAnItem == true
    {
        print("description is \(string)")
        self.horoscopeDescription =   string

    }
}

// didEndElement
func parser(_ parser: XMLParser, didEndElement elementName: String,
namespaceURI: String?, qualifiedName qName: String?)
{

}
```

```
    func parserDidEndDocument(_ parser: XMLParser)
    {
        DispatchQueue.main.async {
            self.spinner.stopAnimating()
            //Display the data on UI

            self.signHoroscope.text =   self.horoscopeDescription
        }

    }

}
```

Figure 6-21 shows the final app, complete with the displayed Aries horoscope from findyourfate.com. To refactor the code we would probably add some URL mocking to the test using the http connection example Cuckoo library example that we explored in Chapter 4.



Aries

Ram

April

Courageous and Energetic.

Over the next few weeks, you will want to get to the bottom of certain financial matters at the moment and will keep digging in your heels until certain people come clean. It is a time for negotiating, looking for compromises and fairer deals all round. You will be handling discreet information well, attracting secrets from others without even appearing to ask. OK, so at times you will also express yourself too forcibly. With Mercury in the deepest area of your chart, you will feel strongly about certain situations, and will not easily change your mind.

Figure 6-21. *Aries detail page with the daily horoscope*

I would be the first to admit that creating the user interface gets in the way of creating a red/green/refactor cadence when you're developing. The unit testing code is limited to code that doesn't directly relate to the iOS UI framework. Ideally we wouldn't have to open Xcode at all and use the command line tools such as xcodebuild to build our app. While that isn't realistic at the moment there are a number of 3rd party tools that can help limit the amount of time we have to spend using Storyboards and the Interface Builder.

We looked at the Stevia library for building interfaces in Chapter 3. There are a number of Swift DSLs (Domain Specific Languages) such as Stevia and SnapKit that aim to dramatically shorten the lines of User Interface code in your applications.

Listing 6-19 shows the Detail View Page code written in Stevia that we wrote for a final refactoring stage.

Listing 6-19. Stevia version of the SignsDetailView

```
import UIKit
import Stevia

class SignsDetailView: UIView
{
    var signName = UILabel()
    var signSymbol =  UILabel()
    var signMonth = UILabel()
    var signDescription = UILabel()
    var signHoroscope = UILabel()
    var spinner = UIActivityIndicatorView()

    convenience init()
    {
       self.init(frame:CGRect.zero)
       // This is only needed for live reload as injectionForXcode
       // doesn't swizzle init methods.
       render()
    }

    func render()
    {
       backgroundColor = .white
       spinner.color = .darkGray
       signName.font = UIFont.boldSystemFont(ofSize: 18)

       sv(
          signName,
          signSymbol,
          signMonth,
          signDescription,
          signHoroscope,
          spinner

       )
```

```
        layout(
           100
           |-10-signName-10-| ~ 30,
           8,
           |-10-signSymbol-10-| ~ 30,
           8,
           |-10-signMonth-10-| ~ 30,
           8,
           |-10-signDescription-10-|,
           8,
           |-10-signHoroscope-10-|,
           |-spinner-|
        )

        self.signDescription.style(self.labelStyle)
        self.signHoroscope.style(self.labelStyle)

    }
    func labelStyle(l:UILabel)
    {
        l.numberOfLines = 0
        l.textAlignment = .left
        l.lineBreakMode = .byWordWrapping
        l.textColor = .black
        l.text = NSLocalizedString("Description", comment: "")
    }
}
```

Instead of using the Interface Builder to drag and drop the Labels we can do all of that in code, see the Layout below for each of the fields.

Listing 6-20. SignsDetailView layout

```
        layout(
           100,
           |-10-signName-10-| ~ 30,
           8,
           |-10-signSymbol-10-| ~ 30,
           8,
           |-10-signMonth-10-| ~ 30,
           8,
           |-10-signDescription-10-|,
           8,
           |-10-signHoroscope-10-|,
           |-spinner-|
        }
```

The layout is created in the `loadView()` SignsDetailViewController as follows, see Listing 6-21.

Listing 6-21. Inflating the SignsDetailView layout

```
let signsDetailView = SignsDetailView()

override func loadView() {
    view = signsDetailView
}
```

As Swift 3 matures hopefully more libraries will emerge that will allow developers to spend more time writing model code and less time connecting interfaces. The complete Stevia example is available with the rest of the source code online.

Conclusion

In this chapter we created a simple three feature Horoscope app using Test Driven Development or TDD.