THE CLEAN SWIFT HANDBOOK

a systematic approach to building maintainable iOS apps

Raymond Law

Table of Contents

0. A Tale of an iOS Developer's Career Journey

Meet Swifty. A computer science fresh grad. Young and enthusiastic. He's motivated to learn new things. New web app frameworks. Javascript libraries. No problem. They are all interesting to him. He picks things up fast, and has plenty of nights and weekends to immerse in coding. Time is never a concern.

A Fresh Beginning

Swifty works for a small consulting agency, doing iOS development. On a typical work day, he checks in to Slack and emails, opens PivotalTracker, finds an open ticket, and gets to work. Occasionally, he needs to get on a standup meeting. He may meet a client at a new project kickoff, or when milestones are reached. His working days are typical. The normal 9-to-5 kind. Or 10-to-6. Or 12-to-8. Regardless of a.m. or p.m.

Life is good. He learns new things and use cool tools. He earns a good salary to pay for his gadgets. Occasionally, he meets up with his developer friends at hackathons. The company sometimes pays for going to conferences such as WWDC. And they provide sodas, snacks, and pizzas. His world is perfect.

Sure, there are sometimes debates and conflicts with his colleagues. But any stress can be cured by the foosball table at work. If that's not enough, there're plenty of snacks and sodas in the fridge. It gives him the turbo boost for the unpaid, overtime, all-nighter. But that's very rare.

Living a Stable Life

Swifty enjoys a good, long, 6 years in his career. No, he didn't stay at one place. He switched jobs a few times for various reasons. At one company, he had a fall out with his manager. At another, he found a better opportunity with a higher raise. At the last place, there was a change in management out of his control.

He's always able to find the next job. He's used to bounce from one place to another. Since he's still single with casual dates, relocation is not a big deal. He just needs a couch he can crash at nights, as long as the salary pays for the rent, plus some for 401k.

With no student loan debt and a frugal lifestyle, he's financially stable. He owns a good, but not fancy car. He saved enough to pay for the down payment for a moderately sized home. He's doing everything that he is supposed to do to live a good life. At this stage of his life, he's ready to settle down and have a family.

Taking the Next Step

With everything going according to plan, Swifty is ready to take the next step. He wants to think about his career long term.

Dœs he want to take on a more senior role in development? A senior software architect who's in charge of designing how all the pieces fit together nicely?

Dœs he want to become a project manager? Dealing with clients and meetings, managing budgets and scope?

Dœs he want to get into business development? Networking with potential clients and customers, bringing more business and making more profits for the company?

There are many career paths he can decide to take. But one thing is certain.

The Love for Coding

After all these years, Swifty still loves coding. He'll have no regrets coding for the rest of his life. And he still loves learning new things. But he found out, in the hard way, that there are more new things than he can afford the time to learn them. It can become overwhelming.

He's observed, over time, new technologies don't interest him as much as before. The motivation to work on side projects on nights and weekends start to disappear. He's looking for more tangible rewards than just being able to say that he did this cool side project.

He also loves to solve hard and interesting problems. It excites him to create something out of nothing. It satisfies him to fix bugs that creep up from ever evolving edge cases. He enjoys improving a feature to perfection.

No mistake. He found a life long career that he enjoys. He realizes that it's just about integrating work into the life he wants.

Achieving Work Life Balance

At some point, Swifty wants to have a family, with a loving wife and kids. He wants to dedicate some of those nights and weekends to his family. He wants to be able to take his wife to vacations, go to his kids soccer games and birthday parties. It'll also be nice to pick up some new hobbies. Maybe learning how to play the piano. Or picking up a new sport. Even just getting some extra rest. Or simply taking a casual walk in the neighborhood. He wants to enjoy life more.

He still wants to work on side projects occasionally. He sees blogging and having a few popular projects on GitHub can really propel his career to new heights. However, he dœsn't want it to occupy all his time. Leaning new things and working on side projects shouldn't feel like taking time away from enjoying life and family. And a great life should add to, not replace, coding.

He feels he has every right to deserve both a loving family and a fulfilling career. He wants to achieve more, as he moves on to the next phase of his life. He's looking for more meanings for the work he dœs.

An Honest Self Assessment

So, Swifty sits down on his couch and has a deep, honest conversation with himself. He assesses where he currently stands, and finds himself at a crossroad.

At his current job, most of his time is spent either fixing bugs or implementing features for client apps, jumping from one ticket to another. It feels like running on a never-ending treadmill. Hours are wildly estimated. Deadlines are tight. Even though he enjoys writing code, he isn't particularly fond of dealing with project management tasks.

Also, many of these client apps no longer exist in the App Store. It's out of his control that the clients decide to kill off the apps. That means he dœsn't have anything to show for his effort. His portfolio is mostly blank because he can't have broken links on his site. His resume is simply a list of employer names. A few years here and there. Some are already bought out by larger agencies, while others are out of business.

If he wants to land the next dream job, how is a potential recruiter going to notice or value him over other candidates? If he wants to go freelancing or consult for other companies, how should he get clients to pay for his services?

Generalist v.s. Specialist

Throughout Swifty’s career, so far, he's become a generalist. A person who knows many things, but not one thing particularly well. He looks back at all the technologies that he has learned in the years before. All those networking libraries, image uploading utilities, Javascript frameworks are no longer the cool kids on the block. Hey, even the very language used to build iOS apps, Objective-C, is being phased out by Swift.

He realizes that, although he knows many things, he hasn't really improved the very core of his craft, that is, writing better code. And knowing many things will never get him to where he wants. There are always going to be newer things that he dœsn't know. Learning them will simply make him a better generalist. Generalists do not get paid well. Specialists do. If he wants to advance in his career, he needs to be a specialist.

He determines that the vertical market for software developers is more lucrative than the horizontal one.

Researching the Space

Swifty decides to level up his game and take it to the next tier. He's going to learn how to write code of the highest quality. He wants to show his future employers and potential clients the quality of his craft. He wants to be trusted and responsible for the success of the most critical software. And he wants to get paid accordingly.

First, he googled for the most popular iOS development resources. There are many books and courses for specific frameworks and technologies. He quickly eliminated these, because he's been on that road before. In his young career so far, he's learned many new things, but has only used them once or twice.

At this moment, he knows better. He knows that he can easily pick something up like Core Bluetooth or HealthKit, if he has to, for a project that needs to use it. He dœsn't need to learn it now only to forget later. And he knows any new tech can become obsolete with a new iOS release at the WWDC every year.

Unit Testing on Steroid

Next, Swifty saw a lot has been talked and written about unit testing in the web app space. He is fully aware of the benefits of having a suite of unit tests to alert you when you're about to break things. Also, TDD seems like a good development practice. The code is born out of unit testing. It means every line of code will be tested when you're driving it with tests. And you don't have to spend extra time to write the tests afterward. What a great idea!

So he read some blogs, bought some books, took some courses to learn how to do unit testing and TDD for iOS apps. But he quickly discovered that reading about testing is very different than applying it to his own code.

It's easy to simply follow along every step in a sample app. Because it's on such a small scale, an average Jœ can understand. But the fact is, without having written a single line of code, you don't know what it's going to look like. It's impossible to write tests for nothingness! He knows there is a deeper issue than simply knowing how to write tests. There is a missing link between good code and unit tests. TDD cannot be fully appreciated before he finds out what that is.

Diving into Architecture

So, Swifty continues his research. This time, he saw a lot of discussions about application architectures. Like most iOS developers, he's been doing MVC, by default, all his life. After all, all of Apple's sample code follow MVC. It can't be wrong, right? Now, he is suddenly overwhelmed with so many choices. The acronym candidates are MVC, MVP, VIP, MVVM, VIPER. The fancy name candidates are Elements, Flux, Redux, Riblets. And variations of each!

The most logical question to ask is: What is the best architecture to use nowadays? After all, who wants to learn a ton of new things just to find out the best one to use? If there's a shortcut to the ultimate answer, who dœsn't want to know it?

Unfortunately, the most common and foolproof answer he sees others offer to this question is: Use the one that fits you and your project most. On the surface, it makes sense. How can you possibly argue against using the best tool for the job? But wait a second, he thought to himself, that's not an argument. It's like saying nothing at all. Dœsn't that apply to anything? Everything?

He realizes there are a lot of noise in the development world. Many people just throw out useless comments and opinions. It's not very helpful. He needs to be able to filter out this noise, and focus on truly improving his code so that he can reap the benefits of unit testing.

Making a Wish List

Swifty wants to improve his code while utilizing his time effectively. So, he makes a list of criteria for the things he is looking for in a good application architecture:

- Free time is only going to be more scarce, as he dedicates more time to his personal live. He doesn't mind spending some, but not all, nights and weekends to learn something new. So whatever he'll learn must be easy and quick to pick up.
- It also must be reusable for many different projects. He doesn't want to learn something new, but only need to use it once or twice. He wants to be able to

apply the same techniques to all his projects. So it must work universally. The return on investment for his time and effort must have a high, positive yield.

- One lesson he learned in the hard way is that you can't rely on a third party library or framework forever, especially if it's free and open source. The original author of a plugin can suddenly stop maintaining it because he's moved on. Also, you don't really read the external code before you use it. You simply follow the directions in the readme to get it to work. So, when there's an issue, it's not easy to fix it yourself. In addition, a future iOS release may make it incompatible. So, it is best to not introduce such a dependency right away.
- A big time sink, he's learned from experience, is that it takes a long time to dig around in a code base to find things. That's true whether it's somebody else's code, or code that he wrote but hasn't looked at for a few months. He knows if this can be improved, productivity will skyrocket.
- He wants a cleaner separation between his personal and professional lives. He used to think about code during his commute, on a date, and sleeping. He doesn't want code to consume all his brain capacity anymore when he's trying to relax. He wants to be at peace with his work when he's not at his desk. This means he wants to be able to get back into the zone easily and quickly the next morning.

By the time you finish reading this book, you'll see why the Clean Architecture can achieve all of these things. The architecture originally comes from the traditional web application space. Clean Swift is just the application of the Clean Architecture to an iOS app.

1. The Massive View Controller Symptom

Your client asks for an estimate to fix a bug. You tell him maybe a couple hours. But you know deep down in your gut that you just aren't sure. You just have to give a number.

If it takes longer than your **guesstimate**, it comes back to bite you. *"I thought you said it would be done in a couple hours."*

Promise broken. Trust lost. Resentment developed. Client fired. Money lost.

To make matters worse, as developers, we have a tendency to underestimate how long something will take.

Maybe the problem lies in the client? They just don't understand how development works.

- They don't know how complicated that thing is.
- They need to stop changing requirements.
- They should focus on functions, not UI first.
- If only they had asked me to do that 2 months ago while the code was still fresh, it would have taken less time.
- *Add your own favorite lines here*

You have to understand the code. Navigate through 50 classes, protocols, and methods. Trace through various conditionals and loops. Identify the relevant line.

Reproduce the bug. Make a change to see how it behaves differently. Rinse and repeat until you fix the bug.

While doing all of these, you fear you may *break something else*. Because there is no unit tests to prevent *regression*.

This same vicious cycle happens every time you need to fix a bug or add a new feature. Thanks to **Massive View Controller**.

2. Failed Attempt at Solving the MVC Problem

If this sounds familiar to you, you may have already attempted to fix the situation. You read a lot about design patterns and how to refactor your view controllers. Extract your data sources and delegates. Put your business logic in the models. How that is supposed to help you write unit tests.

But you are still not doing it. It is 2017. Swift hits 4.0 and iOS 11 is released next month.

So far, the things you read and tried are what I call the first aid kit. Damage has already been done. You are just *putting bandages on your wounds*. You are treating the symptoms, not attacking the root cause.

Can we prevent the damage in the first place, instead of treating the wounds afterward? Why dœs it have a **"re"** in refactoring? Can we just write factored code from the start so we never have to refactor?

3. The Root Cause

About 5 years ago, I decided to do something serious about it. I want to find out why we are still battling massive view controllers. Nonetheless, refactoring and testing are generating all the buzz these days. Is it really possible to write factored code and never have to refactor?

Enter **architecture**. If a building has a shaky foundation, it eventually topples. If your codebase has a shaky architecture, it'll eat you alive. Until you have enough. Let's just rewrite it. We all know how expensive a rewrite is.

I've researched on various iOS architectures out there such as MVC, MVVM, ReactiveCocoa, and VIPER. I've also experimented with different testing and mocking frameworks. I'll write more about these topics in future posts on my blog. In order to help you today, I want to focus on how to apply **Uncle Bob's Clean Architecture** to iOS development using Swift. I'll just call this **Clean Swift**. After reading this book, you'll learn how to:

- Find and fix bugs faster and easier.
- Change existing behaviors with confidence.
- Add new features easily.
- Write shorter methods with single responsibility.
- Decouple class dependencies with established boundaries.
- Extract business logic from view controllers into interactors.
- Build reusable components with workers and service objects.
- Write factored code from the start.
- Write fast and maintainable unit tests.
- Have confidence in your tests to catch regression.

- Apply what you learn to new and existing projects of any size.

What if you know exactly which file to open and which method to look? How much more productive can you be? Imagine your tests run in seconds instead of minutes or hours. And you don't need to learn any testing or mocking frameworks. No CocoaPods to install.

You just follow a simple system and your code just logically falls into place. That's how you know exactly where to look when you look back at your code 3 months later.

4. Introducing Clean Swift - Clean Architecture for iOS

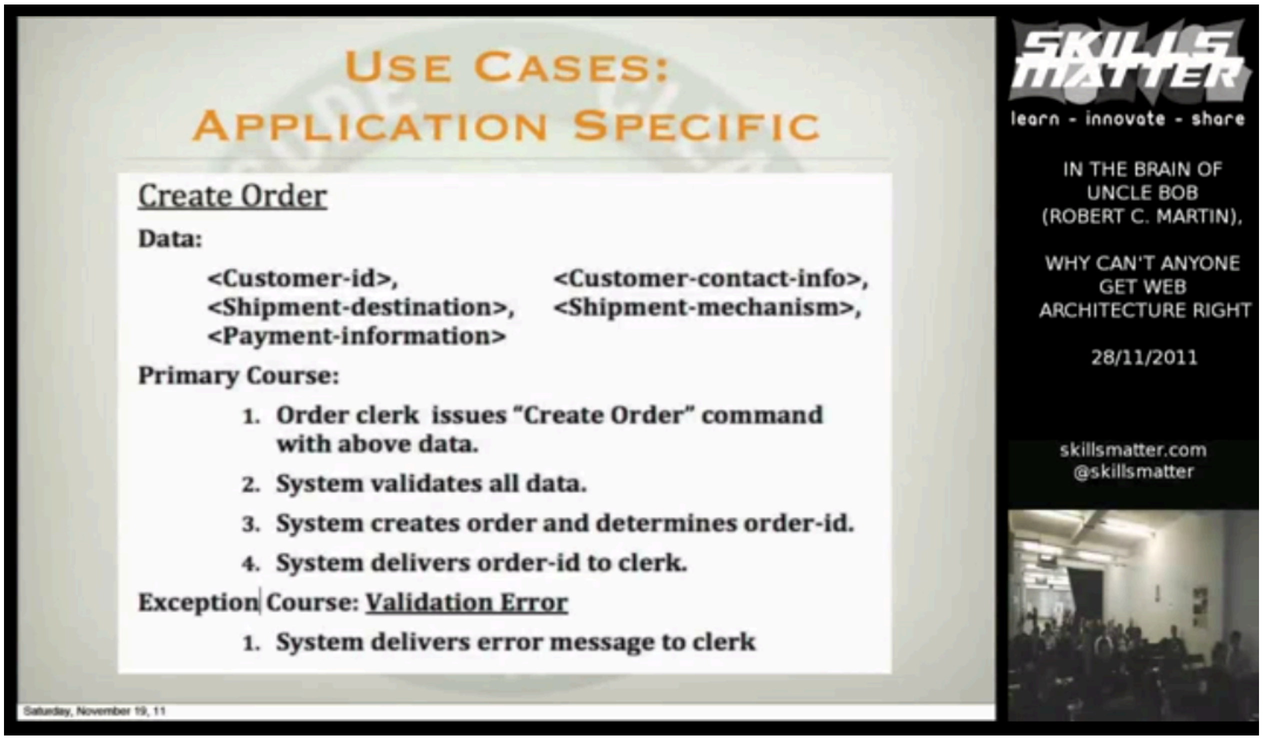
The Clean Swift architecture is derived from the Clean Architecture proposed by Uncle Bob. They share many common concepts such as the components, boundaries, and models. I'm going to implement the same **create order use case** in one of Uncle Bob's talks. While Uncle Bob demonstrates the Clean Architecture using Java in web applications, I'll show you how to apply Clean Architecture using Swift in iOS projects.

Before we begin, make sure you subscribe to my list to get my Xcode templates. Why bother writing all the boilerplate code by hand when you can click a button to generate them?

If you aren't ready for Swift yet, I also have an Objective-C version of my Xcode templates and am looking for people to test it with more project types. If this is you, please email me after you subscribe. I'll send you the Objective-C version. Your feedback will help guide the design of the components as there are differences in the languages.

5. The "Create Order" Use Case

This use case was presented in Uncle Bob's Why can't anyone get Web architecture right? talk. It originates from Ivar Jacobson's book Object Oriented Software Engineering: A Use Case Driven Approach. It is a very good example as it encompasses most of the features of Clean Swift except routing. But worry not, I'll cover routing later in this book.



USE CASES: APPLICATION SPECIFIC

Create Order

Data:

<Customer-id>, <Customer-contact-info>, <Shipment-destination>, <Shipment-mechanism>, <Payment-information>

Primary Course:

1. Order clerk issues "Create Order" command with above data.
2. System validates all data.
3. System creates order and determines order-id.
4. System delivers order-id to clerk.

Exception Course: Validation Error

1. System delivers error message to clerk

SKILLS MATTER
learn - innovate - share
IN THE BRAIN OF UNCLE BOB (ROBERT C. MARTIN),
WHY CAN'T ANYONE GET WEB ARCHITECTURE RIGHT
28/11/2011
skillsmatter.com
@skillsmatter
Saturday, November 19, 11

Data:

- Customer-id
- Customer-contact-info
- Shipment-destination
- Shipment-mechanism

- Payment-information

Primary Course:

1. Order clerk issues “Create Order” command with above data.
2. System validates all data.
3. System creates order and determines order-id.
4. System delivers order-id to clerk.

Exception Course: *Validation Error*

1. System delivers error message to clerk.

We’ll model the data in the use case in our model layer and create special request, response, and view models to pass around at the boundaries between the view controller, interactor, and presenter components. We’ll implement each of the primary course item and validation as business logic in the interactor and, if necessary, workers.

To avoid getting lost, let’s first look at how we organize our code in the Xcode project.

6. Organizing Your Code in Xcode

We'll create a new Xcode project and name it **CleanStore**. Just choose *Single View Application* and *iPhone only* for simplicity's sake. Make sure the language is set to *Swift*. Next, create a nested sub-group **Scenes -> CreateOrder**. When we implement a **delete order use case** in the future, we'll create a new sub-group **Scenes -> DeleteOrder**.

In a typical Xcode project, it is common to see files organized into *model, view, and controller groups*. But every iOS developer knows MVC. It dœsn't tell you anything specific about the project. As Uncle Bob pointed out, group and file names should reveal your intentions for the use cases. It should not reflect the underlying framework structure. So we'll organize each use case under a new group nested within **Scenes**.

Inside the **CreateOrder** group, you can expect all files have something to do with creating an order. Likewise, under the **DeleteOrder** group, you'll find code that deals with deleting an order. If you see a new **ViewOrderHistory** group created by another developer, you already know what to expect.

This organization tells you far more than the *model, view, and controller groups* you are used to see. Over time, you'll accumulate 15 models, 27 view controllers, and 17 views. What do they do? You simply don't know before you inspect each file.

You may ask. What about the shared classes and protocols used by **CreateOrder**, **DeleteOrder** and **ViewOrderHistory**? Well, you can put them in a separate group called **Common -> Order**. How about that for simplicity?

Back to our use case.

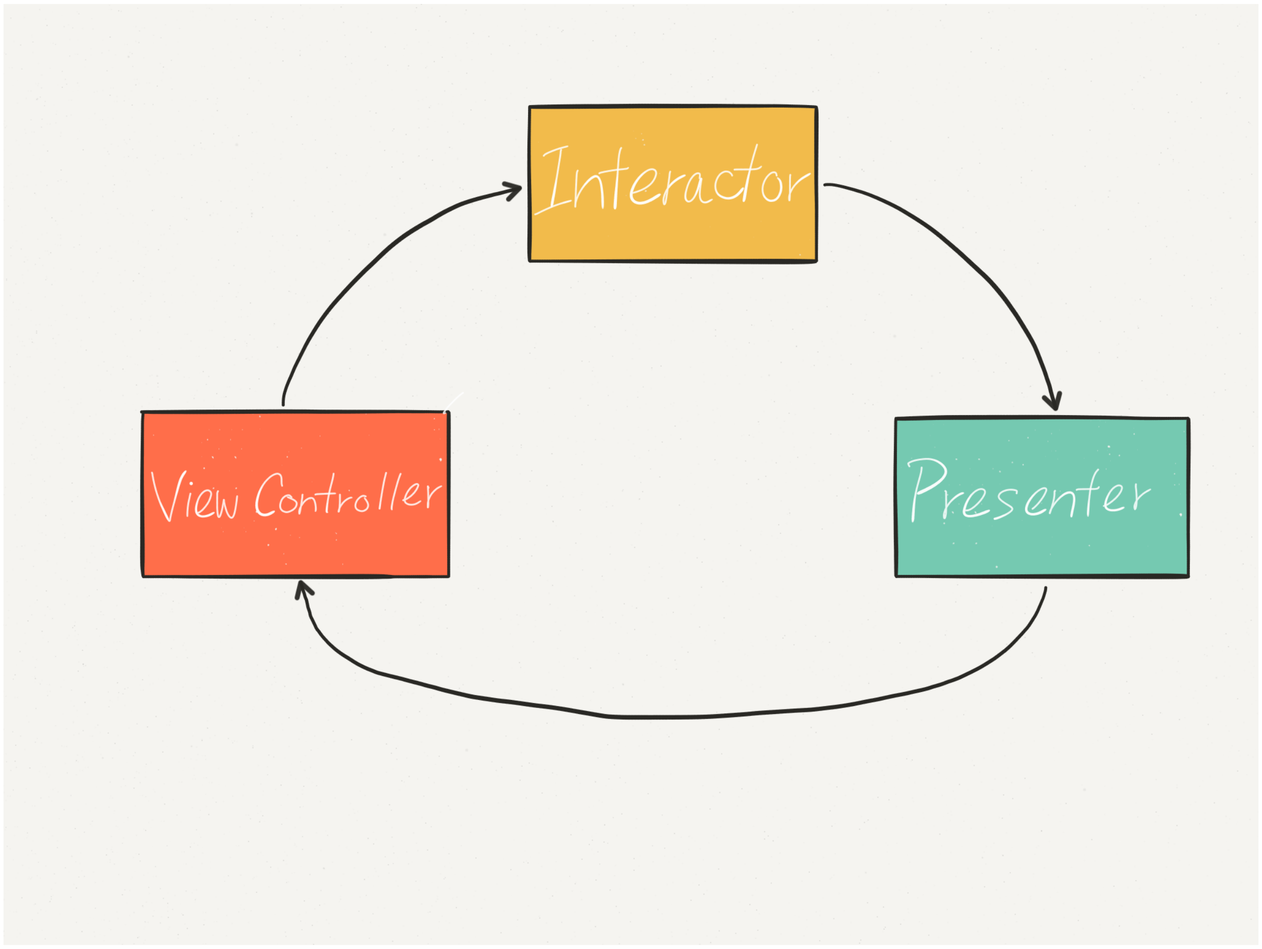
Under the new **CreateOrder** group, we'll create the following Clean Swift components. As we work through the use case, we'll add methods to the component's input and output protocols and then implement them in the components. Make sure you join my email list, so you can use my Xcode templates to create all these components automatically for you with a few clicks.

Watch this screencast now to see how to generate the *seven* Clean Swift components using the Xcode templates to save a lot of time!

You can now compile your project and start building your UI in storyboard. But let's briefly examine the components you've just created first.

7. The VIP Cycle

The view controller, interactor, and presenter are the three main components of Clean Swift. They act as input and output to one another as shown in the following diagram.



To get the full picture of how the VIP cycle fits into your app, take a look at the accompanied VIP diagram.

The view controller's output connects to the interactor's input. The interactor's output connects to the presenter's input. The presenter's output connects to the view controller’s input. We’ll create special objects to pass data through the boundaries between the components. This allows us to decouple the underlying data models from the components. These special objects consists of only primitive types such as Int, Double, and String. We can create structs, classes, or enums to represent the data but there should only be primitive types inside these containing entities.

This is important because when the business rules change that result in changes in the underlying data models. We don’t need to update all over the codebase. The components act as plugins in Clean Swift. That means we can swap in different components provided they conform to the input and output protocols. The app still works as intended.

A typical scenario gœs like. The user taps a button in the app’s user interface. The tap gesture comes in through the IBActions in the view controller. The view controller constructs a request object and sends it to the interactor. The interactor takes the request object and performs some work. It then puts the results in a response object and sends it to the presenter. The presenter takes the response object and formats the results. It then puts the formatted result in a view model object and sends it back to the view controller. Finally, the view controller displays the results to the user.

8. View Controller

What should a view controller do in an iOS app? The base class name `UITableViewController` should tell you something. You want to put code there to control `UITableView` and `UIView` subclasses. But what dœs this control code look like? What qualifies as control code and what dœsn't?

Let's dive in.

```
import UIKit

protocol CreateOrderDisplayLogic: class
{
  func displaySomething(viewModel: CreateOrder.Something.ViewModel)
}

class CreateOrderViewController: UITableViewController, Create-
OrderDisplayLogic
{
  var interactor: CreateOrderBusinessLogic?
  var router: (NSObjectProtocol & CreateOrderRoutingLogic & Create-
OrderDataPassing)?

  // MARK: Object lifecycle

  override init(nibName nibNameOrNil: String?, bundle nibBundleOr-
Nil: Bundle?)
  {
    super.init(nibName: nibNameOrNil, bundle: nibBundleOrNil)
    setup()
  }

  required init?(coder aDecoder: NSCoder)
  {
    super.init(coder: aDecoder)
    setup()
  }
  // MARK: Setup

  private func setup()
  {
    let viewController = self
    let interactor = CreateOrderInteractor()
    let presenter = CreateOrderPresenter()
    let router = CreateOrderRouter()
    viewController.interactor = interactor
    viewController.router = router
    interactor.presenter = presenter
    presenter.viewController = viewController
    router.viewController = viewController
    router.dataStore = interactor
  }

  // MARK: Routing

  override func prepare(for segue: UIStoryboardSegue, sender: Any?)
  {
    if let scene = segue.identifier {
      let selector = NSSelectorFromString("routeTo\(scene)With-
Segue:")
      if let router = router, router.responds(to: selector) {
        router.perform(selector, with: segue)
      }
    }
  }


  // MARK: View lifecycle

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

  // MARK: Do something

  //@IBOutlet weak var nameTextField: UITextField!


  func doSomething()
  {
    let request = CreateOrder.Something.Request()
    interactor?.doSomething(request: request)
  }
```

```
    func displaySomething(viewModel: CreateOrder.Something.ViewModel)
    {
      //nameTextField.text = viewModel.name
    }
  }
```

`CreateOrderDisplayLogic` and `CreateOrderBusinessLogic` Protocols

The `CreateOrderDisplayLogic` protocol specifies the inputs to the `CreateOrderViewController` component (it conforms to the protocol). The `CreateOrderBusinessLogic` protocol specifies the outputs. You'll see this same pattern in the interactor and presenter later.

There is one method `doSomething(request:)` in the output protocol. If another component wants to act as the output of `CreateOrderViewController`, it needs to support `doSomething(request:)` in its input.

From the VIP cycle you saw earlier, we know this output is going to be the interactor. But notice in `CreateOrderViewController.swift`, there is no mention of `CreateOrderInteractor`. This means we can swap in another component to be the output of `CreateOrderViewController` as long as it supports `doSomething(request:)` in its input protocol.

The argument to `doSomething(request:)` is a request object that is passed through the boundary from the view controller to the interactor. This request object is a `CreateOrder.Something.Request` struct. It consists of primitive types, not the whole order data that we identified earlier. This means we have decoupled the underlying order data model from the view controller and interactor. When we make changes to the order data model in the future (for example, add an internal order ID field), we don't need to update anything in the Clean Swift components.

I'll come back to the `displaySomething(viewModel:)` method in the output protocol later when we finish the VIP cycle.

`interactor` and `router` Variables

The `interactor` variable is an object that conforms to the `CreateOrderBusinessLogic` protocol. Although we know it is going to be the interactor, but it dœsn't need to be.

The `router` variable conforms to the `CreateOrderRoutingLogic` & `CreateOrderDataPassing` protocols, and is used to navigate and pass data to different scenes.

`setup()` Method

The `setup()` method is invoked from `init(nibName:bundle:)` and `init?(coder:)` to set up the VIP cycle.

This is where the arrows are drawn in the VIP cycle. Note the arrows are uni-directional. This consistent flow of control makes things very clear. It is the reason why you know exactly which file and method to look for when you are fixing bugs.

Only the view controller is loaded from the storyboard. We need to actually create the interactor, presenter, and router instances manually. The `setup()` method dœs this, and then assigns the corresponding references to the `interactor`, `presenter`, `viewController`, and `router` variables.

The important thing to remember here is the **VIP cycle**:

*The output of the view controller is connected to the input of the interactor. The output of the interactor is connected to the input of the presenter. The output of the presenter is connected to the input of the view controller. This means the flow of control is always **unidirectional**.*

Remembering the VIP cycle will become very handy when you implement features and fix bugs. You'll know exactly which file and method to look for.

It also simplifies your dependency graph. You don't want objects to have references to one another whenever they please. It may seem convenient for the view controller to ask the presenter directly to format a string. But over time, you'll end up with a mess in your dependency graph. Keep this in mind at all times to avoid any unnecessary coupling.

This will become very clear when we implement the create order use case.

Flow of Control

In `viewDidLoad()`, we have some business logic to run, so we call `doSomething()`. In this method, we create a `CreateOrder.Something.Request` object and invoke `doSomething(request:)` on the output (the interactor). That's it. We ask the output to perform our business logic. The view controller dœsn't and shouldn't care who and how it is done.

9. Interactor

The interactor contains your app's business logic. The user taps and swipes in your UI in order to interact with your app. The view controller collects the user inputs from the UI and passes it to the interactor. It then retrieves some models and asks some workers to do the work.

```
import UIKit

protocol CreateOrderBusinessLogic
{
  func doSomething(request: CreateOrder.Something.Request)
}

protocol CreateOrderDataStore
{
  //var name: String { get set }
}

class CreateOrderInteractor: CreateOrderBusinessLogic, CreateOrder-
DataStore
{
  var presenter: CreateOrderPresentationLogic?
  var worker: CreateOrderWorker?

  // MARK: Do something

  func doSomething(request: CreateOrder.Something.Request)
  {
    worker = CreateOrderWorker()
    worker?.doSomeWork()

    let response = CreateOrder.Something.Response()
    presenter?.presentSomething(response: response)
  }
}
```

`CreateOrderBusinessLogic` and `CreateOrderPresentationLogic` Protocols

The `CreateOrderBusinessLogic` protocol specifies the inputs to the `CreateOrderInteractor` component (it conforms to the protocol). The `CreateOrderPresentationLogic` protocol specifies the outputs.

We declare the `doSomething(request:)` method for the use case in the `CreateOrderBusinessLogic` protocol. The output of `CreateOrderViewController` is connected to the input of the `CreateOrderInteractor`.

The `CreateOrderPresentationLogic` protocol has one method `presentSomething(response:)`. The output of `CreateOrderInteractor` needs to support `presentSomething(response:)` in order to act as the output. *Hint: The output is going to be the P in VIP.*

Another thing to note here is the argument to `doSomething(request:)` is the request object of type `CreateOrder.Something.Request`. The interactor peeks inside this request object to retrieve any necessary data to do its job.

Similarly, the argument to `presentSomething(response:)` is the response object of type `CreateOrder.Something.Response`.

`presenter` and `worker` Variables

The `presenter` variable is an object that conforms to the `CreateOrderPresentationLogic` protocol. Although we know it is going to be the presenter, but it dœsn't need to be.

The `worker` variable of type `CreateOrderWorker` is a specialized object that will actually create the new order. Since creating the order likely involves persistence in Core Data and making network calls. It is simply too much work for the interactor to do alone. Keep in mind the interactor also has to validate the order form and this is likely going to be extracted into its own worker.

Flow of Control

When the input to `CreateOrderInteractor` (i.e. `CreateOrderViewController`) invokes `doSomething(request:)`, it first creates the worker object and asks it to do some work by calling `doSomeWork()`. It then constructs a response object and invokes `presentSomething(response:)` on the output.

Let's take a quick look at the worker next.

10. Worker

A profile view may need to fetch the user from Core Data, download the profile photo, allows users to like and follow, …etc. You don't want to swamp the interactor with doing all these tasks. Instead, you can break it down into many workers, each doing one thing. You can then reuse the same workers elsewhere.

The `CreateOrderWorker` is very simple as it just provides an interface and implementation of the work it can do to the interactor.

```
import UIKit

class CreateOrderWorker
{
  func doSomeWork()
  {
  }
}
```

11. Presenter

After the interactor produces some results, it passes the response to the presenter. The presenter then marshals the response into a view model suitable for display. It then passes the view model back to the view controller for display to the user.

```swift
import UIKit

protocol CreateOrderPresentationLogic
{
  func presentSomething(response: CreateOrder.Something.Response)
}

class CreateOrderPresenter: CreateOrderPresentationLogic
{
  weak var viewController: CreateOrderDisplayLogic?

  // MARK: Do something

  func presentSomething(response: CreateOrder.Something.Response)
  {
    let viewModel = CreateOrder.Something.ViewModel()
    viewController?.displaySomething(viewModel: viewModel)
  }
}
```

`CreateOrderPresentationLogic` and `CreateOrderDisplayLogic` Protocols

The `CreateOrderPresentationLogic` protocol specifies the inputs to the `CreateOrderPresenter` component (it conforms to the protocol). The `CreateOrderDisplayLogic` protocol specifies the outputs.

By now, the `presentSomething(response:)` and `displaySomething(viewModel:)` methods don't need to be explained. The `CreateOrder.Something.Response` argument is passed through the interactor-presenter boundary whereas the `CreateOrder.Something.ViewModel` argument is passed through the presenter-view controller boundary as the VIP cycle completes.

`viewController` Variable

The `viewController` variable is an object that conforms to the `CreateOrderDisplayLogic` protocol. Although we know it is going to be the view controller, but it dœsn't need to be. A subtle difference here is we make `viewController` a *weak* variable to avoid a reference cycle when this *CreateOrder* scene is no longer needed and the components are deallocated.

Flow of Control

Since the output of `CreateOrderInteractor` is connected to the input of `CreateOrderPresenter`, the `presentSomething(response:)` method will be called after the interactor finishes doing its work. It simply constructs the view model object and invokes `displaySomething(viewModel:)` on the output.

I promised you that we would come back to the `displaySomething(viewModel:)` method in the view controller. This is the last step in the VIP cycle. It takes any data in the view model object and displays it to the user. For example, we may want to display the customer's name in a text field: `nameTextField.text = viewModel.name`.

Congratulations! You just learned the essence of Clean Swift. You should now be able to extract business and presentation logic from your user interface code. But don't worry. I won't leave you without an example. But let's finish talking about the rest of the Clean Swift components first.

12. Router

When the user taps the next button to navigate to the next scene in the storyboard, a segue is trigged and a new view controller is presented. A router extracts this navigation logic out of the view controller. It is also the best place to pass any data to the next scene. As a result, the view controller is left with just the task of controlling views.

```
import UIKit

@objc protocol CreateOrderRoutingLogic
{
  func routeToSomewhere(segue: UIStoryboardSegue?)
}

protocol CreateOrderDataPassing
{
  var dataStore: CreateOrderDataStore? { get }
}

class CreateOrderRouter: NSObject, CreateOrderRoutingLogic, Create-
OrderDataPassing
{
  weak var viewController: CreateOrderViewController?
  var dataStore: CreateOrderDataStore?

  // MARK: Routing

  //func routeToSomewhere(segue: UIStoryboardSegue?)
  //{
  //  if let segue = segue {
  //    let destinationVC = segue.destination as! SomewhereViewCon-
troller
  //    var destinationDS = destinationVC.router!.dataStore!
  //    passDataToSomewhere(source: dataStore!, destination: &des-
tinationDS)
  //  } else {
```

```
  //    let storyboard = UIStoryboard(name: "Main", bundle: nil)
  //    let destinationVC =
storyboard.instantiateViewController(withIdentifier: "Somewhere-
ViewController") as! SomewhereViewController
  //    var destinationDS = destinationVC.router!.dataStore!
  //    passDataToSomewhere(source: dataStore!, destination: &des-
tinationDS)
  //    navigateToSomewhere(source: viewController!, destination:
destinationVC)
  //  }
  //}


  // MARK: Navigation

  //func navigateToSomewhere(source: CreateOrderViewController,
destination: SomewhereViewController)
  //{
  //  source.show(destination, sender: nil)
  //}

  // MARK: Passing data

  //func passDataToSomewhere(source: CreateOrderDataStore, destina-
tion: inout SomewhereDataStore)
  //{
  //  destination.name = source.name
  //}
}
```

`CreateOrderRoutingLogic` **Protocol**

The `CreateOrderRoutingLogic` protocol specifies its routes - the scenes it can navigates to - to the view controller. The method `routeToSomewhere(segue:)` tells the view controller that: If you use me as your router, I know how to route to a scene called somewhere.

As you can see in the comment inside the `routeToSomewhere(segue:)` method, the router is very flexible in the number of ways it can navigate to another scene. You'll see this in action later when we have more than one scene.

`CreateOrderDataPassing` Protocol

The `CreateOrderDataPassing` protocol declares the `dataStore` variable to be of type `CreateOrderDataStore`. It is later used to pass data to the next scene.

`viewController` Variable

The `viewController` variable is just a reference to the view controller that uses this router. It is a *weak* variable to avoid the reference cycle problem, and is set up by the configurator as you'll soon see. The Apple way of transitioning between segues adds all the *present** and *push** methods to the `UIViewController` class. So we need to have `viewController` here so we can call those methods in the router.

`dataStore` Variable

The `dataStore` variable, by default, is set to be the interactor, but restricted access to those data declared in the `CreateOrderDataStore` protocol. You pass data from one scene to the next by setting the data in the destination data store to those in the source data store.

`navigateToSomewhere(source:destination:)` Method

The `navigateToSomewhere(source:destination:)` method contains the details of how to actually present a view controller, with code that you're already familiar with.

`passDataToSomewhere(source:destination:)` Method

The `passDataToSomewhere(source:destination:)` method dœs the actual data passing from the source to the destination data store.

13. Models

In order to completely decouple the Clean Swift components, we need to define data models to pass through the boundaries between them, instead of just using raw data models. There are 3 primary types of models:

- **Request** - The view controller constructs a request model and passes it to the interactor. A request model contains mostly user inputs, such as text entered in text fields and values chosen in pickers.
- **Response** - After the interactor finishes doing work for a request, it encapsulates the results in a response model and then passes it to the presenter.
- **View Model** - After the presenter receives the response from the interactor, it formats the results into primitive data types such as String and Int, and stuff them in the view model. It then passes the view model back to the view controller for display.

```
import UIKit

enum CreateOrder
{
  enum Something
  {
    struct Request
    {
    }
    struct Response
    {
    }
    struct ViewModel
    {
    }
  }
}
```

In the code generated by the templates, we don't actually have any data in these models. So they are just empty. But you'll see actual data when we implement the create order use case. Let's see what this data looks like next.

Tired of all these boilerplate code? Watch the accompanied Setup Tutorial to generate all these for you automagically. The Quick Start Guide lists the exact steps you carry out to implement a new feature.

I recommend downloading and installing the templates before you continue from this point on, so that you can copy and paste the code to try in your own Xcode. I can wait. If you want to get a quick win by trying out the templates, watch this screencast to see how I use the templates to get a scene up and running in 3 minutes.

Okay. Are you ready to see Clean Swift in action? There're 3 scenes in the CleanStore sample app:

1. CreateOrder
2. ListOrders
3. ShowOrder

And we'll go through the entire app in this order. Let's begin.

14. The `CreateOrder` Scene

Let's start with the "Create Order" use case in the original Clean Architecture presentation by Uncle Bob.

"Create Order" Data

Let's break down the create order use case to come up with the data required to create a new order. We'll then create a form using table view and text fields in the app to collect this data from the user.

- Customer ID
    - Integer
- Customer Contact Info
    - First name
    - Last name
    - Phone
    - Email
- Shipment Destination
    - Shipping address
        - Street 1
        - Street 2
        - City
        - State
        - ZIP
- Shipment Mechanism
    - Shipping method
        - Next day

        - 3 to 5 days
        - Ground
- Payment Information
    - Credit card number
    - Expiration date
    - CVV
    - Billing address
        - Street 1
        - Street 2
        - City
        - State
        - ZIP

That's a gigantic form! In a more realistic app, we'll likely have some of this data already after the user has logged in such as name, phone, email, shipping and billing address, and maybe credit card info. So this form will be dramatically slimmed down.

"Create Order" Business Logic

Now, let's see what business logic we can come up with from the use case's requirements. This can serve as pseudocode that we'll later write acceptance tests for.

1. Order clerk issues "Create Order" command with above data.
    - Display a form to collect data from user.
    - Form uses text fields to collect text data.
    - Form uses a picker to collect Shipping method and Expiration data.
    - Form uses a switch to auto-fill Billing address from Shipping address.
    - Form uses a button to issue the "Create Order" command.
2. System validates all data.
    - Ensure all fields except Street 2 are not blank.
    - If valid, display a "valid" message below the button.
    - If invalid, display error messages next to the invalid fields.

3. System creates order and determines order-id.
    - Generate an unique order ID.
    - Create and store a new order in Core Data.
4. System delivers order-id to clerk.
    - Display order ID on screen to user.

This is a good set of initial features to demonstrate how Clean Swift works and its benefits. As requirements change, Clean Swift is flexible to adapt to them easily. Some future requirements may be:

- Use Core Location to reverse geocode current lat/lng to address to pre-fill shipping address and billing address.
- Integrate Stripe API to collect credit card info.
- Validate fields as data is entered instead of after the button is tapped.
- Add country to shipping and billing addresses to expand business overseas.
- Format phone number, credit card number, and expiration date.

Now we are ready to implement the create order use case.

Design the `CreateOrder` Scene in Storyboard and View Controller

Where should we begin?

You want to start by collecting user inputs such as text and taps in the view controller. The view controller then passes the inputs to the interactor to get some work done. Next, the interactor passes the output to the presenter for formatting. Finally, the presenter asks the view controller to display the results.

This is the flow of control and it is always in one direction. The VIP cycle is not named VIP because of a very important person. It is because there is an order to things. **V then I then P.**

Let's start by creating the create order form. In the storyboard, embed the `CreateOrderViewController` in a `UINavigationController`. Add a title **Create Order** to give it some context.

Make the table view use *static cells*. Add a new section for each group of data and a new cell for each piece of data required in the create order form. For each cell, add a `UILabel` and `UITextField`.

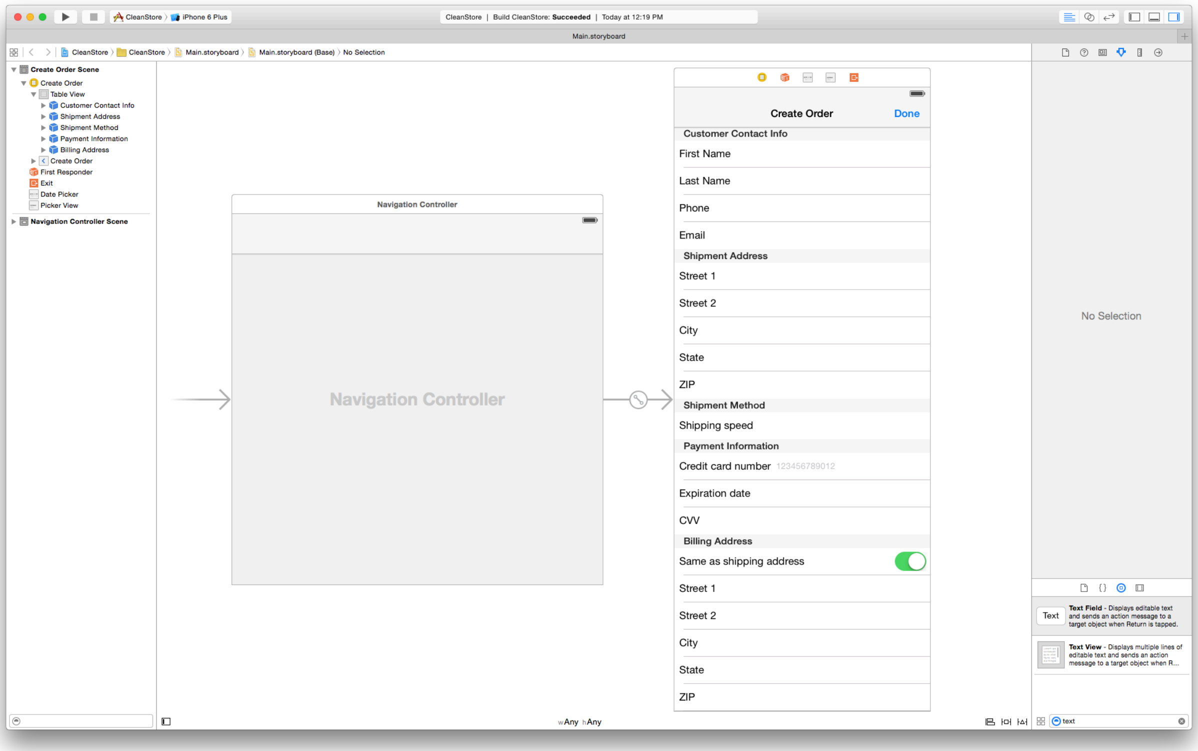
Make the following IBOutlet connections:

- Connect all the text fields to the **`textFields`** IBOutlet collection. Also set their delegates to be our **`CreateOrderViewController`**.
- Connect the text field for shipping method to the **`shippingMethodTextField`** IBOutlet.
- Connect the text field for expiration date to the **`expirationDateTextField`** IBOutlet.

Drag a **`UIPickerView`** and **`UIDatePicker`** from the Object Library to the scene. Then make the following IBOutlet connections:

- Connect the **`UIPickerView`** to the **`shippingMethodPicker`** IBOutlet. Also set the data source and delegate to be our **`CreateOrderViewController`**.
- Connect the **`UIDatePicker`** to the **`expirationDatePicker`** IBOutlet. Also control-drag from the **`UIDatePicker`** to the **`CreateOrderViewController`** in the assistant editor to create an IBAction for the value change event and name it **`expirationDatePickerValueChanged()`**.

Your form should look like:



It won't win any design award, but it satisfies our use case requirements. The beauty of Clean Swift is you can modify the view later without affecting other parts of the app. For example, we can add country to shipping and billing addresses to expand the client's business overseas. Or we can hire a professional designer to style the form.

After setting up the IBOutlets and IBActions, your `CreateOrderViewController` should also contain the following code:

```
// MARK: Text fields

@IBOutlet var textFields: [UITextField]!

// MARK: Shipping method

@IBOutlet weak var shippingMethodTextField: UITextField!
@IBOutlet var shippingMethodPicker: UIPickerView!

// MARK: Expiration date

@IBOutlet weak var expirationDateTextField: UITextField!
```

```
  @IBOutlet var expirationDatePicker: UIDatePicker!

  @IBAction func expirationDatePickerValueChanged(sender: AnyOb-
ject)
  {
  }
```

Now the user interface is all set. Let's tackle user interaction next. When the user taps the *next* button in the keyboard, we want him to be able to enter text for the next text field. Make the `CreateOrderViewController` conform to the `UITextFieldDelegate` protocol. Then add the `textFieldShouldReturn()` method.

```
  func textFieldShouldReturn(textField: UITextField) -> Bool
  {
    textField.resignFirstResponder()
    if let index = textFields.indexOf(textField) {
      if index < textFields.count - 1 {
        let nextTextField = textFields[index + 1]
        nextTextField.becomeFirstResponder()
      }
    }
    return true
  }
```

When the user taps the table view cell (not the text field directly), we still want the user to be able to edit the text field. After all, using `UITextBorderStyleNone` makes it impossible to see the boundary of the text field. Plus, it just feels better and a common behavior to be able to tap a label and edit the field. To achieve this behavior, add the `tableView:didSelectRowAtIndexPath()` method.

```
  override func tableView(tableView: UITableView, didSelectRowAtIn-
dexPath indexPath: NSIndexPath)
  {
    if let cell = tableView.cellForRowAtIndexPath(indexPath) {
      for textField in textFields {
        if textField.isDescendantOfView(cell) {
          textField.becomeFirstResponder()
        }
      }
    }
  }
```

Implement Shipping Methods Business Logic in the Interactor

Getting the pickers for shipping method and expiration date to work is pretty straight forward. But more importantly, this is where we encounter our first business logic.

Let's look at shipping method first.

The available shipping methods can change in the client's business as it partners with different shippers over time. We don't want to leave it in the view controller. So, we'll extract this *business logic* to the *interactor*.

Start by adding the `configurePickers()` method and invoke it in `viewDidLoad()`. When the user taps the `shippingMethodTextField`, the correct picker UI will be shown instead of the standard keyboard.

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

    func configurePickers()
    {
      shippingMethodTextField.inputView = shippingMethodPicker
    }
```

Next, make `CreateOrderViewController` conform to the `UIPickerViewDataSource` and `UIPickerViewDelegate` protocols, and add the following methods.

```
    func numberOfComponentsInPickerView(pickerView: UIPickerView) ->
  Int
    {
      return 1
    }
    func pickerView(pickerView: UIPickerView, numberOfRowsInComponent
  component: Int) -> Int
    {
      return interactor?.shippingMethods.count
    }

    func pickerView(pickerView: UIPickerView, titleForRow row: Int,
  forComponent component: Int) -> String?
    {
      return interactor?.shippingMethods[row]
    }

    func pickerView(pickerView: UIPickerView, didSelectRow row: Int,
  inComponent component: Int)
    {
      shippingMethodTextField.text = interactor?.shippingMethods[row]
    }
```

Since the shipping method business logic is moved to the interactor, we can invoke it using `interactor?.shippingMethods` in the view controller. We then also need to add the `shippingMethods` variable to the `CreateOrderBusinessLogic` protocol. For now, we'll keep things simple by assuming it is an array of fixed literal strings. In the future, we can retrieve the available shipping methods dynamically in Core Data or over the network.

In `CreateOrderInteractor`:

```
protocol CreateOrderBusinessLogic
{
  var shippingMethods: [String] { get }
}

class CreateOrderInteractor: CreateOrderBusinessLogic, CreateOrder-
DataStore
{
  var presenter: CreateOrderPresentationLogic?
  var worker: CreateOrderWorker?
  var shippingMethods = [
    "Standard Shipping",
    "Two-Day Shipping ",
    "One-Day Shipping "
  ]
}
```

Implement Expiration Date Business Logic in the Interactor

The expiration date can be formatted differently depending on the user's language and location. We'll move this *presentation logic* to the *presenter*.

Let's take care of the expiration date picker in `configurePickers()` first.

```
  func configurePickers()
  {
    shippingMethodTextField.inputView = shippingMethodPicker
    expirationDateTextField.inputView = expirationDatePicker
  }
```

Next, modify the `expirationDatePickerValueChanged()` method to look like:

```
  @IBAction func expirationDatePickerValueChanged(sender: AnyOb-
ject)
  {
    let date = expirationDatePicker.date
    let request = CreateOrder.FormatExpirationDate.Request(date:
date)
    interactor?.formatExpirationDate(request)
  }
```

After the user chooses an expiration date in the picker, the `expirationDatePickerValueChanged()` method is invoked. We retrieve the date from the picker, and stuffs it in this weird looking thing called `CreateOrder.FormatExpirationDate.Request`. It is simply a Swift struct that we define in `CreateOrderModels.swift`. It has one data member named `date` of `Date` type.

```
enum CreateOrder
{
  enum FormatExpirationDate
  {
    struct Request
    {
      var date: Date
    }
  }
  }
```

The following discussion about _ is from an old version of the templates. I leave it here because it serves some good historical purpose. You can read about the new and better models in this post.

> *You may wonder why I use _ here. Objective-C and Swift naming convention suggests using upper camel case for token names such as class, struct, and enum. Before you scream at me for breaking convection. Let me explain why I used _.*
>
> `CreateOrder` *is the name of the scene.* `FormatExpirationDate` *is the intention or business rule.* `Request` *indicates the boundary between the view controller and interactor. That's why you get* `CreateOrder_FormatExpirationDate_Request`.
>
> *Contrast* `CreateOrder_FormatExpirationDate_Request` *with* `CreateOrderFormatExpirationDateRequest`. *Which one tells you about the scene, intention, and boundary more clearly? So I chose clarity over dogma.*

Now, back to the `expirationDatePickerValueChanged()` method.

After we create the request object and initialize it with the date the user has picked, we simply ask the output to format the expiration date by calling `interactor?.formatExpirationDate(request: request)`.

Yes, you guessed it right. We do need to add this new method to the `CreateOrderBusinessLogic` protocol.

In `CreateOrderInteractor`:

```
protocol CreateOrderBusinessLogic
{
  var shippingMethods: [String] { get }
  func formatExpirationDate(request: CreateOrder.FormatExpiration-
Date.Request)
}

class CreateOrderInteractor: CreateOrderBusinessLogic, CreateOrder-
DataStore
{
  func formatExpirationDate(request: CreateOrder.FormatExpiration-
Date.Request)
  {
    let response = CreateOrder.FormatExpirationDate.Response(date:
request.date)
    presenter?.presentExpirationDate(response)
  }
}
```

Implement Expiration Date Presentation Logic in the Presenter

In the `formatExpirationDate()` method, we create the `CreateOrder.FormatExpirationDate.Response` response object as defined in `CreateOrderModels.swift`.

```
enum CreateOrder
{
  enum FormatExpirationDate
  {
    struct Response
    {
      var date: Date
    }
  }
}
```

We initialize the response object with the date we get from the request model. The interactor dœs not do anything with the date and just passes is straight through by invoking `presenter?.presentExpirationDate(response: response)`. There is currently no business logic associated with the expiration date. Later, when we add validation to make sure the expiration date dœsn't fall in the past, we'll add that business rule here in the interactor.

Let's continue by adding the `presentExpirationDate(response:)` method to the `CreateOrderPresentationLogic` protocol.

In `CreateOrderPresenter`:

```
protocol CreateOrderPresentationLogic
{
  func presentExpirationDate(response: CreateOrder.FormatExpira-
tionDate.Response)
}

class CreateOrderPresenter: CreateOrderPresentationLogic
{
  weak var viewController: CreateOrderDisplayLogic?
  let dateFormatter: DateFormatter = {
    let dateFormatter = DateFormatter()
    dateFormatter.dateStyle = .short
    dateFormatter.timeStyle = .none
    return dateFormatter
  }()

  func presentExpirationDate(response: CreateOrder.FormatExpira-
tionDate.Response)
  {
    let date = dateFormatter.string(from: response.date)
    let viewModel =
CreateOrder.FormatExpirationDate.ViewModel(date: date)
    viewController?.displayExpirationDate(viewModel: viewModel)
  }
}
```

In `CreateOrderPresenter`, we define a `DateFormatter` constant. The `presentExpirationDate(response:)` method simply asks this date formatter object to convert the expiration date from `Date` to `String`. It then stuffs this date string representation in the `CreateOrder.FormatExpirationDate.ViewModel` struct as defined in `CreateOrderModels.swift`.

Note that the date in the view model is a `String`, not `Date`. A presenter's job is to marshal data into a format suitable for display to the user. Since our UI displays the date in a `UITextField`, we convert a date into a string. In fact, most of the time, the data in your view models will be either strings or numbers since that's what human read.

```
enum CreateOrder
{
  enum FormatExpirationDate
  {
    struct ViewModel
    {
      var date: String
    }
  }
}
```

It finally asks the view controller to display it by calling `viewController?.displayExpirationDate(viewModel: viewModel)`.

Implement Display Expiration Date Display Logic in the View Controller

This also means we need to define the `displayExpirationDate(viewModel:)` method in the `CreateOrderDisplayLogic` protocol.

In `CreateOrderViewController`:

```
protocol CreateOrderDisplayLogic
{
  func displayExpirationDate(viewModel: CreateOrder.FormatExpira-
tionDate.ViewModel)
}
class CreateOrderViewController: UITableViewController, Create-
OrderDisplayLogic, UITextFieldDelegate, UIPickerViewDataSource,
UIPickerViewDelegate
{
  func displayExpirationDate(viewModel: CreateOrder.FormatExpira-
tionDate.ViewModel)
  {
    let date = viewModel.date
    expirationDateTextField.text = date
  }
  }
```

In the `displayExpirationDate(viewModel:)` method, we just need to grab the date string from the view model and assign it to the textField, as in `expirationDateTextField.text = date`.

That's it. This completes the VIP cycle.

You can find the complete code example at GitHub.

You've got a working create order form. The user can enter text in the text fields and choose shipping method and expiration date in the pickers. Your business and presentation logic are extracted away from your view controller into the interactor and presenter. Custom boundary model structs are used to decouple the Clean Swift components at their boundaries.

In this example, we haven't looked at the worker and router yet. When your business logic is more complicated, a little division of labor helps. An interactor's job can be broken down into multiple smaller tasks, which can then be performed by individual workers. When you need to show a different scene after a new order is created, you'll need to use the router to extract away the navigation logic.

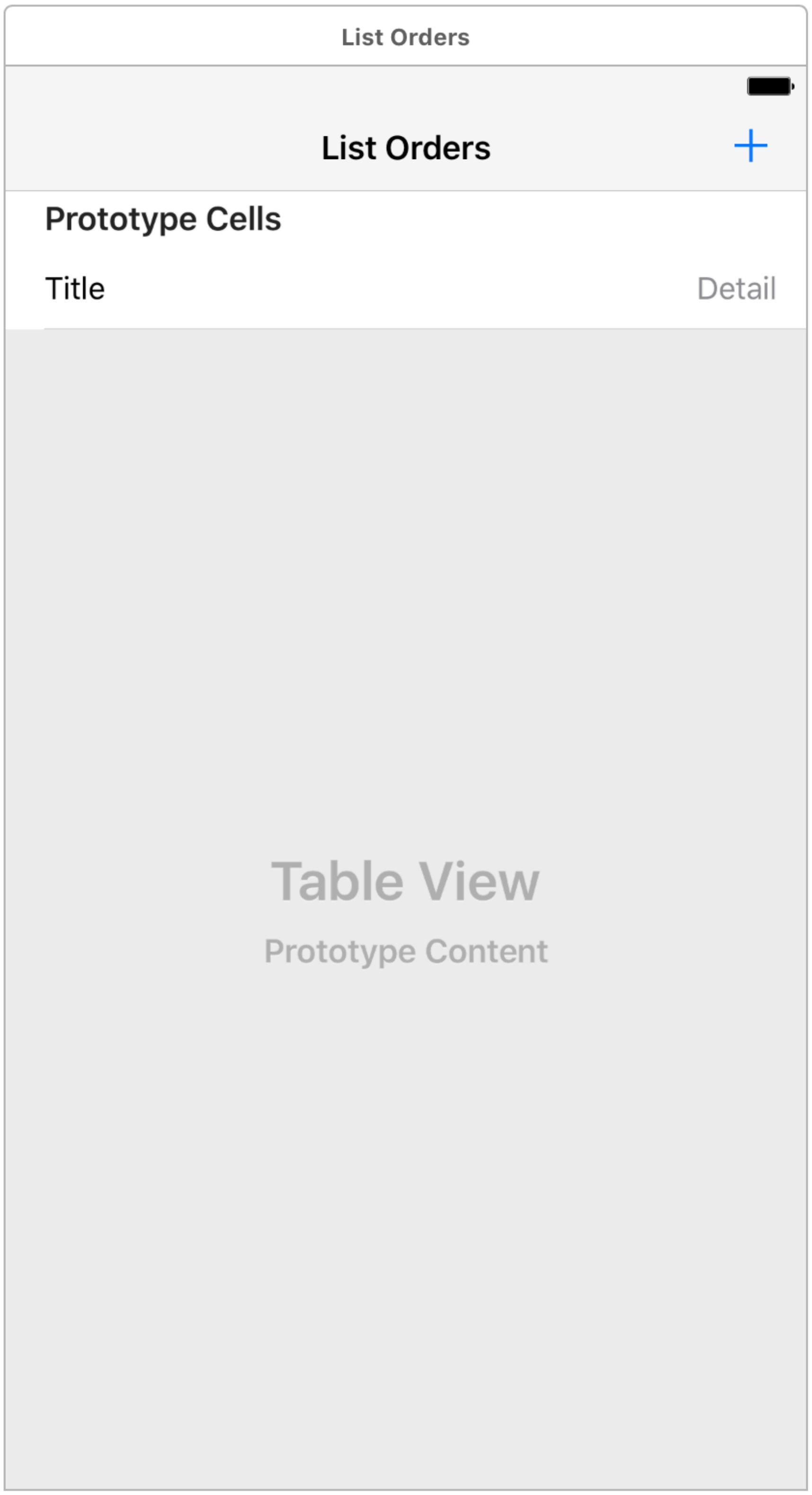
15. The `ListOrders` Scene

Next, we'll tackle the `ListOrders` scene. It's job is simple - showing a list or orders in the all familiar table view. Using traditional MVC, you can probably do this in 15 minutes with one eye closed. Stop! We want to avoid going down the MVC rabbit hole. You're here to learn how to write clean Swift code with architecture in mind. That allows changes to occur without turning your hairs grey.

Design the `ListOrders` Scene in Storyboard and View Controller

Nothing fancy here. It's a simple table view with one dynamic prototype cell that we will use a straight forward name as its identifier - `OrderTableViewCell`. Let's just choose the Right Detail style so I can show the order date on the left and the price on the right. Let's also add a + bar button in the top right corner so that the user can tap this to route to the `CreateOrder` scene.

The finished `ListOrders` design looks something like this:


List Orders
List Orders
+
Prototype Cells
Title
Detail
Table View
Prototype Content

There isn't any IBOutlet or IBAction that you have to write the code for here. Tapping the order table view row and the + button will be handled automatically by iOS. Can't get any simpler than this!

Let's turn to the view controller. When this scene is shown to the user, you want to already have the orders fetched and ready to display. That means `viewDidLoad()`.

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

func fetchOrdersOnLoad()
{
  let request = ListOrders.FetchOrders.Request()
  interactor?.fetchOrders(request: request)
}
```

By now, you probably know what this code dœs. In the `fetchOrdersOnLoad()` method, you first create the request object, and then invoke the interactor's `fetchOrders(request:)` method.

Implement Fetch Orders Business Logic in the Interactor

If you're fetching orders, be it from Core Data or over the network, you'll likely want to do other things to orders. Maybe actually creating and saving a new order?

So it makes sense to extract the fetching code in its own `OrdersWorker`. Below is a trivial implementation:

```
class OrdersWorker
{
  var ordersStore: OrdersStoreProtocol

  init(ordersStore: OrdersStoreProtocol)
```

```
    {
      self.ordersStore = ordersStore
    }

    func fetchOrders(completionHandler: @escaping ([Order]) -> Void)
    {
      ordersStore.fetchOrders { (orders: () throws -> [Order]) ->
  Void in
        do {
          let orders = try orders()
          DispatchQueue.main.async {
            completionHandler(orders)
          }
        } catch {
          DispatchQueue.main.async {
            completionHandler([])
          }
        }
      }
    }
  }
```

One thing to note here is that we use **constructor dependency injection** with the `init(ordersStore:)` initializer, so that we can easily switch to a different data store.

The `fetchOrders(completionHandler:)` method invokes another `fetchOrders(completionHandler:)` method. Huh? Don't worry. It's not a typo. The second one is invoked on an object that conforms to the `OrdersStoreProtocol`, which is show below:

```
  protocol OrdersStoreProtocol
  {
    func fetchOrders(completionHandler: @escaping (() throws -> [Or-
  der]) -> Void)
  }
```

This is the mechanism that allows us to switch to a different type of data store. In the CleanStore sample app on GitHub, you can see I have three data stores:

1. `OrdersMemStore`
2. `OrdersCoreDataStore`
3. `OrdersAPI`

They all conform to the `OrdersStoreProtocol`. That's what makes the dependency injection possible.

If you want to learn the other side of the story about how to write *clean workers*, check out my Clean Swift Mentorship Program.

But let's not distract ourselves. Get back to the interactor.

First, we create the `OrdersWorker` by injecting an `OrdersMemStore`, because it is the fastest and simplest.

Then, in the **`fetchOrders(request:)`** method, we simply invoke the `OrdersWorker` to fetch orders asynchronously:

```
class ListOrdersInteractor: ListOrdersBusinessLogic, ListOrders-
DataStore
{
  var presenter: ListOrdersPresentationLogic?
  var ordersWorker = OrdersWorker(ordersStore: OrdersMemStore())
  var orders: [Order]?

  func fetchOrders(request: ListOrders.FetchOrders.Request)
  {
    ordersWorker.fetchOrders { (orders) -> Void in
      self.orders = orders
      let response = ListOrders.FetchOrders.Response(orders: or-
ders)
      self.presenter?.presentFetchedOrders(response: response)
    }
  }
}
```

When the completion handler block is executed, we would've had our orders ready. At this point, it's a good idea to save the orders array in the interactor, because I can foresee we're going to need to ask for a particular order when the user taps a row. That's right - sending it to the `ShowOrder` scene.

Finally, you just create the response object and pass it onto the presenter by invoking `presentFetchedOrders(response:)`. Look familiar?

Implement Fetch Orders Presentation Logic in the Presenter

So, what is our presentation logic? That's entirely up to you to decide! For starters, I specify what I want in the models:

```
enum ListOrders
{
  enum FetchOrders
  {
    struct Request
    {
    }
    struct Response
    {
      var orders: [Order]
    }
    struct ViewModel
    {
      struct DisplayedOrder
      {
        var id: String
        var date: String
        var email: String
        var name: String
        var total: String
      }
      var displayedOrders: [DisplayedOrder]
    }
  }
}
```

Yes, I know we said we only needed to display the order date and price. This code has more than that. But I want to show you one technique to further isolate and structure your data model. Keeping data independent at each component in the VIP cycle is paramount.

I defined a `DisplayedOrder` struct nested inside the `ViewModel` struct. This means I can't use it outside the `ViewModel` struct. Nice namespacing here. And the members are all strings. Exactly what I want!

This view model has just an array of `DisplayedOrders`. Cool huh?

Let's take a look at the `ListOrdersPresenter`:

```
class ListOrdersPresenter: ListOrdersPresentationLogic
{
  weak var viewController: ListOrdersDisplayLogic?
  let dateFormatter: DateFormatter = {
    let dateFormatter = DateFormatter()
    dateFormatter.dateStyle = .short
    dateFormatter.timeStyle = .none
    return dateFormatter
  }()
  let currencyFormatter: NumberFormatter = {
    let currencyFormatter = NumberFormatter()
    currencyFormatter.numberStyle = .currency
    return currencyFormatter
  }()

  func presentFetchedOrders(response: ListOrders.FetchOrders.Re-
sponse)
  {
    var displayedOrders: [ListOrders.FetchOrders.ViewModel.Dis-
playedOrder] = []
    for order in response.orders {
      let date = dateFormatter.string(from: order.date)
      let total = currencyFormatter.string(from: order.total)
      let displayedOrder = ListOrders.FetchOrders.ViewModel.Dis-
playedOrder(id: order.id!, date: date, email: order.email, name: "\
(order.firstName) \(order.lastName)", total: total!)
      displayedOrders.append(displayedOrder)
    }
    let viewModel = ListOrders.FetchOrders.ViewModel(displayed-
Orders: displayedOrders)
    viewController?.displayFetchedOrders(viewModel: viewModel)
  }
}
```

The `presentFetchedOrders(response:)` method gets the `Order` objects from the response, and turn them into `DisplayedOrder` objects, before stuffing them into the view model and pass it to the view controller.

It dœs this by converting any `Date` and price into strings. I extracted the formatter initialization out of the method and make them constants. The conversion and appending code is pretty straightforward.

Implement Display Fetched Orders Display Logic in the View Controller

Now, back at the view controller where you triggered the fetch orders use case from `viewDidLoad()`.

```
    func displayFetchedOrders(viewModel: ListOrders.FetchOrders.View-
  Model)
    {
      displayedOrders = viewModel.displayedOrders
      tableView.reloadData()
    }
```

When the `displayFetchedOrders(viewModel:)` method is called, it first saves the array of `DisplayedOrders`, and then just ask the table view to `reloadData()`.

I have a few subscribers ask me how to use `NSFetchedResultsController` with table views using Clean Swift because of performance reasons. I'm going to show you how in a future blog post or in the mentorship program. Stay tuned.

To complete the VIP cycle, let's finish the table view data source and delegate methods:

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

```
  override func tableView(_ tableView: UITableView, numberOfRowsIn-
Section section: Int) -> Int
  {
    return displayedOrders.count
  }

  override func tableView(_ tableView: UITableView, cellForRowAt
indexPath: IndexPath) -> UITableViewCell
  {
    let displayedOrder = displayedOrders[indexPath.row]
    var cell = tableView.dequeueReusableCell(withIdentifier: "Or-
derTableViewCell")
    if cell == nil {
      cell = UITableViewCell(style: .value1, reuseIdentifier: "Or-
derTableViewCell")
    }
    cell?.textLabel?.text = displayedOrder.date
    cell?.detailTextLabel?.text = displayedOrder.total
    return cell!
  }
```

You already know this. So I won't waste words here.

16. The `ShowOrder` Scene

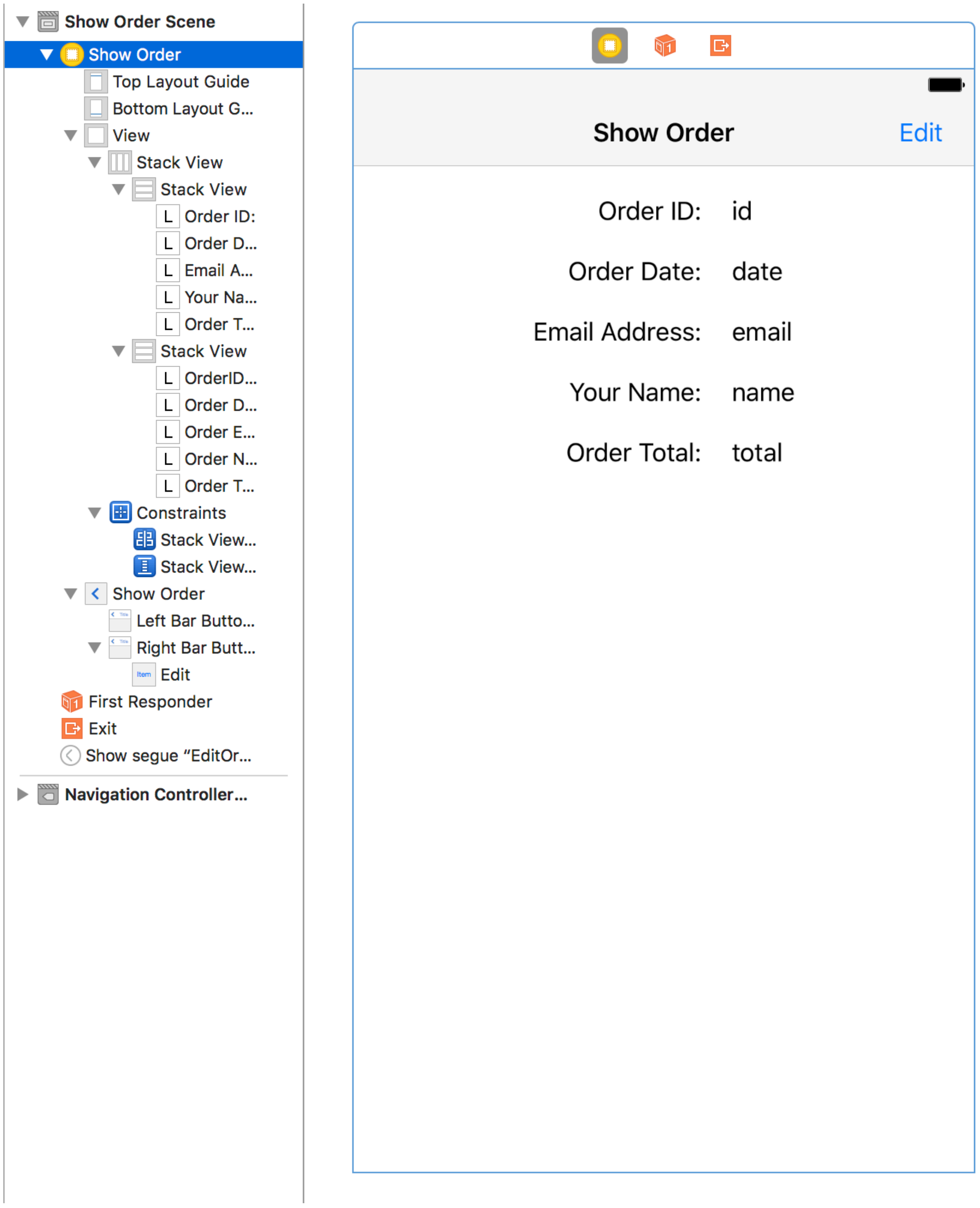
Our third and final scene is the `ShowOrder` scene. It displays the order details to the user.

Design the `ShowOrder` Scene in Storyboard and View Controller

Sticking to the basics. I simply lay out five labels in a stack view to represent the names. Another five labels in another stack view to represent the values. Then I put these two stack views inside yet another stack view. Now they line up perfectly.

I also added an Edit bar button item that when tapped, route to the `CreateOrder` scene so the user can update an existing order.

The final design looks like:



Don't forget to create IBOutlets in the `ShowOrderViewController` and connect them to the second pair of the 5 labels.

In the `viewWillAppear(_:)` method, we invoke `getOrder()`:

```
override func viewWillAppear(_ animated: Bool)
{
  super.viewWillAppear(animated)
  getOrder()
}

func getOrder()
{
  let request = ShowOrder.GetOrder.Request()
  interactor?.getOrder(request: request)
}
```

It simply creates the request object and invoke the interactor's `getOrder(request:)` method.

Why did I choose `viewWillAppear(_:)` instead of `viewDidLoad()` here? Remember the Edit bar button? The intention is to refresh the order details upon returning from editing an order. `viewDidLoad()` is called only once, but `viewWillAppear(_:)` is called every time the view is drawn on the screen. So, in order to trigger the get order use case and make sure we show the latest order details, `viewWillAppear(_:)` is a better fit.

Implement Get Order Business Logic in the Interactor

Let's just dive into the `ShowOrderInteractor`:

```
class ShowOrderInteractor: ShowOrderBusinessLogic, ShowOrderDataS-
tore
{
  var presenter: ShowOrderPresentationLogic?
  var order: Order!

  func getOrder(request: ShowOrder.GetOrder.Request)
  {
    let response = ShowOrder.GetOrder.Response(order: order)
    presenter?.presentOrder(response: response)
  }
}
```

The `getOrder(request:)` method simply creates the response object with the `order` variable, and then invokes the presenter's `presentOrder(response:)` method. Simple, right? But wait, where dœs the `order` variable come from? Who populate it?

Think back on how the user could wind up in this scene? The user must've tapped an order in the `ListOrders` scene. That's the only way.

So the `ListOrders` scene has to pass the selected order to the `ShowOrder` scene. How dœs this happen? Routing. A little patience here is required, as you'll see when we finish this VIP cycle.

Implement Present Order Presentation Logic in the Presenter

Similar to the `ListOrdersPresenter`, the `ShowOrderPresenter` is pretty straightforward too.

```
class ShowOrderPresenter: ShowOrderPresentationLogic
{
  weak var viewController: ShowOrderDisplayLogic?

  let dateFormatter: DateFormatter = {
    let dateFormatter = DateFormatter()
    dateFormatter.dateStyle = .short
    dateFormatter.timeStyle = .none
    return dateFormatter
  }()

  let currencyFormatter: NumberFormatter = {
    let currencyFormatter = NumberFormatter()
    currencyFormatter.numberStyle = .currency
    return currencyFormatter
  }()

  func presentOrder(response: ShowOrder.GetOrder.Response)
  {
    let order = response.order
    let date = dateFormatter.string(from: order.date)
    let total = currencyFormatter.string(from: order.total)!
    let displayedOrder = ShowOrder.GetOrder.ViewModel.Displayed-
Order(id: order.id!, date: date, email: order.email, name: "\(or-
der.firstName) \(order.lastName)", total: total)

    let viewModel = ShowOrder.GetOrder.ViewModel(displayedOrder:
displayedOrder)
    viewController?.displayOrder(viewModel: viewModel)
  }
}
```

I extracted the formatters, and the `presentOrder(response:)` method retrieves the `order` from the response object, and converts everything to strings. Finally, it asks the view controller to display the order by calling `displayOrder(viewModel:)`.

In fact, it is even simpler than `ListOrdersPresenter`. Instead of an array of `DisplayedOrder`, you just have one `DisplayedOrder` to show. So you don't have to write code to append.

Here're the models:

```
enum ShowOrder
{
  enum GetOrder
  {
    struct Request
    {
    }
    struct Response
    {
      var order: Order
    }
    struct ViewModel
    {
      struct DisplayedOrder
      {
        var id: String
        var date: String
        var email: String
        var name: String
        var total: String
      }
```

```
        var displayedOrder: DisplayedOrder
      }
    }
  }
```

Implement Display Order Display Logic in the View Controller

To finish up, let's look at the `displayOrder(viewModel:)` method:

```
func displayOrder(viewModel: ShowOrder.GetOrder.ViewModel)
{
  let displayedOrder = viewModel.displayedOrder
  orderIDLabel.text = displayedOrder.id
  orderDateLabel.text = displayedOrder.date
  orderEmailLabel.text = displayedOrder.email
  orderNameLabel.text = displayedOrder.name
  orderTotalLabel.text = displayedOrder.total
}
```

It just extracts the order details from the view model, and sets the label text. All the presentation work was already done by the presenter. I don't know about you, but I love these simple assignment statements. Can't get any simpler than this.

17. Routing from the `ListOrders` Scene

First, one minute of conceptual stuff before the actual code.

Routing is a 3-step process, which are carried out by the router. These steps are:

1. Getting a hold on the destination - `routeToNextScene(segue:)`
2. Passing data to the destination - `passDataToNextScene(source:destination:)`
3. Navigating to the destination - `navigateToNextScene(source:destination:)`

For the `ListOrders` scene, there are two outgoing routes:

- Tapping the Add button should route to the `CreateOrder` scene to create a new order - `routeToCreateOrder(segue:)`
- Tapping an order row in the table view should route to the `ShowOrder` scene to show the order details - `routeToShowOrder(segue:)`

Route to the `CreateOrder` Scene

When the user taps the Add button, the app should route to the `CreateOrder` scene.

Step 1.
Getting a hold on the destination (`routeToCreateOrder(segue:)`. You can get the destination view controller from the segue as usual, and the destination data store from the view controller's router. Next, you make calls to steps 2 and 3.

```
func routeToCreateOrder(segue: UIStoryboardSegue?)
```

```
  {
    let destinationVC = segue!.destination as! CreateOrderViewCon-
troller
    var destinationDS = destinationVC.router!.dataStore!
    passDataToCreateOrder(source: dataStore!, destination: &desti-
nationDS)
    navigateToCreateOrder(source: viewController!, destination:
destinationVC)
  }
```

Step 2.

Passing data to the destination (`passDataToCreateOrder(source:destination)`). There is no data to be passed, so the method is empty.

```
  func passDataToCreateOrder(source: ListOrdersDataStore, destina-
tion: inout CreateOrderDataStore)
  {
  }
```

Step 3.

Navigating to the destination (`navigateToCreateOrder(source:destination)`). Navigation is already taken care of by the segue. So nothing to do here.

```
  func navigateToCreateOrder(source: ListOrdersViewController, des-
tination: CreateOrderViewController)
  {
  }
```

Why is everything empty? Because this route is triggered by the storyboard segue automatically and there is no data to be passed. You could argue that we didn't have to write all this empty code. You're right. We didn't have to. But I did it to show you the complete routing process, and for consistency's sake. However, later on, you'll see this same 3-step routing process also help facilitate programmatic routing. So it's not too bad that we wrote this code.

Route to the `ShowOrder` Scene

When the user taps an order in the table view row, the app should pass the selected order and navigate to the `ShowOrder` scene.

Step 1.

Getting a hold on the destination (`routeToShowOrder(segue:)`. You can get the destination view controller from the segue as usual, and the destination data store from the view controller's router. Next, you make calls to steps 2 and 3.

```
  func routeToShowOrder(segue: UIStoryboardSegue?)
  {
    let destinationVC = segue!.destination as! ShowOrderViewCon-
troller
    var destinationDS = destinationVC.router!.dataStore!
    passDataToShowOrder(source: dataStore!, destination: &destina-
tionDS)
    navigateToShowOrder(source: viewController!, destination: des-
tinationVC)
  }
```

Step 2.

Passing data to the destination (`passDataToShowOrder(source:destination)`). The order of the selected table view row is passed to the destination data store.

```
  func passDataToShowOrder(source: ListOrdersDataStore, destina-
tion: inout ShowOrderDataStore)
  {
    let selectedRow = viewController?.tableView.indexPathForSelect-
edRow?.row
    destination.order = source.orders?[selectedRow!]
  }
```

Step 3.

Navigating to the destination (`navigateToShowOrder(source:destination`). Navigation is already taken care of by the segue. So we have nothing to do here.

```
  func navigateToShowOrder(source: ListOrdersViewController, desti-
nation: ShowOrderViewController)
  {
    }
```

There's something more interesting in this route here. You need to pass the selected order to the `ShowOrder` scene. So the practice of getting those references early becomes handy here.

18. Routing from the `ShowOrder` Scene

For the `ShowOrder` scene, there is just one outgoing route:

- Tapping the Edit button should route to the `CreateOrder` scene to update an existing order - `routeToEditOrder(segue:)`

Route to the `CreateOrder` Scene

When the user taps the Edit button, the app should route to the `CreateOrder` scene, passing along the order.

Step 1.
Getting a hold on the destination (`routeToEditOrder(segue:)`. You can get the destination view controller from the segue as usual, and the destination data store from the view controller's router.

```
  func routeToEditOrder(segue: UIStoryboardSegue?)
  {
    let destinationVC = segue!.destination as! CreateOrderViewCon-
troller
    var destinationDS = destinationVC.router!.dataStore!
    passDataToEditOrder(source: dataStore!, destination: &destina-
tionDS)
    navigateToEditOrder(source: viewController!, destination: des-
tinationVC)
  }
```

Step 2.
Passing data to the destination (`passDataToEditOrder(source:destination)`). The order being displayed here is passed to the destination data store to be edited.

```
    func passDataToEditOrder(source: ShowOrderDataStore, destination:
  inout CreateOrderDataStore)
    {
      destination.orderToEdit = source.order
    }
```

Step 3.

Navigating to the destination (`navigateToEditOrder(source:destination`). Navigation is already taken care of by the segue. So we have nothing to do here.

```
    func navigateToEditOrder(source: ShowOrderViewController, desti-
  nation: CreateOrderViewController)
    {
    }
```

By now, you're probably getting familiar with this routing process. You should already know what the above code do.

19. More Use Cases for the `CreateOrder` Scene

Now that we've added the `ListOrders` and `ShowOrder` scenes to the app, we also have three new use cases for the `CreateOrder` scene:

- Actually create the new order
- Populate the order form with the existing order details
- Actually update the existing order

We'll finish the app by implementing these use cases. When you're done, you'll have architected your first iOS app using Clean Swift. Bye bye MVC.

Hook up the IBAction in the View Controller

When `saveButtonTapped(_:)` is invoked, that's when you want to trigger this use case.

```
    @IBAction func saveButtonTapped(_ sender: Any)
    {
      // MARK: Contact info
      let firstName = firstNameTextField.text!
      let lastName = lastNameTextField.text!
      let phone = phoneTextField.text!
      let email = emailTextField.text!

      // MARK: Payment info
      let billingAddressStreet1 = billingAddressStreet1TextField.-
  text!
      let billingAddressStreet2 = billingAddressStreet2TextField.-
  text!
      let billingAddressCity = billingAddressCityTextField.text!
      let billingAddressState = billingAddressStateTextField.text!
      let billingAddressZIP = billingAddressZIPTextField.text!

      let paymentMethodCreditCardNumber = creditCardNumberTextField.-
  text!
      let paymentMethodCVV = ccvTextField.text!
      let paymentMethodExpirationDate = expirationDatePicker.date
      let paymentMethodExpirationDateString = ""

      // MARK: Shipping info
      let shipmentAddressStreet1 = shipmentAddressStreet1TextField.-
  text!
      let shipmentAddressStreet2 = shipmentAddressStreet2TextField.-
  text!
      let shipmentAddressCity = shipmentAddressCityTextField.text!
      let shipmentAddressState = shipmentAddressStateTextField.text!
      let shipmentAddressZIP = shipmentAddressZIPTextField.text!

      let shipmentMethodSpeed = shippingMethodPicker.selectedRow(in-
  Component: 0)
      let shipmentMethodSpeedString = ""

      // MARK: Misc
      let id: String? = nil
      let date = Date()
      let total = NSDecimalNumber.notANumber

      let request = CreateOrder.CreateOrder.Request(orderFormFields:
  CreateOrder.OrderFormFields(firstName: firstName, lastName: last-
  Name, phone: phone, email: email, billingAddressStreet1: billingAd-
  dressStreet1, billingAddressStreet2: billingAddressStreet2,
  billingAddressCity: billingAddressCity, billingAddressState:
  billingAddressState, billingAddressZIP: billingAddressZIP, payment-
  MethodCreditCardNumber: paymentMethodCreditCardNumber, payment-
  MethodCVV: paymentMethodCVV, paymentMethodExpirationDate: payment-
  MethodExpirationDate, paymentMethodExpirationDateString: payment-
  MethodExpirationDateString, shipmentAddressStreet1: shipmentAd-
  dressStreet1, shipmentAddressStreet2: shipmentAddressStreet2, ship-
  mentAddressCity: shipmentAddressCity, shipmentAddressState: ship-
  mentAddressState, shipmentAddressZIP: shipmentAddressZIP, shipment-
  MethodSpeed: shipmentMethodSpeed, shipmentMethodSpeedString: ship-
  mentMethodSpeedString, id: id, date: date, total: total))
      interactor?.createOrder(request: request)
    }
```

This method prepares the request object and invoke the interactor.

But this method is pretty long, but do you need to refactor? No, it's long only because the order form is really long. I could have also skipped all those constants and use the `text` property directly in the initializer. But I like to create constants for my data first because I often find the need to tweak them before using them in the initializer. This is especially true in the presenter. This is why I value the thought process a lot more than some vanity metrics.

Implement Create Order Business Logic in the Interactor

When the user taps the Save button, your app needs to persist the order. Let's get back to the VIP cycle.

```
class CreateOrderInteractor: CreateOrderBusinessLogic, CreateOrder-
DataStore
{
  var ordersWorker = OrdersWorker(ordersStore: OrdersMemStore())

  func createOrder(request: CreateOrder.CreateOrder.Request)
  {
    let orderToCreate = buildOrderFromOrderFormFields(request.or-
derFormFields)
    ordersWorker.createOrder(orderToCreate: orderToCreate) { (or-
der: Order?) in
      self.orderToEdit = order
      let response = CreateOrder.CreateOrder.Response(order: order)
      self.presenter?.presentCreatedOrder(response: response)
    }
  }

  private func buildOrderFromOrderFormFields(_ orderFormFields:
CreateOrder.OrderFormFields) -> Order
  {
    let billingAddress = Address(street1: orderFormFields.billing-
AddressStreet1, street2: orderFormFields.billingAddressStreet2,
city: orderFormFields.billingAddressCity, state: orderFormFields.-
billingAddressState, zip: orderFormFields.billingAddressZIP)

    let paymentMethod = PaymentMethod(creditCardNumber: orderForm-
Fields.paymentMethodCreditCardNumber, expirationDate: orderForm-
Fields.paymentMethodExpirationDate, cvv: orderFormFields.payment-
MethodCVV)

    let shipmentAddress = Address(street1: orderFormFields.shipmen-
tAddressStreet1, street2: orderFormFields.shipmentAddressStreet2,
city: orderFormFields.shipmentAddressCity, state: orderFormFields.-
shipmentAddressState, zip: orderFormFields.shipmentAddressZIP)

    let shipmentMethod = ShipmentMethod(speed: ShipmentMethod.Ship-
pingSpeed(rawValue: orderFormFields.shipmentMethodSpeed)!)

    return Order(firstName: orderFormFields.firstName, lastName:
orderFormFields.lastName, phone: orderFormFields.phone, email: or-
derFormFields.email, billingAddress: billingAddress, paymentMethod:
paymentMethod, shipmentAddress: shipmentAddress, shipmentMethod:
shipmentMethod, id: orderFormFields.id, date: orderFormFields.date,
total: orderFormFields.total)
  }
}
```

The `createOrder(request:)` method first calls the private `buildOrderFromOrderFormFields(_:)` method to build the `Order` object, and then asks the `OrdersWorker` to create the order asynchronously.

We inject the `OrdersMemStore` to the `OrdersWorker`'s initializer just to get things going in the simplest way. You can take a look at the three different data stores I created for this sample app on GitHub.

After the order is created, the completion block is called. We first save the order for routing purpose later, as well as for another use case where the user is editing an existing order so that we can check it and execute slightly different code.

Finally, you create the response object and invoke the presenter. `self` is required here because it's inside a closure.

Implement Create Order Presentation Logic in the Presenter

The presenter has nothing to do except passing it along to the view controller.

```
  func presentCreatedOrder(response: CreateOrder.CreateOrder.Re-
sponse)
  {
    let viewModel = CreateOrder.CreateOrder.ViewModel(order: re-
sponse.order)
    viewController?.displayCreatedOrder(viewModel: viewModel)
  }
```

Implement Create Order Display Logic in the View Controller

To complete the VIP cycle back at the view controller, we implement the `displayCreatedOrder(viewModel:)` method as follows:

```
  func displayCreatedOrder(viewModel: CreateOrder.CreateOrder.View-
Model)
  {
    if viewModel.order != nil {
      router?.routeToListOrders(segue: nil)
    } else {
      showOrderFailureAlert(title: "Failed to create order", mes-
sage: "Please correct your order and submit again.")
    }
  }
```

If the order creation is successful, we should get an order inside the view model. The `CreateOrder` scene is no longer useful, so we route back to the `ListOrders` scene by invoking the router directly using programmatic route and pass nil for the `segue` parameter.

If the order fails to be created, we call `showOrderFailureAlert(title:message:)` to display an alert to the user, so he can correct any mistakes and try again.

Populate the Order Form

The same `CreateOrder` scene is used for both new and existing orders. When we have an order, we don't want the user to enter all the details again. We have the data. So we'll call `showOrderToEdit()` to fill in the order form on `viewDidLoad()`.

```
override func viewDidLoad()
{
  super.viewDidLoad()
  configurePickers()
  showOrderToEdit()
}

func showOrderToEdit()
{
  let request = CreateOrder.EditOrder.Request()
  interactor?.showOrderToEdit(request: request)
}
```

The `showOrderToEdit()` method simply triggers the use case on the interactor.

Implement Show Order To Edit Business Logic in the Interactor

This use case looks like this in the interactor:

```
func showOrderToEdit(request: CreateOrder.EditOrder.Request)
{
  if let orderToEdit = orderToEdit {
    let response = CreateOrder.EditOrder.Response(order: orderTo-
Edit)
    presenter?.presentOrderToEdit(response: response)
  }
```

```
    }
```

We first check for a valid `orderToEdit` before we create the response and invoke the presenter. That's familiar.

Another deceivingly interesting tidbit lurking here. What if `orderToEdit` is nil, as in the case for a new order? We do nothing. That's right. You don't necessarily have to invoke the presenter. You can cut off the VIP cycle anywhere it makes sense.

You normally wouldn't. Why would you even have a use case if you don't need the result back? In most cases, we do want to send a *success* or *failure* result back to complete the VIP cycle. But in this particular case, it's safe to silently stop the use case because it's neither right nor wrong to not have an existing order.

Implement Show Order To Edit Presentation Logic in the Presenter

The presenter is quite simple and familiar:

```
    func presentOrderToEdit(response: CreateOrder.EditOrder.Response)
    {
      let orderToEdit = response.order
      let viewModel = CreateOrder.EditOrder.ViewModel(
        orderFormFields: CreateOrder.OrderFormFields(
          firstName: orderToEdit.firstName,
          lastName: orderToEdit.lastName,
          phone: orderToEdit.phone,
          email: orderToEdit.email,
          billingAddressStreet1: orderToEdit.billingAddress.street1,
          billingAddressStreet2:
  (orderToEdit.billingAddress.street2 != nil ? orderToEdit.billingAd-
  dress.street2! : ""),
          billingAddressCity: orderToEdit.billingAddress.city,
          billingAddressState: orderToEdit.billingAddress.state,
          billingAddressZIP: orderToEdit.billingAddress.zip,
          paymentMethodCreditCardNumber: orderToEdit.paymentMethod.-
  creditCardNumber,
          paymentMethodCVV: orderToEdit.paymentMethod.cvv,
          paymentMethodExpirationDate: orderToEdit.paymentMethod.ex-
  pirationDate,
          paymentMethodExpirationDateString:
  dateFormatter.string(from: orderToEdit.paymentMethod.expiration-
  Date),
          shipmentAddressStreet1: orderToEdit.shipmentAd-
  dress.street1,
          shipmentAddressStreet2: orderToEdit.shipmentAddress.street2
  != nil ? orderToEdit.shipmentAddress.street2! : "",
          shipmentAddressCity: orderToEdit.shipmentAddress.city,
          shipmentAddressState: orderToEdit.shipmentAddress.state,
          shipmentAddressZIP: orderToEdit.shipmentAddress.zip,
          shipmentMethodSpeed: orderToEdit.shipmentMethod.speed.raw-
  Value,
          shipmentMethodSpeedString: orderToEdit.shipmentMethod.to-
  String(),
          id: orderToEdit.id,
          date: orderToEdit.date,
          total: orderToEdit.total
        )
      )
      viewController?.displayOrderToEdit(viewModel: viewModel)
    }
```

It dœs some conversions on the `Order` object before passing it to the view controller for the last mile.

Implement Show Order To Edit Display Logic in the View Controller

Back at the view controller:

```
  func displayOrderToEdit(viewModel: CreateOrder.EditOrder.ViewMod-
el)
  {
    let orderFormFields = viewModel.orderFormFields
    firstNameTextField.text = orderFormFields.firstName
    lastNameTextField.text = orderFormFields.lastName
    phoneTextField.text = orderFormFields.phone
    emailTextField.text = orderFormFields.email

    billingAddressStreet1TextField.text = orderFormFields.billing-
AddressStreet1
```

```
    billingAddressStreet2TextField.text = orderFormFields.billing-
AddressStreet2
    billingAddressCityTextField.text = orderFormFields.billingAd-
dressCity
    billingAddressStateTextField.text = orderFormFields.billingAd-
dressState
    billingAddressZIPTextField.text = orderFormFields.billingAd-
dressZIP

    creditCardNumberTextField.text = orderFormFields.paymentMethod-
CreditCardNumber
    ccvTextField.text = orderFormFields.paymentMethodCVV

    shipmentAddressStreet1TextField.text = orderFormFields.shipmen-
tAddressStreet1
    shipmentAddressStreet2TextField.text = orderFormFields.shipmen-
tAddressStreet2
    shipmentAddressCityTextField.text = orderFormFields.shipmentAd-
dressCity
    shipmentAddressStateTextField.text = orderFormFields.shipment-
AddressState
    shipmentAddressZIPTextField.text = orderFormFields.shipmentAd-
dressZIP

    shippingMethodPicker.selectRow(orderFormFields.shipmentMethod-
Speed, inComponent: 0, animated: true)
    shippingMethodTextField.text = orderFormFields.shipmentMethod-
SpeedString

    expirationDatePicker.date = orderFormFields.paymentMethodExpi-
rationDate
    expirationDateTextField.text = orderFormFields.paymentMethodEx-
pirationDateString
  }
```

We simply set some text field labels, and select a row in the `shippingMethodPicker`. Isn't it nice that everything is already in strings? I love assignment statements.

Modify `saveButtonTapped(_:)` in the View Controller

The same Save button in the `CreateOrder` scene is used to create a new order and update an existing order. That means we need to modify the `saveButtonTapped(_:)` method to account for both use cases.

```
    @IBAction func saveButtonTapped(_ sender: Any)
    {
      // MARK: Contact info
      let firstName = firstNameTextField.text!
      let lastName = lastNameTextField.text!
      let phone = phoneTextField.text!
      let email = emailTextField.text!

      // MARK: Payment info
      let billingAddressStreet1 = billingAddressStreet1TextField.-
  text!
      let billingAddressStreet2 = billingAddressStreet2TextField.-
  text!
      let billingAddressCity = billingAddressCityTextField.text!
      let billingAddressState = billingAddressStateTextField.text!
      let billingAddressZIP = billingAddressZIPTextField.text!

      let paymentMethodCreditCardNumber = creditCardNumberTextField.-
  text!
      let paymentMethodCVV = ccvTextField.text!
      let paymentMethodExpirationDate = expirationDatePicker.date
      let paymentMethodExpirationDateString = ""

      // MARK: Shipping info
      let shipmentAddressStreet1 = shipmentAddressStreet1TextField.-
  text!
      let shipmentAddressStreet2 = shipmentAddressStreet2TextField.-
  text!
      let shipmentAddressCity = shipmentAddressCityTextField.text!
      let shipmentAddressState = shipmentAddressStateTextField.text!
      let shipmentAddressZIP = shipmentAddressZIPTextField.text!

      let shipmentMethodSpeed = shippingMethodPicker.selectedRow(in-
  Component: 0)
      let shipmentMethodSpeedString = ""

      // MARK: Misc
      var id: String? = nil
      var date = Date()
      var total = NSDecimalNumber.notANumber

      if let orderToEdit = interactor?.orderToEdit {
        id = orderToEdit.id
        date = orderToEdit.date
        total = orderToEdit.total
        let request = CreateOrder.UpdateOrder.Request(orderForm-
  Fields: CreateOrder.OrderFormFields(firstName: firstName, lastName:
  lastName, phone: phone, email: email, billingAddressStreet1:
  billingAddressStreet1, billingAddressStreet2: billingAddress-
  Street2, billingAddressCity: billingAddressCity, billingAddress-
  State: billingAddressState, billingAddressZIP: billingAddressZIP,
  paymentMethodCreditCardNumber: paymentMethodCreditCardNumber, pay-
  mentMethodCVV: paymentMethodCVV, paymentMethodExpirationDate: pay-
  mentMethodExpirationDate, paymentMethodExpirationDateString: pay-
  mentMethodExpirationDateString, shipmentAddressStreet1: shipmentAd-
  dressStreet1, shipmentAddressStreet2: shipmentAddressStreet2, ship-
  mentAddressCity: shipmentAddressCity, shipmentAddressState: ship-
  mentAddressState, shipmentAddressZIP: shipmentAddressZIP, shipment-
  MethodSpeed: shipmentMethodSpeed, shipmentMethodSpeedString: ship-
  mentMethodSpeedString, id: id, date: date, total: total))
        interactor?.updateOrder(request: request)
      } else {
        let request = CreateOrder.CreateOrder.Request(orderForm-
  Fields: CreateOrder.OrderFormFields(firstName: firstName, lastName:
  lastName, phone: phone, email: email, billingAddressStreet1:
  billingAddressStreet1, billingAddressStreet2: billingAddress-
  Street2, billingAddressCity: billingAddressCity, billingAddress-
  State: billingAddressState, billingAddressZIP: billingAddressZIP,
  paymentMethodCreditCardNumber: paymentMethodCreditCardNumber, pay-
  mentMethodCVV: paymentMethodCVV, paymentMethodExpirationDate: pay-
  mentMethodExpirationDate, paymentMethodExpirationDateString: pay-
  mentMethodExpirationDateString, shipmentAddressStreet1: shipmentAd-
  dressStreet1, shipmentAddressStreet2: shipmentAddressStreet2, ship-
  mentAddressCity: shipmentAddressCity, shipmentAddressState: ship-
  mentAddressState, shipmentAddressZIP: shipmentAddressZIP, shipment-
  MethodSpeed: shipmentMethodSpeed, shipmentMethodSpeedString: ship-
  mentMethodSpeedString, id: id, date: date, total: total))
        interactor?.createOrder(request: request)
      }
    }
```

For an existing order, you can still get most of the order details from the form fields. But you also need to set the `id`, `date`, and `total` in the first conditional branch. This time around, you invoke the `updateOrder(request:)` instead of `createOrder(request:)` method of the interactor.

Implement Update Order Business Logic in the Interactor

Similar to creating a new order, we can delegate the task of updating an existing order to the `OrdersWorker`.

```
  func updateOrder(request: CreateOrder.UpdateOrder.Request)
  {
    let orderToUpdate = buildOrderFromOrderFormFields(request.or-
derFormFields)
    ordersWorker.updateOrder(orderToUpdate: orderToUpdate) { (or-
der) in
      self.orderToEdit = order
      let response = CreateOrder.UpdateOrder.Response(order: order)
      self.presenter?.presentUpdatedOrder(response: response)
    }
  }
```

Again, we take advantage of the `buildOrderFromOrderFormFields(_:)` method to create the order first. The rest is history.

Implement Update Order Presentation Logic in the Presenter

The presenter has nothing to do except passing it along to the view controller.

```
  func presentUpdatedOrder(response: CreateOrder.UpdateOrder.Re-
sponse)
  {
    let viewModel = CreateOrder.UpdateOrder.ViewModel(order: re-
sponse.order)
    viewController?.displayUpdatedOrder(viewModel: viewModel)
  }
```

Implement Update Order Display Logic in the View Controller

The final step is similar to creating an order. It takes care of both success and failure:

```
  func displayUpdatedOrder(viewModel: CreateOrder.UpdateOrder.View-
Model)
  {
    if viewModel.order != nil {
      router?.routeToShowOrder(segue: nil)
    } else {
      showOrderFailureAlert(title: "Failed to update order", mes-
sage: "Please correct your order and submit again.")
    }
  }
```

The app shows an alert to the user if the order cannot be updated. On success, you route back to the `ShowOrder` scene.

Remember, when creating a new order, the user comes from the `ListOrders` scene. When updating an existing order, the user comes from the `ShowOrder` scene.

20. Routing from the `CreateOrder` Scene

For the `CreateOrder` scene, there are two outgoing routes. Both are triggered by tapping the Save button, depending on whether it's a new or existing order.

- Tapping the Save button when creating a new order should route to the `ListOrders` scene - `routeToCreateOrder(segue:)`
- Tapping the Save button when updating an existing order should route to the `ShowOrder` scene - `routeToShowOrder(segue:)`

Route to the `ListOrders` Scene

If the user is creating a new order, when he taps the Save button, the app should persist the new order and route to the `ListOrders` scene, which will refresh with the new order being shown.

Step 1.
Getting a hold on the destination (`routeToListOrders(segue:)`. Since you already know the source view controller needs to be popped off the navigation controller stack, the destination view controller is simply the view controller beneath the top view controller.

```
    func routeToListOrders(segue: UIStoryboardSegue?)
    {
      let destinationVC = segue!.destination as! ListOrdersViewCon-
  troller
      var destinationDS = destinationVC.router!.dataStore!
      passDataToListOrders(source: dataStore!, destination: &destina-
  tionDS)
      navigateToListOrders(source: viewController!, destination: des-
  tinationVC)
    }
```

Step 2.

Passing data to the destination (`passDataToListOrders(source:destination)`). There is no data to be passed, so the method is empty.

```
  func passDataToListOrders(source: CreateOrderDataStore, destina-
tion: inout ListOrdersDataStore)
  {
  }
```

Step 3.

Navigating to the destination (`navigateToListOrders(source:destination`). After the new order is saved, we pop the top view controller off the navigation controller stack.

```
  func navigateToListOrders(source: CreateOrderViewController, des-
tination: ListOrdersViewController)
  {
    source.navigationController?.popViewController(animated: true)
  }
```

Passing data here is already familiar. But we actually see this routing process also works for *backward* routes too. In this case, we pop the `CreateOrderViewController` off the navigation controller stack.

Route to the `ShowOrder` Scene

If the user is updating an existing order, when he taps the Save button, the app should persist the updated order and route to the `ShowOrder` scene, which will refresh with the updated order details.

Step 1.

Getting a hold on the destination (`routeToShowOrder(segue:)`. Since you already know the source view controller needs to be popped off the navigation controller stack, the destination view controller is simply the view controller beneath the top view controller.

```
    func routeToShowOrder(segue: UIStoryboardSegue?)
    {
      let destinationVC = segue!.destination as! ShowOrderViewCon-
troller
      var destinationDS = destinationVC.router!.dataStore!
      passDataToShowOrder(source: dataStore!, destination: &destina-
tionDS)
      navigateToShowOrder(source: viewController!, destination: des-
tinationVC)
    }
```

Step 2.

Passing data to the destination (`passDataToShowOrder(source:destination)`). The updated order is passed to the destination data store so that the order details are refreshed.

```
    func passDataToShowOrder(source: CreateOrderDataStore, destina-
tion: inout ShowOrderDataStore)
    {
      destination.order = source.orderToEdit
    }
```

Step 3.

Navigating to the destination (`navigateToShowOrder(source:destination`). After the new order is saved, we pop the top view controller off the navigation controller stack.

```
    func navigateToShowOrder(source: CreateOrderViewController, des-
tination: ShowOrderViewController)
    {
      source.navigationController?.popViewController(animated: true)
    }
```

The navigation step is the same here. However, there's something new in passing data that can easily be overlooked. On the surface, it simply sets the order of the `ShowOrder` scene to the updated order in the `CreateOrder` scene.

While that's true, but have you noticed this is also a *backward* route?

Before Clean Swift, you had to define a protocol and assign a delegate in order to pass data backward. But now, the new data store takes care of this without having to use delegation! In fact, passing data works the same way whether it's a forward or backward route.

You can save the delegation pattern for real delegated tasks. This cuts down the number of protocol conformance significantly.

21. Programmatic Routing

The `routeToSomewhere(segue:)` methods has an optional `segue` parameter. For segue routes, triggered automatically by iOS or manually by you, will have `segue` populated, and you can skip the navigation step.

For programmatic routes, because you're taking control to gain the flexibility, you need to account for the navigation step. If not, nothing happens. But as you can see, this can be as easy as calling `UIViewController`'s `show(_:sender:)` method.

Since you're manually triggering the route by invoking `routeToSomewhere(segue:)`, you pass nil for the `segue` parameter. Inside `routeToSomewhere(segue:)`, you need to grab the required references a little bit differently:

For example, for the `ListOrders` scene, in the `routeToShowOrder(segue:)` method:

```
  func routeToShowOrder(segue: UIStoryboardSegue?)
  {
    let destinationVC = viewController?.storyboard?.instantiate-
ViewController(withIdentifier: "ShowOrderViewController") as! Show-
OrderViewController
    var destinationDS = destinationVC.router!.dataStore!
    passDataToShowOrder(source: dataStore!, destination: &destina-
tionDS)
    navigateToShowOrder(source: viewController!, destination: des-
tinationVC)
  }
```

Instead of conveniently grabbing the destination view controller from the segue, you want to instantiate it from the storyboard. If you're using nibs, you can just as easily instantiate it from the nib. If storyboard is just not your thing, you can invoke the destination view controller's initializer directly. You get the idea.

The rest is pretty much the same.

If the default native presentation works for you, the navigation step is pretty straightforward as well:

```
  func navigateToShowOrder(source: ListOrdersViewController, desti-
nation: ShowOrderViewController)
  {
    source.show(destination, sender: nil)
  }
```

Again, don't leave it empty, or your destination scene won't be presented!

To put all these together, your `routeToShowOrder(segue:)` method may look like the following:

```
  func routeToShowOrder(segue: UIStoryboardSegue?)
  {
    if let segue = segue {
      let destinationVC = segue.destination as! ShowOrderViewCon-
troller
      var destinationDS = destinationVC.router!.dataStore!
      passDataToShowOrder(source: dataStore!, destination: &desti-
nationDS)
    } else {
      let destinationVC = viewController?.storyboard?.instantiate-
ViewController(withIdentifier: "ShowOrderViewController") as! Show-
OrderViewController
      var destinationDS = destinationVC.router!.dataStore!
      passDataToShowOrder(source: dataStore!, destination: &desti-
nationDS)
      navigateToShowOrder(source: viewController!, destination:
destinationVC)
    }
  }
```

This route now works in all scenarios:

- Segue route triggered automatically
- Segue route triggered manually
- Custom route triggered programmatically

It’s all-you-can-eat. So pick whatever you like.

22. Recap

You have learned:

- The *massive view controller* problem is real.
- MVC is not a suitable architecture for an iOS app.
- Design patterns and refactoring are just techniques, not architectures.
- Architecture first. Then unit testing.
- Testing is only possible with a sound architecture.
- Good architecture makes making changes easy.
- Organize your code with intents.
- Understand the Clean Swift architecture, **VIP cycle**, and **data independence**.
- Break down use cases into **business**, **presentation**, and **display** logic.
- Use Clean Swift to implement the use cases.
- The new **routing** and **data passing** mechanism, with segues or programmatically.

And a reminder on the VIP cycle:

- The view controller accepts an user event, constructs a **request** object, sends it to the interactor.
- The interactor does some work with the request, constructs a **response** object, and sends it to the presenter.
- The presenter formats the data in the response, constructs a **view model** object, and sends it to the view controller.
- The view controller displays the results contained in the view model to the user.

You can find the full source code and tests at GitHub.

Congratulations! You have finished this handbook, and are now equipped with the Clean Architecture and its software design principles. You can use the templates to start a brand new project using Clean Swift. If you're looking to convert your existing project from MVC to Clean Swift, there's also a video course for that. Just log in to your Wistia account accompanying this handbook. Whenever you have questions or get stuck, take advantage of the one-month free trial to my mentorship program on Slack.