THE CLEAN SWIFT HANDBOOK

a systematic approach to building maintainable iOS apps

Raymond Law

Table of Contents

1. 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 codebase 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.

2. The “Massive” View Controller

Swifty’s story should be a familiar one. In fact, you may already recognize him in yourself now. Or in a couple of years. Certainly, you can name someone you know just like him. Like Swifty, almost every iOS developer knows, experiences, and suffers from **Massive View Controllers**.

Painful Symptoms

Developers are humans. And humans have a tendency to do things in the easiest and most direct way. When we try to get something to show on the iPhone screen, we need to write code. Because the view controller file is already opened in Xcode, that’s where we naturally put all the code in by default.

When that something shows up on the screen, we declare victory. We’re excited to move on to the next cool task. We skip the time required to polish what we’ve just done. After all, it’s already working. You can see it. Putting all the code in the view controller is easier than having to create a new file, think of an appropriate name for the file, decide whether you should create a new class, struct, or simply a global function to house the code you’re about to write. Hey, we can see it on the screen. That means it passes the eye test. Why bother writing unit tests? Writing test code is not that fun.

We indulge in this self-inflicting cycle for as long as we can stand the pain.

When the client reports a bug and asks us for an estimate to fix it, we’re suddenly reminded of the work we have to do to find, reproduce, and fix what we left behind.

So, we tell them maybe a couple hours. But we know deep down in our guts that estimates isn't based on anything. We aren't sure how long it'll take, but we need to give a number. So we just *come up* with one.

If the bug fix turns out to take long than our **guesstimate**, it'll come back to bite us. It makes us look bad, because we aren't smart enough. Something that should take 2 hours take us 6 hours. The client may say *"I thought you said it would be done in a couple hours."*. They have a right to be a little upset because they're paying you for 6 instead of 2 hours, as you promised them before. Trust is lost and relationship is strained.

On the other hand, we may choose to hide our guilt and avoid pissing off the client. We stick to our original 2 hours estimate, but we overwork to get it done in 6 hours. The client may be happy, but we aren't. What is worse is that the client will expect a similar quick turnaround in the future. And we'll stubbornly oblige by underestimating our hours again. We sacrifice many nights and weekends to get stuff done , but don't feel appreciated. Resentment builds up and enthusiasm dies.

So, which one are you? Are you the pisser or the pleaser? But you don't have to be either.

Dœs the problem lie in the client? Do they just not understand how development works. They don't know how complicated coding can be. Their idea is too big. They keep changing requirements. They only care about the UI. They have no idea how complex it is underneath the hood.

If only they had asked you to do that 6 months ago while the code was still fresh in you remind, it would have taken less time. Now, you have to read and understand the code you haven't looked at since 6 months ago. You need to navigate through 50 classes, protocols, and methods, trace through various conditionals and loops, identify the relevant line. After you find it (hopefully it is the right place), you have to

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 Controllers.

While we can all agree there are both good and bad clients, we may or may not have control over that. But is there something we can control that we can do better? Can we communicate better? We can certainly give better estimates, if we understand the codebase better. Here is the reality. We're fully responsible for the health of the codebase because we're the ones who write the code.

When Medicines Don't Work

If these symptoms are familiar to you, you're likely suffering from Massive View Controllers too. And you may have already attempted to improve the situation.

You read about the myriad of software design patterns. The most famous and thrown around being dependency injection. You then try to refactor your view controllers, extract your data sources and delegates, wrap your network calls behind a protocol, encapsulate your persistence layer. You may even try to write unit tests for your classes and methods.

It feels good at the beginning, even for a while, when the new code is fresh in your mind. But, over time, the app slowly rots again. Soon, before you realize, you're back at ground zero. Somehow, bad code seems to always find its way back into your codebase.

These things that you read and try are what I call the first aid kit. Damage has already been done to the codebase. You are just *putting bandages on your wounds*. You may stop the bleeding, but infection already spreads. You are treating the symptoms, not curing the disease. Taking pain killers may relieve the pain for a while. But the pain always comes back. Relying on pain killers harms your body. Who wants to have to take medicines for their entire life, if you can help it?

This approach is wrong. We shouldn't keep creating pains and relieving them. We shouldn't need to fix things continuously, forever. Instead, we should do it right from the get-go. To have a healthy body, we start with a good foundation, by eating right and exercise regular. We also need to maintain this healthy lifestyle. Yes, we may get sick once in a while and need to take some pills and shots. But these are minor. We don't want to get seriously injured. We certainly don't want to deal with a chronic disease.

If we do that to our app, we'll have a healthy codebase. We want to prevent the damage from being done to the codebase in the first place. And we follow the same healthy routine that we establish earlier. Then, our codebase will also be in a healthy state. And yes, we may need to occasionally refactor a little here and there. These changes should be small and isolated, and are easy to handle. It shouldn't cause sleeplessness.

Code should be written as *factored* to begin with without the **re**. Refactoring should be done to improve code and performance. It shouldn't be done to fix Massive View Controllers.

Instead of taking medicines forever, the solution should be to put the right application architecture in place at the beginning. We have to understand this. Houses are built for residents to live. Offices are built for workers to do work. *Apps are built for developers to read and write code*. These things are for different purposes, and so must be architected differently to suit the people and their needs.

Minimize Context Switching

In the industrial complex, a company can double the output by buying twice as many machines. For manual labor, the boss can hire twice as many workers. However, with programming, you can't simply buy productivity. A client or manager may think putting 2x developers into a project would knock off 2x number of tickets. He may also think it helps finish the project in half the time. We, as developers, know that's not true.

Non-tech folks can buy a computer twice as fast, or order Internet service with double Mbps. But there's a difference between hardware and software. Software development is highly creative work. It involves a lot of brain activities. Unfortunately, a team of developers don't have the superhuman power to connect neurons in our brains. We still rely on verbal and written forms of communication. That means there needs to be coordination. Whenever there is coordination, that 2x formula dœsn't hold.

In math, a teacher will tell you 1 + 1 = 2. But in software development, any developer will tell you 1 + 1 < 2.

Instead of putting more developers into a project, a more effective way to increase productivity is to allow the developers to get in the zone.

Any developer, regardless of language and platform, understands what it means to get in the zone. It happens when you allow your brain to hold the full context of the problem that you're solving. That is when you're the most productive. When we're in the zone, we want to keep working. We do not want to be disrupted. If we're disrupted, our brains switch off from the current project and onto another task. We'll have to switch it back to the project in order to be productive again.

This context switching process is slow. And it isn't guaranteed to succeed. Sometimes, we aren't able to switch it back to the project. Then, we would be working at maybe half speed. So, in order to be more productive, we want to minimize context switching as much as possible. When a developer is deep coding, it is best not to disrupt him.

There are a lot of things that can cause this context switching process to happen, most often out of our control.

The most common and frequent type of disruption happens in the office, whether you work locally in a physical office or remotely at your own home. These include emails, Slack, meetings, conference calls, and stand ups. They not only block off a block of time, but also require preparation and thoughts. If you know there is going to be a meeting in 30 minutes, you won't try to get in the zone an hour before because you know you'll be interrupted. So you wait until it's after. But the ramp up time afterwards also takes time. So you lose the before, during, and after.

If you consistently work for long hours, you're also subject to burnout. When that happens, you'll have to wait until it's all over before you can be productive again. The time you lose from being burned out far exceeds the long hours that you put in.

Context switching caused outside of the office include being interrupted by a family emergency. Someone you love may be sick or in need of your attention immediately. There is simply no way around it. For the less urgent, you may hope to come back from a vacation rejuvenated. But you'll also find yourself being slow to get back up to speed with the project that you were working on before. Other developers may have changed a lot of code that you don't recognize anymore. Finally, you may simply have a bad day when everything you touch would break.

While not everything is under our control, it'll serve us better if we try to minimize context switching as much as possible.

Let's start off with office distractions. I don't respond to most emails right away. Instead, I label them so I don't forget about them. I'll find another time when I'm not deep thinking and problem solving. I also find that if I wait to reply to an email, I'll have a different perspective and can provide better answers to questions. The same goes for Slack. We shouldn't treat Slack as synchronous communication. When I type something in Slack, whether it's in a public channel, a mention, or even a DM, I don't expect an immediate answer. It's very likely that other people are busy with their own things. They may just want to take some time to give me better answers to my questions. If we learn to respect, Slack will be a much better tool.

Meetings should be for discussing business requirements on a high level. "We want the app to drive higher user engagements. So we want to modify feature X to do Y. And we'll measure the results by doing Z." They aren't meant for communicating low level code details. "The wrong posts are showing up for this user. It seems like the `UsersAPI` is passed the wrong `userID`." Also, only related personnels should be invited to meetings and conference calls. If there are a lot to discuss that involve many people, it's better to have multiple, smaller, shorter meetings. Meetings should be about participation, not attendance.

To avoid burnout, we shouldn't be working inhumanly long hours. If long hours are required, it most likely means the project is in bad shape. When features are broken into smaller pieces, nothing should take this long. If you spend the majority of time in finding the right place in code to make changes just to see how it responds, the rest of this book will be a huge help with that. On the other hand, if the project is in good shape but deadlines are so tight that it requires long hours, that's not a very healthy culture.

Maximize Getting in the Zone

Minimizing context switching is easy to understand, but sometimes hard to execute. However, all hope is not lost. On the other hand, we can maximize getting in the zone

to be more productive. Less external context switching certainly facilitates that. But we can also be more proactive by learning to write better code. If you're disciplined from the beginning, you can avoid Massive View Controllers altogether! Our code is not always perfect. If we can maximize getting in the zone, we'll write good code more often than we write bad code. As a result, the codebase will be healthier than not.

Frequently, I see many developers do things at random. When they need to write some code, they jump right into it without much thinking. Whatever happens to come across in their heads at the moment are what they'll enter in Xcode. This is true whether they implement a new feature, fix a bug, refactor a method, redesign a class, or write a unit test. As a result, the code becomes disjointed and without cohesion. Adding to that is they are in and out of the zone every day and hour. We often work on a team environment with multiple developers. Every developer's brain is wired differently. We all do things differently. Having multiple developers in a team only compounds this *random code effect*.

Instead, we need a system to guide all developers on the team to do things in a consistent and cohesive manner. Code style guide is one example at the syntactic level. At the high level, developers on a team should be able to read and immediately understand the code. The low level implementation details can be up to the individuals. While trying to paint the picture of the overall system in our brain, we should only need to look at the high level system, and skip the low level details. We need to focus on the problem solving part.

Even if you're the solo developer for a project, a system still helps you be more consistent in the code you write. Practicing this system helps you develop good coding habits. You don't have to rely on the bad habits of doing things randomly, based on what comes across your brain at the moment. You'll do things right more often.

The key is not to be more productive. They key is to be productive more often.

This is especially important if you work on a large scale app that serves a lot of users. You could have been working on the same app every day for 3 months. You understand the goals behind the app idea. You gain proficiency in the business domain. You develop a familiarity with the codebase. You know all the technical nuances. But if there are 70 screens and 260 features, you can't possibly put the whole context in your head when you're problem solving. You must first break down the app into smaller, logical pieces that you can hold in your brain. Then, you only need to hold a small subset of context in your head.

How to break an app down is a skill that you must master. A system will help you do that. If you do it right, you'll have a beautiful application architecture that is a joy to work with. Becoming a senior engineer or software architect is not about knowing more stuff. It's about knowing your stuff deeper. If you minimize context switching and maximize getting in the zone, you can 2x your abilities.

Massive `viewDidLoad()` Method

Any iOS app has at least one view controller. For a typical iOS app, there are a number of screens. For each screen, there is a `UIViewController` subclass. You create it either in a storyboard or in code. A view controller is the main driving force behind everything that happens on the screen.

There are several view lifecycle events that you can respond to events. They are:

- `viewDidLoad()`
- `viewWillAppear(_:)`
- `viewDidAppear(_:)`
- `viewWillDisappear(_:)`
- `viewDidDisappear(_:)`

The most popular method to hook into the view lifecycle is the `viewDidLoad()` method. The `viewWillAppear(_:)` and `viewDidAppear(_:)` cousins work too. But there's a good chance that the data won't be ready before the screen is actually displayed. The user can possibly see the screen with some empty labels and image views, and then the data shows up a second later. So the `viewDidLoad()` method is a better and more commonly used method.

You don't want a screen to be blank when displayed to the user. There are a lot of things that you need to prepare before the screen should be displayed. Also, apps are getting more complex nowadays. So there are a lot of things that can happen in `viewDidLoad()`. Most likely, you need to show some text and images when a screen is displayed to the user. You may also want to customize some things for every user such as a greeting like "Welcome back, Steve." Finally, almost all apps need to fetch some data over the network, and/or load some states from `UserDefaults`, Core Data, or iCloud.

As a start, many developers write code in `viewDidLoad()` in their first attempt to get something to work. Once they see something on the screen, they add more code to make it right. When it finally comes together, they leave the code there, and move on to the next task. As a result, the `viewDidLoad()` method becomes longer and longer. A Massive View Controller is thus born from the `viewDidLoad()` method, such as:

```
class ViewController: UITableViewController
{
  override func viewDidLoad()
  {
 super.viewDidLoad()
    // Show some text and images
    // Customize greeting
    // Fetch some data over the network
    // Load some states from iCloud
  }
}
```

Each of those tasks can span many lines of code. I'm just showing the comments for brevity.

A super simple way to simplify `viewDidLoad()` is to just extract these tasks out to their own local methods, still within the view controller. It may look as follows:

```
class ViewController: UITableViewController
{
  override func viewDidLoad()
  {
 super.viewDidLoad()
    showTextAndImages()
    customizeGreeting()
    fetchData()
    loadStates()
  }

  func showTextAndImages()
  {
    // Show some text and images
  }

  func customizeGreeting()
  {
    // Customize greeting
  }

  func fetchData()
  {
    // Fetch some data over the network
  }

  func loadStates()
  {
    // Load some states from iCloud
  }
}
```

Suddenly, `viewDidLoad()` is a lot slimmer. It consists of only 5 function calls. That is certainly an improvement. The names of these new methods convey the actual meanings. So, it is vital to give some thoughts to naming your methods. At a higher level, you know what needs to happen when the view is loaded into memory.

Let's look at a real and more complex massive `viewDidLoad()` method:

```
class ViewController: UITableViewController
{
```

```
 override func viewDidLoad()
 {
super.viewDidLoad()
   let currentUser = userManager.loggedInUser()
   if let currentUser = currentUser {
     var followerPosts = [Post]()
     let followers = userManager.followersForUser(currentUser)
     for follower in followers {
       postManager.fetchPostsForUser(follower) { (posts, error) ->
() in
         if let error = error {
           self.showAlert(error: error)
         } else if let posts = posts {
           followerPosts += posts
         }
         if follower == followers.last! {
           self.recentPosts = Array(followerPosts.sorted().re-
versed()[0...4])
           self.tableView.reloadData()
           self.loginButton.isHidden = true
         }
       }
     }
   } else {
     recentPosts = []
     tableView.reloadData()
     loginButton.isHidden = false
   }
 }
}
```

Right off the bat, do you know what it does? What needs to happen? Here are some questions for you:

- How long did it take you to read and understand what it does?
- Do you think you'll remember a week from today?
- If I ask you to make a change, will you know what to do without having to read it again?
- Can you be sure you won't break something when you make the change?
- What do you feel about passing this along to someone else who needs to make a change?
- What if someone else passes this to you? What is your first reaction?
- How do you know if it works or not? How do you prove it?

- If it doesn't work, how will you go about fixing it?
- Maybe you want to write some tests, but how do you start?

In the simplest terms, this is what the `viewDidLoad()` method dœs. It displays the user's followers' posts. After the view is loaded, it tries to show the 5 most recent posts of a user's followers. If the user isn't logged in, it shows a login button instead. But how long did it take you to figure this out? Are you 100% confident you figure it out correctly?

Maybe you're thinking that after you write some unit tests for it, you don't ever have to look at this `viewDidLoad()` method again. But this method is untestable. It's doing too many things. It violates the **Single Responsibility Principle**. Unfortunately, it follows the *Ten Responsibilities Principle*:

1. Get the currently logged in user.
2. Find the user's followers.
3. Fetch posts for each of the followers.
4. Aggregate all followers' posts.
5. Check for the last follower.
6. Sort the posts.
7. Limit to the first 5 posts.
8. Refresh the table view.
9. Show/hide the login button.
10. Show an alert when there is an error.

It's easy to write a unit test for a method that dœs one thing. But it's impossible and useless to write unit tests for a method that dœs 10 things. In order for any tests to be useful, we first need to break down this massive `viewDidLoad()` method into more manageable pieces. Then we can write unit tests for these responsibilities, separately. The resulting unit tests will be much simpler too.

Do you know there is actually a logical error in this massive `viewDidLoad()` method? The `follower == followers.last!` check isn't sufficient. The `PostManager`'s

`fetchPostsForUser(_:completionHandler:)` method is asynchronous. There is no guarantee the server responses will be in the same order as the requests. You could very well receive the response for the last follower before you receive the responses for all the previous followers. So, checking if a follower is the last in the `followers` array is unpredictable. A better check is `allFollowerPosts.count == followers.count`. So you don't join and sort the posts and invoke the completion handler until you receive all the followers' posts.

With massive methods, something like this is easy to slip through the crack. Let's break up these responsibilities next.

Breaking Up Responsibilities

The Single Responsibility Principle (SRP) states that *a class should have only one reason to change*. We can apply the same SRP to methods. A method should also have only one reason to change.

The above `viewDidLoad()` method has 10 responsibilities. All of them become coupled in the single method. If you need to make a change to one responsibility, it'll cause undesired side effects to the other responsibilities.

For example, when the asynchronous `postManager.fetchPostsForUser()` method completes, it returns `[Post]?` and `ErrorType?` to the closure. Now, assume you need to change it so that it returns a `Response` struct containing success/failure and the payload such as follows.

```
struct Response
{
  var success: Bool
  var posts: [Post]
}
```

You would have to change a lot of code. You actually have to re-read the entire method and manually test it many times to make sure it still works for all the edge cases. All your unit tests, if you have any, will be subject to change. Even the tests for the other 9 responsibilities are up in the air. Passing or failing dœsn't mean anything, until you update all the unit tests to reflect this new change. If you don't update them, you may as well throw them away.

Let's separate these responsibilities into shorter multiple methods. We can then test these single responsibility methods easily. Better yet, these shorter methods and tests don't have to be thrown away. They'll be part of the final refactored code.

The Supporting Cast

First, let's define the `User` and `Post` models that are used throughout the app.

```
// MARK: - User

struct User
{
  var id: String
  var email: String

  init(id: String, email: String)
  {
    self.id = id
    self.email = email
  }
}

extension User: Hashable
{
  var hashValue: Int
  {
    return id.hashValue
  }
}

extension User: Equatable
```

```
{
  static func ==(lhs: User, rhs: User) -> Bool
  {
    return lhs.email == rhs.email
  }
}
```

The `User` model is very straight forward. It has two members: `id` and `email`. The `init(id:email:)` initializer takes these as parameters to uniquely identify each user of the app. It conforms to the `Hashable` and `Equatable` protocols using extensions.

```
// MARK: - Post

struct Post
{
  var timestamp = Date()
}

extension Post: Comparable
{
  static func < (lhs: Post, rhs: Post) -> Bool
  {
    return lhs.timestamp.compare(rhs.timestamp) == .orderedAscend-
ing
  }
}
```

The `Post` model is similarly straightforward, and conforms to the `Comparable` protocol. The `<(lhs:rhs:)` function is used to sort posts by timestamps. Newer posts are *larger than* older posts.

Next, we extract the networking code to their own API manager classes.

```
// MARK: - User API

class UserManager
{
  func loggedInUser() -> User?
  {
    return User(id: "a", email: "a@a.com")
  }
```

```
  func followersForUser(_ user: User) -> [User]
  {
    return [User(id: "b", email: "b@b.com"), User(id: "c", email:
"c@c.com")]
  }
}
```

To keep the focus on `MassiveViewController`, the `UserManager` class has only two stub methods, just enough to make the code compile. The `loggedInUser()` method returns a fake user, and the `followersForUser(_:)` method returns a list of fake followers which are also `User` objects.

```
// MARK: - Post API

class PostManager
{
  func fetchPostsForUser(_ user: User, completionHandler: ([Post]?,
Error?) -> ())
  {
    // Use your favorite networking toolset such as URLSession,
Alamofire, or AFNetworking
  }
}
```

The `PostManager` class is our gateway to the outside world. The `fetchPostsForUser(_:completionHandler:)` method accepts a user as argument (most likely the logged in user). It then fetches all posts created by the given user. When it finishes, it invokes the `completionHandler` block. If the network call succeeds, it returns a list of posts. If it fails, it returns an appropriate error. You can use any networking library you wish to use, as it's just implementation details.

The Main Actors

Without an architecture in place and going with instincts, a *single-class refactoring* of the `viewDidLoad()` method is shown below:

```
// MARK: - View Controller
```

```
class MassiveViewController: UITableViewController
{
  let userManager = UserManager()
  let postManager = PostManager()

  var recentPosts = [Post]()

  @IBOutlet weak var loginButton: UIButton!

  // MARK: - View lifecycle

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

  // MARK: - Steps to follow

  func displayFollowerPosts()
  {
    if let currentUser = userManager.loggedInUser() {
      let followers = userManager.followersForUser(currentUser)
      fetchPostsByAllFollowers(followers, completionHandler:
{ (posts: [Post]?, error: Error?) -> () in
        if let error = error {
          self.showAlert(error: error)
        } else if let posts = posts {
          self.updateFollowerPosts(posts)
        }
      })
      hideLoginButton()
    } else {
      showLoginButton()
    }
  }

  func updateFollowerPosts(_ posts: [Post])
  {
    recentPosts = Array(posts.sorted().reversed()[0...4])
    refreshFollowerPosts()
    hideLoginButton()
  }

  // MARK: - Fetch posts

  func fetchPostsByAllFollowers(_ followers: [User], completionHan-
dler: ([Post]?, Error?) -> ())
```

```
  {
    var allFollowerPosts = [User: [Post]]()
    for follower in followers {
      fetchPostsByFollower(follower) { (posts, error) -> () in
        if let error = error {
          completionHandler(nil, error)
        } else if let posts = posts {
          allFollowerPosts[follower] = posts
          if allFollowerPosts.count == followers.count {

completionHandler(Array(allFollowerPosts.values.joined()), nil)
          }
        }
      }
    }
  }

  func fetchPostsByFollower(_ follower: User, completionHandler:
([Post]?, Error?) -> ())
  {
    postManager.fetchPostsForUser(follower) { (posts, error) -> ()
in
      if let error = error {
        completionHandler(nil, error)
      } else if let posts = posts {
        completionHandler(posts, nil)
      }
    }
  }

  // MARK: - UI

  func refreshFollowerPosts()
  {
    tableView.reloadData()
  }

  func showLoginButton()
  {
    loginButton.isHidden = false
  }

  func hideLoginButton()
  {
    loginButton.isHidden = true
  }

  func showAlert(error: Error)
```

```
    {
      // ...
    }
  }
```

The new `MassiveViewController` class now has many more methods. Let's dissect each of them.

1. User Interface Methods

Let's look at the shorter methods first. The following methods have to do with the UI, and the method names themselves tell you what they do.

- `refreshFollowerPosts()`
- `showLoginButton()`
- `hideLoginButton()`
- `showAlert(_:)`

These methods are quite self explanatory. The `refreshFollowerPosts()` method simply reloads the table view to refresh any newly fetched posts. The `showLoginButton()` and `hideLoginButton()` methods just reverses the `isHidden` flag. The `showAlert(error:)` method accepts an `Error`, the constructs and displays an informative error message to the user. You can use `UIAlertController` to achieve that.

2. Networking Methods

Network operations are asynchronous in nature. The two most common paradigms of waiting for data to be ready are delegation and closure. We choose to use closures in this example. For any networking code, you have no control of the the network condition and remote API endpoint. So you must deal with both the success and failure cases. As a result, methods that make networking calls are longer by necessity. Nonetheless, they are still simple enough that you can tell right away what they do.

- `fetchPostsByAllFollowers(_:completionHandler:)`
- `fetchPostsByFollower(_:completionHandler:)`

The `fetchPostsByAllFollowers(_:completionHandler:)` method takes an array of followers as parameter. It loops through each follower to invoke the `fetchPostsByFollower(_:completionHandler:)` method. If it fails, it returns `nil` for the post array and an error. If it succeeds, it first saves the follower's posts to the `allFollowerPosts` dictionary. This dictionary is keyed by `User` objects. That's why we make the `User` struct conform to the `Hashable` protocol. Next, we check for `allFollowerPosts.count == followers.count`. If true, the dictionary has the same number of elements as the `followers` array. This means we have fetched posts for all the followers. We then join all the posts in the dictionary and return this flattened array of posts to the completion handler.

The `fetchPostsByFollower(_:completionHandler:)` method simply invokes `PostManager`'s `fetchPostsForUser(_:completionHandler:)` method. When it finishes, it returns either the `posts` array or the `error` to the completion handler.

So far, all the methods we just looked at have one clear responsibility. Their method names convey what they do. Each function does only one thing. It's very clear what that thing is. The method bodies are also very short and easy to read.

3. Business Logic Methods

The two remaining methods contain the actual business logic. As a reminder, when the view is loaded, we need to display the user's followers' posts. We should show the 5 most recent posts of a user's followers. If the user isn't logged in, it should show a login button instead. The following two methods help us achieve that.

- `displayFollowerPosts()`
- `updateFollowerPosts(_:)`

The `viewDidLoad()` method is reduced to just one method call - `displayFollowerPosts()`. This `displayFollowerPosts()` method is the driver of the engine. It *coordinates* other subtasks by invoking other methods at the right place and time. First, it checks to see if the user is logged in or not. If not logged in, it calls the `showLoginButton()` method to show the login button so the user can login. If the user is already logged in, it calls the `hideLoginButton()` method. It also calls `UserManager`'s `followersForUser(_:)` method to retrieve the user's followers to pass to the `fetchPostsByAllFollowers(_:completionHandler:)` method. When this method returns in the completion handler, if there is an error, it shows an alert to the user. If it succeeds, it calls the `updateFollowerPosts(_:)` method with the `posts` result.

The `updateFollowerPosts(_:)` method accepts a `Post` array. It sorts them by timestamp from newest and oldest, and stores the 5 most recent posts to the local `recentPosts` instance variable. It then invokes the `refreshFollowerPosts()` and `hideLoginButton()` methods, which you already saw before.

These new `displayFollowerPosts()` and `updateFollowerPosts(_:)` methods have descriptive method names. They are also easy to read and understand. In the original massive `viewDidLoad()` method, the business logic is buried along with the user interface logic and networking logic. Separating out the responsibilities into individual methods not only improves code readability, but also makes it easier to write unit tests for them. Since they have clear, single responsibilities, it's obvious to specify the inputs, outputs, and behaviors.

These two methods are what I called the **coordinator methods**. In simpler terms, you can think of them as managers in a corporation. They tell their employees to do the actual work. But they don't do the work themselves. Rather, they *coordinate* the tasks performed by each employee. This coordination holds the high-level, application logic. It determines what things need to happen, the order in which they should happen, and what should happen instead if conditions are met or not.

We Can Do Even Better

This refactoring is definitely an improvement over the original massive `viewDidLoad()` method. The multiple responsibilities have been separated out into individual methods. Although the refactoring produces more methods, the result is better code. You need a smaller context in your brain when you're problem solving.

Remember the `follower == followers.last!` bug in the original massive `viewDidLoad()` method? We only need to focus on the `fetchPostsByAllFollowers(_:completionHandler:)` method to figure out the error (It should be changed to `allFollowerPosts.count == followers.count`). At the lower level, we already know the `fetchPostsByFollower(_:completionHandler:)` method works. So we don't need to worry about the `PostManager`. On the other hand, at the higher level, we won't have to worry about updating the follower posts, showing/hiding the login button, or showing the alert. That's already taken care of by the `displayFollowerPosts()` method.

However, all methods still live in the same view controller. We don't have a massive `viewDidLoad()`, but we still have a Massive View Controller. In typical MVC, everything still happens in the view controllers. Yes, you can certainly move the `User` and `Post` models and the API managers to their own files. But it's just moving code around. The business logic still lives in the view controllers.

More importantly, you have to go through a lot of mental exercises in order to refactor a method or a class. Our brain capacity is limited, so we want to fit more useful, not more, information to our brains. This is the biggest problem of Massive View Controllers. Refactoring is not the solution. The correct approach should be to minimize the thinking required to write and maintain good code.

In the rest of this book, you'll learn how to *componentize* an app using the Clean Swift architecture. You'll learn a system to guide you through a list of steps to follow. You no longer have to rely on random refactoring. Let's dive right in.

3. Introducing the Clean Swift iOS Architecture

Refactoring and testing generate a lot of buzz in the software development space. Like most developers, that's what I tried as a first attempt. But soon after, I realized that refactoring and testing are not the solution. The effort required is uncomfortably high, and the effects are minimal at best. I said to myself, "That can't be right. Something else must be missing."

So I set out to search for the root cause why we always tend to end up with Massive View Controllers in iOS apps. Are they avoidable? Is it possible to write factored code and never have to refactor? I decided to do something serious about it.

Eventually, I concluded that refactoring and testing are desirable only if the codebase is already built on a solid foundation. So the root cause of Massive View Controllers is not having a solid application architecture.

That makes perfect sense. If a building has a shaky foundation, it'll eventually topples. If your codebase has a shaky architecture, it'll deteriorate over time. You'll suffer every day having to just deal with it. When you finally have enough of it, you would rather just rewrite the app from scratch. We all know how expensive a rewrite can be, both time and money wise. Most of all, how can you guarantee it'll be better the next time around?

Next, I researched about various iOS architectures that were out there at the time, such as MVC, MVVM, and VIPER. I've also experimented with different testing and mocking frameworks. After carefully evaluating the alternatives, I focused on how to

apply **Uncle Bob's Clean Architecture** to iOS development using the Swift programming language. I'll just call this **Clean Swift**.

Architecture v.s. Design Patterns

First things first. An important distinction needs to be made before we begin.

Architecture has to do with the high level, overall, organization of features throughout the app. A good application architecture ensures all features play well together. It makes it easy for developers to work with the code. An application uses one architecture to organize all its features.

Design pattern has to do with the low level programming paradigm to support a single feature requirement. A good design pattern makes use of the available underlying language constructs to make the code for the feature implementation easy to understand. A feature may make use of one or more design patterns in its implementation.

In short, an app uses one architecture for all its features. Each feature uses one or more design pattern for its implementation. By this definition, architecture is not design pattern. That is the most basic and easiest way to explain their differences.

Know Your Benefits

After reading this book, you'll be able 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.

The above are all traits that you can observe from any new code you write using Clean Swift. But there is something more important. What if you know exactly which file to open and which method to look? You can get in the flow right away. How much more productive will you be? Imagine your tests run in seconds instead of minutes or hours. How much more often will you run them? You'll be confident your code always work. To me, these are the real benefits.

After explaining the Clean Swift architecture, I'll also show you a simple system with a list of detailed steps that you can follow to perform your day-to-day programming tasks. You don't have to stop and think where to put a piece of code. Your code logically falls into the right place. That's why you'll know exactly where to find things when you look back at your code 6 months later. It's all part of the same system.

There are 3 keys to this system of writing readable and maintainable code. But, ironically, they don't have anything to do with code. Let's examine each of them now.

Key #1 - Communicating Ideas

Code is the common language developers use to communicate ideas among ourselves. When you see a `for...in` loop, you know it's iterating and operating on every element in a collection of values. When you see a completion handler as the last parameter of a function signature, you know the function is asynchronous and the

result won't be ready right away. So, it's important that we speak in the same language, and agree on some basic rules for us to communicate ideas effectively.

In typical MVC fashion, every developer puts their code in the view controllers. The class becomes very large. Communicating ideas becomes difficult. Even if you move some of the code to other classes, it's still just moving code around. It dœsn't reduce complexities. In a *randomly* developed app like this, every developer makes different decisions, and has different styles and conventions. The resulting codebase looks disjointed and lacks cohesion.

In one extreme, you can completely ignore architecture and write the entire app in a single file. The computer can read that single file just as well as if it was built with a perfect architecture. However, it won't be human readable at all. Even a genius developer would have a hard time to comprehend what the app dœs.

Think of it this way. Software are written for humans, not for the computer. The compiler turns your Swift code into assembly language then machine code, so that it is consumable by the processor. Unlike graphics, Text processing and code compilation are trivial tasks for modern processors. We don't need to optimize code for them any further. Instead, we should optimize code for human consumption.

A project dœs not fail because of the lack of an architecture. It fails because the developers on the team cannot communicate ideas effectively anymore. When the project is still new, there isn't much code. It's easy to put the entire app in your head. But as the app grows, this becomes impossible. It's very difficult to code anything when your brain is mostly occupied with (sometimes even broken) context. Developers then resort to refactoring for rescue, which we already saw, wasn't the root cause or solution.

Remember this. We write software to communicate ideas. Not only to other developers on your team, but also to your future self. Even if you're the only developer

on a solo project, you'll inevitably forget about the code you write. So it's still important to write code in a way that you can easily pick it back up in 6 months. If you don't, you have no one but yourself to blame.

What is good code? It's simple. If you write code in a way that makes it quick and easy to pick back up, that is good code. And you likely have a good architecture in place. A good architecture is critically important to minimize the context switching required to get back up to speed. For you and other developers on the team.

Key #2 - A System Architecture

Remember I mentioned in the last chapter that the most thrown around advice is to use the right tool for the right job. Or, as some say, every app is different, so use the architecture that suits you and your project best. These answers are pleasing to the ears. They are also foolproof, because they are never wrong. But they are completely useless.

In fact, you can use any architecture such as MVC, or even no architecture, and write good and maintainable code. You just have to be superhumanly disciplined in sticking to the good habits and proven principles in every class you design, every method you write, for the entire app. For your present and future self. And for all developers on the same team.

If you can truly do that, you won't need this book. But we are humans. We aren't perfect. As soon as we let things slip through the cracks, technical debts start to build up. Until one day, the debt becomes insurmountable.

What I'm interested in, and you should be too, are concrete advices that are applicable to any project. Not just the commenter's project, but to my projects and your projects. I want something that works universally. I just want to learn once and

use it for all. Having a consistent, dependable system that you can follow for any project can achieve this goal.

The system serves as your guide in your day-to-day programming, so that you can stick to the good habits and proven principles 99% of the time. It frees up your brain, so you can focus on the nitty gritty details of implementing your features. Even when you aren't in the zone, you can still depend on this system to guide you. You can be productive more often. You can give more accurate estimates to clients and customers.

A system serves another purpose. Architecture is intentionally boring. And that's a good thing! It is the boring, boilerplate part of your app that you don't want to work on. It isn't shiny or fancy. It won't get you excited waking up every morning to hammer at it. A system takes care of this boring, boilerplate part for you, get it out of your way, so that you can focus on the interesting part.

For instance, in the old days, many programs run in the terminal, and are mostly text-based. These programs were written to perform some specific tasks such as algorithm calculation or generating reports. They are almost always algorithm centric. They accept user inputs from the keyboard and produces outputs to the monitor or a file.

However, today's software, web and mobile, have to deal with many dependencies. These include user interface, networking, database, persistence, onboard devices, just to name a few. You build modules to interface with these dependencies. This interdependence between modules means you need to write code that talks to two or more modules. Think device drivers and utilities for peripherals.

A change in any one module can affect other modules. As a result, our job becomes much more difficult. We need to write code that is well isolated and encapsulated, so that the effect of changes in one module is minimized for other modules and the rest of the software. A good system takes care of handling this module interdependence.

Key #3 - Architecture Before Testing

If a house isn't built on a good foundation, you wouldn't want to add the fancy kitchen and patio extension. Right? If the house can't stand firm during a storm, all the upgrades and extras you add to the house will get destroyed altogether. And you'll lose all your investments. In fact, adding more things to a house on shaky foundation only makes it heavier and crush faster. Similarly, adding unit tests to a codebase on a shaky architecture will just cripple it further. You're only accelerating the buildup of technical debt.

Many developers blindly write unit tests for broken apps. They believe the mere presence of unit tests can improve the quality of their apps. After all, unit tests are good. So having more of them is good. Right? Their notion that unit tests can improve an app is noble, but not necessarily true. Okay, but at least unit tests serve as a correctness check. That may not be true either.

If you have tried to write unit tests before, you probably have seen it done wrong. Have you ever had the feeling of having to maintain two apps in one project? You have one Xcode project, but it feels like you're writing two apps - one for the app target, and one for the test target.

The truth is that unit testing can be a double edged sword. If done right, a test suite can prove your app dœs work, and alert you when you're about to break something. If done wrong, your tests are fragile.

There are two problems with fragile tests. First, fragile tests break easily. For example, you make one seemingly innocent change in your code and 25 tests fail. You end up spending more time fixing your tests than writing code for your app. Second, the tests no longer represent the desired behaviors. You have to not only fix the tests so they pass again, but also update them so they test the real things going forward. This can be a painful, endless cycle. You change 3 lines of code, 14 tests break, change 7 lines

of code, 22 tests break. You're basically having to build and maintain two apps in one project!

Another misconception is that Test Driven Development always results in better code. In TDD, you write a failing test first before writing the code to make the test pass. So every line of code will be tested, and you only write the minimum amount of code. Isn't that a good thing? Yes or no. In reality, many developers complain that TDD is difficult and slow. That means they are doing it wrong. TDD or not, you still need to have an idea of how to structure and organize your classes and methods so that it is testable. If not, your codebase would still suffer from too much coupling and a lack of cohesiveness.

Worry not. There's good news. If you learn how to write unit tests the right way, it's actually pretty easy and straightforward. Like refactoring, the codebase needs to be on solid ground first. And that's where architecture comes in. I always preach architecture before testing. With a solid architecture already in place, refactoring will be few and far between and unit testing will yield a high return on investment.

If you want to learn how to do unit testing and TDD properly and effectively, check out my other book Effective Unit Testing.

Goals of the Clean Swift Architecture

The goal of the Clean Swift architecture is not to create yet another architecture. It dœsn't try to outduel other architecture choices. There isn't any new library, framework, or plugin to install and learn. Rather, Clean Swift is a system that helps you achieve the following goals:

- Communicate ideas effectively with other developers and your future self
- Get you up to speed quickly now and down the road
- Put the right, smaller context in your brain

- Give you a list of steps to stick to good habits and principles
- Take care of the boring, boilerplate stuff so you can focus on the features
- Facilitate the interdependence between modules to minimize the effect of changes
- Make writing tests easy and obvious

Let's start learning Clean Swift by understanding the concepts in the original Clean Architecture first.

The Clean Architecture

The Clean Swift architecture is derived from the original Clean Architecture proposed by Uncle Bob. They share many common concepts such as components, boundaries, and models, flow of control, source code independence and data independence. While Uncle Bob demonstrates the Clean Architecture using Java for web apps, I'll show you how to apply the Clean Architecture using Swift for iOS apps.

First and foremost, an application architecture, puts the use cases at the forefront. The structures, such as classes and structs, functions and protocols, frameworks and libraries, exist to support the use cases. In practice, when you're working with a new codebase, the first thing you need to look for are the use cases. Do the use cases of an app scream at you? Is it obvious what they do? If you need to make a change, do you know where to make it?

What are use cases then? Think features. A feature is often loosely specified. For example, a common feature is that the user should be able to login with an email and password. This is an understatement. There are many more moving parts. For example, you can break down this feature into the following:

- The user should be able to enter an email into the email text field.
- The user should be able to enter a password into the password text field.

- When the user taps the Login button, the app should call the login API with the email and password to the server.
- While awaiting authentication, a progress bar should be displayed to the user.
- If the login attempt succeeds, the user should see the dashboard.
- If the login attempt fails, the password field should be emptied, and an error message should be displayed to the user.

You can certainly think of even more scenarios than the above. A logical question to ask is this. Do we have 1 use case or 6 use cases? At the app-level, we have 1 use case, that is, the user should be able to login. At the implementation level, we have 6 use cases. You clients and customers aren't going to care about how you actually implement it. They want the users to be able to login. You, as the developer, makes the implementation decision.

If you consider the whole login feature as a single use case, you may decide to implement it in a single module, be it a class or struct, a manager or singleton, whatever. You're in big trouble. Let's say the requirement has changed. After tapping the Login button, the user should see a new screen and ask him to enter a 6-digit code for two factor authentication. Where do you make the code changes to satisfy this new requirement? When you make the change, everything changes. Can you confidently say you haven't broken anything? You'll need to re-test all scenarios and edge cases. All your unit tests become invalid, meaning you have to re-read to make sure they're still testing meaningful behaviors.

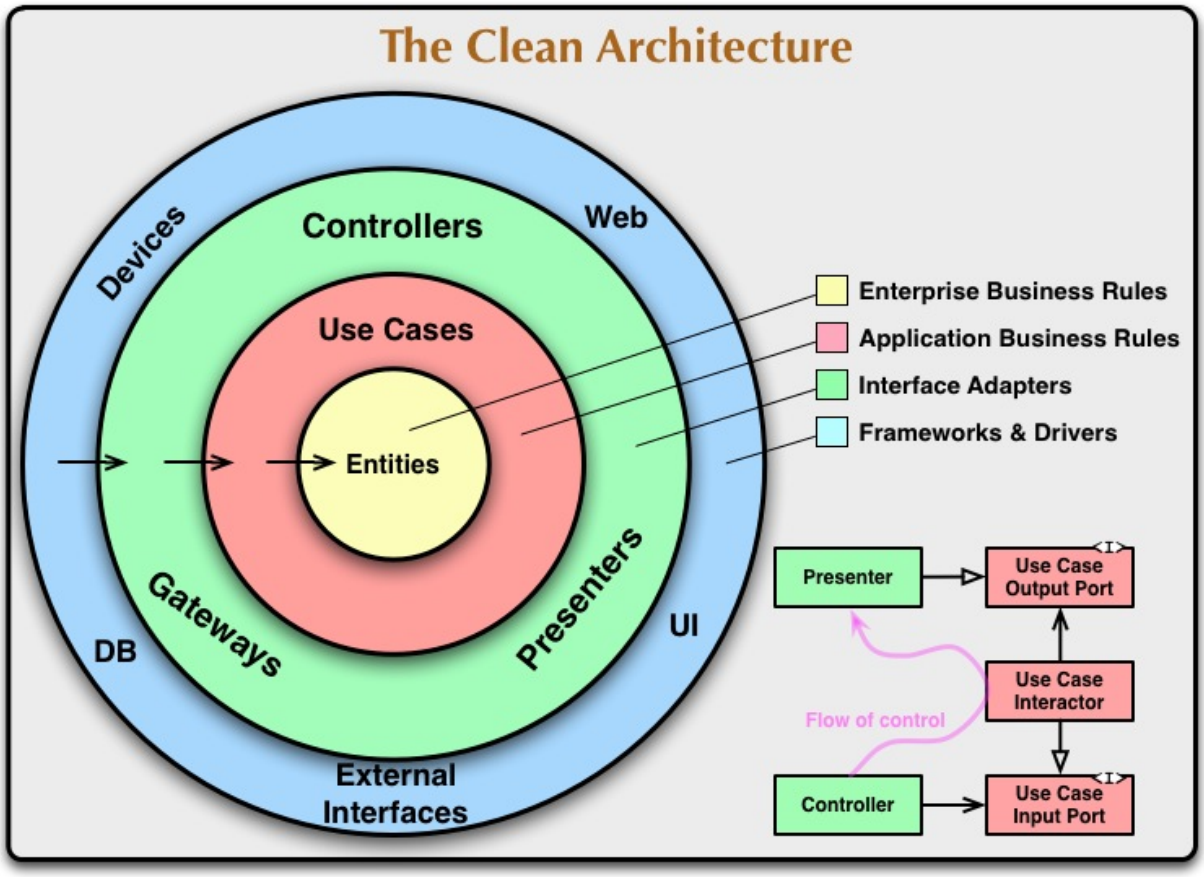
It's very important to break down this monolithic app-level login use case into smaller code-level use cases. Okay, how do you do that? How would other developers on your team do that? The fact is every developer will do it differently. Some may come up with 4 use cases, whiles others may have 7. They'll have different ideas of how to break it down. Some may want to use `URLSession` to talk to the API, while others prefer to use Alamofire.

What about the signup use case? The change password use case? The forget password use case? If everyone dœs things differently, what would the final code base look like? There're going to be a ton of coupling. Everything depends on everything.

The Clean Architecture helps you break down a large app-level use case into many small code-level use cases. It isolates these use cases into a nice separation of concerns. It is easy to find the right places when making changes. What's more is that it also decouples these use cases. You can change your mind later to replace `URLSession` with Alamofire, Core Data with iCloud, remote API versions, …etc. The changes will be isolated in its own module. You won't have to change the rest of your app. Finally, you'll be able to write unit tests for all these tiny use cases with ease because, well, they are tiny. The decoupling also means your unit tests can be run in groups. For example, you'll be able to test how the app handles both successful and failed authentication, without the need to bring up the API server. You do this by mocking the API responses.

In the end, you should be able to just look at all the use cases of the app, and have a good idea of what it dœs, without looking at any of the implementation details.

That sounds very nice!, So, how dœs the Clean Architecture achieve these goals? To understand, take a look at the following diagram from Uncle Bob's The Clean Architecture article.



The Clean Architecture is a layered architecture. Using the terms in the diagram, the inner most layer contains the **Entities** which implements **Enterprise Business Rules**. This is the highest-level business logic of the entire business. For example, you can have a web app, an iOS app, an Android app, and a kiosk app. The business logic implemented by the entities are the same across apps and platforms. It's not likely to change due to other external changes. The next layer contains **Use Cases** which implements **Application Business Rules**. These are the highest level business logic of the app, not necessarily for the entire business. The **Interface Adapters** acts like the driver software to the external world. For example, the **Controller** takes inputs from the keyboard. The **Presenter** drives the UI to display data on the screen. The **Gateways** fetches data from and stores data to the database. Finally, the **Frameworks & Drivers**are the actual external dependencies such as the keyboard, the UI, the database, the API server, and so on.

In terms of programming terms, the Clean Architecture achieves the separation of concerns by decomposing an app-level use case into smaller code-level use cases, and then isolating these use cases into the appropriate layers. These layers are structured like an onion. The innermost layer contains the highest-level enterprise business logic. As we move outward, it gets more low-level. The next two layers contain app-level and code-level business logic, respectively. The outermost layer contains implementation details that interacts with physical devices such as database, user interface, and the web. Any framework or library should belong to the outermost layer.

The most important takeaway here is that an inner layer should not depend on an outer layer. I'll just quote Uncle Bob, as he said it best:

> *The overriding rule that makes this architecture work is **The Dependency Rule**. This rule says that source code dependencies can only point inwards. Nothing in an inner circle can know anything at all about something in an outer circle. In particular, the name of something declared in an outer circle must not be mentioned by the code in the an inner circle. That includes, functions, classes. variables, or any other named software entity.*

This *source code independence* has a huge advantage. You can change the code in an outer layer, leaving the inner layers intact. Your app still compiles and works as expected. For example, you can swap in another database or make changes to the UI, without affecting the underlying business logic. This also means the unit tests that you have written for the business logic can be executed with a mock database and UI. By the same token, you can swap in another CocoaPod or Carthage library without affecting the rest of your app. How powerful is that? In practice, it minimizes the amount of code changes.

The Clean Architecture achieves source code independence by establishing interfaces between layers. The flow of control is shown in the lower right corner of the above diagram. Code execution is at the controller (green) in the outer layer. It then

goes to the interactor (red) in the inner layer, and back out to the presenter (green) in the outer layer. But the flow of control doesn't happen directly. Instead, the controller calls a function declared at the use case input port. The interactor implements that function in its implementation. Notice the directions of both arrows are pointing to the use case input port. Likewise, the interactor calls a function declared at the use case output port. The presenter implements that function in its implementation. Both arrows point to the use case output port.

At the use case input port, it's okay for the controller in the outer layer to call a function on the interactor in the inner layer. The use case input port simply makes it more explicit. However, it's not okay for the interactor to call a function on the presenter in the outer layer. That would violate source code independence. So, we reverse that dependence by introducing the use case output port. Now, the arrow points from the presenter to the use case output port - from the outer layer to the inner layer. On the other hand, it's fine for the arrow to point from the interactor to the use case output port because they belong in the same layer.

Looking at it color-wise, it's okay for the green outer layer to point to the red inner layer. But not the other way around. In essence, the red inner layer never calls any name defined in the green outer layer. We apply this **Dependency Inversion Principle** at all boundaries for all layers to achieve source code independence.

The dependency rule also applies to the data being passed between layers. The data formats and structures used in an outer layer should not be used by an inner layer. To comply to this *data independence*, we instead create isolated, simple, data structures to pass between layers. How exactly?

Let's take a look again at our login use case above. After a successful login, the server may return a user profile with a bunch of information such as auth token, name, phone, address, settings, …etc. But for the purpose of login, only the auth token matters. We shouldn't pass the other information back up to the inner layer.

The API may change such that those information is different and now the user's name is broken up into first name and last name. You have to follow through all the layers and make this change in a lot of places. It is better to create a new data structure to contain only the necessary data to pass to the next layer. This new data structure should only be used for passing data across the boundary. Once received, the inner layer should grab the data inside, and create its own data structure for internal use.

If the API dœs change, we only need to make the changes in the outer layer. The data structure we create to pass through the boundary still only needs to contain just the auth token. Therefore, nothing in the inner layer needs to change at all.

4. The Clean Swift System

Massive View Controllers are chronic. Even though he's not a developer, Seth Godin describes chronic problems best in a recent post on his famous blog.

> *The worst kind of problem is precisely the kind of problem we're not spending time worrying about.*

Developers are busy with implementing features and fixing bugs. We often have little to no time worrying about the health of our apps. Therefore, the codebase deteriorates over time, leading to view controllers becoming chronically massive.

> *It's not the cataclysmic disaster, the urgent emergency or the five-alarm fire.*

It's easy to ignore the health of our apps and leave massive view controllers alone. It's not as urgent as meeting the deadline for the next release, or completing features to please our clients, or responding to bug reports to keep our users engaged.

> *No, the worst kinds of problems are chronic. They grow slowly over time and are more and more difficult to solve if we wait.*

The longer we wait, the more massive it becomes, the more difficult to remedy.

> *Chronic problems are most often solved by building new systems. New ways to engage with the issue over time, methods that create their own habits and their own forward motion.*

The Clean Swift architecture provides you a system - a new way to look at massive view controllers, to respond to them when (even before) it happens, to establish guidelines that create healthy habits to keep your apps healthy going forward.

> *Step one is to realize you have a chronic problem.*

You're already past this step. Yay!

> *It might not be as thrilling as switching to emergency mode, but it's more effective.*

Having a system to keep your apps healthy is not as urgent or glorious as implementing features or fixing bugs. But dedicating time and effort to the health of your apps will make performing all other tasks more effective.

Organizing Your Code in Xcode

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

> *Before we begin, make sure you go to https://clean-swift.com and 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. I'm actively looking for people to test the Objective-C version with more real world projects. If this is you, please email me after you subscribe. I'll send you the Objective-C templates. Your feedback will help guide the design of the templates as there are differences in the two languages.*

Now that you've downloaded and installed the templates. Let's walk through how to set up a Clean Swift project in Xcode. You can also watch this screencast to see how to generate the *seven* Clean Swift components using the Xcode templates.

Let's begin.

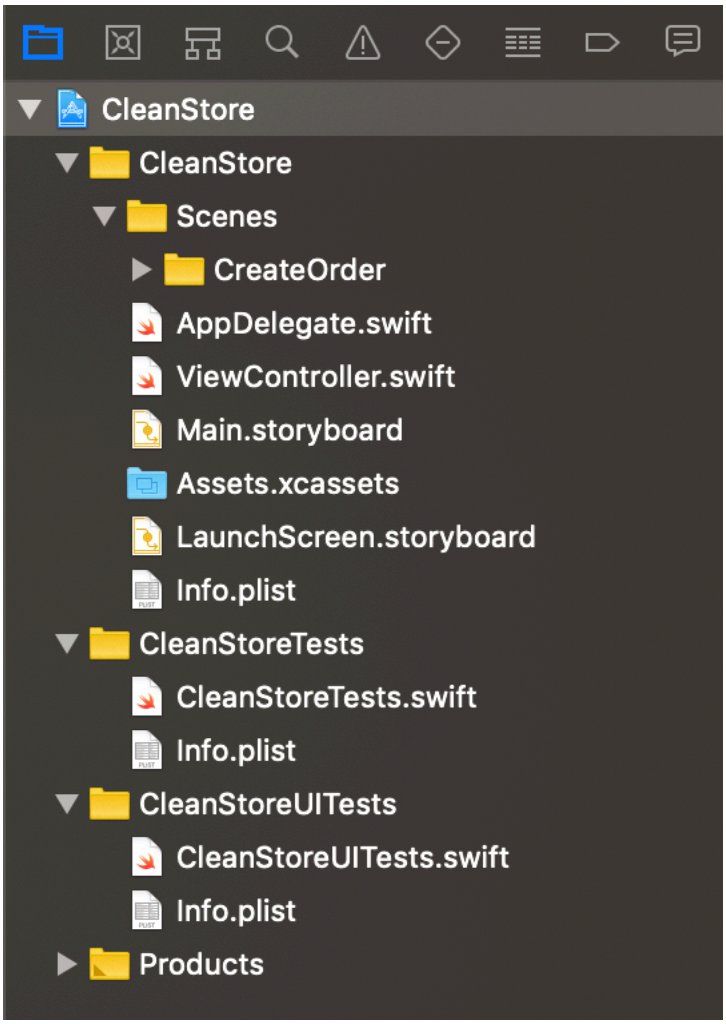
First, we'll create a new Xcode project named **CleanStore**. In the menu, choose `File -> New -> Project`. Under the `Application` section, select `Single View Application` and then click `Next`. In the `Product Name` text field, type **CleanStore**. Choose **Swift** for `Language`. Also make sure the checkbox for `Include Unit Tests` is checked. We're not going to write any UI tests, so `Include UI Tests` dœsn't

matter. Make the appropriate selections for the other fields according to your work settings. In most cases, they're already selected from your last project creation. Click `Next` when you're ready. Choose a location to save the project on your computer and click `Create`. A new and empty project is now created.

Next, let's prepare the project for Clean Swift. Highlight the `CleanStore` group (with the yellow folder icon) on the left. Choose `File —> New —> Group`. A new group (with the same yellow folder icon) is created and nested under the `CleanStore` group. Enter **Scenes** for the name of this new group. We're going to put the generated files inside this `Scenes` group so that they are not mixed up with other project files such as `AppDelegate.swift`, `Main.storyboard`, `Assets.xcassets`, and `Info.plist`. We want to keep our source code away from them.

With the new `Scenes` group still highlighted, choose `File —> New —> Group` one more time to create another nested group named **CreateOrder**. This group will contain all our files for the `CreateOrder` use case. If we were to implement a `DeleteOrder` use case in the future, we'll create another group named `DeleteOrder` under `Scenes`.

Your group structure should now look like:



Under the new `CreateOrder` group, we'll use the templates you downloaded to create the Clean Swift components. The templates can do this automatically with a few clicks, so you don't have to create new files and type in the same code every time. It saves a lot of time.

With the latest version of Xcode, when you create new files and put them under the groups, it'll also match the underlying folder structure on disk to the group structure in Xcode. That's a nice feature! That means you can expect to locate your files the same way in the Finder or Terminal as if you navigate to them in Xcode.

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

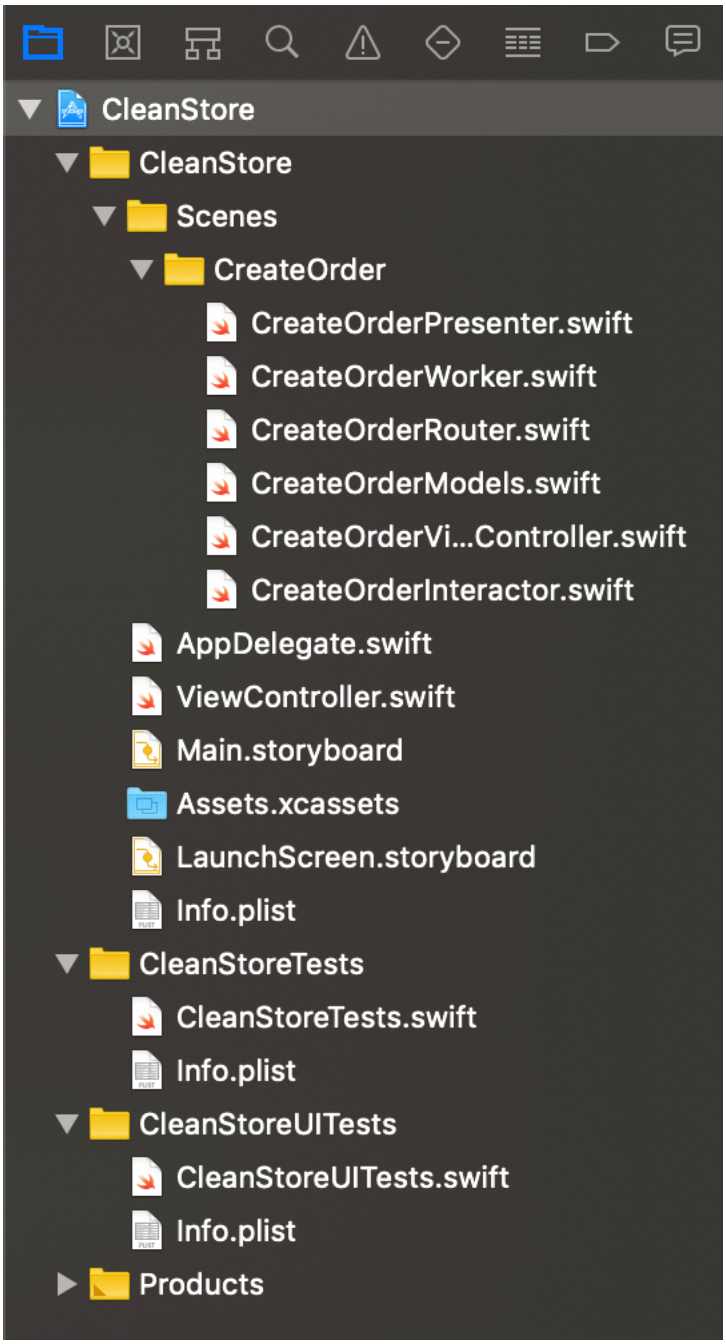
specific about the project. As Uncle Bob pointed out, group and file names should reveal your intentions for the use cases. It should not reflect the underlying framework structure. So we'll organize each use case under its own group nested under `Scenes`.

Inside the `CreateOrder` group, you can expect all files to 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.

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 named `Shared` outside `Scenes`. For example, we can create a shared worker called `OrderAPI.swift` to encapsulate the networking code and put it under the `Shared` group. By putting this new file outside of any specific scene's group, we imply that it can be used by any scene that requires to fetch orders over the network. As your app and use cases evolve, you'll come up with your own sensible grouping.

Finally, we'll create the files for the `CreateOrder` use case. Highlight the `CreateOrder` group first. Choose `File -> New -> File`. Scroll down to the bottom. If you install the templates correctly, you'll now see a new `Clean Swift` section. Select `Scenes` to create files for a new scene and click `Next`. For `New Scene Name`, enter **CreateOrder**. As you type, other fields will be automatically updated to reflect the resulting filenames. For `Subclass of`, choose `UITableViewController`. For `Language`, choose **Swift**. Click `Next`. Then, navigate to the `CreateOrder` group you just created and click `Create`. This'll put the new files under the appropriate scene in both groups and folders.

If you do everything correctly, your new files and groups should now look like:



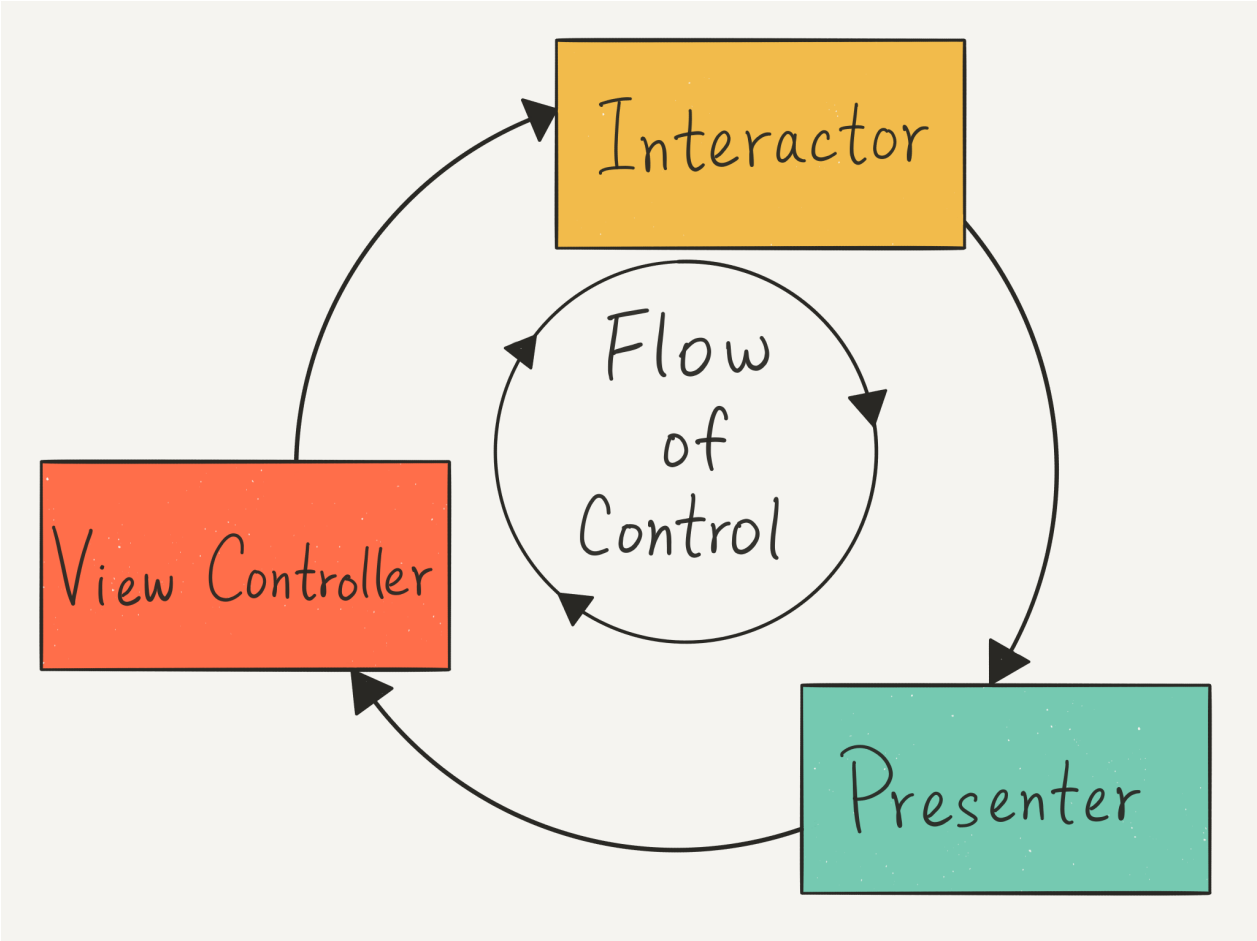
Build your project now to make sure everything is set up correctly before continuing. When you create new scenes for your own projects, these are the same steps that you need to take.

Let's start exploring the Clean Swift system by taking a deep dive to the core of this architecture - the VIP cycle.

The VIP Cycle

There are going to be a lot of new terms and definitions. Don't stress out if they seem foreign to you now. By the end of this book, I guarantee you that they will come as second nature.

The view controller, interactor, and presenter are the three main components of Clean Swift. Taken together, they form the **VIP cycle** which is central to how Clean Swift works. The VIP cycle is shown in the following diagram:



When the user taps a button in the UI, that event is captured in an IBAction method in the view controller. So, the **flow of control** starts at the view controller. The event triggers some business logic that needs to happen in the interactor. The view controller dœs this by invoking a method of the interactor for this use case.

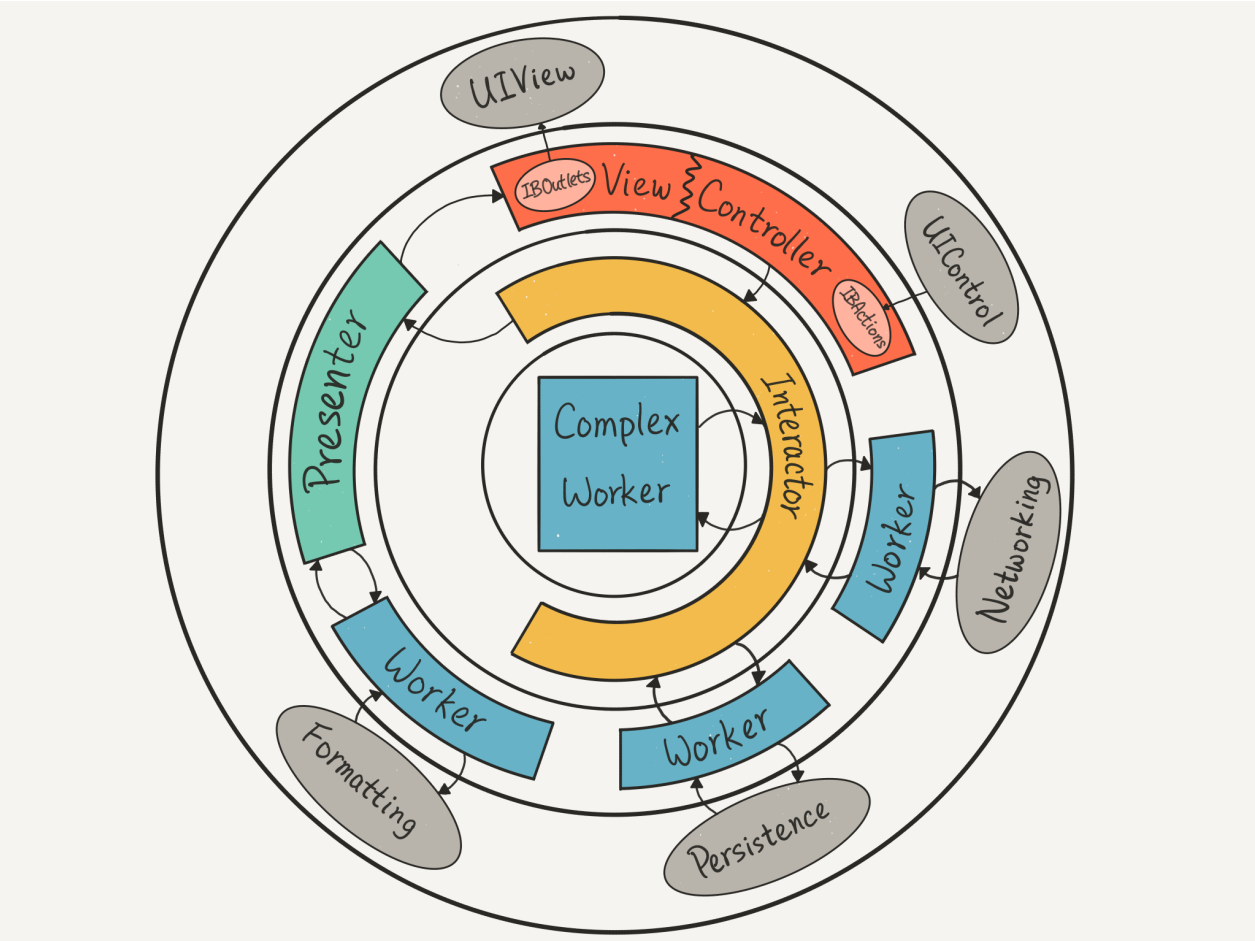
Alternatively, the business logic can also be triggered from the `viewDidLoad()` method. For example, your scene may need to fetch and display some data in a table view immediately upon being displayed.

The flow of control then moves to the interactor. The interactor either carries out the business logic itself, or asks a specialized worker for help. Such a worker may handle networking, persistence, in-app purchase, or some secret sauce of the business. When the work is done, the interactor invokes the presenter.

The job of the presenter is to format the results into an appropriate form for presentation to the user. The presenter then invoked the view controller to display the results in the UI.

It's important to note that the flow of control is **unidirectional**. It starts from the view controller, then gœs to the interactor, then to the presenter, and finally back to the view controller. It's also circular, so that, if desired, another use case can be triggered after the results from the first use case is received. This is why it's called the **VIP cycle**.

Here is how the VIP components fits in the original Clean Architecture.



The **View Controller** in an iOS app actually takes on two roles. First, it acts as the *controller*. When the user interacts with an `UIControl` such as an `UIButton`, the event triggers the invocation of an `IBAction` method. The view controller then triggers the interactor to perform some business logic. Second, it acts as the *view*. After the presenter formats the results, it then invokes the view controller so it can set some attributes of some `IBOutlets`, which are connected to some `UIViews` and subviews (as in a complex UI). The results are displayed to the user.

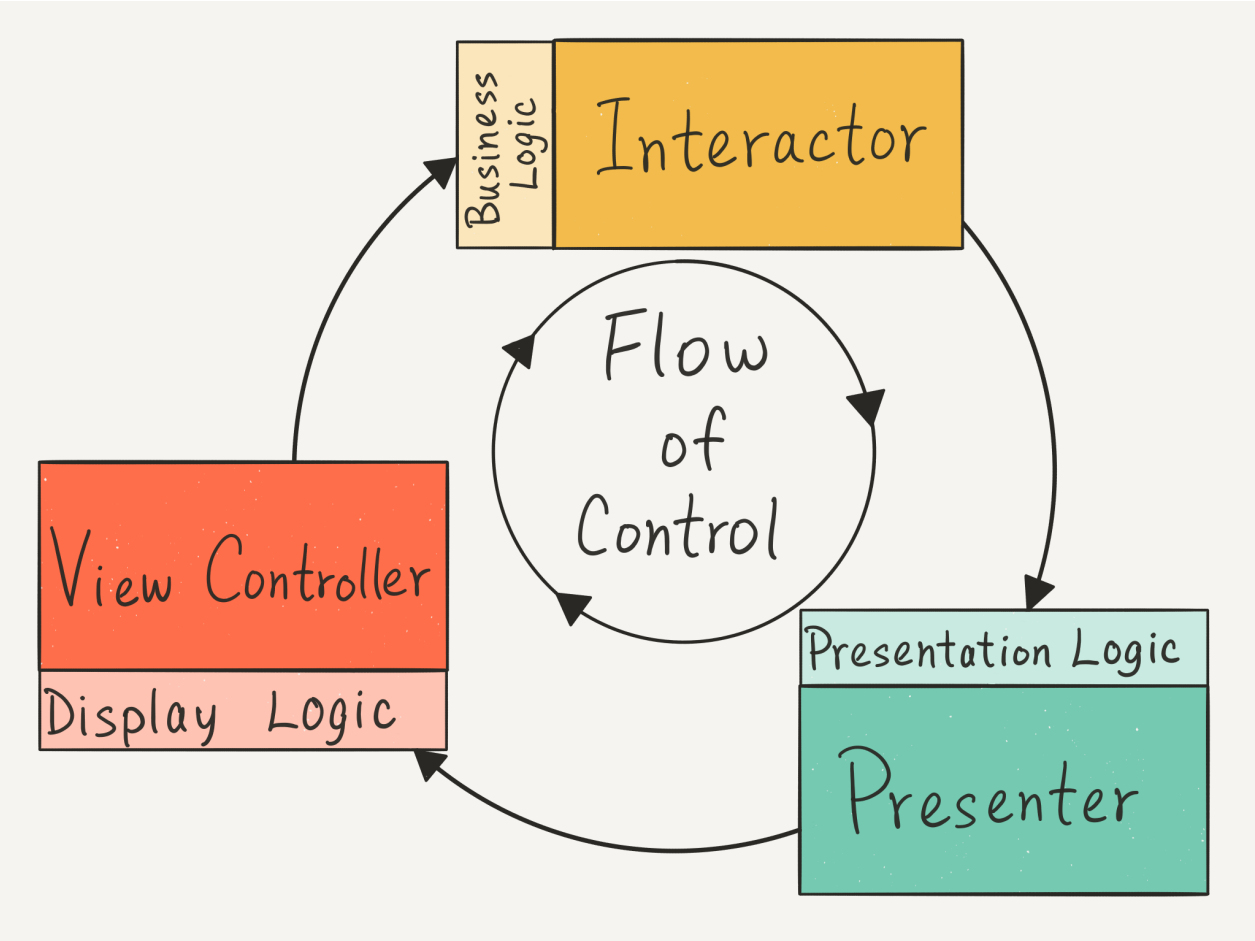
The **Interactor** performs the business logic of the iOS app. For simple things, it can do the work within the interactor. For more complex things, it can invoke the **Complex Worker** in the innermost layer. The work done in this layer is at a higher level of abstraction. It could be enterprise level (as opposed to just app level). The interactor can also invoke **Workers** in an outer layer for work done at a lower level of

abstraction, such as interfacing with other devices, networking, and persistence. This is the kind of low level implementation details, as well as technology choices that you can defer.

The **Presenter** receives the results of performing some business logic from the interactor. It then formats the results into displayable form, and passes it back to the view controller. Similar to the interactor, the presenter can handle simple formatting work within itself, or it can hand off to a worker to do the formatting. A formatting worker can also be shared with presenters in other scenes, if desired.

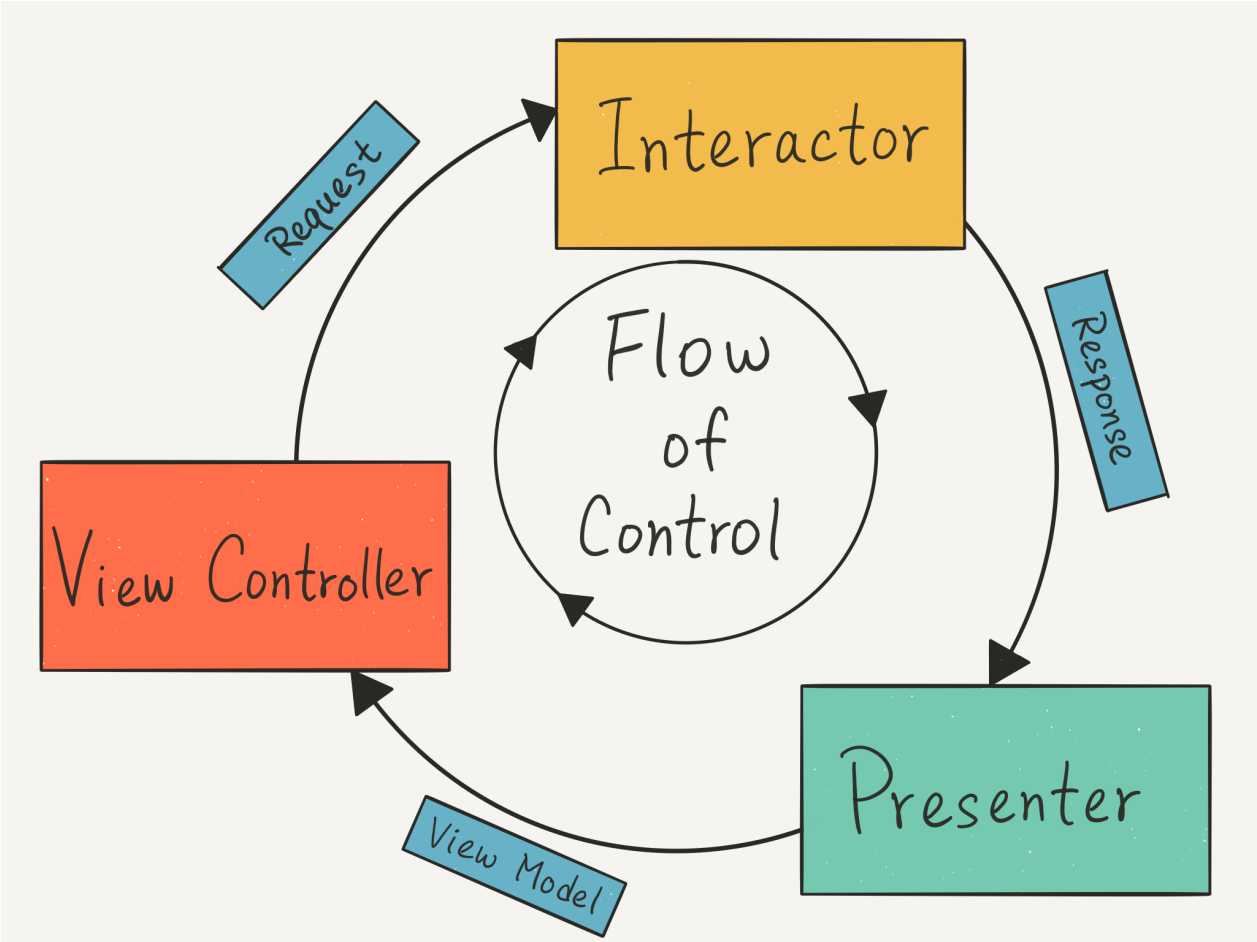
Each screen in an iOS app is called a **scene**, in storyboard terms. In most cases, one scene has one view controller. You can set up more complex parent-child view controller relationship, with the parent and each child have their own scenes. In addition, each scene also has its own worker for *within-the-same-scene* complex business. You can also have shared workers that contain business logic that needs to be used in multiple scenes.

To maintain **source code independence**, the same Dependency Inversion Principle used in the original Clean Architecture is applied at the boundaries between the components in the VIP cycle. For example, to reverse the dependence so that the interactor business logic dœsn't depend on the presenter, we use a Swift protocol so that the interactor invokes a method declared in this protocol instead of a method directly on the presenter. In theory, we can swap in a different presenter that also implements the same protocol. The interactor dœsn't need to change at all, and the app still works.



In order to be explicit about the role of each component, we name the boundary protocols as such. The interactor handles *business logic*, so it conforms to the **business logic** protocol. The presenter handles *presentation logic*, so it conforms to the **presentation logic** protocol. The view controller handles *displaying* results to the user, so it conforms to the **display logic** protocol.

To maintain **data independence**, we also make use of the same technique in the original Clean Architecture. We create special payload models to pass data through the boundaries between the components. This allows us to decouple the underlying data models used in the business entities from those used in the VIP components. We use Swift structs to define them and nest them under the use cases in the model file.



Again, to make explicit about their roles, we name them accordingly. The view controller *requests* the interactor to perform some business logic, so we name the payload model **request model** between the view controller and the interactor. The interactor sends the *response* after performing some business logic to the presenter, so we name the payload model **response model** between the interactor and presenter. The presenter sends the formatted results to the *view* controller for display, so we name the payload model **view model** between the presenter and the view controller.

This is important because the business logic can change such that it results in changes to 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 also conform to the corresponding protocols. The app still works as intended.

Ideally, these payload models 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. But it's okay to relax this rule a little, because many built-in types in iOS makes it convenient for us to use.

For example, your app may need to fetch some images over the network to display in a table view. If there are 250 rows in the table view, it's not efficient to have to fetch 250 images before returning a response. The UI will lock up. Even if you fetch the images asynchronously, the visible rows will be empty until the images are available. A better way to handle this situation is to pre-fetch and cache the images. You can use any CocoaPod library to do that so that you don't have to roll your own. Most such solutions do their magic in the view layer. It takes a `URL` object as an argument to one of its API calls. After the interactor determines the image URL, it passes this URL to the presenter. The presenter can then convert the `URL` object to a `String` in its view model and pass that to the view controller. But it's far more convenient to just pass the `URL` object as is to the view controller. You avoid having to convert from `URL` to `String` and back to `URL` just to stick to the primitive type rule.

To get the full picture of how the VIP cycle fits into your app, take a look at the accompanied VIP diagram that is included in this book. It shows more details and all components in the same diagram to give you the overall picture.

Now that you've been introduced to the VIP cycle. We'll continue our exploration of the Clean Swift iOS architecture by taking a detailed tour of its components. If you've followed the steps above to generate the `CreateOrder` scene, you already have these components in your Xcode project. They are located inside the `CleanStore —> Scenes —> CreateOrder` group.

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 should this *control code* look like? What qualifies as control code and what dœsn't?

Let's dive in. Open up the `CreateOrderViewController.swift` file. We'll walk through the code generated by the templates.

```
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 which 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 `CreateOrder` 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.

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.

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. This frees up your interactor which acts as a *coordinator* to coordinate tasks performed by different workers.

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()
  {
  }
}
```

If some business logic needs to be reused in multiple scenes, you can promote a worker to a shared worker. Just move the file up the group hierarchy so that it's outside of any specific group. This makes it clearer that the worker is not tied into any one particular scene, and thus is safe to be shared and reused.

You can also isolate a 3rd party library or framework dependency by restricting any calls to the dependency only inside a worker. There should not be any direct mention of the dependency through the VIP cycle. If the dependency needs to be updated, only the worker is affected. The rest of the codebase dœsn't need to be touched. You can even replace the dependency altogether. For instance, you can switch between the Apple provided `URLSession` with Alamofire to measure any performance improvements.

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.

```
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.

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.

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 `CreateOrder` 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 at https://clean-swift.com 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.

The Systematic Steps

When you're just starting, you can follow the steps below when you implement a new feature using Clean Swift. After you've done a couple features, it'll become second nature. For now, stick to these steps to get familiar with the process.

Assume we want to implement a use case named `Feature` in a screen named `Scene`.

1. Is this new feature to be implemented on a new or existing screen?
   - If new, create a new scene using the templates.
   - If existing, find the related scene by dry running in the simulator or inspecting the storyboards.
   - The scene is `.`
2. Is this new feature to be triggered from a view lifecycle event, or by some user action?
   - If former, override the desired method in the view controller.
   - If latter, open the related scene in the storyboard, create the `UIControl` element, and hook up to an `IBAction` method in the view controller.
   - The trigger point is `.`
3. Create the use case enum and the payload structs in `SceneModels.swift`:
   - `enum Feature`
     - `struct Request`
     - `struct Response`
     - `struct ViewModel`
4. Add method declarations to the VIP protocols:
   - `SceneBusinessLogic`
     - `func feature(request: Scene.Feature.Request)`
   - `ScenePresentationLogic`
     - `func presentFeature(response: Scene.Feature.Response)`
   - `SceneDisplayLogic`
     - `func displayFeature(viewModel: Scene.Feature.ViewModel)`

5. Add method definitions to the VIP classes:
    - `SceneViewController`
        - `func displayFeature(viewModel: Scene.Feature.ViewModel) { ... }`
    - `SceneInteractor`
        - `func feature(request: Scene.Feature.Request) { ... }`
    - `ScenePresenter`
        - `func presentFeature(response: Scene.Feature.Response) { ... }`
6. Set up the VIP cycle by having them call the next component in the chain.
    - From the trigger point in the view controller you identified above, invoke the business logic on the interactor:
        - `interactor?.feature(request: request)`
    - From the interactor, invoke the presentation logic on the presenter:
        - `presenter?.presentFeature(response: response)`
    - From the presenter, invoke the display logic on the view controller:
        - `viewController?.displayFeature(viewModel: viewModel)`
    - From the view controller, update the related `IBOutlet` elements to display the results to the user.
7. Finish the new feature by filling in the implementation in these methods.
8. All public methods need to be tested. Private methods shouldn't be tested. So we need to write unit tests for methods in the VIP protocols:
    - `SceneBusinessLogic`
        - `func feature(request: Scene.Feature.Request)`
    - `ScenePresentationLogic`
        - `func presentFeature(response: Scene.Feature.Response)`
    - `SceneDisplayLogic`
        - `func displayFeature(viewModel: Scene.Feature.ViewModel)`
    - Refer to Effective Unit Testing for everything you need to know about writing non-fragile unit tests and TDD for your iOS apps.
9. If your business logic is more complex, you have two options:

- You can move it to the `SceneWorker` class.
- If you also intend to share this business logic with other scenes, move it to a shared worker such as `AuthenticationWorker`.
- Remember to write unit tests for the public (not private) methods in any worker.

When you generate a new scene, an example use case named `Something` is already included for your convenience. You can copy/paste and rename it to save time from having to manually type the code above. So, steps 1–6 above are already taken care of by my Clean Swift Xcode templates. You can simply modify them to suit the needs of your new feature. These templates can also generate the unit test files for all VIP components, ready for you to write your first test out of the box. If you haven't, you can go to https://clean-swift.com to download them.

See what I mean by taking care of the boring stuff for you so that you can focus on the interesting stuff? The Clean Swift architecture sets things up properly for you. In steps 7–9, you have total freedom to do what you want.

Throughout the lifespan of your app, you inevitably have to make changes to the codebase. You may think of new features you want to add, existing features you want to improve, and critical bugs you need to fix. The Clean Swift system really shines in managing changes to your codebase. It helps maintain your app in good health. The following steps outline this process in much details.

Assume there is a screen named `Scene` with a use case named `Feature` that we want to improve or fix a bug for.

1. Figure out what kind of logic is involved so that you know which VIP component you should investigate:
   - If `SceneBusinessLogic`, look in `SceneInteractor`.
   - If `ScenePresentationLogic`, look in `ScenePresenter`.
   - If `SceneDisplayLogic`, look in `SceneViewController`.

    - If the change involves more than one kind of logic. Just start with the view controller, then interactor, then worker, then presenter, in that order.
2. Finish the change by filling in the implementation in these methods.
3. Find and update the related tests so that it specifies the changed requirements or correct behaviors.

Using the Clean Swift architecture, you simply figure out which kind of logic the changes should be made. You'll know exactly where the relevant code is. You no long need to read the entire massive view controllers to guess where to make your changes. Imagine how much time you can save and confidence you can gain when you maintain your apps for months and years.

The `CreateOrder` Use Case

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

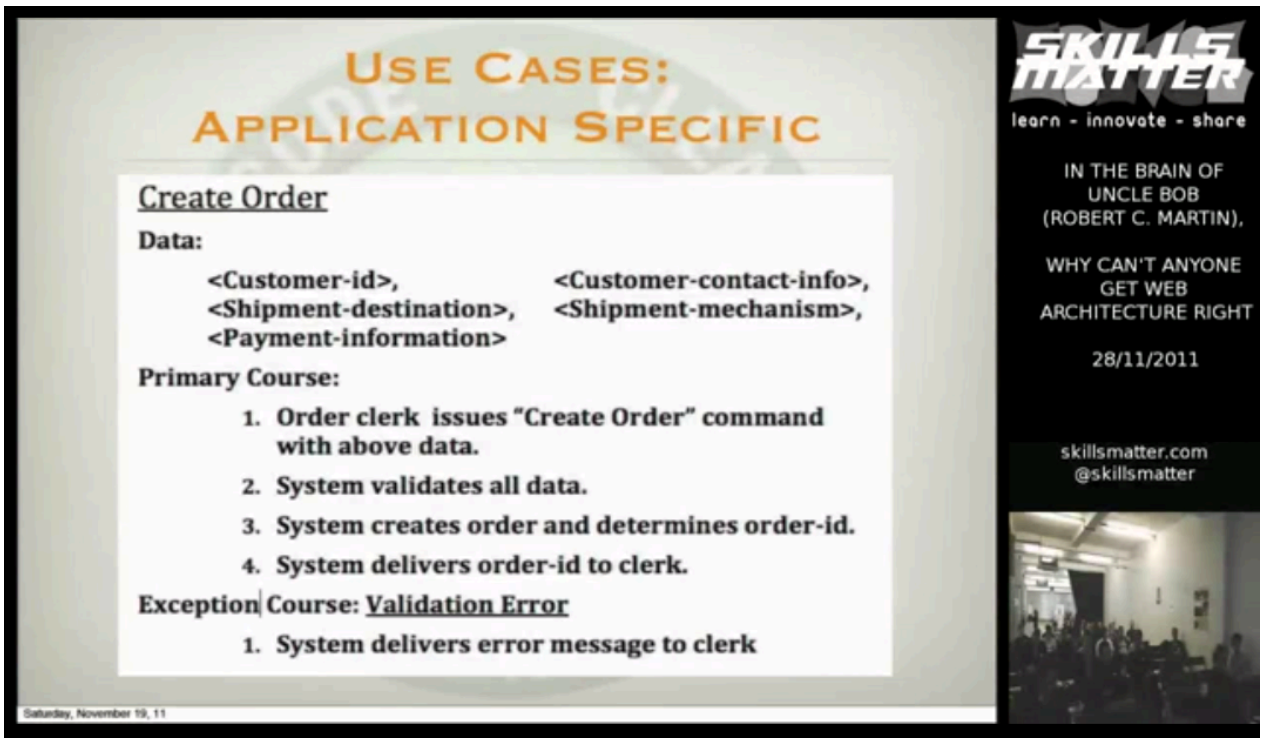
1. `CreateOrder`
2. `ListOrders`
3. `ShowOrder`

And we'll go through the entire app in this order. Let's begin with the `CreateOrder` scene, which implements the `CreateOrder` use case as outlined below.

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

To make it easy for you to see the parallelism between Clean Architecture for web apps and Clean Swift for iOS apps, I'm going to implement this same `CreateOrder` use case when I explain each component of the architecture for the rest of this book.

Here is the slide with the `CreateOrder` use case in Uncle Bob's presentation:



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.

Here's an overview of what we're going to do. We'll model the **entity data** as Swift structs in our model layer. To maintain data independence throughout the app, we create special **payload models** to cross the boundaries between components. These models are called **request**, **response**, and **view model**. The items listed under Primary Course and Exception Course are the **features** for this use case. We'll implement each of these features in isolated components. These components are called **view controller**, **interactor**, and **presenter**. The flow of control starts at the view controller to the interactor to the presenter and finally back to the view controller. Taking the first letter of these components, this cycle of control flow, as you already learned, is called the **VIP cycle**.

5. The `CreateOrder` Scene

Let's start with the `CreateOrder` use case in the original Clean Architecture presentation by Uncle Bob.

`CreateOrder` Data

Let's break down the `CreateOrder` 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.

`CreateOrder` 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 `CreateOrder` 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`.

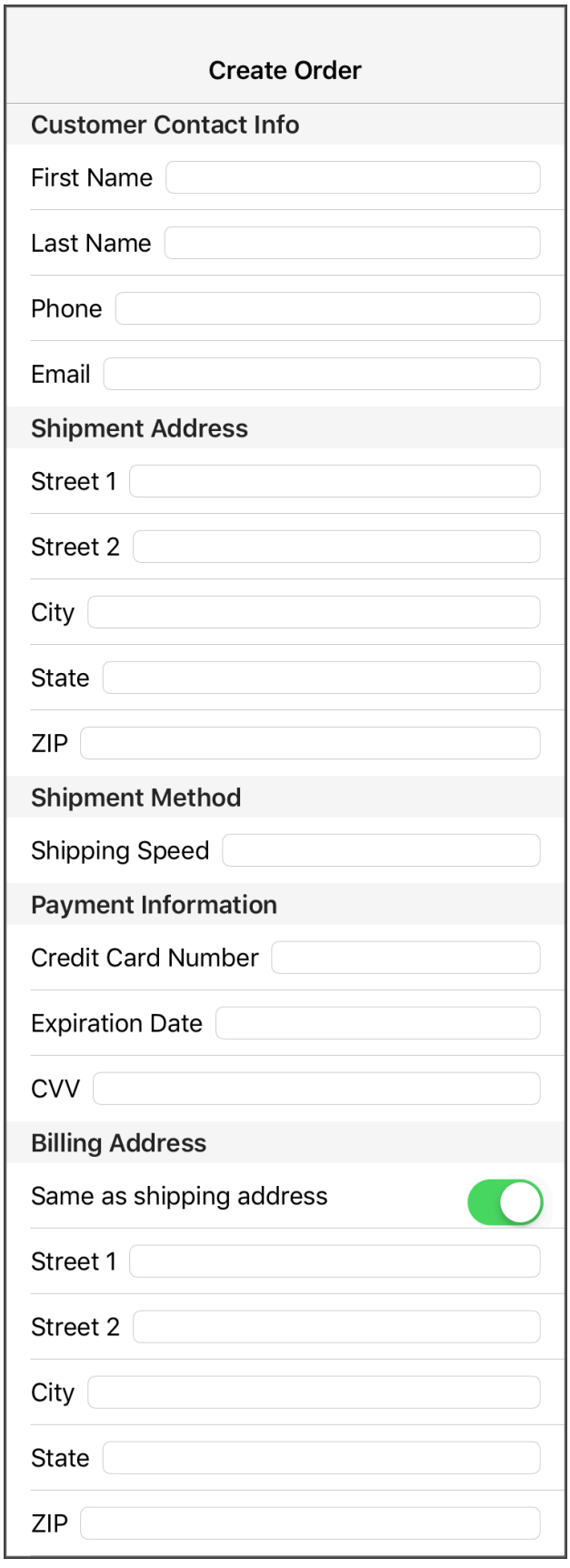
Make the following IBOutlet connections:

- 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 Changed* event and name it `expirationDatePickerValueChanged`.

Your form should look like:


Create Order
Customer Contact Info
First Name
Last Name
Phone
Email
Shipment Address
Street 1
Street 2
City
State
ZIP
Shipment Method
Shipping Speed
Payment Information
Credit Card Number
Expiration Date
CVV
Billing Address
Same as shipping address
Street 1
Street 2
City
State
ZIP

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: Any)
{
}
```

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.index(of: 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(_:didSelectRowAt:)` method.

```
  override func tableView(_ tableView: UITableView, didSelectRowAt
indexPath: IndexPath) {
  {
    if let cell = tableView.cellForRow(at: indexPath) {
      for textField in textFields {
        if textField.isDescendant(of: 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 numberOfComponents(in pickerView: UIPickerView) -> Int
  {
    return 1
  }

  func pickerView(_ pickerView: UIPickerView, numberOfRowsInCompo-
nent component: Int) -> Int
  {
    return interactor?.shippingMethods.count ?? 0
  }

  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: Any)
```

```
    {
      let date = expirationDatePicker.date
      let request = CreateOrder.FormatExpirationDate.Request(date:
  date)
      interactor?.formatExpirationDate(request: 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: response)
  }
}
```

Implement Expiration Date Presentation Logic in the Presenter

In the `formatExpirationDate(request:)` 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 does 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 doesn'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.

6. The `ListOrders` Scene

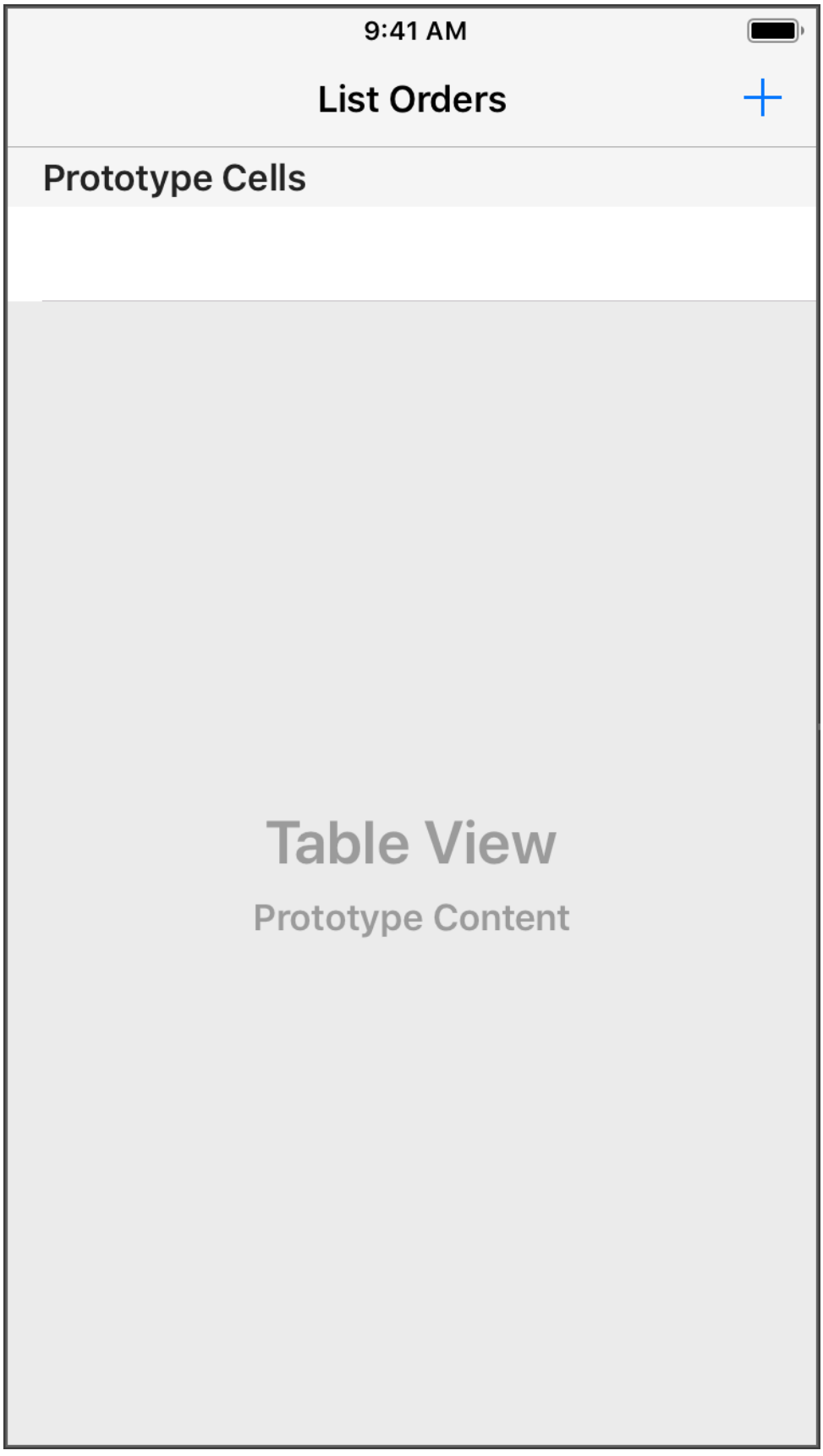
Next, we'll tackle the `ListOrders` scene. Its job is simple - showing a list or orders in the all familiar table view. Using traditional MVC, you can probably do this in 15 minutes with one eye closed. Stop! We want to avoid going down the MVC rabbit hole. You're here to learn how to write clean Swift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

Design the `ListOrders` Scene in Storyboard and View Controller

First, follow the exact same steps as you did above to create the `ListOrders` scene.

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:


9:41 AM
List Orders
Prototype Cells
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 does. 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`. So, highlight the `CleanStore` group again, and choose `File -> New -> Group`. Name it **Workers** this time. This new group should be parallel to the `Scenes` group, and will contain all our shared workers that we can share with multiple scenes.

Next, highlight this new `Workers` group, choose `File –> New –> File`, and simply pick **Swift File** to create the `OrdersWorker`. Below is a trivial implementation for the `OrdersWorker` class:

```
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.

For now, let's create the missing `Order` model first. Create a new `Models` group parallel to the `Workers` group, and then a new file named `Order.swift`. Let's not distract ourselves and keep the focus on architecture. So, we simply use the following `Order` struct.

```
struct Order: Equatable
{
  // MARK: Contact info
  var firstName: String
  var lastName: String
  var phone: String
  var email: String

  // MARK: Payment info
  var billingAddress: Address
  var paymentMethod: PaymentMethod

  // MARK: Shipping info
  var shipmentAddress: Address
  var shipmentMethod: ShipmentMethod

  // MARK: Misc
  var id: String?
  var date: Date
  var total: NSDecimalNumber
```

```
}

func ==(lhs: Order, rhs: Order) -> Bool
{
  return lhs.firstName == rhs.firstName
    && lhs.lastName == rhs.lastName
    && lhs.phone == rhs.phone
    && lhs.email == rhs.email
    && lhs.billingAddress == rhs.billingAddress
    && lhs.paymentMethod == rhs.paymentMethod
    && lhs.shipmentAddress == rhs.shipmentAddress
    && lhs.shipmentMethod == rhs.shipmentMethod
    && lhs.id == rhs.id
    && lhs.date.timeIntervalSince(rhs.date) < 1.0
    && lhs.total == rhs.total
}

// MARK: - Supporting models

struct Address
{
  var street1: String
  var street2: String?
  var city: String
  var state: String
  var zip: String
}

func ==(lhs: Address, rhs: Address) -> Bool
{
  return lhs.street1 == rhs.street1
    && lhs.street2 == rhs.street2
    && lhs.city == rhs.city
    && lhs.state == rhs.state
    && lhs.zip == rhs.zip
}

struct ShipmentMethod
{
  enum ShippingSpeed: Int {
    case Standard = 0 // "Standard Shipping"
    case OneDay = 1 // "One-Day Shipping"
    case TwoDay = 2 // "Two-Day Shipping"
  }
  var speed: ShippingSpeed
```

```
    func toString() -> String
    {
      switch speed {
      case .Standard:
        return "Standard Shipping"
      case .OneDay:
        return "One-Day Shipping"
      case .TwoDay:
        return "Two-Day Shipping"
      }
    }
  }

  func ==(lhs: ShipmentMethod, rhs: ShipmentMethod) -> Bool
  {
    return lhs.speed == rhs.speed
  }

  struct PaymentMethod
  {
    var creditCardNumber: String
    var expirationDate: Date
    var cvv: String
  }

  func ==(lhs: PaymentMethod, rhs: PaymentMethod) -> Bool
  {
    return lhs.creditCardNumber == rhs.creditCardNumber
      && lhs.expirationDate.timeIntervalSince(rhs.expirationDate) <
  1.0
      && lhs.cvv == rhs.cvv
  }
```

Let's go back to the `CreateOrderInteractor` class quickly and update our `shippingMethods` to use the new `ShipmentMethod` as follows:

```
    var shippingMethods = [
      ShipmentMethod(speed: .Standard).toString(),
      ShipmentMethod(speed: .OneDay).toString(),
      ShipmentMethod(speed: .TwoDay).toString()
    ]
```

Getting back to the `ListOrdersInteractor`, we first instantiate an `OrdersWorker` object by injecting an `OrdersMemStore`. Running it off the memory is always the

fastest and simplest. Let's create a new `Services` group parallel to the `Workers` group, and then a new file named `OrdersMemStore.swift`. Again, we'll just use a quick and dirty `OrdersMemStore` implementation:

```
class OrdersMemStore: OrdersStoreProtocol
{
  // MARK: - Data

  static var billingAddress = Address(street1: "1 Infinite Loop",
street2: "", city: "Cupertino", state: "CA", zip: "95014")
  static var shipmentAddress = Address(street1: "One Microsoft
Way", street2: "", city: "Redmond", state: "WA", zip: "98052-7329")
  static var paymentMethod = PaymentMethod(creditCardNumber: "1234-
123456-1234", expirationDate: Date(), cvv: "999")
  static var shipmentMethod = ShipmentMethod(speed: .OneDay)

  static var orders = [
    Order(firstName: "Amy", lastName: "Apple", phone:
"111-111-1111", email: "amy.apple@clean-swift.com", billingAddress:
billingAddress, paymentMethod: paymentMethod, shipmentAddress:
shipmentAddress, shipmentMethod: shipmentMethod, id: "abc123",
date: Date(), total: NSDecimalNumber(string: "1.23")),
    Order(firstName: "Bob", lastName: "Battery", phone:
"222-222-2222", email: "bob.battery@clean-swift.com", billingAd-
dress: billingAddress, paymentMethod: paymentMethod, shipmentAd-
dress: shipmentAddress, shipmentMethod: shipmentMethod, id: "de-
f456", date: Date(), total: NSDecimalNumber(string: "4.56"))
  ]

  // MARK: - CRUD operations - Optional error

  func fetchOrders(completionHandler: @escaping (() throws -> [Or-
der]) -> Void)
  {
    completionHandler { return type(of: self).orders }
  }
}
```

This `OrdersMemStore` class conforms to the `OrdersStoreProtocol` we defined earlier. It implements the only method in the protocol by returning a hardcoded list of `Order` objects we defined right in the class. For a real app, you want to fetch the orders from a database such as Core Data or Realm, or over the network using an API backend. But we'll keep it focused here.

Continuing with the `ListOrdersInteractor`, we need to implement the `fetchOrders(request:)` method as follows:

```
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)
    }
  }
}
```

We simply invoke the `OrdersWorker`, which we instantiated earlier, to fetch the orders asynchronously. 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.

7. The `ShowOrder` Scene

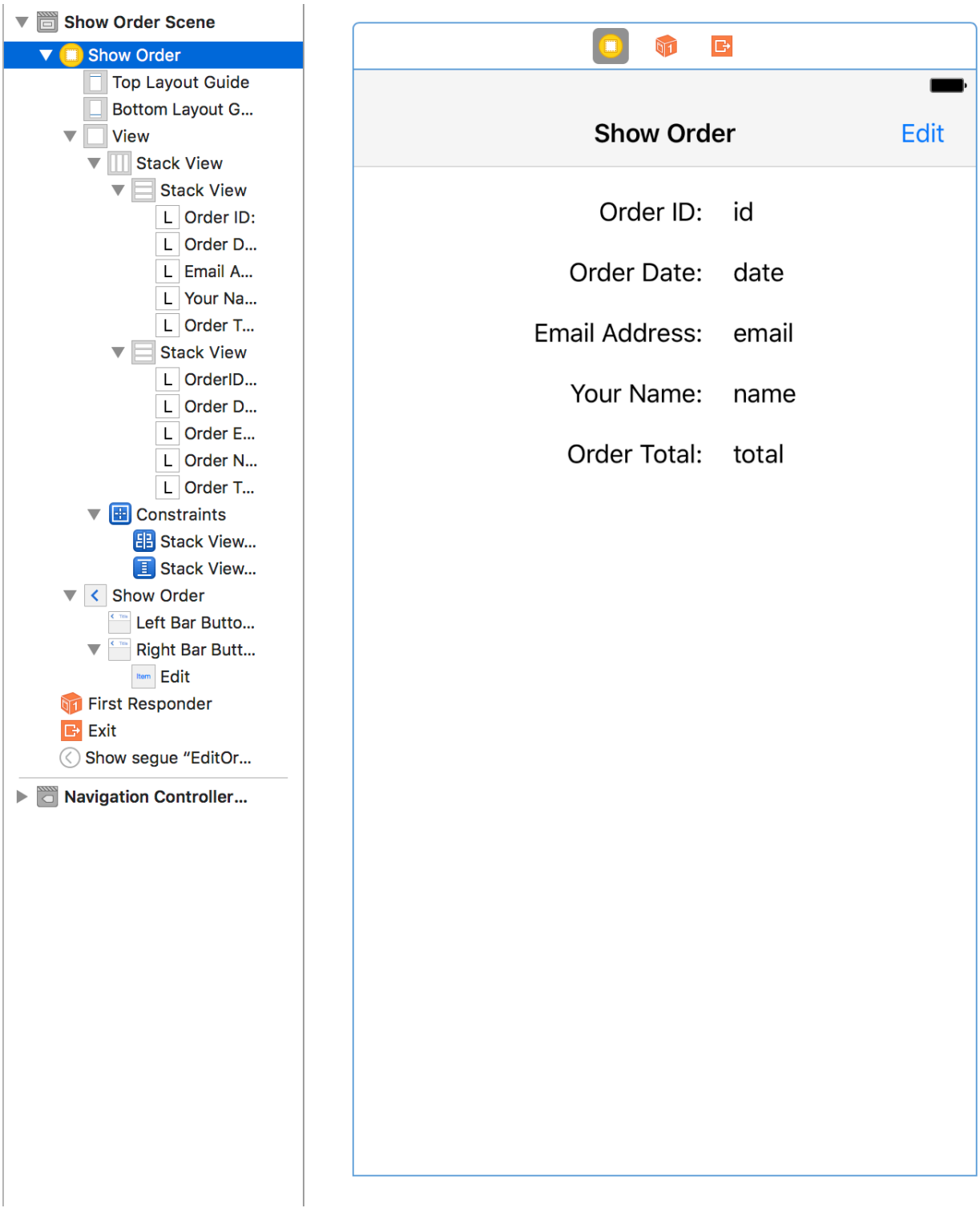
Our third and final scene is the `ShowOrder` scene. It displays the order details to the user.

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 does 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 does 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.

8. 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 of the destination - `routeToNextScene(segue:)`
2. Passing data to the destination - `passDataToNextScene(source:destination:)`
3. Navigating to the destination - `navigateToNextScene(source:destination:)`

That's all you need to know at this point to proceed with adding routes to the `ListOrders` scene. In a future chapter, you'll learn all the details about routing in Clean Swift.

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 of 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 of 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 from the source data store (i.e. `ListOrdersDataStore`) is passed to the destination data store (i.e. `ShowOrderDataStore`).

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

We simply need to specify the data to be passed *from* the **source** data store:

```
protocol ListOrdersDataStore
{
  var orders: [Order]? { get }
}

class ListOrdersInteractor: ListOrdersBusinessLogic, ListOrders-
DataStore
{
  // ...
  var orders: [Order]?
  // ...
}
```

And *to* the **destination** data store:

```
protocol ShowOrderDataStore
{
  var order: Order! { get set }
}

class ShowOrderInteractor: ShowOrderBusinessLogic, ShowOrderDataS-
tore
{
  // ...
  var order: Order!
  // ...
}
```

It's a simple copy-and-paste. Because we only need to read the order out from the source data store, a getter suffices. But we need to set the order in the destination data store, so we need a getter and setter.

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.

9. 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 of 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 in the `ShowOrder` scene is passed to the `CreateOrder`

scene to be edited (yes, we're reusing the same `CreateOrder` scene to both create and update an order).

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

The source data store (i.e. `ShowOrderDataStore`) already specifies the order when we did the `ListOrders` –> `ShowOrder` route, so we don't need to do it again. But we do need to add the XXX to the destination data store (i.e. `CreateOrderDataStore`).

```
protocol CreateOrderDataStore
{
  var orderToEdit: Order? { get set }
}

class CreateOrderInteractor: CreateOrderBusinessLogic, CreateOrder-
DataStore
{
  // ...
  var orderToEdit: Order?
  // ...
}
```

You've seen this before. We just need to specify the data to be received with a getter and setter in the protocol.

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 dœs.

10. 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:

- Save the created order
- Populate the order form with the existing order details
- Save the updated 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. Since you already saw the VIP cycle and how the architecture works, I'll go a little faster in the explanation that follows. If you need help, just jump in the Slack team for the included mentorship program and ask a question.

For these use cases, we work extensively with the order form. Specifically, we need to retrieve the text field values. It would be hugely convenient if we define the following `OrderFormFields` struct in the `CreateOrderModels.swift` file:

```
enum CreateOrder
{
  struct OrderFormFields
  {
    // MARK: Contact info
    var firstName: String
    var lastName: String
    var phone: String
    var email: String

    // MARK: Payment info
    var billingAddressStreet1: String
    var billingAddressStreet2: String
```

```
    var billingAddressCity: String
    var billingAddressState: String
    var billingAddressZIP: String

    var paymentMethodCreditCardNumber: String
    var paymentMethodCVV: String
    var paymentMethodExpirationDate: Date
    var paymentMethodExpirationDateString: String

    // MARK: Shipping info
    var shipmentAddressStreet1: String
    var shipmentAddressStreet2: String
    var shipmentAddressCity: String
    var shipmentAddressState: String
    var shipmentAddressZIP: String

    var shipmentMethodSpeed: Int
    var shipmentMethodSpeedString: String

    // MARK: Misc
    var id: String?
    var date: Date
    var total: NSDecimalNumber
  }


  // MARK: Use cases

  // ...


  enum CreateOrder
  {
    struct Request
    {
      var orderFormFields: OrderFormFields
    }
    struct Response
    {
      var order: Order?
    }
    struct ViewModel
    {
      var order: Order?
    }
  }
}
```

We also define the `CreateOrder` use case here. The user inputs are collected from the text fields in the view controller. Instead of listing every field, we just use `OrderFormFields` in the request. An `Order` object is returned in the response.

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.

Alternatively, you can use the `text` property, or put this in an extension or another file for brevity, it that's reasonably called.

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)
  }
}
```

For simplicity, we just 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.

The `createOrder(request:)` method first calls the private `buildOrderFromOrderFormFields(_:)` method to build the `Order` object, and then calls the `OrdersWorker`'s `createOrder(orderToCreate:completionHandler:)` method to create the order asynchronously. This means we need to add this method to the `OrdersStoreProtocol` protocol:

```
protocol OrdersStoreProtocol
{
  // ...
  func createOrder(orderToCreate: Order, completionHandler: @escap-
ing (() throws -> Order?) -> Void)
}
```

And the method implementation to the `OrdersWorker` class:

```
class OrdersWorker
{
  // ...
  func createOrder(orderToCreate: Order, completionHandler: @escap-
ing (Order?) -> Void)
  {
    ordersStore.createOrder(orderToCreate: orderToCreate) { (order:
() throws -> Order?) -> Void in
      do {
        let order = try order()
        DispatchQueue.main.async {
          completionHandler(order)
        }
```

```
        } catch {
          DispatchQueue.main.async {
            completionHandler(nil)
          }
        }
      }
    }
  }
```

For the `OrdersMemStore`'s implementation of the `createOrder(orderToCreate:completionHandler:)` method, we need to create a complete order by generating an order ID and calculating the order total. Finally, we add the newly created order to *memory* by appending it to the `orders` array that we keep in the store itself.

```
class OrdersMemStore: OrdersStoreProtocol, OrdersStoreUtilityProto-
col
{
  // ...
  func createOrder(orderToCreate: Order, completionHandler: @escap-
ing (() throws -> Order?) -> Void)
  {
    var order = orderToCreate
    generateOrderID(order: &order)
    calculateOrderTotal(order: &order)
    type(of: self).orders.append(order)
    completionHandler { return order }
  }
  // ...
}
```

Since it's a memory store just to get things going, we don't really care how it's done. So we just fake it with the following `OrdersStoreUtilityProtocol`, and provide a default implementation using a Swift extension:

```
protocol OrdersStoreUtilityProtocol {}

extension OrdersStoreUtilityProtocol
{
  func generateOrderID(order: inout Order)
  {
    guard order.id == nil else { return }
    order.id = "\(arc4random())"
  }
```

```
    func calculateOrderTotal(order: inout Order)
    {
      guard order.total == NSDecimalNumber.notANumber else { return }
      order.total = NSDecimalNumber.one
    }
  }
```

In a real app, you definitely want to make sure each order ID is unique, probably generated by the server. After all, you don't want a customer's iPhone order ends up in another customer's doorstep!

Back to the `CreateOrderInteractor`'s `createOrder(request:)` method. 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. We'll take another good look at routing in a future chapter to learn about programmatic routes v.s. segue routes. For now, to get the code to compile, simply add an empty `routeToListOrders(segue:)` method to the `CreateOrdersRouter` class as follows:

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

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

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

```
    // MARK: Routing

    func routeToListOrders(segue: UIStoryboardSegue?)
    {

    }

    // MARK: Navigation

    // MARK: Passing data

  }
```

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. A simple implementation using the built-in `UIAlertController` is shown below:

```
    private func showOrderFailureAlert(title: String, message:
  String)
    {
      let alertController = UIAlertController(title: title, message:
  message, preferredStyle: .alert)
      let alertAction = UIAlertAction(title: "OK", style: .default,
  handler: nil)
      alertController.addAction(alertAction)
      showDetailViewController(alertController, sender: nil)
    }
```

To actually have the newly created order show up in the `ListOrders` scene, we need to modify the `ListOrdersViewController` a little bit:

```
  class ListOrdersViewController: UITableViewController, ListOrders-
  DisplayLogic
  {
    // ...
    override func viewWillAppear(_ animated: Bool)
    {
      super.viewWillAppear(animated)
      fetchOrdersOnLoad()
    }
    // ...
  }
```

We move the call to `fetchOrdersOnLoad()` from `viewDidLoad()` to `viewWillAppear(_:)`, so that the orders are re-fetched whenever the `ListOrders` scene re-appears on the screen. This happens when we just finished creating a new order.

Populate the Order Form

The same `CreateOrder` scene is used for both creating a new order and updating an existing order. When we have an existing order, we don't want the user to enter all the order details again. We have the order details from the existing order that we can pre-populate the order form. So, let's 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. The `EditOrder` use case is added to the `CreateOrderModels.swift` file:

```
enum CreateOrder
{
  // ...
  enum EditOrder
  {
    struct Request
    {
    }
    struct Response
    {
```

```
      var order: Order
    }
    struct ViewModel
    {
      var orderFormFields: OrderFormFields
    }
  }
  // ...
}
```

The interactor finds and returns an `Order` object. The presenter converts the order details to a `OrderFormFields` object, such that the view controller can use to fill in the text fields in the UI.

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)
    }
```

The presenter turns the `Order` object into a `OrderFormFields` object, and pass it back to the view controller for the last mile

Implement Show Order To Edit Display Logic in the View Controller

Back at the `CreateOrderViewController`, add the following method to complete the VIP cycle for the `EditOrder` use case:

```
    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 the appropriate value 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.

Change the `saveButtonTapped(_:)` method to look like the following:

```
@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.

To allow for this check, we need to add the `orderToEdit` property to the `CreateOrderBusinessLogic` protocol so that the view controller can access it.

```
protocol CreateOrderBusinessLogic
{
  var shippingMethods: [String] { get }
  var orderToEdit: Order? { get }
  func formatExpirationDate(request: CreateOrder.FormatExpiration-
Date.Request)
  func createOrder(request: CreateOrder.CreateOrder.Request)
  func showOrderToEdit(request: CreateOrder.EditOrder.Request)
}
```

As you should know by now, you also need to add the corresponding `UpdateOrder` use case in `CreateOrderModels.swift`:

```
enum CreateOrder
{
  // ...
  enum UpdateOrder
  {
    struct Request
    {
      var orderFormFields: OrderFormFields
    }
    struct Response
    {
      var order: Order?
    }
    struct ViewModel
    {
      var order: Order?
    }
  }
  // ...
}
```

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.

Update the `OrdersStoreProtocol` protocol to look like:

```
protocol OrdersStoreProtocol
{
  func fetchOrders(completionHandler: @escaping (() throws -> [Or-
der]) -> Void)
  func createOrder(orderToCreate: Order, completionHandler: @escap-
ing (() throws -> Order?) -> Void)
  func updateOrder(orderToUpdate: Order, completionHandler: @escap-
ing (() throws -> Order?) -> Void)
}
```

And implement the new `updateOrder(orderToUpdate:completionHandler:)` method in the `OrdersWorker` class:

```
class OrdersWorker
{
  // ...
  func updateOrder(orderToUpdate: Order, completionHandler: @escap-
ing (Order?) -> Void)
  {
    ordersStore.updateOrder(orderToUpdate: orderToUpdate) { (order:
() throws -> Order?) in
      do {
        let order = try order()
        DispatchQueue.main.async {
          completionHandler(order)
        }
      } catch {
        DispatchQueue.main.async {
          completionHandler(nil)
        }
      }
    }
  }
}
```

Finally, implement it in the `OrdersMemStore`:

```
class OrdersMemStore: OrdersStoreProtocol, OrdersStoreUtilityProto-
col
```

```
{
  // ...
  func updateOrder(orderToUpdate: Order, completionHandler: @escap-
ing (() throws -> Order?) -> Void)
  {
    if let index = indexOfOrderWithID(id: orderToUpdate.id) {
      type(of: self).orders[index] = orderToUpdate
      let order = type(of: self).orders[index]
      completionHandler { return order }
    } else {
      completionHandler { throw OrdersStoreError.CannotUpdate("Can-
not fetch order with id \(String(describing: orderToUpdate.id)) to
update") }
    }
  }
}
```

Let's add a little twist to this use case to keep it from being too boring. What if updating an order generates an error? It can fail for many reasons. Let's say the order to be updated dœsn't exist. That means the order ID cannot be found. This happens when the `indexOfOrderWithID(id:)` method returns nil. We want to return a nice error object with a message upstream for the interactor to decide what to do.

In the `OrdersWorker.swift` file, add the following `OrdersStoreError` enum:

```
enum OrdersStoreError: Equatable, Error
{
  case CannotFetch(String)
  case CannotCreate(String)
  case CannotUpdate(String)
  case CannotDelete(String)
}

func ==(lhs: OrdersStoreError, rhs: OrdersStoreError) -> Bool
{
  switch (lhs, rhs) {
  case (.CannotFetch(let a), .CannotFetch(let b)) where a == b: re-
turn true
  case (.CannotCreate(let a), .CannotCreate(let b)) where a == b:
return true
  case (.CannotUpdate(let a), .CannotUpdate(let b)) where a == b:
return true
  case (.CannotDelete(let a), .CannotDelete(let b)) where a == b:
return true
```

```
    default: return false
    }
  }
```

We'll use Swift enum's associate values feature to add an error message string. The == equality operator allows us to easily compare errors.

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.")
    }
  }
```

On failure, the app shows an alert to the user if the order cannot be updated, because the order ID is not found. On success, it routes back to the `ShowOrder` scene. If you're

a careful reader, you'll notice this is a backward route. We're almost at the routing chapter, and your patience will be rewarded.

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. Similarly, simply add an *empty* `ShowOrder` route to the router for now.

```
@objc protocol CreateOrderRoutingLogic
{
  func routeToListOrders(segue: UIStoryboardSegue?)
  func routeToShowOrder(segue: UIStoryboardSegue?)
}

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

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

  // MARK: Routing

  func routeToListOrders(segue: UIStoryboardSegue?)
  {

  }

  func routeToShowOrder(segue: UIStoryboardSegue?)
  {

  }

  // MARK: Navigation

  // MARK: Passing data

}
```

11. Routing from the `CreateOrder` Scene

Let's fill in the empty routes now. 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 back to the `ListOrders` scene, which will refresh the table view with the new order being listed.

Step 1.
Getting a hold of 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 back to the `ShowOrder` scene, which will refresh with the updated order details begin shown.

Step 1.

Getting a hold of 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.

12. Routing in Details

As you've already learned when you added the routes to the three scenes for the CleanStore app, routing is a 3-step process. The router cleanly separates and carries out these 3 responsibilities. The steps are:

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

At each step, you only need to do one thing. In step 1, you retrieve the destination scene (view controller and data store). In step 2, you pass data to the destination data store. In step 3, you navigate to the destination view controller.

What about the source scene? You already have the source scene. If you're routing from the `ListOrders` scene to the `ShowOrder` scene, you put the code for the route in the source scene's router - the `ListOrdersRouter` class. The router generated by the templates already provide you with the source view controller and data store.

```
class ListOrdersRouter: NSObject, ListOrdersRoutingLogic, List-
OrdersDataPassing
{
  weak var viewController: ListOrdersViewController?
  var dataStore: ListOrdersDataStore?
  // ...
}
```

Another benefit of following this 3-step routing process is its flexibility. You have the option to trigger a route in 3 different ways:

1. Automatic segue route
2. Manual segue route
3. Programmatic route

Automatic Segue Route

To create an **automatic segue route**, in the storyboard, you control-drag from a `UIControl` to the destination view controller. For example, you can drag from a button or table view cell to the destination view controller. Because the trigger is already defined to be the button or table view cell, iOS will automatically perform the segue when the UI element is acted upon by the user. That is, when the button is tapped or the table view cell is selected. The segue is performed automatically because you, the developer, don't have to write any code to make that happen. In terms of the 3-step routing process, you only need to take care of step 1 (`routeToNextScene(segue:)`) and step 2 (`passDataToNextScene(source:destination:)`). You let iOS take care of step 3.

Manual Segue Route

To create a **manual segue route**, in the storyboard, you control-drag from the view controller object in the source scene to the destination view controller. In essence, you're telling iOS that there needs to be a route going from source to destination. But the trigger point is not fixed to any UI element, as it is to be determined at runtime. Therefore, you're responsible for triggering the router when appropriate, by manually calling the `performSegue(withIdentifier:sender:)` method yourself. In terms of the 3-step routing process, you only need to take care of step 1 (`routeToNextScene(segue:)`) and step 2 (`passDataToNextScene(source:destination:)`). You let iOS take care of step 3.

With both automatic and manual segues, you want to remember to give an identifier for the segue in the storyboard. This is important because the `prepare(for:sender:)` method in the view controller uses this segue identifier to match the route method name in the router.

```
    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)
        }
      }
    }
```

The `NSSelectorFromString("routeTo\(scene)WithSegue:")` macro creates a selector with the segue identifier `scene` as part of the method name and a parameter named `segue`. The `responds(to:)` method checks to see if the router has a method with this selector. If it does, it calls `perform(_:with:)` to trigger the route. So it's critical that your segue identifier matches your route method name.

Programmatic Route

With a `programmatic route`, you don't create any segue in the storyboard. Instead, you trigger the route by calling the `routeToNextScene(segue:)` method and pass nil for the `segue` parameter (It's not a segue route, so we don't have a segue). In terms of the 3-step routing process, you need to take care of step 1 (`routeToNextScene(segue:)`), step 2 (`passDataToNextScene(source:destination:)`), and step 3 (`navigateToNextScene(source:destination:)`). Steps 1 and 2 are the same as with segue routes. In step 3, inside the `navigateToNextScene(source:destination:)` method, you write the code to present the view controller programmatically, by calling one of these methods:

- On `UIViewController`:
  - `show(_:sender:)`
  - `showDetailViewController(_:sender:)`
  - `present(_:animated:completion:)`
- On `UINavigationController`:
  - `pushViewController(_:animated:)`
- On `UITabBarController`:
  - `selectedIndex`

Getting a hold of the destination in step 1 is a little different with programmatic routes. Let's take a look at the `routeToShowOrder(segue:)` method of the `ListOrdersRouter` class:

```
    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)
    }
```

Since we don't have a segue to conveniently get the destination view controller, we have to get it some other way. We can invoke the `UIStoryboard`'s `instantiateViewController(withIdentifier:)` method to instantiate an object of the destination view controller type. 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 of the method is pretty much the same as for segue routes. Make sure to include a call to `navigateToNextScene(source:destination:)` at the end.

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)
  }
```

One Size Fits All

To get the best of all three worlds, we can make routing work for all 3 route types. You just need to check for the availability of a segue in order 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)
    }
  }
```

If `segue` is not nil, this is either an automatic or manual segue route. The first branch of the `if-else` statement takes care of both segue route types. It retrieves the destination view controller from the `segue` argument. It also skips step 3 by not calling the `navigateToShowOrder(source:destination:)` method.

If `segue` is nil, this is a programmatic route. The second branch of the `if-else` statement retrieves the destination view controller by instantiating one from the

storyboard. It also performs step 3 by calling the `navigateToShowOrder(source:destination:)` method.

This modified `routeToShowOrder(segue:)` method works for all scenarios. Likewise, you can modify other route methods in all routers throughout your app in the same way. This gives you the flexibility as to how you want to trigger your routes. It's all-you-can-eat. So pick whatever you like.

The Full Picture

Let's go back to our first `CreateOrder` scene, and take a look at its router in its completeness. We'll walk through all routing types in order to understand the routing process from end to end.

The complete `CreateOrderRouter` class is shown below:

```
@objc protocol CreateOrderRoutingLogic
{
  func routeToListOrders(segue: UIStoryboardSegue?)
  func routeToShowOrder(segue: UIStoryboardSegue?)
}

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

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

  // MARK: Routing

  func routeToListOrders(segue: UIStoryboardSegue?)
  {
    if let segue = segue {
```

```
      let destinationVC = segue.destination as! ListOrdersViewCon-
troller
      var destinationDS = destinationVC.router!.dataStore!
      passDataToListOrders(source: dataStore!, destination: &desti-
nationDS)
    } else {
      let index = viewController!.navigationController!.viewCon-
trollers.count - 2
      let destinationVC =
viewController?.navigationController?.viewControllers[index] as!
ListOrdersViewController
      var destinationDS = destinationVC.router!.dataStore!
      passDataToListOrders(source: dataStore!, destination: &desti-
nationDS)
      navigateToListOrders(source: viewController!, destination:
destinationVC)
    }
  }

  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 index = viewController!.navigationController!.viewCon-
trollers.count - 2
      let destinationVC =
viewController?.navigationController?.viewControllers[index] as!
ShowOrderViewController
      var destinationDS = destinationVC.router!.dataStore!
      passDataToShowOrder(source: dataStore!, destination: &desti-
nationDS)
      navigateToShowOrder(source: viewController!, destination:
destinationVC)
    }
  }

  // MARK: Navigation

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

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

  // MARK: Passing data

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

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

The `CreateOrderRoutingLogic` protocol declares all the possible routes originating from the `CreateOrder` source scene. There are two routes this router can perform - `routeToListOrders(segue:)` and `routeToShowOrder(segue:)`. The method name reveals which destination scene the route is going to. It is also used to match the selector from the view controller's `prepare(for:sender:)` method at runtime. The selector is based on the segue identifier you specify in the storyboard. So it's important to choose a segue identifier and route method name that is easy to understand and remember.

The simplest convention is to just use the name of the scene as the segue identifier and part of the route method name. For a route going from the `First` scene to the `Second` scene, you first create the segue by control-dragging from the `FirstViewController` to the `SecondViewController` in the storyboard, and give the segue an identifier of `Second`. Next, in the `FirstRoutingLogic` protocol, you declare the route with the method name `routeToSecond(segue:)`, and implement this method in the `FirstRouter` class.

Step 1 - Getting a hold of the destination

The first step of the 3-step routing process starts with calling the `routeToSecond(segue:)` method. The `routeToSecond(segue:)` method can be invoked in a number of ways, depending on the route type.

1. Automatic Segue Route

   If you control-drag from an `UIControl` element to create the segue in the storyboard, the segue will be automatically performed by iOS when that element is acted upon. When that happens, the `prepare(for:sender:)` method is invoked, the segue identifier is used to form a selector which is used to match for a route method name in the router to see if such a method exists. If it does, it is invoked. In essence, for automatic segue routes, you don't have to do anything. The `routeToSecond(segue:)` method will be invoked for you. You just need to take care of things from that point forward.
2. Manual Segue Route

   If you control-drag from the view controller object icon to create the segue in the storyboard, the segue will need to be manually performed by you. You do this by calling the `UIViewController`'s `performSegue(withIdentifier:sender:)` method. Then, iOS will be in charge from there by calling the `prepare(for:sender:)` method. The rest is the same as for automatic segue routes. For you, you only need to decide when the route needs to happen and perform the segue manually.
3. Programmatic Route

   There are no segue involved. iOS won't perform any segue. The `prepare(for:sender:)` method won't be called. So, you have to programmatically invoke the `routeToSecond(segue:)` method. Since there is no segue, you just pass nil for the `segue` parameter. The rest is the same as for segue routes.

The `if-else` statement in the `routeToListOrders(segue:)` and `routeToShowOrder(segue:)` methods above is designed such that they are flexible enough to be used for all three route types.

If this is a segue route (automatic or manual), you can easily get the destination view controller by asking `segue.destination`. The data store is conveniently retrieved from `destinationVC.router!.dataStore!`.

For a programmatic route, the destination view controller can be retrieved based on how your view controller hierarchy is structured. For a navigation interface, such as the CleanStore sample app, you can route backward to the previous scene using `viewController?.navigationController?.viewControllers[index]` where `index` is the view controller just below the top view controller in the stack. For a tab bar interface, you can get the desired tab from `tabBarController?.viewControllers[index]`. Any custom view controller you create should also provide a way to traverse the view controller hierarchy.

Step 2 - Passing data to the destination

After all is said and done, you should now have `destinationVC` and `destinationDS` ready for the second phase. Often, we want to pass some data from the current scene to the next scene. For example, the user may select a row in the table view to see more details of the selected data item. You want to pass the selected item to the next scene. You can do this by calling the `passDataToListOrders(source:destination:)` method, passing in the source and destination data stores.

When the `passDataToListOrders(source:destination:)` method is invoked, you pass any data you need from the `FirstDataStore` to the `SecondDataStore`. Most of the time, this is done by simple assignment statements. That means the data needs to be available in the data stores. So, you add the data you want to pass from

the source to the destination scene in the respective data stores. It can be as simple as `destination.data = source.data`. Even if it's more complex, the code is still isolated in the `passDataToListOrders(source:destination:)` method.

If there is no data to pass, you can leave this method blank, or not have this method at all.

Step 3 - Navigating to the destination

Finally, the third phase is only necessary for programmatic routes. In the `navigateToListOrders(source:destination:)` method, you write the code to present the destination view controller on screen. Before Apple introduced the concept of storyboard and segue, you display another view controller by programmatically present it. Over several iterations of iOS releases, these methods have changed slightly. Nowadays, you can pretty much present anything with the following method:

- On `UIViewController`:
  - `show(_:sender:)`
  - `showDetailViewController(_:sender:)`
  - `present(_:animated:completion:)`
- On `UINavigationController`:
  - `pushViewController(_:animated:)`
- On `UITabBarController`:
  - `selectedIndex`

Depending on your view controller hierarchy, you pick the most appropriate way. If you have a custom view controller, you should provide your own presentation mechanism, or inherit from one of the above methods as a way of presenting the destination view controller.

13. 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.