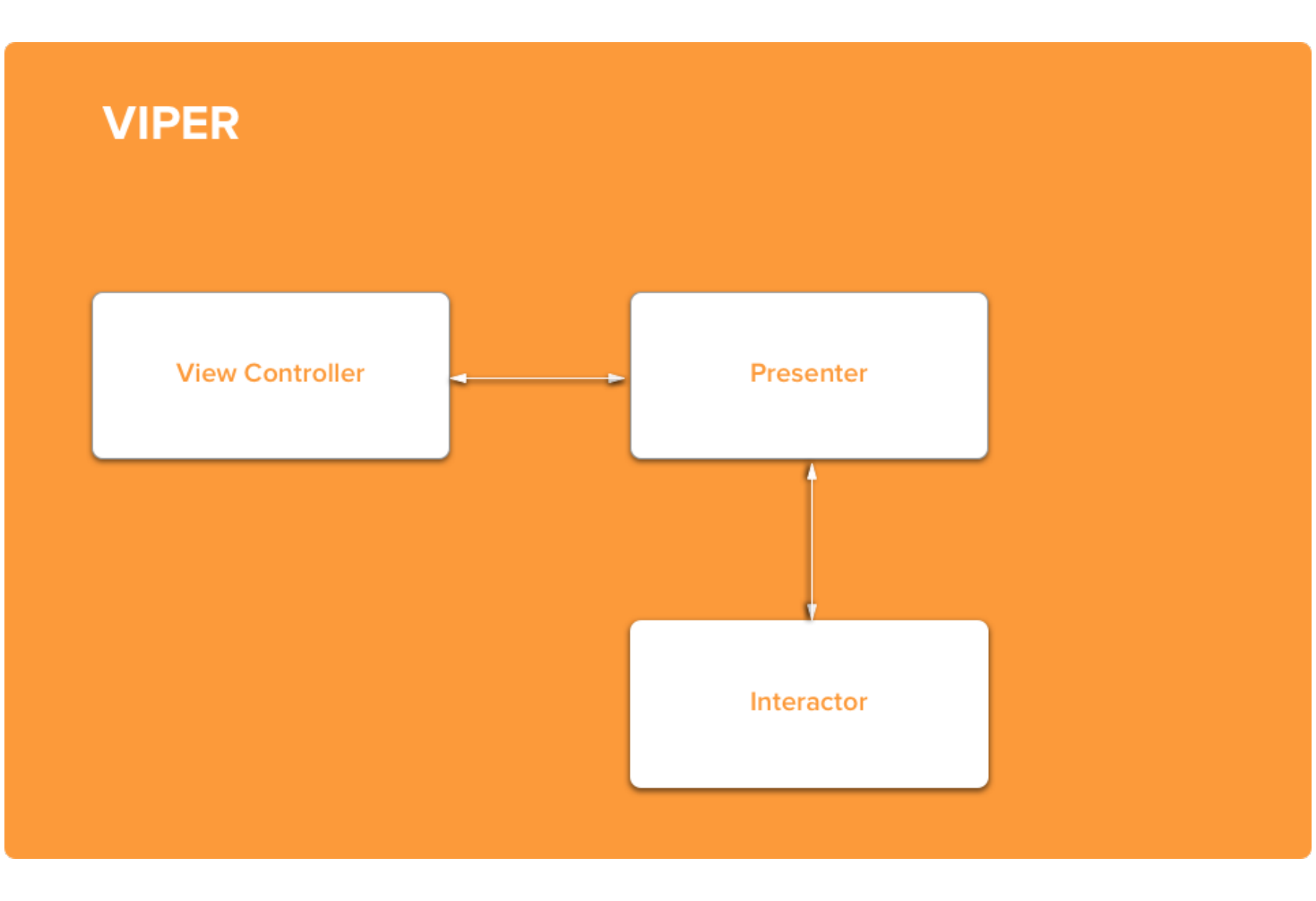
Clean swift архитектура как альтернатива VIPER

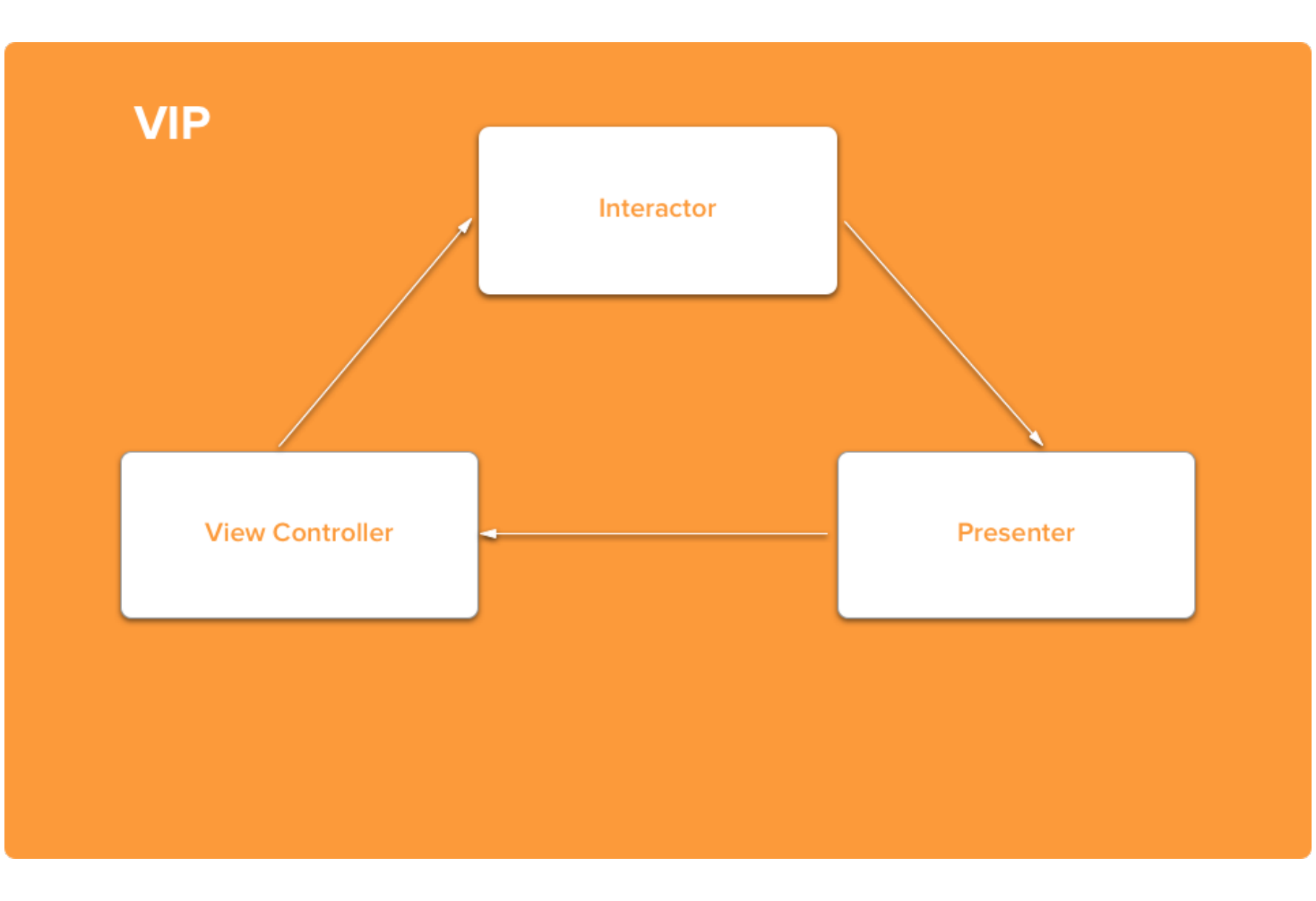
Введение

На данный момент существует множество статей про VIPER — clean архитектуру, различные вариации которой в свое время стали популярны для iOS проектов. Если вы не знакомы с Viper, можете прочитать тут, тут или тут.

Я бы хотел поговорить об альтернативе VIPER — Clean Swift. Clean Swift на первый взгляд похож на VIPER, однако отличия становятся видны после изучения принципа взаимодействия модулей. В VIPER основу взаимодействия составляет Presenter, он передает запросы пользователя Interactor'у для обработки и форматирует полученные от него назад данные для отображения на View Controller:



В Clean Swift основными модулями так же, как и в VIPER, являются View Controller, Interactor, Presenter.



Взаимодействие между ними происходит циклично. Передача данных основана на протоколах (опять же, аналогично VIPER), что позволяет при будущем изменении какого-то из компонентов системы просто подменить его на другой. Процесс взаимодействия в общем виде выглядит так: пользователь нажимает на кнопку, View Controller создает объект с описанием и отправляет его в Interactor. Interactor в свою очередь осуществляет какой-то конкретный сценарий в соответствии с бизнес логикой, создает объект результата и передает его Presenter. Presenter формирует объект с отформатированными для отображения пользователю данными и отправляет его во View Controller. Давайте подробнее рассмотрим каждый модуль Clean Swift подробнее.

View (View Controller)

View Controller, как и в VIPER, выполняет все конфигурации VIew, будь то цвет, настройки шрифта UILabel или Layout. Поэтому каждый UIViewController в данной архитектуре реализует Input протокол для отображения данных или реакции на действия пользователя.

Interactor

Interactor содержит в себе всю бизнес логику. Он принимает действия пользователя от контроллера, с параметрами (например измененный текст поля ввода, нажатие той или иной кнопки) определёнными в Input протоколе. После отработки логики, Interactor при необходимости должен передать данные для их отображения на экране в Presenter. Однако Interactor принимает на вход запросы только от View, в отличие от VIPER, где эти запросы проходят Presenter.

Worker

Чтобы излишне не усложнять Interactor и не дублировать детали бизнес логики, можно использовать дополнительный элемент Worker. В простых модулях он не всегда нужен, но в достаточно нагруженных позволяет снять с Interactor часть задач. Например, в worker может быть вынесена логика взаимодействия с базой данных, особенно, если одни и те же запросы к базе могут использоваться в разных модулях.

Presenter

Presenter обрабатывает данные для показа пользователю. Результат в данном случае — это Input протокол ViewController'a, здесь можно например поменять формат текста, перевести значение цвета из enum в rgb и т.п.

Router

Router ответственен за передачу данных другим модулям и переходы между ними. У него есть ссылка на контроллер, потому что в iOS, к сожалению, контроллеры помимо всего прочего исторически ответственны за переходы. При использовании segue можно упростить инициализацию переходов благодаря вызову методов Router из Prepare for segue, потому что Router знает, как передать данные, и сделает это без лишнего кода цикла со стороны Interactor/Presenter. Данные передаются используя протоколы хранилищ данных каждого модуля, реализуемых в Interactor. Эти протоколы также ограничивает возможность доступа к внутренним данным модуля из Router.

Models

Models — это описание структур данных для передачи данных между модулями. Каждая реализация функции бизнес логики имеет свое описание моделей.

- Request — для передачи запроса из контроллера в интерактор.
- Response — ответ интерактора для передачи презентеру с данными.
- ViewModel — для передачи данных в готовом для отображения в контроллере виде.

Пример реализации

Рассмотрим подробнее данную архитектуру на простом примере. Им послужит приложение ContactsBook в упрощенном, но вполне достаточном для понимания сути архитектуры виде. Приложение включает себя список контактов, а также добавление и редактирование контактов.

Пример input протокола:

```
protocol ContactListDisplayLogic: class {
    func displayContacts(viewModel: ContactList.ShowContacts.ViewModel)
}
```

Каждый контроллер содержит ссылку на объект, реализующий протокол input Interactor

```
var interactor: ContactListBusinessLogic?
```

а также на объект Router, который должен реализовывать логику передачи данных и переключения модулей:

```
var router: (NSObjectProtocol & ContactListRoutingLogic & ContactListDataPassi
```

Можно реализовать конфигурирование модуля в отдельном приватном методе:

```
private func setup() {
        let viewController = self
        let interactor = ContactListInteractor()
        let presenter = ContactListPresenter()
        let router = ContactListRouter()
        viewController.interactor = interactor
        viewController.router = router
        interactor.presenter = presenter
        presenter.viewController = viewController
        router.viewController = viewController
        router.dataStore = interactor
    }
```

либо создать синглтон Configurator, чтобы вынести этот код из контроллера (для тех кто считает, что контроллер не должен участвовать в конфигурации) и не соблазнять себя доступом до частей модуля в контроллере. Класса конфигуратора нет в представлении дяди Боба и в классическом VIPER. Использование конфигуратора для модуля добавления контакта выглядит так:

```
override func awakeFromNib() {
        super.awakeFromNib()
        AddContactConfigurator.sharedInstance.configure(self)
    }
```

Код конфигуратора содержит в себе единственный метод конфигурации, абсолютно идентичный setup методу в контроллере:

```
final class AddContactConfigurator {
    static let sharedInstance = AddContactConfigurator()

    private init() {}

    func configure(_ control: AddContactViewController) {
        let viewController = control
        let interactor = AddContactInteractor()
        let presenter = AddContactPresenter()
        let router = AddContactRouter()
        viewController.interactor = interactor
        viewController.router = router
        interactor.presenter = presenter
        presenter.viewController = viewController
        router.viewController = viewController
        router.dataStore = interactor
    }
}
```

Еще одним очень важным моментом в реализации контроллера является код в стандартном методе prepare for segue:

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

Внимательный читатель скорее всего заметил, что Router также обязан реализовывать NSObjectProtocol. Делается это для того, чтобы мы могли воспользоваться стандартными методами этого протокола для роутинга при использовании segues. Для поддержки такой простой переадресации именование segue identifier должно соответствовать окончаниям названий методов Router. Например, для перехода к просмотру контакта есть segue, который завязан на выбор ячейки с контактом. Его identifier — "ViewContact", вот соответствующий метод в Router:

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

Запрос на отображение данных к Interactor тоже выглядит очень просто:

```
private func fetchContacts() {
        let request = ContactList.ShowContacts.Request()
        interactor?.showContacts(request: request)
    }
```

Перейдем к Interactor. Interactor реализует протокол ContactListDataStore, ответственный за хранение/доступ к данным. В нашем случае это просто массив контактов, ограниченный только getter-методом, чтобы показать роутеру недопустимость его изменения из других модулей. Протокол, реализующий бизнес логику для нашего списка, выглядит следующим образом:

```
func showContacts(request: ContactList.ShowContacts.Request) {
        let contacts = worker.getContacts()
        self.contacts = contacts
        let response = ContactList.ShowContacts.Response(contacts: contacts)
        presenter?.presentContacts(response: response)
    }
```

Он получает из ContactListWorker данные контактов. За то, каким образом загружаются данные, в этом случае несет ответственность Worker. Он может обратиться к сторонним сервисам, которые решают к примеру, взять данные из кеша или загрузить из сети. После получения данных Interactor отдает ответ (Response) в Presenter для подготовки к отображению, для этого Interactor содержит ссылку на Presenter:

```
var presenter: ContactListPresentationLogic?
```

Presenter реализует только один протокол — ContactListPresentationLogic, в нашем случае он просто принудительно меняет регистр имени и фамилии контакта, формирует из модели данных модель представления DisplayedContact и передает это в Controller на отображение:

```
func presentContacts(response: ContactList.ShowContacts.Response) {
        let mapped = response.contacts.map {
            ContactList
                .ShowContacts
                .ViewModel
                .DisplayedContact(firstName: $0.firstName.uppercaseFirst,
                                  lastName: $0.lastName.uppercaseFirst)
        }

        let viewModel = ContactList.ShowContacts.ViewModel(displayedContacts:
        viewController?.displayContacts(viewModel: viewModel)
    }
```

После чего цикл завершается и контроллер отображает данные, реализуя метод протокола ContactListDisplayLogic:

```
func displayContacts(viewModel: ContactList.ShowContacts.ViewModel) {
        displayedContacts = viewModel.displayedContacts
        tableView.reloadData()
    }
```

Вот так выглядят модели для отображения контактов:

```
enum ShowContacts {
        struct Request {
        }

        struct Response {
            var contacts: [Contact]
        }

        struct ViewModel {
            struct DisplayedContact {
                let firstName: String
                let lastName: String

                var fullName: String {
                    return firstName + " " + lastName
                }
            }

            var displayedContacts: [DisplayedContact]
        }
    }
```

В данном случае запрос не содержит данных, так как это просто общий список контактов, однако, если, например экран списка содержал бы фильтр, в данный запрос можно было бы включить тип фильтра. Модель ответа Intrecator содержит в себе нужный список контактов, ViewModel же содержит массив готовых для отображения данных — DisplayedContact.

Почему Clean Swift

Рассмотрим плюсы и минусы этой архитектуры. Во-первых, Clean Swift имеет шаблоны кода, которые упрощают создание модуля. Эти шаблоны можно написать для множества архитектур, но когда они имеются из коробки — это как минимум экономит несколько часов вашего времени.

Во-вторых, данная архитектура, так же как VIPER, хорошо тестируется, примеры тестов доступны в проекте. Так как модуль, с которым происходит взаимодействие, легко заменить заглушкой, определение функционала каждого модуля при помощи протоколов позволяет реализовать это без головной боли. Если мы одновременно создаем бизнес логику и соответствующие к ней тесты (Interactor, Interactor tests), это хорошо вписывается в принцип TDD. Благодаря тому, что выход и вход каждого кейса логики определен протоколом, достаточно просто сначала написать тест, определяющий его поведение, а затем реализовать непосредственно логику метода.

В-третьих, в Clean Swift (в отличие от VIPER) реализуется однонаправленный поток обработки данных и принятия решений. Всегда выполняется только один цикл — View — Interactor — Presenter — View, что в том числе упрощает рефакторинг, так как изменять чаще всего приходится меньше сущностей. Благодаря этому, проекты с логикой, которая часто меняется или дополняется удобнее рефакторить при использовании методологии Clean Swift. Используя Clean Swift вы разделяете сущности двумя путями:

1. Изолируете компоненты, декларируя протоколы Input и Output
2. Изолируете фичи, используя структуры и инкапсулируя данные в отдельные запросы/ответы/модели UI. Каждая фича имеет свою логику и контролируется в рамках одного процесса, не пересекаясь в одном модуле с другими фичами.

Clean Swift не стоит использовать в небольших проектах без долгосрочной перспективы, в проектах — прототипах. Например, приложение для расписания конференции разработчиков реализовывать с помощью данной архитектуры слишком накладно. Долгосрочные проекты, проекты с большим количеством бизнес логики, напротив, хорошо ложатся в рамки этой архитектуры. Очень удобно использовать Clean Swift, когда проект реализуется для двух платформ — Mac OS и iOS, или планируется портировать его в дальнейшем.