Обзор паттернов Viper, MVP

Viper

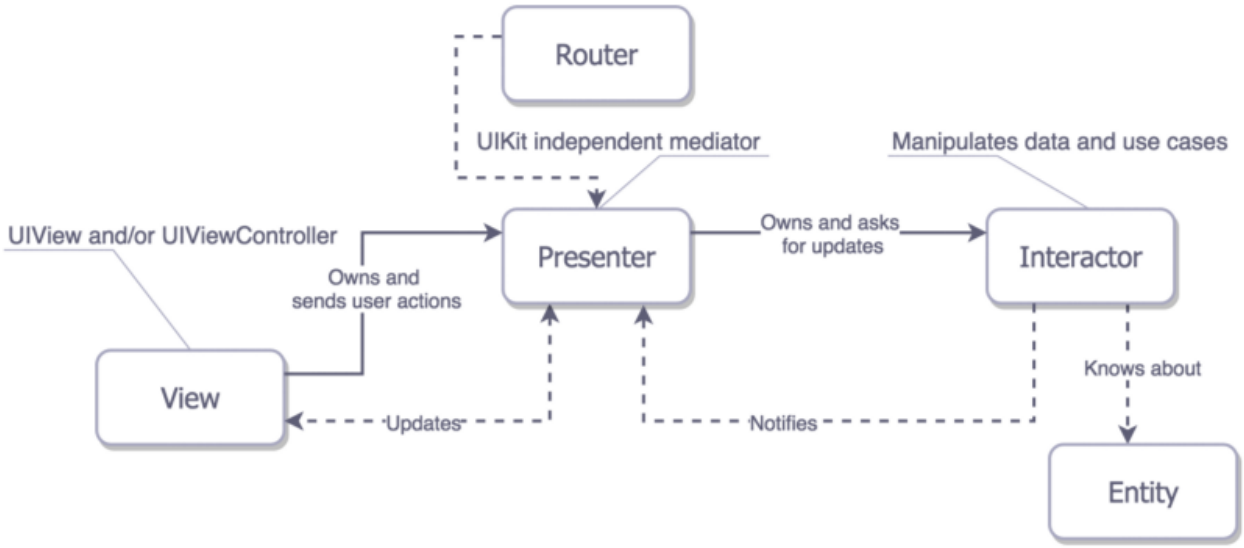
Когда MVC превратился в Massive View Controller, разработчикам понадобилась другая архитектура, более соответствующая принципам SOLID и особенно Single Responsibility Principle. Так появился Viper (View–Interactor–Presenter–Entity–Router)

Особенности Viper:

- соблюдение чистой архитектуры в силу изолированности каждого модуля от других
- легкость изменений и исправления багов
- легкость тестирования
- следует использовать только, когда требования к приложения хорошо сформированы
- не подходит для применения в быстро изменяющемся окружении - в маленьких приложениях, где менее структурированные MVP или MVC могут быть более уместными

1 экран (1 вьюконтроллер) = 1 модуль Viper.

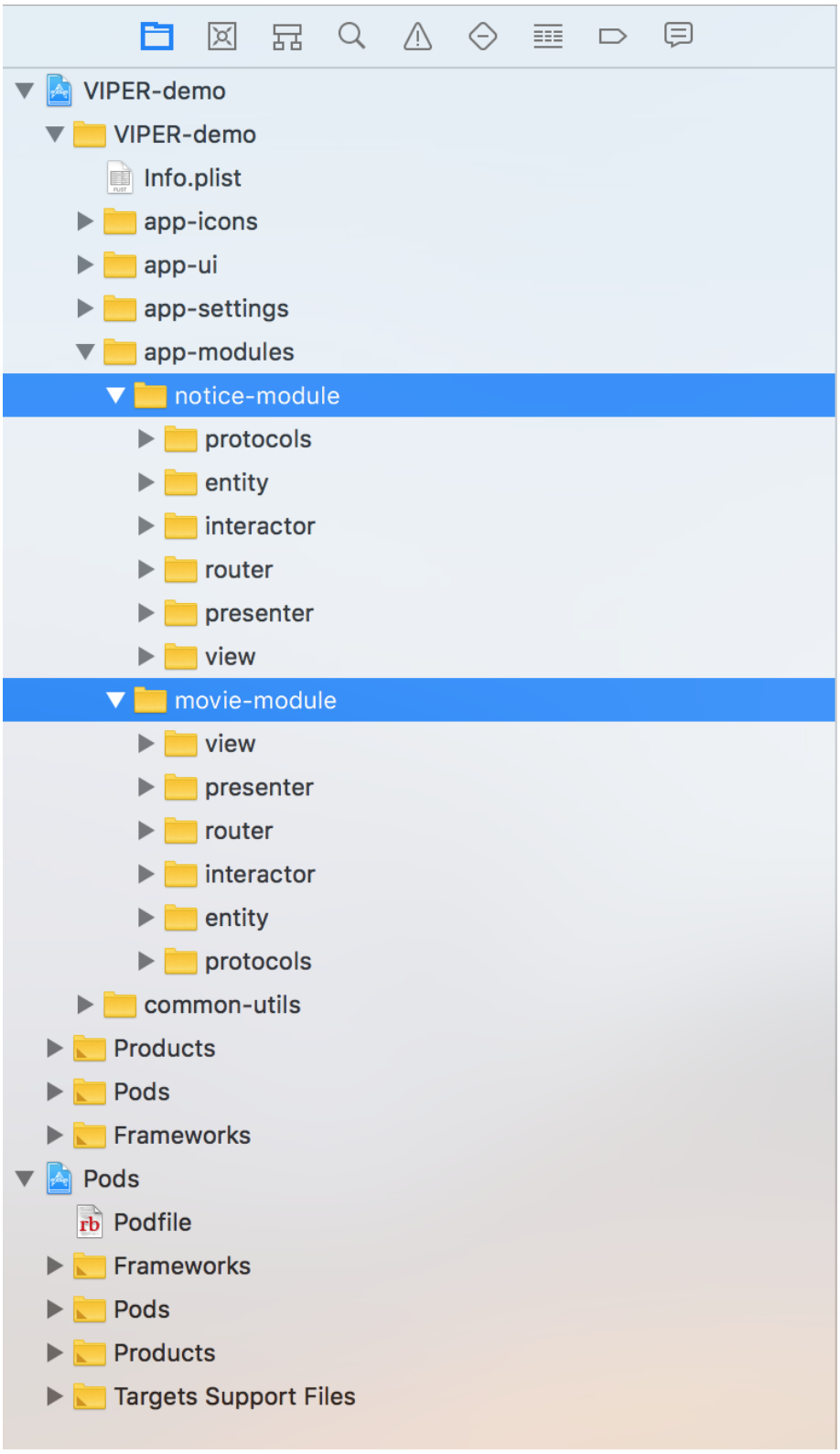
Один модуль разбит на слои, где каждый выполняет свою роль.



или

На сложных экранах модуль можно делить на подмодули, где у каждого будут свои презентеры и интеракаторы.

Типичная структура проекта на Viper


VIPER-demo
VIPER-demo
Info.plist
app-icons
app-ui
app-settings
app-modules
notice-module
protocols
entity
interactor
router
presenter
view
movie-module
view
presenter
router
interactor
entity
protocols
common-utils
Products
Pods
Frameworks
Pods
Podfile
Frameworks
Pods
Products
Targets Support Files

View

Класс, который содержит код для вывода интерфейса пользователю и получить от него ответы. После получения ответов, View оповещает Presenter. Никогда не запрашивает данные у Presenter'а.

Interactor

Содержит бизнес-логику для управления объектами (Entity), чтобы выполнить определенную задачу. Задача выполняется в Interactor'е, независимо от любого UI, т.е. Interactor – *UIKit independent*.

Presenter

Presenter — ядро модуля. Получает оповещения о действиях пользователя из View и действует соответственно. Единственный класс, который общается с другими компонентами. Обращается к Router для wire-framing, к Interactor за данными (сетевые запросы, локальные запросы данных), к View чтобы обновить UI.

Presenter (как и Interactor) – *UIKit independent.*

Entity

Entity — это объекты, которыми управляет Interactor. Entity только управляет Interactor. Он никогда не передает сущности уровню представления

Entity никогда не передаются из Interactor'а к Presenter'у. Вместо этого простые структуры данных, у которых нет поведения, передаются из Interactor'а к Presenter'у. Это препятствует любой ‘реальной работе’ в Presenter'е. Presenter может только подготовить данные для отображения на View.

Router

Entity — это объекты, которыми управляет Interactor. Entity только управляет Interactor. Он никогда не передает сущности уровню представления

Entity никогда не передаются из Interactor'а к Presenter'у. Вместо этого простые структуры данных, у которых нет поведения, передаются из Interactor'а к Presenter'у. Это препятствует любой ‘реальной работе’ в Presenter'е. Presenter может только подготовить данные для отображения на View.

В общем Viper это архитектура, построенная на delegates. Почти все общение между слоями реализовано через делегаты. Один слой вызывает другой через протокол. Обращающийся слой вызывает функцию из протокола. Слой приемник обращения соответствует этого протоколу и реализует функцию.

Инструменты Viper

Стоит использовать автоматический генератор структуры проекта.

- Generamba
- VIPER Code
- VIPER Gen

Полезные ссылки

https://betterprogramming.pub/how-to-implement-viper-architecture-in-your-ios-app-rest-api-and-kingfisher-f494a0891c43

https://medium.com/@smalam119/viper-design-pattern-for-ios-application-development-7a9703902af6

https://medium.com/@smalam119/viper-design-pattern-for-ios-application-development-7a9703902af6

https://www.youtube.com/watch?v=fFpYuVmctZs

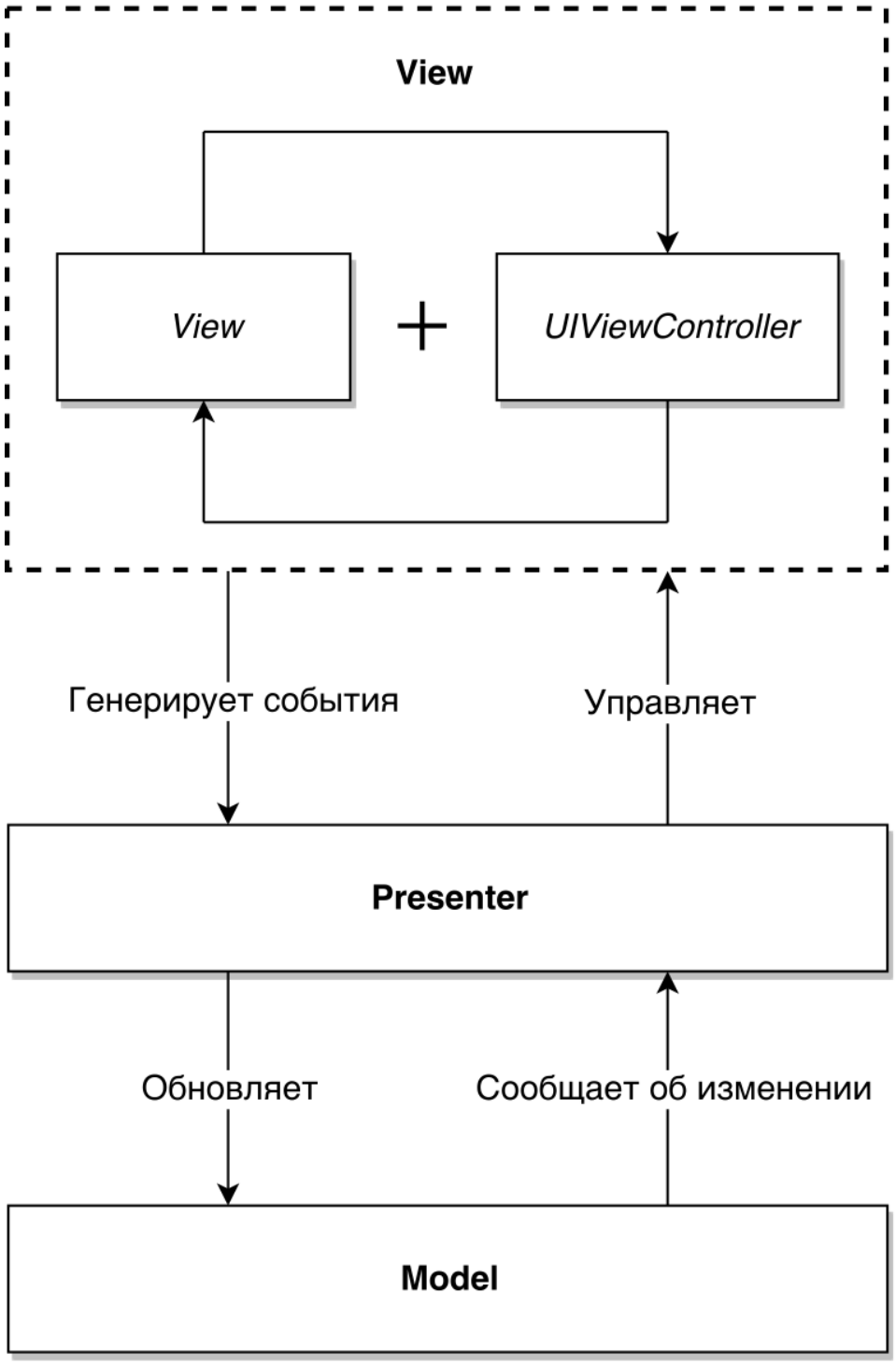
https://coderoad.ru/44796176/Реализация-архитектуры-VIPER-в-iOS

https://www.youtube.com/watch?v=CkylrfKvf1A&list=PLyjgjmI1UzlSWtjAMPOt03L7InkCRlGzb&index=6

MVP

Компромиссный вариант между громоздким по количеству кода VIPER и не очень простым для понимания MVVM. Часто рекомендуется маленьких и средних проектов наравне с MVC.

MVP

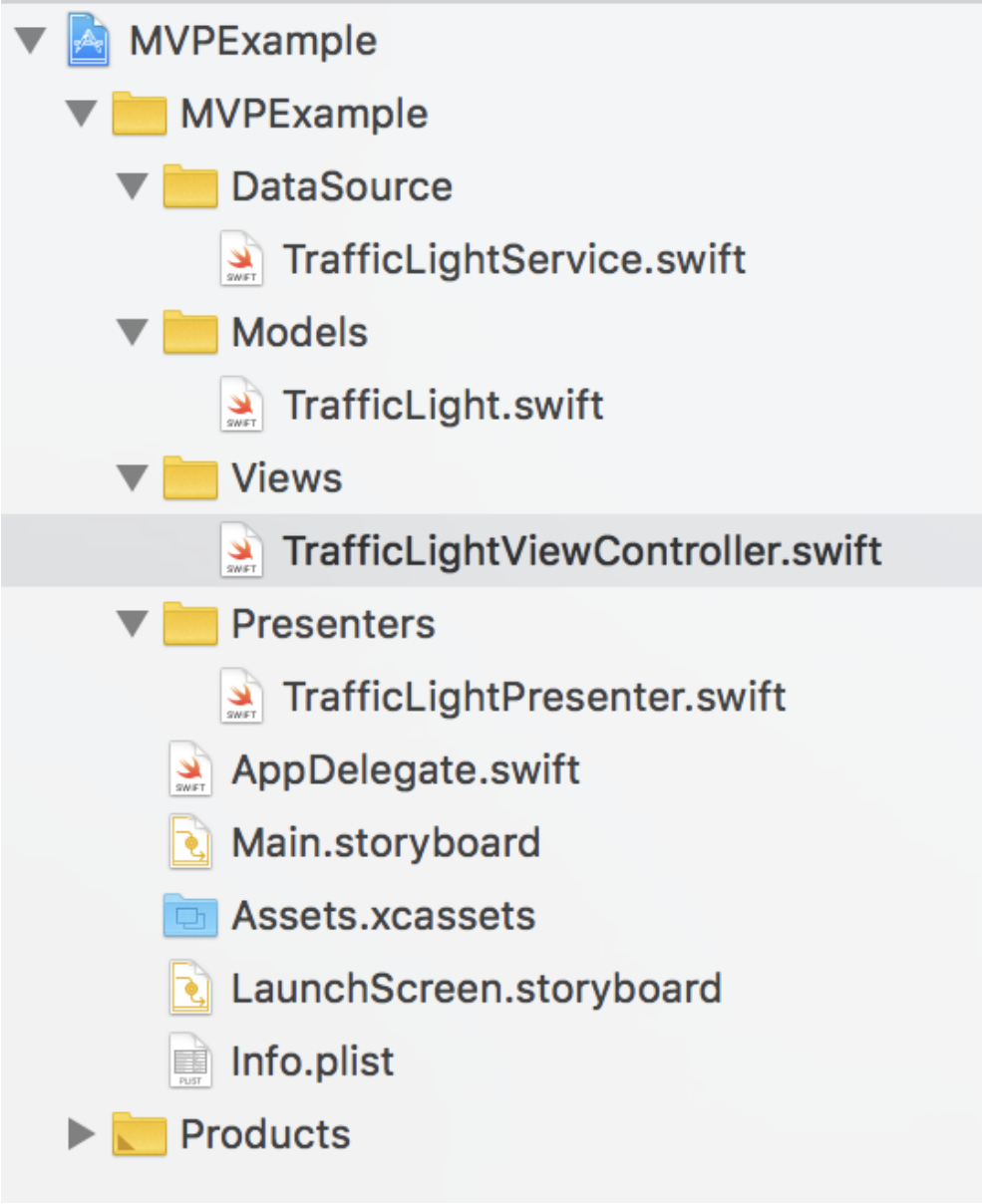


View – отображает данные на экране и оповещает **Presenter** о действиях пользователя. View никогда не запрашивает данные, только получает их от **Presenter**.

Presenter – получает от **View** информацию о действиях пользователя и реагирует на них. Передает события в **Model** для обновления или обработки внутри себя. Ничего не должен знать о UIKit, за исключением UIImage.

Model – заключает в себе всю бизнес-логику, необходимую для работы модуля: работа с загрузкой данных по API, извлечение данных из БД и их кэширование.

Пример вида проекта



Особенности правильного MVP

- Бизнес-логика отделена от ViewController и закрыта протоколами для лучшей покрываемости тестами
- Модули обособлены друг от друга и доступны для переиспользования в других приложениях
- Каждый компонент модуля отвечает за свои задачи, что позволяет сделать код компактнее

Отличия MVP от MVC

- View в MVC тесно связан с Controller, в MVP слой View состоит из UIView и UIViewController и ViewController вообще отделен от бизнес логики
- MVP View глуп насколько это возможно (как в MVVM), MVC View включает в себя некоторую бизнес логику и может подавать запросы в Model
- MVP View работает с действиями пользователя и делегирует взаимодействие Presenter'y, в MVC Controller обрабатывает действия пользователя и управляет Model
- MVP активно поддерживает Unit Testing, MVC обладает ограниченной поддержкой тестирования
- MVC Controller содержит много UIKit зависимостей, **MVP Presenter UIKit-независим!!!**

Универсальность

Плюсом использования паттерна MVP является его универсальность для iOS и Android. Если у вас большое количество проектов разрабатывается под обе платформы, то логика Presenter-а будет универсальной, и если разработка приложений происходит по очереди, то адаптация на другой платформе будет проходить быстрее и предсказуемее, так как код логики в Presenter-е практически одинаковый. Более того, на этапе начала работы по адаптации для другой платформы уже будут готовые тест-кейсы.

Пример

Dog.swift

```
import Foundation

enum Breed: String {
    case bulldog  = "Bulldog"
    case doberman = "Doberman"
    case labrador = "Labrador"
}
```

```
struct Dog {
    let name:  String
    let breed: String
    let age:   Int
}
```

DoggyListViewController.swift

```
import UIKit

class DoggyListViewController: UIViewController, UITableViewDataSource {

    @IBOutlet weak var emptyView: UIView?
    @IBOutlet weak var tableView: UITableView?
    @IBOutlet weak var spinner:   UIActivityIndicatorView?

    fileprivate let dogPresenter = DoggyPresenter(dogService: DoggyService())
    fileprivate var dogsToDisplay = [DoggyViewData]()

    override func viewDidLoad() {
        super.viewDidLoad()

        tableView?.dataSource = self
        spinner?.hidesWhenStopped = true
        dogPresenter.attachView(true, view: self)
        dogPresenter.getDogs()
    }

    // MARK: DataSource
    func tableView(_ tableView: UITableView, numberOfRowsInSection section: Int) -> Int {
        return dogsToDisplay.count
    }

    func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell {
        let cell = UITableViewCell(style: .subtitle, reuseIdentifier: "Cell")
        let userViewData = dogsToDisplay[indexPath.row]
        cell.textLabel?.text = userViewData.name
        cell.detailTextLabel?.text = userViewData.age
        return cell
    }
}

extension DoggyListViewController: DoggyView {

    func startLoading() {
        spinner?.startAnimating()
    }

    func finishLoading() {
        spinner?.stopAnimating()
    }

    func setDoggies(_ doggies: [DoggyViewData]) {
        dogsToDisplay = doggies
```

```
        tableView?.isHidden = false
        emptyView?.isHidden = true;
        tableView?.reloadData()
    }

    func setEmpty() {
        tableView?.isHidden = true
        emptyView?.isHidden = false;
    }
}
```

DoggyPresenter.swift

```
import Foundation

class DoggyPresenter {

    // MARK: - Private
    fileprivate let dogService: DoggyService
    weak fileprivate var dogView: DoggyView?

    init(dogService: DoggyService){
        self.dogService = dogService
    }

    func attachView(_ attach: Bool, view: DoggyView?) {
        if attach {
            dogView = nil
        } else {
            if let view = view { dogView = view }
        }
    }

    func getDogs(){
        self.dogView?.startLoading()

        dogService.deliverDoggies { [weak self] doggies in
            self?.dogView?.finishLoading()

        if doggies.count == 0 {
            self?.dogView?.setEmpty()
        } else {
            self?.dogView?.setDoggies(doggies.map {
                return DoggyViewData(name: "\($0.name) \($0.breed)",
                                     age: "\($0.age)")
                })
            }
        }
    }
}

struct DoggyViewData {
    let name: String
    let age:  String
}
```

DoggyService.swift

```
import Foundation

typealias Result = ([Dog]) -> Void

class DoggyService {

    func deliverDoggies(_ result: @escaping Result) {

        let firstDoggy  = Dog(name: "Alfred", breed: Breed.labrador.rawValue, age: 1)
        let secondDoggy = Dog(name: "Vinny",  breed: Breed.doberman.rawValue, age: 5)
        let thirdDoggy  = Dog(name: "Lucky",  breed: Breed.labrador.rawValue, age: 3)

        let delay = DispatchTime.now() + Double(Int64(Double(NSEC_PER_SEC)*2)) / Double(NSEC_PER_SEC)

        DispatchQueue.main.asyncAfter(deadline: delay) {
            result([firstDoggy,
                    secondDoggy,
                    thirdDoggy])
        }
    }
}
```

DoggyView.swift

```
import Foundation

protocol DoggyView: NSObjectProtocol {
    func startLoading()
    func finishLoading()
    func setDoggies(_ doggies: [DoggyViewData])
    func setEmpty()
}
```

Полезные ссылки

https://riptutorial.com/ios/topic/9467/mvp-architecture

https://surfstudio.github.io/Surf-iOS-Developers/architectures/Surf_MVP_Coordinators.html

https://medium.com/omisoft/lightweight-mvp-architecture-in-ios-a16b3da01563

https://habr.com/ru/post/343438/

https://saad-eloulladi.medium.com/ios-swift-mvp-architecture-pattern-a2b0c2d310a3