GeekBrains

Архитектуры и шаблоны проектирования на Swift

Архитектурные паттерны. Часть 2. MVVM, VIPER

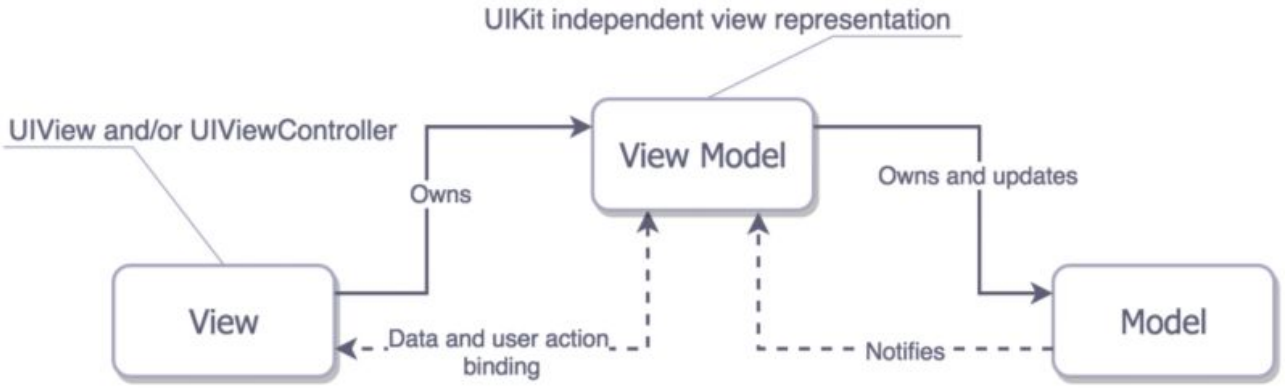
Продвинутые архитектуры MVVM и VIPER — еще большее разделение ответственностей между классами.

Оглавление

MVVM

Теория

MVVM — архитектурный паттерн, состоящий из трех слоев (аналогично MVC и MVP). Эти слои — **Model**, **View** и **ViewModel**. В этом паттерне **UIViewController** рассматривается как View, как и в паттерне MVP. Поэтому схема для нас, в принципе, знакома:



Можно даже не заметить разницы с MVP (кроме того, что слой посредника называется не **Presenter**, а **ViewModel**), но есть одна существенная деталь, которая отделяет два этих паттерна. Между **View** и **ViewModel** есть взаимодействие в обе стороны через **binding** — связывание.

Что такое **binding**? По сути, это использование паттерна проектирования **Observer** для того, чтобы **View** всегда знала об изменениях **ViewModel**. На практике это выглядит так: класс **ViewModel** содержит логику по обработке пользовательского ввода, обращается к модели, запрашивает данные — в общем, аналогично Presenter'у в MVP. Но также **ViewModel** содержит свойства, к которым имеет доступ View и на изменение которых View подписана. Например, когда приходят данные с сервера, **ViewModel** записывает их у себя, и View сразу же узнает об изменениях и о том, что ей надо отрисовать новые данные.

Binding можно делать разными способами. В **Cocoa** (нативном фреймворке для разработки под **macOS**) есть механизм для биндингов, но в iOS SDK такого стандартного механизма нет. **KVO** и **Notifications**, с помощью которых в принципе можно осуществить **binding**, неудобны. Поэтому для реализации связывания **View** и **ViewModel** в паттерне **MVVM** обычно используется одна из сторонних библиотек для реактивного программирования, например **RxSwift** или **Bond**.

Практика

Задача

Продолжим использовать проект с предыдущего урока — приложение, выполняющее поиск по приложениям в iTunes.

На уроке 2 при изучении паттерна **Observer** мы воспользовались собственной оберткой **Observable**, написанной на Swift и реализующей **observing**. В практическом примере по **MVVM** мы воспользуемся этой же оберткой для реализации **binding** в **MVVM**.

Паттерн **MVVM** особенно хорошо себя показывает в тех случаях, когда данные часто меняются прямо во время работы приложения, и это нужно отображать на **UI**. Поэтому рассмотрим такую задачу: в каждой ячейке, отображающей найденные приложения на главном экране, должна быть кнопка

«Загрузить», по нажатию на которую надо инициировать загрузку приложения и отображать прогресс загрузки на UI. Конечно, никакую загрузку мы на самом деле производить не будем (у нас нет доступа к такому API ни на уровне iTunes API, ни на уровне iOS). Вместо этого мы создадим класс-сервис, который будет имитировать скачивание, прибавляя по 5 % прогресса каждые полсекунды. Его реализация не представляет интереса для нашего урока, поэтому мы просто добавим его код в проект.

Подготовка

1. Добавим обертку **Observable** в проект, в папку **Core** (код можно найти в методичке № 2 этого курса).

2. Добавим код класса-сервиса, который имитирует скачивание приложения, в отдельный файл в папке **Services**:

```
protocol DownloadAppsServiceInterface: class {
    var downloadingApps: [DownloadingApp] { get }
    var onProgressUpdate: (() -> Void)? { get set }
    func startDownloadApp(_ app: ITunesApp)
}

final class DownloadingApp {
    enum DownloadState {
        case notStarted
        case inProgress(progress: Double)
        case downloaded
    }

    let app: ITunesApp

    var downloadState: DownloadState = .notStarted

    init(app: ITunesApp) {
        self.app = app
    }
}

final class FakeDownloadAppsService: DownloadAppsServiceInterface {

    // MARK: - DownloadAppsServiceInterface

    private(set) var downloadingApps: [DownloadingApp] = []

    var onProgressUpdate: (() -> Void)?

    func startDownloadApp(_ app: ITunesApp) {
        let downloadingApp = DownloadingApp(app: app)
        if !self.downloadingApps.contains(where: { $0.app.appName == app.appName
}) {
            self.downloadingApps.append(downloadingApp)
            self.startTimer(for: downloadingApp)
        }
    }
```

```
    // MARK: - Private properties

    private var timers: [Timer] = []

    // MARK: - Private

    private func startTimer(for downloadingApp: DownloadingApp) {
        let timer = Timer.scheduledTimer(withTimeInterval: 0.5, repeats: true) {
[weak self] timer in
            switch downloadingApp.downloadState {
            case .notStarted:
                downloadingApp.downloadState = .inProgress(progress: 0.05)
            case .inProgress(let progress):
                let newProgress = progress + 0.05
                if newProgress >= 1 {
                    downloadingApp.downloadState = .downloaded
                    self?.invalidateTimer(timer)
                } else {
                    downloadingApp.downloadState = .inProgress(progress:
progress + 0.05)
                }
            case .downloaded:
                self?.invalidateTimer(timer)
            }
            self?.onProgressUpdate?()
        }
        RunLoop.main.add(timer, forMode: .common)
        self.timers.append(timer)
    }

    private func invalidateTimer(_ timer: Timer) {
        timer.invalidate()
        self.timers.removeAll { $0 === timer }
    }
}
```

Нам отсюда важно знать только то, что:

- стартовать фейковую скачку приложения мы будем, обращаясь к функции **startDownloadApp(_ app: ITunesApp)**;
- получать оповещения о том, что прогресс скачивания приложений изменился, будем через замыкание **onProgressUpdate: (() -> Void)?**;
- и получать информацию обо всех якобы скачиваемых в данный момент приложениях — через свойство **downloadingApps: [DownloadingApp]**.

Реализация

Удалим файлы **SearchPresenter.swift** и **SearchViewInterface.swift**, поскольку они больше не нужны в архитектуре **MVVM**. Создадим класс **SearchViewModel** по архитектуре **MVVM**. В основном, он будет

содержать то, что было в **SearchPresenter**, а также в нем будут Observable-свойства, на которые в дальнейшем подпишется вью-контроллер:

```
final class SearchViewModel {

    // MARK: - Observable properties

    let cellModels = Observable<[SearchAppCellModel]>([])
    let isLoading = Observable<Bool>(false)
    let showEmptyResults = Observable<Bool>(false)
    let error = Observable<Error?>(nil)

    // MARK: - Properties

    weak var viewController: UIViewController?

    private var apps: [ITunesApp] = []

    private let searchService: SearchServiceInterface
    private let downloadAppsService: DownloadAppsServiceInterface

    // MARK: - Init

    init(searchService: SearchServiceInterface, downloadAppsService:
DownloadAppsServiceInterface) {
        self.searchService = searchService
        self.downloadAppsService = downloadAppsService
        downloadAppsService.onProgressUpdate = { [weak self] in
            guard let self = self else { return }
            self.cellModels.value = self.viewModels()
        }
    }

    // MARK: - ViewModel methods

    func search(for query: String) {
        self.isLoading.value = true
        self.searchService.getApps(forQuery: query) { [weak self] result in
            guard let self = self else { return }
            result
                .withValue { apps in
                    self.apps = apps
                    self.cellModels.value = self.viewModels()
                    self.isLoading.value = false
                    self.showEmptyResults.value = apps.isEmpty
                    self.error.value = nil
                }
                .withError {
                    self.apps = []
                    self.cellModels.value = []
                    self.isLoading.value = false
                    self.showEmptyResults.value = true
                    self.error.value = $0
```

```
            }
        }
    }

    func didSelectApp(_ appViewModel: SearchAppCellModel) {
        guard let app = self.app(with: appViewModel) else { return }
        let appDetaillViewController = AppDetailViewController(app: app)

self.viewController?.navigationController?.pushViewController(appDetaillViewCont
roller, animated: true)
    }

    func didTapDownloadApp(_ appViewModel: SearchAppCellModel) {
        guard let app = self.app(with: appViewModel) else { return }
        self.downloadAppsService.startDownloadApp(app)
    }

    // MARK: - Private

    private func viewModels() -> [SearchAppCellModel] {
        return self.apps.compactMap { app -> SearchAppCellModel in
            let downloadingApp = self.downloadAppsService.downloadingApps.first
{ downloadingApp -> Bool in
                return downloadingApp.app.appName == app.appName
            }
            return SearchAppCellModel(appName: app.appName,
                                      company: app.company,
                                      averageRating: app.averageRating,
                                      downloadState:
downloadingApp?.downloadState ?? .notStarted)
        }
    }

    private func app(with viewModel: SearchAppCellModel) -> ITunesApp? {
        return self.apps.first { viewModel.appName == $0.appName }
    }
}
```

Класс **SeachAppCellModel** — структура и контейнер данных, по ним будут настраиваться ячейки в **tableView** у вью-контроллера:

```
struct SearchAppCellModel {
    let appName: String
    let company: String?
    let averageRating: Float?
    let downloadState: DownloadingApp.DownloadState
}
```

Посмотрите на реализацию **SearchViewModel**. К методам, отделенным **// MARK: - ViewModel methods**, может обращаться вью-контроллер для обработки пользовательского

ввода. В секции **// MARK: - Observable properties** находятся свойства вью-модели, которые будут меняться по ходу получения данных классом **ViewModel** из слоя **Model** (в нашем случае — через **SearchService** и **FakeDownloadAppsService**). На изменение значений этих свойств должен быть подписан вью-контроллер, чтобы сразу обновлять отображение на экране. Давайте это сделаем.

Перейдем в **SearchViewController.swift**. Теперь вью-контроллер должен инициализироваться со своей вью-моделью:

```
private let viewModel: SearchViewModel

init(viewModel: SearchViewModel) {
    self.viewModel = viewModel
    super.init(nibName: nil, bundle: nil)
}
```

В **viewDidLoad** добавим вызов функции, которая будет выполнять **binding** — в ней подпишемся на observable-свойства вью-модели:

```
override func viewDidLoad() {
    super.viewDidLoad()
    self.configureUI()
    self.bindViewModel()
}
```

Вот сама функция:

```
private func bindViewModel() {
    // Во время загрузки данных показываем индикатор загрузки
    self.viewModel.isLoading.addObserver(self) { [weak self] (isLoading, _) in
        self?.throbber(show: isLoading)
    }
    // Если пришла ошибка, то отобразим ее в виде алерта
    self.viewModel.error.addObserver(self) { [weak self] (error, _) in
        if let error = error {
            self?.showError(error: error)
        }
    }
    // Если вью-модель указывает, что нужно показать экран пустых результатов,
то делаем это
    self.viewModel.showEmptyResults.addObserver(self) { [weak self]
(showEmptyResults, _) in
        self?.emptyResultView.isHidden = !showEmptyResults
        self?.tableView.isHidden = showEmptyResults
    }
    // При обновлении данных, которые нужно отображать в ячейках, сохраняем их и
перезагружаем tableView
    self.viewModel.cellModels.addObserver(self) { [weak self] (searchResults, _)
in
        self?.searchResults = searchResults
    }
}
```

Также сделайте соответствующие изменения в классе **SearchViewController**, учитывая, что вместо презентера используется **ViewModel**, а протокола **SearchViewInput** больше нет — все методы, бывшие в нем, должны теперь стать приватными методами класса **SearchViewController**.

Конфигурация ячейки теперь содержит код, который отображает прогресс загрузки приложения в лейбле:

```
private func configure(cell: AppCell, with app: SearchAppCellModel) {
    cell.onDownloadButtonTap = { [weak self] in
        self?.viewModel.didTapDownloadApp(app)
    }
    cell.titleLabel.text = app.appName
    cell.subtitleLabel.text = app.company
    cell.ratingLabel.text = app.averageRating >>- { "\($0)" }
    switch app.downloadState {
    case .notStarted:
        cell.downloadProgressLabel.text = nil
    case .inProgress(let progress):
        let progressToShow = round(progress * 100.0) / 100.0
        cell.downloadProgressLabel.text = "\(progressToShow)"
    case .downloaded:
        cell.downloadProgressLabel.text = "Загружено"
    }
}
```

У **AppCell** должен быть лейбл **downloadProgressLabel** под кнопкой «Загрузить» (добавить лейбл просто — по аналогии с другими view в **AppCell**, поэтому это отдельно не рассматриваем).

Теперь **builder** модуля должен создавать **ViewModel**, инициализировать с ней вью-контроллер и установить этот вью-контроллер в свойство **ViewModel**. И MVVM-модуль будет настроен и готов к использованию. Перейдем в **SearchModuleBuilder.swift** и заменим его код на следующий:

```
final class SearchModuleBuilder {
    static func build() -> UIViewController {
        let searchService = ITunesSearchService()
        let downloadAppsService = FakeDownloadAppsService()
        let viewModel = SearchViewModel(searchService: searchService,
downloadAppsService: downloadAppsService)
        let viewController = SearchViewController(viewModel: viewModel)
        viewModel.viewController = viewController
        return viewController
    }
}
```

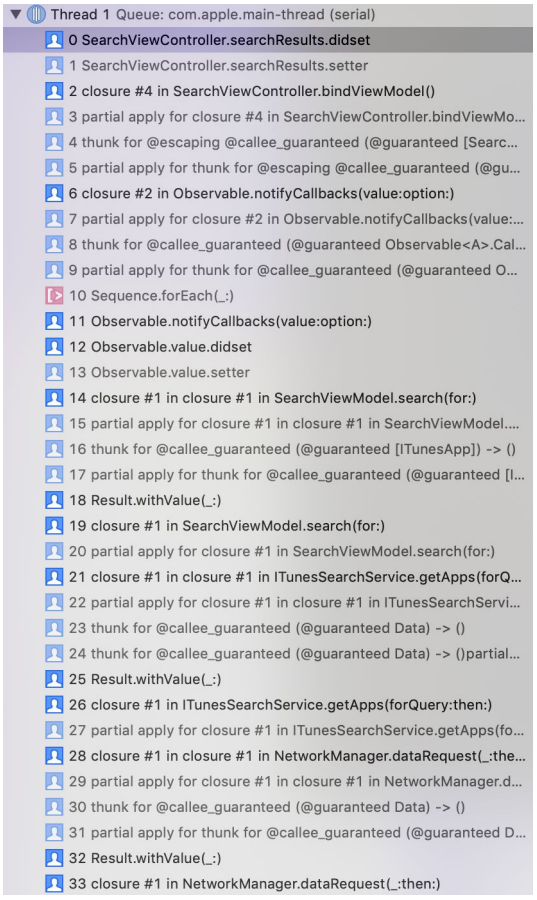
Теперь все готово. Запустите приложение и посмотрите, как при обновлении данных в **ViewModel** вью-контроллер автоматически обновляет вид.

Проблемы MVVM

MVVM дает большое преимущество в том, что касается удобного обновления отображения при изменении данных. Но, как и у любого архитектурного паттерна, у MVVM есть недостатки.

1. **MVVM** наследует проблему **MVP** — слой-посредник все еще может быть очень большим, просто раньше этим слоем был **Presenter**, а теперь **ViewModel**. О решении мы также говорили — надо лучше разделять ответственности между классами, предварительно продумывая этот вопрос.

2. В MVVM возникает новая проблема — из-за активного использования **binding** дебаг приложения затрудняется. Из-за множественных вызовов замыканий усложняется **stack trace**. Например, если поставить **breakpoint** в **didSet** свойства **searchResults** у вью-контроллера, то после получения данных запроса **stack trace** будет выглядеть примерно так:



Бывает, что изменение данных в observable-свойстве приводит к тому, что вью-контроллер запрашивает еще данные из **ViewModel**, а это приводит к изменению других observable-свойств, и так по цепочке. В таком случае в **stack trace** будет действительно сложно разобраться.

3. Поскольку для реализации **binding** в реальных проектах обычно используется **RxSwift**, нужно знать и его.

VIPER

Теория

Архитектура **MVVM**, которую мы только что рассмотрели, основывается на **MVP** и добавляет в нее свою фишку — биндинг данных. Архитектура **VIPER** тоже основывается на **MVP**, поэтому в практической части нам необходимо будет откатить проект до состояния начала этого урока, когда модуль **Search** был написан на **MVP**. **VIPER** продолжает идеи **MVP** и еще больше разделяет ответственности между классами.

Вспомним разделение ответственностей в архитектуре MVP:

View

1. Конфигурировать UI.
2. Обновлять UI.

3. Ловить user interaction.

Presenter

4. Обрабатывать user interaction (через ViewOutput).
5. Управлять отображением View (через ViewInput).
6. Реализовывать бизнес-логику.
7. Обеспечивать навигацию между экранами.

VIPER не изменяет ничего в слое **View**, да это и не нужно. Поэтому сконцентрируемся на последних четырех ответственностях. VIPER предлагает следующее разделение:

Presenter

1. Обрабатывать user interaction (через ViewOutput).
2. Управлять отображением View (через ViewInput).

Interactor

3. Реализовывать бизнес-логику.

Router

4. Обеспечивать навигацию между экранами

То есть мы делаем слой **Presenter** еще тоньше, чтобы он не разрастался чрезмерно в модулях со сложной логикой.

Какой именно кусок откалывается от **Presenter**? Начнем с последнего слоя, **Router**. Его ответственность достаточно проста — навигация в приложении. Причем для каждого модуля в **VIPER** будет создаваться свой **Router**. Например, в нашем приложении будут **SearchRouter**, **AppDetailsRouter**, **SongDetailsRouter** и т. д. **SearchRouter** будет отвечать за то, на какой экран можно перейти с экрана **Search**. Аналогично и другие роутеры. Если в приложении используется tab bar, то стоит иметь отдельный роутер для него (возможно, и целый VIPER-модуль для tab bar, с вью-контроллером и презентером). И теперь, когда презентер решает, что нужно открыть другой экран, он это делает не самостоятельно, а вызывает соответствующий метод у **Router**. Как именно показать экран — модально или в navigation stack, с какой анимацией и т. д., — решает класс **Router**.

Теперь рассмотрим, какая именно часть презентера обычно переходит в интерактор.

Вспомните пример из второй методички для паттерна **Facade** (фасад) — есть класс, который скачивает файл из сети, и класс, который дешифрует его. Еще есть класс, который сохраняет его на диск, и над всем этим стоит обработчик ошибок. Тогда мы обсуждали, что в этом случае стоит создать класс-фасад, который бы скрывал в себе сложную взаимосвязанную работу этих внутренних классов, а снаружи имел бы удобный интерфейс с несколькими методами типа **downloadFile()**. Если работа этих классов нужна только в одном экране приложения, а при проектировании мы решили не создавать отдельный класс-фасад, то в архитектуре MVP, очевидно, эта логика находилась бы в презентере. А архитектура VIPER как раз предписывает такие вещи переносить в **Interactor**.

Это лишь один пример. В целом можно сказать, что интеракторы являются прослойками для нижних компонентов системы и скрывают в себе логику по их взаимодействию. То есть задача интерактора — освободить презентер от логики, которая не завязана на отображение. При этом презентер

продолжает отдавать команды вью-контроллеру, получать от него оповещения о пользовательских действиях и реагировать на них.

Рассмотрим пример, иллюстрирующий взаимодействие слоев VIPER-модуля. Пользователь нажал на кнопку. Это нажатие ловит View и через протокол **ViewOutput** с помощью вызова соответствующего метода передает слою **Presenter** ответственность по обработке этого нажатия. **Presenter** знает, что после нажатия кнопки надо запросить данные, а пока данные запрашиваются — показывать на интерфейсе индикатор загрузки. **Presenter** через **ViewInput** вызывает метод показа индикатора загрузки. Как он будет показываться — решает **View**, **Presenter** здесь просто отдал команду. Дальше **Presenter** обращается к **Interactor** за данными. **Interactor** знает, откуда взять эти данные, причем они могут лежать в базе данных или в сети — чтобы получить их, возможно, придется сделать еще много всего, и этим занимается интерактор. Также интерактор знает, что запрос этих данных нужно залогировать, и делает это. Когда данные получены, **Interactor** возвращает данные в **Presenter**, а он через **ViewInput** приказыват слою **View** отобразить эти данные и убрать индикатор загрузки.

В общем, концептуально **VIPER** не так сложен, он просто идет еще дальше MVP и предлагает разделять ответственности между классами совсем атомарно. Такой подход будет отлично работать, если нужна максимальная переиспользуемость и тестируемость в особенно сложных экранах приложения.

И еще пара слов про VIPER. Аббревиатура означает **V**iew **I**nteractor **P**resenter **E**ntity **R**outer. Расположение слов (да и отчасти сами названия) сделано таким ради звучного сокращения. С буквами V, I, P, R мы только что разобрались, не обсудили только **Entity**. По сути, это та же **Model**, присутствующая во всех **MV(...)**-паттернах (**MVC**, **MVP**, **MVVM**). Только если **MVC** прямо говорит о том, что классы **Model** могут содержать логику, то **VIPER** придерживается концепции, что **Entity** — это просто контейнеры данных, в них нет логики, а должна она быть в интеракторах.

И еще очень важный момент. **VIPER** — это больше концепция и идея, нежели свод строгих правил. А конкретная реализация может отличаться у разных разработчиков. Можно сказать, что все, кто использует VIPER, применяют его немного по-разному. Именно поэтому, если вы будете искать дополнительные материалы в сети про архитектуру VIPER, то найдете много толкований.

- Есть вариант **VIPER**'а (который на самом деле изначально и был описан авторами архитектуры, но именно в таком виде понравился не всем), когда **Router** занимается еще и настройкой модулей, то есть задачей **Builder**. Называется такой компонент **Wireframe**.

- Можно заменить слой **Router** паттерном **Coordinator**.

- Команда **Uber** пошла еще дальше и создала свой вариант **VIPER**'а — **RIBlets** (сокращение от **Router Interactor Builder**).

В нашем варианте **VIPER** будет являться именно прямым потомком **MVP**, так как наша задача — освоить концепцию, а не применить ее к конкретной задаче (под каждую VIPER обычно настраивают по-разному).

Практика

Откроем снова код проекта, который соответствует началу урока (не переписан на **MVVM**). Добавим файлы **SearchRouter.swift** и **SearchInteractor.swift**. Начнем, как и в примере в теоретической части, с роутера:

```
protocol SearchRouterInput {

```

```
    func openDetails(for app: ITunesApp)

    func openAppInITunes(_ app: ITunesApp)
}

final class SearchRouter: SearchRouterInput {

    weak var viewController: UIViewController?

    func openDetails(for app: ITunesApp) {
        let appDetaillViewController = AppDetailViewController(app: app)

self.viewController?.navigationController?.pushViewController(appDetaillViewCont
roller, animated: true)
    }

    func openAppInITunes(_ app: ITunesApp) {
        guard let urlString = app.appUrl, let url = URL(string: urlString) else
{
            return
        }
        UIApplication.shared.open(url, options: [:], completionHandler: nil)
    }
}
```

Роутер имеет два метода — один открывает экран деталей приложения, другой открывает приложение в нативном **AppStore** (на симуляторе не работает, только на устройстве). Роутеру нужно иметь ссылку на текущий вью-контроллер, чтобы показать новый экран из него. Это должна быть слабая ссылка, иначе будет **reference cycle** (**ViewController -> Presenter -> Router -> ViewController**).

Перейдем к интерактору. В нашем примере он будет очень простой, так как на экране поиска приложений у нас нет бизнес-логики. Интерактор будет заниматься получением данных из сети:

```
import Alamofire

protocol SearchInteractorInput {

    func requestApps(with query: String, completion: @escaping
(Result<[ITunesApp]>) -> Void)
}

final class SearchInteractor: SearchInteractorInput {

    private let searchService = ITunesSearchService()

    func requestApps(with query: String, completion: @escaping
(Result<[ITunesApp]>) -> Void) {
        self.searchService.getApps(forQuery: query, then: completion)
    }
}
```

В нашем случае интерактор является просто прокси к классу **ITunesSearchService**, но в проекте может появиться еще много логики на этом экране, и ее лучше держать именно в интеракторе, а не в презентере. Например, интерактор может сохранять в кеш данные от **ITunesSearchService**, управлять временем жизни этого кеша и т. д.

Внесем правки в **Presenter**. Добавим в него свойства **Interactor** и **Router**, их **Presenter** будет держать сильными ссылками. **Presenter** должен получить конкретные экземпляры этих классов при инициализации:

```
let interactor: SearchInteractorInput
let router: SearchRouterInput

init(interactor: SearchInteractorInput, router: SearchRouterInput) {
    self.interactor = interactor
    self.router = router
}
```

Навигацией теперь занимается **Router**, поэтому надо заменить реализацию метода **viewDidSelectApp**:

```
func viewDidSelectApp(_ app: ITunesApp) {
    self.router.openDetails(for: app)
}
```

Также надо удалить свойство **private let searchService = ITunesSearchService()** и заменить обращение к нему для получения данных на обращение к соответствующему методу интерактора.

И, наконец, **Builder** теперь обрастает кодом по настройке более разделенного на классы модуля:

```
final class SearchModuleBuilder {

    static func build() -> (UIViewController & SearchViewInput) {
        let router = SearchRouter()
        let interactor = SearchInteractor()
        let presenter = SearchPresenter(interactor: interactor, router: router)
        let viewController = SearchViewController(output: presenter)

        presenter.viewInput = viewController
        router.viewController = viewController

        return viewController
    }
}
```

Преимущества и недостатки VIPER

Мы уже обсуждали плюсы и минусы архитектуры MVP. Поскольку VIPER движется в том же направлении, разделяя ответственности между классами еще сильнее, то и преимущества с

недостатками у них будут одни и те же. Только плюсы в VIPER становятся еще более весомыми, как минусы — те, что есть у MVP, в VIPER бьют еще больнее.

Напомним их.

Плюсы

- Лучше разделение — значит, лучше переиспользуемость и тестируемость, легче поддерживать код, легче вносить атомарные изменения, не задевая все сразу. Это гигантский плюс, который может перекрыть все остальное.
- Разделение ответственностей единообразное — так что просматривая незнакомый VIPER-модуль, мы уже знаем, чем конкретно занимается каждый из его классов-слоев.

Минусы

- Больше классов, в целом больше кода, в том числе шаблонного, а из-за этого растет время компиляции проекта.
- **Presenter** все равно подвержен опасности разрастания до очень большого класса, так как он является центральным элементом в модуле. Следить за этим — ответственность разработчика, предписаний у самой архитектуры по этому поводу нет.
- Все модули строятся от **View**, то есть являются UI-зависимыми (кстати, именно эту проблему решали Uber внедрением своей собственной архитектуры **RIBs**).

Какую архитектуру использовать?

MVC проще, но у **MVP** и **VIPER** такие шикарные преимущества. А еще есть **MVVM**, который тоже часто используется. Так что вопрос выбора архитектуры всегда решается внутри команды разработчиков, работающих над конкретным продуктом. Вот общие советы:

1. Если проект маленький и не планирует разрастаться — лучше использовать **MVC** и не заморачиваться с архитектурами.
2. Если вы один работаете над проектом — в большинстве случаев **MVC** будет достаточно, но в зависимости от сложности UI-составляющей плюсы **MVP** и **MVVM** могут оказаться очень полезными.
3. Если проект большой или в перспективе станет таким — надо задуматься над использованием более разделенной архитектуры. Если при этом над проектом работает более 3–5 человек одновременно, проблемы **MVC** могут оказаться очень болезненными на определенном этапе.
4. **MVVM** стоит использовать, когда нужна жесткая привязка отображения к изменяющимся данным.
5. **VIPER** — это хорошо, но в широком классе проектов он будет избыточным.
6. **MVP** — золотая середина среди всех архитектур. Так что, выбирая, можно начать с нее.

Практическое задание

1. Переписать экран поиска по песням (из предыдущего практического задания) на архитектуру **VIPER**.

2. * Сделать экран проигрывания песни. По-настоящему звук можно не проигрывать, просто сделать сервис, аналогичный **FakeDownloadAppsService** с урока. По **MVVM** сделать экран с обновлением **UI** по ходу проигрывания песни. Дизайн в минимальном виде повторите из нативного приложения «Музыка».

3. ** В интеракторах модулей реализуйте логику сохранения полученных данных в кеш.

Дополнительные материалы

1. Паттерны для новичков: MVC vs MVP vs MVVM.
2. Архитектурные паттерны в iOS.
3. Книга VIPER.
4. Козел отпущения, или MVC в iOS.

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. Model — View — Controller.
2. Э. Гамма, Р. Хелм, Р. Джонсон, Дж. Влиссидес «Приемы объектно-ориентированного проектирования. Паттерны проектирования».