GeekBrains

Архитектуры и шаблоны проектирования на Swift

# Архитектурные паттерны. Часть 1. MVC, MVP

Основной архитектурный паттерн MVC. Архитектурный паттерн MVP и его отличия от MVC.

## Оглавление

# Введение

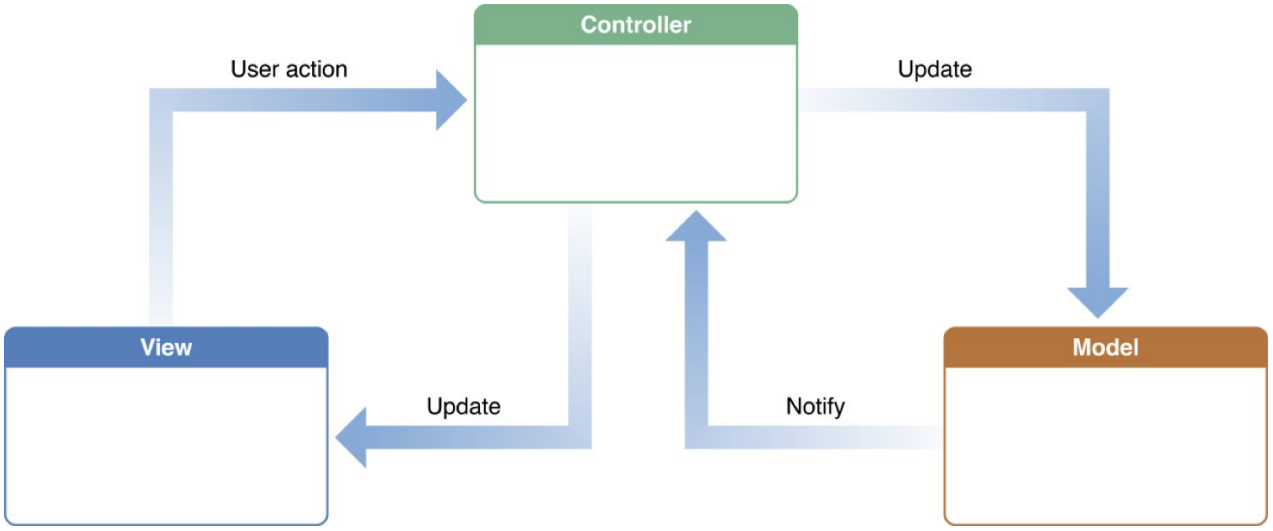
На этом и следующем уроке рассмотрим архитектурные паттерны для мобильной iOS-разработки. Они похожи на структурные паттерны проектирования, но имеют намного более широкий охват. Архитектурный шаблон предписывает выполнять разделение ответственностей между классами на самом высоком уровне. Если не следовать ни одному архитектурному паттерну, возникнут проблемы в кодовой базе, из-за которых станет сложно поддерживать и изменять код.

На этих двух уроках мы рассмотрим четыре наиболее важных для iOS-разработчика архитектурных паттерна: **MVC**, **MVVM**, **MVP**, **VIPER**. Есть еще множество других, менее популярных и используемых реже. У каждого архитектурного паттерна есть плюсы и минусы. Есть задачи, которые превосходно решает один паттерн, а бывает, что система будет более понятной и поддерживаемой на основе другого паттерна. Отличия, преимущества и недостатки, области применения архитектурных паттернов мы также изучим на этих уроках.

# MVC

## Теория

**MVC** — аббревиатура от **Model — View — Controller**. Это архитектурный паттерн, который предписывает все верхнеуровневые компоненты системы разделять на три группы: слой модели (**Model**), слой отображения (**View**) и слой, который связывает модель и отображение логикой (**Controller**). Схема паттерна:

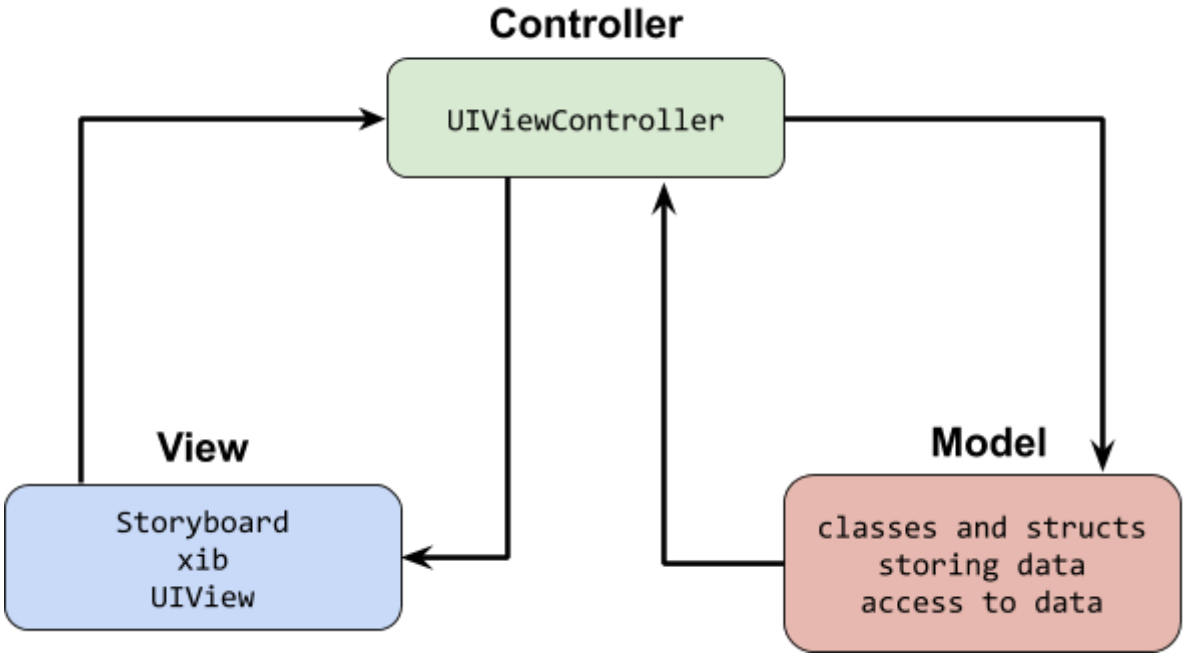



- Модель хранит внутренние данные о системе, предоставляет к ним доступ и содержит бизнес-логику приложения.
- View (отображение) занимается только пользовательским интерфейсом. Она отображает то, что ей скажут, причем знает, как именно это нужно отобразить (верстка, анимации, шрифты и т. д.).
- Контроллер выступает в роли медиатора между View и Model:
    - имеет доступ к модели и узнает, когда она обновляется;
    - сам может обновить модель, перезаписать данные;

- указывает View, какие данные нужно отобразить (и иногда — как их нужно отобразить);
- пользователь взаимодействует только с View через интерфейс. Controller обрабатывает это взаимодействие и решает, что из перечисленного выше нужно сделать.

# MVC в iOS

**iOS SDK** спроектирован так, чтобы легко поддерживать паттерн MVC из коробки. Паттерн можно нарушить, если не следовать его принципам (об этом позже), но концепция классов в iOS SDK все же располагает к тому, чтобы использовать MVC по умолчанию.




- View, как пользовательский интерфейс, включает в себя сториборды и xib'ы, то есть все элементы **interface builder**, а также все сабклассы **UIView**, в которых содержится логика по отображению. Сабклассы **UIView** не должны заниматься хранением данных или бизнес-логикой приложения, они только отображают эти данные.
- **Model** представлена совокупностью классов и структур в приложении, которые отвечают за хранение, передачу и любые другие манипуляции с данными.
- Связующим звеном является наследник **UIViewController**. Он же представляет собой и экран приложения со своей View. Именно он берет данные из модели, а затем отображает их на View. Оповещение об изменении данных модели может быть реализовано через паттерн **Observer**. Также **UIViewController** ловит все пользовательские действия (о них он узнает от view) и решает, как обработать эти действия.

Как правило, каждый экран приложения представлен своим **UIViewController** и своей View. При этом классы модели могут использоваться разными контроллерами, а модель — общая для всего приложения.

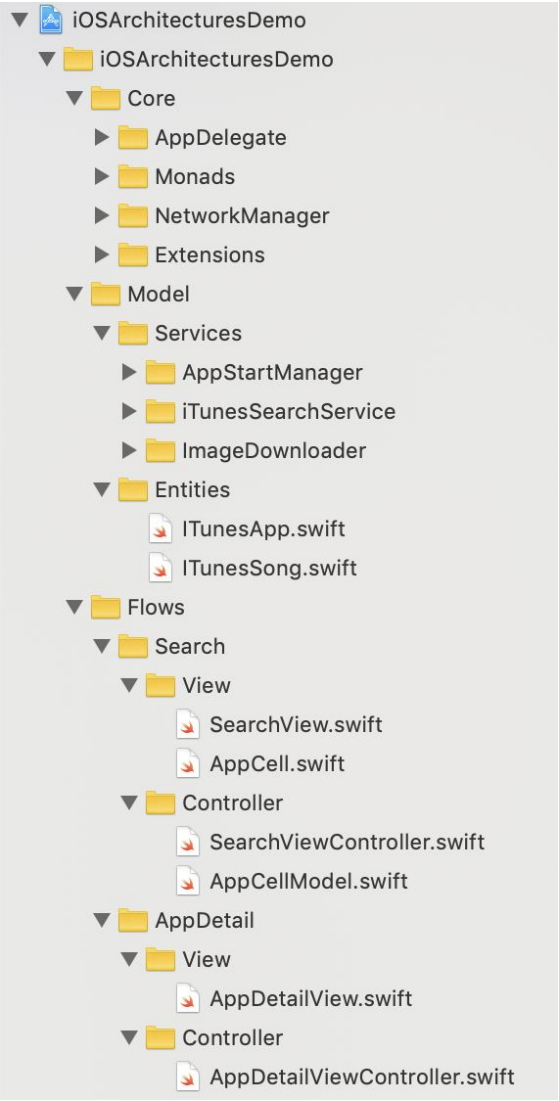
Такое разделение выглядит логичным. Более того, само наличие такого класса, как **UIViewController**, и необходимость наследоваться от него автоматически обеспечивает минимальное следование этому паттерну.

# Практика

Посмотрим, как MVC реализуется в коде. Для изучения архитектурных паттернов мы воспользуемся проектом, состоящим из двух экранов — экрана поиска приложений по iTunes и экрана, отображающего данные о выбранном приложении.

Код находится по ссылке: https://drive.google.com/open?id=12rD7ellmgxJunwkXCVKw3zJS70zwcElP.

Структура проекта выглядит следующим образом:




На слое **Model** у нас присутствуют структуры **ITunesApp** и **ITunesSong** (в папке **Entities**), которые хранят данные о сущностях. Это структуры для хранения данных о приложении и данных о песне из iTunes соответственно. На уроке мы будем запрашивать только приложения, а в практическом задании нужно будет использовать запросы к списку песен. Эти структуры используются во вью-контроллерах для отображения данных. Для доступа к этим данным из интернета служит класс **iTunesSearchService**.

Также на слое **Model** лежат более низкоуровневые компоненты, которые отвечают за логику приложения: **AppStartManager** конфигурирует приложение при его старте, **ImagesDownloader** позволяет скачивать картинки из сети.

В папке **Flows** расположены верхнеуровневые компоненты приложения. Это два экрана — экран поиска со списком и экран деталей конкретного приложения. Каждый **Flow** содержит слой **View** и слой

**Controller** по архитектуре **MVC**. **Controller** представлен соответствующим вью-контроллером (**SearchViewController** для экрана поиска и **AppDetailViewController** для экрана деталей). На слое View находится ячейка **AppCell**. Ее код не содержит логики — она только отображает информацию о приложении в трех лейблах (позже мы добавим на нее кнопки). Сторибордов и xib'ов в проекте нет, вся остальная настройка пользовательского интерфейса происходит в коде в наследниках **UIView**. Мы не воспользовались **interface builder**, чтобы лучше продемонстрировать архитектурные паттерны и их минусы.

Так в данном проекте происходит разделение между тремя слоями по архитектуре **MVC**.

Рассмотрим на примере **SearchViewController**, как взаимодействуют слои. Вью-контроллер обрабатывает пользовательский ввод в текстовое поле посредством **UISearchBarDelegate**:

```
extension SearchViewController: UISearchBarDelegate {
    func searchBarSearchButtonClicked(_ searchBar: UISearchBar) {
        guard let query = searchBar.text else {
            searchBar.resignFirstResponder()
            return
        }
        if query.count == 0 {
            searchBar.resignFirstResponder()
            return
        }
        self.requestApps(with: query)
    }
}
```

Если был совершен пользовательский ввод текста, то вью-контроллер обращается к сервису для получения данных, то есть обращается к слою **Model**. Это происходит в функции **requestApps(with query: String)**:

```
private func requestApps(with query: String) {
    self.throbber(show: true)
    self.searchResults = []
    self.tableView.reloadData()

    self.searchService.getApps(forQuery: query) { [weak self] result in
        guard let self = self else { return }
        self.throbber(show: false)
        result
            .withValue { apps in
                guard !apps.isEmpty else {
                    self.searchResults = []
                    self.showNoResults()
                    return
                }
                self.hideNoResults()
                self.searchResults = apps

                self.tableView.isHidden = false
                self.tableView.reloadData()

                self.searchBar.resignFirstResponder()
            }
            .withError {
                self.showError(error: $0)
            }
    }
}
```

Во время выполнения этого запроса вью-контроллер полностью управляет состоянием своей view: показывает активити индикатор; скрывает, а затем показывает таблицу с результатами поиска; если ничего не нашлось, то показывает лейбл, информирующий о пустой выдаче; если произошла ошибка, то показывает информационный алерт.

Отображение данных в таблице вью-контроллер делает посредством **UITableViewDataSource**:

```
//MARK: - UITableViewDataSource
extension SearchViewController: UITableViewDataSource {

    func tableView(_ tableView: UITableView, numberOfRowsInSection section: Int)
-> Int {
        return searchResults.count
    }

    func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath)
-> UITableViewCell {
        let dequeuedCell = tableView.dequeueReusableCell(withIdentifier:
Constants.reuseIdentifier, for: indexPath)
        guard let cell = dequeuedCell as? AppCell else {
            return dequeuedCell
        }
        let app = self.searchResults[indexPath.row]
        let cellModel = AppCellModelFactory.cellModel(from: app)
        cell.configure(with: cellModel)
        return cell
    }
}
```

Обратите внимание, как вью-контроллер конфигурирует ячейку, то есть слой View. Он не передает ей модель, а создает промежуточную структуру **AppCellModel** с помощью паттерна «фабрика» и передает ячейке эту структуру, которая уже содержит готовые данные для отображения. View не знает ничего о модели, отображением данных продолжает управлять вью-контроллер.

Также пользовательское взаимодействие происходит через нажатия на ячейки. Эти нажатия ловятся вью-контроллером через **UITableViewDelegate**:

```
extension SearchViewController: UITableViewDelegate {
    func tableView(_ tableView: UITableView, didSelectRowAt indexPath:
IndexPath) {
        tableView.deselectRow(at: indexPath, animated: true)
        let app = searchResults[indexPath.row]
        let appDetaillViewController = AppDetailViewController()
        appDetaillViewController.app = app
        navigationController?.pushViewController(appDetaillViewController,
animated: true)
    }
}
```

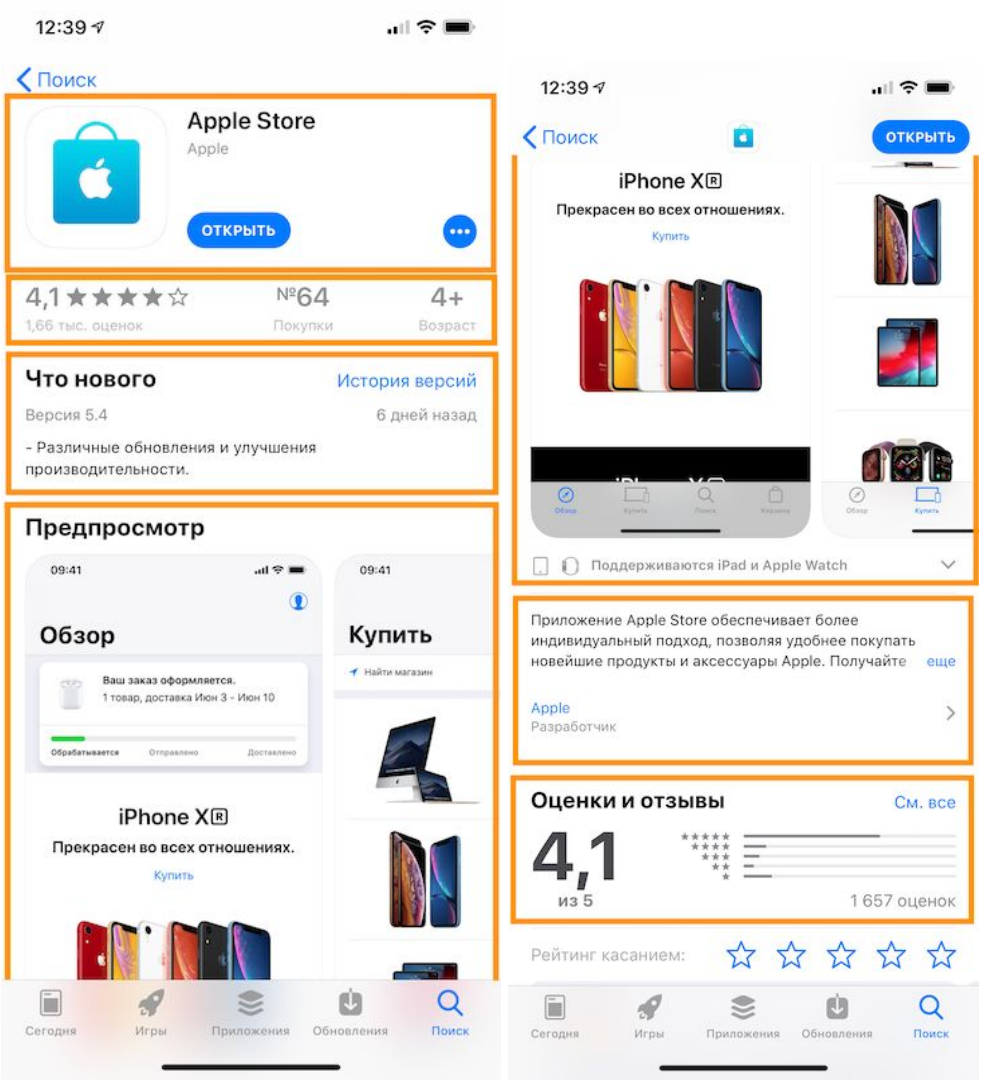
Вью-контроллер решает, что нужно сделать по нажатию на ячейку. В нашем случае вью-контроллер инициализирует экран деталей приложения и открывает его, добавляя на стек своего **navigation controller**.

Тут нужно оговориться: хоть навигация по приложению и относится к его бизнес-логике (а значит, должна по архитектуре MVC осуществляться во вью-контроллере), iOS SDK позволяет делать такую навигацию прямиком во вью-слое — через механизм **Segue**.

# MVC с разделением на Child View Controllers

Часто мы следуем концепции «один экран — один **UIViewController**». Но представим очень сложный экран — например, экран просмотра приложения в AppStore или страницы в соцсети. На них много разносторонних данных и сложная логика. Очевидно, что такой класс-наследник **UIViewController** будет невероятно сложным. В паттерне MVC эту проблему можно решить путем разделения одного экрана на несколько наследников **UIViewController** со своей кастомной **UIView**, каждый из которых будет отвечать за отдельный логический модуль.

Например, вот так могло бы выглядеть разбиение сложного экрана на сабмодули, где каждый прямоугольник соответствует отдельному наследнику **UIViewController** со своим отображением и логикой:




Как сделать так, чтобы экран был составлен из нескольких **UIViewController**? Это делается с помощью механизма **child view controllers**. Суть проста: на один **UIViewController** добавляется другой **UIViewController** аналогично тому, как мы на одну **UIView** добавляем другую в качестве сабвью. Кстати, вспомните пятый урок и паттерн **Composite** (компоновщик) — механизм **child view controllers** работает именно на нем.

Добавление **child view controller** делается в несколько шагов:

```
let parent = UIViewController()
let child = UIViewController()

// 1
parent.view.addSubview(child.view)
// 2
parent.addChild(child)
// 3
child.didMove(toParent: parent)
```

А затем для лэйаута **child.view** создаются констрейнты, как для обычной view, которую бы мы разместили на **parent.view**.

Таким образом, сложный экран — это один большой **UIViewController**, который вмещает множество мелких сабмодулей, тоже представленных **UIViewController**. Каждый сабмодуль реализует свой UI и свою логику, а общий вью-контроллер размещает на себе сабмодули через механизм **child view controllers** и занимается общей логикой — например, навигацией, общими запросами к данным и т. д.

Потренируемся использовать данный подход в приложении, с которым мы начали работать на этом уроке.

Перейдем в **AppDetailViewController**. Сейчас он прост и только отображает картинку. Усложним этот экран, дизайн приблизительно скопируем с AppStore. Напишем наш первый сабмодуль для этого экрана — **AppDetailHeaderViewController**. Он будет содержать картинку, название приложения и некоторые другие UI-элементы. Верстка делается в коде в классе **AppDetailHeaderView** — это наследник **UIView**, который будет подключаться к контроллеру как его **self.view**.

Сначала взглянем на view **AppDetailHeaderView** с верстkой:

```
final class AppDetailHeaderView: UIView {

    // MARK: - Subviews

    private(set) lazy var imageView: UIImageView = {
        let imageView = UIImageView()
        imageView.translatesAutoresizingMaskIntoConstraints = false
        imageView.layer.cornerRadius = 30.0
        imageView.layer.masksToBounds = true
        return imageView
    }()

    private(set) lazy var titleLabel: UILabel = {
        let label = UILabel()
        label.translatesAutoresizingMaskIntoConstraints = false
        label.textColor = .black
        label.font = UIFont.boldSystemFont(ofSize: 20.0)
        label.numberOfLines = 2
        return label
    }()

    private(set) lazy var subtitleLabel: UILabel = {
```

```
        let label = UILabel()
        label.translatesAutoresizingMaskIntoConstraints = false
        label.textColor = .lightGray
        label.font = UIFont.systemFont(ofSize: 14.0)
        return label
    }()

    private(set) lazy var openButton: UIButton = {
        let button = UIButton(type: .system)
        button.translatesAutoresizingMaskIntoConstraints = false
        button.setTitle("Открыть", for: .normal)
        button.backgroundColor = UIColor(white: 0.9, alpha: 1.0)
        button.layer.cornerRadius = 16.0
        return button
    }()

    private(set) lazy var ratingLabel: UILabel = {
        let label = UILabel()
        label.translatesAutoresizingMaskIntoConstraints = false
        label.textColor = .lightGray
        label.font = UIFont.boldSystemFont(ofSize: 20.0)
        return label
    }()

    // MARK: - Init

    override init(frame: CGRect) {
        super.init(frame: frame)
        self.setupLayout()
    }

    required init?(coder aDecoder: NSCoder) {
        super.init(coder: aDecoder)
        self.setupLayout()
    }

    // MARK: - UI

    private func setupLayout() {
        self.addSubview(self.imageView)
        self.addSubview(self.titleLabel)
        self.addSubview(self.subtitleLabel)
        self.addSubview(self.openButton)
        self.addSubview(self.ratingLabel)

        NSLayoutConstraint.activate([
            self.imageView.topAnchor.constraint(equalTo:
self.safeAreaLayoutGuide.topAnchor, constant: 12.0),
            self.imageView.leftAnchor.constraint(equalTo: self.leftAnchor,
constant: 16.0),
            self.imageView.widthAnchor.constraint(equalToConstant: 120.0),
            self.imageView.heightAnchor.constraint(equalToConstant: 120.0),
```

```
            self.titleLabel.topAnchor.constraint(equalTo:
self.safeAreaLayoutGuide.topAnchor, constant: 12.0),
            self.titleLabel.leftAnchor.constraint(equalTo:
self.imageView.rightAnchor, constant: 16.0),
            self.titleLabel.rightAnchor.constraint(equalTo: self.rightAnchor,
constant: -16.0),

            self.subtitleLabel.topAnchor.constraint(equalTo:
self.titleLabel.bottomAnchor, constant: 12.0),
            self.subtitleLabel.leftAnchor.constraint(equalTo:
self.titleLabel.leftAnchor),
            self.subtitleLabel.rightAnchor.constraint(equalTo:
self.titleLabel.rightAnchor),

            self.openButton.leftAnchor.constraint(equalTo:
self.imageView.rightAnchor, constant: 16.0),
            self.openButton.bottomAnchor.constraint(equalTo:
self.imageView.bottomAnchor),
            self.openButton.widthAnchor.constraint(equalToConstant: 80.0),
            self.openButton.heightAnchor.constraint(equalToConstant: 32.0),

            self.ratingLabel.topAnchor.constraint(equalTo:
self.imageView.bottomAnchor, constant: 24.0),
            self.ratingLabel.leftAnchor.constraint(equalTo:
self.imageView.leftAnchor),
            self.ratingLabel.widthAnchor.constraint(equalToConstant: 100.0),

            self.ratingLabel.bottomAnchor.constraint(equalTo: self.bottomAnchor)
            ])
    }
}
```

А вот ее контроллер, который уже будет знать о модели данных и заполнит view нужной информацией для отображения:

```
final class AppDetailHeaderViewController: UIViewController {

    // MARK: - Properties

    private let app: ITunesApp

    private let imageDownloader = ImageDownloader()

    private var appDetailHeaderView: AppDetailHeaderView {
        return self.view as! AppDetailHeaderView
    }

    // MARK: - Init

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

```
        super.init(nibName: nil, bundle: nil)
    }

    required init?(coder aDecoder: NSCoder) {
        fatalError("init(coder:) has not been implemented")
    }

    // MARK: - Lifecycle

    override func loadView() {
        super.loadView()
        self.view = AppDetailHeaderView()
    }

    override func viewDidLoad() {
        super.viewDidLoad()
        self.fillData()
    }

    // MARK: - Private

    private func fillData() {
        self.downloadImage()
        self.appDetailHeaderView.titleLabel.text = app.appName
        self.appDetailHeaderView.subtitleLabel.text = app.company
        self.appDetailHeaderView.ratingLabel.text = app.averageRating >>- {
"\($0)" }
    }

    private func downloadImage() {
        guard let url = self.app.iconUrl else { return }
        self.imageDownloader.getImage(fromUrl: url) { [weak self] (image, _) in
            self?.appDetailHeaderView.imageView.image = image
        }
    }
}
```

Перейдем обратно в «родительский» вью-контроллер и добавим только что написанный сабмодуль в качестве **child view controller**. Перепишем **AppDetailViewController** следующим образом:

```
final class AppDetailViewController: UIViewController {

    let app: ITunesApp

    lazy var headerViewController = AppDetailHeaderViewController(app: self.app)

    init(app: ITunesApp) {
        self.app = app
        super.init(nibName: nil, bundle: nil)
    }
```

```
    required init?(coder aDecoder: NSCoder) {
        fatalError("init(coder:) has not been implemented")
    }

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

    private func configureUI() {
        self.view.backgroundColor = .white
        self.navigationController?.navigationBar.tintColor = UIColor.white;
        self.navigationItem.largeTitleDisplayMode = .never
        self.addHeaderViewController()
        self.addDescriptionViewController()
    }

    private func addHeaderViewController() {
        self.addChild(self.headerViewController)
        self.view.addSubview(self.headerViewController.view)
        self.headerViewController.didMove(toParent: self)

        self.headerViewController.view.translatesAutoresizingMaskIntoConstraints
= false
        NSLayoutConstraint.activate([
            self.headerViewController.view.topAnchor.constraint(equalTo:
self.view.safeAreaLayoutGuide.topAnchor),
            self.headerViewController.view.leftAnchor.constraint(equalTo:
self.view.leftAnchor),
            self.headerViewController.view.rightAnchor.constraint(equalTo:
self.view.rightAnchor)
            ])
    }

    private func addDescriptionViewController() {
        // TODO: ДЗ, сделать другие сабмодули
        let descriptionViewController = UIViewController()

        self.addChild(descriptionViewController)
        self.view.addSubview(descriptionViewController.view)
        descriptionViewController.didMove(toParent: self)

        descriptionViewController.view.translatesAutoresizingMaskIntoConstraints
= false
        NSLayoutConstraint.activate([
            descriptionViewController.view.topAnchor.constraint(equalTo:
self.headerViewController.view.bottomAnchor),
            descriptionViewController.view.leftAnchor.constraint(equalTo:
self.view.leftAnchor),
            descriptionViewController.view.rightAnchor.constraint(equalTo:
self.view.rightAnchor),

descriptionViewController.view.heightAnchor.constraint(equalToConstant: 250.0)
```

```
            ])
    }
}
```

Посмотрите на метод **addDescriptionViewController** — сейчас он добавляет пустой **UIViewController**, потому что никакого **DescriptionViewController** мы не написали — это будет частью вашего практического задания, где вы дальше будете тренироваться использовать данный подход.

Осталось обсудить возможные нарушения паттерна и его минусы.

# Нарушения MVC

**MVC** — несложный архитектурный паттерн с четким разделением обязанностей, и архитектура iOS SDK к нему располагает. Тем не менее встречаются и нарушения, которые могут привести к нежелательным последствиям. Рассмотрим их.

1. *View взаимодействует с Model напрямую.*

Как мы видели на схеме, **Controller** является посредником между **View** и **Model**, которые друг о друге ничего не знают. Взаимодействие между **View** и **Model** исключается для того, чтобы уменьшить связность между классами.

Однако нередки случаи непроизвольного нарушения этого принципа. Допустим, нам не нравится хранить код по настройке ячейки в методе **tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath)** у вью-контроллера. Мы решили сделать метод у ячейки, который принимает модель и сам у себя настраивает отображение, а затем поместили этот метод в класс **AppCell**:

```
func configure(with app: ITunesApp) {
    self.titleLabel.text = app.appName
    self.subtitleLabel.text = app.company
    self.ratingLabel.text = app.averageRating >>- { "\($0)" }
}
```

Очевидно, теперь класс **AppCell** (представитель слоя View) знает о существовании класса **ITunesApp** (представителя слоя Model). Взаимодействие между слоями нарушено. Это не самый страшный пример нарушения MVC, потому что от такой связности между классами легко избавиться, но один из самых частых.

2. *Слой выполняет не свои функции.*

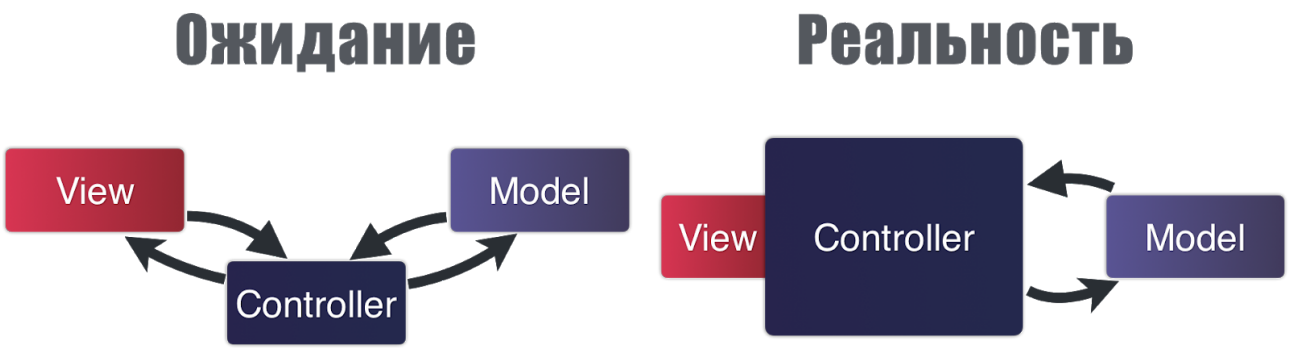
Такое нарушение MVC может возникнуть из-за того, что объекты слоя View начинают делать слишком много. По идее, они должны заниматься только отображением и исключительно тех данных, которые предоставляет им контроллер. Однако этот принцип легко нарушить. Посмотрим, как это можно сделать, на примере ячейки **AppCell**. Если мы добавим на эту ячейку отдельную кнопку, по которой можно будет перейти еще на какой-нибудь экран, то это и будет нарушением ответственности **View**. Аналогично, если по нажатию на эту кнопку будет обращение к данным из самой **View**, запросы в сеть и т. д. Это обязанность контроллера.

Например, если в ячейку при ее конфигурации передавать модель **ITunesApp**, а по нажатию на кнопку в этой ячейке вызывать открытие страницы приложения в Safari (представьте, что метод расположен в классе ячейки), то это и будет примером нарушения MVC:

```
@objc private func openButtonPressed() {
```

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

## Проблемы MVC




Главная проблема MVC — слишком большие вью-контроллеры, которые берут на себя гигантское количество ответственностей. На самом деле зачастую это проблема проектирования, а не архитектурного паттерна, ведь часть логики можно вынести в другие классы, сервисы и в модель, а часть конфигурации отображения — в само отображение, то есть во View. Но проблема все равно есть. Взглянем на все ответственности, которые обычно несет вью-контроллер:

1. Конфигурировать UI;
2. Обновлять UI;
3. Ловить user interaction;
4. Обрабатывать user interaction;
5. Реализовывать бизнес-логику;
6. Обеспечивать навигацию между экранами.

Из-за такого количества ответственностей у класса **UIViewController** он становится чрезвычайно большим и запутанным. Разработчики назвали эту проблему **Massive View Controller** (массивный вью-контроллер) — аббревиатура такая же, как и у названия паттерна.

С этой проблемой можно бороться в рамках паттерна MVC. Ответственность № 6 -— навигацию — можно вынести в отдельные классы, воспользовавшись, например, паттерном **Coordinator** ( о нем рассказывалось на других курсах). Часть бизнес-логики желательно выносить во фреймворки, оставляя вью-контроллеру как можно меньше логики — и только ту, которая непосредственно связана с отображением. Так что вынесением ответственностей № 5 и № 6 в другие классы можно существенно разгрузить вью-контроллер.

Но существует другая проблема в MVC на iOS. Как известно, **UIViewController** контролирует жизненный цикл своего отображения (**viewDidLoad**, **viewDidAppear** и т. д.). У контроллера всегда есть **view**, и она неотделима от контроллера. На практике получается очень сложно отделить слои

View и Controller друг от друга так, чтобы они работали абсолютно независимо. Unit-тестирование и переиспользование компонентов такой системы оказывается затруднено.

# MVP

## Теория

Мы только что обсудили важную проблему архитектуры MVC — **UIViewController**, хоть и является контроллером, но неотделим от своей View. Возникает идея — а что если мы будем рассматривать **UIViewController** не как контроллер, а как View? То есть из тех же самых слоев (View, Controller, Model) мы представляем слой View уже как совокупность и сторибордов/xib'ов, и наследников **UIView**, и наследников **UIViewController**. Теперь не делаем большого различия между UIView и UIViewController с теоретической точки зрения, просто эти классы немного по-разному умеют отображать пользовательский интерфейс. Роль **Controller** отведем другому классу — это будет обычный объект, исполняющий роль медиатора.

Взглянем на ту же самую идею с позиций концепции ответственностей классов. Вю-контроллер исполнял следующие ответственности:

1. Конфигурировать UI;
2. Обновлять UI;
3. Ловить user interaction;
4. Обрабатывать user interaction;
5. Реализовывать бизнес-логику;
6. Обеспечивать навигацию между экранами.

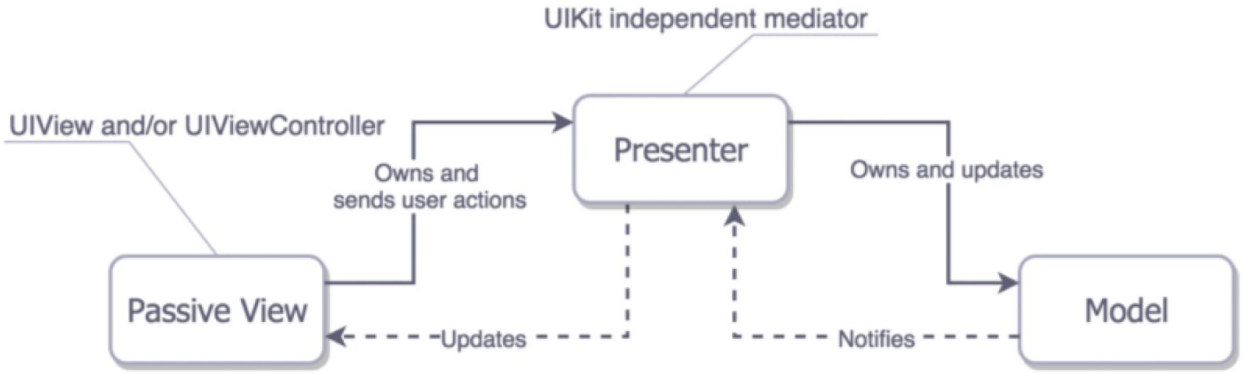
Разделим ответственности на две части:

1. Конфигурировать UI;
2. Обновлять UI;
3. Ловить user interaction.

4. Обрабатывать user interaction;
5. Реализовывать бизнес-логику;
6. Обеспечивать навигацию между экранами.

Первые три — очевидные ответственности слоя View — будет брать на себя **UIViewController** и все его сабвью. 4, 5 и 6 — это уже логика работы приложения, которая должна быть на плечах **Controller**.

Это и есть архитектура **MVP** — **Model — View — Presenter.** Посредник между моделью и View — Presenter, и суть в том, что он не является наследником **UIViewController** и вообще никак не зависит от **UIKit**. В остальном все остается так, как и было ранее. Схема паттерна:




# Практика

Есть экран **Search**, написанный по архитектуре **MVC** и представленный вью-контроллером **SearchViewController**, а на его слое **View** есть ячейка **AppCell**. Поставим себе задачу перевести этот экран на архитектуру **MVP**. Ключевой момент — разделить ответственности. Сейчас все они лежат на **SearchViewController**, а нам предстоит создать новый класс **SearchPresenter** и вынести в него часть кода из вью-контроллера. В итоге презентер должен будет содержать всю логику, а от вью-контроллера останется только UI.

Зайдем в класс **SearchViewController** и перечислим задачи, которые делает он, а должен — Presenter:

- вью-контроллер сам запрашивает данные из сети в функции **func requestApps(with query: String)**;

- вью-контроллер занимается навигацией по приложению и открывает экран деталей приложения в реализации метода **tableView(_ tableView: UITableView, didSelectRowAt indexPath: IndexPath)**.

Создадим класс **SearchPresenter** и перетащим туда эту логику:

```
final class SearchPresenter {

    private let searchService = ITunesSearchService()

    private func requestApps(with query: String) {
        self.searchService.getApps(forQuery: query) { [weak self] result in
            // а тут надо отобразить полученные данные на view
        }
    }

    private func openAppDetails(with app: ITunesApp) {
        // тут надо проинициализировать и запушить вью-контроллер деталей
приложения
    }
}
```

Тут пока оставлены заготовки функций, реализацию скоро добавим.

Приложения мы запросили из сети, использовав наш удобный класс, но как теперь их отобразить? Ведь презентер — это не **UIViewController**, где все было в одном месте, а обычный класс. Мы подошли к самому важному — к обеспечению «общения» между слоями **View** и **Presenter**.

Вспомним концепцию ответственностей слоев. View просто отображает что-то на UI, но сама ничего не решает. Еще ловит пользовательский ввод, но не обрабатывает его, а «говорит» Presenter'у, что пользовательский ввод произошел. Исходя из этой концепции, создадим два протокола: **SearchViewInput** и **SearchViewOutput**. Первый — это набор сообщений, которые презентер может «передать» вью-контроллеру (например, отрисуй это, покажи то). Это такая входная точка для слоя View, поэтому протокол и называется **ViewInput**. Второй протокол — это, наоборот, выходная точка слоя View. Это тот набор сообщений, которые View может передать презентеру — юзер нажал на эту кнопку, юзер ввел такой-то символ и т. д.

Напишем эти протоколы:

```
protocol SearchViewInput: class {

    var searchResults: [ITunesApp] { get set }

    func showError(error: Error)

    func showNoResults()

    func hideNoResults()

    func throbber(show: Bool)
}

protocol SearchViewOutput: class {

    func viewDidSearch(with query: String)

    func viewDidSelectApp(_ app: ITunesApp)
}
```

Теперь должно быть понятно, какими «сообщениями» будут обмениваться **ViewController** и **Presenter**. **SearchViewController** реализует протокол **SearchViewInput**, а **SearchPresenter** — протокол **SearchViewOutput**. У вью-контроллера будет ссылка на презентер, и у него он сможет вызывать методы из **SearchViewOutput**. У презентера будет ссылка на вью-контроллер, и у него он сможет вызывать методы **SearchViewInput**. При этом здесь есть еще два правила:

- Вью-контроллер держит сильной ссылкой объект **SearchViewOutput** (то есть презентер). Презентер держит **слабой** ссылкой объект **SearchViewInput** (вью-контроллер). Это нужно, чтобы избежать циклической ссылки.

  *Примечание: может возникнуть вопрос — почему сильная ссылка именно у вью-контроллера на презентер, а не наоборот? Это связано с тем, что когда вью-контроллер отображается на экране, на него держится сильная ссылка и он не может уйти из памяти. Поэтому, когда мы отображаем вью-контроллер, вместе с ним держится и презентер, а когда вью-контроллер уходит с экрана, то вместе с ним освобождается и объект презентера. Это просто более удобно, чем вручную управлять удержанием и освобождением объекта презентера (но и такой подход возможен).*

- Вью-контроллер имеет свойство именно типа протокола **SearchViewOutput**, а не конкретного класса **SearchPresenter**. У презентера свойство именно типа протокола **SearchViewInput**, а не конкретного класса **SearchViewController**. Благодаря этому мы получаем весомые преимущества в такой архитектуре. Например, можно для одного презентера создать несколько вью с разным UI. Также это необходимо для unit-тестирования.

Теперь мы полностью готовы. Возвращаемся в класс **SearchPresenter** и дописываем его:

```
final class SearchPresenter {

    weak var viewInput: (UIViewController & SearchViewInput)?

    private let searchService = ITunesSearchService()

    private func requestApps(with query: String) {
        self.searchService.getApps(forQuery: query) { [weak self] result in
            guard let self = self else { return }
            self.viewInput?.throbber(show: false)
            result
                .withValue { apps in
                    guard !apps.isEmpty else {
                        self.viewInput?.showNoResults()
                        return
                    }
                    self.viewInput?.hideNoResults()
                    self.viewInput?.searchResults = apps
                }
                .withError {
                    self.viewInput?.showError(error: $0)
                }
        }
    }

    private func openAppDetails(with app: ITunesApp) {
        let appDetaillViewController = AppDetailViewController(app: app)

self.viewInput?.navigationController?.pushViewController(appDetaillViewControlle
r, animated: true)
    }
}

// MARK: - SearchViewOutput
extension SearchPresenter: SearchViewOutput {

    func viewDidSearch(with query: String) {
        self.viewInput?.throbber(show: true)
        self.requestApps(with: query)
    }

    func viewDidSelectApp(_ app: ITunesApp) {
        self.openAppDetails(with: app)
    }
}
```

Перейдем в **SearchViewController**. Добавим ему свойство **presenter** (еще можно назвать **output**, или **viewOutput**, но в любом случае должен быть указан протокол, а не конкретный класс) и потребуем инициализировать вью-контроллер с объектом презентера:

```
private let presenter: SearchViewOutput
init(presenter: SearchViewOutput) {
    self.presenter = presenter
    super.init(nibName: nil, bundle: nil)
}
```

Уберем код, который мы перенесли в презентер:

- Удалим метод **requestApps(with query: String)**;
- В реализации **tableView(_ tableView: UITableView, didSelectRowAt indexPath: IndexPath)** заменим строчки по созданию и открытию контроллера деталей приложения на передачу ответственности презентеру **self.presenter.viewDidSelectApp(app)**;
- По нажатию юзером кнопки «Найти» мы вместо вызова **self.requestApps(with: query)** (который мы удалили) передаем управление презентеру: **self.presenter.viewDidSearch(with: query)**.

Добавим реализацию протокола **SearchViewInput** (методы у нас уже есть, просто нужно перенести их в одно место и сделать неприватными):

```
// MARK: - Input
extension SearchViewController: SearchViewInput {

    func showError(error: Error) {
        let alert = UIAlertController(title: "Error", message:
"\(error.localizedDescription)", preferredStyle: .alert)
        let actionOk = UIAlertAction(title: "OK", style: .cancel, handler: nil)
        alert.addAction(actionOk)
        self.present(alert, animated: true, completion: nil)
    }

    func showNoResults() {
        self.emptyResultView.isHidden = false
        self.searchResults = []
        self.tableView.reloadData()
    }

    func hideNoResults() {
        self.emptyResultView.isHidden = true
    }

    func throbber(show: Bool) {
        UIApplication.shared.isNetworkActivityIndicatorVisible = show
    }
}
```

И, наконец, у свойства **var searchResults: [ITunesApp]** добавим обсервер, который будет релоадить таблицу:

```
var searchResults = [ITunesApp]() {
    didSet {
        self.tableView.isHidden = false
        self.tableView.reloadData()
        self.searchBar.resignFirstResponder()
    }
}
```

*Примечание: слой View знает о модели ITunesApp. Это не очень здорово, но мы осознанно сделали такое допущение. Во-первых, потому что так проще и быстрее. Во-вторых, эта модель по сути является просто контейнером данных и не содержит логики — значит, мы не сильно нарушаем абстракцию, работая с моделью прямо в слое View. То есть такое допущение оправдано для простого приложения вроде нашего.*

И теперь перед нами встает еще одна (последняя на этом уроке) проблема. Для того чтобы собрать MVP-модуль, надо:

1. Создать объект презентера.
2. Создать объект вью-контроллера, передав ему при инициализации объект презентера.
3. У презентера выставить в его свойство **viewInput** только что созданный вью-контроллер.

Только после этого MVP-модуль будет работать как надо. А создание такого модуля похоже на еще одну ответственность. Кто ее возьмет? Уж точно не вью-контроллер — мы договорились, что он занимается только UI. В принципе, все эти три пункта можно сделать в презентере другого модуля. Но это не очень удобно, если представить, что нам нужно показывать один и тот же MVP-модуль из разных точек приложения, — тогда придется везде писать одинаковый код. Для решения этой проблемы идеально подходит паттерн **Builder**. Мы создадим еще один класс, который будет принадлежать тому же MVP-модулю и заниматься его сборкой и настройкой:

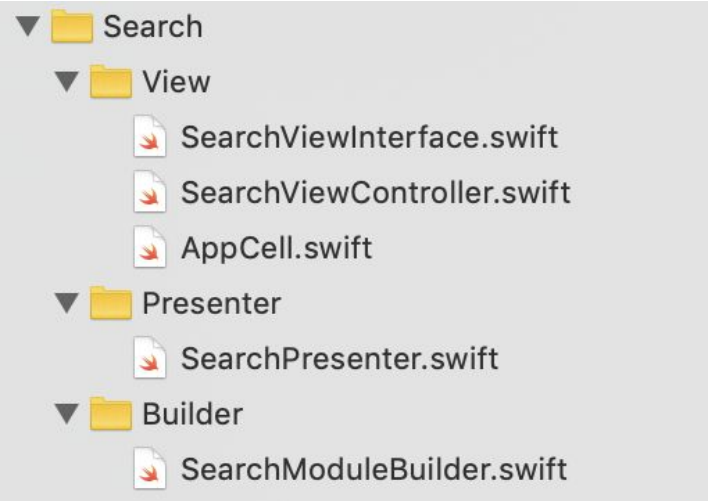
```
final class SearchModuleBuilder {

    static func build() -> (UIViewController & SearchViewInput) {
        let presenter = SearchPresenter()
        let viewController = SearchViewController(presenter: presenter)
        presenter.viewInput = viewController
        return viewController
    }
}
```

Вью-контроллер **SearchViewController()** раньше создавался и презентился в классе **AppStartManager**. Перейдем туда. Теперь вместо инициализации отдельно вью-контроллера необходимо обращаться к билдеру:

```
let rootVC = SearchModuleBuilder.build()
```

У нас получилась следующая структура MVP-модуля **Search**:




Вспомните, что раньше в папке **Search** было всего два класса — ячейка и вью-контроллер, а теперь сущностей намного больше. Такова плата за те плюсы, которые дает нам архитектура с более разделенными ответственностями. С другой стороны, поскольку мы использовали общий подход, который будет работать и для остальных модулей приложения, это не выглядит страшным — ведь везде у вас будет примерно одинаковая архитектура, в которой легко разобраться.

Обсудим теперь плюсы и минусы архитектуры MVP.

# Плюсы MVP

MVP — это реализация всех обещаний MVC о полной разделенности слоя **View** и слоя **Controller**. Это именно та цель, которой добивается MVP, преобразуя стандартную концепцию MVC в iOS. Все плюсы архитектуры MVP следуют именно из этой разделенности компонентов:

1. **Лучшая разделенность ответственностей и как следствие — читаемость кода.**

   В Presenter нет кода, ответственного за layout, а в View (то есть **UIViewController** и **UIView**) нет кода, ответственного за бизнес-логику и навигацию.

2. **Возможность unit-тестирования.**

   Бизнес-логику Presenter'а можно протестировать отдельно от View, просто заменив ее mock-объектом.

3. Переиспользуемость компонентов.

   В теории для одного и того же **Presenter** можно создать совершенно разные View, ведь они независимы друг от друга. На практике такая задача может возникнуть, например, если экран приложения находится в A/B-тесте с двумя разными UI. Иногда нужно поддерживать несколько таргетов в одном проекте: каждый таргет имеет свой UI, но бизнес-логика у приложений одинаковая.

# Проблемы MVP

Первая очевидная проблема — теперь придется писать больше кода. Раньше у нас был один класс для всего — **UIViewController** (кстати, вспомните про анти-паттерн God object). Теперь это как минимум два класса — **UIViewController** и **Presenter**, причем между ними надо проставить

зависимости, то есть этот единый модуль из разных слоев нужно еще собрать. И для этого применить один из подходов, который почти наверняка приведет к созданию еще большего количества классов.

Главное неудобство по сравнению с одним **UIViewController** заключается в том, что теперь необходимо руками писать взаимодействие между View и Presenter, прокидывая вызовы методов либо через делегат, либо с помощью замыканий.

Есть проблемы, которые MVP все еще решить не в силах. Дело в том, что ответственность Presenter'а все еще достаточно широка. Это отлично, что он не завязан на View, но он все еще занимается и бизнес-логикой, и работой с core-слоями приложения, и зачастую навигацией. Так что **Presenter** все еще может оставаться большим и запутанным классом. Рекомендации для исправления проблемы **Massive View Controller** из раздела «Проблемы MVC» будут работать и здесь. Также одним из решений этой проблемы может быть переход к еще более дробной архитектуре — **VIPER**, которую мы рассмотрим на следующем уроке.

# Практическое задание

1. В проекте из этого урока продолжите наполнение экрана деталей приложения. Создайте сабмодуль с информацией о последней версии приложения («Что нового»). Для этого необходимо дополнить модель данных теми данными, которые приходят с сервера, но пока не парсятся. Отображайте (как и в самом **appStore**) заголовок, номер версии, ее описание и дату выхода.

2. * Добавьте сабмодуль со скриншотами. Ссылки на скриншоты уже есть в модели. Все скриншоты расположите в **collection view**.

   ** По тапу на скриншот модально открывайте новый экран с большими скриншотами, как у **appStore**. Его можно реализовать на **UIPageViewController** либо **UICollectionView**.

3. Добавьте новый экран поиска — поиск по песням в iTunes. Модель песни уже есть (при необходимости дополняйте ее данными с сервера, которые пока не декодятся), сервис также умеет запрашивать песни. Нужно сделать экран поиска по песням на архитектуре MVP. Сделайте экран структурно похожим на экран поиска по приложениям, можете добавить свои идеи в UI. Ячейку песни сделайте с другим лэйаутом — добавьте на нее, например, иконку ноты и т. п.

# 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.
5. Child View Controllers.

# 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».