



- этот класс — наследник UIViewController;



- данные сохраняются прямо в UIViewController;
- UIView-подклассы ни за что не отвечают;
- Model — это просто контейнер для данных;
- вы не делаете юнит-тесты.

И это может произойти, даже если вы следуете рекомендациям Apple и реализуете их паттерн [Cocoa MVC](#), так что не огорчайтесь. С «яблочным» [MVC](#) не все в порядке, но мы вернемся к нему позже.

А сейчас давайте определим **признаки** хорошей архитектуры:

- сбалансированное **распределение** обязанностей между сущностями с жесткими ролями;
- **тестируемость**. Обычно вытекает из первого признака (без паники, это легкоосуществимо при соответствующей архитектуре);
- **простота использования** и низкая стоимость обслуживания.

### Почему распределение?

Распределение уменьшает нагрузку на мозг, когда мы пытаемся выяснить, как работает та или иная сущность. Если вы думаете, что чем больше развиваетесь, то тем лучше мозг будет адаптироваться к пониманию сложных концепций — вы правы. Но у всего есть предел, и он достигается довольно быстро. Таким образом, самый простой способ уменьшить сложность заключается в разделении обязанностей между несколькими сущностями по [принципу единой ответственности](#).

### Почему тестируемость?

Тестируемость архитектуры определяет, насколько легко нам будет писать юнит-тесты, а чаще всего — сможем ли мы их писать в принципе. А стоит ли тестировать вообще? Как правило, это не вопрос для тех, у кого **завалились** юнит-тесты после добавления нового функционала или после рефакторинга каких-то тонкостей класса. Это означает, что тесты **уберегли** разработчиков от обнаружения проблемы в «рантайме». Что могло бы произойти с приложением уже на устройстве пользователей, а исправление было бы возможно только [через неделю](#).

### Почему простота использования?

Тут все понятно, но стоит отметить, что лучший код — это код, который никогда не был написан. И чем меньше у вас кода, тем меньше ошибок. Поэтому желание писать меньше кода вовсе не говорит о том, что разработчик ленился. А выбирая самое умное решение, нужно всегда учитывать стоимость его поддержки.

## Основы MV(X)

Сегодня у нас есть много вариантов архитектурных паттернов проектирования:

- [MVC](#);
- [MVP](#);
- [MVVM](#);
- [VIPER](#).

Первые три из них предполагают назначение сущностей приложения в одну из 3 категорий:

- **Models** — ответственные за данные домена или слой доступа к данным, который манипулирует данными, например, класс **Person** или **PersonDataProvider**;
- **Views** — ответственные за уровень представления (**GUI**); для окружающей среды iOS это все, что начинается с префикса **UI**;
- **Controller / Presenter / ViewModel** — посредник между **Model** и **View**; в целом отвечает за изменения **Model**, реагируя на действия пользователя, выполненные на **View**, и обновляет **View**, используя изменения из **Model**.

Имея разделенные сущности, мы можем:

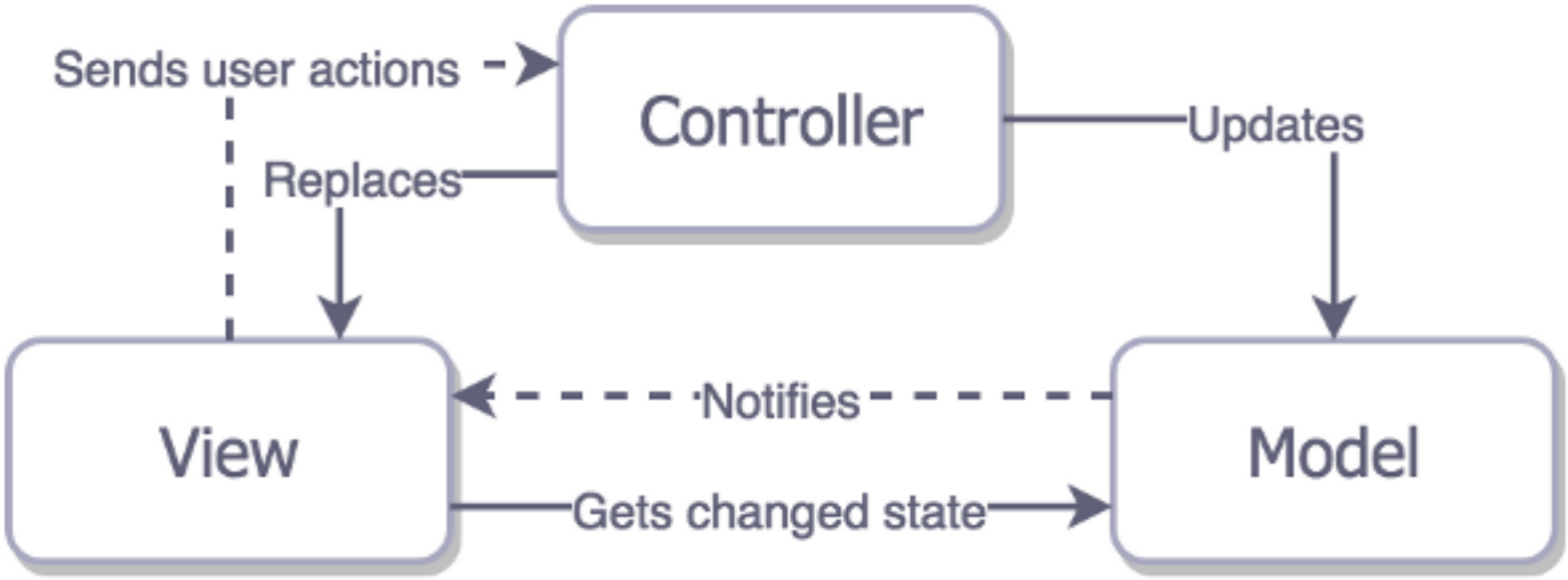
- лучше понимать их;
- повторно их использовать (в основном применимо к **View** и **Model**);
- тестировать их отдельно друг от друга.

*Давайте начнем с MV(X) паттернов и позже вернемся к VIPER.*

## MVC

### Как было раньше

Прежде чем обсуждать видение MVC компанией Apple, давайте посмотрим на [традиционную версию](#).

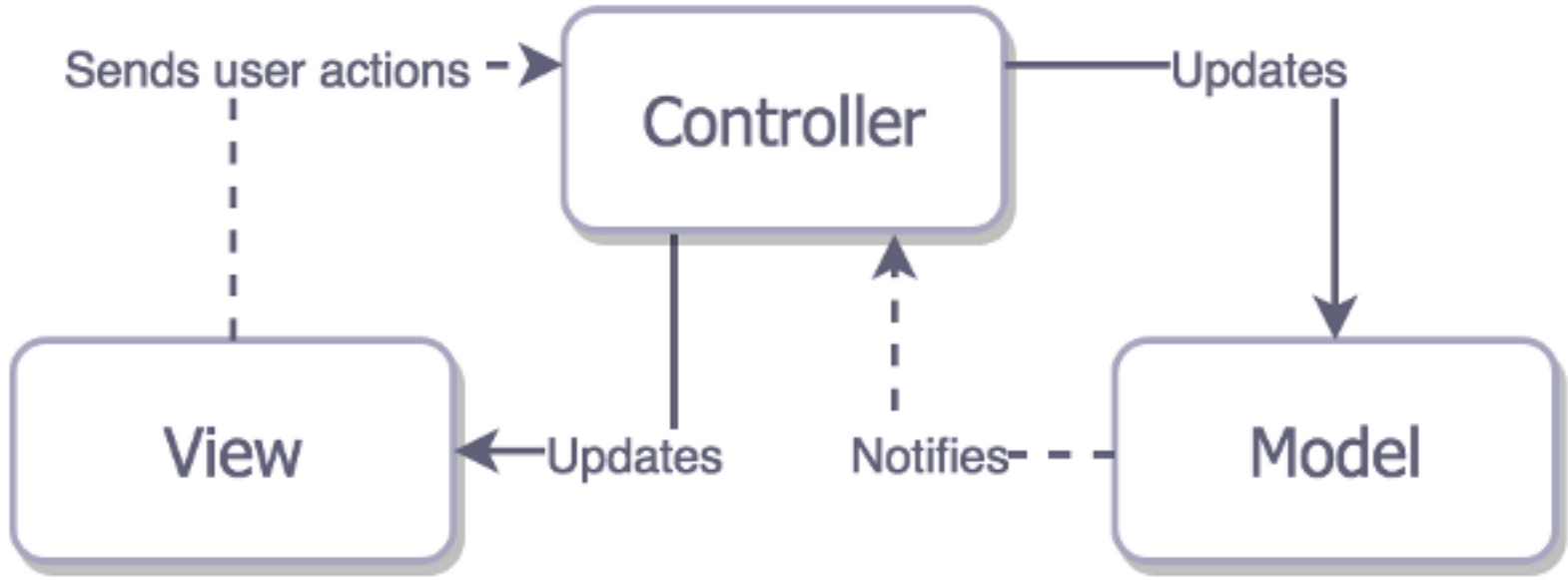


В традиционном MVC **View** не хранит состояния в себе. **Controller** просто «рендерит» **View** при изменениях **Model**. Например, веб-страница полностью перегружается после того, как вы нажмете на ссылку для перехода в другое место. Хотя можно реализовать традиционный MVC в среде iOS, это не имеет особого смысла из-за архитектурной проблемы: все три сущности тесно связаны, каждая сущность **знает** о двух других. Это сильно снижает возможность повторного использования каждого из элементов. По этой причине мы не будем даже пытаться написать пример канонического MVC.

Традиционный MVC кажется неприменимым к современной iOS разработке.

## MVC от Apple

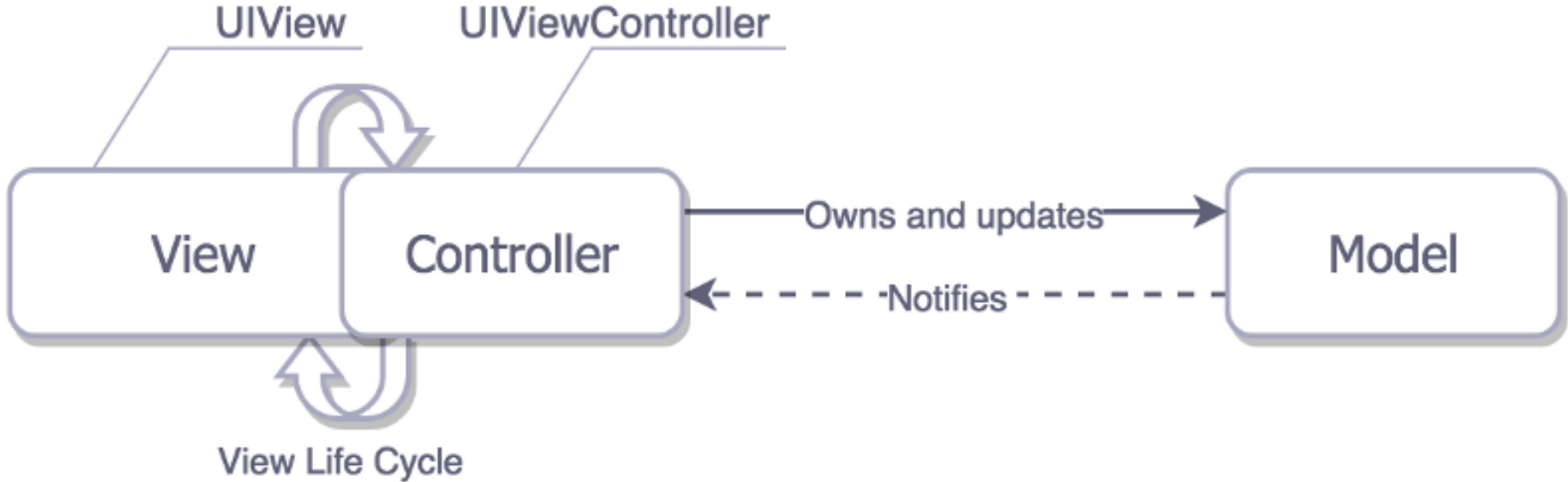
### Ожидания



**Controller** является посредником между **View** и **Model**, следовательно, две последних не знают о существовании друг друга. Поэтому **Controller** трудно повторно использовать, но это, в принципе, нас устраивает, так как мы должны иметь место для той хитрой бизнес-логики, которая не вписывается в **Model**.

В теории все выглядит очень просто, но вы чувствуете, что что-то не так, верно? Вы наверняка слышали, что люди расшифровывают MVC как **Massive View Controller**. Кроме того, [разгрузка ViewController](#) стала важной темой для iOS-разработчиков. Почему это происходит, если в Apple просто взяли традиционный MVC и немного его улучшили?

### Реальность



Cocoa MVC поощряет вас писать **Massive** View Controller, потому что контроллер настолько вовлечен в жизненный цикл **View**, что трудно сказать, что он является отдельной сущностью. Хотя у вас все еще есть возможность отгрузить часть бизнес-логики и преобразования данных в **Model**, когда дело доходит до отгрузки работы во **View**, у вас не так много вариантов. В большинстве случаев вся ответственность **View** состоит в том, чтобы отправить действия к контроллеру. В итоге все заканчивается тем, что View Controller становится делегатом и источником данных, а также местом запуска и отмены серверных запросов и, в общем-то, всего чего угодно.

Сколько раз вы видели такой код:

```
var userCell = tableView.dequeueReusableCellWithIdentifier("identifier") as UserCell
userCell.configureWithUser(user)
```

**View**-ячейка конфигурируется непосредственно с **Model**. Таким образом нарушаются принципы MVC, но такой код можно увидеть очень часто, и, как правило, люди не понимают, что это неправильно. Если вы строго следуете MVC, то должны настраивать ячейку внутри контроллера и не передавать **Model** во View, что увеличит **Controller** еще больше.

Cocoa MVC обосновано расшифровывают как Massive View Controller.

Проблема не очевидна, пока дело не доходит до [юнит-тестов](#) (надеюсь, что в вашем проекте оно все же доходит). Так как View Controller тесно связана с **View**, ее становится трудно тестировать, и приходится идти изощренным путем, заменяя **View** [Mock-объектами](#) и имитируя их жизненный цикл, а также писать код View Controller таким образом, чтобы бизнес-логика была по максимуму отделена от кода view layout.

Давайте посмотрим на простой пример из «плейграунда»:

```
import UIKit

struct Person { // Model
    let firstName: String
    let lastName: String
}

class GreetingViewController : UIViewController { // View + Controller
    var person: Person!
    let showGreetingButton = UIButton()
    let greetingLabel = UILabel()

    override func viewDidLoad() {
        super.viewDidLoad()
        self.showGreetingButton.addTarget(self, action: "didTapButton:", forControlEvents: .TouchUpInside)
    }

    func didTapButton(button: UIButton) {
        let greeting = "Hello" + " " + self.person.firstName + " " + self.person.lastName
        self.greetingLabel.text = greeting
    }

    // layout code goes here
}

// Assembling of MVC
let model = Person(firstName: "David", lastName: "Blaine")
let view = GreetingViewController()
view.person = model;
```

Сборка MVC может быть выполнена в «презентирующей» View Controller.

Кажется, это сложно протестировать, не так ли? Мы можем выделить генерацию приветствия в новый класс *GreetingModel* и тестировать ее отдельно, но мы не можем протестировать логику представления (хоть в примере ее не так много) внутри *GreetingViewController* без вызова методов жизненного цикла **View** напрямую (*viewDidLoad*, *didTapButton*), что может привести к загрузке всех UIView, и это плохо для юнит-тестов.

На самом деле, тестирование UIViews на одном симуляторе (например, iPhone 4S) не гарантирует, что он будет работать нормально на других устройствах (например, iPad), так что я рекомендую убрать галочку *Host Application* из конфигурации таргета юнит-тестов и запускать их на симуляторе, не включая само приложение.

Взаимодействие между **View** и **Controller** на самом деле *не особо поддается тестированию с помощью юнит-тестов*.

После всего сказанного может показаться что, Сосоа MVC является довольно плохим выбором паттерна. Но давайте оценим его с точки зрения **признаков хорошей архитектуры**, определенных в начале статьи:

- **распределение**: **View** и **Model** на самом деле разделены, но **View** и **Controller** тесно связаны;
- **тестируемость**: из-за плохого распределения вы, вероятно, будете тестировать только **Model**;
- **простота использования**: наименьшее количество кода среди других паттернов. К тому же он выглядит понятным, поэтому его легко может поддерживать даже неопытный разработчик.

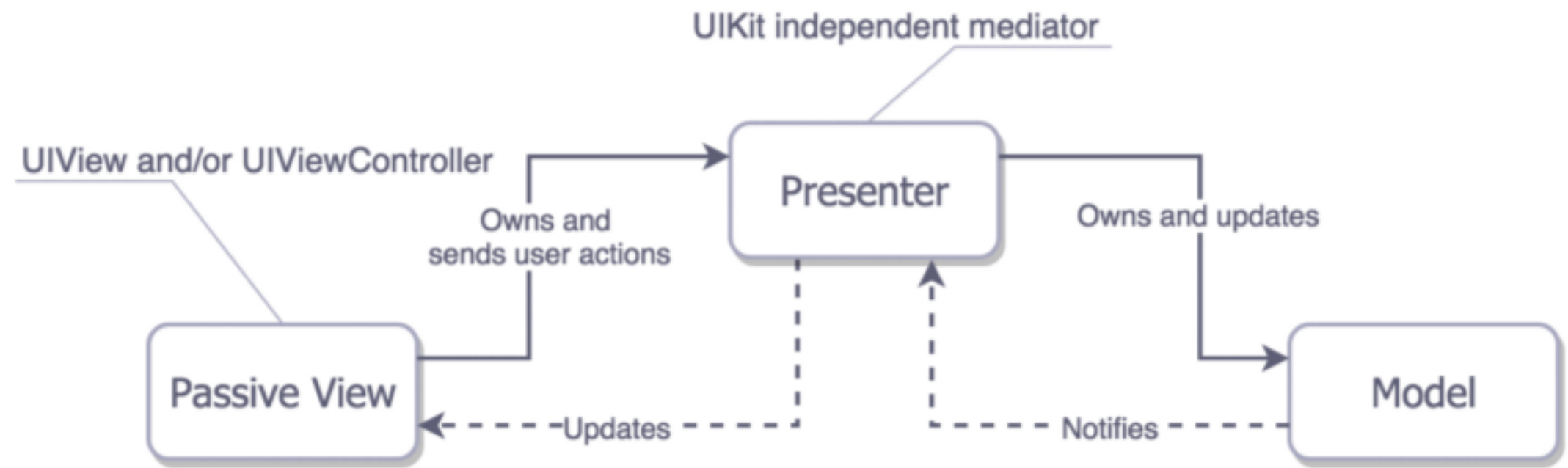
Сосоа MVC — это разумный выбор, если вы не готовы инвестировать много времени в свою архитектуру и чувствуете, что паттерн с более высокой стоимостью обслуживания вашему небольшому проекту или стартапу будет не по карману.

Сосоа MVC является лучшим архитектурным паттерном с точки зрения скорости разработки.

## MVP

### Реализация обещаний Сосоа MVC





Разве это не похоже на «яблочный» MVC? На самом деле — очень, а имя ему — [MVP](#) (вариант с пассивной **View**). Но означает ли это, что MVC от Apple на самом деле является MVP? Нет, не является, потому что, как вы помните, там **View** и **Controller** тесно связаны, в то время как посредник в MVP — **Presenter** — не имеет отношения к жизненному циклу View Controller. **View** может быть легко заменена [Mock-объектами](#), поэтому в **Presenter** нет layout-кода, но он отвечает за обновление **View** в соответствии с новыми данными и состоянием.



— А что если я скажу тебе, что **UIViewController** — это **View**.

С точки зрения MVP, подклассы UIViewController на самом деле есть **View**, а не **Presenter**. Это различие обеспечивает превосходную тестируемость, которая идет за счет скорости разработки, потому что вы должны связывать вручную данные и события именно между **View** и **Presenter**, как можно увидеть на примере ниже.

```
import UIKit

struct Person { // Model
    let firstName: String
    let lastName: String
}

protocol GreetingView: class {
    func setGreeting(greeting: String)
}

protocol GreetingViewPresenter {
    init(view: GreetingView, person: Person)
    func showGreeting()
}

class GreetingPresenter : GreetingViewPresenter {
```

```
unowned let view: GreetingView
let person: Person
required init(view: GreetingView, person: Person) {
    self.view = view
    self.person = person
}
func showGreeting() {
    let greeting = "Hello" + " " + self.person.firstName + " " + self.person.lastName
    self.view.setGreeting(greeting)
}
}

class GreetingViewController : UIViewController, GreetingView {
    var presenter: GreetingViewPresenter!
    let showGreetingButton = UIButton()
    let greetingLabel = UILabel()

    override func viewDidLoad() {
        super.viewDidLoad()
        self.showGreetingButton.addTarget(self, action: "didTapButton:", forControlEvents: .TouchUpInside)
    }

    func didTapButton(button: UIButton) {
        self.presenter.showGreeting()
    }

    func setGreeting(greeting: String) {
        self.greetingLabel.text = greeting
    }

    // layout code goes here
}
// Assembling of MVP
let model = Person(firstName: "David", lastName: "Blaine")
let view = GreetingViewController()
let presenter = GreetingPresenter(view: view, person: model)
view.presenter = presenter
```

Важное примечание относительно сборки

MVP является первым паттерном, выявляющим проблему сборки, которая происходит из-за наличия трех *действительно* отдельных слоев. Так как нам не нужно, чтобы **View** знала о **Model**, выполнять сборку в презентующей View Controller (который на самом деле **View**) неправильно, следовательно, это нужно сделать в другом месте. Например, можно создать сервис **Router**, который будет отвечать за выполнение сборки и презентацию **View-to-View**. Эта проблема возникает не только в MVP, ее также нужно решать во **всех последующих паттернах**.

Давайте посмотрим на **признаки хорошей архитектуры** для MVP:

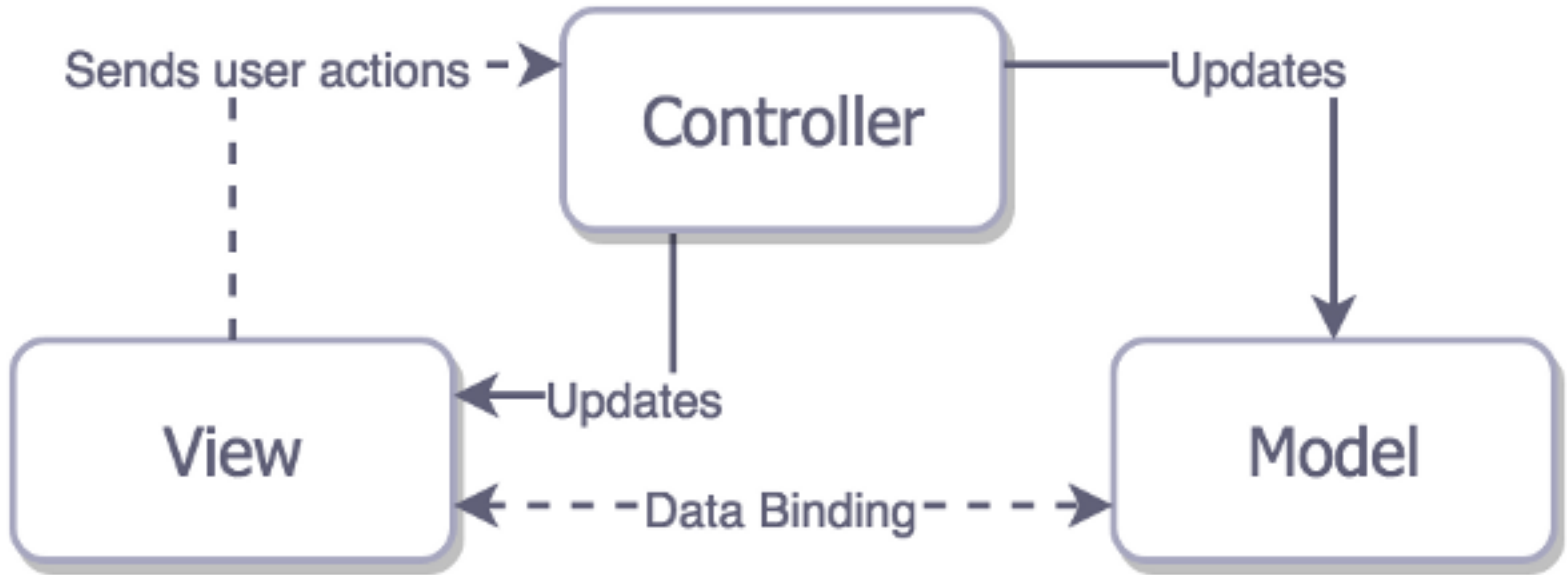
- **распределение**: большая часть ответственности разделена между **Presenter** и **Model**, а **View** ничего не делает;
- **тестируемость**: отличная, мы можем проверить большую часть бизнес-логики благодаря бездействию View;
- **простота использования**: в нашем нереально простом примере количество кода в два раза больше по сравнению с MVC, но в то же время идея MVP очень проста.

MVP в iOS означает превосходную тестируемость и много кода.

MVP

С «блек-джеком» и «биндингами»

Существует другой вариант MVP — MVP с надзирающим контроллером. Он включает в себя прямое связывание **View** и **Model**, в то время как **Presenter** (надзирающий контроллер) по-прежнему обрабатывает действия с **View** и способен изменять ее.



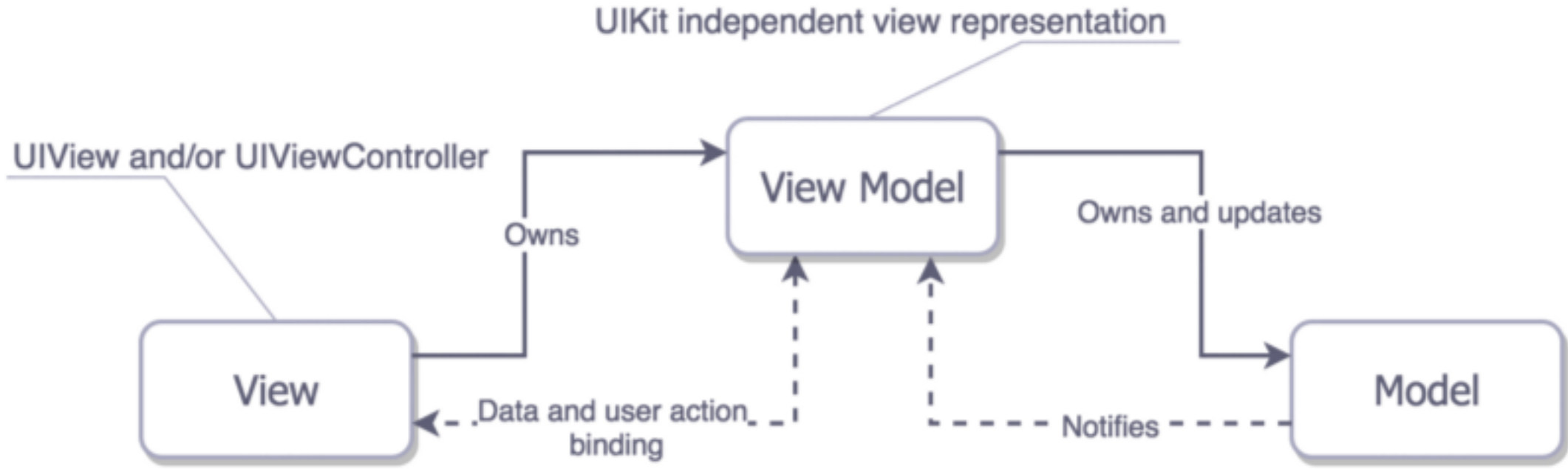
Но, как мы уже узнали ранее, расплывчатое разделение ответственности плохо само по себе, равно как и тесная связанность между **View** и **Model**. А я не вижу смысла в написании примера для плохой архитектуры.

## MVVM

Самая новая из MV(X) вида.

**MVVM** является новейшим из MV(X) паттернов, так что будем надеяться, что он появился с учетом всех проблем, присущих MV(X).

В теории Model-View-ViewModel выглядит очень хорошо. **View** и **Model** уже нам знакомы, как и **View Model** в качестве посредника.



Он очень похож на MVP:

- MVVM рассматривает View Controller как **View**;
- в нем нет тесной связи между **View** и **Model**.

Кроме того, он делает биндинг как надзирающая версия MVP, но не между **View** и **Model**, а между **View** и **View Model**.

Так что такое **View Model** в среде iOS? Это **независимое** от UIKit представление **View** и ее состояния. **View Model** вызывает изменения в **Model** и самостоятельно обновляется с уже обновленной **Model**. И так как биндинг происходит между **View** и **View Model**, то первая, соответственно, тоже обновляется.

### Биндинги

Я упоминаю их, начиная с части про MVP, но давайте познакомимся с ними поближе. Биндинги доступны «из коробки» для разработки OS X, но их нет в арсенале iOS-разработчика. Конечно, у нас есть KVO и Notifications, но они не такие удобные, как биндинги.

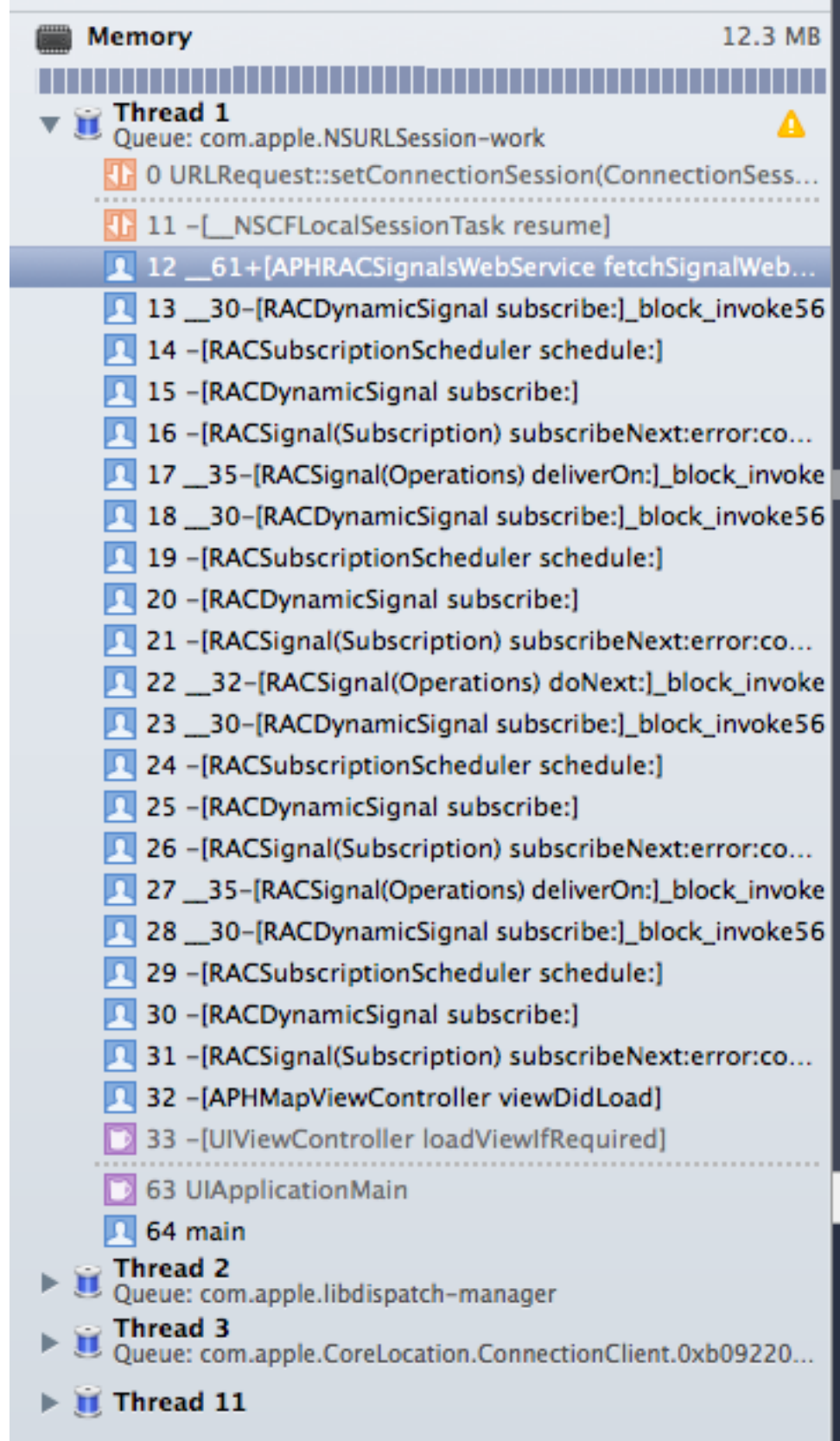
Поэтому, при условии что мы не хотим писать их сами, можно выбрать:

- одну из биндинг-библиотек, основанных на KVO (например, [RZDataBinding](#) или [SwiftBond](#));
- полноразмерный фреймворк для **функционального реактивного программирования**, такой как [ReactiveCocoa](#), [RxSwift](#) или [PromiseKit](#).

Сегодня, на самом деле, когда вы слышите MVVM, вы думаете о ReactiveCocoa, и наоборот. Хотя и можно делать MVVM с простыми биндингами, ReactiveCocoa (или его одноклассники) позволит вам выжать все из MVVM-паттерна.

Есть одна горькая правда об FRP-фреймворках: великая сила приходит с большой ответственностью. Очень легко все поломать, когда вы пишете *реактивно*. Другими словами, если что-то пошло не так, вы можете потратить много времени на отладку приложения. Стоит просто взглянуть на этот стек вызовов.





В нашем простом примере реактивный фреймворк или даже KVO является излишним. Мы явно попросим **View Model**, чтобы она обновилась с помощью метода *showGreeting*, и используем простое свойство для функции «колбека» *greetingDidChange*, чтобы узнать об изменениях.

```
import UIKit

struct Person { // Model
    let firstName: String
    let lastName: String
}

protocol GreetingViewModelProtocol: class {
    var greeting: String? { get }
    var greetingDidChange: ((GreetingViewModelProtocol) -> ())? { get set } // function to call when greeting did change
    init(person: Person)
    func showGreeting()
}

class GreetingViewModel : GreetingViewModelProtocol {
    let person: Person
    var greeting: String? {
        didSet {
            self.greetingDidChange?(self)
        }
    }
    var greetingDidChange: ((GreetingViewModelProtocol) -> ())?
    required init(person: Person) {
        self.person = person
    }
    func showGreeting() {
        self.greeting = "Hello" + " " + self.person.firstName + " " + self.person.lastName
    }
}

class GreetingViewController : UIViewController {
    var viewModel: GreetingViewModelProtocol! {
        didSet {
            self.viewModel.greetingDidChange = { [unowned self] viewModel in
                self.greetingLabel.text = viewModel.greeting
            }
        }
    }
}
```



```
    }
}

let showGreetingButton = UIButton()
let greetingLabel = UILabel()

override func viewDidLoad() {
    super.viewDidLoad()
    self.showGreetingButton.addTarget(self.viewModel, action: "showGreeting", forControlEvents: .TouchUpInside)
}

// layout code goes here
}

// Assembling of MVVM
let model = Person(firstName: "David", lastName: "Blaine")
let viewModel = GreetingViewModel(person: model)
let view = GreetingViewController()
view.viewModel = viewModel
```

И снова вернемся к нашей оценке **признаков хорошей архитектуры**:

- **распределение**: из нашего крошечного примера это неясно, но на самом деле в MVVM **View** имеет больше обязанностей, чем **View** из MVP. Потому что первая обновляет свое состояние с View Model за счет установки биндингов, тогда как вторая направляет все события в Presenter и не обновляет себя (это делает Presenter);
- **тестируемость**: View Model не знает ничего о представлении, это позволяет нам с легкостью тестировать ее. View также можно тестировать, но так как она зависит от UIKit, вы можете просто пропустить это;
- **простота использования**: тот же объем кода, как в нашем примере MVP, но в реальном приложении, где вам придется направить все события из View в Presenter и обновлять View вручную, MVVM будет гораздо стройнее (если вы используете биндинги).

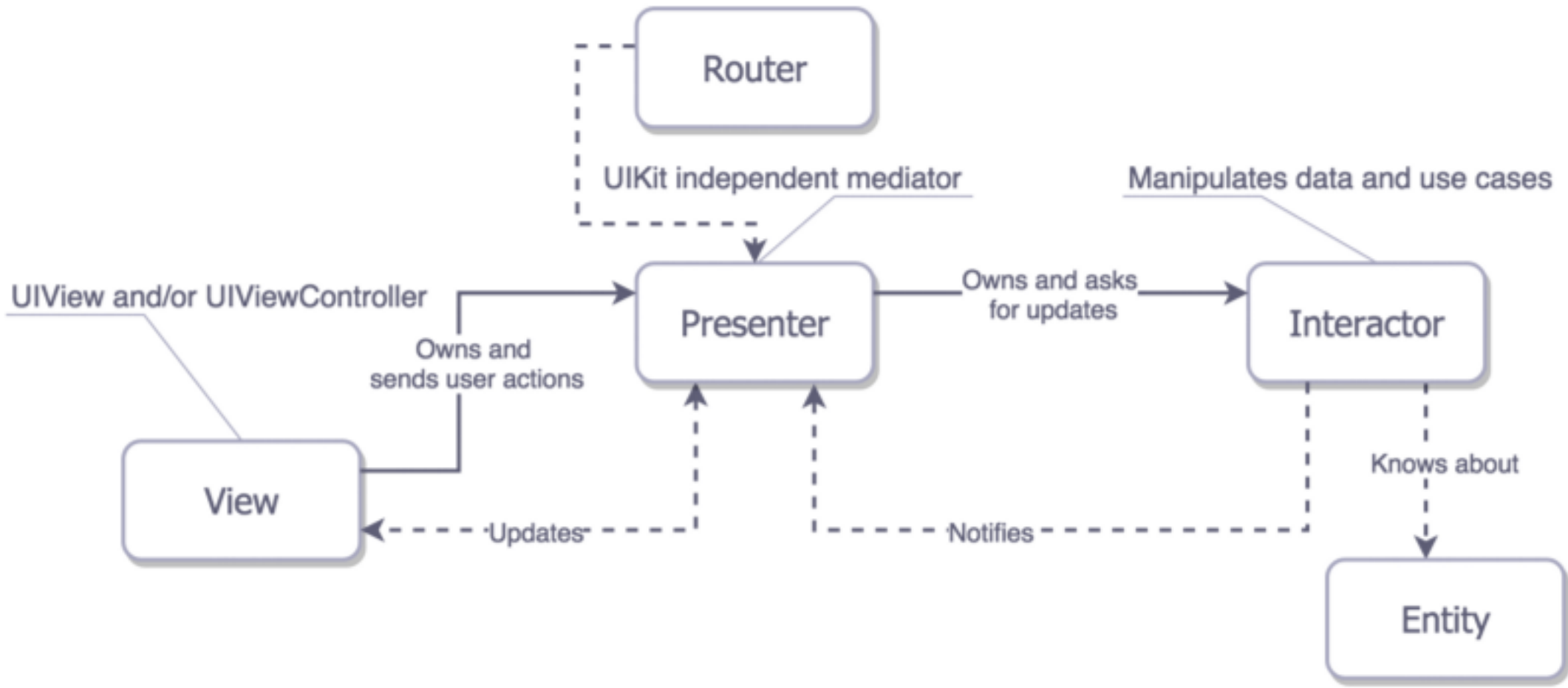
MVVM является очень привлекательным паттерном, так как он сочетает в себе преимущества вышеупомянутых подходов и не требует дополнительного кода для обновления View в связи с биндингами на стороне View. Тем не менее, тестируемость все еще находится на хорошем уровне.

## VIPER

### Опыт строительства из кубиков Lego, перенесённый на проектирование iOS-приложений

VIPER — наш последний кандидат, который особенно интересен, потому что он не из категории MV(X).

К настоящему моменту вы уже должны согласиться с тем, что разделение обязанностей — это очень хорошо. VIPER делает еще один шаг в сторону разделения обязанностей и вместо привычных трех слоев предлагает **пять**.



- **Interactor** содержит бизнес-логику, связанную с данными (**Entities**): например, создание новых экземпляров сущностей или получение их с сервера. Для этих целей вы будете использовать некоторые Сервисы и Менеджеры, которые рассматриваются скорее как внешние зависимости, а не как часть модуля VIPER.
- **Presenter** содержит бизнес-логику, связанную с UI (но UIKit-независимую), вызывает методы в **Interactor**.
- **Entities** — простые объекты данных, не являются слоем доступа к данным, потому что это ответственность слоя **Interactor**.
- **Router** несет ответственность за переходы между VIPER-модулями.

В принципе, модулем VIPER может быть один экран или целая user story вашего приложения (например, аутентификация может быть на один экран или несколько связанных экранов). Вам решать, насколько маленькими будут ваши «лего-блоки».

Если мы сравним VIPER с паттернами MV(X)-вида, то увидим несколько отличий в распределении обязанностей:

- логика из **Model** (взаимодействие данных) смещается в **Interactor**, а также есть **Entities** — структуры данных, которые ничего не делают;
- из **Controller**, **Presenter**, **ViewModel** обязанности представления UI переехали в **Presenter**, но без возможности изменения данных;
- **VIPER** является первым шаблоном, который пробует решить проблему навигации, для этого есть **Router**.

То, что MV(X)-паттерны не решают проблему маршрутизации, не значит, что она не существует для iOS-приложений.

В примере нет маршрутизации или взаимодействия между модулями, так как эти темы совсем не охвачены MV(X)-паттернами.

```
import UIKit

struct Person { // Entity (usually more complex e.g. NSManagedObject)
    let firstName: String
    let lastName: String
}

struct GreetingData { // Transport data structure (not Entity)
    let greeting: String
    let subject: String
}

protocol GreetingProvider {
    func provideGreetingData()
}

protocol GreetingOutput: class {
    func receiveGreetingData(greetingData: GreetingData)
}

class GreetingInteractor : GreetingProvider {
    weak var output: GreetingOutput!

    func provideGreetingData() {
        let person = Person(firstName: "David", lastName: "Blaine") // usually comes from data access layer
        let subject = person.firstName + " " + person.lastName
        let greeting = GreetingData(greeting: "Hello", subject: subject)
        self.output.receiveGreetingData(greeting)
    }
}

protocol GreetingViewEventHandler {
    func didTapShowGreetingButton()
}

protocol GreetingView: class {
    func setGreeting(greeting: String)
}

class GreetingPresenter : GreetingOutput, GreetingViewEventHandler {
    weak var view: GreetingView!
    var greetingProvider: GreetingProvider!

    func didTapShowGreetingButton() {
        self.greetingProvider.provideGreetingData()
    }

    func receiveGreetingData(greetingData: GreetingData) {
        let greeting = greetingData.greeting + " " + greetingData.subject
        self.view.setGreeting(greeting)
    }
}

class GreetingViewController : UIViewController, GreetingView {
    var eventHandler: GreetingViewEventHandler!
    let showGreetingButton = UIButton()
    let greetingLabel = UILabel()

    override func viewDidLoad() {
        super.viewDidLoad()
        self.showGreetingButton.addTarget(self, action: "didTapButton:", forControlEvents: .TouchUpInside)
    }

    func didTapButton(button: UIButton) {
        self.eventHandler.didTapShowGreetingButton()
    }
}
```



```
func setGreeting(greeting: String) {
    self.greetingLabel.text = greeting
}

// layout code goes here
}

// Assembling of VIPER module, without Router

let view = GreetingViewController()
let presenter = GreetingPresenter()
let interactor = GreetingInteractor()
view.eventHandler = presenter
presenter.view = view
presenter.greetingProvider = interactor
interactor.output = presenter
```

И все же, еще раз вернемся к **признакам**.

- **Распределение.** Несомненно, VIPER является чемпионом в распределении обязанностей.
- **Тестируемость.** Здесь нет ничего удивительного: лучше распределение — лучше тестируемость.
- **Простота использования.** Как вы уже догадались, первые два преимущества идут за счет стоимости сопровождения. Вам придется писать огромное количество интерфейсов для классов с незначительными обязанностями.

## Так что там с Lego?

При использовании VIPER вам может показаться, что вы строите Эмпайр Стейт Билдинг из кубиков Lego, и это говорит о том, что у вас [есть проблемы](#). Может быть, вы слишком рано взялись за VIPER и стоит рассмотреть что-то попроще. Некоторые люди игнорируют это и продолжают стрелять из пушки по воробьям. Я предполагаю, что они верят, что их приложения получают выгоду из VIPER когда-нибудь в будущем, даже если сейчас стоимость обслуживания неоправданно высока. Если вы считаете, что оно того стоит, то я рекомендую вам попробовать [Generamba](#) — инструмент для генерации скелетов VIPER. Хотя лично мне кажется, что это сродни использованию *автоматического прицела* для стрельбы из той же пушки вместо *рогатки*.

## Вывод

Мы рассмотрели несколько архитектурных паттернов, и я надеюсь, что вы нашли ответы на некоторые свои вопросы. Я не сомневаюсь, что вы поняли, что **не существует «серебряной пули»** среди паттернов, а выбор архитектуры является вопросом взвешивания компромиссов в вашей конкретной ситуации.

Мне кажется вполне естественным сочетать нескольких архитектур в одном приложении. Например, вы начали с MVC, но поняв, что какой-то конкретный экран (use case) стало слишком трудно поддерживать с MVC, перешли к MVVM, но только для этого конкретного экрана. Потому что на самом деле нет необходимости рефакторить другие экраны, для которых MVC работают отлично, тем более что обе архитектуры легко совместимы.

Сделай настолько просто, насколько это возможно, но не проще. (с)Альберт Эйнштейн

Английская версия доступна [здесь](#). Слайды которые я презентовал на NSLondon доступны [здесь](#).

Богдан Орлов,  
iOS разработчик в Badoo

ios, architecture, patterns, objective-c, swift, mvc, mvvm, mvp, viper

↑ +28 ↓

👁 48,5k ⭐ 298

🐦

📧

📘

📺

👤

Автор: @KAIHAJIOT

b

рейтинг

**Badoo** 395,18

Сайт

Подписаться

## ПОХОЖИЕ ПУБЛИКАЦИИ

4 марта 2016 в 14:59

## История создания Chatto

 +25
  11,8k
  59
  10

11 июля 2013 в 16:02

## Система офлайн-уведомлений Badoo

↑ +52    👁 32,5k    ★ 89    💬 8

ВАКАНСИИ КОМПАНИИ <b>Badoo</b>	Мой круг
Senior PHP/MySQL Developer	от 180 000 до 250 000 руб.
Москва • Полный рабочий день	
iOS Developer (Relocate to London)	
Москва • Полный рабочий день	
Android Developer (Relocate to London)	
Москва • Полный рабочий день	
Вакансии компании	Создать резюме

Комментарии (14)

отслеживать новые: ☐ в почте ☐ в трекере



lookman

7 апреля 2016 в 18:56

#

★

+1

↑

Спасибо, интересный материал.

[ответить](#)



zm\_lill

7 апреля 2016 в 22:09

#

★

0

↑

Говоря про **VIPER** — мне кажется, многие уже реализуют подобные архитектурные решения, и не только в iOS, при этом продолжая называть это **MVC**.

[ответить](#)



agee

9 апреля 2016 в 14:35

#

★

0

↑

Богдан, спасибо за материал. Он вполне неплох, как ориентир. Однако примеры, приведенные здесь, к сожалению, нерелевантны. Усложнение (или просто изменение) архитектуры всегда должно быть направлено на решение проблем, а не прои

[ответить](#)



agee

9 апреля 2016 в 14:50

#

★

↩


↑

0

↑

Извините, к сожалению, не очень удобно ведет себя моб. приложение хабра, случайно отправил незаконченное сообщение.  
В общем, пример, приведенный вами, слишком мелковат для того, чтобы увидеть реальную проблему, и не понятно, что конкретно решают переходы на ту или иную архитектурную "парадигму".

[ответить](#)



KAIPIAJIOT

12 апреля 2016 в 12:57

#

★

↩

↑

+1

↑

Примеры, на самом деле, очень маленькие, и проблему в них никак не увидеть. А статья, все-таки, обзорная, и полезная, в первую очередь, тем, кто сам заметит проблемы с Епловским MVC и захочет попробовать использовать что-то другое.

[ответить](#)



CRivaldo

10 апреля 2016 в 21:17

#


★

+1

↑

Есть какая-нибудь литература по паттернам, применимым к iOS? Не так давно занимаюсь разработкой под эту платформу. Apple'овская документация мне понравилась больше, чем «iOS Programming. Big Nerd Ranch». Сейчас читаю «Effective Objective C 2.0». Мне понравилась статья, хотел бы больше узнать по данной теме.

[ответить](#)



KAIPIAJIOT

12 апреля 2016 в 12:58

#

★

↩

↑

0

↑

Есть такая [книга](#) со слегка надуманными примерами. И вот эту [статью](#) можно прочитать для общего развития.

[ответить](#)



kmk

11 апреля 2016 в 12:25

#

★

+2

↑

Практически в каждой статье о шаблоне MVC в iOS делают одну и ту же ошибку:  
Изначально авторы приводят в качестве примера ТТУК и вместо того, чтобы исправить нарушение шаблона MVC, начинают искать решение в использование MVVM VIPER и т.п.  
В MVC, контроллер не должен знать детали реализации представление, а тут автор считает правильным конфигурировать ячейки таблицы именно в контроллере — из-за такой реализации контроллер и представлени жестко зависят друг от друга, и именно такой подход нарушает шаблон MVC и превращают контроллеры в ТТУК.  
Есть простое правило: представьте что вам нужно полностью заменить представление, при этом данные, бизнес логика, обработка действий пользователя остаются неизменными, например, вместо таблицы отображать данные в виде коллекции — если в этом случаи вам придется полностью менять контроллер — вы нарушили шаблон.  
Также передача объекта user в представление userCell никак не нарушает шаблон — здесь user — это пассивная модель, которая отвечает за хранение данных и представление может и должно знать о ней. В MVC представление может зависеть от пассивной модели, модель не может зависеть от представления.

[ответить](#)



 **yozka** 11 апреля 2016 в 12:25 # ★ +1 ↑

Вся проблема в том, что все помешались на MVC, любую архитектуру пытаются решить именно этим паттерном. Вставляют туда, где она изначально не нужна. Самый лучший совет — почитать перед сном «Банду четырёх» а на утро все забыть, и спроектировать как душа пожелает

[ответить](#)

 **AlexIzh** 12 апреля 2016 в 10:18 # ★ h ↑ +1 ↑


Проблема не только в том, что помешались на MVC, а в том, что еще и этот MVC они не понимают или понимают не правильно. Полностью согласен с комментарием выше(коммент). Очень много авторов, кто пишет о паттернах и не понимает MVC, так что уж говорить об остальных разработчиках

[ответить](#)

 **taviscaron** 11 апреля 2016 в 15:01 # ★ +1 ↑

Интересно, как грамотно применить тот же MVP, когда представление — это не лейбл с кнопкой, а Collection View с несколькими секциями и хидером еще каким-нибудь хитрым.

[ответить](#)

 **KAIPIAJIOT** 12 апреля 2016 в 12:57 # ★ h ↑ +1 ↑

Вот [пример](#).

[ответить](#)

 **agee** 11 апреля 2016 в 21:01 # ★ +1 ↑

Тут некоторые не понимают, что это другой MVC. Автор в самом начале написал об этом. Стоит, наверное выделить жирным и подчеркнуть: **Эппловский вариант MVC — не то же самое, что классический MVC**. И это не неправильный, а отличающийся от привычного в вебе MVC.

<https://developer.apple.com/library/mac/documentation/General/Conceptual/DevPedia-CocoaCore/MVC.html> — посмотрите на первую картинку. Посмотрите несколько раз. Контроллер в их шаблоне является посредником между вью и моделью:

A major purpose of view objects is to display data from the application’s model objects and to enable the editing of that data. **Despite this, view objects are typically decoupled from model objects in an MVC application.**

A controller object interprets user actions made in view objects and communicates new or changed data to the model layer. When model objects change, a controller object communicates that new model data to the view objects so that they can display it.

<https://developer.apple.com/library/ios/featuredarticles/ViewControllerPGforiPhoneOS/>

A view controller acts as an intermediary between the views it manages and the data of your app.

Так что все комментарии в духе «в MVC должно быть вот так» тут никак не уместны. Пишу это не столько ради бьющих себя пяткой в грудь, сколько ради начинающих разрабатывать под iOS. Читайте документацию и составляйте собственное мнение.

[ответить](#)

 **coyOne** 12 апреля 2016 в 14:07 # ★ +1 ↑

Впервые за последнее время встречаю так хорошо отредактированный и оформленный материал. Спасибо, приятно читать.

[ответить](#)

Вы не можете комментировать эту публикацию

Можно комментировать публикации, которые не старше 10 дней, а также те, которые вы уже комментировали ранее.

САМОЕ ЧИТАЕМОЕ

Разработка

Сейчас

Сутки

Неделя

Месяц

Введение в React и Redux для бекенд-разработчиков

↑ +18

👁 8,4k

★ 167

💬 83

Guard классы — использовать или нет?

↑ +12

👁 3,3k

★ 15

💬 53

Реверс-инжиниринг радиоуправляемого танка с помощью GNU Radio и HackRF

↑ +44

👁 9,1k

★ 91

💬 21

Об опасностях беспроводных клавиатур и мышей

↑ +54

👁 38,6k

★ 211

💬 42

Продвинутое туннелирование: атакуем внутренние узлы корпоративной сети

↑ +24    👁 3,8k    ★ 71    💬 1

ИНТЕРЕСНЫЕ ПУБЛИКАЦИИ



OWASP TOP 10 2017 RC

👁 69    ★ 1    💬 0

Полярное в декартово или причем тут кольца из монет GT

👁 1,7k    ★ 7    💬 4

Внутренние механизмы TCP, влияющие на скорость загрузки: часть 1

👁 1,3k    ★ 64    💬 0

Война машин: PVS-Studio vs TensorFlow

👁 1,7k    ★ 9    💬 0

Зарядные устройства Rombica обеспечат энергией все — от ноутбуков до коптеров GT

👁 1,3k    ★ 0    💬 10

Strelka

Разделы

Информация

Услуги

Приложения

Профиль

Публикации

О сайте

Реклама

Трекер

Хабы

Правила

Тарифы

Настройки

Компании

Помощь

Контент

Пользователи

Соглашение

Семинары

Песочница

Помощь стартапам



© 2006 – 2017 «**TM**»

Служба поддержки

Мобильная версия

