МГТУ
им. Н.Э. Баумана
ТЕХНОПАРК
Mail.Ru Group

Архитектура

Лекция №6

Носов Павел, Артур Сардарян,
Павел Тихонов

Организационная часть

- Отметиться - важно

- О чем пойдет речь в сегодняшнем занятии
  - Принципы проектирования
  - Архитектурные паттерны

- Оставить отзыв (после занятия)

Структура лекции

- Принципы проектирования
- Архитектуры:
  - MVC
  - MVP
  - MVVM
  - VIPER

Почему стоит позаботиться о выборе архитектуры?

Потому что если вы этого не сделаете. то в один прекрасны ласс с десятками вы окажетесь ить в нем ошибки. Ес держать в голове ка егда будете тер ые детали.

Плохая архитектура

DRY – don’t repeat yourself

Принцип DRY требует, чтобы части информации встречались в вашем коде один, и только один раз*.

*максимум 2 раза, но осознанно!

KISS – keep it simple stupid

Смысл в том, что стоит делать максимально простую и понятную архитектуру, применять шаблоны проектирования и не изобретать велосипед.

YAGNI – You ain’t gonna need it

Суть в том, чтобы реализовать только поставленные задачи и отказаться от избыточного функционала.

SOLID

SOLID


S
Принцип единой ответственности (SRP)
O
Принцип открытости/закрытости (OCP)
L
Принцип заменяемости (LCP)
I
Принцип разделения интерфейсов (ISP)
D
Принцип инверсии зависимостей (DIP)

Single Responsibility Principle

Принцип единственной ответственности

Существует лишь одна причина, приводящая к изменению класса.

Один класс должен решать только какую-то одну задачу. Он может иметь несколько методов, но они должны использоваться лишь для решения общей задачи. Все методы и свойства должны служить одной цели. Если класс имеет несколько назначений, его нужно разделить на отдельные классы.

Single Responsibility Principle

МГТУ им. Н.Э. Баумана ТЕХНОПАРК Mail.Ru Group

Класс делает много вещей!

```
class AirConditioner {
    func turnOn {}
    func turnOff {}
    func changeMode {}
    func changeSpeed {}
}
```

Определим протоколы

```
protocol SwitchOption {
    func turnOn()
    func turnOff()
}
protocol ModeOption {
    func changeMode()
}
protocol FanSpeedOption {
    func controlWindSpeed()
}
```

Делегируем обязанности отдельным объектам

```
class Switch: SwitchOption {
    func turnOn() {
        print("Turn on")
    }

    func turnOff() {
        print("Turn off")
    }
}
class Mode: ModeOption {
    func changeMode() {
        print("Mode changed")
    }
}
class FanSpeed: FanSpeedOption {
    func controlWindSpeed() {
        print("Fan Speed changed")
    }
}
```

Single Responsibility Principle

```
class AirConditionerNew: SwitchOption, ModeOption, FanSpeedOption {
    let modeController = Mode()
    let fanspeedController = FanSpeed()
    let switchController = Switch()

    func turnOn() {
        switchController.turnOn()
    }
    func turnOff() {
        switchController.turnOff()
    }
    func changeMode() {
        modeController.changeMode()
    }
    func controlWindSpeed() {
        fanspeedController.controlWindSpeed()
    }
}
```

Open-closed Principle

Принцип открытости/закрытости

Программные сущности должны быть открыты для расширения, но закрыты для модификации.

Программные сущности (классы, модули, функции и прочее) должны быть расширяемыми без изменения своего содержимого. Если строго соблюдать этот принцип, то можно регулировать поведение кода без изменения самого исходника.

Open-closed Principle

Представьте, что продакт решил добавить в кондиционер новую функцию под названием «Влажность» и попросил вас поддержать эту функцию в вашем текущем приложении.

Как бы вы решили эту задачу?

Open-closed Principle


```
protocol Humidable {
    func changeHumidity(_ value: Int)
}
class HumidityController: Humidable {
    func changeHumidity(_ value: Int) {
        print("You have changed airhumidity to \(value)")
    }
}
extension AirConditionerNew:Humidable {
    func changeHumidity(_ value: Int) {
        let humidController = HumidityController()
        humidController.changeHumidity(value)
    }
}
let acNew = AirConditionerNew()
acNew.changeHumidity(10) // I can change airhumidity - WOW!!!
```

Liskov Substitution Principle

Принцип подстановки Барбары Лисков

Функции, использующие базовые классы, должны уметь использовать сабклассы этих классов, даже не зная об этом.

Подкласс/производный класс должен быть взаимозаменяем с базовым/родительским классом.

Liskov Substitution Principle


```swift
class AirConditioner {

    private let _price: Double
    init(price: Double) {
        self._price = price
    }

    func price() -> Double {
        return _price
    }
}
```

```swift
class DiscountedAirConditioner: AirConditioner {
    override func price() -> Double {
        return super.price() * 0.75
    }
}
```

Научимся выводить цену

```swift
func printPrice(object: AirConditioner) {
    print(object.price())
}

let ac = AirConditioner(price: 110)
let discounted = DiscountedAirConditioner(price: ac.price())

printPrice(object: ac) // Prints 110.0
printPrice(object: discounted) // Prints 85.5
```

Interface Segregation Principle

МГТУ им. Н.Э. Баумана ТЕХНОПАРК Mail.Ru Group

Принцип разделения интерфейса

Нельзя заставлять клиента реализовать интерфейс, которым он не пользуется.

Это означает, что нужно разбивать интерфейсы на более мелкие, лучше удовлетворяющие конкретным потребностям клиентов.

Цель принципа разделения интерфейса заключается в минимизации побочных эффектов и повторов за счёт разделения ПО на независимые части.

Dependency Inversion Principle

МГТУ им. Н.Э. Баумана ТЕХНОПАРК Mail.Ru Group

Принцип инверсии зависимостей

Высокоуровневые модули не должны зависеть от низкоуровневых. Оба вида модулей должны зависеть от абстракций.

Абстракции не должны зависеть от подробностей. Подробности должны зависеть от абстракций.

Dependency Inversion Principle

```
class ConversationDataController {

    let database : CoreDataController

    init(inDatabase:CoreDataController) {
        database = inDatabase
    }

    func getAllConversations(){
        let conversations = [Any]() // array of previous Conversations
        //array of previous conversation is downloaded from API, parsed and created.
        database.saveToDatabase(conversations: conversations)
    }
}

class CoreDataController {
    func saveToDatabase(conversations:[Any]){
        // save conversations to CoreDataController
    }
}
```

Dependency Inversion Principle


```swift
protocol Database{
    func saveToDatabase(conversations:[Any])
}

class ConversationDataController {

    let database : Database

    init(inDatabase:Database) {
        database = inDatabase
    }

    func getAllConversations(){
        let conversations = [Any]() // array of previous Conversations
        //array of previous conversation is downloaded from API, parsed and created.
        database.saveToDatabase(conversations: conversations)
    }
}

class CoreDataController : Database {
    func saveToDatabase(conversations:[Any]){
        // save conversations to CoreDataController
    }
}
```

Проще говоря: зависьте от абстракций, а не от чего-то конкретного.

Применяя этот принцип, одни модули можно легко заменять другими, всего лишь меняя модуль зависимости, и тогда никакие перемены в низкоуровневом модуле не повлияют на высокоуровневый.

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.

Типовые составляющие приложения

1. Пользовательский интерфейс (UI)
2. Бизнес-логика
3. Сетевое взаимодействие
4. Хранение/кеширование данных

Layered архитектура приложения

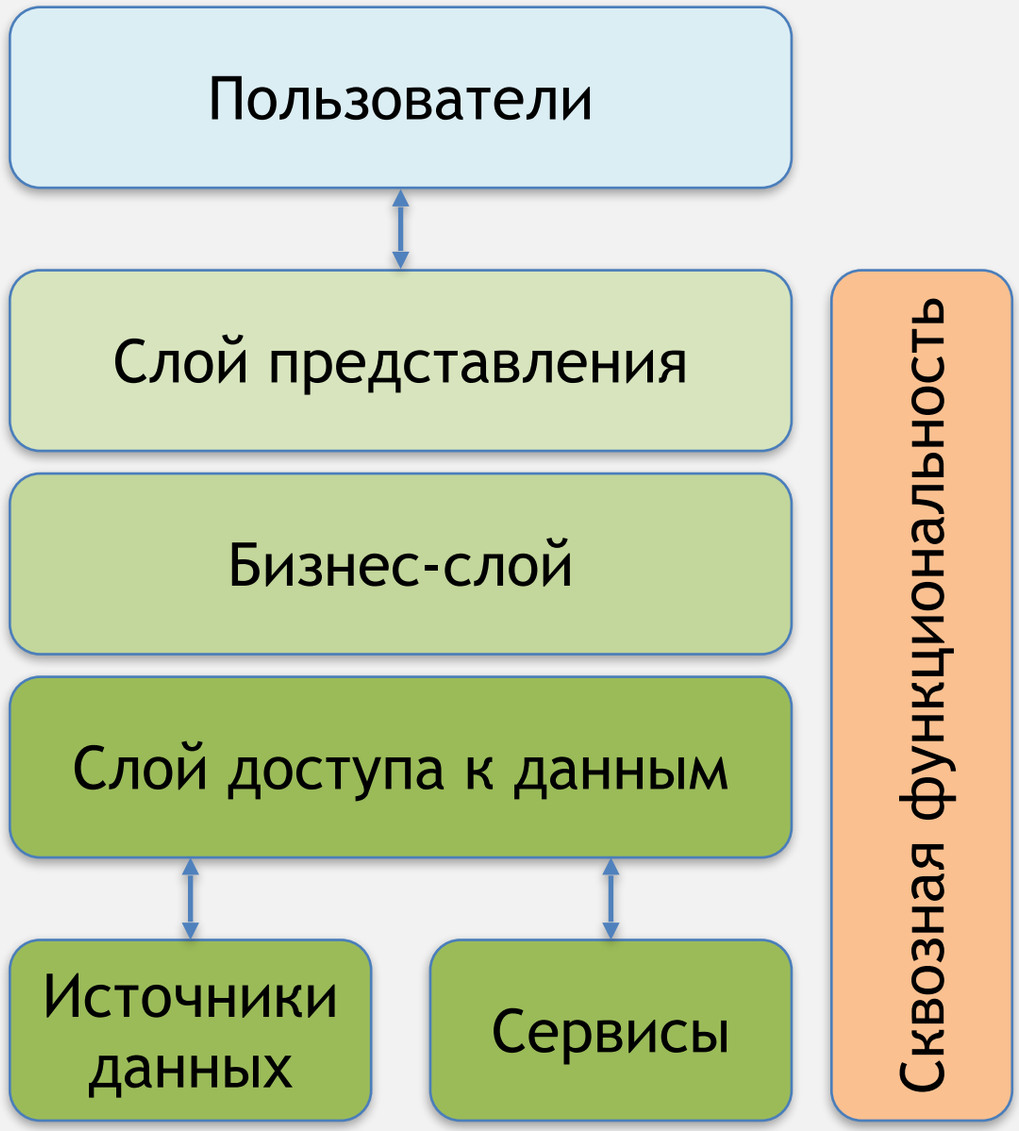
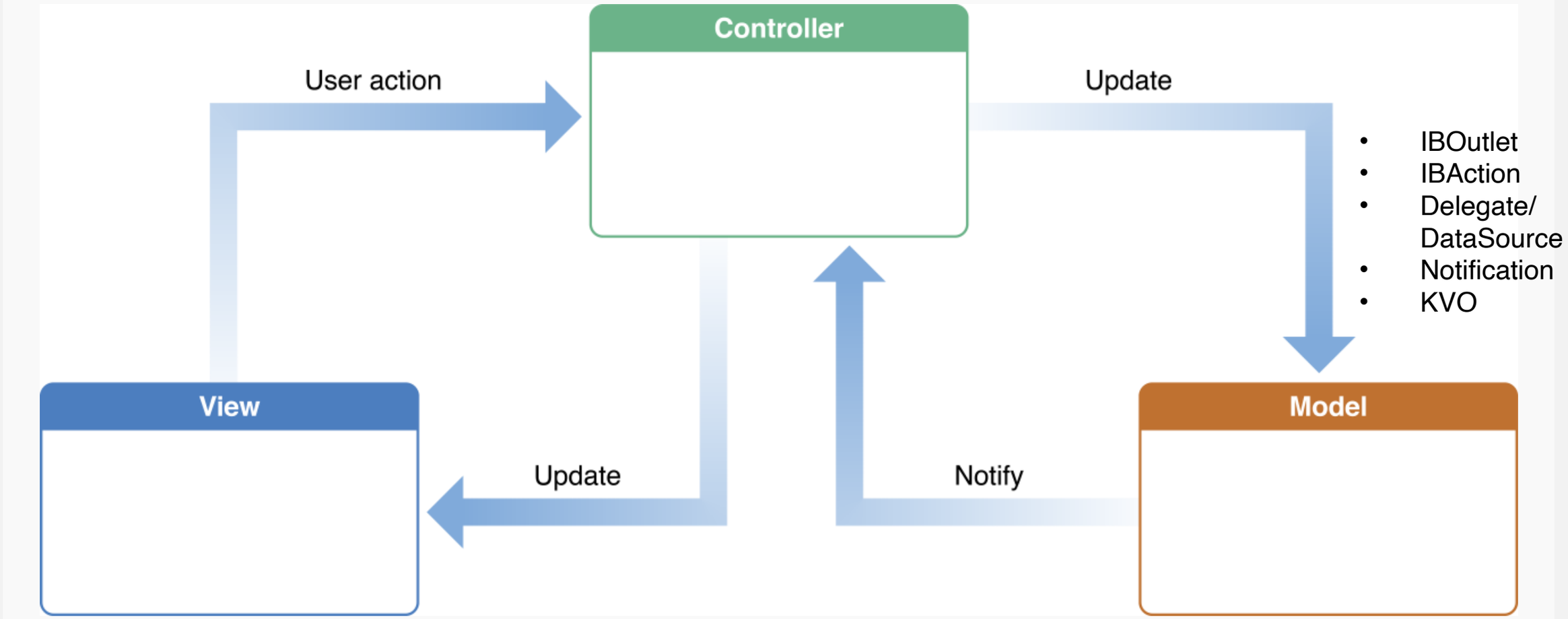


Схема MVC



https://developer.apple.com/library/content/documentation/General/Conceptual/DevPedia-CocoaCore/MVC.html

Model

Слой модели включает в себя данные вашего приложения. Может включать в себя слои:

- Бизнес логика
- Сетевой код
- Взаимодействие с БД
- Константы
- Классы помощники
- и т.д

View

- Стандартные элементы (view, label и т.п.)
- Показывают данные
- Общаются с пользователем
- Ничего не знают о модели

Является максимально простой без какой-либо бизнес логики!

Controller

Координирует взаимодействие между View и Model (посредник)

- Заполняет view данными
- Обрабатывает события от view

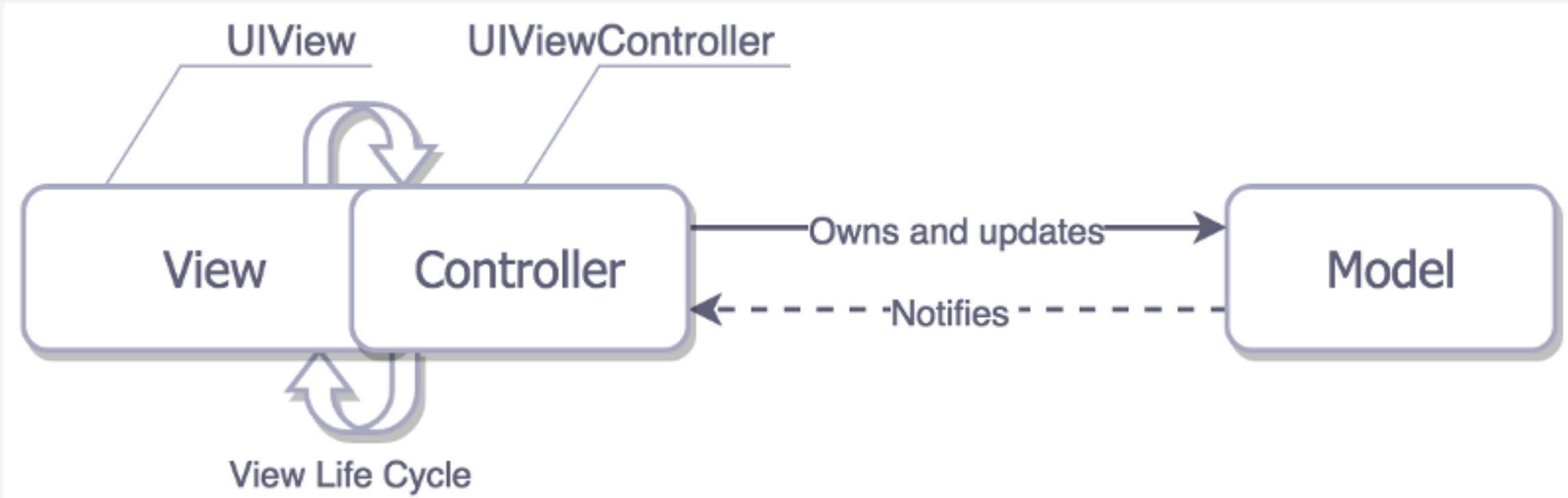
- Передаёт данные в модель
- Берёт данные из модели (модель уведомляет об изменении данных)

MVC: 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.

MVC: реальность

MVC: реальность



Контроллер настолько вовлечен в жизненный цикл **View**, что трудно сказать, что он является отдельной сущностью. В большинстве случаев вся ответственность **View** состоит в том, чтобы отправить действия к контроллеру.

MVC: итоги

Минусы

- 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;

Плюсы

- Простота использования: наименьшее количество кода среди других паттернов. Его легко может поддерживать даже неопытный разработчик.
- Cocoa MVC является лучшим архитектурным паттерном с точки зрения скорости разработки.

Архитектуры приложений

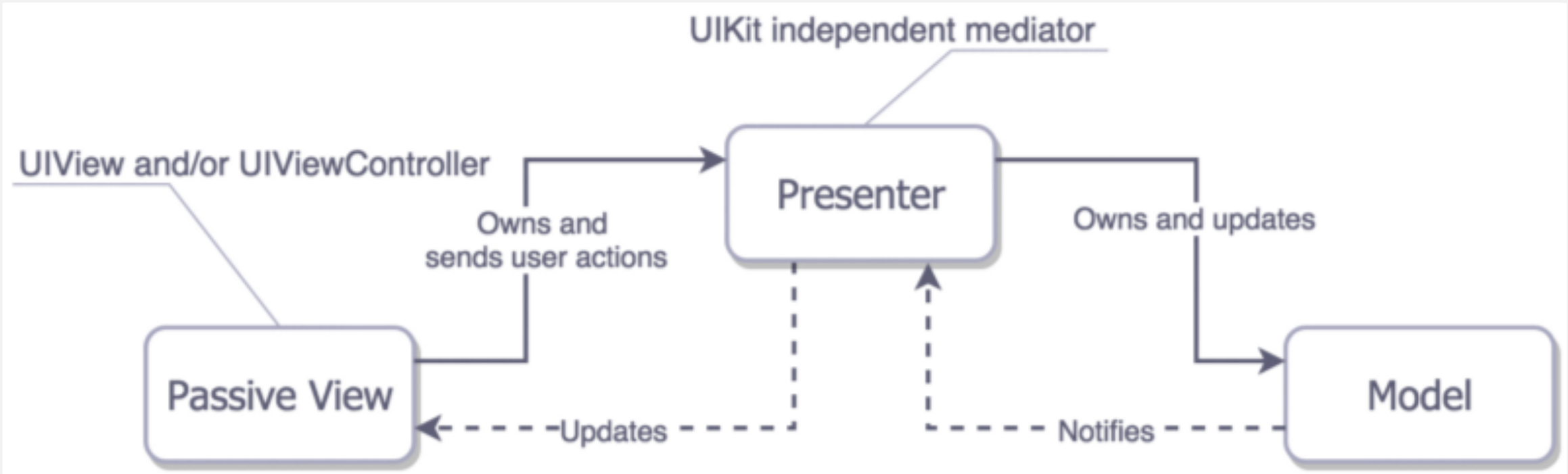
Предпосылки к возникновению разных архитектур:

- MVC не до конца учитывает специфику мобильных приложений
- MVC может трактоваться кучей способов, и это плохо

Что решают другие архитектуры?

- Унификация кода для того, чтобы можно было легче в нём разобраться
- Уменьшение неопределённости при написании кода

MVP



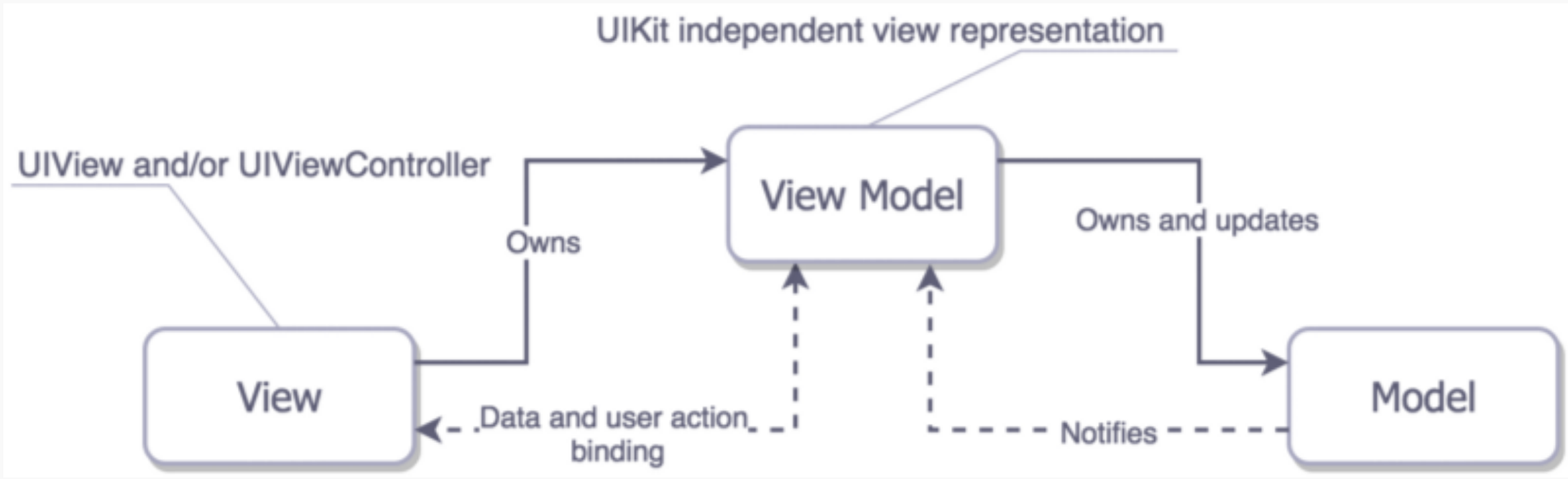
В MVC View и Controller тесно связаны, в то время как посредник в MVP — **Presenter** — не имеет отношения к жизненному циклу View Controller. В **Presenter** нет layout-кода, но он отвечает за обновление **View** в соответствии с новыми данными и состоянием.

MVP: итого

1. Распределение: большая часть ответственности разделена между Presenter и Model, а View ничего не делает;
2. Тестируемость: отличная, мы можем проверить большую часть бизнес-логики благодаря бездействию View;
3. 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

Архитектуры приложений (MVVM)

- MVVM (Model View ViewModel)
  - Отлично работает в связке с Rx* штуками
  - Отчасти решает проблему работы со сложными данными



Она очень похожа на MVP:

1. MVVM рассматривает View Controller как View;
2. в нем нет тесной связи между View и 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.

MVVM: итого

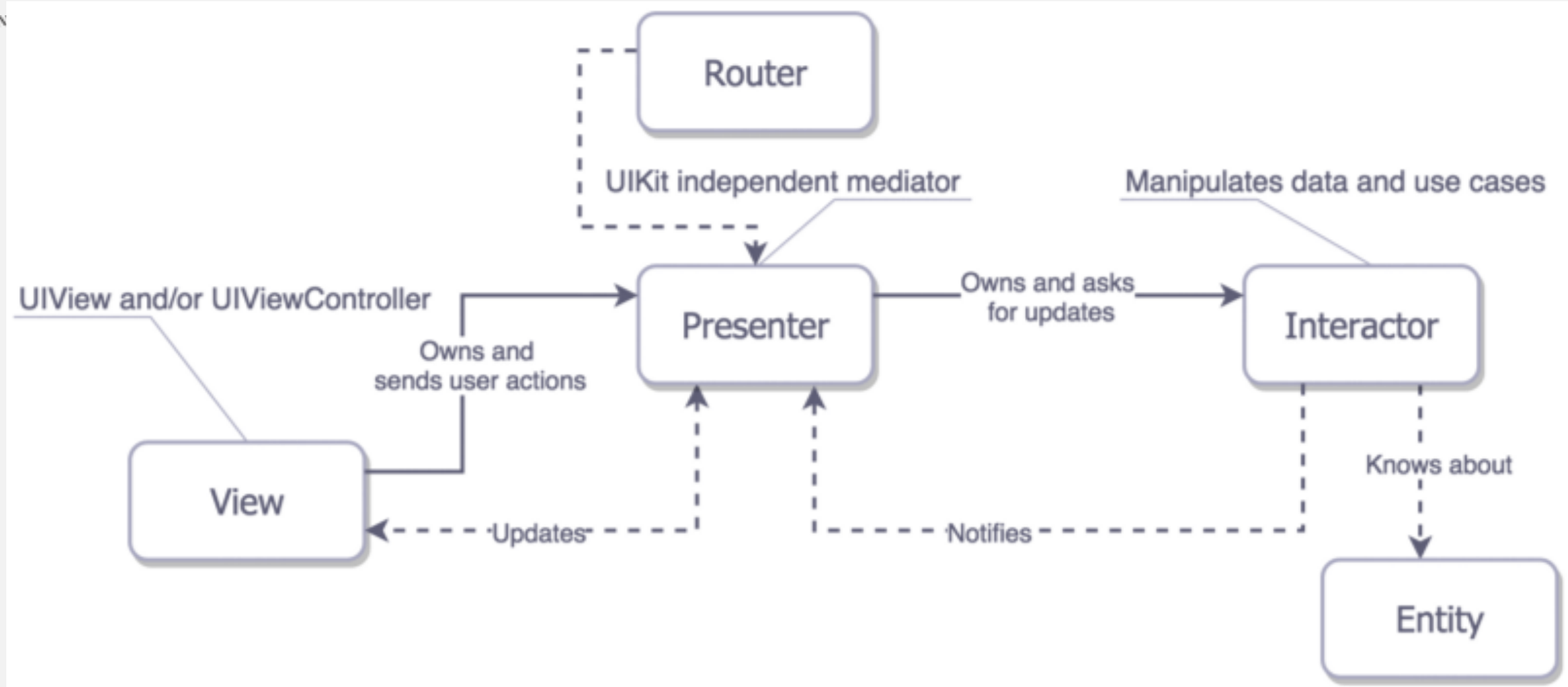
1. распределение: большая часть ответственности разделена между View Model и Model, а View ничего не делает;
2. тестируемость: View Model не знает ничего о представлении, это позволяет нам с легкостью тестировать ее. View также можно тестировать, но так как она зависит от UIKit, вы можете просто пропустить это;
3. 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.

Архитектурные паттерны: итого

1. MVC
   - чем плох?
2. MVP
   - похож на MVC, но
     - presenter не занимается layout
     - вместо view можно использовать mock-объекты
     - view controller относится к view
3. MVVM
   - похож на MVP, но связь не view и model, а view и view model (биндинг)
4. VIPER
   - ещё большее разделение обязанностей
     - view - UI
     - presenter - бизнес-логика, связанная с UI (всё инициализирует в модуле)
     - interactor - бизнес-логика, связанная с данными
     - router - переходы между модулями
     - entity - объекты данных

https://habrahabr.ru/company/badoo/blog/281162/

VIPER



- Основная единица — модуль
- Модули независимы друг от друга
- Модуль это не обязательно == экран, на сложном экране может быть несколько модулей
- Взаимодействие между модулями осуществляется через интерфейсы ModuleInput и ModuleOutput

VIPER

Плюсы

- Переиспользуемость модулей (хотя на практике это не часто применяется)
- Тестируемость

Минусы

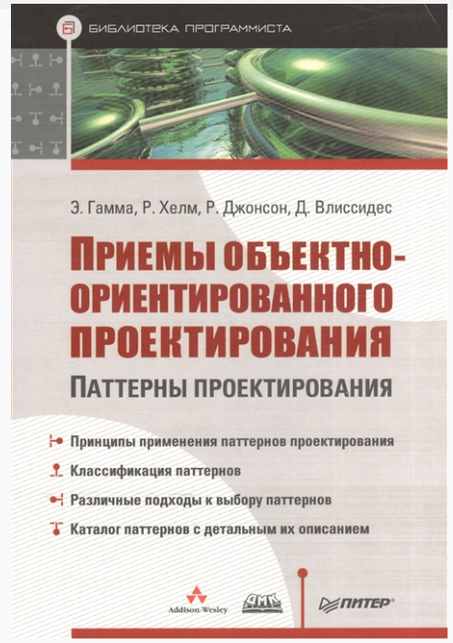
- Многословность, на экран надо писать минимум 5 классов и кучу интерфейсов (отчасти решается кодогенерацией)

Какие еще бывают

1. VIP (View Interactor Presenter)
2. RIBs от Uber
3. Сбербанк, 100 разработчиков, ~1B строк кода
   - Все разбито на frameworks

Ссылки

- **Банда четырех. Паттерны проектирования**
  - Их применение в ios



- Про «тяжёлые» view controller’ы
  https://www.objc.io/issues/1-view-controllers/

- Про структуру проекта в Xcode:
  https://habrahabr.ru/post/261907/

- «Архитектурный дизайн мобильных приложений»
  часть 1: https://habrahabr.ru/company/redmadrobot/blog/246551/
  часть 2: https://habrahabr.ru/company/redmadrobot/blog/251337/

- Про архитектурные паттерны:
  https://habrahabr.ru/company/badoo/blog/281162/

Заключение

- **Сегодня**
  - MVC и другие архитектуры
- **Отзыв**
- **Вопросы?**
  - по лекции
  - по проектам