



Архитектура

Лекция №5



Бабков Андрей, Кулаков Константин,
Носов Павел, Артур Сардарян

Организационная часть



- Отметиться 
- О чём пойдет речь в сегодняшнем занятии
 - Протоколы
 - Принципы проектирования
 - Архитектурные паттерны
 - MVC
 - MVP
 - MVVM
 - VIPER 
- Оставить отзыв (после занятия)

Protocols



Протокол определяет образец методов и свойств, которые соответствуют определенной функциональности.

```
protocol SomeProtocol {  
    // определение протокола...  
}
```

DRY – don't repeat yourself



```
func work() {  
    print("Log 1 line")  
    print("Log 2 line")  
    print("Log 3 line")  
    print("Log 4 line")  
    print("Log 5 line")  
  
    print("Error log 1 line")  
    print("Error log 2 line")  
    print("Error log 3 line")  
    print("Error log 4 line")  
    print("Error log 5 line")  
}
```

Copy-paste - это плохо

DRY – don't repeat yourself



```
func log() {  
    for i in 1...5 {  
        print("Log \$(i) line")  
    }  
  
    for i in 1...5 {  
        print("Error log \$(i) line")  
    }  
}
```

Copy-paste - это плохо

DRY – don't repeat yourself



```
func log(with format: String, count: Int) {  
    for i in 1...count {  
        print(String(format: format, i))  
    }  
}  
  
log(with: "Log %d line", count: 5)  
log(with: "Error log %d line", count: 5)
```

Copy-paste - это плохо

KISS – keep it simple stupid



```
public enum LogBuildLevel {
    case debug // лог только для дебага. Используется по умолчанию.
    case release // лог и для дебага, и для релиза.
}

public protocol Logger {
    func log(_ message: Any?, level: LogLevel, buildLevel: LogBuildLevel)
}

public extension Logger {
    func log(_ message: Any?, level: LogLevel = .info, buildLevel: LogBuildLevel = .debug) {
        self.log(message, level: level, buildLevel: buildLevel)
    }
}

public func makeLogger(tag: String, environment: Environment) -> Logger {
    var loggers = [Logger]()
    loggers.append(CrashlyticsLogger(with: CrashlyticsServiceFactory.crashlyticsService(for:
        if environment.modulesEnvironment.isDebugEnabledYoulaLogger {
            loggers.append(ConsoleLogger())
        }
        return LoggerImpl(tag: tag, loggers: loggers, environment: environment)
    })
}
```

Стоит делать максимально просто и понятно, не стоит усложнять то, что возможно не понадобится

```
private final class LoggerImpl: Logger {

    private let tag: String?
    private let loggers: [Logger]
    private let environment: Environment

    init(tag: String, loggers: [Logger], environment: Environment) {
        self.tag = tag
        self.loggers = loggers
        self.environment = environment
    }

    func log(_ message: Any?, level: LogLevel, buildLevel: LogBuildLevel) {
        guard let message = message else { return }
        if environment.modulesEnvironment.isDebugEnabledYoulaLogger {
            if level == .critical {
                assertionFailure("\(prepare(message, level: level))")
            } else {
                if buildLevel == .debug {
                    return
                }
                let targetMessage = prepare(message, level: level)
                loggers.forEach { $0.log(targetMessage, level: level, buildLevel: buildLevel) }
            }
        }
    }

    private func prepare(_ message: Any, level: LogLevel) -> Any {
        if let error = message as? Error {
            return prepareError(from: error as NSError, level: level)
        } else {
            return buildMessage(from: "\(message)", level: level)
        }
    }

    private func buildMessage(from message: CustomStringConvertible, level: LogLevel) -> String {
        let targetMessage: String
        if environment.modulesEnvironment.isDebugEnabledYoulaLogger {
            if let string = message as? String {
                targetMessage = string
            } else if let debugConvertible = message as? CustomDebugStringConvertible {
                targetMessage = debugConvertible.debugDescription
            } else {
                targetMessage = message.description
            }
        }
```

SOLID



Сложный свод простых
правил:

- S** - single responsibility
- O** - open–closed
- L** - Liskov substitution
- I** - interface segregation
- D** - dependency inversion



S – single responsibility



Каждый класс должен решать **одну** узко-направленную задачу и делать ее хорошо.

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



```
class SwitchController {  
    func turnOn() {}  
    func turnOff() {}  
}  
  
class ModeController {  
    func changeMode() {}  
}  
  
class SpeedController {  
    func changeSpeed() {}  
}
```

```
class AirConditioner {  
    let switchController = SwitchController()  
    let modeController = ModeController()  
    let speedController = SpeedController()  
  
    func turnOn() {  
        switchController.turnOn()  
    }  
  
    func turnOff() {  
        switchController.turnOff()  
    }  
  
    func changeSpeed() {  
        speedController.changeSpeed()  
    }  
  
    func changeMode() {  
        modeController.changeMode()  
    }  
}
```

O – open-closed



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

Возможность легкого добавления нового функционала классу
без изменения кода существующего.

```
class Logger {  
    func log() {  
        for i in 1...5 {  
            print("Printing ine \\" + String(i))  
        }  
    }  
  
    let logger = Logger()  
    logger.log()
```

O – open-closed



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

Возможность легкого добавления нового функционала классу
без изменения кода существующего.

```
class Logger {  
    func log(useAlert: Bool) {  
        for i in 1...5 {  
            if useAlert {  
                //present UIAlertController  
            } else {  
                print("Printing ine \\\(i)")  
            }  
        }  
    }  
  
    let logger = Logger()  
    logger.log(useAlert: true)
```

O – open-closed



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

Возможность легкого добавления нового функционала классу без изменения кода существующего.

```
protocol Logger: AnyObject {
    func log()
}

final class ConsoleLogger: Logger {
    func log() {
        for i in 1...5 {
            print("Printing line \\" + String(i))
        }
    }
}

final class AlertLogger: Logger {
    func log() {
        // UIAlertController
    }
}

var loggers: [Logger] = [ConsoleLogger()]
loggers.append(AlertLogger())

loggers.forEach { logger in
    logger.log()
}
```

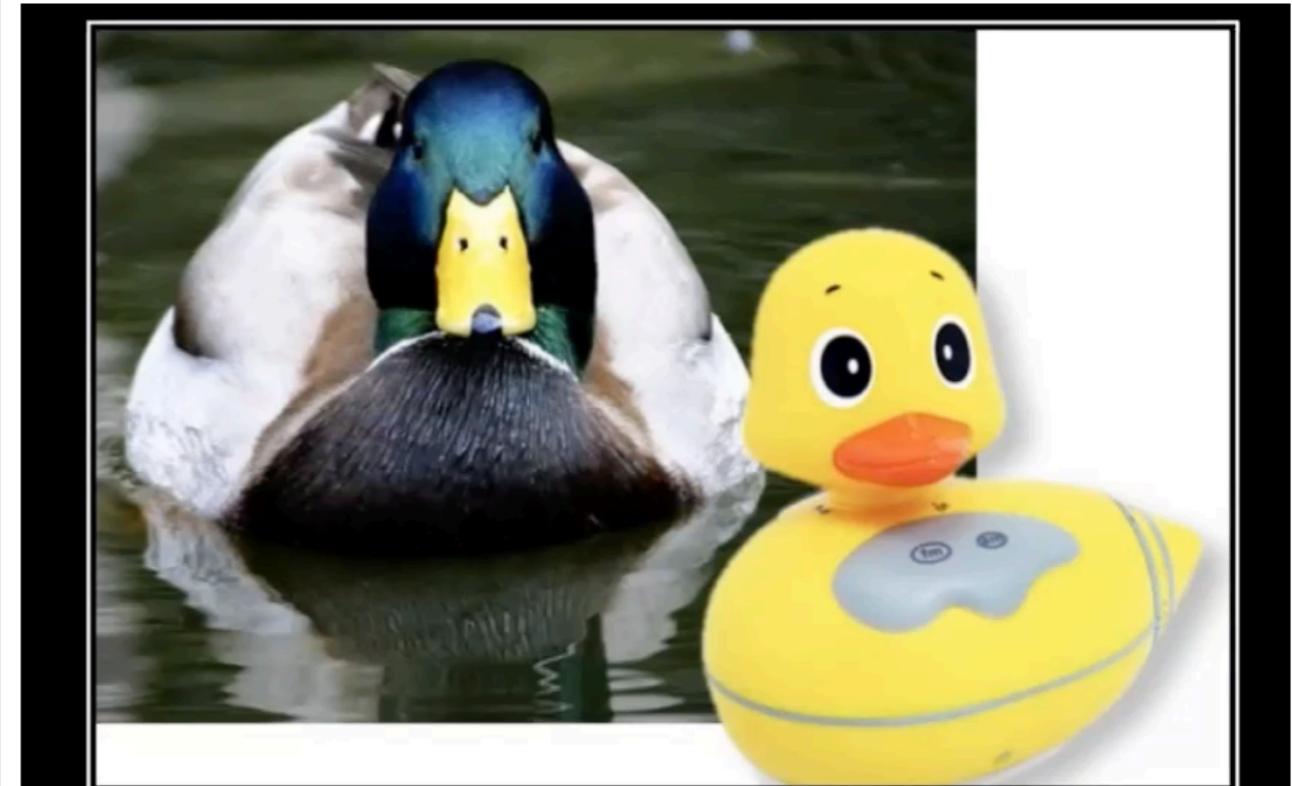
L – Liskov substitution



Принцип подстановки Барбары Лисков.

Функции, использующие базовые классы, должны уметь работать с их сабклассами.

Производные классы не должны изменять поведения базовых классов, а только **дополнять**.



LISKOV SUBSTITUTION PRINCIPLE

If It Looks Like A Duck, Quacks Like A Duck, But Needs Batteries - You Probably Have The Wrong Abstraction

L – Liskov substitution



Принцип подстановки Барбары Лисков.

```
protocol Logger: AnyObject {
    func log()
}

final class ConsoleLogger: Logger {
    func log() {
        for i in 1...5 {
            print("Printing line \(i)")
        }
    }
}
```

```
final class BadLogger: Logger {
    func log() {
        for i in 1...5 {
            DispatchQueue.global(qos: .background).async {
                print("Bad printing line \(i)")
            }
        }
    }
}
```

```
let loggers: [Logger] = [ConsoleLogger(), BadLogger()]

loggers.forEach { logger in
    logger.log()
}
```

```
Printing line 1
Printing line 2
Printing line 3
Printing line 4
Printing line 5
Bad printing line 1
Bad printing line 4
Bad printing line 2
Bad printing line 5
Bad printing line 3
```

| – interface segregation



Принцип разделения интерфейсов.

Нельзя заставлять клиента реализовывать интерфейс, которым он не пользуется.

Много маленьких интерфейсов (протоколов) лучше, чем один большой.

```
public protocol UITableViewDataSource : NSObjectProtocol {

    @available(iOS 2.0, *)
    func tableView(_ tableView: UITableView, numberOfRowsInSection section: Int) -> Int

    // Row display. Implementers should *always* try to reuse cells by setting each cell's reuseIdentifier and querying for available reusable cells with
    // dequeueReusableCellWithIdentifier:
    // Cell gets various attributes set automatically based on table (separators) and data source (accessory views, editing controls)

    @available(iOS 2.0, *)
    func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell

    @available(iOS 2.0, *)
    optional func numberOfSections(in tableView: UITableView) -> Int // Default is 1 if not implemented

    @available(iOS 2.0, *)
    optional func tableView(_ tableView: UITableView, titleForHeaderInSection section: Int) -> String? // fixed font style. use custom view (UILabel) if you want something
    // different

    @available(iOS 2.0, *)
    optional func tableView(_ tableView: UITableView, titleForFooterInSection section: Int) -> String?

    // Editing

    // Individual rows can opt out of having the -editing property set for them. If not implemented, all rows are assumed to be editable.
    @available(iOS 2.0, *)
    optional func tableView(_ tableView: UITableView, canEditRowAt indexPath: IndexPath) -> Bool

    // Moving/reordering

    // Allows the reorder accessory view to optionally be shown for a particular row. By default, the reorder control will be shown only if the datasource implements
    // -tableView:moveRowAtIndexPath:toIndexPath:
    @available(iOS 2.0, *)
    optional func tableView(_ tableView: UITableView, canMoveRowAt indexPath: IndexPath) -> Bool

    // Index

    @available(iOS 2.0, *)
    optional func sectionIndexTitles(for tableView: UITableView) -> [String]? // return list of section titles to display in section index view (e.g. "ABCD...Z#")

    @available(iOS 2.0, *)
    optional func tableView(_ tableView: UITableView, sectionForSectionIndexTitle title: String, at index: Int) -> Int // tell table which section corresponds to section
    title/index (e.g. "B",1)

    // Data manipulation - insert and delete support
}
```

D – dependency inversion



Принцип иверсии зависимостей.

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

Абстракции не зависят от деталей.

```
class BooksController {
    let database = DataBaseController()

    func getAllBooks() {
        database.getAllBooks()
    }

    func getBook(with identifier: String) {
        database.getBook(with: identifier)
    }

    func deleteAllBooks() {
        database.deleteAllBooks()
    }
}
```

```
protocol DataBaseControllerDescription {
    func getAllBooks()
    func getBook(with identifier: String)
    func deleteAllBooks()
}

class DataBaseController: DataBaseControllerDescription {
    // ...
}

class BooksController {
    let database: DataBaseControllerDescription = DataBaseController()

    func getAllBooks() {
        database.getAllBooks()
    }

    func getBook(with identifier: String) {
        database.getBook(with: identifier)
    }

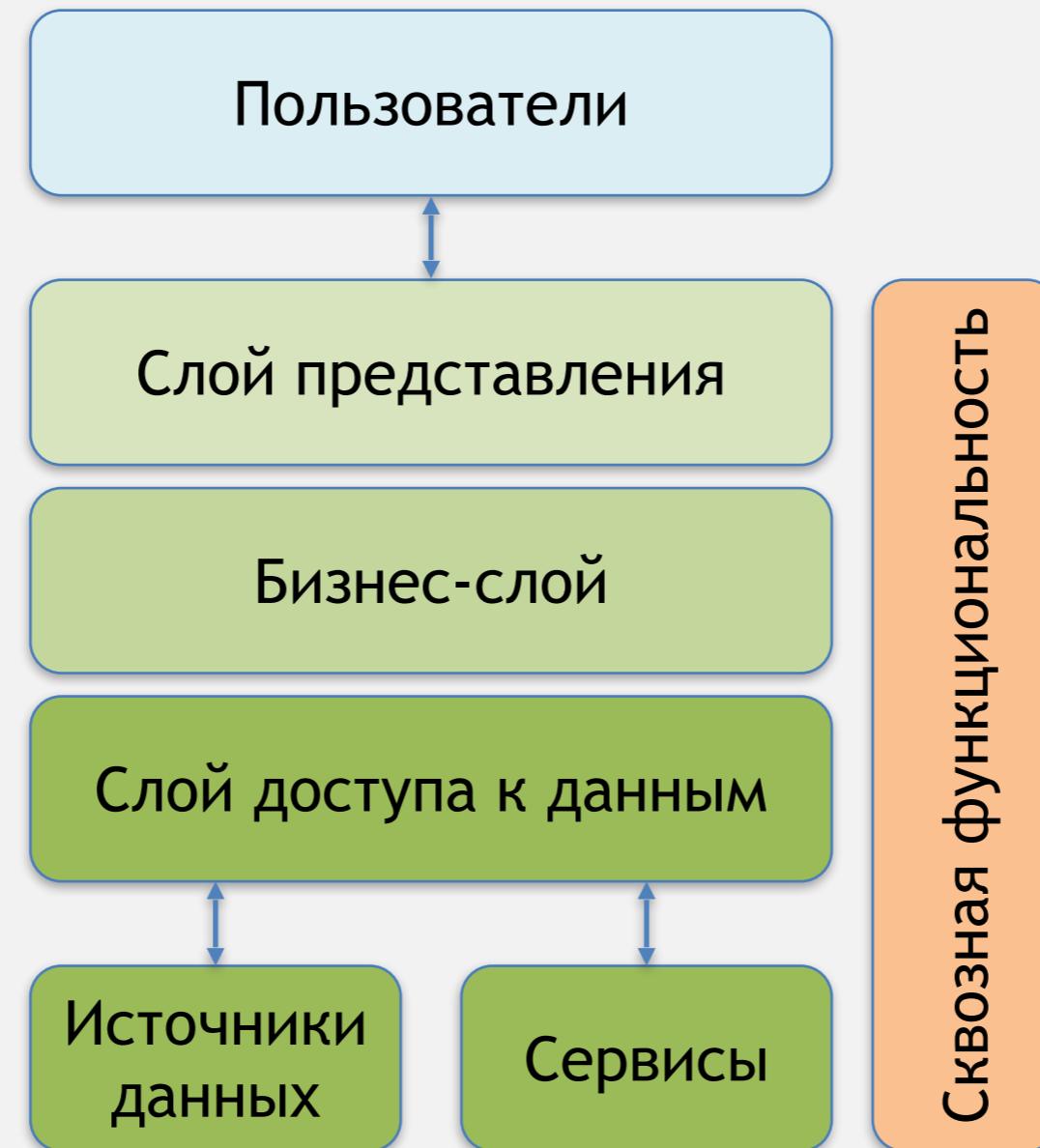
    func deleteAllBooks() {
        database.deleteAllBooks()
    }
}
```

Архитектурные паттерны



Самая важная тема дня

Layered архитектура приложения



Model - представление данных.
Подразумевается слой как самих сущностей,
так и доступ к ним.

View - UIKit based классы

Controller - посредник между Model & View

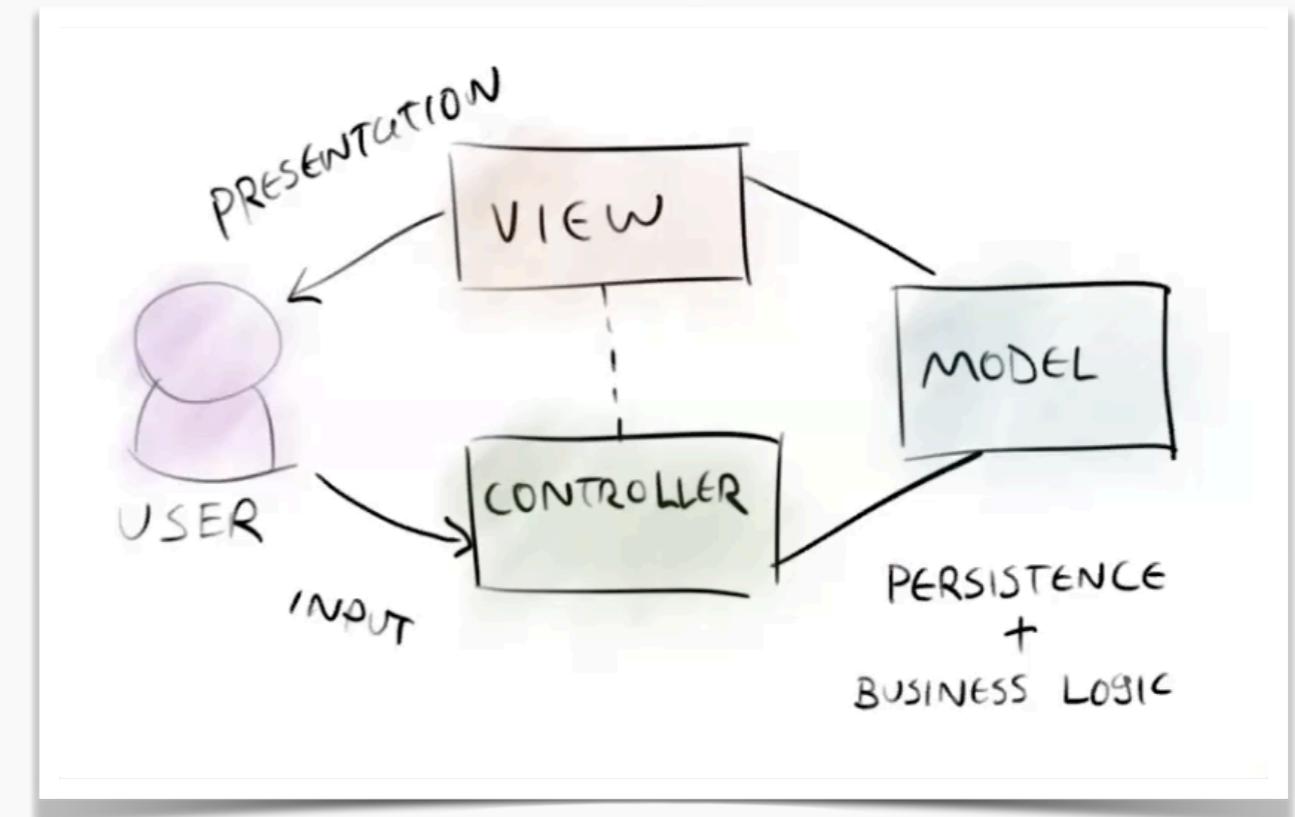


MVC & Apple MVC

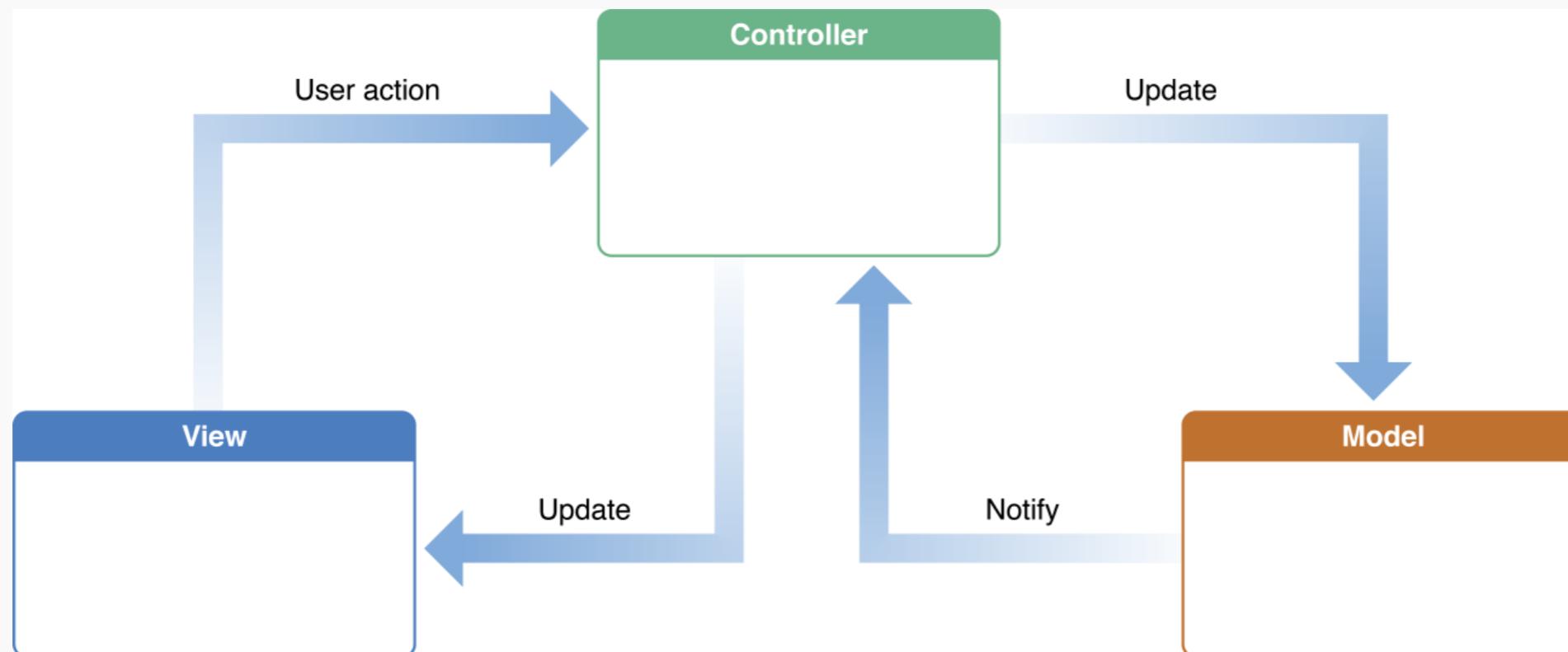


Classic MVC - имеет связку модели с контроллером и модели с отображением

Apple MVC - первая попытка разделить View & Model



Apple MVC



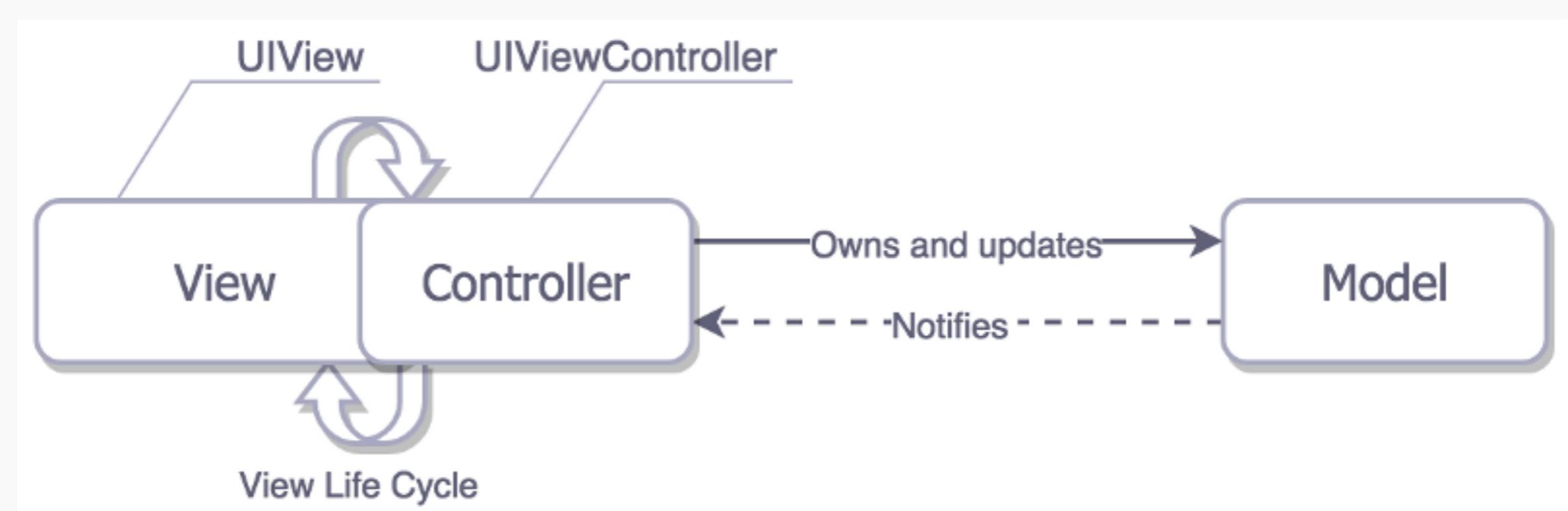
В Apple MVC модель активная

Демо MVC



{ code }

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.

Почти вся бизнес логика и роутинг в контроллере.

Используем в простых VC.

Архитектуры приложений

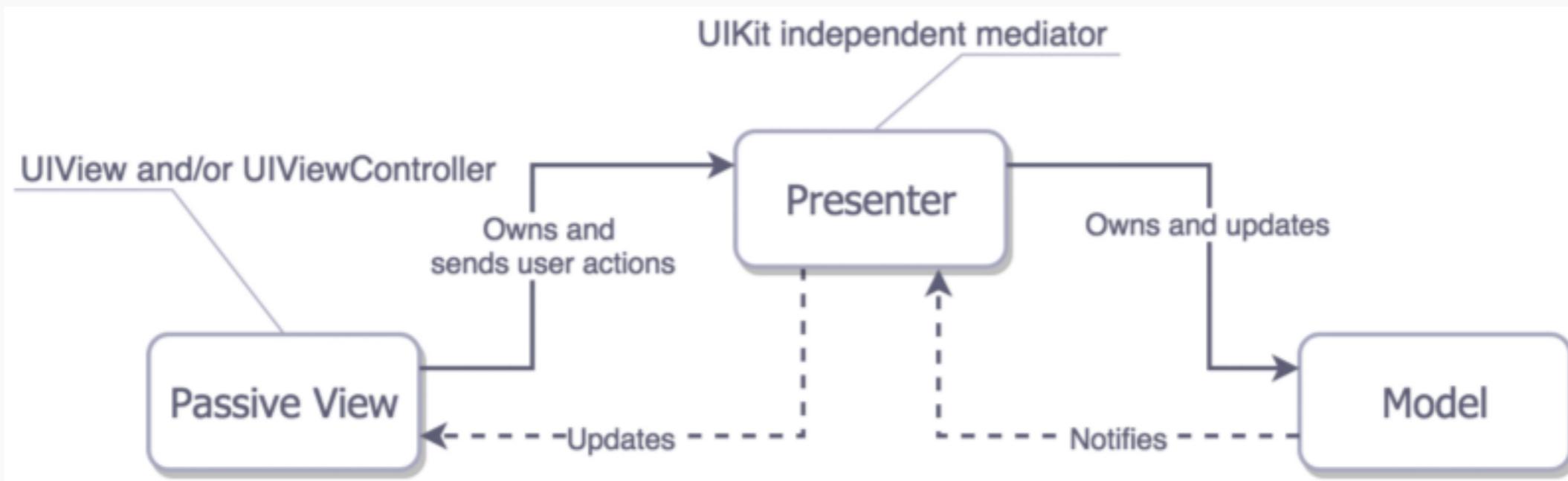


Предпосылки к возникновению разных архитектур:

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

Что решают другие архитектуры?

- Разделение ответственостей
- Уменьшение неопределённости при написании кода



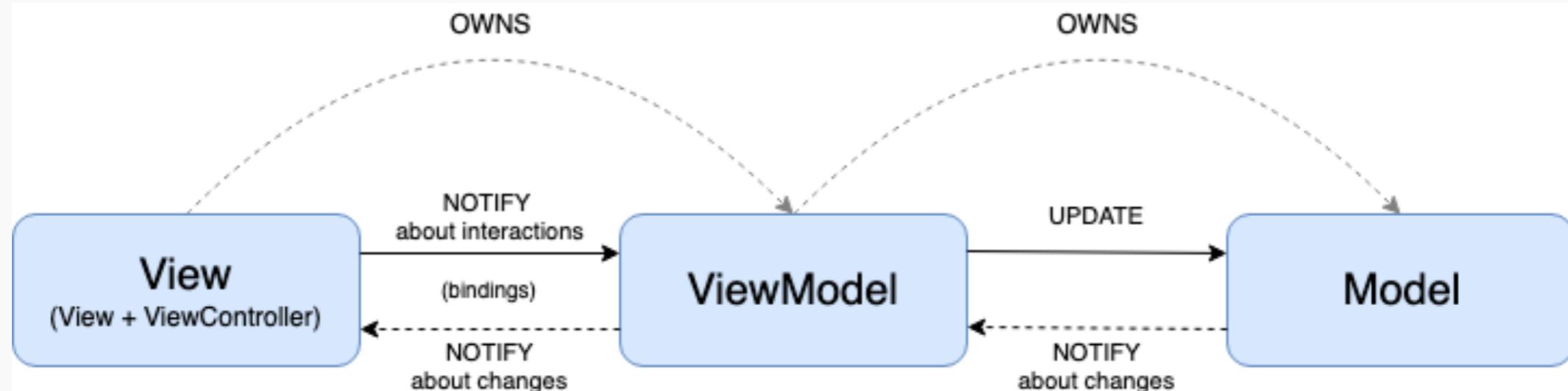
Presenter **не знает** о UIKit-е.

Бизнес логика отделена 

Демо MVP



{ code }



Model - слой самих сущностей и доступа к ним.

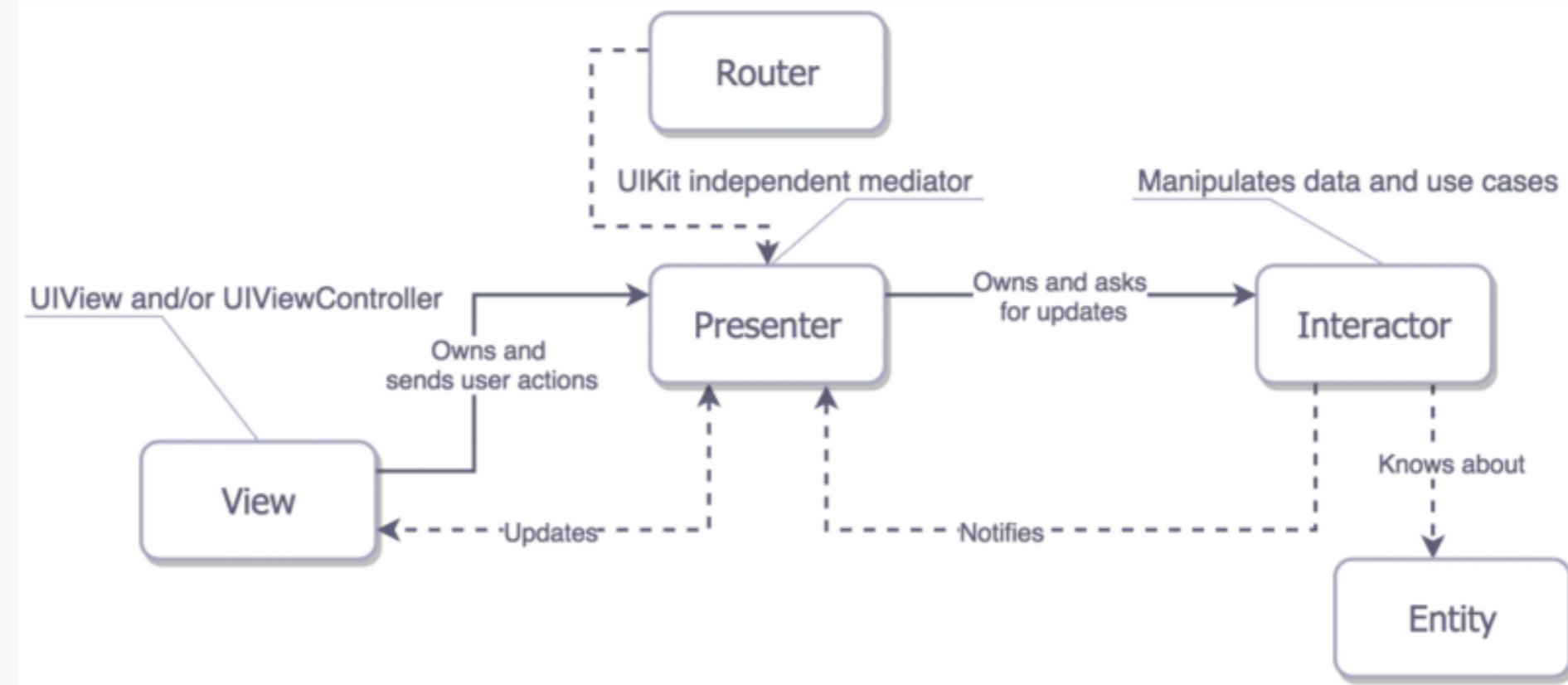
View - UIKit based классы.

ViewModel - содержит независимое от UI фреймворков состояние интерфейса.

Демо MVVM



{ code }



View - отображение интерфейса

Interactor - содержит бизнес-логику модуля. Взаимодействует с сервисным слоем + Entity.

Presenter - посредник между View & Interactor.

Entity - объектная модель бизнес-логики приложения. Представляет собой набор свойств, не содержит логики.

Router - навигация

Демо VIPER



{ code }

Заключение



- Сегодня
 - Архитектуры с примерами
- Отзыв !
- Вопросы
 - по лекции
 - по проектам

Архитектурные паттерны: итого



1. MVC
 - монолит, но быстро
2. MVP (похож на MVC, но)
 - presenter не знает про UIKit
 - Бизнес логика отделена и ее удобнее тестировать
3. MVVM
 - похож на MVP, но с data binding-ами
4. VIPER (четкое разделение ответственостей)
 - view - UI
 - presenter - посредник
 - interactor - бизнес-логика
 - router - переходы между модулями
 - entity - объекты данных

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

