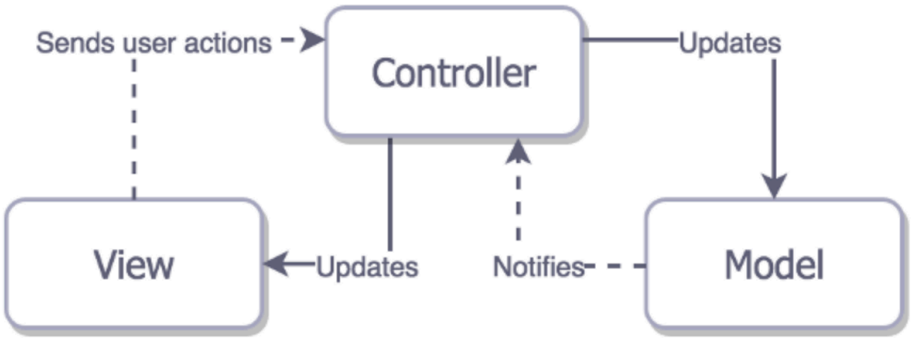
Lesson 4. Архитектурные паттерны в iOS.

Причинно-следственные связи. MV(X) архитектура, как альтернатива MVC.

MVC от Apple



Советы по созданию вью контроллеров

- **Используйте поставляемые со стандартным SDK классы View Controllers.** В iOS SDK имеется множество контроллеров представления, решающих конкретные задачи. Хорошей практикой является использование таких, поставляемых системными библиотеками, контроллеров.

- **Создавайте View Controller максимально автономным.** View Controller не должен знать о внутренней логике другого контроллера или о его иерархии view.
- **Не храните во View Controller данные**. View Controller выступает посредником между модельным слоем и слоем представления при обмене данными. Его основная обязанность — гарантировать, что view отображает правильную информацию.
- **Используйте View Controller для реакции на внешние события**. К внешним событиям относятся: взаимодействие пользователя с интерфейсом и различные системные уведомления.

Недостаток данных рекомендация в том, что они провоцируют неконтролируемый рост сложности View Controller. Их неразрывная связь с жизненным циклом вью, является особенностью iOS SDK и требует формировать реакции на события этого цикла непосредственно во View Controller. Реализация обработки внешних событий пользовательского ввода также происходит во View Controller. Помимо этого модель часто становится слишком пассивной и используется исключительно для доступа к данным, а вся бизнес-логика оказывается во View Controller.

В итоге, вью контроллеры разрастаются до гигантских размеров, превращаясь в то, что называется Massive-View-Controller.

Недостатки Massive-View-Controller

- **Высокая сложность поддержки и развития**. Код сложно модифицировать и расширять из-за его высокой связности и неочевидных потоков данных. Существует риск, что при внесении изменений можно сломать текущую функциональность приложения и до определенного момента не замечать этого.
- **Высокий порог вхождения**. Структура кода запутана и требует время на изучение.
- **Код слабо тестируем**. Из-за высокой связности код не покрывается модульными тестами. Попытка проверить логику вью приводит к явному вызову методов жизненного цикла, которые неявно могут повлечь за собой загрузку всех связанных сабвью, что нарушает принципы unit-тестирования.

- **Код практически невозможно переиспользовать**. Реализация новой функциональности приложения происходит непосредственно в контроллере представления. Переиспользовать такой код без переиспользования всего контроллера не представляется возможным. Сложно выполнять рефакторинг и декомпозицию.

Именно эти проблемы и привели к тому, что начали появляться альтернативные классической схеме MVC решения. И все эти альтернативы решают одну общую задачу: это разгрузка вью контроллера и распределение его обязанностей.

Признаки хорошей архитектуры

1. Сбалансированное распределение обязанностей между сущностями

Распределение позволяет сделать класс более простым и понятным, за счет того, что он становится гораздо меньше. Таким образом, самый простой способ уменьшить сложность заключается в разделении обязанностей между несколькими сущностями по принципу *единой ответственности* (The Single Responsibility Principle).

2. Каждая сущность должна иметь строго-определенную роль

Когда сущности отделены друг от друга они становятся проще для понимания и их можно переиспользовать, не дублируя код.

3. Тестируемость

Тестирование является важным аспектом при построении приложений и данный пункт определяет, то на сколько просто будет писать юнит-тесты.

4. Простота использования, масштабируемость и низкая стоимость обслуживания.

Масштабируемость так же является немаловажным фактором при выборе архитектуры. Чем проще масштабировать код, тем проще внедряются новые фичи. Цена тут определяется как в натуральном выражении: это количество кода и время на разработку; так и в денежном, так как время влияет на стоимость. Также необходимо учитывать затраты на последующее обслуживание.

Виды архитектурных паттернов в iOS

- MVC
- MVP
- MVVM
- VIPER
- Clean Swift

Категории сущностей MVC, MVP и MVVM

- **Model** — это сущность, которая описывают объект
- **View** — это сущность, которая отвечает за пользовательский интерфейс. Все классы фреймворка UIKit.
- **Controller / Presenter / ViewModel** — посредник между **Model** и **View**. Отвечает за изменения модели данных, реагируя на манипуляции пользователя с интерфейсом. Так же передает данные для отображения в интерфейс, используя изменения из **Model**.

Шаблон «Скромный объект»

Первоначально данный шаблон был спроектирован для использования в юнит тестировании, как способ отделения поведения, легко поддающегося тестированию, от поведения, с трудом поддающегося тестированию. Для этого необходимо разделить

поведение на два модуля или класса. Один из модулей называется «*скромным*» и содержит все, что с трудом поддается тестированию. Во втором содержится все остальное.

Графические пользовательские интерфейсы сложны для модульного тестирования. Но если отделить поведение интерфейса от его внешнего вида, то большая его часть будет легко поддаваться тестированию. Используя шаблон «Скромный объект», можно разделить два вида поведения на два разных класса, которые называют Презентатором (Presenter) и Представлением (View).

View представляет из себя «*скромный*» объект, но при этом он сложен для тестирования. Поэтому код в этом объекте необходимо упростить до предела. Он должен просто переносить данные в графический интерфейс, никак не обрабатывая их.

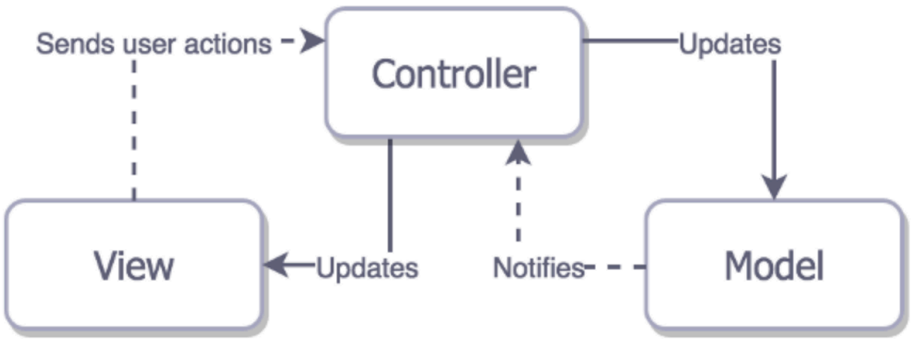
Presenter наоборот легко поддается тестированию. Его задача — получить данные от приложения и преобразовать их так, чтобы View могло просто отобразить их на экране.

Все, что отображается на экране и чем так или иначе управляет приложение, представлено во View Model простыми базовыми типами. На долю вью остается только перенести данные из Model View на экран. То есть View играет скромную роль.

Разница подходов MVC, MVP и MVVM

MVC от Apple

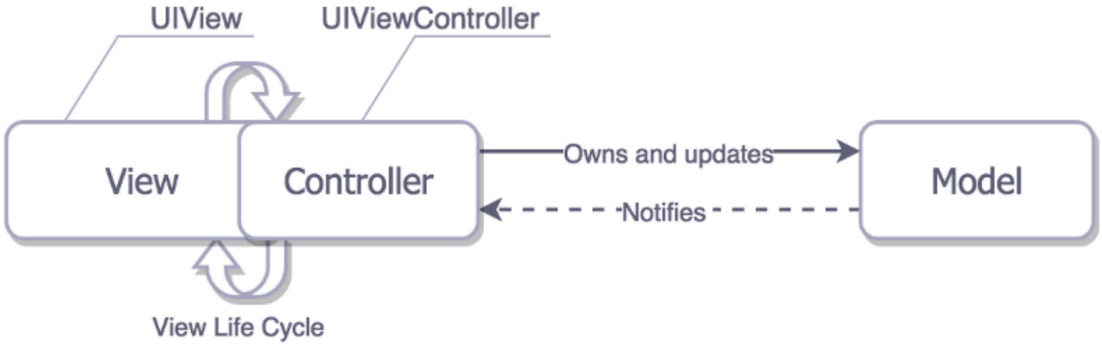
Ожидания



Controller — это посредник между View и Model, а значит, две последних не знают о существовании друг друга.

В теории все выглядит достаточно просто, но на практике несколько иначе:

Реальность



MVC заставляет использовать View Controller для реализации любых задач, превращая его тем сам в Massive View Controller. Это связано с тем, что контроллер настолько вовлечен в жизненный цикл View, что практически является с ним одним целым. Вся деятельность View сводится к тому, чтобы отправить действия контроллеру. В итоге

View Controller превращается в контейнер для хранения кода, который выполняет сразу все что необходимо для реализации бизнес логики.

Пример сборки по MVC

```
// Model
struct Person {
    let name: String
    let surname: String
}
```

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

    override func viewDidLoad() {
        super.viewDidLoad()

        showGreetingButton.addTarget(self, action: #selector(didTapButton),
    for: .touchUpInside)
    }

    @objc func didTapButton() {
        let greeting = "Hello" + " " + person.name + " " + person.surname
        greetingLabel.text = greeting
    }

    // layout code goes here
}
```

```
// Assembling of MVC
let model = Person(name: "Tim", surname: "Cook")
let view = GreetingViewController()

view.person = model
```

Схема MVC с точки зрения признаков хорошей архитектуры

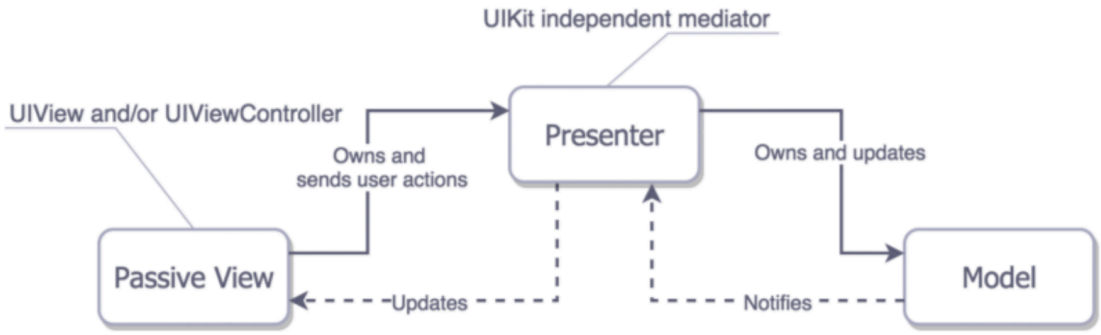
- **распределение**: данный пункт относится только к *View* и *Model*;
- **строгое определение ролей**: *View* и *Controller* тесно связаны, а это плохо;

- **тестируемость**: из-за плохого распределения вы, покрывать бизнес логику тестами очень сложно;
- **простота использования**: Работая по MVC будет написано наименьшее количество кода по сравнению альтернативными решениями. MVC выглядит понятным, поэтому его легко может поддерживать даже начинающий разработчик.

Выводы:

С точки зрения скорости разработки MVC является лучшим выбором.

MVP



Тут представлены те же три блока во главе которых вместо **Controller** стоит **Presenter**. В отличии от MVC, в качестве вью тут представлена пассивная **View**, при этом третий модуль так же представлен моделью. Так же как и в схеме MVC модель и пассивная вью ни чего не знают друг о друге и общаются через посредника, которым выступает **Presenter**.

В отличии от MVC где **View** и **Controller** тесно связаны между собой, в данной схеме **Presenter** не имеет отношения к жизненному циклу вью контроллера. Это значит, что **View** можно с легкостью заменить Mock-объектами, поэтому **Presenter** в данном

случае избавлен от кода, связанным с жизненным циклом вью и лэйаутом, и при этом он отвечает за обновление **View** в соответствии с новыми данными и состоянием.

В архитектуре MVP вью контроллеры относятся не к **Presenter**, а к **View**. Это ключевое отличие от MVC и за счет этого обеспечивается превосходная тестируемость. Но при таком подходе нужно вручную связывать данные и события между **View** и **Presenter**.

Пример сборки по MVP

```
// Model
struct Person {
    let name: String
    let surname: 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" + " " + person.name + " " + person.surname
        view.setGreeting(greeting)
    }
}


class GreetingViewController: UIViewController {
```

```
    var presenter: GreetingViewPresenter!
    let showGreetingButton = UIButton()
    let greetingLabel = UILabel()

    override func viewDidLoad() {
        super.viewDidLoad()

        showGreetingButton.addTarget(self, action: #selector(didTapButton),
    for: .touchUpInside)
    }

    @objc func didTapButton(_ button: UIButton) {
        presenter.showGreeting()
    }

    // layout code goes here
}



extension GreetingViewController: GreetingView {
    func setGreeting(_ greeting: String) {
        greetingLabel.text = greeting
    }
}



// Assembling of MVP
let model = Person(name: "Tim", surname: "Cook")
let view = GreetingViewController()
let presenter = GreetingPresenter(view: view, person: model)
view.presenter = presenter
```

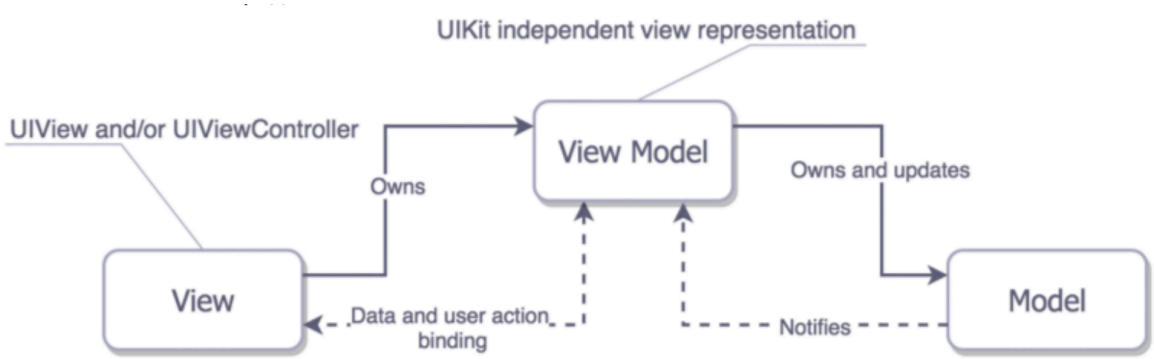
Схема MVP с точки зрения признаков хорошей архитектуры

- **распределение**: При данном подходе большая часть ответственности разделена между **Presenter** и **Model**, тогда как **View** (вью контроллер) ничего не делает;
- **строгое определение ролей**: Данный пункт выполняется
- **тестируемость**: отличная, мы можем проверить большую часть бизнес-логики благодаря бездействию View;
- **простота использования**: Количество кода в два раза больше, чем в том же примере с использованием MVC.

Выводы:

MVP в iOS означает превосходную тестируемость и много кода.

MVVM



- **Model** в схеме MVVM на самом деле не отличается от модели, представленной в схеме MVC. В общем и целом модель используют для описания полей какого конкретного объекта.
- **View** представляет из себя пользовательский интерфейс, сюда же относится и вью контроллер. В данной схеме вью берет на себя часть обязанностей, выполняемых классом UIViewController.
- **View Model** - С одной стороны View Model представляет из себя статическую модель для отображения вью, т.е. модель интерфейса; а с другой — отвечает за сбор, интерпретацию и преобразование данных. Это позволяет разгрузить View Controller на столько, чтобы оставить в нем только те задачи, которые необходимы для отображения интерфейса и для реагирования на действия пользователя. При таком подходе вью контроллер выступает посредником между пользователем и моделью данных.

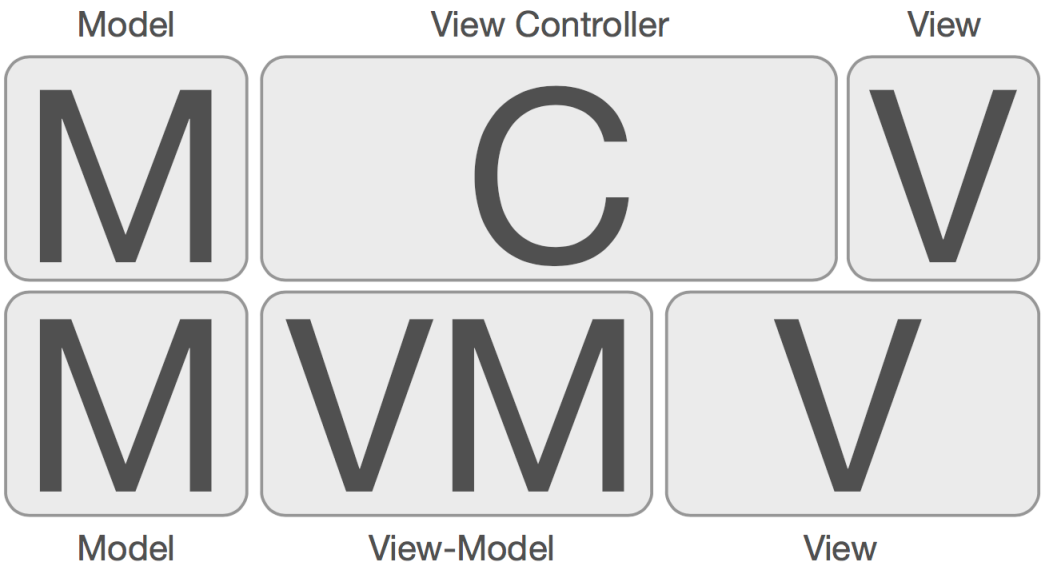
View-Model представляет из себя независимое от UIKit представление View и ее состояния. С одной стороны она передает новые значения в модель, а с другой стороны самостоятельно обновляется в соответсвии с данными модели. Связь между моделью и View Model происходит посредством байндингов.

> Байндинг (Связывание данных) — это процесс, который устанавливает соединение между пользовательским интерфейсом и бизнес-логикой.

На самом деле схема MVVM очень похожа на схему MVP:

- Так же, как и в MVP вью контроллер тут рассматривается, как вью
- Так же, как и в MVP тут нет тесной связи между моделью и вью

Сравнение MVVM и MVC:

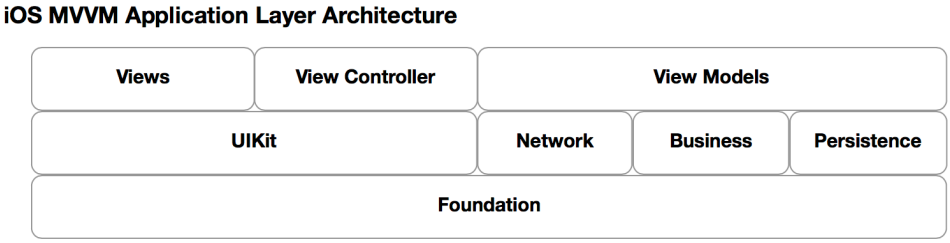


- Размеры блоков тут распределены пропорционально своим обязанностям
- В схеме MVC большую часть обязанностей берет на себя View Controller
- В схеме MVVM большую часть обязанностей вью контроллера берет на себя View-Model, оставшуюся часть — View

В аббревиатуре MVVM, нет упоминания о вью контроллере, но это не значит, что он полностью исключается из этой архитектуры. Большая часть его обязанностей распределяется между View и View Model, а на View Controller остаются только обязанности, связанные с отображением интерфейса и с взаимодействием пользователя, что значительно разгружает его и делает намного проще.

Связь между View-Model и вью контроллером устанавливается на уровне объявления экземпляра View-Model во вью контроллере. При этом данная связь является односторонней, т.к. View-Model ни чего не знает о вью контроллере. Сам вью контроллер знает о View-Model тоже не очень много, так как ему доступны только публичные свойства экземпляра View-Model.

Следующая схема показывает, как различные компоненты приложения сочетаются друг с другом, а так же где начинается и где заканчивается ответственность этих компонентов



Пример сборки по MVVM

```
// Model
struct Person {
    let name: String
    let surname: String
}
```

```
protocol GreetingViewModelProtocol: class {

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

```
class GreetingViewModel: GreetingViewModelProtocol {

    let person: Person
    var greeting: String? {
        didSet {
            greetingDidChange?(self)
        }
```

```
    }
    var greetingDidChange: ((GreetingViewModelProtocol) -> ())?

    required init(person: Person) {
        self.person = person
    }

    func showGreeting() {
        greeting = "Hello" + " " + person.name + " " + person.surname
    }
}



class GreetingViewController: UIViewController {

    var viewModel: GreetingViewModelProtocol! {
        didSet {
            self.viewModel.greetingDidChange = { [unowned self] viewModel in
                self.greetingLabel.text = self.viewModel.greeting
            }
        }
    }
    let showGreetingButton = UIButton()
    let greetingLabel = UILabel()

    override func viewDidLoad() {
        super.viewDidLoad()

        showGreetingButton.addTarget(viewModel, action: Selector(("showGreeting")),
    for: .touchUpInside)
    }

    // layout code goes here
}



// Assembling of MVVM
let model = Person(name: "Tim", surname: "Cook")
let viewModel = GreetingViewModel(person: model)
let view = GreetingViewController()
view.viewModel = viewModel
```

Схема MVVM с точки зрения признаков хорошей архитектуры

- **распределение**: В MVVM на вью ложится больше обязанностей, чем в MVP. Разница в том, что в MVVM мы устанавливаем связь с View-Model и за счет установки байндингов вью сама обновляет свое состояние, тогда как в MVP вью себя не обновляет. Это делает Presenter, в который вью направляет все события;

- **строгое определение ролей**: Здесь небольшая часть ответственности распределена между View-Model и View;
- **тестируемость**: View-Model легко поддается тестированию, так как она ни чего не знает о View;
- **простота использования**: MVVM будет выигрывать у MVP за счет байндингов, благодаря которым обновление вью происходит автоматически, тогда как в MVP вью приходится обновлять вручную

Выводы:

> MVVM сочетает в себе преимущества вышеупомянутых подходов и при этом, благодаря байндингам, не требует дополнительного кода для обновления View. Кода не так много, как в MVP, тестируемость гораздо выше, чем в MVC.