Архитектурные паттерны в iOS. Конспект. Habr 2

Архитектуры MV(X)

Вид архитектур, часто использующихся в iOS-разработке, включающих так или иначе 3 элемента:

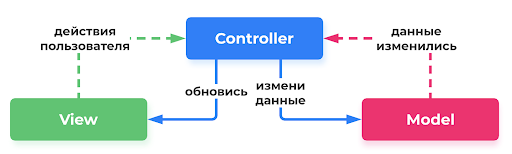
* Модели. Отвечают за использование бизнес-данных (логику).
* Вид. Отображает текущую картинку и принимает события от пользователя.
* Контроллер/Презентер/ViewModel. Преобразую события. Передают события вида в модель, а события модели в вид.

Каждый элемент может быть активным - уведомлять окружение об изменениях или пассивным – т.е. не уведомлять.

Хорошая архитектура:

* Делает код читабельнее
* Дает возможность переиспользовать код
* Позволяет проводить Unit-тестирование необходимых модулей проекта
* Позволяет масштабировать проекта с меньшими затратами

Cocoa MVC



\*под сплошными линиями стоит понимать сильную связь, под пунктирными - слабую

Здесь:

* Контроллер владеет (держит сильной ссылкой) видом и моделью. Является посредником между видом и моделью, принимающим от них события и посылающим в них в ответ на эти события запросы на изменения.
* Вид не знает о модели и не владеет контроллером. Посылает уведомление о событиях контроллеру. Зависим от контроллера поскольку последний в ответ на уведомление о действиях пользователя посылает запрос на изменения данных в модель. На основании обновленных данных модели происходит изменение вида.
* Модель не знает о виде и не владеет моделью вида. Посылает уведомление об изменении данных контроллеру. Зависима от контроллера, поскольку последний в ответ на уведомление об изменении данных посылает запрос на изменение вида.

Основной минус:

* Вид напрямую не отрабатывает actions(например переходы), а контроллер не управляет видом. Для корректной работы приложения приходиться стирать границы между ними, в связи с чем происходит сильная завязка контроллера и вида друг на друге.

Дополнительные минусы:

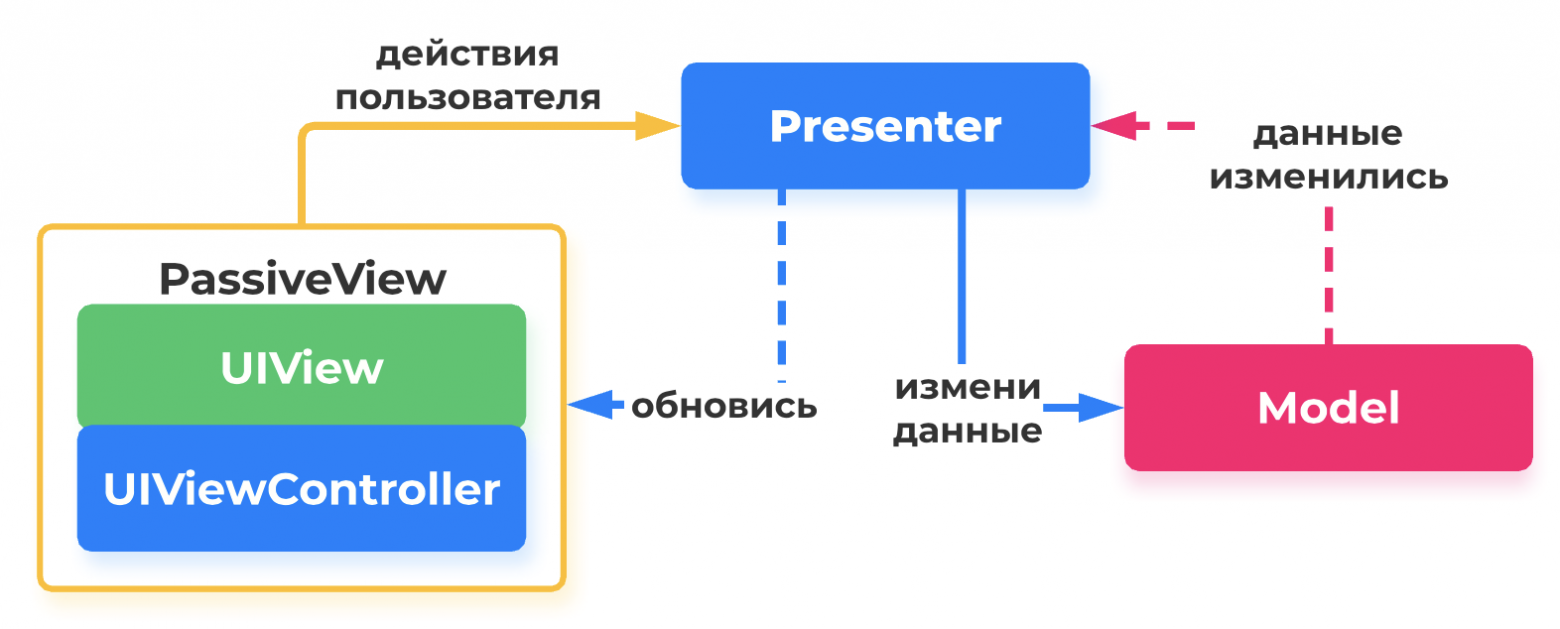
* Нереализуема в чистом виде в больших проектах
* Конкретно не определен ответственный за вид элемент
* Сложность тестирования проекта

Из-за “разрастаимости” контроллера данную модель также называют Massive View Controller.

Задумываемые плюсы:

* Изменение вида без внесения изменений в код модели
* Изменение реакции на действия пользователя путем создания другого контроллера без изменения вида.
* Бизнес-логика не зависит от представления

MVP



Здесь:

* Презентер занимается передачей данных и событий между видом и моделью. Зависит от вида, поскольку он передает ему действия пользователя, на основании которых презентер посылает запрос на изменение данных в модель.
* Вид включает в себя View + ViewController. Не владеет моделью, изменяющей данные, на основании которых меняется вид. Зато владеет презентером, у которого запрашивает реакцию на действия пользователя.
* Модель не знает о виде и не владеет моделью вида. Посылает уведомление об изменении данных презентеру. Зависима от презентера, поскольку последний в ответ на уведомление вида о действиях пользователя посылает запрос на изменение данных в модель.

Основной минус - проблема сборки:

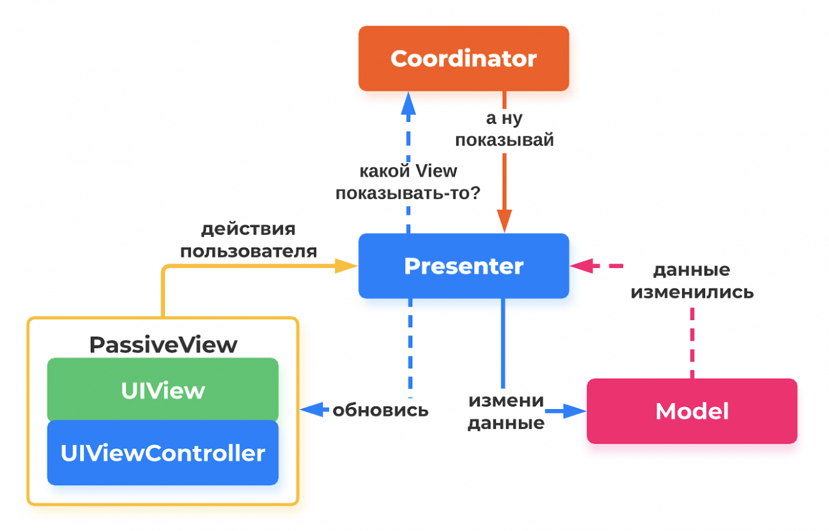
* Неявно требуется ориентация на UIViewController в качестве единицы, отвечающей за сборку и навигацию. Это приводит к тому, что вид должен содержать ссылку на объект контроллера, который в свою очередь через презентер содержит ссылку на модель, что приводит к нарушению принципа разделения ответственности.

Дополнительные минусы:

* Переиспользовать можно только модель

Решается проблема сборки созданием ещё одного управляющего слоя – роутера (координатор), который собирает MVP-кусочки, управляет их жизненным циклом и передает их друг другу при необходимости.

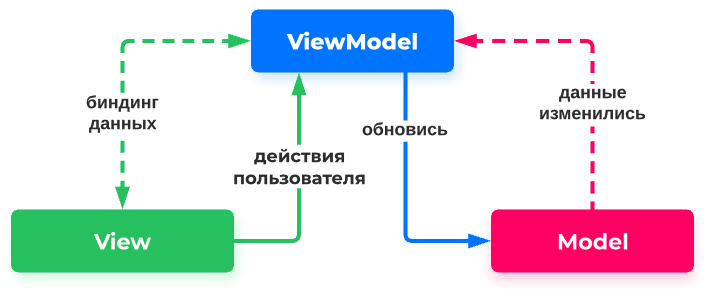
MVP + C



Задумываемые плюсы:

* Позволяет проводить Unit-тестирование
* Позволяет масштабировать проект за счет конкретного разделения сущностей
* Бизнес-логика не зависит от представления

MVVM



\*биндинг (связывание своих данных)

Здесь:

* Модель вида. Биндится с данными вида. Зависит от вида, поскольку получает от него уведомления о действиях пользователя, на основании которых делает запрос на обновление данных в модель.
* Вид не знает о модели. Вид включает в себя View + ViewController. Владеет моделью вида для реактивного обновления своих данных. Биндится с данными модели вида.
* Модель не знает о виде. Зависима от модели вида, поскольку последний в ответ на уведомление вида о действиях пользователя должен владеть тем, кому посылает запрос на изменение данных в модель. Посылает уведомление об изменении данных модели вида.

Основной минус:

* Модель вида занимается управлением состояниями своего вида и зависимостей, и различного рода логикой. В связи с чем её код разрастается, и мы возвращаемся к проблеме MassiveVC.

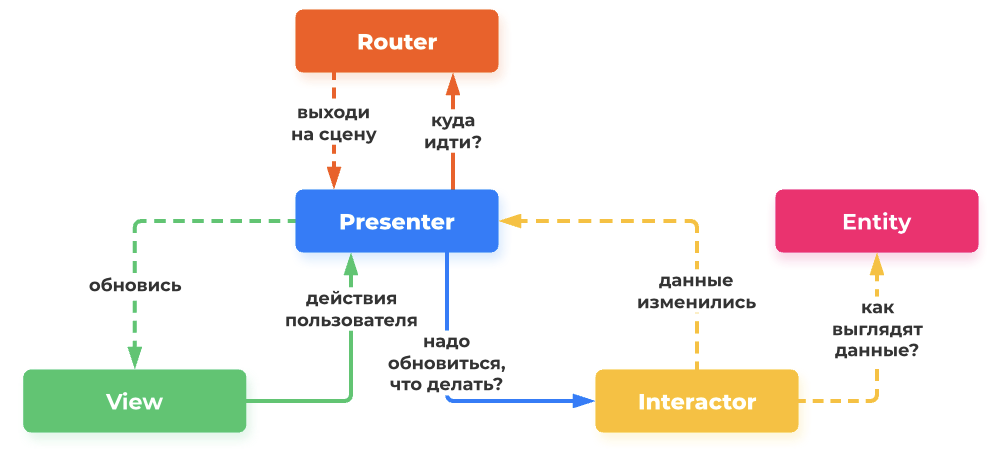
Дополнительные минусы:

* Сложность управления памятью
* Проблема сборки (когда при инициализации сущности узнают друг о друге)

Задумываемые плюсы:

* Реактивное обновление данных
* Позволяет проводить Unit-тестирование
* Позволяет масштабировать проект за счет конкретного разделения сущностей
* Модель вида можно переиспользовать

VIPER



Здесь:

* **View** показывает, что скажет Презентер и передает ввод пользователя Презентеру
* **Interactor** содержит описание сценария использования
* **Presenter** содержит логику отображения и умеет подготавливать данные для представления пользователю, а также реагировать на ввод пользователя
* **Entity** описание предметной модели
* **Routing** описывает логику навигации между экранами