АРХИТЕКТУРЫ ПРОЕКТИРОВАНИЯ ПРИЛОЖЕНИЙ НА БАЗЕ IOS

Белорусский государственный университет информатики и радиоэлектроники г. Минск, Республика Беларусь

Будевич К. В., Нестерович Н. С.

Казека А. А. - канд. техн. наук, доц.
Лавренов А. В. - канд. техн. наук, доц.

В данной статье рассмотрены основные концепции проектирования мобильных приложений на базе iOS, такие как MVC, MVP, MVVM, VIPER. В проектировании iOS приложений ни одна из разобранных архитектур не будет идеально подходить для всех типов проектов. Архитектуру необходимо выбирать из доступных средств разработки, уровня подготовленности команды или разработчика, и общей сложности приложения.

В проектировании приложений на базе iOS существуют 4 основные архитектурные концепции - MVC, MVP, MVVM, VIPER. Общая направленность этих концепций такова, что в любых приложениях, построенных по данным моделям, должно обеспечиваться разделение данных от внешнего интерфейса.

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

- **Models** — ответственные за данные домена или слой доступа к данным, который манипулирует данными;
- **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** [1].

Перед разбором каждой концепции необходимо определить **признаки** хорошей архитектуры:

- сбалансированное **распределение** обязанностей между сущностями с жесткими ролями;
- **тестируемость**;
- **простота использования** и низкая стоимость обслуживания.

Концепция MVC. Controller является посредником между **View** и **Model**, следовательно, два последних компонента не знают о существовании друг друга. Поэтому **Controller** трудно переиспользовать. В данной концепции **Controller** тесно связан с жизненным циклом **View**, и сложно сделать вывод, что он является отдельной сущностью. Есть возможность разгрузить часть бизнес-логики и преобразования данных с **Controller** в **Model**, но, когда дело доходит до отгрузки работы во **View**, не так много вариантов. В большинстве случаев вся ответственность **View** состоит в том, чтобы отправить действия к **Controller**. В итоге **View Controller** в iOS становится делегатом и источником данных, а также местом запуска и отмены серверных запросов и, в общем-то, всего чего угодно. После выше сказанного может показаться что, Cocoa MVC является довольно плохим выбором архитектуры iOS приложения. Но необходимо оценить 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 возможно будет 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. Cocoa MVC — это разумный выбор, если нету возможности инвестировать много времени в архитектуру, так же эта концепция имеет низкий порог входа для других разработчиков.

Концепция MVP. MVP производный от MVC, который используется в основном для построения пользовательских интерфейсов. Элемент **Presenter** в данном шаблоне берёт на себя функциональность посредника (аналогично **Controller** в MVC) и отвечает за управление событиями пользовательского интерфейса так же, как в других шаблонах обычно отвечает представление. Слой **Presenter** здесь является абстракцией над **View**, который управляет отображением, но не содержит в себе детали реализации отображения. Также, **Presenter** является посредником между данными и отображением.

Выделим **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**: 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.

Концепция MVVM. View Model в среде iOS это **независимое** от UIKit представление **View** и ее состояния. **ViewModel** вызывает изменения в **Model** и самостоятельно обновляется с уже обновленной **Model**. И так как биндинг(связывание) происходит между **View** и **ViewModel**, то первая, соответственно, тоже обновляется.

Выделим **признаки хорошей архитектуры** для MVVM:

– **распределение**: в MVVM **View** имеет больше обязанностей, чем **View** из MVP. Потому что первая обновляет свое состояние с **ViewModel** за счет установки биндингов, тогда как вторая направляет все события в **Presenter** и не обновляет себя (это делает **Presenter**);

– **тестируемость**: **ViewModel** не знает ничего о представлении, это позволяет легко тестировать ее. **View** также можно тестировать;

– **простота использования**: в реальном приложении, где необходимо будет направлять все события из **View** в **Presenter** и обновлять **View** вручную, в MVVM будет гораздо меньше кода (если использовать 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.

Концепция VIPER. 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 необходимо 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**.

Модулем VIPER может быть один экран или целая user story вашего приложения (например, аутентификация может быть на один экран или несколько связанных экранов).

Если сравнить VIPER с паттернами MV(X)-вида, то можно выделить 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**.

Выделим **признаки хорошей архитектуры** для VIPER.

– **Распределение**. Несомненно, в VIPER компоненты разделены максимально из-за их количества.

– **Тестируемость**. Лучше распределение — лучше тестируемось.

– **Простота использования**. Необходимо писать огромное количество интерфейсов для классов с незначительными обязанностями.

В данной статье рассмотрены основные концепции проектирования мобильных приложений на базе iOS, такие как MVC, MVP, MVVM, VIPER. В проектировании iOS приложений ни одна из разобранных архитектур не будет идеально подходить для всех типов проектов. Архитектуру необходимо выбирать из доступных средств разработки, уровня подготовленности команды или разработчика, и общей сложности приложения.

Список использованных источников:

1. iOS Architecture Patterns. Demystifying MVC, MVP, MVVM and VIPER [Электронный ресурс]. – Режим доступа : https://medium.com/.