**УДК 004.273**

# СРАВНИТЕЛЬНЫЙ АНАЛИЗ ПОДХОДОВ РЕАЛИЗАЦИИ УРОВНЯ ПРЕДСТАВЛЕНИЯ ДАННЫХ МОБИЛЬНОГО ПРИЛОЖЕНИЯ

***Е.Н. ГОРОВОЙ***
**(*Представлено: М.В. ДЕКАНОВА*)**

*В данной статье рассматриваются и сравниваются основные подходы к реализации уровня представления данных мобильного приложения для операционной системы Андроид.*

**Введение.** Существует множество подходов к реализации уровня представления данных, в данной статье рассматриваются три основных: Model View Controller (MVC), Model View Presenter (MVP), а также Model View ViewModel (MVVM). Данные подходы имеют общую цель – отделить логику приложения от отображения, но реализация отличается, и выбор зачастую зависит от взглядов конкретного разработчика.

Рассмотрим понятия общие для трех подходов, а именно Model и View.

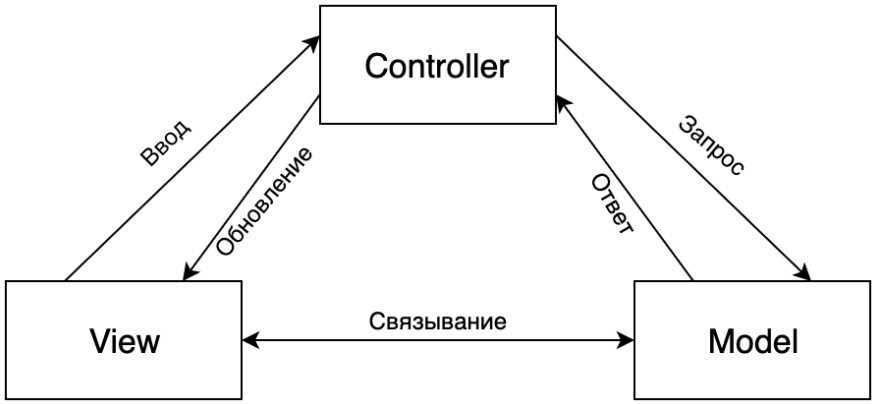
Под Моделью, обычно понимается часть, содержащая в себе функциональную бизнес-логику приложения. Модель должна быть полностью независима от остальных частей продукта. Модельный слой ничего не должен знать об элементах дизайна, и каким образом он будет отображаться. Достигается результат, позволяющий менять представление данных, то, как они отображаются, не трогая саму Модель.

В обязанности Представления входит отображение данных, полученных от Модели. Однако, представление не может напрямую влиять на модель. Можно говорить, что представление обладает доступом «только на чтение» к данным.

**Model View Controller**. Подход MVC можно охарактеризовать двумя пунктами:

- Представление — это визуальная проекция модели;
- Контроллер — это соединение между пользователем и системой.

Ниже представлена диаграмма, иллюстрирующая идеологию подхода (рисунок 1).




.

**Рисунок 1. – Диаграмма подхода MVC**

Характеризующим элементом данного подхода является то, что представление связано с моделью, а основной идеей – разделить вводом и выводом данных. Контроллер принимает входные данные, а представление – выходные, однако большое число операций происходит и между ними [1].

Из плюсов можно выделить малое количество кода, по сравнению с другими подходами. Из минусов – слабое разделение ответственности, сложная инкапсуляция, сложное юнит-тестирование.

Данный подход хорошо подходит только для небольших проектов так как очень плохо масштабируется.

**Model View Presenter.** Решение этих проблем кроется за созданием абстрактного интерфейса для представления. Таким образом данная архитектура облегчает юнит-тестирование, а также может многократно использоваться, потому что представление может реализовать несколько интерфейсов.

Характеризующим элементом данного подхода является то, что view закрыта интерфейсом от presenter'а, а главной идеей – сделать presenter независимым от view. Диаграмма идеологии подхода MVP представлена на рисунке 2.

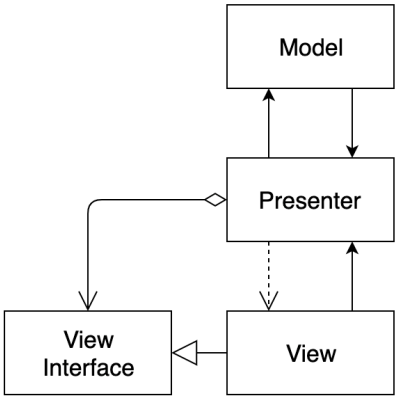



**Рисунок 2. – Диаграмма подхода MVP**

Каждое представление должно реализовывать соответствующий интерфейс. Интерфейс представления определяет набор функций и событий, необходимых для взаимодействия с пользователем. Презентер должен иметь ссылку на реализацию соответствующего интерфейса. Логика представления должна иметь ссылку на экземпляр презентера. Все события представления передаются для обработки в презентер и практически никогда не обрабатываются логикой представления (в т.ч. создание других представлений) [2].

Таким образом можно выделить следующие плюсы: презентер легко тестируется, презентер не зависит от представления, таким образом, презентер можно повторно использовать для других представлений.

Из минусов: необходимость создавать и поддерживать интерфейсы для представлений и лишний шаблонный код.

Данная архитектура хорошо масштабируется, тестируется, и хорошо подходит для больших проектов.

**Model View ViewModel**. ViewModel так же, как и презентер, содержит логику пользовательского интерфейса. Когда пользователь нажимает на кнопку, это событие направляется в ViewModel, который затем решает, что с ним делать. ViewModel может преобразовывать данные из модели так, чтобы они были легко отображены на экране. Часто информация, содержащаяся в модели, не может непосредственно использоваться на экране. Это наиболее вероятно, когда у вас нет полного контроля над моделью. Например, если вы получаете данные от сторонних веб-сервисов или же из базы данных существующего приложения[3].

Основным отличием данного подхода является то, что ViewModel не может общаться со View напрямую. Вместо этого она представляет легко связываемые свойства и методы в виде команд. View может привязываться к этим свойствам, чтобы получать информацию из ViewModel и вызывать на ней команды (методы). Это не требует того, чтобы View знала о ViewModel.

Плюсы подхода: легко тестируется и масштабируется, View и ViewModel не зависят друг от друга.

Минусы: возрастает сложность кода, так как методы, в отличии от MVP, вызываются не на прямую, а происходит подписка на изменение состояния, или отправленные команды.

Данный подход также подходит для больших проектов, так как легко масштабируется и имеет хорошую тестируемость.

**Заключение**. В заключении стоит отметить, что строго придерживаться только одному подходу – не всегда лучший выбор, ведь главная цель – это отделить представление от бизнес-логики и логики, которая их связывает. Несмотря на то, что внедрение разделения ответственности требует усилий, это хороший способ повысить качество кода в целом, сделать его масштабируемым, легким в понимании и надежным.

ЛИТЕРАТУРА

1. Model-View-Controller. [Электронный ресурс]. Режим доступа: https://ru.wikipedia.org/wiki/Model-View-Controller. – Дата доступа: 16.09.2019.
2. Common Android Architectures (MVC vs MVP vs MVVM) [Электронный ресурс]. Режим доступа: https://medium.com/@mr.anmolsehgal/common-android-architectures-mvc-vs-mvp-vs-mvvm-afd8461e1fee. – Дата доступа: 17.09.2019.
3. MVP vs MVVM: A Review of Patterns for Android. [Электронный ресурс]. Режим доступа: https://thinkmobiles.com/blog/mvp-vs-mvvm-android-patterns/. – Дата доступа: 17.09.2019.