УДК 004.415.2

СОВРЕМЕННЫЙ ПОДХОД К РЕАЛИЗАЦИИ АРХИТЕКТУРНОГО ПАТТЕРНА В ANDROID-ПРИЛОЖЕНИЯХ

А. М. Сарматин

Высшая школа информационных технологий и интеллектуальных систем Казанского (Приволжского) федерального университета

antonsarmatin@googlemail.com

Аннотация

Рассмотрены архитектурные паттерны, использующиеся в разработке Android-приложений, описаны их слабые и сильные стороны, особенности при использовании с Android-фреймворком. Предложен обновленный подход, который позволяет устранить недостатки существующих шаблонов. Сформулирована концепция архитектурного фреймворка для разработки Android-приложений, реализующего предложенный подход.

Ключевые слова: *android, architecture, mvvm, mvp, mvi, mvc, presentation, архитектура, мобильные приложения, фреймворк, библиотека, разработка.*

ВВЕДЕНИЕ

Построение архитектуры мобильного приложения и выбор архитектурного паттерна зависят от многих факторов, к которым относятся принятые в компании стандарты разработки, специфика проекта или опыт работы с тем или иным подходом. С точки зрения разработки мобильного приложения, применение определенного шаблона позволяет добиться масштабируемости, сопровождаемости и надежности.

Архитектурные паттерны – часть общей архитектуры мобильного приложения, но одно их использование позволяет избежать многих проблем при внедрении нового или изменении текущего функционала, а также на этапе разработки и при покрытии кода тестами. Несмотря на то, что каждый из паттернов имеет свои сильные и слабые стороны, в первую очередь, все они нацелены на достижение одних и тех же целей.

Задачей архитектурных шаблонов является разделение ответственности

представления между различными компонентами системы. Без разделения ответственности невозможно достичь масштабируемости и тестируемости.

В MV* паттернах имеются общие части, это View и Model. View отвечает за отображение, то есть вывод данных полученных от Model на экран. На примере системы Android это могут быть Activity или Fragment. Model – абстрактное понятие, под которым скрываются и сами данные, и множество других компонентов архитектуры, позволяющих получить эти данные для дальнейшего их отображения. Третий компонент системы зависит от выбранного архитектурного паттерна и отвечает за связь View и Model, а также способ передачи действий и данных между ними.

Среди MV* паттернов в разработке мобильных приложений можно встретить такие паттерны, как MVP, MVVM, MVI и MVC.

СУЩЕСТВУЮЩИЕ АРХИТЕКТУРНЫЕ ПАТТЕРНЫ

MVC (Model-View-Controller) – один из самых известных паттернов проектирования в сфере разработки программного обеспечения, однако в разработке мобильных приложений для операционной системы Android встречается достаточно редко ввиду особенностей системы.

Если рассматривать реализацию такого паттерна при разработке для Android, то можно столкнуться с проблемой, что компоненты Activity и Fragment, одновременно являются и View, и Controller (рис. 1), то есть для реализации паттерна MVC поверх существующих компонентов системы требуется выносить Controller в отдельный класс, а сами компоненты системы рассматривать как отображение (рис. 2). В таком случае View будет передавать все действия пользователя в Controller, который будет работать с Model и передавать команды View [1, 2].

Признаки MVC:

- View напрямую взаимодействует с Controller;
- Controller напрямую воздействует на View.

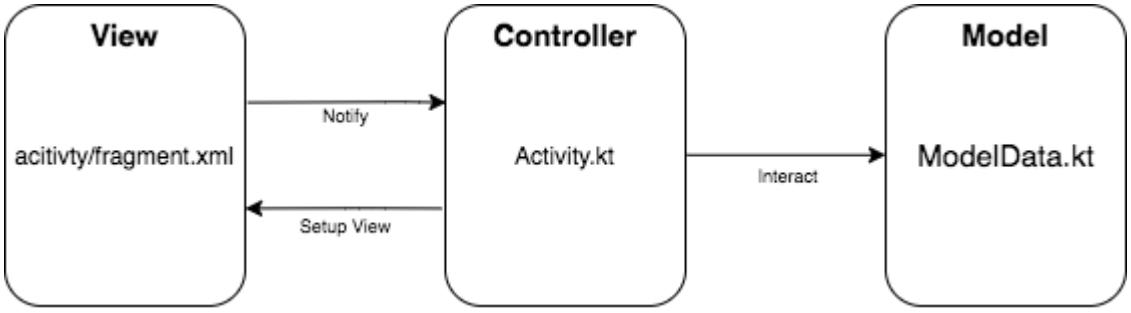

Рисунок 1. Android MVC

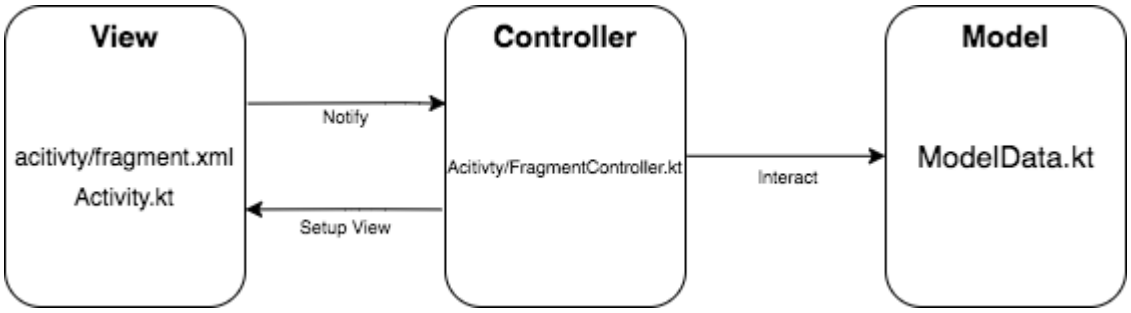

Рисунок 2. MVC с отдельным Controller-классом

В итоге логика, вынесенная в отдельный Controller, позволяет разгрузить Activity/Fragment от лишнего кода по подготовке данных к представлению, но при этом сохраняется жесткое связывание View и Controller, что мешает масштабируемости и тестированию.

Таким образом, в качестве достоинств описанного подхода можно выделить низкий порог вхождения и малое количество шаблонного кода. Недостатками являются жесткое связывание View и Controller, слабое разделение ответственности и сложное unit-тестирование.

MVP (Model–View–Presenter) – самый популярный подход к реализации архитектурного паттерна в приложениях для ОС Android. Данный паттерн позволяет облегчить unit-тестирование при помощи внедрения абстракции между View-слоем и Presenter-классом. Presenter обращается к интерфейсу IView, который реализуется View, то есть компонентом Android – Activity или Fragment (рис. 3). Также это позволяет снова использовать Presenter в различных Activity или Fragment благодаря тому, что каждый из них может реализовывать интерфейс IView.

Признаки MVP:

- двухсторонняя коммуникация с представлением;

- View напрямую воздействует на Presenter путем вызова соответствующих функций у экземпляра Presenter;
- Presenter взаимодействует с View через интерфейс IView, реализованный View, ссылку на который Presenter хранит в себе.

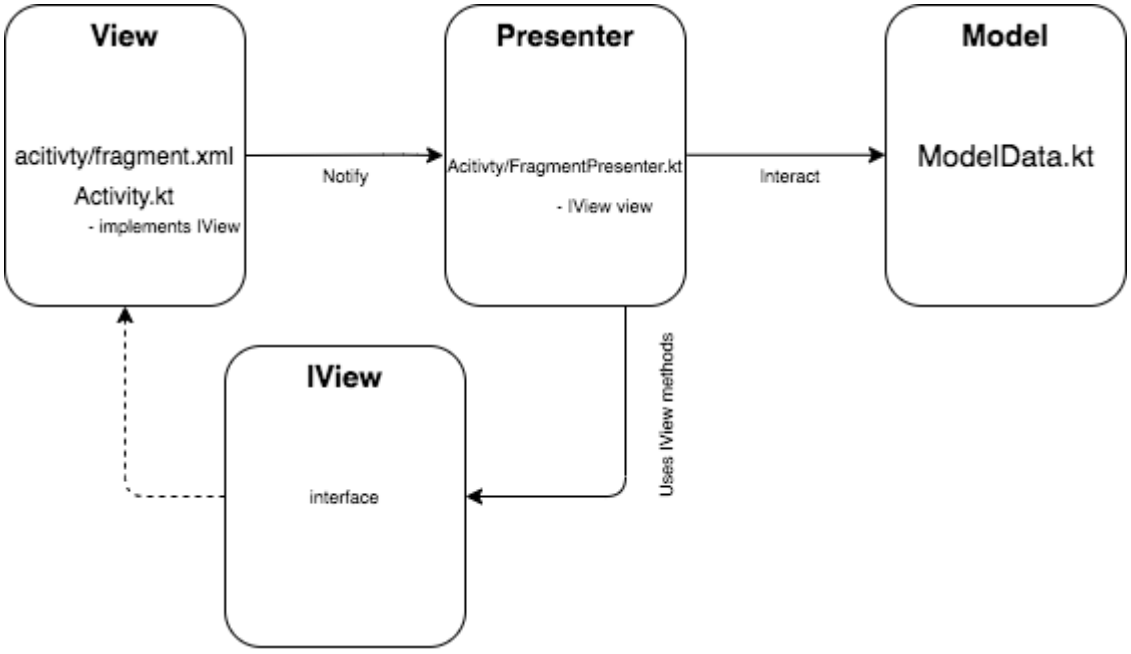

Рисунок 3. MVP

Однако введение дополнительной абстракции увеличивает количество требуемого кода для реализации паттерна MVP, что в свою очередь ведет к усложнению масштабируемости и поддержки. Еще одним минусом данного паттерна является неконсистентность состояния данных, это возникает из-за того, что View может влиять на Presenter через множество функций, а Presenter воздействует на View через интерфейс IView при помощи множества функций. Это вызывает проблему тестирования состояния View.

Среди достоинств данного паттерна можно отметить тестируемость и возможность повторного использования Presenter, а также тот факт, что Presenter не зависит от реализации View. В качестве недостатков могут быть рассмотрены необходимость поддерживать не только Presenter, но и интерфейс IView и его реализацию в Activity/Fragment, большое количество шаблонного кода и сложность отслеживания состояния View из-за неконсистентности данных.

MVVM (Model–View–ViewModel) – набирающий популярность подход, рекомендуемый для использования в разработке для ОС Android и поддерживаемый компанией Google при помощи программных библиотек для быстрой реализации данного паттерна [3].

Этот паттерн позволяет связывать элементы View со свойствами ViewModel, при этом ViewModel не имеет ссылки на представление View. Изменение свойств ViewModel автоматически изменяет View, а действия View изменяют свойства ViewModel, это достигается при помощи механизма связывания данных (Data Binding) (рис. 4). В разработке для Android это достигается при помощи использования паттерна Observer, при помощи которого View подписывается на изменение свойств в ViewModel.

Признаки MVVM:

- ViewModel не имеет ссылки на View;
- View следит за изменениями данных в ViewModel;
- View может изменить данные в ViewModel.

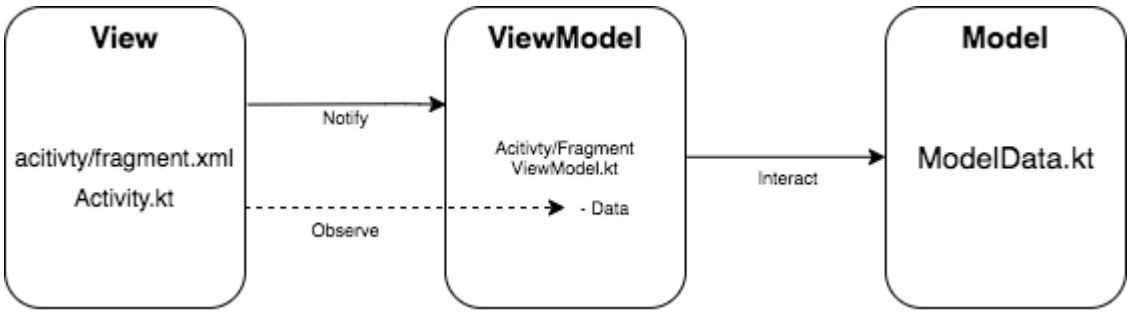

Рисунок 4. MVVM

В отличие от MVP использование MVVM позволяет избежать лишнего шаблонного кода за счет отсутствия дополнительной абстракции для связи с View, но для решения проблемы жесткой связности из MVC требуется использовать другой механизм связывания данных путем наблюдения изменений во ViewModel, то есть ViewModel содержит определенный набор свойств, за каждым изменением которых наблюдает View и отображает их. Таким образом, получается, что ViewModel служит отдельной абстракцией представления и содержит свойства, совпадающие со свойствами View. Однако это не решает проблему консистентности

данных, каждое свойство View наблюдает за соответствующими свойствами ViewModel независимо от других свойств View, что ведет к сложности отслеживания и тестирования состояния View в целом.

MVI (Model–View–Intent) – паттерн, реализующий unidirectional data flow (UDF; однонаправленный поток данных).

Суть этого паттерна заключается в том, что View изменяется только под воздействием некого состояния (State), при этом View воздействует на Model при помощи намерения (Intent или Action в Android), попадающего в Reducer, который затем изменяет State. Это позволяет достичь однонаправленного потока данных и иметь единую точку входа и выхода.

Данный паттерн возможно реализовать, используя принципы любого представленного ранее паттерна, например, MVP и MVVM (рис. 5 и 6), разница лишь в том, как View получит измененный State [4].

Признаки MVI:

- View сообщает о событиях в класс (Presenter/VIewModel), который содержит в себе Reducer;
- View реализует функцию, которая получает на вход State;
- State содержит себе всю информацию, которая описывает состояние экрана.

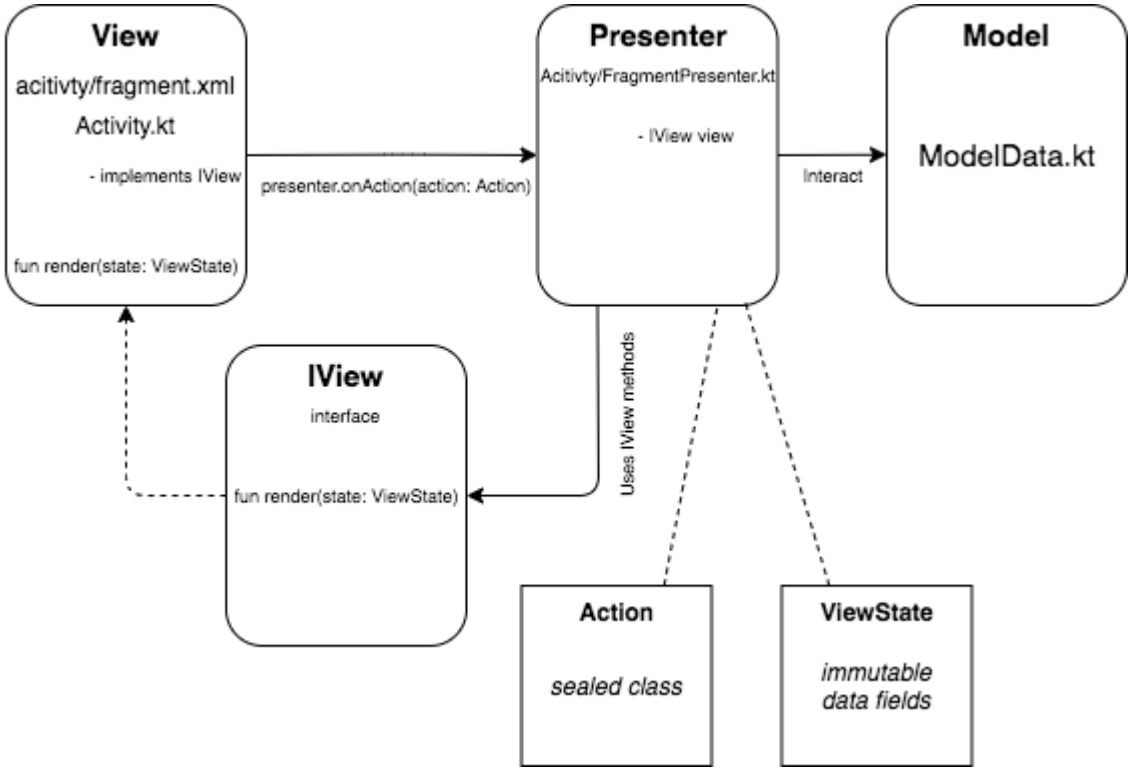

Рисунок 5. MVI на MVP

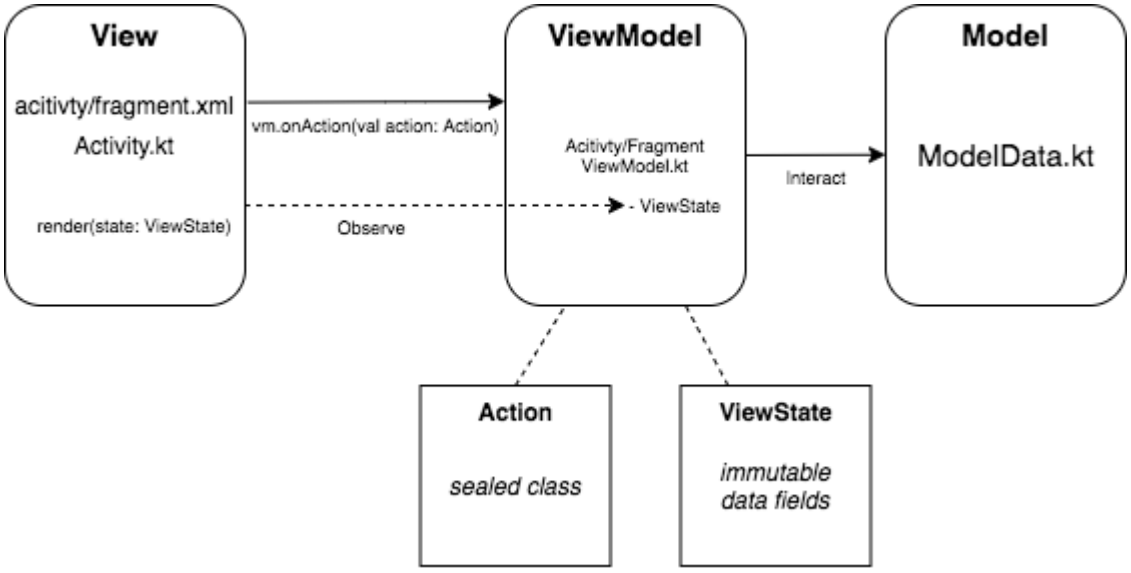

Рисунок 6. MVI на MVVM

Использование единого состояния влечет за собой ряд негативных последствий. Например, на любое действие требуется полностью пересоздать состояние, даже если данные были изменены минимально либо не были изменены. При этом каждое такое состояние будет передано во View, который отреагирует

на пришедшие данные, даже если они не были изменены. В случае с Android это вызовет перерисовку UI элементов либо потребует отслеживания изменений между прошлым состоянием и нынешним. Другой проблемой состояния является его размер, так как количество свойств состояния растет с увеличением функционала конкретного экрана приложения. Еще одной проблемой является передача каких-либо событий во View, которые должны быть воспроизведены однократно, не затрагивая само представление данных на экране (например, всплывающие окна).

Исходя из анализа различных, представленных выше архитектурных паттернов, можно сделать вывод, что каждый из них имеет свои плюсы и минусы и может найти свое применение в том или ином проекте, однако существуют определенные подходы к их использованию, помогающие внедрить определенный паттерн в платформу с учетом ее специфики [5].

ПОДХОДЫ К ИСПОЛЬЗОВАНИЮ АРХИТЕКТУРНЫХ ПАТТЕРНОВ

Рассмотренные выше архитектурные паттерны могут использоваться при разработке ПО для разных платформ, и конкретная реализация будет зависеть от особенностей платформы.

К особенностям ОС Android, которые следует учитывать при разработке ПО с использованием одного из описанных архитектурных паттернов, можно отнести:

- жизненный цикл компонентов Android;
- сохранение состояния данных;
- навигация между различными экранами приложения;
- многопоточность при работе с данными и их отображением.

Реализация любого из представленных выше паттернов внутри приложения или экрана является объемной задачей, которая потребует большого количества времени на написание шаблонного кода, а также повысит вероятность допустить ошибку, что влечет за собой лишние затраты на тестирование тех частей, которые напрямую не связаны с приложением.

Для решения этой проблемы при реализации паттернов существуют библиотеки и фреймворки, которые избавляют от написания шаблонного кода, учиты-

вают особенности ОС Android, предоставляют удобный инструментарий для реализации паттерна с дополнительными возможностями. К ним можно отнести такие программные инструменты, как Moxy, MVICore, Cicerone, Android Jetpack Architecture Components.

Moxy – реализация MVP, избавляющая от написания шаблонного кода, имеет встроенный механизм обработки жизненного цикла View компонента и сохранение состояния данных и действий при помощи ViewState [6].

MVICore – реализация MVI, избавляющая от написания шаблонного кода и реализации State, Reducer, Middleware (Interactor) и других компонентов паттерна для работы в реактивном виде, имеет встроенный механизм обработки жизненного цикла и сохранения состояния данных.

Cicerone – библиотека для реализации навигации внутри приложения, созданная, в первую очередь, для работы с приложением, использующим паттерн MVP, имеет встроенный механизм обработки жизненного цикла, предоставляет возможность Unit-тестирования навигации, позволяет избежать шаблонного кода при осуществлении навигации между экранами [7].

Android Jetpack Architecture Components [8] – это набор различных зависимостей, предоставляющих инструменты для продвинутой разработки приложений для ОС Android. Одними из таких компонентов являются ViewModel и LiveData, а также Lifecycles, с их помощью можно реализовать паттерн MVVM и его базовый функционал [9]. При этом такой вариант реализации будет обрабатывать жизненный цикл View-компонента, сохранять свое состояние и состояние данных. LiveData позволит реализовать связывание данных между View и ViewModel при помощи паттерна Observable. Еще одним из компонентов является Navigation, предоставляющий возможность простой навигации между экранами при помощи построения графа навигационных связей [10].

ОПИСАНИЕ ИДЕИ ОБНОВЛЕННОГО ПОДХОДА

Нашей базовой идеей является создание архитектурного фреймворка, сочетающего в себе не только реализацию MVVM-паттерна с решением его недостатков, но также и другие компоненты, целью которых является обеспечить эффективную разработку приложений для операционной системы Android с использованием архитектурных решений для достижения масштабируемости, сопровождаемости и надежности. Компоненты данного фреймворка можно разделить на четыре категории, в дальнейшей эти категории будут называться компонентами фреймворка (рис. 7):

- *Presentation* – реализация MVVM-паттерна с централизованной обработкой состояний экрана и данных;
- *Navigation* – реализация навигации между экранами, основанная на состояниях и не имеющая прямой зависимости от View;
- *Architecture* – инструменты для разработки приложения, следующие рекомендациям построения чистой архитектуры [11];
- *Utils* – набор инструментов и классов для разработки.

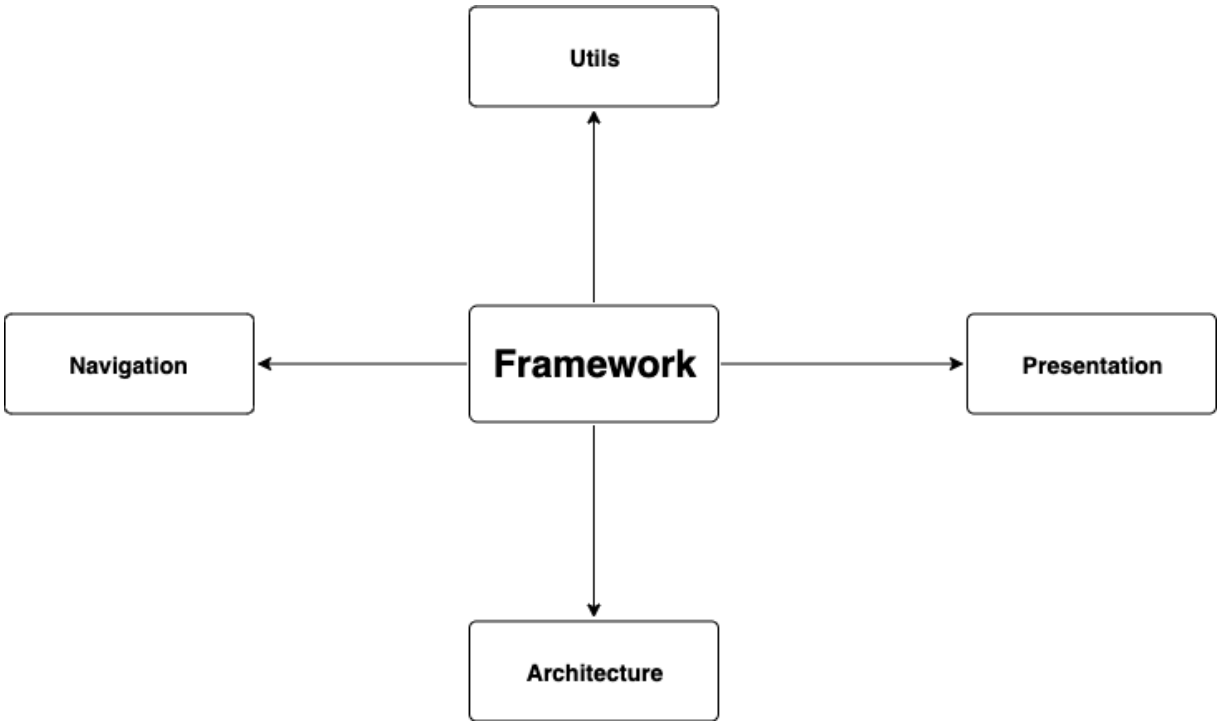

Рисунок 7. Компоненты фреймворка

Между собой эти компоненты связаны зависимостями (рис. 8), соответственно, использование определенного компонента, зависящего от другого, без его использования будет невозможно.

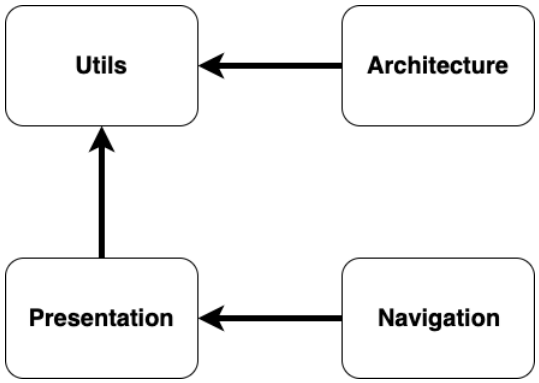

Рисунок 8. Зависимости между компонентами фреймворка

Основной идеей применения компонента Presentation служит реализация паттерна MVVM для операционной системы Android с учетом недостатков абстрактной реализации и возможностью расширения функционала. Использование Android Jetpack в качестве основы для разработки данного компонента фреймворка позволит достичь высокой интеграции с нативными компонентами платформы Android, что, в свою очередь, увеличивает надежность и поддерживаемость как самого разрабатываемого приложения, так и фреймворка, использованного при его разработке.

Особенности компонента Presentation:

- избавление от шаблонного кода для сохранения данных и состояния ViewModel;
- централизованная обработка базовых состояний (ошибка, загрузка, сообщение);
- возможность реализации собственных состояний ViewModel путем расширения базового состояния или создания полностью независимого состояния;
- разделение состояния данных и сами данных;
- возможность производить навигацию, основываясь на состояниях с дальнейшей интеграцией с навигационным компонентом фреймворка.

Сохранение состояния ViewModel и обработка жизненного цикла реализуются во ViewModel благодаря наследованию от класса ViewModel из Jetpack, который в свою очередь реализует интерфейс LifecycleObserver, который получает события изменения жизненного цикла от LifecycleOwner. В случае с MVVM в качестве View выступают Activity или Fragment, который реализует интерфейс LifecycleOwner. Это позволяет обрабатывать события жизненного цикла от View и совершать какие-либо действия с данными или состоянием, отталкиваясь от текущего состояния жизненного цикла [12].

Сохранение данных реализуется при помощи передачи во ViewModel ссылки на хранилище SavedStateHandle, которое представляет собой хранилище уникальных пар «ключ – значение» и способен сохранять любые объекты, которые реализуют интерфейсы Parcelable или Serializable. Стоит учитывать тот момент, что данное хранилище не зависит от жизненного цикла View-компонента, но зависит от жизненного цикла Application и не является персистентным, то есть оно служит лишь для кеширования или сохранения данных в рамках рантайма приложения, например, при навигации между экранами и возвратом назад либо переворотом экрана, что вызывает пересоздание родительской Activity, значит, и View.

Для реализации многопоточности ViewModel имеет отдельный Scope для запуска корутин. Корутины – реализация многопоточности в языке Kotlin, позволяющая избавиться каждый раз от затрат на запуск отдельных потоков и вместо этого использовать набор потоков и запускать задачи на нем.

Главным недостатком MVVM-паттерна является неконсистентность состояний данных, что может привести к некорректному отображению данных или элементов интерфейса, связанных с этими данными.

Частный пример такой проблемы представлен на рис. 9, на нем представлена трансформация данных из состояния 1 в другие состояния. При этом почти одновременно запускаются два процесса – основной и побочный, которые производят работу с данными, находящимися в состоянии 1. Однако основной процесс закончил свою работу раньше и перевел данные в состояние 2, после которого запустился еще один основной процесс работы с данными, которые находятся в состоянии 2, переводящий их в состояние 3 и завершившийся раньше побочного

процесса, который еще был начат, когда данные находились в состоянии 1. Когда побочный процесс начинал свою работу, он оперировал данными в состоянии 1, а на момент его завершения данные находятся в другом состоянии, которое будет изменено на то состояние, в которое перевел данные этот побочный процесс. При этом сами данные могут быть утратившими ценность или неактуальными для пользователя.

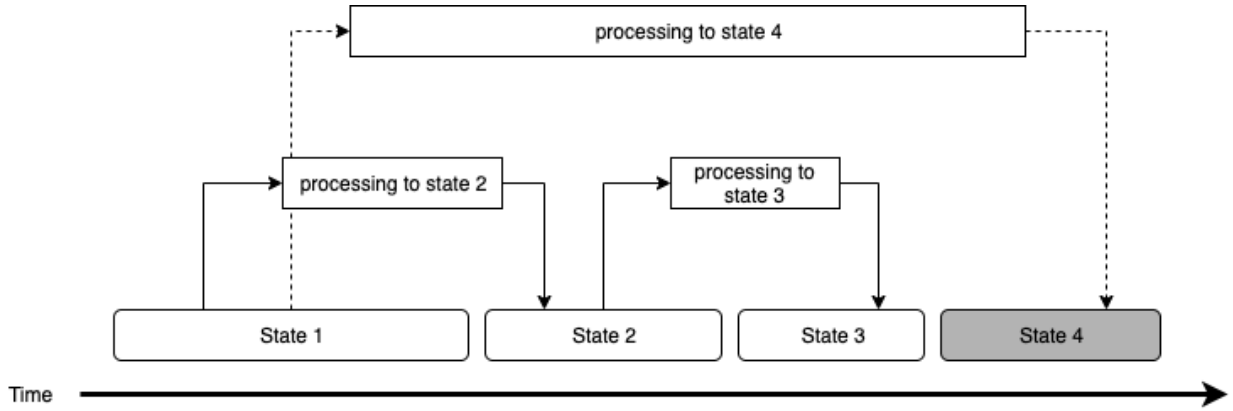

Рисунок 9. Пример некорректного состояния данных

Для решения названной проблемы может быть предложено несколько решений, одно из которых – сохранение состояния данных, например, из MVI. Такой подход тоже несет в себе серьезные недостатки, одним из которых является то, что сами данные также включаются в глобальное состояние. Это ведет к проблемам поддержки такого состояния, которое требует своего пересоздания на каждое действие, при котором будет происходить процесс перерисовки всех элементов экрана, которые зависят от состояния или данных в нем. Такого поведения можно избежать, если реализовывать механизм, отслеживающий, изменилось ли конкретное поле в состоянии, и отдающий команды элементу экрана на обновление.

Решением проблемы неконсистентности данных при использовании паттерна MVVM может служить подход с состоянием, которое напрямую не несет в себе данных и разделяется на два подсостояния: CommonState и FeatureState, где первое является общим состоянием экрана, а второе является состоянием экрана и данных на нем. CommonState содержит в себе данные, общие для всех экранов

приложения, такие, как загрузка, ошибки, сообщения и т. д. FeatureState содержит в себе ту информацию, которая укажет, какие данные и в каком состоянии находятся, благодаря которой View-компонент будет иметь возможность обрабатывать и отображать только те данные, которые требуются в текущем состоянии. Значит stateObserver во View-компоненте ответственен за то, чтобы обработать пришедший ему ViewState, обработать CommonState и FeatureState и, опираясь на последний, решить, на какие данные из ViewModel требуется подписаться сейчас (рис. 10).

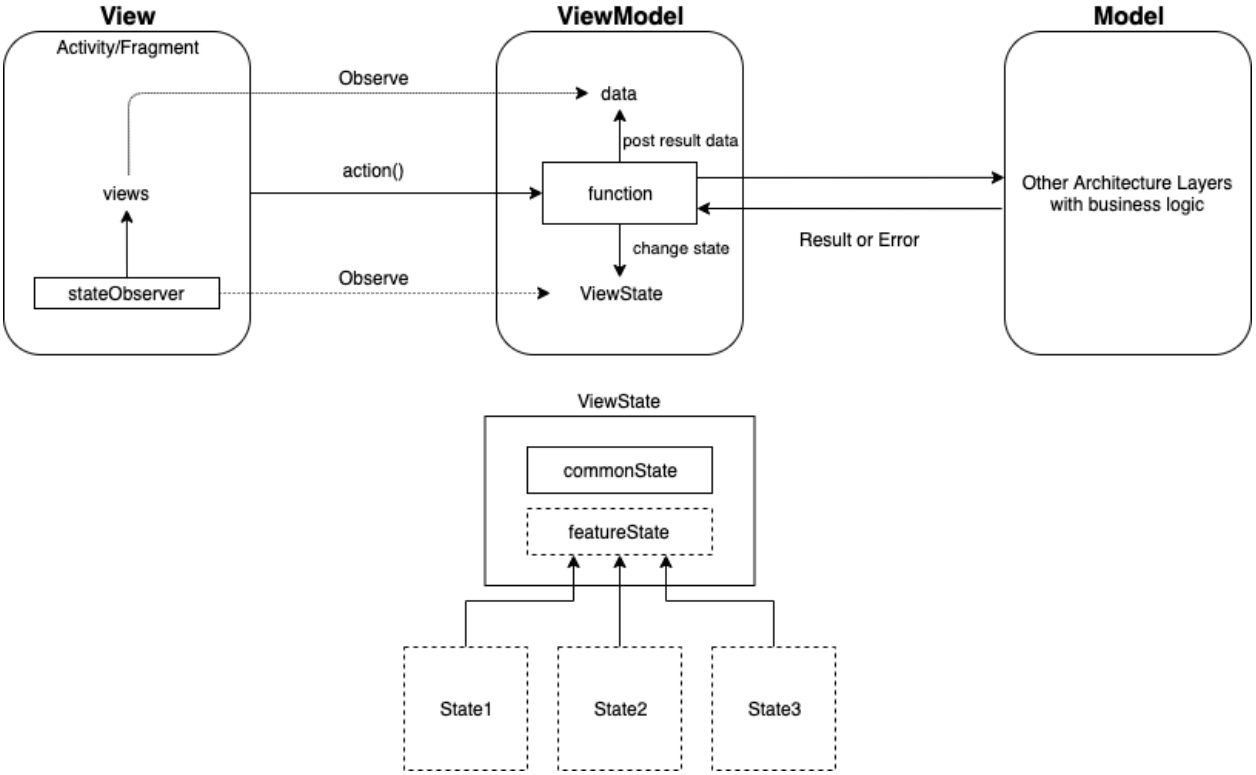

Рисунок 10. Обработка состояний на базе MVVM без включения данных

CommonState позволяет добиться централизованной обработки ошибок и состояния загрузки, содержав в себе поля типов Failure и Loading, соответствующие этим состояниям, а также дефолтным функциям обработки этих состояний во View-компоненте. При этом такой подход позволяет сохранить гибкость при помощи наследования от Failure или Loading, а также переопределения базовых функций для обработки кастомных событий.

Действия со стороны View могут поступать как через единую точку входа,

аналогично MVI, так и путем вызова различных функций ViewModel, ответственных за конкретное действие. При этом любое действие внутри ViewModel, оказывающее влияние на состояние, будет зависеть от начального состояния, состояния, в котором оно было завершено, и состояния, готового к изменениям текущего.

ЗАКЛЮЧЕНИЕ

В статье проведен анализ преимуществ и недостатков существующих архитектурных паттернов в разработке программного обеспечения, в частности, мобильных приложений для операционной системы Android, и подходов к их использованию. На основании полученных данных описана идея разработки фреймворка для разработки мобильных приложений для операционной системы Android, включающая в себя компонент с реализацией паттерна MVVM с решением его недостатков, а также дополнительным функционалом, увеличивающим эффективность разработки Android-приложений.

СПИСОК ЛИТЕРАТУРЫ

1. *Marcin Moskala*. MVC vs MVP vs MVVM vs MVI. URL: https://academy.realm.io/posts/mvc-vs-mvp-vs-mvvm-vs-mvi-mobilization-moskala/.

2. *MVC vs MVP vs MVVM*. URL: https://habr.com/ru/post/215605/.

3. *Guide to App Architecture*. URL: https://developer.android.com/jetpack/docs/guide.

4. *Zsolt Kocsi*. MVI beyond state reducers. URL: https://badootech.badoo.com/a-modern-kotlin-based-mvi-architecture-9924e08efab1

5. *Frederick P. Brooks, Jr.* No Silver Bullet-Essence and Accident in Software Engineering. URL: http://worrydream.com/refs/Brooks-NoSilverBullet.pdf.

6. *Moxy*. URL: https://habr.com/ru/post/276189/.

7. *Cicerone*. URL: https://habr.com/ru/company/mobileup/blog/314838/.

8. *Android Architecture Components*. URL: https://developer.android.com/topic/libraries/architecture.

9. *Android ViewModel Overview.* URL: https://developer.android.com/topic/ libraries/architecture/viewmode.

10. *Android Navigation Component*. URL: https://developer.android.com/guide/navigation.

11. *Robert C. Martin.* The Clean Code Blog. URL: https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html.

12. *Handling Lifecycles with Lifecycle-Aware Components.* URL: https://developer.android.com/topic/libraries/architecture/lifecycle.

MODERN IMPLEMENTATION OF THE DESIGN PATTERN IN ANDROID APPLICATIONS

A. M. Sarmatin

Higher School of Information Technology and Intelligent Systems, Kazan (Volga region) Federal University

antonsarmatin@googlemail.com

Abstract

Presentations patterns used in android application development are analyzed. Pros and cons of implementation of presentation patterns with android framework requirements are described. The idea of an architectural framework for android development is proposed.

Keywords: *android, architecture, mvvm, mvp, mvi, mvc, presentation, architecture, mobile applications, framework, library, development*

REFERENCES

1. *Marcin Moskala*. MVC vs MVP vs MVVM vs MVI. URL: https://academy.realm.io/posts/mvc-vs-mvp-vs-mvvm-vs-mvi-mobilization-moskala/.

2. *MVC vs MVP vs MVVM*. URL: https://habr.com/ru/post/215605/.

3. *Guide to App Architecture*. URL: https://developer.android.com/jetpack/docs/guide.

4. *Zsolt Kocsi*. MVI beyond state reducers. URL: https://badootech.badoo.com/a-modern-kotlin-based-mvi-architecture-9924e08efab1

5. *Frederick P. Brooks, Jr.* No Silver Bullet-Essence and Accident in Software

Engineering. URL: http://worrydream.com/refs/Brooks-NoSilverBullet.pdf.

6. *Moxy*. URL: https://habr.com/ru/post/276189/.

7. *Cicerone*. URL: https://habr.com/ru/company/mobileup/blog/314838/.

8. *Android Architecture Components*. URL: https://developer.android.com/topic/libraries/architecture.

9. *Android ViewModel Overview.* URL: https://developer.android.com/topic/ libraries/architecture/viewmode.

10. *Android Navigation Component*. URL: https://developer.android.com/guide/navigation.

11. *Robert C. Martin.* The Clean Code Blog. URL: https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html.

12. *Handling Lifecycles with Lifecycle-Aware Components.* URL: https://developer.android.com/topic/libraries/architecture/lifecycle.

СВЕДЕНИЯ ОБ АВТОРЕ

САРМАТИН Антон Михайлович – студент Высшей школы информационных технологий и интеллектуальных систем Казанского (Приволжского) федерального университета, Андроид-разработчик.

Anton Mikhailovich SARMATIN – student of the Higher School of Information Technologies and Intelligent Systems at Kazan Federal University, Android developer.

email: antonsarmatin@googlemail.com

Материал поступил в редакцию 12 апреля 2020 года