FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of master's thesis

Title:	Comparison of different architecture approaches on Android OS
Student:	Azad Mamiyev
Supervisor:	Ing. Petr Špaček, Ph.D.
Study program:	Informatics
Branch / specialization:	Web and Software Engineering
Department:	Department of Software Engineering
Validity:	until the end of summer semester 2021/2022

Instructions

The objective of the thesis is to compare various architecture approaches in mobile operational system Android and find advantages and disadvantages of each implementation. In addition, show how to implement design patterns in modern android development.

Tasks:
1. Get familiar with Design Patterns using in Android Development.
2. Discuss MVP Design Pattern.
3. Discuss MVVM Design Pattern.
4. Discuss MVI Design Pattern.
5. Implement identical apps using theses patterns.
6. Perform unit-testing on all 3 outputs for major use-cases of the app
7. Discuss the results of implementing these patterns.

Electronically approved by Ing. Michal Valenta, Ph.D. on 13 October 2020 in Prague.

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Master’s thesis

Comparison of different architecture approaches on Android OS

Bc. Azad Mamiyev

Department of theoretical computer science
Supervisor: Ing. Špaček Petr Ph.D.

May 3, 2021

Acknowledgements

I would like to thank my family and friends for support during writing this thesis.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. In accordance with Article 46 (6) of the Act, I hereby grant a nonexclusive authorization (license) to utilize this thesis, including any and all computer programs incorporated therein or attached thereto and all corresponding documentation (hereinafter collectively referred to as the "Work"), to any and all persons that wish to utilize the Work. Such persons are entitled to use the Work in any way (including for-profit purposes) that does not detract from its value. This authorization is not limited in terms of time, location and quantity. However, all persons that makes use of the above license shall be obliged to grant a license at least in the same scope as defined above with respect to each and every work that is created (wholly or in part) based on the Work, by modifying the Work, by combining the Work with another work, by including the Work in a collection of works or by adapting the Work (including translation), and at the same time make available the source code of such work at least in a way and scope that are comparable to the way and scope in which the source code of the Work is made available.

In Prague on May 3, 2021 . . . .

Czech Technical University in Prague
Faculty of Information Technology
© 2021 Azad Mamiyev. All rights reserved.
This thesis is school work as defined by Copyright Act of the Czech Republic. It has been submitted at Czech Technical University in Prague, Faculty of Information Technology. The thesis is protected by the Copyright Act and its usage without author's permission is prohibited (with exceptions defined by the Copyright Act).

Citation of this thesis

Mamiyev, Azad. *Comparison of different architecture approaches on Android OS*. Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2021. Also available from: ⟨http://site.example/thesis⟩.

Abstrakt

Model View Presenter, Model View ViewModel a Model View Intent patří mezi tři nejpopulárnější a nejvíce probírané softwarové architektury zaměřené na návrh aplikací s grafickým uživatelským rozhraním. Tyto architektury se prosadily i ve světě Android aplikací.

Tato práce si klade za cíl poskytnout důkladnou analýzu architektonických vzorů MVP, MVVM a MVI. Metodou srovnání těchto vzorů v oblastech: implementační náročnost, testovatelnosti, složitosti a nároků na údržbu, se pokouší zjistit, který vzor je nejvhodnější na užití pro vývoj Android aplikací s uživatelským rozhraním. Pro nalezení odpovědi na tuto otázku implementujeme třikrát stejný ukázkový modulární projekt, pokaždé s využitím jiné ze tří zkoumaných architektur.

Klíčová slova Android, Android architektura aplikace, MVP, MVVM, MVI, Vícemodulový projekt, Implementace, Testovatelnost, Složitost, Udržitelnost.

Abstract

Model View Presenter, Model View ViewModel and Model View Intent are three the most popular and well-studied software architectures which are

widely used in GUI-heavy applications, these architectures have now also emerged in Android development.

This thesis aims to provide a thorough analysis of MVP, MVVM and MVI architecture patterns. We make a comparison of these patterns in implementation complexity, testability and maintainability to figure out which pattern or patterns are the best in the modern world. We are going to implement the multi-module project written three times using selected architecture patterns to answer this question.

Keywords Android, Android App Architecture, MVP, MVVM, MVI, Multi-module Project, Implementation, Testability, Complexity, Maintainability.

Contents

Introduction **1**
Development background . . . 2
Motivation and current situation with patterns . . . 2
Structure . . . 3

1 Architecture Patterns **5**
1.1 MVP Model-View-Presenter . . . 5
1.1.1 MVP Architecture . . . 5
1.1.2 MVP realization in Android development . . . 7
1.2 MVVM Model-View-ViewModel . . . 8
1.2.1 MVVM Architecture . . . 8
1.2.2 MVVM realization in Android development . . . 10
1.3 MVI Model-View-Intent . . . 11
1.3.1 MVI Architecture . . . 11
1.3.2 MVI realization in Android development . . . 12

2 Use-Cases of the application **15**
2.1 A list of trending movies . . . 15
2.2 Open movie details . . . 16
2.3 Save the movie . . . 17
2.4 Search functionality . . . 18
2.5 Saved movies screen . . . 19
2.6 Delete the saved movie . . . 19
2.7 Delete all saved movies . . . 20

3 Configuration stage **23**
3.1 One project vs Several projects . . . 23
3.2 Configuration of Multi-Module project . . . 24

4 Implementation process **27**

4.1 Common module . . . 27
4.2 MVP Implementation . . . 32
4.3 MVVM Implementation . . . 39
4.4 MVI Implementation . . . 43

5 Comparison **49**
5.1 General details of comparison . . . 49
5.2 Implementation Complexity . . . 50
5.3 Maintainability . . . 50
5.4 Testability . . . 51

6 Conclusion **53**

Bibliography **55**

A Sources **59**

B List of abbreviations **61**

List of Figures

1.1 MVP Components 5
1.2 Main MVP Components 6
1.3 UML diagram of MVP Architecture 8
1.4 MVVM Components 9
1.5 UML Diagram of MVVM Architecture 10
1.6 Diagram of MVI Architecture 11
1.7 Realization of MVI Architecture 13
1.8 Communication between View and Model of MVI Architecture . . 13

2.1 The diagram of all use cases 15
2.2 Use Case: A list of trending movies 16
2.3 Use Case: Open movie details 17
2.4 Use Case: Save the movie 17
2.5 Use Case: Search functionality 18
2.6 Use Case: Saved Movies Screen 19
2.7 Use Case: Delete the saved movie 20
2.8 Use Case: Delete all saved movies 21

3.1 Diagram of interaction between modules 23
3.2 Runnable modules 25

4.1 Structure of the common module 28
4.2 Structure of the java package of mvp module 32
4.3 Navigation graph of the application 34
4.4 The diagram of Observer pattern 40
4.5 The diagram of MVI Interaction 43

List of Tables

5.1 Code lines of each pattern 50
5.2 Implementation testing 50

Introduction

Many people know that in mobile phones there are two options of Operating Systems (shortly called OS): Android and iOS. There are other OS as well, but the percentage of uses of them are unnoticeable. These OS got less than 1% of users in the latest statistic reports. [1] However, according to statistics, the majority part of phones operating by Android. The reason for that is because Android is an open-source mobile operating system. [2] Thus, various manufacturers can produce their own phone models and use Android as an operating system. Android initially was an idea of Andy Rubin and hereafter was bought by Google. The key to success indeed was the success of Google as they provide Android with Google services and upgrade Android and other environments for it year by year.

Since the beginning of the era of smartphones, Apple and Google started to create a unique store for the applications separately. Therefore many people started to discover a bunch of new applications with different ideas and use from individual developers, big companies and new startup companies as well.

Nowadays, mobile applications have the most frequently using among all systems. There are higher probabilities that your application will be noticeable and useful as people turn on their smartphones more than laptops or smart TVs. Moreover, mobile app stores designed by Apple and Google have provide great user experience with app version system, recommendations, top charts and feedback system.

Regarding to Android, programmers created apps using Java programming language and Eclipse integrated development environment (shortly called IDE) with the special plugin for Android.

Google and other companies modernized it and now the process of development almost new than 10 years ago. In the modern world, most developers use Kotlin as the main programming language by Jetbrains which headquarter is in Prague.

As an IDE Google presented Android Studio as the main development tool and replacement of Eclipse in 2013 at Google I/O.

Development background

Talking about the development of apps, there are three types of development: Native, Web and Hybrid. Web apps based on web technologies, such as HTML, CSS and JavaScript which is quite ordinary. Even though the hybrid development is growing and showing some achievements for example Flutter, the native development still stays as the main point and most companies prefer this way. [3] Considering this, the thesis focuses on Native apps.

Initially, MVC (Model-View-Controller) architecture pattern was used to create mobile applications. This architecture was already used in web development. The reason why it also used in Android development is because people thought that the development process for mobile and web are the same and considered as a front-end development. However, despite the fact that in Android we create user interface with buttons, texts and other widgets, we also need to consider about local databases, energy consumption, efficiency, notifications, accessibility. Nevertheless MVC pattern met some troubles in Android development. First and the foremost, View and Controller components are tightly coupled which makes harder maintain and development. [4] [5]

Motivation and current situation with patterns

According to the project by Google on GitHub namely Android Architecture Blueprint (v1 and v2) with more than 35,000 stars there are other architecture patterns that can be replacement to the MVC pattern. [6] There architectures are called MVP (Model-View-Presenter) and MVVM (Model-View-ViewModel). These patterns have benefits over MVC for several significant reasons. Firstly, extensibility is better, you can create more complex projects and do not get issues with high coupling. Secondly, the interaction with view is much easier. We also need to consider that Google now prefer MVVM pattern and it's Jetpack Components (Libraries for Android by Google) are based on behavior of MVVM pattern. [7]

The listed patterns are considered as imperative programming approaches. With this approach even though most of our challenges will be resolved, we still face some challenges regarding the thread safety, maintaining states of the application. Therefore, we frequently are able to see MVI (Model-View-Intent) pattern in web blogs. [8] This pattern is new in Android development world and we will try to understand advantages and drawbacks of this approach in this thesis.

Structure

The structure of the master thesis will contain several chapters to cover the whole topic. In the first chapter we will study about all these three patterns in details as a theoretical part. In the next chapter, there will be use-cases of the application which I will implement to show the difference between MVP, MVVM and MVI. In the third chapter there will be an implementation of the project which will be written using MVP, MVVM and MVI architecture patterns respectively. After that, unit-testing will be also provided there. In the fourth chapter there will be results of what we got from the implementation and finally summarizes the thesis in the fifth chapter.

Chapter 1

Architecture Patterns

1.1 MVP Model-View-Presenter

Initially, Taligent operation system the first introduce Model-View-Presenter shortly MVP in early 90's. [9] Originally based on MVC pattern with much more clear separation of the components among.

1.1.1 MVP Architecture

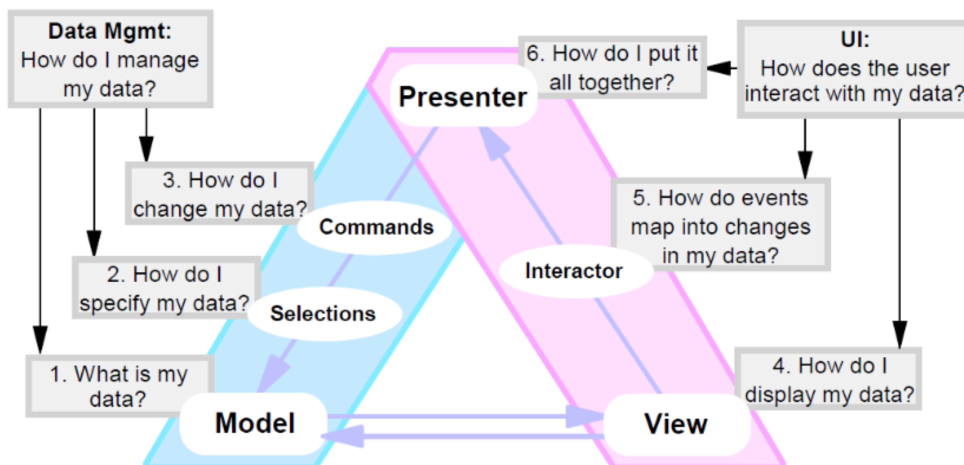

Figure 1.1: MVP Components

In general, we have 6 architecture components in MVP. There are 3 main and 3 following components are recognized in MVP architecture pattern. [9] The first there are model, view and presenter from the name itself (MVP). The other components are called selections, commands and interactor. How-

ever, we need to remember that these three components are mostly put into presenter component to generalize it. Each of these six components has less but clear duties and they are loosely coupled. Talking about Model component, it's the most popular component among all patterns and has similar presentation in these patterns. It indicates what data in this application is. View is the second most popular patters and the meaning of it is what the user is able to see. Shortly saying, it is about graphical user interface seen by user. Presenter is needed to manage, organize and coordinate all the intermediate components which we describe after. Interactor component indicates all events that will be triggered by user. It can be typing on keyboard, clicking by mouse or scrolling list on your phone. Command component provides a list of actions which available and can be executed. Selection component specifies the subset of the data to operate. In the Figure 1.1 we can see all these 6 components of MVP.

To make everything clear, let us imagine a basic situation with note app. First and the foremost, all note apps has a functionality to select some text and make some basic operations on it like copy, paste, cut or delete. In these situation, the text which is String will be our Model. View is simply what we can see on the screen: Highlighted text, other text, the color of text, actions above highlighted text. Selection will be the highlighted text which we selected. It can be a single character, one word, one sentence or maybe several paragraphs if it is needed. Commands will be the operations which we have mentioned above: Delete, Cut, Copy, Paste. Interactor is actions: click, tap, swipe and etc. Presenter is what combines Selections, Commands and Interactor and make a logical flow.

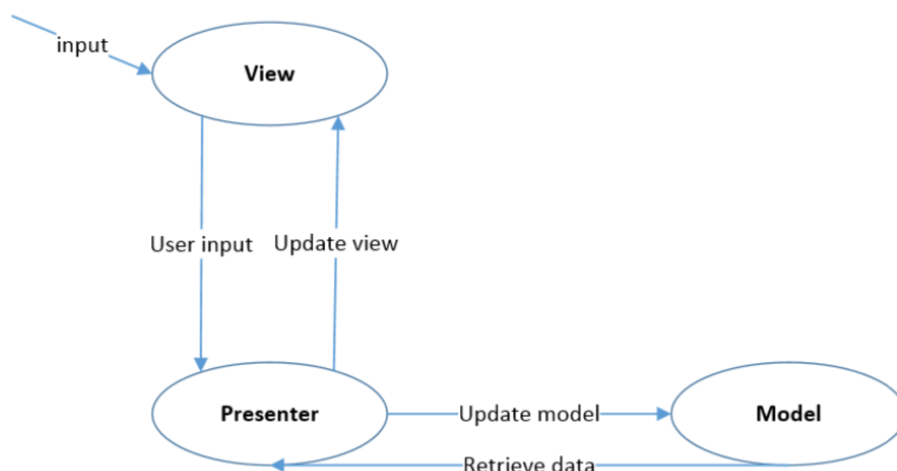

Figure 1.2: Main MVP Components

In the Figure 1.2 we can observe the simpler version of the first picture with only three main components and interactions between them. From this imagine it is obvious that the action from view goes to the presenter and presenter compute something calling model and the model retrieve information to presenter which send it to the view to show. As we see, the coupling between view and model is eliminated. Model does not know anything about the presenter and the presenter does not know about model. Presenter just send the command to let's say repository(Model) and retrieve the result.

1.1.2 MVP realization in Android development

Originally, the are two options of MVP architecture called: Supervising Controller and Passive View. [10] Although the Supervising Controller is original version of MVP, where the view controls simple part of logic while the presenter take care about more complicated logic, in Android is commonly to use Passive View. [11] The difference is that in Passive View we consider the view as a view and let all logic to the presenter. After Google's Android Architecture Blueprints, there are a bunch of examples on the internet where almost all coders use the same principles and it is kind of generalized. In the Figure 1.3 we can see the UML diagram which illustrates MVP implementation on Android. From this diagram, it says that BaseView and BasePresenter are the parents of all views and all presenters respectively. Thanks to these interfaces, we can ensure the fact that the presenter and the view components are actually binding with each other and the required data will be load. While the BaseView interface is the base class of the view components with a method setPresenter() to set the presenter of the view, BasePresenter interface is the base class of presenter component with method start() to prepare data to be shown in the view.

The class called Contact is a class to manage the interface between these views and presenters. Fragment and Activity here are XML files which describes the views of the screens. The Activity basically contains Fragments. It can be several fragments in one screen or just one fragment screen. The instance of the fragment is created in onCreate() method of an Activity. The instance of the presenter is created by calling the constructor method of the presenter. In this step, both view and presenter are bind with each other.

There are also different libraries which in theory should make the whole process easier. One of them is Moxy and widely uses and required in Russia. Moxy is a library that allows for hassle-free implementation of the MVP pattern in an Android apps. The key features of this library are presenter stays alive when Activity is being recreated (it simplifies multi-thread handling) and automatical restoration of all content in the recreated Activity (including additional dynamic content). [12] [13]

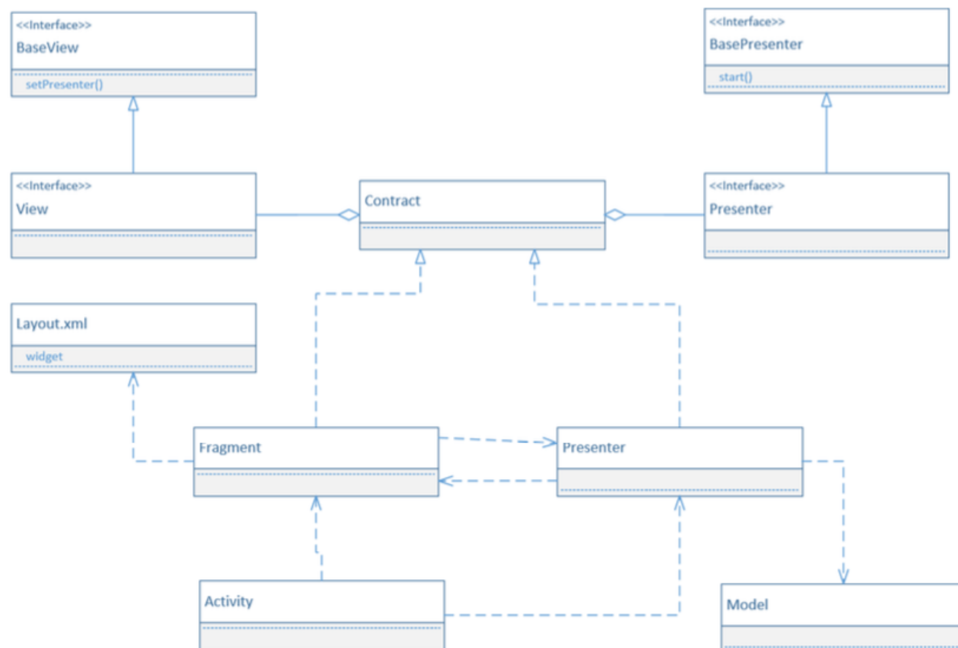

Figure 1.3: UML diagram of MVP Architecture

1.2 MVVM Model-View-ViewModel

At a first glance, MVVM or Model-View-ViewModel seems very similar to the Model-View-Presenter pattern. The difference is that instead of Presenter we have ViewModel.

1.2.1 MVVM Architecture

MVVM as we know stands for Model, View and ViewModel components. It is an architecture which is tailored for user interface development platforms where our view has a responsibility of a designer rather than a coder. [14] Talk about these components individually, Model components is responsible for the representation of the data. View component depicts the user interface of the application: various screens with texts, buttons, sliders and etc. As we can see these two components are same to what they means in MVP pattern. What is most interesting is ViewModel. It stands for a model of the view and intended to manage the state of the view. [15] If the MVP pattern meant that the Presenter was telling the View directly what to display, in MVVM, ViewModel exposes streams of events to which the Views can bind to. It will pass the data to view and also manage the logic of the view and its behavior. Moreover, not only data but also actions, operations. Ideally,

ViewModel should contains the specific data as a replacement of the view-specific data in naming and type. As an example let's say we have to save the data only when the next button is disabled, instead naming the variable like isNextButtonEnabled, the state-specific data goNext or canGoNext is preferred.

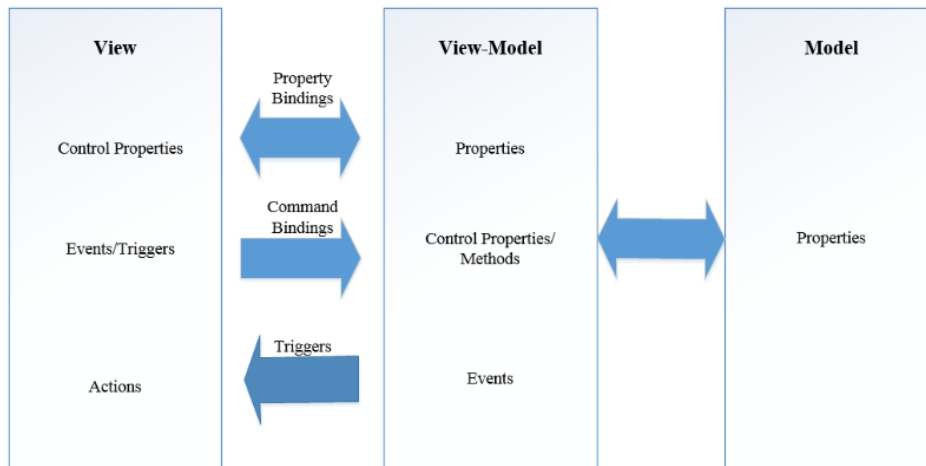

Figure 1.4: MVVM Components

In the Figure 1.4 we can see the interaction between these three components: Model, View and ViewModel. From this picture, the connection between the components View and ViewModel is more complicated than the connection of View with Presenter in the MVP architecture pattern. In MVVM there are exist 2 know types pf connections called Databinding and classical connection. The classical one is the traditional connection which is similar to MVC and MVP where ViewModel will change the view from the code. On the other side, Databinding is more modern and it allows the view directly bound to the properties and operations from the ViewModel itself. [16] Databinding assists ViewModel components from avoiding to notify the changes in the view through the code. Additionally, the view will know about all data when it will be loaded and shows the data by view itself. As a contradiction, in MVP you have to set the loaded data to the view manually. Databinding is used between the View and the ViewModel components and it can be either be directional or bi-directional. The difference is that bi-directional is when the data bound in View is changed, the data in ViewModel is also have a piece of knowledge about this change. [17] As an alternative, directional databinding is when an operation created in the ViewModel component and wad bound to a widget of the View. [18]

1.2.2 MVVM realization in Android development

The whole point of MVVM is to separate layers containing logic from the view layer. [19] On Android, we can use the DataBinding Library which could assist us with it and make a majority of parts of our logic Unit-testable without worrying about Android dependencies. [16] To demonstrate how this works, we will show how Google recommend it from the official website. First and the foremost, to employ the databinding library we need to make several changes based on the current Android implementation after adding this library. We need to set the binding object in the Activity class when inflating an XML layout file in a method called onCreate(). After that in the XML file itself, a new data section with bind variables is declared.

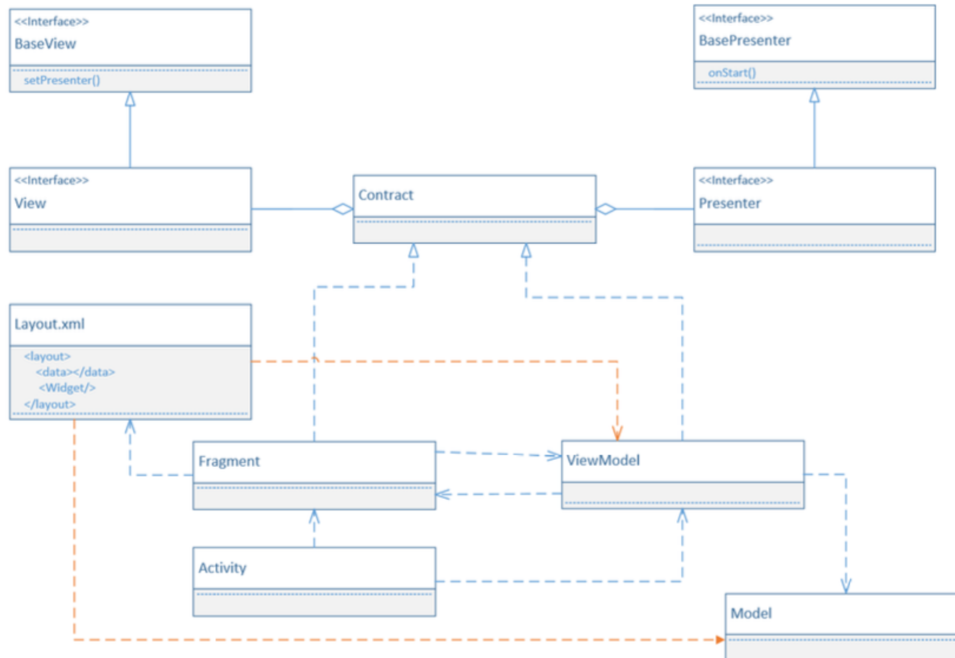

Figure 1.5: UML Diagram of MVVM Architecture

As an example, let's assume that we need to use a username in the application. In Activity class, we have to inflate the XML layout file with databinding methods instead of the original Android implementation with setContent() method. Subsequently, we are getting the reference to this binding and passing the User object (where the first name is Test and the family name is User) to the XML file via the reference. From this point forward, on the top of this XML file, we are adding information about the Object class and method class which will be used in this XML layout file. In the widget properties, use the @{...} to refer to the related events and attributes.

In the Figure 1.5 we can see the UML diagram of MVVM architecture using in Android. The major difference from MVP is that the dependency from the XML layout file to ViewModel and Model class. View XML file should have a piece of knowledge about the structure in the Model component.

1.3 MVI Model-View-Intent

The MVI or Model-View-Intent architecture pattern fist of all appeared in one of the popular JavaScript frameworks called Cycle.js. [20] This framework presented the first view and an idea of the MVI pattern. [21]

1.3.1 MVI Architecture

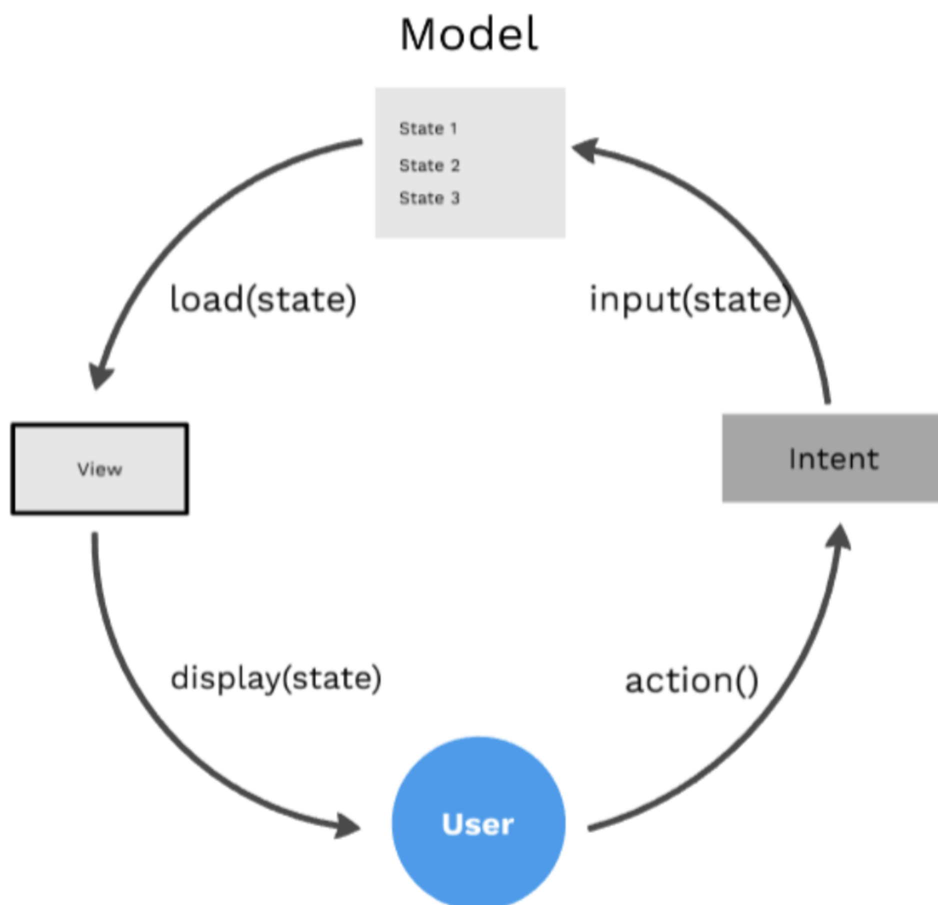

Figure 1.6: Diagram of MVI Architecture

MVI just like other patterns which we mentioned have several components. In this case, there are Model, View and Intent components. Here Model is a representation of the state. They are immutable or should be like that to ensure a unidirectional data flow between them and the other layers in the application. The view is the same as in MVP and MVVM and represents the graphical interface, the user interface of the application. What is interesting indeed is the Intent component. An intent represents an intention or a desire to perform actions by the user or by the app when data loaded or some states change. Each action tells that a View receives an Intent, the ViewModel observes the Intent and Model translate it into a new state. [22]

In the Figure 1.6 there is a diagram which shows the cyclic flow of the MVI representation. Here it can be stated that the User does an action that will be an Intent. The intent is a state which will say about the input to the Model component. Consequently, the Model stores all states and send the requested state to the View component. Subsequently, the View loads the state from the Model and instantly display it to the final user. In case if we observe, the data will always flow from all these steps and will be cyclic. There is no other way of it as it is unidirectional, not bidirectional architecture.

There are several benefits of using the MVI pattern instead of others. One of the is that MVI is purely reactive. [23] Meaning that it makes the whole process much easier to coordinate asynchronous tasks and also brings all the benefits of the declarative way of programming. In the case of the frameworks which was mentioned above, it makes the view much more testable and the view becomes an observable. Another benefit of MVI could be a unidirectional data flow, where data follows just a straight path of model, view and intent. This means that you as a programmer have to learn and adapt how to organize your whole code to use the MVI pattern. Each planned feature should be well written in the same code style, pattern as the other part of the code. Additionally, the view layer is represented in Model-View-Intent by a single object. Meaning that the entire view state is represented by a unique source of truth, including the states as loading and error. You must observe and manipulate one place in order to display the view in the correct way.

1.3.2 MVI realization in Android development

The realization of the MVI architecture patter is depicted in the Figure 1.7 below. In this diagram we can observe that it's more complicated than the previous patterns. It is more complex to implement it fully on Android OS. If we look at this diagram detaily: There is a ViewModel component which emits changes which comes from the model layer to the activity itself. [24] The note is going to pass-through from the Activity to the XML layout file through Data Binding. It is all happening in the View layer. We need to mention that the view layer should have little to no logic. Additionally, the state has to be immutable from the moment. Moreover, the Activity that is receiving entity

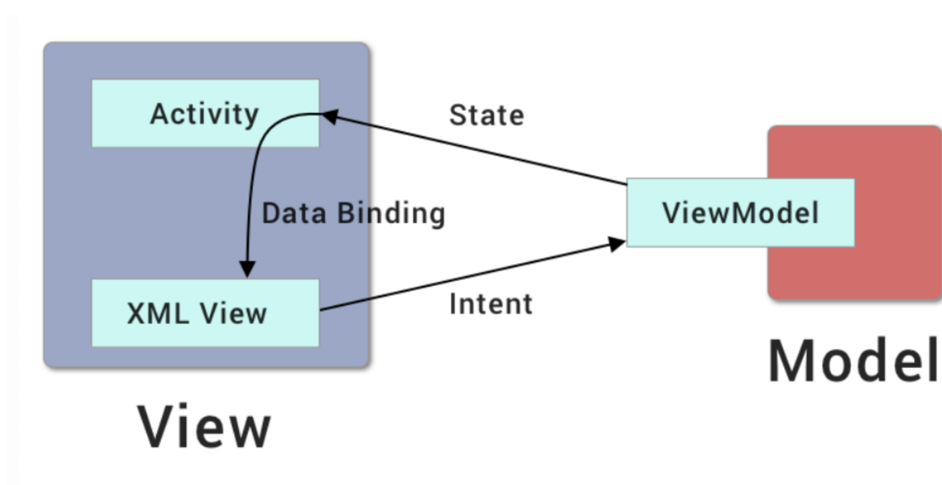

Figure 1.7: Realization of MVI Architecture

should consume the state and also does not allowed to communicate back with the producing entity which is ViewModel component. The next node which goes from XML layout View to the ViewModel produces an Intent.

At this point, both the consumer and the producer is a key portion of our implementation of the MVI pattern. Here, the consumer should have only a single entry point to receive some data from the our producer. Meanwhile, the producer has the ability to have several methods of transmitting data to the our consumer. However, we should also remember that the number should be kept to an absolute minimum.

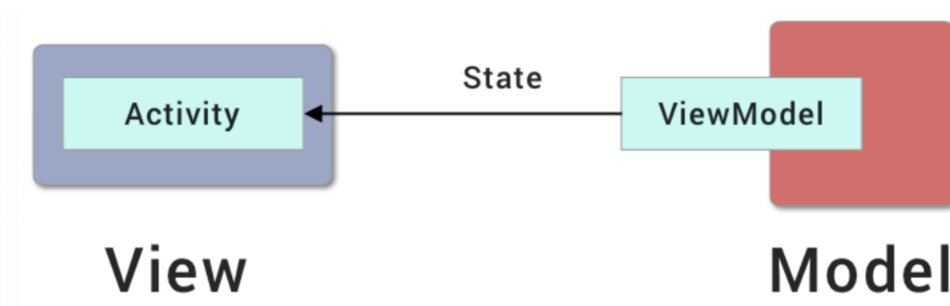

Figure 1.8: Communication between View and Model of MVI Architecture

If we look more detaily into the process of communication between two layers (View and Model) in the Figure 1.8 we will see that ViewModel here has a role of the producer, while the Activity is a consumer. To do some operations we can use external libraries such as Coroutines for Kotlin, RxAndroid which is a supplement to RxJava and has our eyes to the reactive programming world or simply use LiveData from Google. [25]

Chapter 2

Use-Cases of the application

A use-case is an action, event or a step which defines the interaction between a role and a system to achieve a needed goal. It will show each process of the application also represented it on several diagrams. In the Figure 2.1 there a diagram illustration of all use cases.

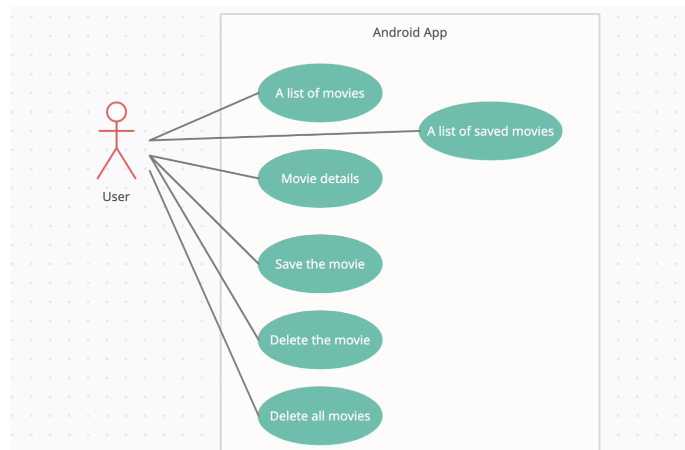

Figure 2.1: The diagram of all use cases

2.1 A list of trending movies

Domain entities: A grid list of movies where each movie item has a poster of the movie and the corresponding title of the movie. The title of the movie will be located after poster at the bottom.

Domain processes: A scrolling of the list. There is an option to load more movies when the list reaches the end of the list. When the end of the list will reach the loading animation will say that the next bunch of movies is loading. After that, the animation will hide and the more data will be shown.

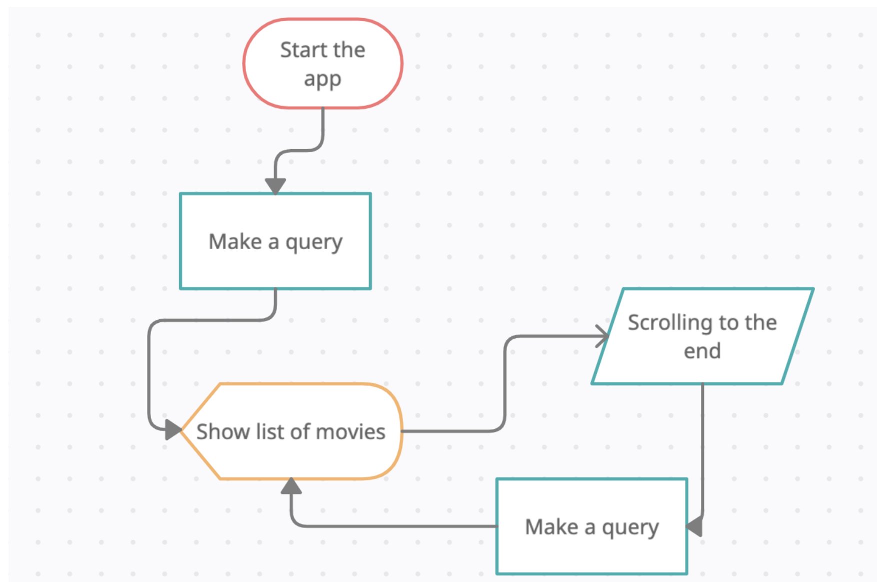

Figure 2.2: Use Case: A list of trending movies

Scope definition: I am going to implement a get query to the themoviedb.org website API to get a list of movies. I will show them on the main screen. An implementation of the pagination will assist to show more data and do it in an efficient way.

2.2 Open movie details

Domain entities: Several images and text fields about the movie. There will be a background image of the movie along with poster of the movie. There will be text information like: the movie title, the year of release of the movie, the rating, how long the movie goes, description and the basic information about budget and revenue.

Domain processes: When the user clicks to the movie in the main list with movies, he directly will navigate to the screen with detailed information of the selected movie. There will be button to return back to the list. The information in the screen will be scrollable because of many details and additionally some movie descriptions could be big enough.

Scope definition: I am going to implement the loading state before data about the movie will not be ready. There also be an error state when the movie cannot load by any cases. I am not going to implement animations when the screen is scrolling as there is no difference for architecture patterns. It will be the same on each architecture.

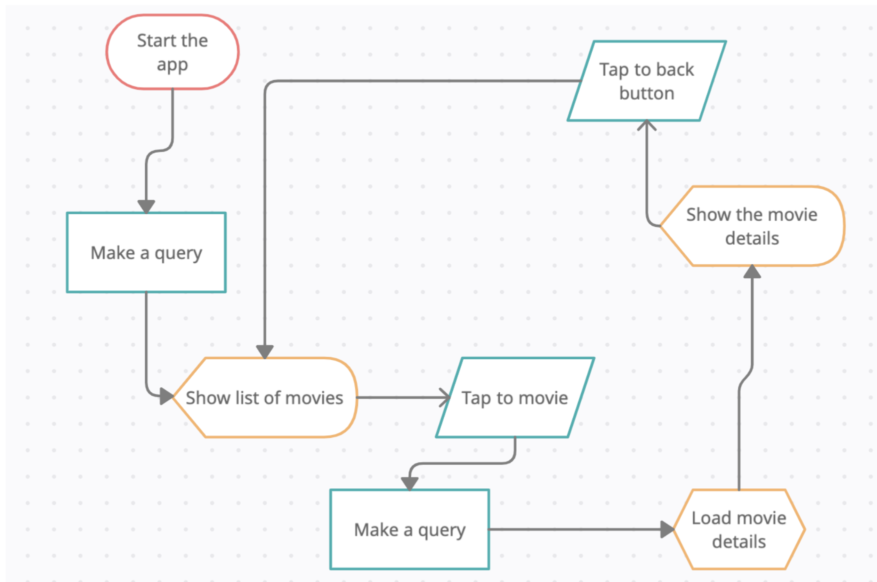

Figure 2.3: Use Case: Open movie details

2.3 Save the movie

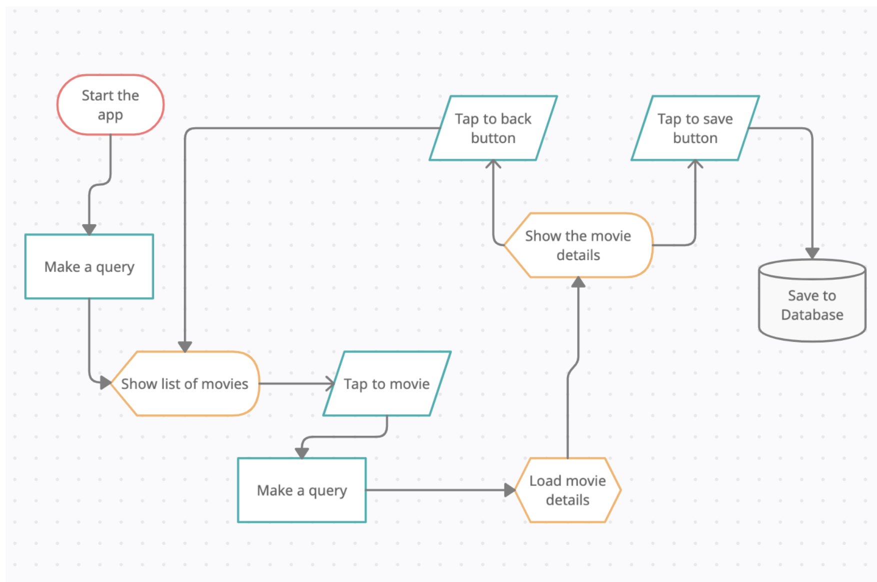

Figure 2.4: Use Case: Save the movie

Domain entities: There will a button on movie details screen to save the limited information about the movie. It will be located at the top of the

screen in an opposite to back button.

Domain processes: By tapping to the button it will change it is state. Basically unfilled star button will be filled. Meaning that, the movie details are saved in the storage.

Scope definition: I am going to implement only a repository to save a limited amount of information about the movie. Basically, I will store only id and the movie title. I am not going to store additional information as it redundant in our case. There is no any profit of stores other fields.

2.4 Search functionality

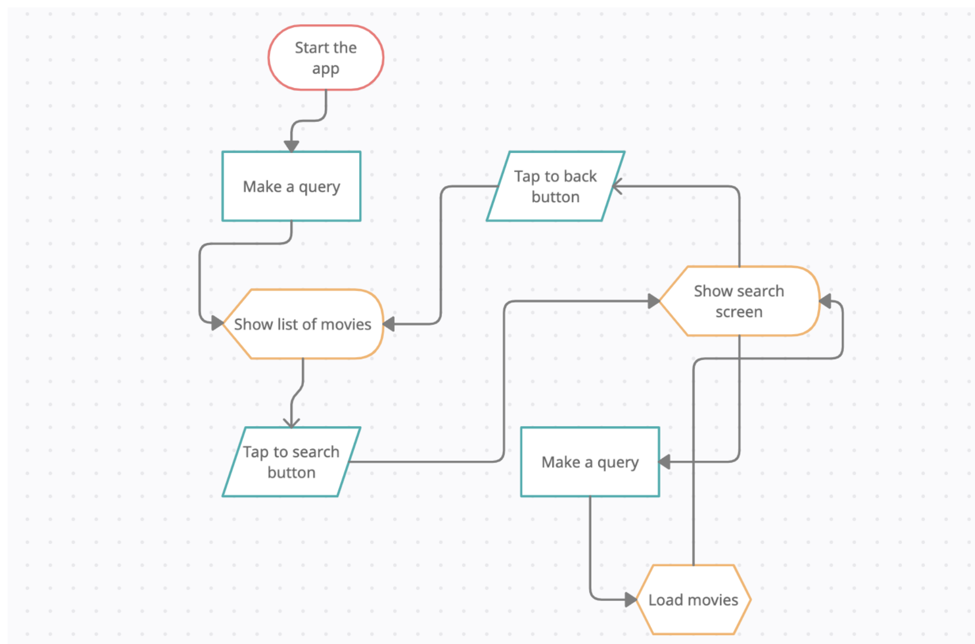

Figure 2.5: Use Case: Search functionality

Domain entities: From the main movies list there will be a button to search a movie what the user want. The location of the button will be on top near the button of saved movies. There will be button back as well to return the initial screen from search movies screen.

Domain processes: By tapping on it there will be search field and the virtual keyboard will be available to use. By searching a movie there will be similar screen as in main screen. In other words, the user will observe the list of movies as he will type a characters.

Scope definition: As per character typed, we will ask server for a list of movies. There is no need to create virtual keyboard, we will use standard android keyboard using InputMethodManager.

2.5 Saved movies screen

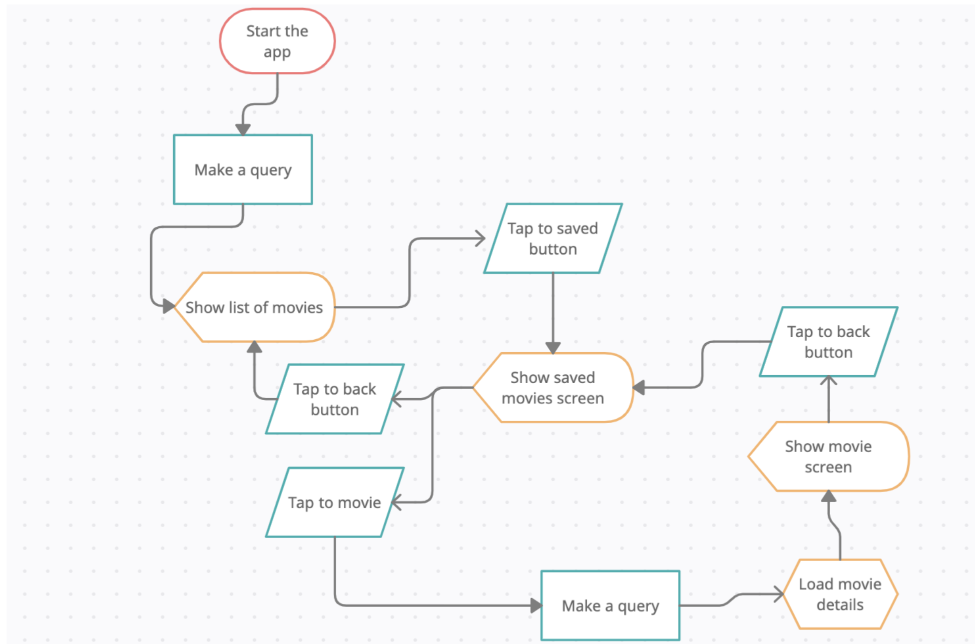

Figure 2.6: Use Case: Saved Movies Screen

Domain entities: From the main movies list there will be a button to look at your saved movies. The location of the button will be on top near the button of search the movies. There will be button back as well to return the initial screen from this saved movies screen.

Domain processes: There will be a list of movies with only text fields. By tapping to one of it the user will navigate to the detailed movie screen.

Scope definition: There will be basic list of movies and by tapping one of them should as server to find this movie by id which we also store but not show the user as it is not going to look well.

2.6 Delete the saved movie

Domain entities: From the saved screen click there will be a dialog on long click to the movie. There will be text to inform the user about if he is sure about his action and two buttons. The first one will be positive action and the other negative/neutral which just will close the dialog.

Domain entities: On long click to the movie from saved movie list we will launch a dialog. In case if the answer will be positive to the question in the dialog we will close the dialog and remove the movie from the list. Otherwise

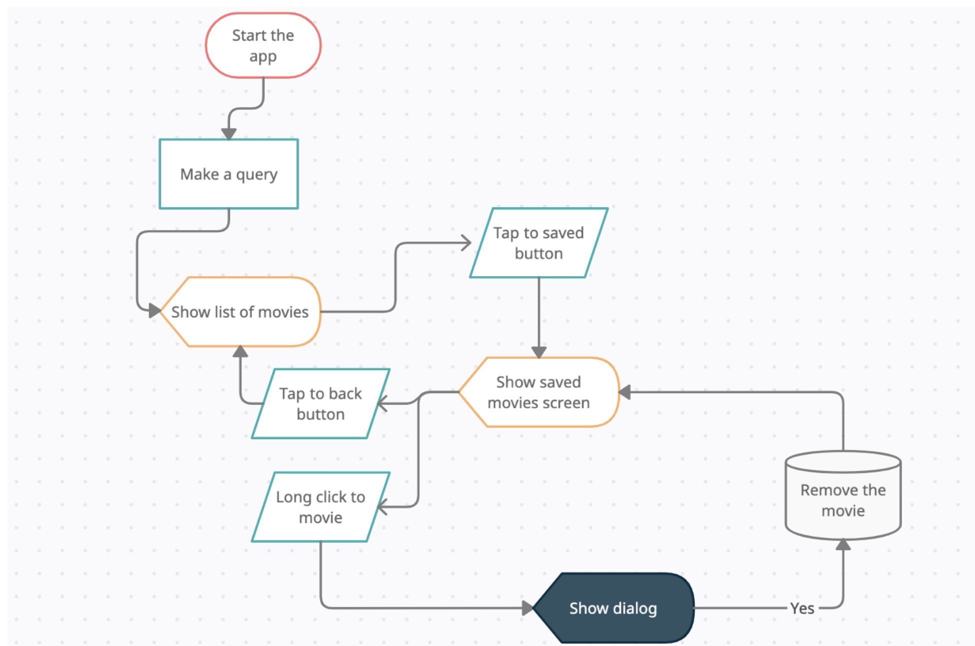

Figure 2.7: Use Case: Delete the saved movie

in the case when the user will tap on the negative button the dialog will close and that is all.

Scope definition: I am not going to implement custom dialog there as there exists standard Android dialog which will be enough in our case. In case of positive action I will remove the object from the repository and will update the adapter of the list.

2.7 Delete all saved movies

Domain entities: From the saved screen there will be a button on top of the screen in opposed direction to the back button. When the user clicks there will be a dialog with text to inform the user about if he is sure about his action and two buttons. The first one will be positive action and the other negative/neutral which just will close the dialog.

Domain entities: On click to this button we will launch a dialog. In case if the answer will be positive to the question in the dialog we will close the dialog and remove all movies from the list. Otherwise in the case when the user will tap on the negative button the dialog will close.

Scope definition: As in other case we I will reuse this dialog with some changes in text. In case when the action will be positive I will clean the repository. In other case I will dismiss the dialog.

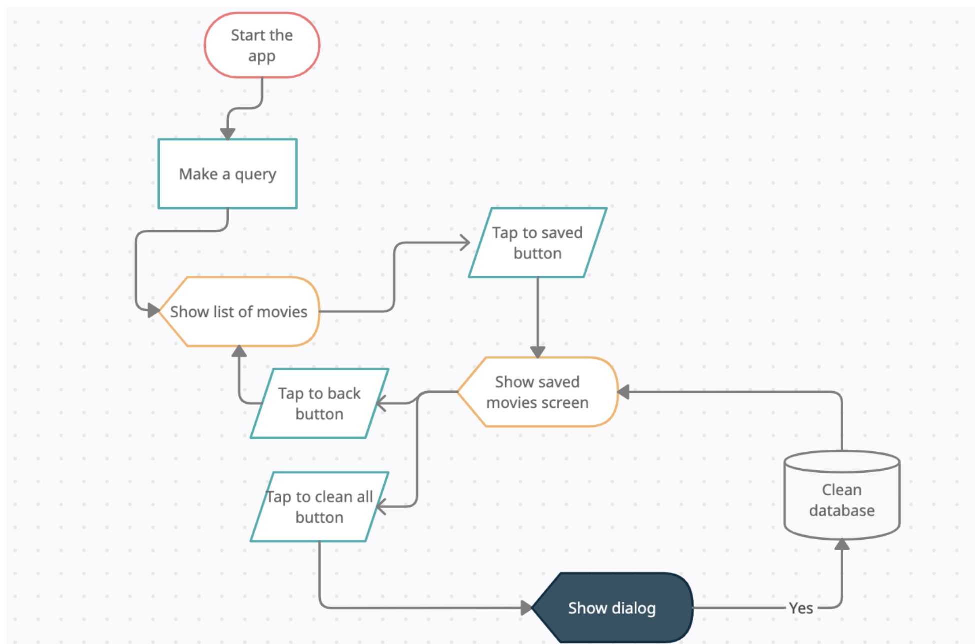

Figure 2.8: Use Case: Delete all saved movies

Chapter 3

Configuration stage

3.1 One project vs Several projects

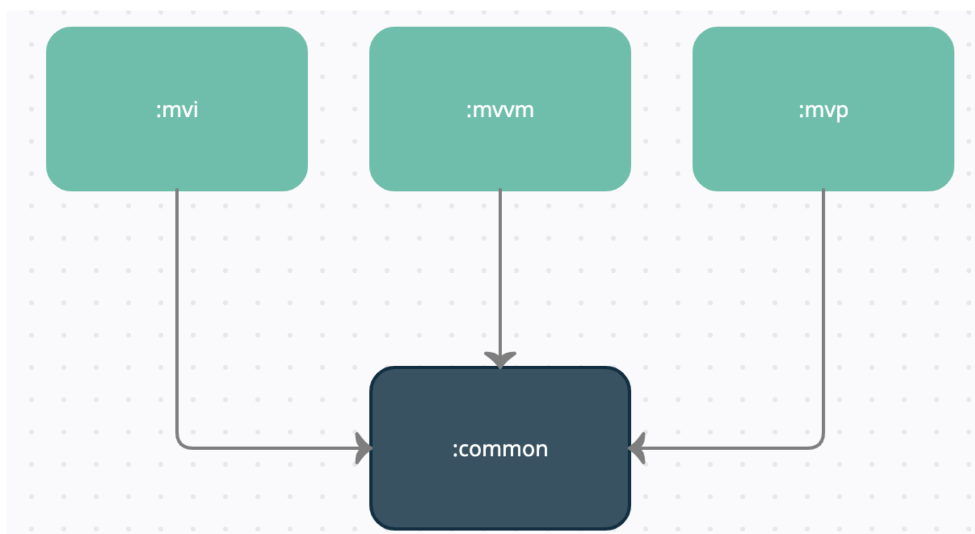

Figure 3.1: Diagram of interaction between modules

Before the implementation, we need to figure out how we will configure our project or maybe several projects. There are two known options to handle this situation. The first and the most common one is to create several projects for each architecture pattern. [26] In this way, we need to create three projects in Android Studio and write our code in all projects apart from each other. This mode of application tends to use juniors or those who do not understand how to configure multi-module projects. [27] Even though it might look very easy to implement, there are other options which we will mention now. As we have started to mention multi-module projects, it is actually the second way

how we are able to create our project with multiple architecture patterns. In this case, we will create each module for each pattern. In addition, there will be also other modules for example for networking, for user interface and for the repository. There will be as many as you can, but need to not forget that the more modules you have, the more complex it will be for your machine and for you. [27] In our case, I will implement only 4 modules, 3 of them for each architecture patterns and one common module which will be as a library of our app. This option will give us the opportunity to not overwrite common things of the app as for example networking or repository stuff. Furthermore, we can run each of these three modules as separate applications. This also will help us to figure out how each architecture pattern will work and go through testing. In Figure 3.1 we can see the diagram of how these modules interact with each other. The :mvi, :mvvm, :mvp modules have an access to the modules :common. Therefore, they are able to get hands-on with the files of this module.

3.2 Configuration of Multi-Module project

To start the process of creating a multi-module project we need to create a single project in Android Studio. This step is very trivial and common. Basically, we need to give a project name, package name and also choose the minimum Software development kit (SDK) of Android OS. Here we are going to select the minimum level SDK as 23. This means that smartphones powered by Android OS 6 and higher are actually able to install the application which we will develop. The reason why we chose this version of SDK is indeed written in the same stage of the project configuration. There are almost 85% of Android users running on 6 and higher. This is a sufficient amount of users and in addition, there are a lot of features using SDK 23 instead of SDK 21 (Android 5.0). [28] By clicking finish the project will open and we demand to wait a while before the configuration process ends.

After the process of the configuration ends, we need to remodel the default package name :app to one of the names of architecture patterns. Let us take the :mvp for it, as we started to describe it first in the first chapter. To do this refactoring process, we demand to do right-click to the module and choose *Refactor* → *Rename*. After this process, we desire to create other modules as well. As there only one module, we choose *New* → *New Module* from right-click to the project itself. 2 of them will be *Phone & Tablet Module* and one of them is *Android Library* (this will be our :common module). The reason for the ladder module is as we said to create the part of code and user interface resources reusable. As an example, we can create some XML resource files, as some parts of user screens, various drawables(basically icons) or for string resources. Therefore, utilize them in other modules to prevent the repetition of the same files.

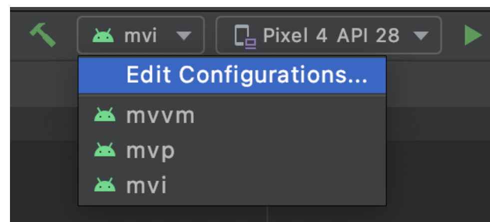

Figure 3.2: Runnable modules

As we can see in Figure 3.2, the modules which we created as a *Phone & Tablet Module* are actually runnable. We can build and run the project as 3 different projects and test them. Basically saying, we can easily test each configuration project just by running what we want to test. If there will be an error or crush in a certain project, we easily fill understand that for instance, the crush has happened in the project with mvp architecture patter, so we do not demand to check other modules, only :mvp one.

To sum up, we have in total 4 modules where 3 of them are runnable as separate applications and we also have a common library to minimize repetition of the same code in the project.

Chapter **4**

Implementation process

First and foremost, we need to decide which architecture pattern we will use first. As we know, there are three major options: mvp, mvvm and mvi. To make everything easy, I am going to implement the most popular and old option in Android Application Development. It is Model-View-Presenter as we mentioned in the first chapter about this architecture pattern. This choice is also acquitted by the fact that the MVP is the most known and so easy to understand the implementation. By working on the app on the base of Model-View-Presenter we also obliged to cover what we are going to implement in our mutual module for all architecture patterns, called :common. The reason why I am going to do that is simply can be proved by the fact that in the common module we will write only the code which we will utilize and reuse in all applications based on different architecture patterns. As we mentioned the :common module, let first of all look at this module. The module has features which will be used in all build apps.

4.1 Common module

Figure 4.1 actually shows us all the information about the common module as a structure. There are 3 major folders called manifest, java and res. The manifest is an XML file that describes key information about the application to the Android build tools. As well as for Android OS and Google Play Console. However, as each of the modules will have their own manifest XML file, where they will have their own name, icon and etc, in the manifest of the common module will get permissions. In our case, we have to write 2 permissions to access the global internet. These 2 permissions look like the following:

```
<uses-permission android:name="android.permission.INTERNET" />
<uses-permission
    android:name="android.permission.ACCESS_NETWORK_STATE" />
```

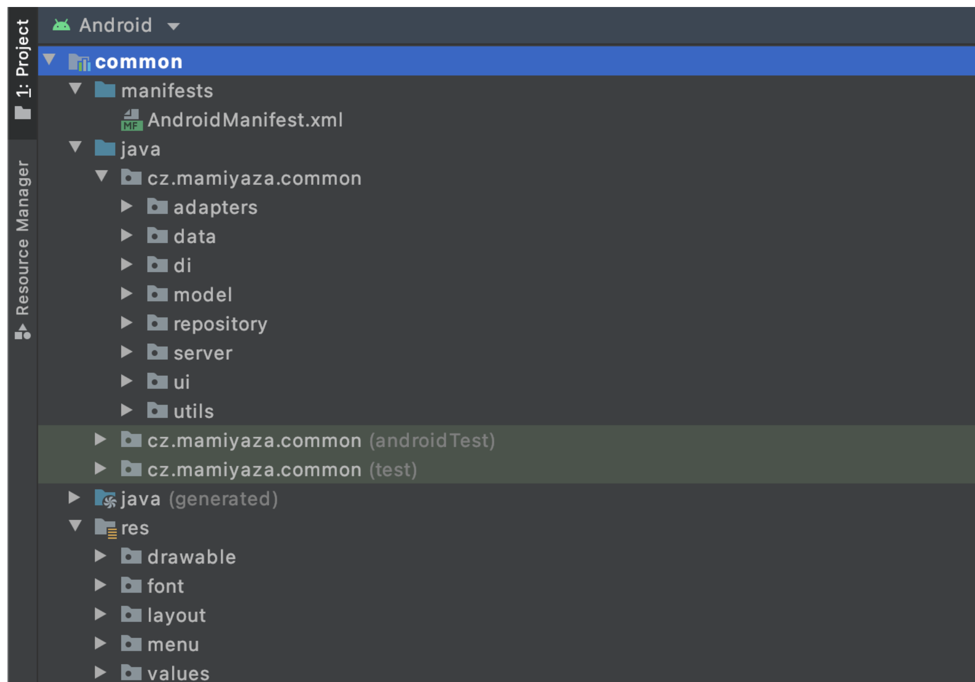

Figure 4.1: Structure of the common module

The other interesting folder is res. Res stands for the resources. Here we can store everything about icons, fonts, menu configurations, colours, styles and the screens themselves. For example, in the folder drawable we have icons in XML format file, while in the layout folder there are some screens and parts of the screens which also can be reusable. In our case, there are 2 of them: One for loading and the ladder one is to show an error with a special button to reload the screen state. The value folder has special files to declare the colours of the app, all string texts as well as styles. The other 2 folders can be described as toolbar buttons (menu folder) and font, where you can write all fonts which will be used in the app. These resources can be used not only in the common module but also for others.

Regarding the java directory, there are different folders that will describe the main mutual code which can be easily reused in the other modules which are runnable. There is no need to write all of this code in others as we will get not good performance, as the Gradle will build the same code more than once. Additionally, we will be lost time while developing each application with different architecture approaches. All the code in the common module will save our time and we will get good performance and fast builds. As we can see from Figure 4.1 there are 8 main folders in the java package. The first thing from this list to describe is **utils** and **ui** files. While **utils** folder has some tools which can be useful like extensions for *Coroutines* and the

formatter for our currency system as well as the object with constant string variables, in UI there only one file which is actually a custom view file to create grid layout list for our movie lists. It will be used on two screens: the Main and the Search screen.

In the **server** folder we have an interface with all requests to the server as well as the custom interceptor class, which helps us to avoid writing the API key each time and it looks like the following:

```
class HttpInterceptor : Interceptor {
override fun intercept(chain: Interceptor.Chain): Response {
    val request: Request = chain.request()

    val sign: String =
        if (TextUtils.isEmpty(request.url.encodedQuery)) "?"
        else "&"

    val newRequest: Request.Builder =

    request
        .newBuilder()
        .url(request.url.toString()+sign+"api_key="+API_KEY)

    return chain.proceed(newRequest.build())
    }
}
```

In the folder **repository** there are 2 files. One of them is used for the local database, to store movies in the storage of the phone, to delete movies from the database, while the other repository gets data from the server. The server one looks like the following:

```
class ServerRepository
    @Inject constructor(private val service: MovieService) {

    suspend fun getMovies(page: Int)
        = service.getTrendingMovies(page)

    suspend fun getMovie(id: Int)
        = service.getMovie(id)

    suspend fun makeSearch(query: String, page: Int)
        = service.getMovieSearch(query, page)
}
```

The one for the main screen has the same structure and the main difference is the use of DAO(Data Access Object) instead of Service.

In the **model** folder there are various data files to get information about movies. For example to get genres of the movie we need a special entity. This entity in our case will be data class:

```
data class ApiMovieGenre(
    @SerializedName("id") @Expose val id: Int,
    @SerializedName("name") @Expose val name: String
)
```

The thing which is grave and very useful is the *Dependency Injection*. Dependency Injection allows the creation of dependent objects outside of a class and provides those objects to a class in different ways. Using DI, we move the creation and binding of the dependent objects outside of the class that depends on them. In our case, we are going to use Dagger Hilt. [29] To be short, The Hilt library is making the whole installation process of the Dagger much easier by code generation with annotations. We are going to create only one module for this project to save our time. In this module, we are injecting our repositories to get access them from the presenters and viewmodels depend on the architecture component which we will use.

```
@Module
@InstallIn(SingletonComponent::class)
class AppModule {

    @Singleton
    @Provides
    fun provideDatabase(@ApplicationContext context: Context)
        = Room
            .databaseBuilder(context, Database::class.java,
                                   MOVIE_TABLE)
            .build()

    @Singleton
    @Provides
    fun provideMovieDAO(database: Database)
        = database.movieDao()

    ...
}
```

Here SingletonComponent used as an injector for the whole application. To be more precise, we can use other components like ViewModelComponent, ActivityComponent or ServiceComponent. However, in our case, it is better to use only one module with SingletonComponent as we do not have many dependencies. In addition, this component will be created in the creation of the application and will be destroyed when the app will shut down.

In **data** folder there are files to configure the *Room* library. [30] There is a DAO interface to describe commands: save, delete, get movies. Furthermore, there is a data class that has only some information about the movie which we will save in the SQL table in the storage of the phone. The DAO interface will look like the following:

```
@Dao
interface MovieDAO {

    @Query("SELECT * FROM movie")
    fun getMovies(): Flow<List<Movie>>
    @Query("SELECT * FROM movie")
    suspend fun getAllMovies(): List<Movie>

    @Insert(onConflict = OnConflictStrategy.REPLACE)
    suspend fun addMovie(movie: Movie)

    @Delete
    suspend fun deleteMovie(movie: Movie)

    @Transaction @Query("DELETE FROM movie")
    suspend fun deleteAllMovies()
}
```

And our database will be an abstract class with an abstract function. To configure the room persistence library we need to extend our class from *RoomDatabase()* and use an annotation *@Database*. We need to declare the version of the database in order to migrate to a modified/upgraded version of our database structure.

```
@Database(entities = [Movie::class], version = 1)
abstract class MovieDatabase: RoomDatabase() {

    abstract fun movieDao(): MovieDAO
}
```

Finally, the last folder called **adapters** has 3 adapters which will be used to create our lists. There one to create a list of movies on main and saved screens, the one to create a list on saved movies screen and the last one is used in the main screen for the genres of the movie.

4.2 MVP Implementation

The implementation of the Model-View-Presenter (or simply MVP) architecture pattern will be written in the corresponding runnable module called mvp. Our application will be based on Single Activity architecture. Using this architecture we will create only one activity which will be main for the entire application. We will view an Activity as a huge container with the various fragments inside the Activity representing the screen, instead of having only one Activity represent one screen. The screens will be considered fragments. Fragments are reusable and they weigh much less than an android activity.

To take a look into the structure of the module, we can three the same pattern which we have already have seen in the common module. There also a package of *manifest* with a manifest file, where we are going to change the name of the application, the icon of the application and we also have to register the application class in the manifest. There is also *res* resource package where we have to look at the navigation folder. This folder is not mentioned in the common module, as it is used for the navigation through the app between our screens (fragments) by using the *Navigation Component* library. [31] Although we started to mention it, we will discuss it in more detail further.

The last package is our java package and the structure of it has been depicted in the Figure 4.2.

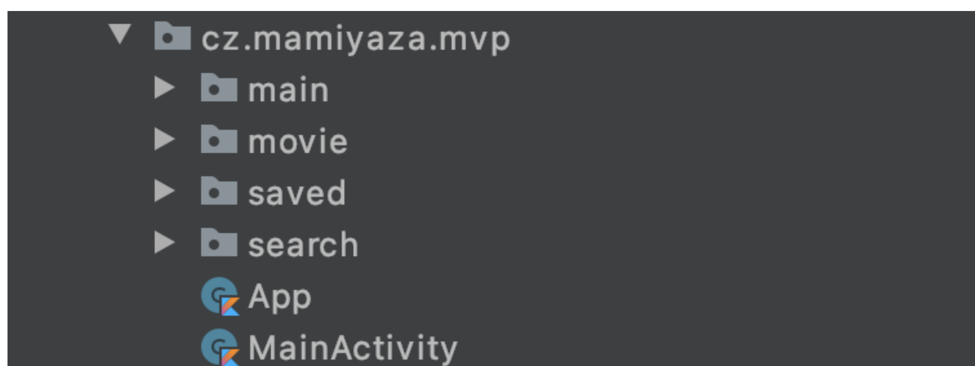

Figure 4.2: Structure of the java package of mvp module

As we are able to look at this structure. It has 2 classes and 4 folders for each screen. This organization is might be helpful for us to see the difference between each screen and between other architecture patterns in terms of the

comparison of each screen by its folder. In other words, we achieve this structure because other parts of the application such that repositories, networking, dependency injection, entities are located in a common module.

First and foremost, we need to discuss about *App* and *MainActivity* classes as it is a part of the configuration of the project. As we said earlier, we are going to register the application class in the manifest. To make this we have to create a special class which will extend by the Application class. Subsequently, we are going to annotate this class by *@HiltAndroidApp* annotation. This annotation generates all the component classes which we have to do manually while using Dagger, so it is crucial to use. The class will be looking at the following code part:

```
@HiltAndroidApp
class App : Application() {

    override fun onCreate() {
        super.onCreate()
    }
}
```

After that, we can easily find this class from the manifest file and update by adding following into the application tag:

```
android:name=".App"
```

In terms of the MainActivity class, the purpose of it to create an activity where we will attach our fragments as screens. To achieve it, we are going to create a class that extends from the *AppCompatActivity*. This class is a base class for activities that wish to use some of the newer platform features on older Android devices. Additionally, we have to use *@AndroidEntryPoint* annotation. This is needed if we want the hilt to inject bindings there. Annotating this will trigger a code generation process for setting up Dependency Injection for this. The class itself will look like the following code. If we would like to add bottom navigation for our application in Single Activity architecture we also need to configure it in the activity class.

```
@AndroidEntryPoint
class MainActivity : AppCompatActivity(R.layout.activity_main) {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
    }
}
```

Before the actual code of the Model-View-Presenter let us look at the navigation graph of our application. To create this graph, we need to create a *navigation* folder into the *res* resources package. After that, it is time to create an XML file in this folder. By writing XML code we will get generated graph of our application with all navigation paths which we declared in the XML code.

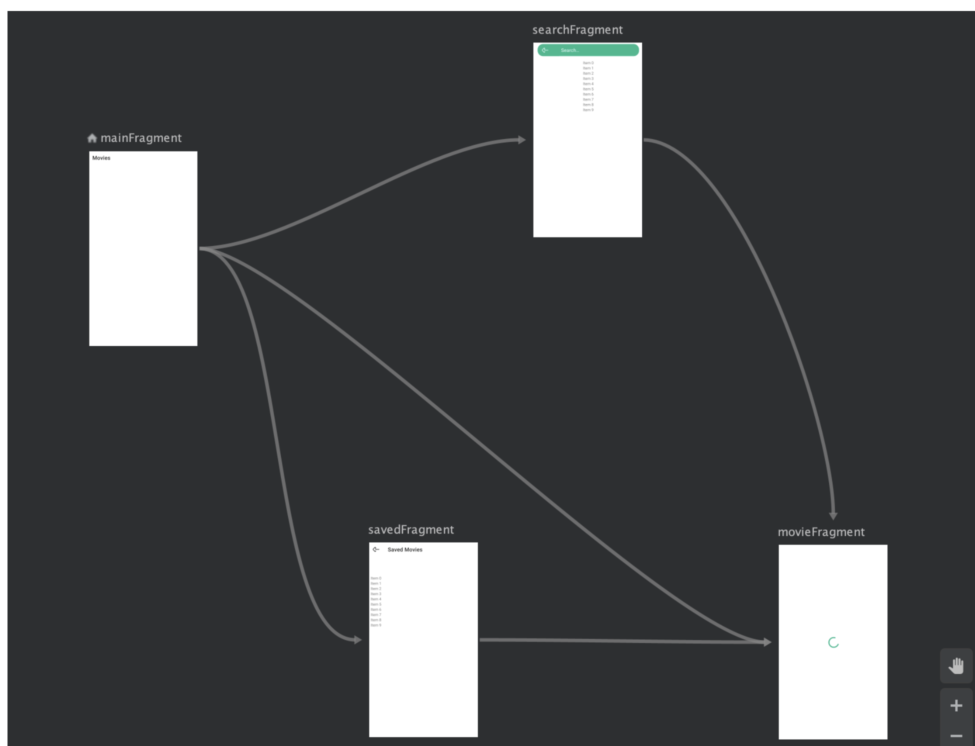

Figure 4.3: Navigation graph of the application

Regarding the graph, there are four screens and the start point is the main fragment which has a navigation path for all other three fragments. The movie fragment here considered as a final destination as we do not have use-cases for the navigation from this screen. The other two fragments can be achievable from the main fragment and also able to communicate to the movie fragment.

And as an instance, we are going to countenance part of the XML code. The reason for that, the other fragment tags are similar to what we are going to show. The key part is to have a fragment tag for each screen and for the navigation path we require to write action tags. We crave to make attention to where these action tags are located. In general, it is mandatory to write action tags into fragment tags which is a start point for the navigation path. Furthermore, we need to write the start destination attribute in the navigation tag.

```
<?xml version="1.0" encoding="utf-8"?>
<navigation xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    xmlns:android="http://schemas.android.com/apk/res/android"
    app:startDestination="@id/mainFragment">
    <fragment
        android:id="@+id/mainFragment"
        android:name="cz.mamiyaza.mvp.main.MainFragment"
        android:label="Blank"
        tools:layout="@layout/include_main_screen" >
        <action
            android:id=
            "@+id/action_mainFragment_to_searchFragment"
            app:destination="@id/searchFragment" />
        <action
            android:id=
            "@+id/action_mainFragment_to_movieFragment"
            app:destination="@id/movieFragment" />
        <action
            android:id=
            "@+id/action_mainFragment_to_savedFragment"
            app:destination="@id/savedFragment" />
    </fragment>

    ...

</navigation>
```

In general, our MVP has three modules. The model one is actually the code which we already described in our library module called common. The reason why we are going to make this is because the Model module is persistent in all these three architecture patterns.

For MVP we need the following classes and interfaces: Fragment class, Presenter class and the View interface. The Fragment will initiate the Presenter and we will send the view instance to the presenter. In addition, Fragment will implement the View interface. While the Fragment will implement the interface and initialize the Presenter class, in the Presenter we are going to initiate the View.

To keep on it in details, we are going to cast a look at our created main folder in the java package of the MVP runnable module. Even though there are only two files inside of this folder: MainFragment and MainPresenter. The View interface is located inside of the MainPresenter class. The View is in this case called as *MainView* and it looks like the following interface with specific set of functions inside:

```
interface MainView {
        fun showMovieList(movies: List<ApiMovieLite>)
        fun showMoreMovieList(movies: List<ApiMovieLite>)
        fun showLoading()
        fun hideLoading()
        fun showConnectionError()
    }
```

Here we can see that there are 5 functions inside of this interface. The first one is to upload initial data with the list of movies, while the second is for the other data. Simply saying, it is used to get more pages from the API and show them as an addition to our list. In this way, we remain our data and the additional data will be a continuation of our list. The method *showConnectionError()* is using when the data empty or there is a problem with the server or with getting the data (Lack of internet connection). The other 2 left methods are using to show and hide the loading state when getting the initial data.

To take a look into the Presenter we are able to observe the two methods of the presenter class itself to attach and detach the view. It is useful for the implementation process of the MVP to send the data from Presenter to Fragment via the View and we need to detach the view when *onDestroy()* method of the fragment will be triggered.

```
class MainPresenter @Inject constructor(
    private val serverRepository: ServerRepository,
){

    var view: MainView? = null

    fun attachView(view: MainView) {
        this.view = view

        loadMovies()
    }

    fun detachView() {
        this.view = null
    }
}
```

The function called *loadMovies()* is also private function of the presenter to get the data from the *serverRepository* and send the data to the view. First of all, we are calling the show loading method and when data will be achieved we hide the loading and send the data to the view.

```
private fun loadMovies() {
        loadingMovies()

        CoroutineScope(Dispatchers.Main).launch {
            withContext(Dispatchers.Main) {
                try {
                    val movies = serverRepository.getMovies(1)
                    if (movies.results.isEmpty()) showError()
                    else showMovies(movies.results)
                } catch (e: Exception) {
                    e.printStackTrace()
                    showError()
                }
            }
        }
    }

    private fun loadingMovies() {
        view?.showLoading()
    }

    private fun showMovies(movies: List<ApiMovieLite>) {
        view?.hideLoading()
        view?.showMovieList(movies)
    }

    private fun showError() {
        view?.hideLoading()
        view?.showConnectionError()
    }
```

On the fragment side aside from the implementation of the view and initiation of the presenter, we need to override all methods of our MainView and write the implementation of these functions.

```
override fun showMovieList(movies: List<ApiMovieLite>) {
    mainAdapter.addAllMedia(movies)
}

override fun showMoreMovieList(movies: List<ApiMovieLite>) {
    mainAdapter.showMoreMedia(movies)
}

override fun showLoading() {
```

```
    binding.loadingScreen.root.visibility = View.VISIBLE
}

override fun hideLoading() {
    binding.loadingScreen.root.visibility = View.GONE
}

override fun showConnectionError() {
    binding.errorScreen.root.visibility = View.VISIBLE
}
```

And we need to attach the view to the presenter and detach it in the following methods:

```
override fun onCreateView(inflater: LayoutInflater,
container: ViewGroup?, savedInstanceState: Bundle?): View? {
    ...
    presenter.attachView(this)
    ...
}

override fun onDestroy() {
    presenter.detachView()
    super.onDestroy()
}
```

Overall, it is the implementation of the MVP in general. We need three major things for it: The Fragment, The Presenter and the View interface which is going to connect communication between the presenter and the fragment.

We also need to take into attention that we need to somehow send the movie ID to the next screen if it will be a Movie Screen. To Achieve this, we are going to use the benefits of the Navigation Component library. We are obliged to send it to throw the *navigate()* method as an argument.

```
val args = Bundle().apply {
    putInt(Constants.MOVIE_ID, movie.movieId)
}

findNavController()
    .navigate(R.id.action_mainFragment_to_movieFragment, args)
```

And to get this argument we have to look at the MovieFragment. To get this argument into the MovieFragment we need to take it in *onViewCreated()* method like the following code part:

```
override fun onViewCreated(view: View,
                           savedInstanceState: Bundle?) {

    val id = arguments?.getInt(Constants.MOVIE_ID, -1) ?: -1
    ...
}
```

Regarding other screens, we can say that the implementation of the Model-View-Presenter architecture pattern still remains the same as it should be. For each screen, we have the fragment, the presenter and the following view interface with a corresponding naming depending on the screen name. The difference obviously depends on the use cases. In some presenters, we inject both local and server repositories as in MoviePresenter. The code part will be following:

```
class MoviePresenter @Inject constructor(
    private val mainRepository: MainRepository,
    private val serverRepository: ServerRepository,
){
    ...
}
```

4.3 MVVM Implementation

The implementation of the Model-View-ViewModel is different from the MVP realization. Even though the model remains the same and we do not change it fundamentally, the other part of the implementation has its own features. In the implementation of the MVVM the application and MainActivity classes do not have any changes aside from MVP one. There still the same annotations and the same code.

To look into the key part of the MVVM implementation we need to take a glance at one of the folders in the java package of the mvvm runnable module. If we look at the main folder, there are only two files, which represents the fragment and the viewmodel called: MainFragment and MainViewModel.

First of all, let us have a look at the structure of the fragment and the difference between implementations of the MVP and MVVM. The difference starts with the initialization of the viewModel. Here we get a reference to the ViewModel scoped to this Fragment by using *viewModels()*.

```
private val viewModel: MainViewModel by viewModels()
```

The another option for it looks like this:

```
private val viewModel by viewModels<MyViewModel>()
```

The interesting part of the using of the *Observer* pattern in the MVVM. [32] Basically, we have our view (in our case it is a fragment) and we need to subscribe to the changes in the ViewModel. Ideally, the ViewModel does not know about android components (framework classes). Considering this fact, the ViewModel actually does not have a piece of knowledge about how Android is killing the view frequently. Counting this, there are several advantages:

1. the ViewModel is persisted during all configuration changes, so there is no need to re-query an external source for data when a rotation of the screen happens.
2. When long-running operations finish, the observables in the ViewModel are updated. It does not matter if the information is being observed or not.
3. ViewModel does not reference view so there is a low risk of memory leaks. The diagram of the Observer pattern is shown in the Figure 4.4:

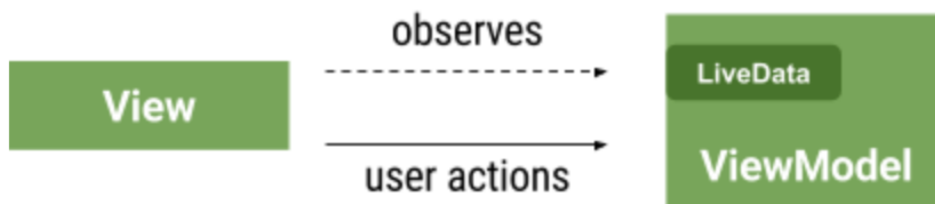

Figure 4.4: The diagram of Observer pattern

LiveData ensures that User Interface is always up to date with the data even when the app's activity is restarted while in use. This is a key feature of using the MVVM architecture pattern. We do not need to update the data by ourselves, it will update and change itself once we will subscribe to it. In our case, we are going to subscribe to different LiveData-s.

```
viewModel.loading.observe(viewLifecycleOwner) { loading ->
    if (loading) binding.loadingScreen.root.visibility
                                      = View.VISIBLE
    else binding.loadingScreen.root.visibility = View.GONE
}

viewModel.error.observe(viewLifecycleOwner) { error ->
    if (error) binding.errorScreen.root.visibility
                                      = View.VISIBLE
    else binding.errorScreen.root.visibility = View.GONE
}

viewModel.data.observe(viewLifecycleOwner) { data ->
```

```
    mainAdapter.addAllMedia(data)
}

viewModel.moreData.observe(viewLifecycleOwner) { data ->
    mainAdapter.showMoreMedia(data)
}
```

As we can observe, there are 4 subscriptions. One of them is for loading screen state, the second for the error screen state, other 2 is to get data and store it into the adapter. The whole code with observe methods are written into the *onViewCreated()* method. The *viewLifecycleOwner* there means the lifecycle of the fragment's view.

Regarding the ViewModel, there is a new annotation by our Dagger Hilt Dependency Injection library called *@HiltViewModel*. This will identify a ViewModel for construction injection. Additionally, we need to extend our ViewModel class with *ViewModel()*.

```
@HiltViewModel
class MainViewModel @Inject constructor(
    private val serverRepository:ServerRepository):ViewModel() {

    ...
}
```

In the code below I am using the states which will say about the loading, error or loaded data state.

```
private val state = MutableLiveState<List<ApiMovieLite>>()
private val dataState = MutableLiveState<List<ApiMovieLite>>()

val loading: LiveData<Boolean> = MediatorLiveData<Boolean>()
    .apply {
        addSource(state.mapLoading()) { value = it }
        addSource(dataState.mapLoading()) { value = it }
}

val error: LiveData<Boolean> = MediatorLiveData<Boolean>()
    .apply {
        addSource(state.mapError()) { value = it }
        addSource(dataState.mapError()) { value = it }
    }

val data: LiveData<List<ApiMovieLite>>
= state.mapLoaded().mapNotNull { it }
```

```
val moreData: LiveData<List<ApiMovieLite>>
= dataState.mapLoaded().mapNotNull { it }
```

The *MutableLiveState* here is an custom extension which is actually

```
    MutableLiveData<State<T>>
```

The State here is a sealed class.

```
sealed class State<out T> {

    object Loading : State<Nothing>()

    data class Error(val error: Error) : State<Nothing>()

    data class Loaded<out T>(val data: T) : State<T>()
}
```

We needed this MutableLiveData with a State to send the various commands inside the ViewModel like Loading, Error or Loaded. For instance, let us have a look at the method of getting the initial data for the main screen.

```
fun getMovies() {

        state.loading()

        viewModelScope.launch {

            when(val result = wrapResult
                            { serverRepository.getMovies(1) }) {
                is Result.success ->
                        state.loaded(result.value.results)
                is Result.failure ->
                        state.error(result.error)
            }
        }
    }
```

As we can see from there we set the command for loading and when the data will be loaded we send loaded command or in case of fail, we send the error command. A *viewModelScope* here is defined for each ViewModel in the app. Any coroutine that launched in this scope is automatically cancelled if the ViewModel is cleared. Coroutines are useful here for when we have work that needs to be done only if the ViewModel is active.

There is another option as well in the Saved Movies screen. As it is a simple screen without API handling, we do not need states there. If we look at *SavedMoviesViewModel* we will see the following code lines:

```
val savedMovies = mainRepository.getMovies()
                    .asLiveData(viewModelScope.coroutineContext)
```

As a conclusion of what we discovered during the Model-View-ViewModel implementation, we can say that this approach is more modern than the MVP one. Furthermore, it is supported by Google and the power of LiveData makes the interaction between View and ViewModel components smooth. The other part of the application will not be shown here as the implementation of the MVVM is the same for other screens as well.

4.4 MVI Implementation

The Model-View-Intent is quite a new approach in the world of Android development. There are different ways to write the implementation. As we mentioned in the description of the MVI pattern, it based on the known principle of unidirectional and cylindrical flow inspired by the Cycle.js framework. The MVI pattern has three major components: Model, View and Intent.

The model component has the same role as in the other two patterns. However, the code part will be different as we will add states and the repository methods will be different.

In Figure 4.5 we can observe that the main idea is similar to MVVM but with *User Interaction*. The meaning of user interaction is in our case is an Intent. This intent is a state that is an input to our model. The model will store all states and send the requested one to the View. The view will load the state which came from the Model and depict it to the user.

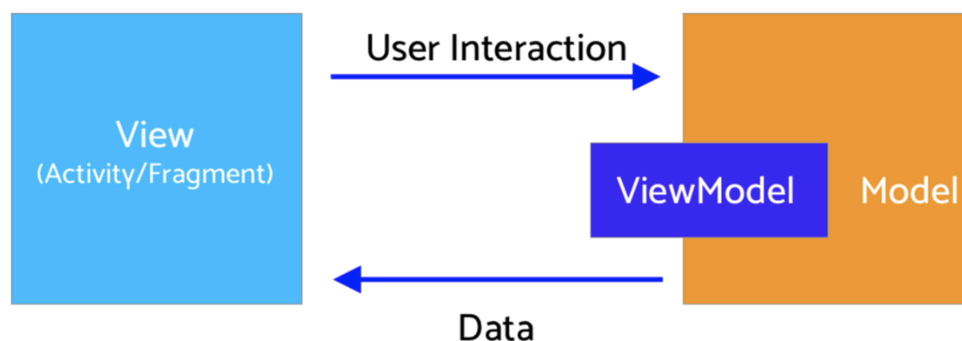

Figure 4.5: The diagram of MVI Interaction

If we look at the difference in the Model component, we are going to find there *DataState* sealed class. This class will be useful to set states for our MVI project. There will be three states: Success, Error and Loading. The Success

state is a case when the network operation is successful or what if some kind of operation is successful. For this case, we will return some data. The next state is an Error state which is a case if something gets wrong during the operation and we will receive an exception from this state. The last state is the Loading state which is a case when something will be in a loading process in an operation and will tell the User Interface to show the loading animation. This class looks like the following code:

```
sealed class DataState<out R> {

    data class Success<out T>(val data: T) : DataState<T>()

    data class Error(val exception: Exception) :
                                        DataState<Nothing>()

    object Loading : DataState<Nothing>()
}
```

The next stage is looking into the repository changes. We are going to look at the *ServerRepository* class. Here, as we remember, were three methods: *getMovies(page: Int), getMovie(id: Int)* and *makeSearch(query: String)*. The first one is used to get the trending movies for our main screen. The second is for getting data of the movie which was selected from the main screen with trending movies, from search results or from saved movies screen. The last function is used to make a search query and receive the result of our search.

The additional functions for the MVI pattern will be the same functions with the difference in the included code. We are going to return *Flow* which is a Kotlin Coroutines data structure. Then we are going to use a *Flow builder*. Let us take on of these functions to look at the code:

```
suspend fun getMoviesFlow(page: Int):
Flow<DataState<List<ApiMovieLite>>> = flow {

    emit(DataState.Loading)
    try {
        emit(DataState.Success(
            service.getTrendingMovies(page).results))
    } catch (e: Exception) {
        emit(DataState.Error(e))
    }
}
```

The first thing we can observe here that it is much-complicated code than what we had earlier, where we just wrote the function as a single line:

```
suspend fun getMovies(page: Int)=service.getTrendingMovies(page)
```

We are going to emit a loading state, so it is going to be interpreted to the user interface as a loading view. The fragment code part where we are going to write *DataState.Loading* will be triggered. The main operation we are going to write in try-catch block as a purpose. We are going to emit the result as *DataState.Success(*our result*)* and in case of the exception we also going to emit, however, it will be *DataState.Error(e)*.

In terms of the class that is responsible for preparing and managing the data for an Activity or a Fragment in the MVI architecture pattern, we are using the ViewModel as in the MVVM. However, the first thing we are going to do is to create outside a sealed class for each ViewModel class. If we take a glance at the main folder we are going to write there a class called *MainStateEvent*. Inside of this sealed class, we are going to describe all of the different events that we can fire off. When the event gets fired off we are going to trigger that action into the ViewModel. For the Main screen this *StateEvent* class will look like:

```
sealed class MainStateEvent{

    object GetMoviesEvent: MainStateEvent()

    object GetMoreMovies: MainStateEvent()
}
```

In the ViewModel itself we are going to create a data state object which is going to be MutableLiveData object which will return DataState.

```
private val _dataState:
MutableLiveData<DataState<List<ApiMovieLite>>>=MutableLiveData()
```

Furthermore, we need a way to take in-state events and convert them into the data state. For this, we need to create a getter function for the data state:

```
val dataState: LiveData<DataState<List<ApiMovieLite>>>
    get() = _dataState
```

In addition, we need a special function for setting the state event, interpreting the state event and doing our operations given a certain state event. As we can see from the code below, there is a function that takes the MainStateEvent and inside of the function we write our operations depending on the event. Inside on *onEach ...* we going to set received data to the data state.

```
fun setStateEvent(mainStateEvent: MainStateEvent){
    viewModelScope.launch {
        when(mainStateEvent){
            is MainStateEvent.GetMoviesEvent -> {
```

```
                serverRepository.getMoviesFlow(1)
                    .onEach {
                        _dataState.value = it
                    }
                    .launchIn(viewModelScope)
            }
            is MainStateEvent.GetMoreMovies -> {
                serverRepository.getMoviesFlow(page)
                    .onEach {
                        _moreDataState.value = it
                    }
                    .launchIn(viewModelScope)
            }
        }
    }
}
```

Inside of the Fragment in terms of to call our movie list we are going to write the following code line:

```
viewModel.setStateEvent(MainStateEvent.GetMoviesEvent)
```

And to observe our data we are able to write our code inside of the *onViewCreated()* function. The code is similar to the code that we have in the MVVM implementation, but we need to observe only one live data instead of 3 for receiving movies. Basically saying, we are not obliged to observe loading and error live data-s as in MVVM. Based on the event of the *dataState* we will show either loading, either error, either the data which we received into the list.

```
viewModel.dataState.observe(viewLifecycleOwner, { dataState ->
    when (dataState) {
        is DataState.Success<List<ApiMovieLite>> -> {
            binding.loadingScreen.root.visibility = View.GONE
            mainAdapter.addAllMedia(dataState.data)
        }
        is DataState.Error -> {
            binding.loadingScreen.root.visibility = View.GONE
            binding.errorScreen.root.visibility = View.VISIBLE
        }
        is DataState.Loading -> {
            binding.loadingScreen.root.visibility = View.VISIBLE
        }
    }
})
```

For the conclusion of what we did in the implementation of the Model-View-Intent architecture type, we see how we used states and called them from our View. Additionally, we used the ViewModel here, which tells us about mixing them in our project. We can use a combination of architecture patterns so that it fits the unique needs and goals of your mobile application. We can use our StateEvent as feedback of user's interaction and other code parts do as in the MVVM pattern. Another example is using MVI in one screen and the MVVM in another which much simpler and do not have many things to do here.

Chapter 5

Comparison

In this chapter we will talk about the various aspects of the architecture patterns: MVP, MVVM and MVI. The key point is to understand the main differences between these patterns and figure out the advantages and disadvantages of each of them. Furthermore, we will understand how many code lines, man-days and the complexity to implement and test all these architecture patterns. As an addition, we will mention which approach is modern and mostly used nowadays.

5.1 General details of comparison

First of all, let us take a look at table 5.1. In this table we can see how many code lines are used in each architecture pattern. We will compare common module and runnable module of each architecture separately. Model-View-Presenter approach has fewer code lines in terms of common module. Meaning that we do not need to write many code lines in the Model component. However, if we take a look into the other components we can see the reserve effect: There are more code lines than in the other two patterns. Although in common module most of the code is the same for each pattern, it may differs in some parts. As an example - repository. In the long term when we will have 25+ screens we get an exponential amount of code lines in total. The reason for that, our common module will be mostly the same despite the amount of screen, while we will write different fragments, presenter classes and view interfaces in a runnable module. Regarding the other two patterns, they have approximately the same amount of code, but the Model View Intent is more complex and has more code lines for both modules.

In terms of man-days, we also need to mention the complexity and the maintainability of the code. Usually, man-days depend on the complexity of the implementation. The reason for that is we will waste less time using the proper architecture pattern. Thus, we need to consider here complexity and maintainability.

Table 5.1: Code lines of each pattern

Patterns	Common module	Runnable Module
MVP	513	732
MVVM	590	565
MVI	622	607

Table 5.2: Implementation testing

Patterns	Complexity	Maintainability	Testability
MVP	Standard	Standard	Standard
MVVM	Less than MVP	Better than MVP	Better than MVP
MVI	More than MVVM	Better than MVP	Better than MVP

5.2 Implementation Complexity

If we look at table 5.2 called implementation testing we can see four columns: patterns, complexity, maintainability and testability. Here we will take the MVP approach as the standard from which we will deviate.

The *complexity* is the way how we understand the difficulty of implementation. In this qualification, we will count how hard it is to understand and implement each of the architecture patterns. We also need to take into attention that each architecture patterns also has different approaches. Thus, the difference between approaches may lead to misunderstanding for developers.

As we can see from the table, the complexity of the MVVM is less than MVP because we do not need to create the view interfaces for each screen as in MVP. Additionally, we can use one ViewModel for several screens, while we cannot do the same with MVP and MVI approaches. Unfortunately, it is not so easy to reuse ViewModel-s in MVP, since we have State, which can be quite specific. Furthermore, there are different approaches of the MVI architecture, which make it to understand less obvious than MVVM.

5.3 Maintainability

The *maintainability* is the process of keeping the application as better as his first version. Meanwhile that, it means the application which we develop should have the same coupling level. The less is maintainability, the less its possible to update the application. If we lost in maintainability, it also means to get less efficiency in the architecture pattern which we decided to use in the project. In other words, architecture patterns are useful for keeping the maintainability of the application. Maintainability depends on the developers and the architecture pattern they use.

The MVVM and MVI patterns will be better in maintainability than MVP. Each update for our application will take less time to implement because we

do not need to write view interfaces and communication processes, easy to test and less complex to change the existing code. From a long term perspective, we will get much more code than in the other two patterns.

5.4 Testability

Testability is the process where we are going to define it as the degree of efficiency and effectiveness with which test criteria can be established for a product/component, system and tests can be performed to determine whether those criteria have been met. The indication of how easy can be the software to test is testability.

The *Testability* has the same result as a *Maintainability.* We cannot cover the whole app with a test as in MVVM and MVI. They are better in testing because you need to write test cases for both the ViewModel and Model layer without the need to reference the View and mock its objects. The weakness of MVP compared with MVVM is you have reference to the View in your Presenter layer so you need to struggle with views reference in your Presenter unit tests.

Chapter 6

Conclusion

In conclusion of this thesis, this chapter will summarize and make final results on both theoretical and practical parts of work that has been done.

In this thesis, we described and showed the implementation parts of three architecture patterns as well as their testability of them. We figured out which architecture approach has more code lines and made a decision about the complexity, the maintainability, the repeatably and the testability.

To go through the pros and cons of each architecture patterns, the main drawback of the MVP is that we have to create a View interface for each screen. On large projects, we will have a lot of unnecessary code and files that make it difficult to navigate through the packages. Moreover, the Presenter is difficult to reuse, since it is tied to the View, and it can have specific methods. The advantage of this approach is that we can implement this pattern in small projects and has fewer man-days and code lines.

One of the advantages of the MVVM is the View no longer has an interface, as it simply subscribes to observable fields in the ViewModel. Also, the ViewModel is easier to reuse since it knows nothing about the View. It follows from the first advantage. As a drawback, we can mention the complexity to understand for juniors and more time-consuming in the first stages of the project.

In MVI, we also don't need to create a bunch of contracts for the View. You just need to define the render (State) function. However, it is not so easy to reuse this, since we have State, which can be quite specific depends on the screen. Another advantage is in MVI, we have a certain state that we can change centrally through the reduced function. This allows us to track changes in state. For example, write all changes to the log. Then we can read the last state if the application crashed. Plus State can be persistent to handle the death of a process.

Based on all the information above, I would advise choosing between MVVM and MVI when designing your application. This will give you a more modern and convenient approach to Android realities. Additionally, we can

mix both MVVM and MVI architecture patterns in one project. In that case, we still get the benefits of MVI and MVVM. For example, we can use shared ViewModel-s as a benefit of MVVM and use State-s in big main screens with lots of logic.

Bibliography

[1] Statista, ”Mobile operating systems’ market share worldwide from January 2012 to January 2021.”, February 2021. [Online]. Available: https://www.statista.com/statistics/272698/global-market-share-held-by-mobile-operating-systems-since-2009/

[2] Android, ”About the Android Open Source Project”, [Online] Available: https://source.android.com/

[3] Michael Long, ”Why Flutter Isn’t the Next Big Thing”, [Online] Available: https://betterprogramming.pub/why-flutter-isnt-the-next-big-thing-e268488521f

[4] G. E. Krasner and S. T. Pope, ”A Description of the Model-View-Controller User Interface Paradigm in the Smalltalk-80 System,” Journal of object oriented programming, vol. 1, no. 3, pp. 26 - 49, 1982.

[5] S. Burbeck, ”Applications programming in smalltalk-80 (tm): How to use model-view-controller (mvc),” Smalltalk-80 v2, vol. 5, 1992.

[6] Android, ”Android Architecture Blueprints v2” [Online], Available: https://github.com/android/architecture-samples

[7] Android, ”Android Jetpack”, [Online], Available: https://developer.android.com/jetpack

[8] Luca Mezzalira, ”What Developers Need to Know about MVI (Model-View-Intent)”, 5 May 2016, [Online], Available: https://thenewstack.io/developers-need-know-mvi-model-view-intent/

[9] M. Potel, ”MVP: Model-view-presenter the taligent programming model for C++ and java,” Taligent Inc, p. 20, 1996.

[10] M. Fowler, ”GUI Architecture,” 18 July 2006, [Online], Available: http://martinfowler.com/eaaDev/uiArchs.html

[11] Android, 2017, ”TODO MVP App”, [Online], Available: https://github.com/android/architecture-samples/tree/todo-mvp

[12] Moxy Community, 2019, ”Moxy”, [Online], Available: https://github.com/moxy-community/Moxy

[13] Alexander Blinov, Aug 7, 2016, ”Android with no Lifecycle: MPVsV approach with Moxy library”, [Online], Available: https://medium.com/redmadrobot-mobile/android-without-lifecycle-mpvsv-approach-with-moxy-6a3ae33521e

[14] J. Grossman, ”Introduction to Model View ViewModel”, 2005.

[15] C. Anderson, ”The MVVM Design Pattern”, 2012.

[16] Google, ”Data Binding Library”, [Online], Available: https://developer.android.com/topic/libraries/data-binding

[17] Google, ”Two-way data binding”, [Online], Available: https://developer.android.com/topic/libraries/data-binding/two-way

[18] Francesco Stranieri, April 28, 2018, ”Android DataBinding basics: one-way, two-way and handler”, [Online], Available: https://medium.com/@frankStrangerZ/android-databinding-basics-one-way-two-way-and-handler-2dcef824aa0c

[19] Ashvin Bera, ”WPF with MVVM – Easily Separate UI and Business Logic”, [Online], Available: https://www.clariontech.com/blog/wpf-with-mvvm-easily-separate-ui-and-business-logic

[20] Cycle.js, ”Official website of Cycle.js”, [Online], Available: https://cycle.js.org/

[21] Cycle.js, ”Model-View-Intent”, [Online], Available: https://cycle.js.org/model-view-intent.html

[22] Yun Cheng & Aldo Olivares, ”Advanced Android App Architecture”, Apr 28 2019, Section 3.

[23] Hari Sudhan, Feb 10, 2019, ”MVI - a Reactive Architecture Pattern”, [Online], Available: https://medium.com/quality-content/mvi-a-reactive-architecture-pattern-45c6f5096ab7

[24] Oozou, ”Android MVI Architecture With Data Binding”, [Online], Available: https://oozou.com/blog/android-mvi-architecture-with-data-binding-148

[25] Timo Tuominen, April 27, 2019, ”RxJava for Android Developers: With ReactiveX and FRP”.

[26] Android, ”Create an Android project”, [Online], Available: https://developer.android.com/training/basics/firstapp/creating-project

[27] Ricardo Costeira, ”Multi-Module Apps”, [Online], Available: https://www.raywenderlich.com/books/real-world-android-by-tutorials/v1.0/chapters/8-multi-module-apps

[28] Mishaal Rahman, April 10, 2020, ”Android Version Distribution statistics will now only be available in Android Studio”, [Online], Available: https://www.xda-developers.com/android-version-distribution-statistics-android-studio/

[29] Dagger, ”Hilt Documentation”, [Online], Available: https://dagger.dev/hilt/

[30] Google, ”Room Documentation”, [Online], Available: https://developer.android.com/jetpack/androidx/releases/room#declaring_dependencies

[31] Google, ”Navigation Documentation”, [Online], Available: https://developer.android.com/guide/navigation/navigation-getting-started

[32] Wikipedia, ”Observer pattern”, [Online], Available: https://en.wikipedia.org/wiki/Observer_pattern

Appendix **A**

Sources

- `readme.txt` ...... the file with CD contents description
- `images` ...... the directory of images of thesis
  - `*.png` ...... the images used in thesis
- `project` ...... the project of the thesis
  - `common` ...... the common module of the application
  - `mvp` ...... the mvp module of the application
  - `mvvm` ...... the mvvm module of the application
  - `mvi` ...... the mvi module of the application
- `thesis.pdf` ...... the Diploma thesis in PDF format
- `thesis.tex` ...... the LaTeX source code files of the thesis

Appendix **B**

List of abbreviations

OS Operation System
IDE Integrated Development Environment
HTML HyperText Markup Language
CSS Cascading Style Sheets
MVC Model View Controller
MVP Model View Presenter
MVVM Model View ViewModel
MVI Model View Intent
UML Unified Modeling Language
XML Extensible Markup Language
API Application Programming Interface
SDK Software Development Kit
DAO Data Access Object
DI Dependency Injection