Design Patterns in Mobile Architecture

**MARIA KRITOU**

Mobile coursework

Part IV

Table of Contents

[**1.** **Design Patterns** 1](#_Toc74524762)

[1.1 MVVM 2](#_Toc74524763)

[1.2 MVI 3](#_Toc74524764)

[1.3 Comparison of MVVM & MVI 4](#_Toc74524765)

[**2.** **Implementation of MVVM** 5](#_Toc74524766)

# **Design Patterns**

In order to produce code that is more dynamic, maintainable with a modular design we use design patterns. Many design patterns and android architectures exist but we are going to focus on 2 of them, briefly analyze and compare them.

The 2 design patterns are the MVVM and MVI.

# MVVM

MVVM stands for Model-View-ViewModel and is a design pattern that was derived from the MVP and its core purpose is to completely separate the data presentation (the UI) from the business logic of the application.

It consists of three parts/layers :

* Model : This part includes all the data related components like classes, API, DTO but also the repository.
* View : This layer is responsible for informing the ViewModel on the user’s actions. It represents the UI part, the information that is currently visible to the user.
* ViewModel : This part is the connecting link between the Model and the View. It provides the data that are required in the View layer which are stored in the Model.

Advantages of MVVM :

* Separates the UI logic from the business logic which makes code easier to maintain.
* No reference exists from the ViewModel to the View which makes the ViewModel reusable.
* Reduces the coupling in UI and business logic making the testing more manageable.

Disadvantages of MVVM :

* Generalizing ViewModel gets more difficult depending on the scale of the application whilst for small simple apps this model might seem like an overkill
* Sometimes the code can be seen in XML which complicates the development and the debugging but also confuses the developer.
* The user is provided with only 2 ways on working with the View, by using data binding or with View methods.

# MVI

MVI stands for Model-view-Intent it is similar to MVVM model with 2 new concepts (intent & state) but it uses an unidirectional data flow inspired by the Cycle.js framework : View > Intent > Model > View.

It consists of three parts/layers :

* Model : This layer represents a state which also is immutable to ensure that the update is done only from place. A state could be the data that needs to be displayed or a change in the UI through actions.
* View : This part represents the interface, like the activity or fragment. Observable intents are used in response to a user action.
* Intent : Here we pass as input values the user actions. As the name suggests it just describes an intention generated by the user.

Advantages of MVI :

* Since it has an unidirectional flow, the data can be tracked easily and makes the debugging less complex.
* Testing the application is not that much of a hassle since each state has each own business logic which can be mapped easily.
* Since the focus with this model falls on the states their maintainability and management is not a challenge any more.

Disadvantages of MVI :

* The maintainability of the states is easier but since now we have to maintain a state for each action some boilerplate code will occur.
* Many objects need to be created for all these states which might result in putting a bit of a strain on the app’s memory management.

# Comparison of MVVM & MVI

Both of the design patterns have some common grounds. Firstly, the most important aspect of the patterns in general, the separation of concerns, is achieved by the use of MVVM or MVI. Each pattern follows the creation of multiple components each with their individual role.

Furthermore, as we mentioned in their advantages, the reduce of the coupling is a result of both of the patterns since every component does its individual job and owns a well-defined structure. For this reason, the testing for each component is not that much of a challenge.

Now, even though MVI offers a strong adhesion to the single source of truth and the data are well represented in the View with the use of the View State still there is quite the complexity in the creation and maintenance of many state reducers and also involves some lifecycle problems. Regarding the latter, the MVVM pattern comes to solve it, it might be difficult to implement but lifecycle problems will not occur on changes. On top of which MVVM can have multiple Views while using the View Model.

In conclusion, MVVM and MVI have quite a high complexity for the developer to come against but once this is pulled though with the MVVM pattern, the application will require a minimal effort for its maintenance. So, in my belief I think MVVM should be used also in my project since it is not a big application, and its implementation will not be that hard while at the same time I will gain a high level of maintenance and testing ability.

# **Implementation of MVVM**

**STEP 1.**

The first thing to do is to change the project structure like below, as we mentioned in the beginning the data related files should go together in one place, Model. Of course, UI will go into the View and all the View Models inside the ViewModel folder :

*(bold🡪 folders , normal🡪 files)*

* **MoviesProject**
  + **Model**
    - Item
    - MovieDetails
  + **View**
    - MainActivity
    - LoginActivity
    - **Fragments**(as it is in the project)
      * **Adapters**
        + MovieAdapter
        + ViewPageAdapter
      * FavoritesFragment
      * HomeFragment
      * SearchFragment
  + **ViewModel**
    - HomeViewModel
    - SearchViewModel
    - FavoritesViewModel

**STEP 2.**

We also need to add to build.gradle file the dependency :

**implementation** "androidx.lifecycle:lifecycle-viewmodel- ktx:$lifecycle\_version”

**STEP 3.**

Making the :

HomeViewModel.kt , SearchViewModel.kt, FavoritesViewModel.kt

These will update the fragments and get the data from the models. In order to

do that we have to create a model object inside each ViewModel that will

hold the data for our UI, also create a MutableLiveData to which the

fragments can subsribe to and whenever a change is made to the data, it will

trigger the UI to do an update

**STEP 4.**

Next, we have to change a bit the fragments in order to listen to the

ViewModels for updates on the UI.

HomeFragment :

setDataList function should be implemented from the HomeViewModel and

an Observer in the fragment would update the UI, also for the onItemClick the

click listener should call a method with this functionality in ViewModel

(call these from onViewCreated)

The same thing applies to Search and Favorites fragments.