MJRET

OPEN ACCESS

Multidisciplinary Journal of Research in Engineering and Technology

www.mjret.in

eISSN:2348-6953

Android Design Patterns

[1]Mr. Jatin Yadav,[2] Dr. Abhijit Banubakode

[1,2] MET Institute of Computer Science,
Bandra West,Mumbai,India.

Abstract: *Developing Android applications which are maintainable has been a demanding scope ever since the first android smartphone was published. With an ever-growing community of open source contributors and better hardware, Android development has seen extreme changes over the earlier years. That demands more maintainable software which can be complex also. Model View Controller, Model View Presenter, and Model View ViewModel are three well-liked and well-studied software architectures which are widely used in GUI-heavy applications, these architectures have now also developed in Android development. In this work, we analyze widely used architectural design patterns and recommend an architecture model accommodated to Android development.*

Keywords*: Smart mobile devices (smartphones, tablets); design patterns; Model-View-Controller; Android architecture model; Fragments; Android passive MVC*

1. INTRODUCTION

The mobile market has rapidly in recent years. Many companies feel the need to be present in mobile markets and introduce their services with mobile applications. Compared to computer programs, mobile apps often have narrowed functionalities, smaller shelf life, and cheaper price. New applications should develop fast to be cost-effective and updated often to keep users engrossed. The quality of the apps should not be dismissed, as mobile users are very particular and competition is strong. Architecture choice remains crucial for mobile applications to ensure quality: mobile applications, as well as other systems, could be complex and evolve.

Books for developing Android tutorials are mainly focused on technical details of the Android SDK and user interface. Only a few tasks are dedicated to the creation of the Android application, while the Android community points to the architecture as an integral part of system design and development. Developers are opening up more discussions about the proper Android builds on forums, blogs and groups.

In this work, we provide an overview of some widely used architectural patterns and suggest MVC architecture specific to the Android system. Android Passive MVC simplifies the development process and provides guidelines and solutions for standard Android functions that enable the creation of minimal, complex, expandable and sustainable resources.

2. ARCHITECTURAL DESIGN PATTERNS

We present three architectural design patterns in the traditional sequence. These are the patterns used in desktop and web application development. If mobile development grasps a similar design, developers moving from other systems could take advantage of their knowledge.

2.1. Model-View-Controller (MVC)

MVC is divided into three areas with different responsibilities:

Model

The Model handles business logic and often consists of objects which describe the system (for a hotel system this could be Order, Transaction or Customer).

View

The View means the user current state of the system, the form of a UI.

Controller

The Controller handles user interaction and updates the Model.

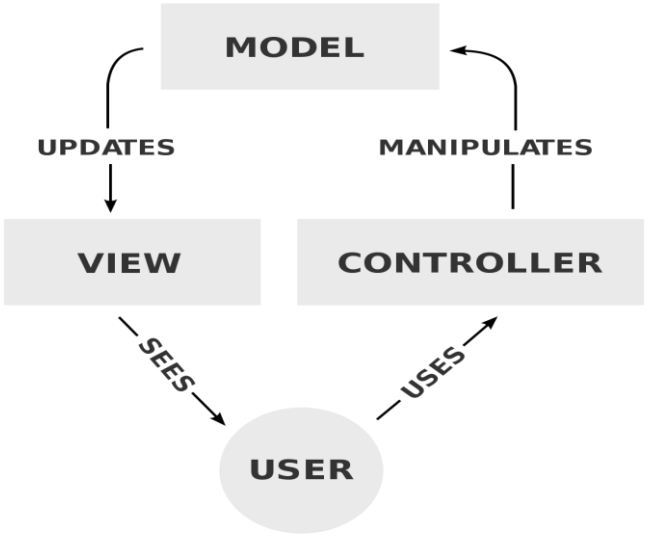

Figure 1: MVC

The following is an example that is used to explain how these things work together: In an application that is used to convert finances a user can invest and read the converted currency in the UI. The user first contacts the controller and passes the amount to be converted. The controller then transmits the input to the User in the Model, which takes care of the business concept The Model also alerts the viewer in a way that the View can refresh itself and display the converted currency.

2.2 Model View Presenter (MVP)

It is derived from the MVC but aims to strengthen the separation between the Model and the View-Controller. This can be represented as two basic ideas; data management and UI. The biggest difference compared to MVC is the way the idea is handled. Instead of having a controller running the user input, the View registers user actions and passes them to the Presenter. The presenter can interact with the Model. When the Model is updated it notifies the affected view that is being updated to

reflect the current state of the system. This puts the dependency on the look and model in the same way as the MVC.

To remove dependencies between Model and View, MVP can implement Passive View. This makes a complete overview that doesn't mean you don't need to update yourself on the Model. It is the job of Presenters now to manage the user input (transferred to view) and to update the view to reflect the current status of the program. The relationship between Model, View, and Presenter using the same model and currency as in MVC Status Modification with Passive View construction becoming more flexible, it is possible to replace the simple one. elements are Model, View, or Presentation. Because Views are only updated with Presenter this also makes testing easier.

2.3 Model View ViewModel (MVVM)

Unlike the MVP where the View does not know about its Presenter, in this case, the View does not know about the Presentation Model, but the Presentation Model knows nothing about views.

Upgrade to Presentation Model to extend user interaction and interact with Model. To refresh the View to show status changes, some kind of syncing method needs to be performed. This introduces unnecessary boilerplate code, the problem is addressed by MVVM.

The ViewModel has the same functions as the Presentation Model but uses a standard data method that binds to easily synchronize with the View. MVVM tries to separate the UI and Model build obligations. Gossman means that software developers should focus on building ViewModel and Model, while designers and user experience developers focus on visualization. Views are usually structured and organized in some kind of XML-enabled language, which usually does not need to be packaged in the same format separately from another program. The parts of the view are tied to the ViewModel using the Data binding, which makes it possible to combine the two parts. In addition to representing the view state, ViewModel also converts data objects from Model so that View can directly use these; e.g. to display data in the UI.

3. IMPLEMENTATION DETAILS

Even though the applications are implemented with various architectures, they deal with implementation specifications. As discussed in Section 2, MVC, MVP, and MVVM all have similarities, but what important is how the view is managed. Therefore, the model for all three applications is the same. The sample applications contain the same Entities; Movie. A movie is a simple Java class representing movie details. The class holds basic information about a movie such as its id, username, profile picture, and an URL to their repositories. These Entities are the business rules of the applications, the information which they hold are crucial for satisfying the functional requirements.

The applications interact with the Entities by using Use Cases. A Use Case does what its name suggests, it is a class representing a use case of the application. Use cases are tightly coupled with the functional requirements. For example, the requirement “The user must be able to search for a movie” is reflected in the application with a SearchMovieUseCase. To reduce complexity, all Use Cases are injected with Dagger 2 to the Presenter/Controller/ViewModel (depending on the architecture). Every Use Case takes a Subscriber as an argument, and the Subscriber is implemented as an anonymous class that receives the data from the Use Case. To search for a Movie, a call to SearchMovieUseCase is made with the second parameter specifying the search query:

```
searchMovieUseCase.execute(new SearchMovieSubscriber(), moviename);
```

The SearchUserSubscriber is implemented in the same class which makes the call to the Use Case, and can look like the following (excerpt from the MVP application):

```
private final class SearchMovieSubscriber extends DefaultSubscriber {

@Override public void onCompleted() {

        searchMovieView.hideProgressIndicator();

}

@Override public void onError(Throwable e) {

        searchMovieView.hideProgressIndicator();

        searchMovieView.showMessage("Error loading movie");

}

@Override public void onNext(SearchResponse searchResponse) {

List movies= searchResponse.getMovies();

if (movies.isEmpty())

{

        searchMovieView.showMessage("No movies found");

}

searchMovieView.showMovies(movies);

}

}
```

When all requested data is received or an error has occurred Subscriber-implementation decides what should be done. Three arguments need to be provided; MovieService, ThreadExecutor, and PostExecutionThread for creating a new Use Case. MovieService is an interface describing the web API which is used to fetch and update movie Entities. ThreadExecutor and PostExecutionThread are places to provide Case Use organizers. The three meeting places are provided by dagger 2.

3.1 Model View Controller

An overview of how the MVC application handles is shown in the Figure 2. XML layout files in conjunction with Fragment make Views. The controller contains fragments and functions. This can separate the ties from each other, the function usually starts in the other functions while the Fragment handles calls to the Model. This Model contains MovieDBService, which captures information on the web. From Section 2.2.1 it is clear that the view should be made aware of by the Model, and then update itself. That's why Fragments acts both as a Viewer and a Controller when a call to the Model Subscriber is provided as a callback. This Subscription serves as an anonymous class in Fragment,

and when the model is made to download the data you are subscribed to it is called and can update the View with Control. If the subscriber is considered to belong to the Model, Figure 2 is more straightforward.

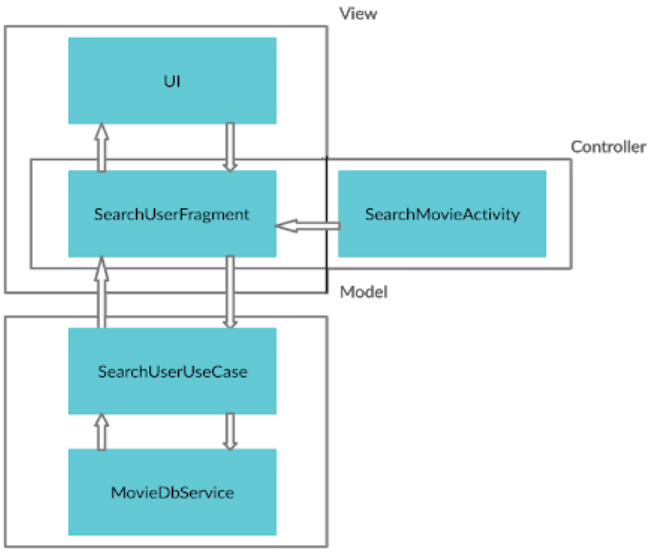

Figure 2: Implementation of MVC

3.2 Model View Presenter

The pieces function as Passive Views, transmitting updates, and running in the Presenter. Fragment responsibility should be for UI design only and for registering user interaction. The registered communications must be forwarded directly to the Co-Presenter. Views transmit user interaction to the Presenter which when downloading data from Model and updates the view when ready. It may be good to include tasks in Views, but this is difficult due to the responsibilities of implementing a new view.

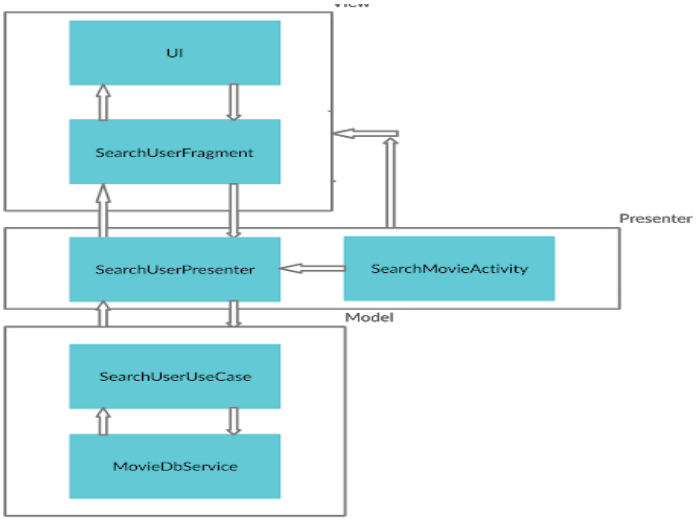

Figure 3: Implementation of MVP

3.3 Model View ViewModel

From Figure 4, the UI is the only part that works as a view. This is an XML file that describes different UI elements, and by binding data, it is possible to define the methods that reside in SearchUserViewModel. ViewModel also includes SearchUserFragment and SearchUserActivity, since these interact with SearchUserViewModel to access specific parts of the Android framework. This makes SearchUserViewModel free from any dependency from the Android framework, which enables JUnit to be tested.

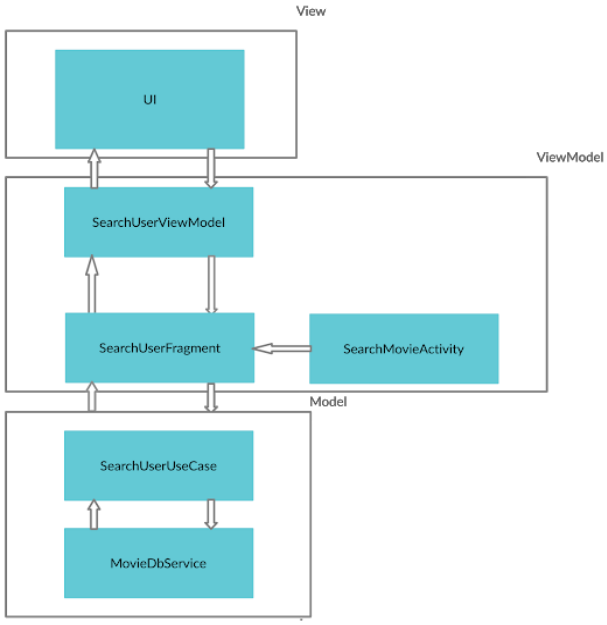

Figure 4: Implementation of MVVM

4. CONCLUSION

MVC is best suitable for smaller applications whereas MVP and MVVM are preferable for larger and more complex applications.Data Binding, used by MVVM, is a very powerful paradigm to use in heavy UI applications. Looking at the Android framework directly, integrating data makes it easy to map properties from an XML file containing UI elements in Java code that contain logic and listeners such as onClickListeners. Binding data also removes the main boilerplate code from adapters like the RecyclerView-adapters, which makes the complex list more flexible and is being tested. The MVP, on the other hand, does not use any particular library, it is commonly used Java classes to control Fragments and Activities. There is also a clear border between that should be managed with the Android framework and what should be handled Introducers, no dependency on the Android framework.

REFERENCES

[1]. Google www.google.com.
[2]. Microsoft. The MVVM pattern docs.microsoft.com/MVVM.
[3]. Wikipedia. MVC — en.wikipedia.org/wiki/Model view controller.
[4]. Movie DB Service — www.themoviedb.org/documentation/api.
[5]. Jan Bosch. Software architecture: The next step. In Software architecture, pages 194–199. Springer, 2004.