A comparison of Android Native App Architecture

MVC, MVP and MVVM

Lou, T.

Award date:
2016

Awarding institution:
Aalto University

Aalto University
School of Science
Master's Programme in ICT Innovation

Tian Lou

A Comparison of Android Native App Architecture – MVC, MVP and MVVM

Master's Thesis

Espoo, September 06, 2016

Supervisor: Professor Heikki Saikkonen, Aalto University

Advisor: Håkan Mitts, M.Sc.

Aalto University
School of Science
Master's Programme in ICT Innovation

ABSTRACT OF
THE MASTER'S THESIS

Author: Tian Lou

Title: A Comparison of Android Native App Architecture - MVC, MVP and MVVM

Number of pages: 45 | Date: 06/09/2016 | Language: English

Major: Service Design and Engineering

Supervisor: Professor Heikki Saikkonen, Aalto University

Advisor: Håkan Mitts, Aalto University

Fast iteration in Android application is a big challenge for development efficiency and quality that are influenced by architecture. It is claimed that Model-View-Presenter (MVP) and Model-View-ViewModel (MVVM) are better than the default Android native app architecture Model-View-Controller (MVC). But there is no empirical data to support this point of view. In addition, refactoring the project to adopt the new architecture requires a lot of efforts from development to test. Thus, some app organizations are skeptical of employing MVP and MVVM in Android.

This thesis aims to provide a thorough analysis to find out whether MVP and MVVM architecture are better than MVC from quality perspective.

To answer this question, Architecture Tradeoff Analysis Method are selected. Then, we set three criteria: testability, modifiability and performance. For each quality attribute, key factors are recognized. Comparison is based on these selected criteria.

Analysis and experiments show MVP and MVVM have better testability, modifiability (low coupling level) and performance (consuming less memory). That is, MVP and MVVM are better than MVC on the selected three criteria. But for MVP and MVVM, there is no evidence showing that one is superior to another. These two architectures have similar performance, while MVP provides better modifiability and MVVM provides better testability.

Keywords: Android App Architecture, ATAM, MVC, MVP, MVVM, Testability, Modifiability, Performance, Tradeoff

Table of Contents

Table of Contents iii

List of Figures v

List of Tables vi

Acknowledgements vii

1 Introduction 1

1.1 Background 1

1.2 Motivation and research problem 2

1.3 Structure 3

2. Literature Review 4

2.1 Software Quality 4

2.2 Software Architecture 4

2.3 ATAM 5

2.4 Android Concepts 8

3 Android Native Application Architectures 10

3.1 Model-View-Controller 10

3.1.1 MVC architecture 10

3.1.2 MVC Implementation in Android 13

3.2 Model-View-Presenter 14

3.2.1 MVP Architecture 14

3.2.3 MVP Implementation in Android 16

3.3 Model-View-ViewModle 18

3.3.1 MVVM Architecture 18

3.3.2 MVVM implementation in Android 20

4 Testability 23

4.1 Test in Android 23

4.2 Testability Criteria 24
4.3 Evaluation 25
4.3.1 MVC 25
4.3.2 MVP 25
4.3.3 MVVM 26
4.3.4 Evaluation results 26
5 Modifiability 27
5.1 Modification 27
5.1.1 Modification scenario 27
5.1.2 Modification criteria 28
5.2 Evaluation 29
5.2.1 Cohesion level 29
5.2.2 Coupling level 30
6 Performance 34
6.1 Criteria 34
6.2 Evaluation 35
7 Results analysis 37
8 Conclusion 39
References 41
Appendices 46
Appendix A: “What’s New” log 46
Appendix B: Screenshot for to-do app 49
Appendix C: Memory usage experiments results 49

List of Figures

Figure 1. ATAM evaluation flow 6
Figure 2. MVC components relations 11
Figure 3. MVC components interaction [14] 11
Figure 4. MVC flows 13
Figure 5. MVC Android implementation UML class diagram 13
Figure 6. MVP components 15
Figure 7. MVP components interaction 16
Figure 8. MVP class diagram 17
Figure 9. MVVM interaction 19
Figure 10. MVVM implementation in Android 21
Figure 11. Delete news sequence diagram 31
Figure 12. Memory usage for each architecture application 36

List of Tables

Table 1. Testability for MVC, MVP and MVVM 26
Table 2. Cohesion level 28
Table 3. Coupling level 29
Table 4. MVC coupling level 32
Table 5. MVP coupling level 32
Table 6. MVVM coupling level 33
Table 7. Allocated memory for application per minutes 36

Acknowledgements

I would like to express my gratitude to my supervisor Heikki Saikkonen and advisor Håkan Mitts who provide valuable tips. These tips gives me the direction.

Then, I would like to thank the KLM mobile team where I did my internship and wrote my thesis. My instructor Ovidiu Lutea always supported my decisions and tried his best to help me. The internship experience in the development team is a good start for my work career.

Besides, I'd like to thank my friends in Aalto University. Specially thank Taufik A. Sitompul. I did this thesis in the Netherlands. He helped me to communicate with the teachers in the University.

Amstelveen, September 2016
Tian Lou

1 Introduction

Android, as an open source mobile operating system, has achieved big success. According to statistics from the research company Gartner, Android operating system took 80.7% of the market share on Q4 in 2015 [1]. Currently, there are more than 2 million applications available on Google Play [2]. What's more, Google Play got more than 65 million installations in last year.

These applications on Google Play are native applications who require APK package for installation. Except for native apps, there are also web apps and hybrid apps which are based on web technology, e.g. HTML5, CSS and JavaScript. Typically, such apps are based on web browser. Thus native app is more powerful than web app since it interacts directly with the mobile device hardware. The thesis focuses on native apps.

1.1 Background

Mobile applications have high update frequency. Research on 100 mobile developers' work shows that it takes about 18 weeks to develop the first version of the native app [3]. After that, the most successful app will update 1 - 4 times per month to keep up with market pace [4]. This is a big challenge for mobile developers since they need to finish the development in a short time and meanwhile, provide quality assurance. Work efficiency and quality are related to the programmers personal skills. However, there are some external methods to improve it.

For instance, in 2013, Google released the development IDE Android Studio to replace the Eclipse. On Google I/O 2016, Google announced the new function instant run to speed up the compile process. The third-party open source libraries avoid developers to re-invent wheels and save them much efforts. Kotlin language released the first version which is more concise and small. Later, Google released the newest Android N SDK that started to support Java 8 to make the programming more efficient. New native app architectures are replacing the default MVC architecture.

1.2 Motivation and research problem

At present, the official Android native app architecture is Model-View-Controller (MVC) pattern. This is a classic pattern that has been successfully implemented in web development. Nevertheless in Android development, this pattern met troubles. View and Controller are tightly coupled which makes maintain and development harder.

Recently, Google released a project -- *Android Architecture Blue Print* on GitHub. In this project, two basic examples was developed in Model-View-Presenter (MVP) and Model-View-ViewModel (MVVM) architecture. Based on MVP, MVVM introduces a new library to reduce the code. So far[1], this repository got about 8,000 starts.

It is claimed that MVP/MVVM is better than MVC architecture. But firstly, there is little article explaining MVP and MVVM architecture on mobile platform. And secondly, there is no empirical data and thorough analysis to support this point of view. Thus, people still doubt if it worth to migrate from the MVC to new architectures. Since such decisions need to refactor the whole project which requires a lot of efforts from development to test. Similarly, for those who have employed MVP, reducing coding work with MVVM seems attractive. Nevertheless, they also worried that this will cause new problems

The thesis aims to fill the research gap of MVP/MVVM architecture on Android platform. Thus, research question in this thesis is:

Are MVP and MVVM better than the MVC from quality perspective?

After the thesis, readers are supposed to 1) understand the MVP and MVVM architecture on Android platform; 2) know about the advantages and disadvantages for MVP and MVVM; and 3) select the their preferred architecture based on their quality requirements.

[1] On July 6th, 2016

1.3 Structure

Section 2 studies a list of existed research papers and derived the comparison process and factors. Sections 3 explains three architecture MVC, MVP and MVVM on theory and practice in Android. Then, the testability, modifiability, and performance are compared separately on the next three sections respectively. Comparison results are analyzed in section 7 and the last section summarizes the thesis.

2. Literature Review

2.1 Software Quality

IEEE defines software quality as the composition of characteristics and attributes for the software to meet the stakeholders' requirements [5]. In software requirement engineering, software quality is also considered as non-functional requirement.

Software quality can be divided into two categories: development requirement and operational requirement. Development requirements are from developer's perspective which must be easy for the development activities, e.g. maintainability and understandability. While the operational requirements are more important for users, e.g. usability and performance.

Software quality is influenced by multiple factors, of which software architecture is the most important one. Medvidovic and Taylor claim that the good software quality is correlated to the good software architecture design [6]. Chen et al. describe the software quality as the *architectural significant requirements* which implies that the architecture has a high impact on the software quality [7].

2.2 Software Architecture

"The software architecture of a program or computer system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them. [8]"

Good software architecture is one of the key factors contributing to software success [6]. In software development lifecycle, the earlier we find the problem, the less it costs to fix it. Architecture design is a perfect phase to find potential problems. It is in the early stage where after requirements collection but before resources commitment [9]. In addition, software system architecture has a significant impact on software quality. Therefore, architecture evaluation is critical for software.

Architecture evaluation aims to ensure that system architecture meets the *business goals* and *software quality requirements* [9] [10]. Researchers have developed a

number of evaluation methods which can be categorized in four types: scenario-based, simulation-based, experience-based and mathematical modeling [7] [8]. Among these types, the scenario-based evaluation method is more mature [10], including ATAM (Architecture Trae-off Analysis Method), SAAM (Software Architecture Analysis Method) and etc.

To date, several studies have reviewed various evaluation methods [9] [10] [11]. Patidar and Suman [10] compare eight different scenario-based methods from various aspects: method's goal, quality attributes, activities, tool support and etc. Clements et al. [9] explain three evaluation methods from four perspective reasons, time, approaches, people and results.

These papers reveal some similarities and differences in evaluation methods. Different evaluation methods have similar work flow. Typically there are four phases: 1) finding goals, 2) converting goals into implementable criteria, 3) analyzing criteria and 4) presenting outputs. While, their differences fall on details. There might be several steps in each phase. Different method presents results in various formats, e.g. score, ranking, documents and etc. Also, selected qualities are different from each other. According to comparison papers [10] [11], most of the methods focus on single attribute: maintainability/modifiability for scenario-based methods and performance for simulation-based methods.

In this paper, ATAM evaluation method is selected based on four reasons. Firstly, ATAM is the scenario-based evaluation which is more mature. Secondly, this method allows us to evaluate more than one quality. Thirdly, it has been successfully implemented by other users. Fourthly, the results of ATAM evaluation show tradeoffs for each candidate architecture which help stakeholders to make better decision based on their requirements.

2.3 ATAM

"Architecture Tradeoff Analysis Method (ATAM) is a method for evaluating architecture-level designs that considers multiple quality attributes such as modifiability, performance, reliability, and security in gaining insight as to whether the fully fleshed out incarnation of the architecture will meet its requirements [12]"

There are four phases in this method: scenario & requirements collection, scenario and architectural views realization, analysis and outputs [12]. Each phase contains multiple steps. Paper [13] lists nine steps and for each step, some documents are provided to show the results to stockholders.

The thesis made some modifications based on ATAM method. This is out of the following two considerations. Firstly, the evaluation methods target the specific software. Nevertheless, the architecture presented in the thesis is generic that are adapted to all Android native applications. Thus, the results are supposed to be generic that are more valuable. Secondly, during the evaluation process, users, developers, testers and other persons are involved. Actually, an evaluation team should be formed which is not feasible in the thesis. The changes will be explained in the remaining part of this section. Figure 1 shows the progress employed in this thesis and the detail explanation for each process is illustrated below the figure.

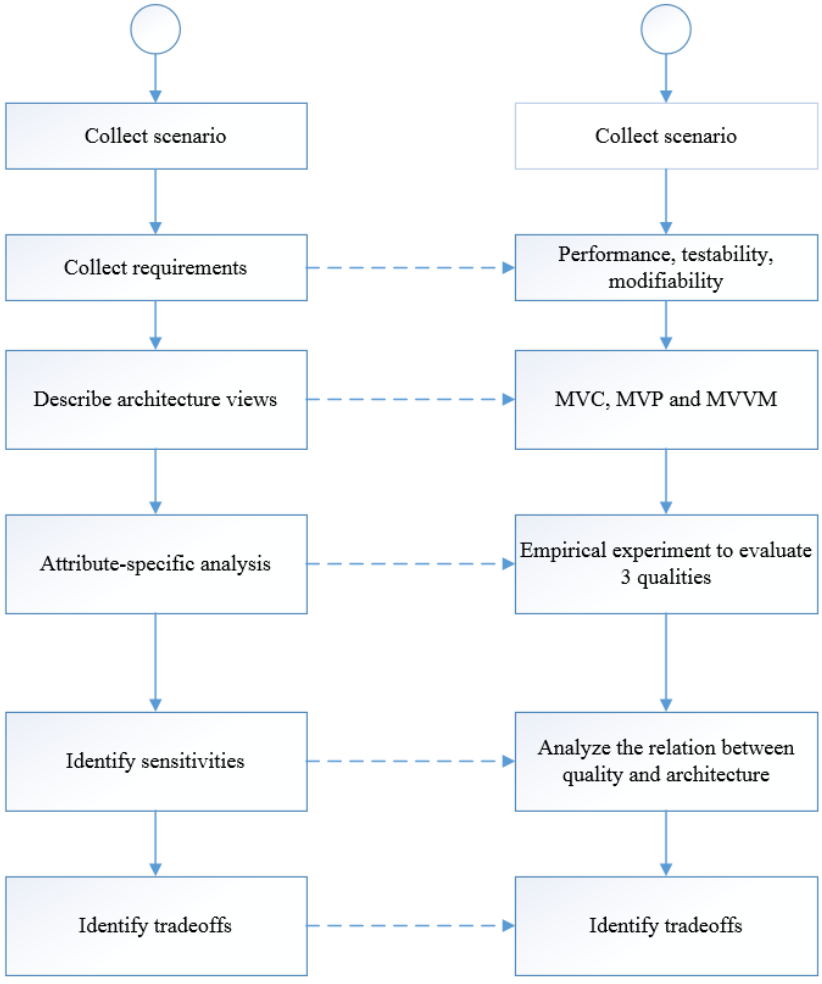


Figure 1. ATAM evaluation flow

Step 1: Collect Scenarios

In this step, the common and important usage scenarios of the system are identified. What kind of problems will the system solve? How does system take actions? This step is performed based on the specific system in real-world context. As stated before, we hope that the results are not software-specific. Thus, collecting scenarios are not quite fit in this thesis since there are no specific software requirements.

Nevertheless, it does not mean that this step is completely ignored. Instead, scenarios collecting are adopted to select the criteria. For given quality, there are many available criteria from different articles. When selecting the most suitable criteria for Android applications, the development scenarios are taken into account.

Step 2: Collect requirements

Requirements collected in this step are evaluation criteria. These requirements are the most important characteristics for this system agreed by all stakeholders. The architecture that meet the requirements will be selected and accepted. In ATAM method, requirements are those attribute-based requirements - the software qualities. Different projects might focus on different qualities. Paper [13] explains the details for performance, modifiability and availability. Paper [11] selected four quality performance, testability, maintainability and portability. Based on the papers and experience, in this thesis, testability, modifiability and performance are selected. These three attributes will be explained in section 4, 5 and 6.

Step 3: Describe Architectural Views

Multiple competing architecture are presented in this step. Each architecture description should explain the core component and attributes that are important to evaluate qualities. In this paper, three architecture MVC, MVP and MVVM will be explained in detail in section 3.

Step 4: Attribute-specific Analysis

This step evaluates selected qualities and presents the evaluation results. In order to make the result clear and persuasive, some empirical experiments are performed. ATAM method does not provide the specific analysis method for each attribute. It

makes the process complicated, because there is no standard to follow. For each quality, the architect have to define their own methods. On the other hand, this makes the analysis process flexible, because we can take actions based on the situation.

Step 5: Identify Sensitivities

Based on the results in step 4, relations between quality attribute (in step 2) and architecture component (in step 3) are analyzed. From this step, we are able to know what kind of changes in architecture will influence the quality attribute.

Step 6: Identify Tradeoffs

The tradeoffs are the gains and losses if we choose one architecture. It is also a description of the relations between the quality attribute and architecture component. But different from step 5, this relation is one to many. For example, in client-server architecture, increasing the number of servers might improve the performance and availability while decrease the security [12].

2.4 Android Concepts

There are three most important concepts in this thesis: Activity, Fragment and Resources.

“An activity is a single, focused thing that the user can do.” Users’ interaction with the application is mostly via activity. In short, an activity is a full screen shown on the devices.

“A Fragment is a piece of an application's user interface or behavior that can be placed in an Activity.” It was first introduced in Android 3.0 (Android 11). From the definition, fragment is similar to activity. The major difference is that activity is the full screen while the fragment is part of the screen. The fragment makes the application UI more flexible. For example, in tablet, the screen usually is split into two parts: left and right. Most probably, the left and right part are two fragments that embedded in to an activity. Fragment is attached to Activity, nevertheless, its life cycle is much complex than the activity.

“Resources are the additional files and static content that the code uses.” There are various types of resources: layout, bitmaps, dimension, translation and etc. The most important resource in this thesis is the layout which determines the static UI of the application to great degree. Layout resource file is in XML format that arranges the some widgets (button, text, checkbox and etc.) positions and appearance.

3 Android Native Application Architectures

This section discusses fundamentals and Android implementation for each architecture. Components and their interactions are described in introduction part. After that, Android implementation is presented with class diagram to capture the framework and data flow.

3.1 Model-View-Controller

Model-View-Controller (MVC) was introduced in smalltalk-80 programming system in paper [14]. This architecture is the first attempt to separate the user interface. It is also the foundation of the MVP and MVVM architecture.

3.1.1 MVC architecture

MVC divides the system into three components, as its name specified, model, view and controller. *Model* component stores data/state in complex object class or simple primary data. It might not store the data in itself but instead, it retrieves the data remotely. *View* component is the graphical user interface seen by users. *Controller* component gets the user input from view, changes the state in model and sometimes, passes the changes back to the view.

Relations among three components are presented in Figure 2. In a perfect design, view and controller are always in pair [15] [14]. They are tightly coupled. In other words, one view has a specific controller and the controller only take charge of its specific view. One view/controller pair has one model, but one model might have more than one pairs.

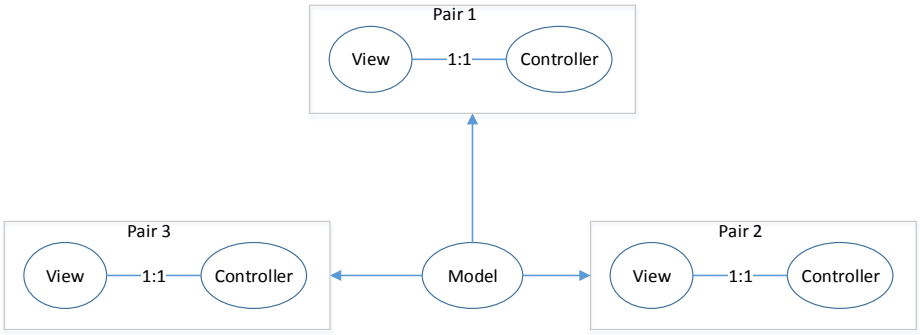


Figure 2. MVC components relations

Once change happens, one component will notify the relevant. For View-Controller pair, the view has an instance of its controller and the controller also has the view instance. Both of them can directly contact model. The only problem is how the model notify all its views when the state stored in model changed. Since one model might have more than one views, therefore, the data update/change activity will have an impact on many views. To solve this problem, in smalltalk-80 programing system, a special object class field to maintain all of this model's dependent (views). In this way, when the data changed, the model will notify all its registered dependent via this field.

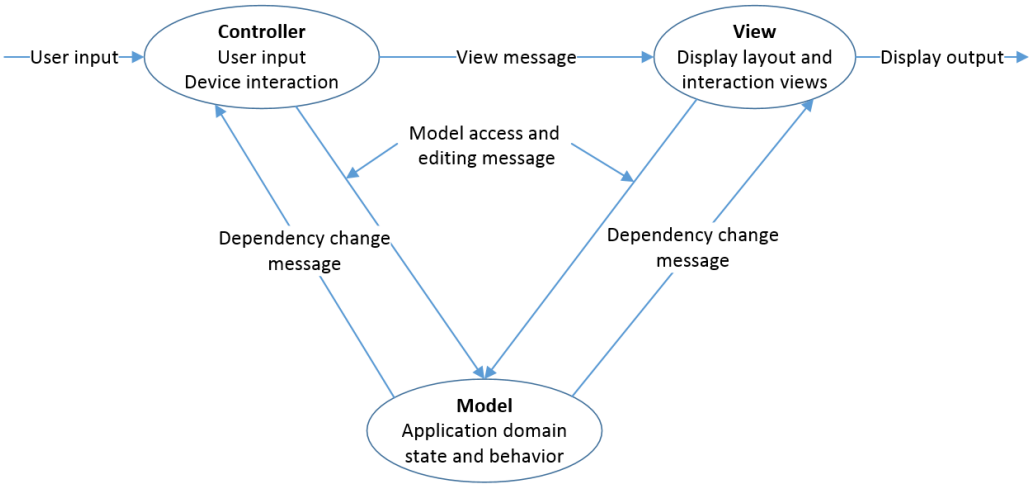


Figure 3. MVC components interaction [14]

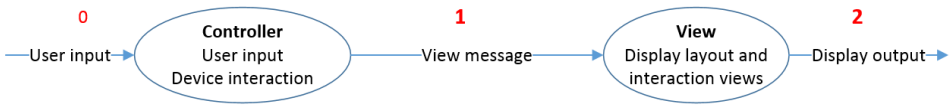
Interactions among three components are displayed in Figure 3. In MVC, the flow starts from the controller component. Firstly, controller component receives user's input (move the mouse or hit the keyboard). All inputs are captured by controller component. After that, flows are various. Three simple scenarios are listed in Figure

4. In practice, the scenario might be a combination of the three situation and more complicated.

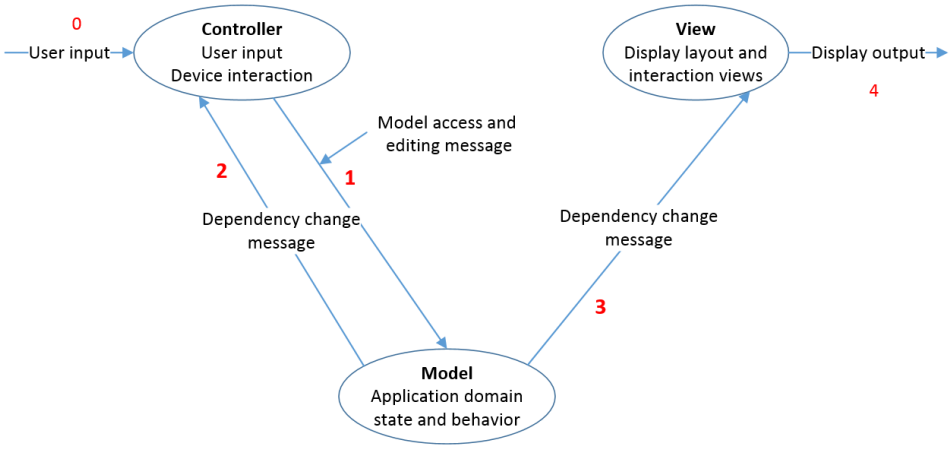
Scenario 1(Figure 4.a): Controller notifies the view to update the state. This is the most direct one. For example, the user clicked the change color command, the color of view will change.

Scenario 2 (Figure 4.b): The controller notifies the model and then the model notify all of its dependent (controller-view pairs).

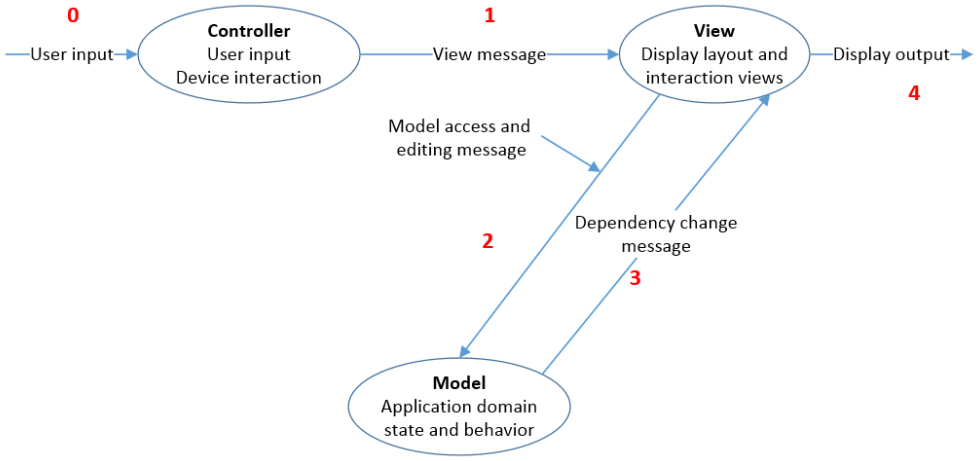
Scenario 3(Figure 4.c): The controller sends message to view. Next, view sends request to model to acquire the most recent states and update itself.



4.a



4.b



4.c

Figure 4. MVC flows

3.1.2 MVC Implementation in Android

MVC architecture is the default architecture for Android native app. Its UML class diagram is described in Figure 5 which shows that view and controller boundary is not clear. Please note that the layout.xml is *NOT* Java class and therefore should not be showed in UML diagram, but it is an important part in architecture. Thus, in this thesis, it is included in class diagram.

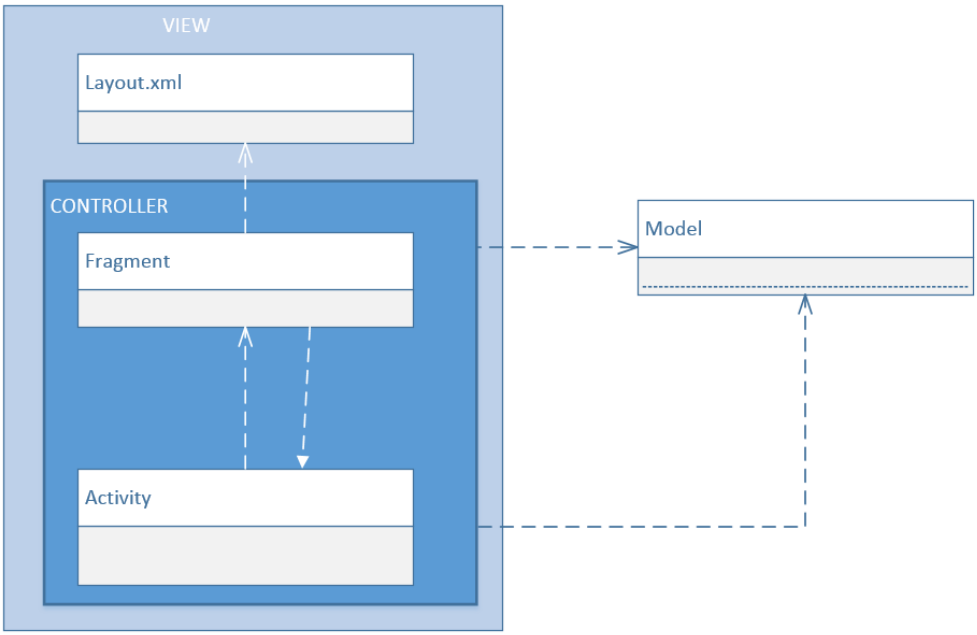


Figure 5. MVC Android implementation UML class diagram

Model is typically plain java object class. It is responsible for business logic. For example, retrieve data (pictures, video, text and etc.) from remote server.

View is the combination of layout resource file and Activity/Fragment. It's not only the layout resource because the layout resource does not have full control over the UI. For example, all widgets in layout files have to be inflated into activities/fragments before they are displayed on screen. In addition, layout file cannot control the visibility of the widget dynamically

Controller is the Activity or Fragment. It captures a serious of events from view and send response back. Or it send requests to model component to get and update data. For example, the click event is captured in onClickListener() method in Activity.

In Android, View component is the xml file and Activity/Fragment. Controller is the Activity/Fragment. Apparently, View and Controller components have overlap in Activity and Fragment.

3.2 Model-View-Presenter

Model-View-Presenter (MVP) was first introduced in Taligent operation system in 1996 [16]. The concept was based on MVC but has more clear separation for each component.

3.2.1 MVP Architecture

MVP was designed based on two questions: data management and UI. That is, how to manage data and how does users interact with the data.

Six components are recognized in MVP architecture: model, selections, commands, presenter, interactor and view [16]. Every component has less but clear duties, they are loosely coupled. *Model* component is the same as the concept in MVC which indicates what the data in this application is. *Selection* component specifies the subset of the data to operate. *Command* component presents a list of actions that can be executed. *View* component is the same in MVC. *Interactor* component indicates the events that will be triggered by the user, e.g. keyboard, mouse move and click, scroll. *Presenter* component is the same as the controller in MVC that aim at

organize and coordinate all intermediate components – interactor, selections and commands. The former three are for data management and the latter three are for UI. Figure 6 [16] is summarizes the 6 components in MVP.

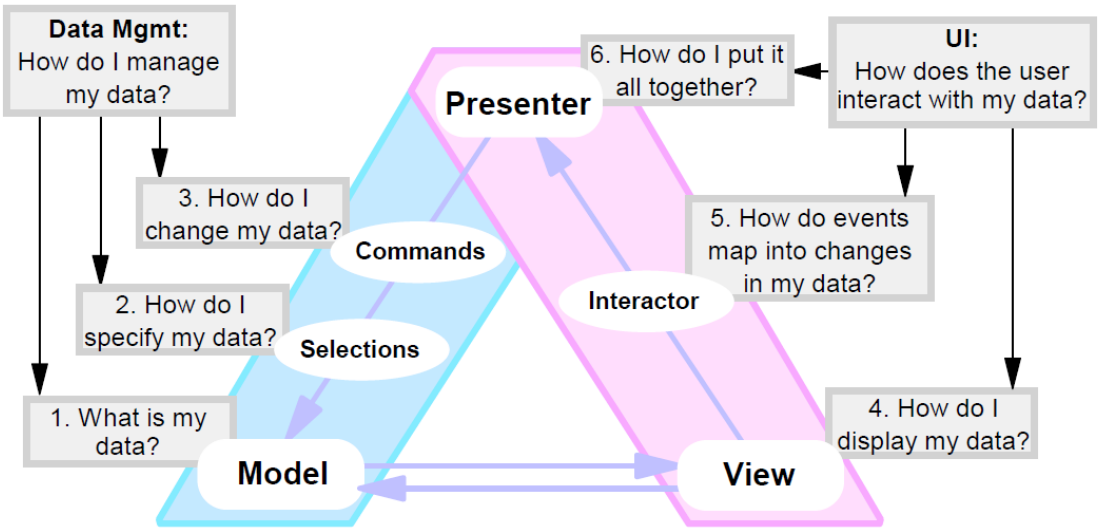


Figure 6. MVP components

To make the concept clear, we employ a simple example to explain – "text editing". One of the common situation is to select the text and then copy, paste. In this scenario, the *Model* is the string. *Selection* is the highlighted text which might be several words, a sentence, a row, a paragraph and etc. *Commands* are the actions shows after the selection, e.g. copy, cut and paste. All these three components are the operations on data. For UI part, the display of the text is the *View* which decides text orientation (vertical or horizontal) and text style (red color, bold). We can simply consider the view as the appearance. If the action is in computer, users first need to click the mouse, move and at last release the mouse. The *Interactor* here is the click, move and release action. *Presenter* is to manage all these components and make it a logical flow.

Although in original MVP paper, six components are recognized, while most time MVP architecture are described as 3 components: Model, View and Presenter. Interactor, commands and selections are put into presenter component.

With MVP, the coupling between the view and model is eliminated. Model knows nothing about the presenter and vice versa. MVP interactions are described in Figure 7.

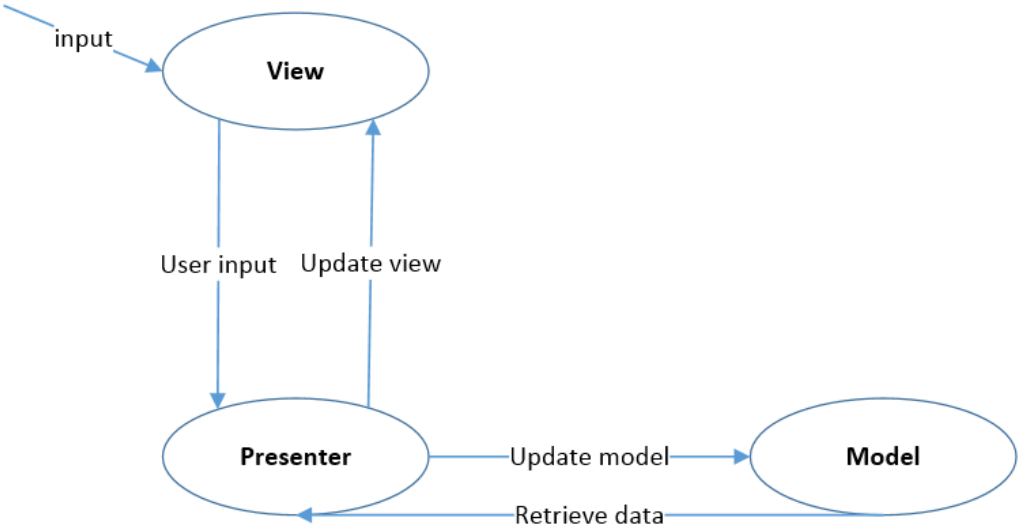


Figure 7. MVP components interaction

Different from MVC, MVP flow starts from view component. View captures user input events and forward them to presenter. For simple event, presenter makes a decision and updates the view. If it is a complex action, presenter will send message to model to retrieve relevant data and then update the view. Sometimes, user's input has an impact on data in model. In this case, *View* will not affect model directly but ask presenter to update the model.

3.2.3 MVP Implementation in Android

MVP architecture has two variants based on the duty of the presenter: *Supervising Controller* and *Passive View* [17]. *Supervising controller* version is the original version of MVP. In this version, the view will controls simple part of the logic while the presenter more complicated logic. *Passive view* prefers to consider view as dummy and pass all logic to presenter.

Android employs passive view. But there is no specific and formal regulations for MVP implementation. Fortunately, Google published MVP open source sample in GitHub [18]. They call this sample Android Architecture Blueprints [beta]. It is an

attempt to formalize MVP implementation regulations. Therefore, the thesis takes this sample as the standard.

UML class diagram in Figure 8 illustrates MVP implementation. *Model* and *View* is the same as those in MVC. Presenter is plain java object.

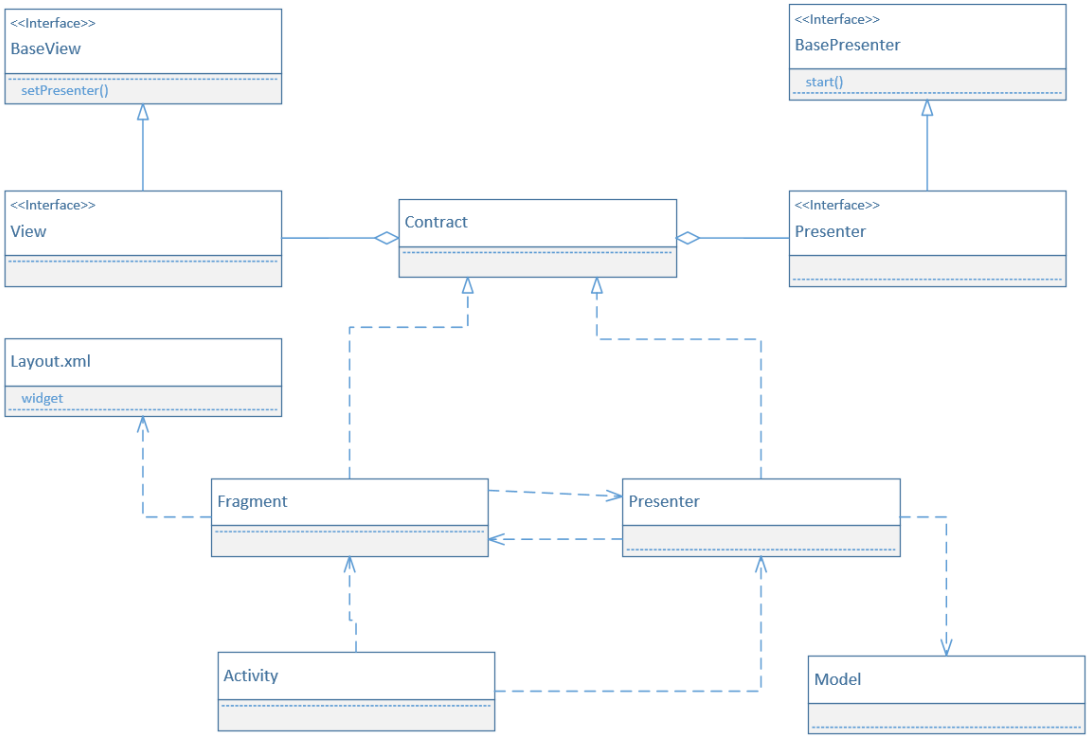


Figure 8. MVP class diagram

Base Interface. BaseView and *BasePresenter* are the parent of all views and presenters. These two basic interfaces ensure that the Presenter and View components are bind together and required data is loaded. BaseView interface is the base class of View component which has one most important method setPresenter() to set the presenter of this view. *BasePresenter* interface is basic for Presenter component. The most important method is inside onStart() to prepare data to be shown in the view. This onStart() method in BasePresenter is usually called in onResume() method in Fragment.

Contract Class is to manage the interface between a specific view and its presenter which is composition of View interface and Presenter interface. View can only call Presenter methods showed in Presenter interface and vice versa.

Fragment/Activity and layout xml are the view. Most time, Fragment affords the view responsibilities. Activity is intended to create the instance of Fragment(view) and Presenter, then connect the two component.

The interaction flow of all classes are interpreted as follows.

1. The Activity class is called with view and presenter bind.
    a. Fragment instance is created in Activity OnCreate() method.
    b. The instance of Presenter is created by calling the constructor method of the presenter. In this step, View and Presenter are bind to each other in this constructor. 1) *View* instance is passed as the parameter, so the view is bind to this presenter. 2) The setPresenter method of View is called with this presenter parameter passed, thus the presenter is bind to view.
2. The view is visible to users which indicates that the Fragment is in resume state. The onStart() method is usually called in onResume() method to load data.

3.3 Model-View-ViewModle

Model-View-ViewModel (MVVM) architecture was explained by an architect John Grossman in his blog [19]. This architecture was applied in Microsoft Silverlight and WPF. It is also based on Model-View-Controller architecture.

3.3.1 MVVM Architecture

"Mode/View/ViewModel (MVVM) architecture is an architecture that is tailored for modern UI development platforms where the View is the responsibility of a designer rather than a classic developer." [19]

MVVM architecture has three components, as its name states, Model, View and view-model. *View* component shows UI of the application. In MVVM, it is supposed to be more designer-friendly that could be easily implemented by designer instead of the code developer. *Model* represents the data, the same as the model explained in previous architecture. *View-Model*, which stands for a model of view, is intended to manage the state of the view. It will pass the data and operations to view and also manage the view's logic and behavior [20]. A good View-Model Component should

only contains the state-specific data instead of view-specific data both in naming and type [20]. For example, if we want to save data when the save button is enabled, instead naming the variable *isSaveButtonEnabled*, the state-specific data *canSave* is preferred.

Interactions among these components are explained in Figure 9 [20]. The connection between the ViewModel and View is complex than in MVP architecture. There are two types of connections: traditional connection and databinding connection. Traditional connections is similar in MVC and MVP that a View-Model components will change the View in java code.

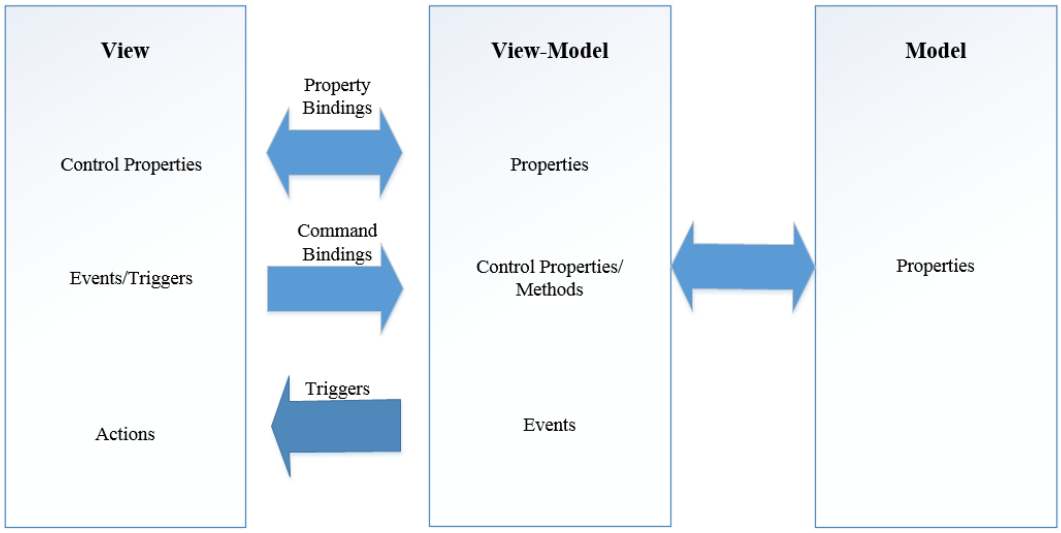


Figure 9. MVVM interaction

Databinding is a new mechanism introduced in MVVM. It allows the view directly bound to the properties and operations of the View-Model. With the databinding, View-Model component does not have to notify the view changes via code and the view knows data is loaded and shows the data by view itself. For example. In MVP and MVC architecture, after the data loaded, the presenter and controller will set the view in code. With databinding mechanism, the view is bound to this data and when data is loaded, view will change automatically.

The databinding between the View and View-Model can be directional and bi-directional. When the data bound in view is changed, the data in View-Model component also know the change. This binding, which is referred as property binding

is bi-directional. There is also databinding called operation binding which is directional. An operation created in View-Model was bound to a widget in View. This binding is like the JavaScript code in HTML form view. When user click the save button in a form, it will call the method pre-defined in JavaScript code. With this way, in View-Model component, we do not need to write code to capture the click event.

3.3.2 MVVM implementation in Android

Databinding in Android

To demonstrate how databinding works in Android, this section shows the instruction examples from official Android developer website [21]. In Android, to enable the databinding, third party library – databinding library is imported. The databinding library page explains the detail usage [21]. Here in this section, only the data binding and event binding are explained which are also called the properties binding and operation binding respectively.

To employ the databinding library in Android, we need to make several changes based on the current Android implementation. Firstly, import the library. Secondly, set the binding object in Activity class when inflating the layout file in OnCreate() method. Thirdly, in related xml file, a new data section with bind variables are declared.

For example, the app will show the user name. In Activity, inflate the layout file with databinding methods instead of the default setContent() method. After this, we get the reference to this binding. Then pass the *User* object whose first name is Test and last name is User to xml file via the reference.

```
@Override
protected void onCreate(Bundle savedInstanceState) {
   super.onCreate(savedInstanceState);
   MainActivityBinding binding = DataBindingUtil.setContentView(this, R.layout.main_activity);
   User user = new User("Test", "User");
   binding.setUser(user);
}
```

```
public class MyHandlers {
    public void onClickFriend(View view) { ... }
    public void onClickEnemy(View view) { ... }
}
```

Next, on top of the basic xml file, add a new section to declare related Object class and method class which will be used in this XML file. In widget properties, user the @{} to refer the related events and attributes.

```
<?xml version="1.0" encoding="utf-8"?>
<layout xmlns:android="http://schemas.android.com/apk/res/android">
   <data>
       <variable name="handlers" type="com.example.Handlers"/>
       <variable name="user" type="com.example.User"/>
   </data>
   <LinearLayout
       android:orientation="vertical"
       android:layout_width="match_parent"
       android:layout_height="match_parent">
       <TextView android:layout_width="wrap_content"
           android:layout_height="wrap_content"
           android:text="@{user.firstName}"
           android:onClick="@{user.isFriend ? handlers.onClickFriend : handlers.onClickEnemy}"/>
       <TextView android:layout_width="wrap_content"
           android:layout_height="wrap_content"
           android:text="@{user.lastName}"
           android:onClick="@{user.isFriend ? handlers.onClickFriend : handlers.onClickEnemy}"/>
   </LinearLayout>
</layout>
```

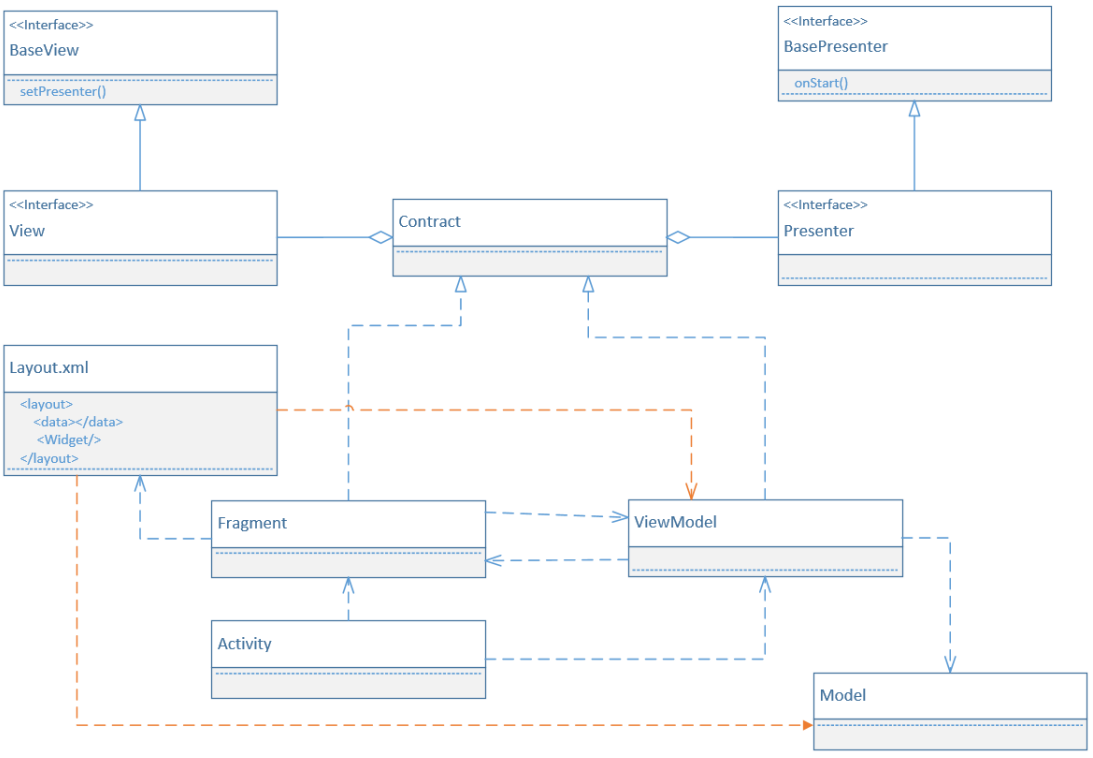


Figure 10. MVVM implementation in Android

Figure 10 shows UML diagram of MVVM architecture in Android. From the published blue print, the MVVM architecture is similar to the MVP architecture. It is the improvement with databinding library in MVP architecture. The major difference is that the dependency from view xml file to ViewModel/and Model class (in orange line). View xml file needs to know the structure in Model. In the above example, it needs to understand in User object has firstName, lastName and isFriend attributes. Similarly, in order to call methods, XML file needs to know class name MyHandler() and its methods.

4 Testability

Testability, in ISO/IEC 25010, is defined as the "degree of effectiveness and efficiency with which test criteria can be established for a system, product or component and tests can be performed to determine whether those criteria have been met". It indicates how ease the software to test [22].

Test is an essential activity in mobile development. As mentioned before, the mobile app upgrade frequency is very high, around one to four release per month. Fast development is typically at the cost of quality [23]. A paper in 2013 shows that one of the critical challenge in mobile development, after the interviews and surveys, is no sufficient tools support for test [24]. But now, there are multiple test libraries in Android platform that, from another perspective, proves the importance of test.

4.1 Test in Android

Android develop guide categorizes four types of tests: local unit tests, instrumented unit tests, components within app tests and cross-app tests. For each test type, there are special supporting library.

Local unit test and instrumented tests are called the Unit Tests which test part of the application. Local unit tests run on local JVM machine. It is the same as the test in Java project. Instrumented unit tests are related to Android-specific environment, e.g. emulator or physical devices. This feature implies instrumented tests costs more time than the local unit tests. Since to run instrumented tests, we have to start emulator/devices and install the Android app every time.

The remaining two tests are called integrated tests. The *components within app tests* focus on the interactions between user and UI. When user click the button or input something, this kind of tests will verify if the output is the same as expected. *Cross-app tests* concentrate on tests with different applications. For example, some apps allow users to add schedule in system calendar and some apps can control the brightness of the screen.

4.2 Testability Criteria

To date, several studies have explored the factors influencing the testability.

Freedman proposes the concept of *domain testability* which is a combination of observability and controllability [25]. Observability requires that for the same input arguments, the output is always the same. Controllability requires that the inferred input set from the output sets is the same as the original input set. A software component is testable if it is controllable and observable. A testable software component has 4 attributes: small and easily constructed test sets, non-redundant test sets, easily interpreted test outputs and easily traced faults in component.

Baudry and Le Traon claim that the testability of the software is influenced by three parameters: global test effort, controllability and observability [22]. Global test efforts refer to the testing effort to achieve the test goal, including test sets size, ease to generate test data and ease to validate the test results [22].

Binder analyzes various testability factors in Object-Oriented system and presents all factors with the fishbone diagram [26]. Six major factors are recognized: process capability, built-in test, the test suit, test support environment representation and implementation (need to explain). Bruntink and Arie evaluate the class testability by estimating amount of test cases and ease to create single test case [27].

Up to now, several testability factors have been recognized: *observability, controllability, size of the test cases, effort to construct the test case, ease to interpret test results* and *ease to locate the errors*.

Observability and *controllability* are not suitable in this thesis. Architecture analysis in section 3 shows that the major difference for three architecture comes from the de-couple of the components. During the decoupling procedure, methods and functions are moved to different components. But function details remain the same. If the method and function are observable and controllable in one architecture. It should be the same in other architecture, because the function/method implementation is not changed.

Ease to interpret the test case in different architectures are the same. With the help the third-party library, the results are given directly after running the test case. In

Android Studio, passed test cases are in green color while failed cases are in red with reasons. However, different test types consumes different time. Apparently, tests running on JVM costs less time than those run on emulators. Therefore, we change this factor to new factor *consumed time to run test cases*.

Ease to locate the errors. When errors happen, Android Studio will print the trace of errors. This process is automatic with IDE. If developers want to know more details of errors. Another method is to set breakpoints to trace the error step by step. Thus, we make this factor more specific in Android - *ease to debug with break points*.

In short, three criteria are employed to evaluate the testability**:** (1) *Size of test cases*. The architecture with less test cases is better. (2) *Consumed time to run test cases* - The application with less instrumented tests consumes less time. And (3) *ease to debug with break points* - ease for developers to debug the application with breakpoints.

4.3 Evaluation

4.3.1 MVC

In comparison, the testability factors in MVC are considered as standards. Then we will try to compare the factors in MVP and MVVM with these standards to check if they are better or worse than MVC.

4.3.2 MVP

Size of test cases is the same as in MVC. When two application have the same functionality, their size of test cases are the same.

Consumed time to run the test case: Some methods was separate from the Fragment/Activity and put in the Presenter class, the pure Java class. In other words, some UI test cases was converted to the unit test. There are less instrumented tests but more local unit test. Therefore, the total testing time is decreased.

Ease to debug with break points: Overall, it is the same as the MVC. Nevertheless, in MVP, some methods are moved from the Fragment/Activity to Presenter. The

breakpoints are in two class. Thus, developers have to navigate between these two classes continuously which makes this debug less convenient.

4.3.3 MVVM

Size of the test case: Local Unit Testing: As we can see from the UML class diagram, methods in presenter are less than in MVP. Therefore, there are less test cases in unit testing, Instrumented Testing: The same as the MVP

Consumed time to run test case: Based on the fact that there are less local unit test cases, the total consumed time is less than MVP.

Ease to debug with breakpoints: The data was in three files, except for the Fragment/View and the Presenter, layout files contain part of the model data. Currently. the layout xml file does not support the breakpoint. It is harder to debug the errors in layout files.

4.3.4 Evaluation results

Table 1 summarizes the testability for three architectures. In the table, we assume that MVC testability is the standard. The results in other architecture is compared to the MVC. Overall, for testability, MVC< MVVM <MVP.

Table 1. Testability for MVC, MVP and MVVM

	MVC	**MVP**	**MVVM**
Amount of test cases	Std.	Same	Less (less functions in presenter)
Consumed time	Std.	Less (less instrumented tests)	Less – (less test cases)
Ease to debug with breakpoints	Std.	Harder (navigation between classes)	Harder + (no support for xml file)

5 Modifiability

Modifiability of the software system is the cost to make the changes [28] [29]. Modification plays an important role in software life cycle. Studies show that modification costs 50% - 70% in software lifecycle after the first development [29] [30]. It is also the most important part in mobile development. To evaluate the modifiability, two questions must be answered: what is the common modification scenario and what are factors to measure the modifiability.

5.1 Modification

5.1.1 Modification scenario

Changes for the software system can be environment, requirement and functional specification [29]. In order to find the most common modification activities of Android app, I investigated the update (What's New) log of 10 apps on Google Play. The details of apps and corresponding logs are showed in Appendix A [31].

Based on the analysis, four common upgrade activities are identified: *new features*, *feature improvement*, *bug fix* and *quality improvement*. *New features* are features that are not in the app previously but added in the new version. *Feature improvement* indicates that the app has this feature before but in new version, they make some refinements. The boundary between the function improvement and new features are not strict. Sometimes, the improvement of the functions requires add new features. For example, Angary Birds 2 adds a new type of pig – gravity pig. We can say this is a feature improvement, because this game app has pig character before. While it can also be considered as the new feature based on the fact that the app does not have gravity pig. *Bug fix* refers to the behavior to fix the errors or crashes in the app. *Quality improvement* is to improve the software quality e.g. speed and security.

The selected scenario for modification is *adding new feature* and *bug fix.* Improving the current features, from another perspective, is bug fix. Common quality improvement activities include algorithm change and library update that are usually after implementation. These are suitable in architecture evaluation phase.

5.1.2 Modification criteria

Zhao et al. claim that the amount of functionalities within one component will have a negative impact on the modifiability [30]. It is more complicated to make changes if there are more functionalities in one component. ISO functional size measurement is employed to calculate the number of functionalities which contains files, interfaces and interactions. Considering the fact that architectures are designed from the same requirements, they tend to have the same functionality. Therefore, we will not evaluate this factor.

Bachmann et al. believe that coupling and cohesion will influence the modification of the software [28]. Coupling refers to the strength of associations cross different modules. *Cohesion* indicates strength of relations within a specific module. Reducing coupling and enhancing cohesion will improve the modifiability. Components with high cohesion and low coupling have weak ripple effects when making changes [30]. Seven levels of cohesion [32] and six levels of coupling [33] [34] are recognized. Paper [30] summarizes the levels showed in Table 2 and Table 3.

Table 2. Cohesion level [30]

Category	Cohesion level	Characteristics
High level	Functional	A component performs a single task.
Moderate levels	Sequential	A data item is sequentially transferred across tasks within a component.
	Communicational	All tasks within a component share the same input or output data items.
	Procedural	Tasks within a component are connected by control connectors.
Low levels	Temporal	Tasks within a component are correlated by temporal relations.
	Logical	Tasks that are logically grouped to perform the same type of functionalities.

Table 3. Coupling level [30]

Category	Coupling level	Characteristics
High levels	Content	A component uses data or control maintained by another component.
	Common	Components share global data items.
	External	Components are tied to external entities such as devices or external data.
Moderate levels	Control	Controls flow across components.
Low levels	Data structured	Structured data are transferred among components.
	Data	Primitive data or arrays of primitive data are passed among components.
	Message	Components communicate through standardized interfaces.

In this thesis, to evaluate the architecture modifiability, we will evaluate cohesion and coupling level of each architecture according to the given level description on Table 2 and Table 3.

5.2 Evaluation

5.2.1 Cohesion level

To evaluate the cohesion level, we need to analyze each component. For *Model* components, there is no difference in three architectures. *View* component in three architectures are also similar: Fragment/Activity class. These two classes follow basic lifecycle from create to destroy which is the *procedural* cohesion level. Thus the model and view components are not included in the discussion. Only the third components, controller in MVC, presenter in MVP and view-model in MVVM are discussed.

In MVC architecture, the *controller* are the combination of the Activity and Fragment. While in in MVP architecture, a separate component presenter takes the controller duties. That is, the controller in MVC and the presenter in MVP do the same tasks. Therefore, these two components have the similar cohesion level. In MVVM architecture, the cohesion level for *ViewModel* component is the same as the presenter in MVP.

Overall, these three architectures share the same cohesion level. In short, cohesion level for three architectures are: MVC = MVP = MVVM.

5.2.2 Coupling level

Coupling level describes interactions among different components which are more complex. As discussed in section 5.1.1, two most common modification scenario are adding new feature and fixing bugs. To make it clearly, we employ a concreate adding new feature example to explain coupling level. In this example, a menu item is added to allow users to delete news by clicking it. Related sequence diagram is shown on Figure 11. The second object- C/P/VM represents controller, presenter or ViewModel which cannot be determined now.

1. User clicks the delete command in menu item.
2. Click event is sent to C/P/VM component.
3. Event is caught by OnClickListener() and the deleteNews() function is called.
4. Parameters are sent to model component, in this case – newsId is sent.
5. Model delete the news based on the id on background.
6. After delete, a response is returned to C/P/M.
7. Based on response, if the news is deleted successfully, refresh the view while if the delete failed, show error message. C/P/M passes this command to View.
8. View gets updated based on command from C/P/M.

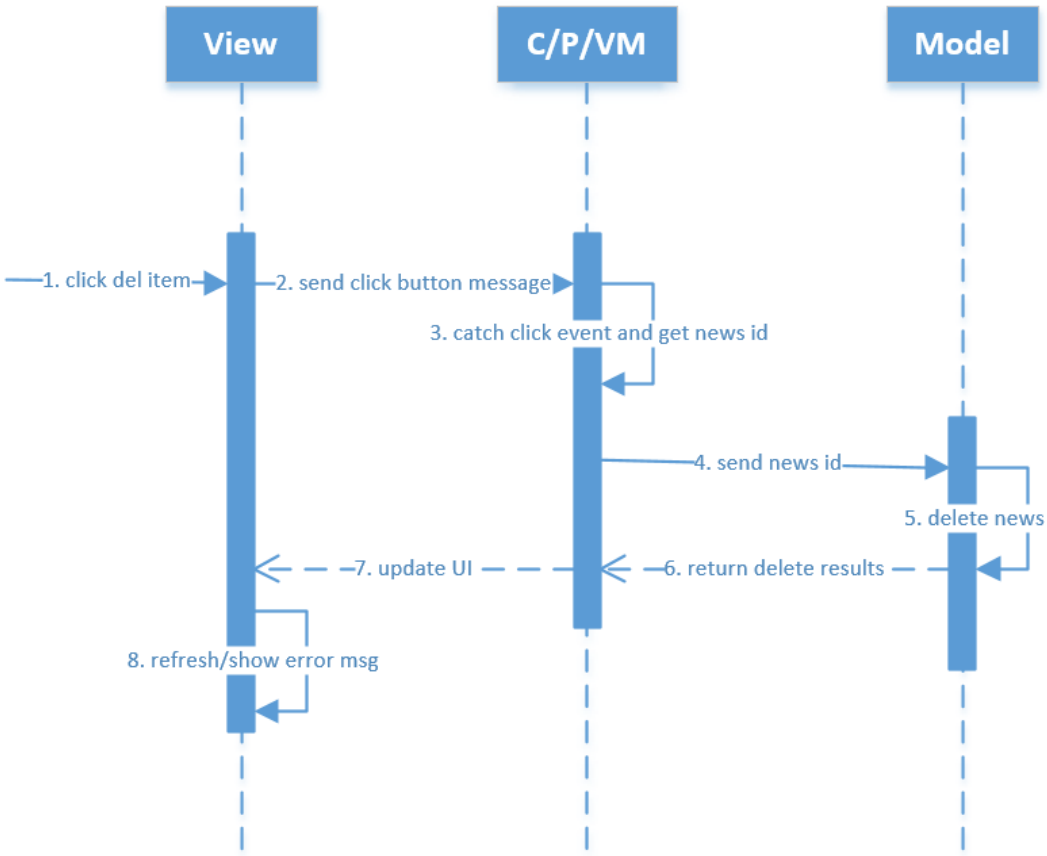


Figure 11. Delete news sequence diagram

MVC. The coupling level for MVC architecture are summarized in Table 4.

Connection from *Model* to *View* and *Controller* are both in Data structured level. Figure 3 shows that the model will update the View and Controller by sending the change message. In the example, the model sends the delete results(success or fail) back to controller. In most complicated condition, this message might be in structured data format.

Connection from View to Model is in *data structured* level. There might be some data changed in view and then updated in model. The edit message which might be simple data or structured data is passed to Model.

The connection from Controller to Model and from Controller to View are both in *control* level. *Controller* component catches event, calls delete command and sends update UI command. It maintains the data flow.

Table 4. MVC coupling level

Source	Destination	Connection	Coupling Level
Model	View	Model sends data to view.	Data structured
Model	Controller	Model sends data to controller	Data structured
View	Model	View sends data to model	Data structured
Controller	Model	Controller maintains the data flow.	Control
Controller	View	Controller maintains the data flow	Control

MVP. As shown in Figure 7, there is no connection between the View and Model. View and Presenter component coupling is in *message* level that means the communication is via interface. The presenter component pass parameters to model to perform tasks. After tasks done, model will return the response which might include data or error message. Thus, connections between model and presenter is in *data structured* level. Components interactions are summarized in Table 5. Overall, the coupling level in MVP architecture is in **low level**.

Table 5. MVP coupling level

Source	Destination	Interactions	Coupling level
View	Presenter	Interface	Message level
Presenter	View	Interface	Message level
Presenter	Model	Parameters: primitive data/ object	Data structured
Model	Presenter	Response: primitive data/ object	Data structured

MVVM. Table 6 shows the coupling level in MVVM architecture which is very similar to the MVP architecture. Tow coupling level are different. Firstly, because of the databinding usage, there is one more connection between view and model component. View fetches data from model and show in widget. Secondly, view xml file will call methods in ViewModel directly. That is, the view layout xml file manage the event. When clicking the button, the application will call different actions defined in ViewModel component. This is the control coupling.

Table 6. MVVM coupling level

Source	**Destination**	**Interactions**	**Coupling level**
View	ViewModel	View pass commands via interface.	Message level/ control
ViewModel	View	Interface	Message level
ViewModel	Model	Parameters: primitive data/ object	Data structured
Model	ViewModel	Response: primitive data/ object	Data structured
Model	View	Object (User, News) is passed to view.	Data structured

From Table 4 – 6, MVC components have more connections than the others. And coupling levels are mostly data structured level and control level which are higher than the other two architectures. MVP architecture has the least connections and its coupling level mostly falls on message level and data structured level. MVVM architecture is similar to MVP architecture, but it has one more connection and a higher coupling level.

Coupling level for three architectures are: MVC > MVVM > MVP. For modifiability, MVC < MVVM < MVP.

6 Performance

Performance is considered as the “ the degree to which a system or component accomplishes its designated functions within given constraints, such as speed, accuracy or memory usage.”

6.1 Criteria

Performance is a significant factor in application success. Typical performance factors are response time, throughput, latency [11, 35]. Slow response time will make the user impatient and decrease the user number. A good architecture might be not always create the good performance, but the poor-designed architecture definitely has a bad effect on performance [36].

Schmerl et al. categorize performance evaluation methods into two types: analytical performance model and simulation performance model [37]. The former method based on mathematical model is hard to create and cannot capture the system complexity well. The simulation-based method tries to run the system in real environment so the results are relatively accurate.

Mattsson [11] summarizes different architecture evaluation methods, of which 6 performance evaluation methods are all simulation-based. Except the Layered Queuing Network model is widely used, other 5 methods are only used by the author. Layered queueing network model is especially created to build the concurrent and distributed software system model [35]. Since in a server/client system, it is a common scenario that many clients require the resources from the server at the same time. However, this method is not feasible in Android native applications.

ATAM method is not really suitable for performance evaluation because the performance evaluation is simulation-based while ATAM is scenario-based. But considering the fact the performance is an important quality, this factor is also included. But the evaluation process is a little different. We do not analyze based on the theory, but use the real application to do the experiment. Then based on the results to evaluate the architecture.

To evaluate the performance of the Android application, some factors are take into consideration. Corral et al. [38] believe that the execution time, battery consumption or memory usage are the key factors for Android app performance. To evaluate the execution time on native app and web app, Corral developed the application to execute different tasks: write/read file, request data and etc. It turns out the native app performance is better than web app. Official Android website suggests the performance factors in Android app contains response time, battery consumption , memory usage, CPU usage and GPU usage. In this thesis, we focus on the memory usage.

6.2 Evaluation

In this section, memory usage of the app with different architecture is tested. We look into the official architecture sample to-do app. With this application, the user can write and edit the to-do tasks. When the user complete the task, they can mark it as complete. The statistics function in app will show the number of completed task. The screenshot for this app is shown in Appendix B. Currently, samples only contain the MVP and MVVM architecture app. I created the MVC application app based on the existing application.

The experiment is performed on Android Nexus 6 emulator in API 21. Android Monitor in Android Studio is used to detect the memory usage.

To evaluate the memory usage, use all the functions in the app as a user: add, delete, update, refresh and navigate between different views. The process lasts for 5 minutes. During this period, memory usage data is recorded every minute (0, 1, 2, 3, 4 and 5). In order to eliminate the possible errors, for each architecture application, 3 experiments are performed. The detail results for each experiment can be found in Appendix C. After data collection, average value is calculated which is shown on Table 7. Based on statistics, a line chart in Figure 12 is created to accent the difference.

Table 7. Allocated memory for application per minutes

	0	1	2	3	4	5
MVC	2,46	6,43	11,84	21,84	29,84	28,38
MVP	2,39	5,20	8,32	16,17	15,35	12,30
MVVM	2,42	7,74	9,89	16,49	17,30	9,35

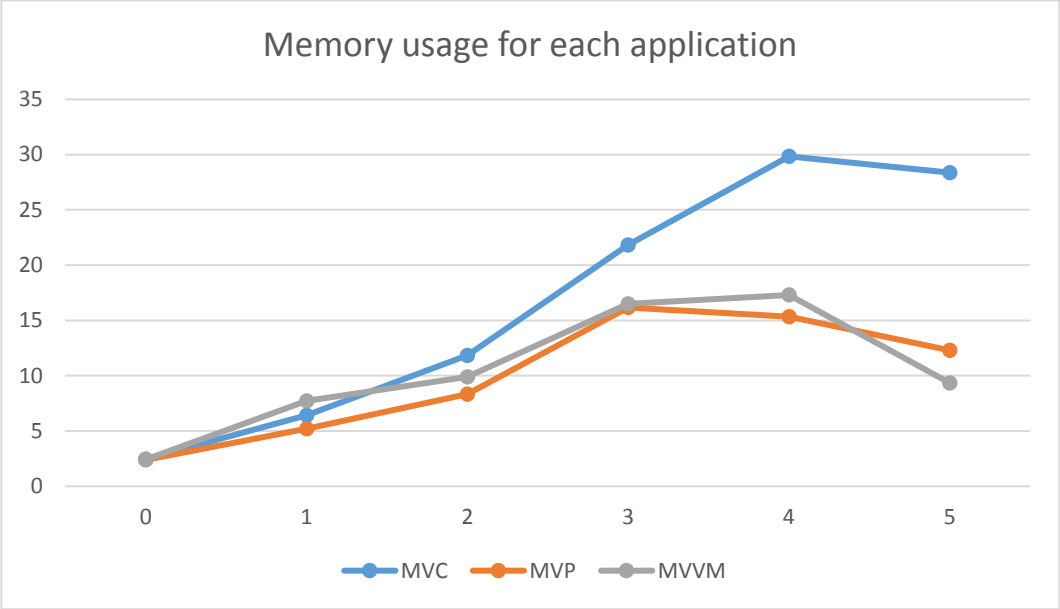


Figure 12. Memory usage for each architecture application

Table 7 and Figure 12 show that when the application was started, the allocated memory is almost the same. However, with more operations/tasks executing, the difference increases. The MVC architecture app use more memory than MVP and MVVM apps which are almost the same. From memory usage perspective, performance quality for three architecture is MVC < MVP = MVVM.

7 Results analysis

This section summarizes evaluation results. Then based on these results, tradeoffs are analyzed.

In MVC architecture, there are three components model, view and controller. View and controller are tightly coupled in Fragment and Activity class.

In MVP architecture, a new layer Presenter is introduced to replace the controller. Presenter component separates some methods to the plain Java class. View component is like dummy. With this new component, (1) view and presenter have clear duties; and (2) connection between model and view are eliminated.

Based on MVP architecture, MVVM imports databinding mechanism. View xml file can get data from model and show in the widget directly. Besides, view xml file can call methods in ViewModel components directly.

Testability. The architecture has good testability if it has less test cases, consumes less time to run tests and is easy to debug with break points.

- Size of test case: MVC = MVP >MVVM
- Consumed time: MVC >MVP >MVVM
- Ease to Debug with breakpoint: MVC = MVP > MVVM

Based on the criteria, results show that MVC testability is the worst and MVVM is the best. Thus, Presenter component and ViewModel component contributes to better testability.

If we dive into the specific factor, test cases in MVVM are less than in the MVP because of the import of databinding mechanism. Consumed time to run test case is less in MVP and MVVM because of test size is smaller. But the ease to debug with breakpoints is lower. Thus, the testability is also sensitive to databinding in MVVM. Databinding has a positive effect on testability.

Modifiability. Higher cohesion level and lower coupling level indicates good modifiability.

- Cohesion level: MVC = MVP = MVVM.
- Coupling level: MVC > MVVM > MVP

Results indicate that modifiability for MVP and MVVM are better than MVC which implies modifiability is sensitive to separate of the *Presenter* component. Presenter component lead to better modifiability.

But for coupling level, MVVM is worse than MVP which is because of the databinding mechanism. Thus, coupling level is sensitive to databinding. Databinding will increase the coupling level.

Performance. Less memory usage indicates better performance.

- Consumed memory: MVC > MVP = MVVM

The result shows that MVC architecture app consumes more memory than MVP and MVVM. Thus, MVP and MVVM has better performance than MVC.

Overall, quality attributes are sensitive to *Presenter* component. The separation of the Presenter layer improves the quality of the app: testability, modifiability and performance.

Testability and modifiability are sensitive to databinding. Databinding will improve the testability but decreases the modifiability. Databinding decreases the amount of the test cases and test time, but meanwhile it increase the difficulty to debug with breakpoint. Besides, it increases the coupling level.

8 Conclusion

The thesis aims to figure out whether the MVP and MVVM architecture in Android native application are better than the MVC architecture. To achieve this goal, the thesis firstly studies various architecture evaluation methods and select the ATAM method.

Then three quality factors: testability, modifiability and performance are considered as the standard to perform the evaluation. After that, for each quality, the criteria to evaluate the quality is established and assessed. At last, the sensitivity analysis and tradeoff analysis was performed.

For testability, Size of test cases, consumed time to run test cases and ease to debug with breakpoints factors are employed. MVC and MVP have the same results while MVVM shows better testability since less test cases are required.

Cohesion and coupling level are used in modifiability assessment. MVP and MVVM have lower coupling level which implies better modifiability. While MVVM has higher coupling level than MVP because the databinding creates new connections among components.

For performance evaluation, the thesis tests memory usage of the sample app on emulator. Simulation shows that MVP and MVVM are better than MVC.

After the analysis and experiments, there are two most important conclusions. Firstly, MVP and MVVM architecture are better than MVC architecture on testability, modifiability and performance qualities. Presenter layer has a positive effect on these qualities. Secondly, MVVM is not necessarily better than MVP. Databinding mechanism in MVVM has a positive effect on testability but a negative effect on modifiability. Thus, MVVM has better testability but worse modifiability than MVP.

Therefore, it is better to migrate from the MVC to MVP/MVVM architecture to gain better software qualities. But when it comes to choice between MVP and MVVM architecture, it depends on the requirements. If developers want to write less code

and gain better testability , MVVM is better than MVP. While if the team cares more about modifiability than testability, MVP is a better option.

Suggested future research might be the best practice to implement MVP architecture in Android. In this thesis, MVP architecture was abstracted from the Google example. Some argue that the View and Presenter specific interfaces can be removed. They believe that the main point for these is to make the test easier by decouple Android environment and Java environment. But actually, with some test libraries, there is no need to make such separation.

The architecture presented in this thesis is completely implemented without any third party libraries. Nevertheless, some have come up with MVP framework with the help of the libraries, e.g. the Mortar and Dagger from Square. In addition, the current MVP sample showed by Google is the blue print. In the future, officially there might be some third party libraries provided by Android team themselves. In this case, the similarities and differences, advantages and disadvantages are also good topic to discuss.

Another future research question is based on the results in the thesis. In performance evaluation, result shows that the MVC consumes more memory than MVP and MVVM. But the reason is not investigated. More experiments should be carried to study in depth.

References

[1] V. Woods and R. v. d. Meulen, "Gartner Says Worldwide Smartphone Sales Grew 9.7 Percent in Fourth Quarter of 2015," Gartner, 18 February 2016. [Online]. Available: http://www.gartner.com/newsroom/id/3215217. [Accessed 28 May 2016].

[2] AppBrain, "Number of Android applications," AppBrain, 28 5 2016. [Online]. Available: http://www.appbrain.com/stats/number-of-android-apps. [Accessed 28 May 2016].

[3] CrispyCodes, "How Long Does It Take To Build An IOS or Android App?," Cripsy codes, 18 Feb 2014. [Online]. Available: http://visual.ly/how-long-does-it-take-build-ios-or-android-app. [Accessed 29 5 2016].

[4] K. YARMOSH, "How Often Should You Update Your App?," savvyapps, 12 January 2016. [Online]. Available: http://savvyapps.com/blog/how-often-should-you-update-your-app. [Accessed 29 May 2016].

[5] IEEE, "IEEE Standard Glossary of Software Engineering Terminology," Institute of Electrical and Electronics Engineers, New York, 1990.

[6] N. Medvidovic and R. N. Taylor, "Software architecture: foundations, theory, and practice," in *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering* , Cape Town, South Africa, ACM, 2010, pp. 471-472.

[7] L. Chen, M. A. Babar and B. Nuseibeh, "Characterizing Architecturally Significant Requirements," *IEEE Software,* vol. 30, no. 2, pp. 38 - 45, 2013.

[8] L. Bass, Software Architecture in Practice, Pearson Education India,, 2007.

[9] P. Clements, R. Kazman and M. Klein, "Evaluating a Software Architecture," in *Evaluating Software Architectures: Methods and Case Studies*, Addison-Wesley Professional, 2001, pp. 19-42.

[10] A. Patidar and U. Suman, "A survey on software architecture evaluation methods," in *Computing for Sustainable Global Development (INDIACom), 2015 2nd International Conference on*, Indore, 2015.

[11] M. Mattsson, H. Grahn and F. Mårtensson, "Software architecture evaluation methods for performance, maintainability, testability, and portability," in *Second International Conference on the Quality of Software Architectures*, 2006.

[12] R. Kazman, M. Klein, M. Barbacci, T. Longstaff, H. Lipson and J. Carriere., "The architecture tradeoff analysis method," in *Engineering of Complex Computer Systems, 1998. ICECCS'98. Proceedings. Fourth IEEE International Conference on*, 1998.

[13] R. Kazman, M. Klein and P. Clements, "ATAM: Method for architecture evaluation," Carnegie-Mellon Univ Pittsburgh PA Software Engineering Inst, 2000.

[14] 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.

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

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

[17] M. Fowler, "GUI Architecture," 18 July 2006. [Online]. Available: http://martinfowler.com/eaaDev/uiArchs.html. [Accessed 26 05 2016].

[18] Google, "Android Architecture Blueprints [beta]," Google, 3 2016. [Online]. Available: https://github.com/googlesamples/android-architecture. [Accessed 26 5 2016].

[19] J. Grossman, "Introduction to Model/View/ViewModel pattern for building WPF apps," Microsoft, 8 October 2005. [Online]. Available: https://blogs.msdn.microsoft.com/johngossman/2005/10/08/introduction-to-modelviewviewmodel-pattern-for-building-wpf-apps/. [Accessed 28 May 2016].

[20] C. Anderson, "The Model-View-ViewModel (MVVM) Design Pattern," in *Pro Business Applications with Silverlight 5*, Berkeley, Apress, 2012, pp. 461--499.

[21] Google, "Data Binding Library," Google, [Online]. Available: https://developer.android.com/topic/libraries/data-binding/index.html. [Accessed 7 8 2016].

[22] B. Baudry and Y. Le Traon, "Measuring design testability of a UML class diagram," *Information and software technology*, vol. 47, pp. 859--879, 2005.

[23] L. Zhifang, L. Bin and G. Xiaopeng, "Test automation on mobile device.," in *Procedings of the 5th Workshop on Automation of Software Test*, New York, 2010.

[24] M. E. Joorabchi, V. B. C. Univ. of British Columbia, A. Mesbah and P. Kruchten, "Real Challenges in Mobile App Development," in *Empirical Software Engineering and Measurement, 2013 ACM/IEEE International Symposium on*, 2013.

[25] R. S. Freedman, "Testability of software components," *IEEE Transactions on Software Engineering*, vol. 17, no. 6, pp. 553 - 564, 2002.

[26] R. V. Binder, "Design for testability in object-oriented systems," *Commun. ACM*, vol. 37, no. 9, pp. 87-101, 1994.

[27] M. Bruntink and A. van Deursen, "An empirical study into class testability," *Journal of systems and software*, vol. 79, no. 9, pp. 1219 -- 1232, 2006.

[28] F. Bachmann, L. Bass and R. Nord, "Modifiability tactics," CARNEGIE-MELLON UNIV PITTSBURGH PA SOFTWARE ENGINEERING INST, 2007.

[29] P. Bengtsson, N. Lassing, J. Bosch and H. v. Vliet, "Analyzing software architectures for modifiability," 2000.

[30] X. Zhao, F. Khomh and Y. Zou, "Improving the Modifiability of the Architecture of Business Applications," in *2011 11th International Conference on Quality Software*, Madrid, 2011.

[31] Google, "Google Play," Google, [Online]. Available: https://play.google.com/store/apps. [Accessed 22 05 2016].

[32] R. S. Pressman, Software Engineering: A Practitioner's Approach, Palgrave Macmillan, 2005.

[33] W. P. Stevens, G. J. Myers and L. L. Constantine, "Structured design," *IBM Systems Journal*, vol. 13, no. 2, pp. 115-139, 1974.

[34] W. Li and S. Henry, "Object-oriented metrics that predict maintainability," *Journal of systems and software*, vol. 23, no. 2, pp. 111-122, 1993.

[35] D. Petriu, C. Shousha and A. Jalnapurkar, "Architecture-based performance analysis applied to a telecommunication system," *IEEE Transactions on Software Engineering* , vol. 26, no. 11, pp. 1049 - 1065, 2000.

[36] L. G. Williams and C. U. Smith, "Performance evaluation of software architectures," in *Proceedings of the 1st international workshop on Software and performance*, New York, 1998.

[37] B. Schmerl, S. Butler and D. Garlan, "Architecture-based Simulation for Security and Performance," 2006.

[38] L. Corral, A. Sillitti and G. Succi, "Mobile multiplatform development: An experiment for performance analysis," *Procedia Computer Science*, vol. 10, pp. 736--743, 2012.

Appendices

Appendix A: "What's New" log

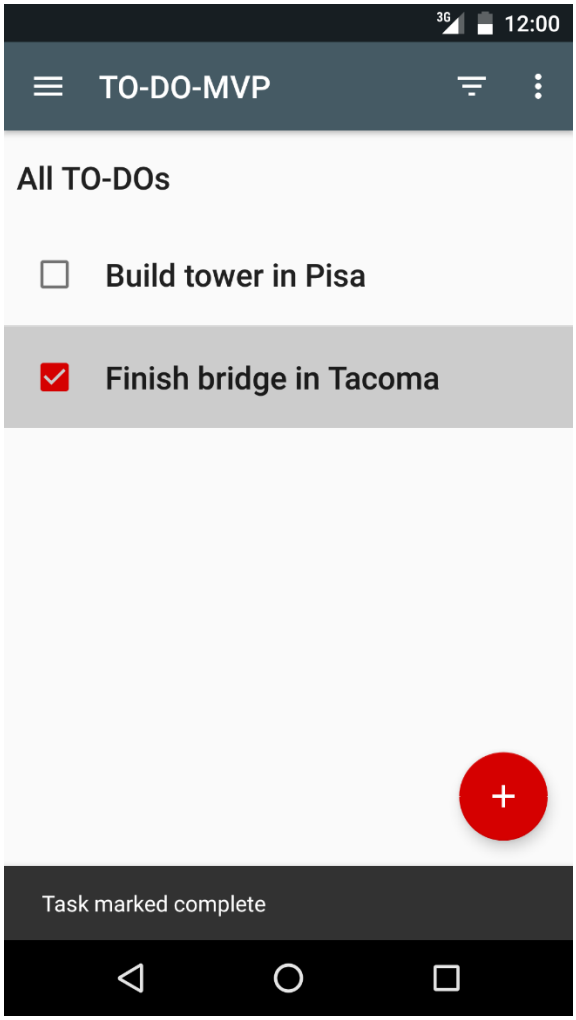
The 'what's new' log of Android apps on Google Play.

Facebook	Improvements for reliability and speed	Quality Improvement
WhatsApp Messenger	Reply to new messages right from notifications	New feature
	Tapping the quick camera button in a chat lets you pick a photo or video from your camera roll	New feature
	Quickly archive, delete, or mute multiple chats at once. Just tap and hold a chat in the chats tab and tap on other chats to select them.	Function Improvement
	Format text in your messages by surrounding your text with special characters: *bold*, _italics_, or ~strikethrough~	Function improvement
Slither.io	New skins!	New feature
	Faster rendering!	Quality improvement
	Fixed a lag bug!	Bug fix
	New control method available! It works like a mixture between a joystick and a mouse. You can try it out if you tap the settings icon in the top right corner!	New feature

Messenger	Now you can see your call history and missed calls—all in one place.	New feature
Snapchat	Bug fixes and improvements	Bug fix
Spotify Music	Meet our new typeface. It's called Circular and we hope you like it.	New feature
Angary birds 2	NEW Chapters! Keep your eyes peeled for The Hamalayas and Madagooscar!	New feature/ function improvement
	NEW PIG – Gravity Pig! Pop this guy and unsettle gravity itself.	New feature/ function improvement
	WIN STREAKS! Get consecutive wins in the Arena for awesome rewards.	New feature
	NEW CARD TIERS! Take your maxed cards to the next level.	New feature/function improvement
Minecraft: Pocket Edition	24 new Biome Settlers Skins! Play a Nether Extinguisher, a Mooshroom Forager, and much more!	New feature/ function improvement
	- Various bug fixes.	Bug fix
Google photos	• Comment on photos in shared albums	New feature/ function improvement
	• Add your photos to a received shared album in one tap with smart suggestions	Function improvement
	• Device folder performance improvements	Quality improvement
Evernote	Many enhancements to the editing experience	Function improvement

	Support for strikethrough, subscript, and superscript text styles	New feature
	Camera now automatically detects and captures business cards, documents, whiteboards, Post-it ® Notes and receipts	Function improvement
	- Ability to mark up images and PDFs	New feature
	- Select multiple notes at once	Function improvement
	- ‘Trash’ can now be emptied	Function improvement
	- Fixed many note display and note editing bugs	Bug fix
	- Various bug fixes and stability improvements	Bug fix

Appendix B: Screenshot for to-do app



Appendix C: Memory usage experiments results

The memory usage experiments results for different architecture app.

		0	1	2	3	4	5
MVVM	1	2.45	5.8	5.8	5.77	14.64	9.9
	2	2.28	7.43	9.59	23.7	22.93	8.56
	3	2.52	10	14.28	20	14.32	9.6
MVP	1	2.42	4.79	8.72	16.5	15.45	16.06
	2	2.39	4.5	8.49	17.76	14.57	10
	3	2.35	6.3	7.74	14.26	16.03	10.83
MVC	1	2.43	4.53	10.75	18.22	26.40	28.07
	2	2.47	7.39	13.96	25.36	31.23	20.41
	3	2.48	7.36	10.82	21.95	31.89	36.66