SOFTWARE DEVELOPMENT FOR MOBILE DEVICES

COS80019

ASSIGNMENT 11 – Research Report

Android Architecture Pattern: MVC, MVP and MVVM

Student name: Duy Thuc Pham

Student number: 101767225

ABSTRACT

In Android development, MVC has been widely used regardless of having various limitations. Due to the high coupling of View and Controller, it is difficult to test, extend as well as maintain the application. Meanwhile, the raise of MVP and MVVM recently in Android development has led to the question which one is the best one to develop Android application. Therefore, by reviewing related works and also implementing MVC, MVP and MVVM patterns, this study aims to evaluate the testability, extensibility and maintenance between these patterns. Thereby, the best suited pattern of Android development will be proposed based on the result of the evaluation so that Android developers can gain insight about the Android architecture patterns as well as can implement it precisely. Furthermore, the study proposes a foundation of Android architecture patterns which can help future researchers to apply it to other clean architectures to improve the quality of software development.

Table of Contents

1 INTRODUCTION

Mobile applications have become an essential part of human being nowadays. According to Statista, the number of applications in the Google Play Store was 2.6 million in March 2018 [1]. However, the rapid change of new technology, as well as user requirements, have led to the need for the improvement of architecture pattern, which may help developers quickly adapt to the alternation and reduce the development time. Many architecture patterns have been applied to the mobile application throughout the year, among them, MVC is considered to be top the most famous in terms of "separate of concerns" and easy for implementation. Also, it has been used as the default architecture pattern for both iOS and Android. In Android development, though it is quite popular, it is concerned for the extensibility, reusability and testability [2]. Furthermore, the various Android developers have applied MVP or MVVM instead of MVC as they consider MVP and MVVM are more extensible and testable [2]. Therefore, this paper aims to evaluate MVC, MVP and MVVM to justify the most appropriate architecture to apply in Android development with respect to maintainability, reusability and testability.

In this research, the brief information of MVC, MVP and MVVM will be introduced. Thereafter, it will illustrate the design of these three architectures in Android development. Then, these architectures will be implemented with the given scenario by following the design section. Finally, the evaluation for these architectures will be conducted to understand which architecture is sufficient with regards to maintainability, reusability and testability.

2 METHODOLOGY

This study illustrates how to apply MVC, MVP and MVVM patterns to Android development so that researchers and developers are able to implement it for their application as well as assist further research. Also, by performing the survey on related works, it will evaluate these patterns to identify the best architecture pattern for Android development with respect to testability, extensibility and maintenance of the program. In order to understand how MVC, MVP and MVVM work in Android, a short introduction of these patterns will be presented in the next part.

2.1 Background

2.1.1 MVC

MVC is a well-known architecture in software development. By following "separate of concerns" principle, it divides the program into three components: model, view and controller. Additionally, it leverages the benefit of "separate of concerns" by decoupling significant features and hence the code is loose coupling and high cohesion.

Model

Model represents the data layer which is used to hold the application data and manage the data logic [3].

View

View is the UI layer which provides the user interface to the user based on the data model.

Controller

Controller is the logic layer which considers as the mediator between the model and view. It takes responsibility for updating the data model and notifying to the view to update the user interface.

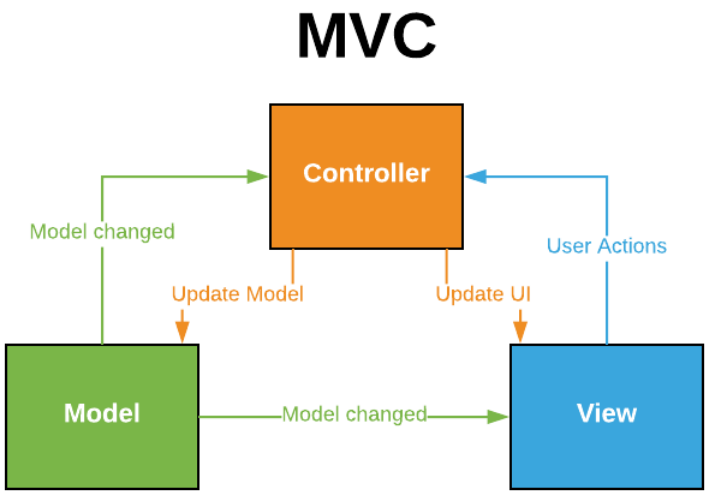

Figure 1: MVC pattern

2.1.2 MVP

Theoretically, MVP pattern is almost the same as MVC pattern by replacing the Controller to the Presenter and also, they both have three separate components.

Model

Just like the model of MVC, it is responsible for managing data.

View

View displays the data to the user interface and notifies the user actions to the Presenter.

Presenter

Presenter manipulates the user actions from the View. Then, it updates the data of the Model and notify the updated information back to the View. It is slightly different from the Controller as the Presenter does not have a reference with the View and vice a versa, but instead they use the interface to communicate with each other [4].

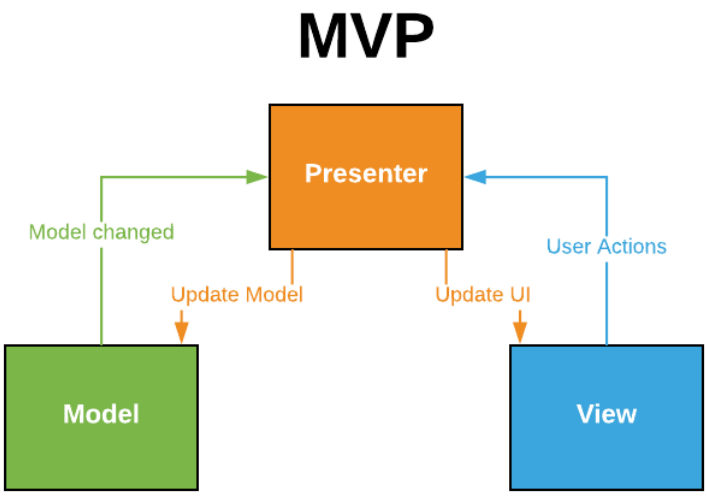

Figure 2: MVP architecture pattern

In addition to the difference between MVC and MVP, in MVP, the Model and View are entirely decoupled. The Presenter is responsible for controlling the user events and the data model.

2.1.3 MVVM

Model-View-ViewModel (MVVM) is an architecture pattern which was announced in 2005 by John Gossman [6]. The aim of this pattern is to optimize the event driven programming by providing the data binding between View and ViewModel [5].

Model

The Model is similar to these patterns above, which controls the business logic of the data layer.

View

The View retrieves the user actions and sends notification to the ViewModel.

ViewModel

The ViewModel acts as an intermediary in this pattern. It is on the one hand, provides methods, properties to the View with the purpose of maintaining the state of it. On the other hand, it manages the Model based on the user actions receiving from the View and invoking the view to update the UI.

Figure 3: MVVM architecture pattern

Unlike MVP, while the Presenter must inform to the View what to show, the MVVM supports the data binding mechanism in which the View can bind to. Therefore, it eliminates various interfaces in MVP pattern. Moreover, as in the Figure 3, in MVVM, many View can bind to one ViewModel in a many-to-one relationship [5], which decouples both components because the ViewModel does not need information from the View.

2.2 Design and Implementation Architecture in Android

In this part, the appliance of MVC, MVP and MVVM in Android are illustrated based on an example application. In this research, the example of application is to convert inch, feet and mile into centimeter which has been already implemented in task 3.2P. The research starts with the design and implementation for MVC, MVP and MVVM respectively.

2.2.1 MVC

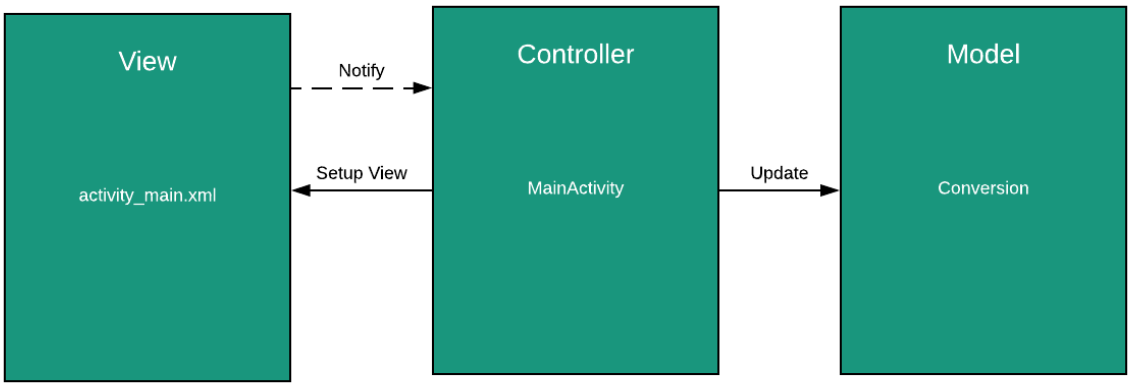

Figure 4: Apply MVC in Android

In Android, MVC is considered as the default pattern to develop Android application. The View is the layout xml, the Controller is Activity or Fragment and Model is the Java class which presents the data layer of the application. The Controller layer seems like the most crucial part of the MVC pattern due to the plenty of responsibilities. Firstly, it retains references with view components (e.g. EditText, TextView, ImageView, etc.). Secondly, it manages the

model instance so that it can update the data for the model right after receiving the user actions. Thirdly, it handles the user actions from the View layer by implementing the listener event such View.onClickListener, View.onTouchListener, etc. Finally, it manipulates the business logic to display the appropriate UI to the user.

```
public class MainActivity extends AppCompatActivity implements
View.OnClickListener {

    // Establish references with the view layout
    EditText edtFeet;
    EditText edtMiles;
    EditText edtInches;
    CheckBox chkboxDisplayMetres;
    TextView txtViewResult;

    private Conversion conversion;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        initializeUI();
        // Instantiate the model
        conversion = new Conversion();
    }

    private void initializeUI() {
        edtInches = findViewById(R.id.edtInches);
        edtFeet = findViewById(R.id.edtFeet);
        edtMiles = findViewById(R.id.edtMiles);
        txtViewResult = findViewById(R.id.txtViewResult);
        chkboxDisplayMetres = findViewById(R.id.chkboxDisplayMetres);

        Button btnConvertMiles = findViewById(R.id.btnConvert);
        btnConvertMiles.setOnClickListener(this);

    }

    /**
     * This method uses to check whether these edit text have value or not
     * @return
     */
    private boolean isValid() {
        return edtInches.getText().toString().length() > 0 &&
edtMiles.getText().toString().length() > 0 && edtFeet.getText().length() > 0;
    }

    @Override
    public void onClick(View view) {
        String result;
        String metres = getResources().getString(R.string.metres);
        if (isValid()) {
            conversion.setData(Double.valueOf(edtInches.getText().toString())
,Double.valueOf(edtFeet.getText().toString()),
Double.valueOf(edtMiles.getText().toString()), chkboxDisplayMetres.isChecked());
            if (chkboxDisplayMetres.isChecked()) {
                result = String.format("%.2f %s", conversion.toValue(), metres);
            } else {
                result = String.format("%.2f cm", conversion.toValue());
            }
        } else {
```

```
            result = "";
            Toast.makeText(getApplicationContext(), "You have to enter a valid
value", Toast.LENGTH_SHORT).show();
        }
        txtViewResult.setText(result);
    }
}
```

Figure 5: Controller in Android

2.2.2 MVP

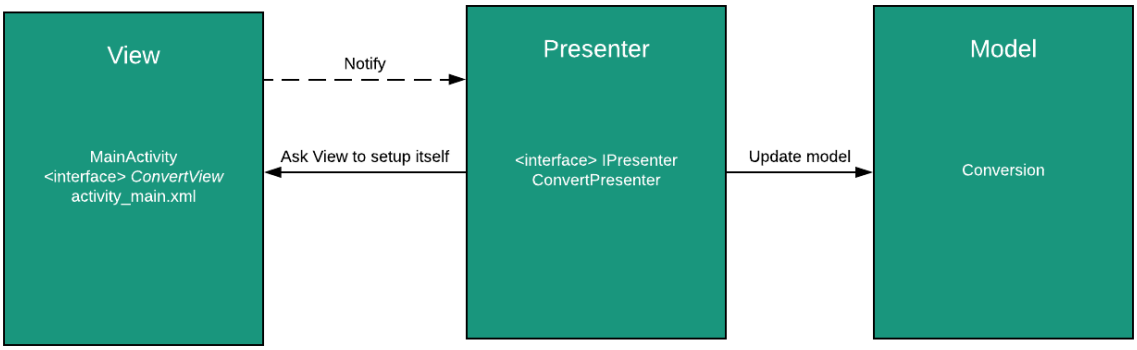

Figure 6: Apply MVP in Android

The design and implementation for MVP pattern are slightly different from MVC pattern. While the model layer is the same, View and Presenter are now communicating with each other via an interface. The View is now including the Activity/Fragment along with the xml layout and also, the View interface which illustrates the information of the View. Presenter layer has a reference with the Model layer and a reference with an interface of the View so that View is notified to update UI when data has been updated. The code snippet below describes how the MVP pattern is implemented in Android.

```
public class ConvertPresenter implements IPresenter {

    private ConvertView view;
    private Conversion model;
   // Instantiate the view and model instances
    public ConvertPresenter(ConvertView view) {
        this.view = view;
        this.model = new Conversion();
    }

    @Override
    public void onCreate() {
        model = new Conversion();
    }

    @Override
    public void onPause() {

    }
```

```
    @Override
    public void onResume() {

    }

    @Override
    public void onDestroy() {

    }
    // When the user select convert button, the presenter receives the actions
from
    // View layer to set the data. Then, after computing successfully,
    // it invokes the callback to notify the view to update the UI based on the
new data
    @Override
    public void onConvertButtonClick(double inch, double feet, double mile,
boolean isMetres) {
        model.setData(inch, feet, mile, isMetres);
        String result;
        if (isMetres) {
            result = String.format("%.2f meters", model.toValue());
        } else {
            result = String.format("%.2f cm", model.toValue());
        }
        view.displayResult(String.valueOf(result));
    }
}
```

Figure 7: Presenter in practical

```
public class MainActivity extends AppCompatActivity implements
View.OnClickListener, ConvertView {

    EditText edtFeet;
    EditText edtMiles;
    EditText edtInches;
    CheckBox chkboxDisplayMetres;
    TextView txtViewResult;
    ConvertPresenter presenter = new ConvertPresenter(this);

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        initializeUI();
        presenter.onCreate();
    }

    @Override
    protected void onPause() {
        super.onPause();
        presenter.onPause();
    }

    @Override
    protected void onResume() {
        super.onResume();
        presenter.onResume();
    }

    @Override
    protected void onDestroy() {
        super.onDestroy();
        presenter.onDestroy();
    }

    private void initializeUI() {
        edtInches = findViewById(R.id.edtInches);
        edtFeet = findViewById(R.id.edtFeet);
        edtMiles = findViewById(R.id.edtMiles);
        txtViewResult = findViewById(R.id.txtViewResult);
        chkboxDisplayMetres = findViewById(R.id.chkboxDisplayMetres);
        Button btnConvertMiles = findViewById(R.id.btnConvert);
        btnConvertMiles.setOnClickListener(this);

    }

    /**
     * This method uses to check whether these edit text have value or not
     * @return
     */
    private boolean isValid() {
        return edtInches.getText().toString().length() > 0 &&
edtMiles.getText().toString().length() > 0 && edtFeet.getText().length() > 0;
    }

    @Override
    public void onClick(View view) {
        if (isValid()) {

presenter.onConvertButtonClick(Double.valueOf(edtInches.getText().toString())
,Double.valueOf(edtFeet.getText().toString()),
Double.valueOf(edtMiles.getText().toString()), chkboxDisplayMetres.isChecked());
        }
```

```
    }


    @Override
    public void displayResult(String displayLabel) {
        txtViewResult.setText(displayLabel);
    }
}
```

Figure 8: View in MVP

```
public interface ConvertView {
    void displayResult(String displayLabel);
}
```

Figure 9: View Interface in MVP

For implementing the MVP in Android application, developers must create two interfaces, one for an interface for each View layer (Activity or Fragment) and the other is for a Presenter. The Presenter interface is in charge of defining methods that need to use in the Presenter, while the View interface is used as the callback method which notifies the View to update the UI by the Presenter.

2.2.3 MVVM

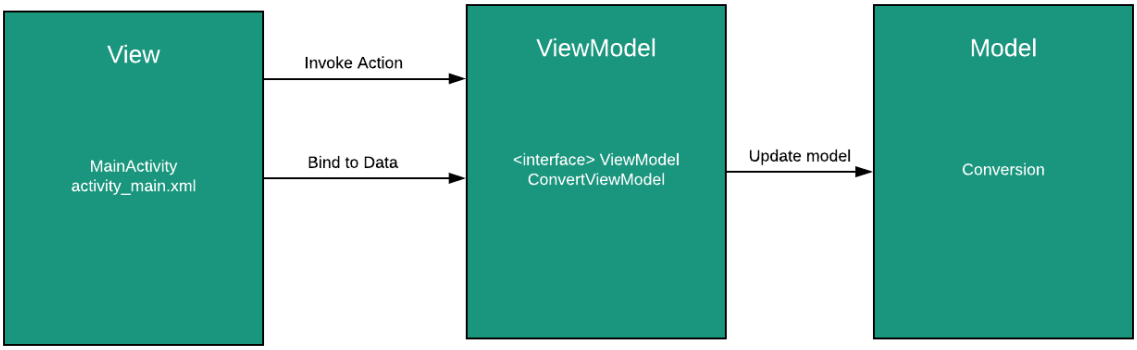

Figure 10: Apply MVVM in Android

The ViewModel in MVVM does not need control many responsibilities as the MVC pattern. Also, by using data binding, the ViewModel does not need to have a reference with an interface to trigger the callback method as the MVP pattern. More importantly, it reduces the massive code for the program and thus it improves the quality, testability and extensibility for the application.

Implementation of View

In order to implement the View in MVVM, the Data Binding Library of Android must be installed in gradle. Also, the root element of layout is <layout> including <data> - a define variable which uses for binding expressions.

```
<?xml version="1.0" encoding="utf-8"?>
<layout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools">

    <data>
        <import type="android.view.View" />
        <variable name="viewModel"
type="com.example.swinburne.w2_conversion.viewmodel.ConvertViewModel" />
    </data>
</layout>
```

Thereby, the UI components that we should implement the two-way data binding to notify the ViewModel when data is changed. This is the example of two-way data binding for the EditText in the application.

```
<EditText
    android:id="@+id/edtMiles"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_marginEnd="5dp"
    android:layout_marginStart="5dp"
    android:layout_weight="1"
    android:hint="Miles"
    android:inputType="numberDecimal"
    android:text="@={viewModel.mile}"/>
```

The EditText now notifies to the ViewModel whenever the text is changed so that the ViewModel can update the data and automatically update the data in this EditText. To listen to this event, the ViewModel uses the Observable pattern.

```
@Bindable
public String getMile() {
    return String.valueOf(model.getMile());
}

public void setMile(String mile) {
    if (!mile.isEmpty()) {
        if (Double.valueOf(mile) != model.getMile()) {
            model.setMile(Double.valueOf(mile));
            notifyChange();
        }
    }
}
```

The EditText sets its text by calling setMile() method and update its text with the the data from the model by invoking getMile() method in ViewModel. The ViewModel does not need to have a reference with the View but instead using the data binding to automatically trigger the event.

2.3 Evaluation

After implementing three architecture patterns MVC, MVP and MVVM (see Appendix for a full code implementation for these patterns and also, the link of them in github), in this part the evaluation is conducted to compare these three architectures in terms of testability, extensibility and maintenance of the application.

Starting with MVC, even though it is reckoned as the easiest one to start learning Android development, it should not be applied to the real application. Firstly, due to the dependence of the View and the Controller, performing a unit test on the Controller is such a challenge. Second of all, the Controller has many responsibilities (i.e. handle the app life cycle, handle the user actions, update the data model and manipulate the business logic) and thus when the application need to have more functionalities, the number of code in the Controller will dramatically raise. Last but not least, in case the application need to change the UI, the Controller must update as well to suit with the new UI which violates open-closed principle [7]. Based on these reasons, the MVC pattern in Android is only for developers to start learning Android as well as to practice, as it is simple to do and does not require much coding experience.

The MVP architecture is, on the other hand, believed to be more effective than the MVC. In MVP, the Activity and Fragment are reckoned as the View which takes responsibility for handling the UI logic and referencing the layout UI components. Therefore, unlike the Controller in MVC, it reduces massive tasks and spaghetti code for the Presenter. Additionally, by decoupling the view and the presenter completely, it improves the testability for the program. Nevertheless, the reference between View and Presenter (each View must have a Presenter to control) leads to the increasing of many Presenter classes and interfaces. Furthermore, it is hard to maintain a complicated function including multiple UI components in MVP [2]. In other words, the more complicated the screen is, the difficult the Presenter must handle these business logics.

The last architecture pattern is MVVM. By using data binding, it simplifies the ViewModel by detaching the dependence between View and ViewModel, which improves the testability of the application. Also, data binding reduces various interface in the MVP improving the extensibility of the application. The adding function can be implemented without impacting any running functions. To illustrate, in order to add new button, the developer simply creates a

button in layout and binds it with the view model method. In the ViewModel, the method with the same name is created to get the notification from the View. Whereas in MVP, the developer must create a button, receive the user action in the View, create a method in the View interface and the method in the Presenter to notify the View after data is updated.

In a nutshell, the evaluation of these patterns can be described as Table 1.

	MVC	MVP	MVVM
Testability	No – High coupling between View and Controller	Yes – View, Presenter and Model are decoupled	Yes – View, ViewModel and Model are decoupled
Extensibility	No – Changing the View leads must update the Controller	Yes - but it is not considered for a large project (more views lead to more presenters)	Yes – with the assistance of data binding, adding new features can be implemented easily.
Maintenance	No – The massive code in the Controller	No – many views and presenters for the application to control.	Yes – one ViewModel can be reused for many Views.

Table 1: Summary of evaluation

3 DISCUSSION

With the purpose of finding the best suited architecture patterns for Android development, this research has illustrated and implemented MVC, MVP and MVVM in Android. As the result, based on the evaluation, MVVM is the most appropriate pattern that Android developers should learn and apply to their application due to the testability, extensibility and maintenance. While MVC is recommended for beginner Android developers to start learning Android, as it is easy to learn and implement, but it is hard to manage in real project. MVP is better than MVC and requires less knowledge to implement compare to MVVM, but it is difficult to scale up as well as maintain the application.

Nevertheless, MVVM is not completely perfect as it is still a new architecture. Therefore, in order to improve the limitation of it, it is suggested to combine MVVM with RxJava to leverage the strength of data binding. Furthermore, future researchers can also research how to apply Clean Architecture in Android development and compare them with MVVM to identify the better pattern to improve the quality of Android development.

4 CONCLUSION

In recent years, the need for the design architecture in order to improve the testability, extensibility and maintenance for the application has led to the develop of many architecture patterns. In Android development, MVC, MVP and MVVM are 3 architecture designs that have been suggested by Google blueprints [2]. Meanwhile, MVC is the most common used, MVP and MVVM are the new one. Nevertheless, due to many limitation, MVC is not a sufficient pattern to use in the scalable project. Therefore, in this study, the rigorous evaluation of these three articles has been conducted to identify the best suited for Android development. As the result, although it is a need to improve in future work, MVVM is reckoned as the finest pattern for Android in terms of scalability and testability. This study also proposes the effective way to leverage the benefit of MVVM by using along with RxJava. This research also wants to help developers understand how to implement MVVM in their application.

5 REFERENCES

[1] N. 2018, "Google Play Store: number of apps 2018 | Statistic", Statista, 2018. [Online]. Available: https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/. [Accessed: 07- Nov- 2018]

[2] E. Maxwell, "MVC vs. MVP vs. MVVM on Android", *Academy.realm.io*, 2018. [Online]. Available: https://academy.realm.io/posts/eric-maxwell-mvc-mvp-and-mvvm-on-android/. [Accessed: 07- Nov- 2018].

[3] F. Muntenescu, "Android Architecture Patterns Part 1: Model-View-Controller", *Medium*, 2018. [Online]. Available: https://medium.com/upday-devs/android-architecture-patterns-part-1-model-view-controller-3baecef5f2b6. [Accessed: 07- Nov- 2018].

[4] T. Karpouzis, "Android Architecture – AndroidPub", *AndroidPub*, 2018. [Online]. Available: https://android.jlelse.eu/android-architecture-2f12e1c7d4db. [Accessed: 07- Nov- 2018].

[5] F. Muntenescu, "Android Architecture Patterns Part 3:
Model-View-ViewModel", 2018. [Online]. Available: https://upday.github.io/blog/model-view-viewmodel/. [Accessed: 07- Nov- 2018].

[6] "Introduction to Model/View/ViewModel pattern for building WPF apps", Tales from the Smart Client, 2018. [Online]. Available: https://blogs.msdn.microsoft.com/johngossman/2005/10/08/introduction-to-modelviewviewmodel-pattern-for-building-wpf-apps. [Accessed: 07- Nov- 2018].

[7] E. Gamma, R. Helm, R. Johnson and J. Vlissides, Design patterns. Reading, Mass.; Sydney: Addison-Wesley, 1994.

[8] F. Muntenescu, "Android Architecture Patterns Part 2: Model-View-Presenter", *Medium*, 2018. [Online]. Available: https://medium.com/upday-devs/android-architecture-patterns-part-2-model-view-presenter-8a6faaae14a5. [Accessed: 07- Nov- 2018].

6 APPENDIX

Link github: https://github.com/duythuc28/Android_Patterns

6.1 MVC Implementation

Model

```
public class Conversion {
    private double inch;
    private double feet;
    private double mile;
    private boolean isMetres;

    /**
     * Constructor of conversion class
     * @param inch inch value
     * @param feet feet value
     * @param mile the value of mile
     * @param isMetres the indicator to show whether metres or cm
     */
    public Conversion(double inch, double feet, double mile, boolean isMetres) {
        this.inch = inch;
        this.feet = feet;
        this.mile = mile;
        this.isMetres = isMetres;
    }

    public Conversion() {
        this.inch = 0;
        this.feet = 0;
        this.mile = 0;
        this.isMetres = false;
    }

    public void setData(double inch, double feet, double mile, boolean isMetres) {
        this.inch = inch;
        this.feet = feet;
        this.mile = mile;
        this.isMetres = isMetres;
    }

    /**
     * This method uses to convert the data from inch, mile, feet to cm or metres
     * @return
     */
    public double toValue() {
        double inch = this.inch + this.feet * 12 + this.mile * 5280 * 12;
        if (isMetres)  {
            return inch * 2.54/100;
        }
        return (inch * 2.54);
    }
}
```

View

```
<TableLayout
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:layout_marginTop="8dp">
```

```
    <TableRow >

        <EditText
            android:id="@+id/edtMiles"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:layout_marginEnd="5dp"
            android:layout_marginStart="5dp"
            android:layout_weight="1"
            android:hint="Miles"
            android:inputType="numberDecimal" />

        <EditText
            android:id="@+id/edtFeet"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:hint="Feet"
            android:inputType="numberDecimal"
            android:layout_marginStart="5dp"
            android:layout_marginEnd="5dp"
            android:layout_weight="1"
            />

        <EditText
            android:id="@+id/edtInches"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:hint="Inches"
            android:inputType="numberDecimal"
            android:layout_marginStart="5dp"
            android:layout_marginEnd="5dp"
            android:layout_weight="1">
        </EditText>

    </TableRow>

    <TableRow>
        <Button
            android:id="@+id/btnConvert"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="@string/btn_convert"
            android:layout_weight="1"
            />
    </TableRow>

    <LinearLayout
        android:layout_marginTop="5dp"
        >
        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_marginStart="5dp"
            android:text="@string/result"
            android:textStyle="bold" />
        <TextView
            android:id="@+id/txtViewResult"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_marginStart="5dp"
            android:text=""
            android:textStyle="bold"
            android:layout_weight="1"/>
```

```
        <CheckBox
            android:id="@+id/chkboxDisplayMetres"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_marginEnd="5dp"
            android:text="@string/checkbox_metres"/>
    </LinearLayout>

</TableLayout>
```

Controller

```
public class MainActivity extends AppCompatActivity implements View.OnClickListener
{

    // Establish references with the view layout
    EditText edtFeet;
    EditText edtMiles;
    EditText edtInches;
    CheckBox chkboxDisplayMetres;
    TextView txtViewResult;

    private Conversion conversion;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        initializeUI();
        // Instantiate the model
        conversion = new Conversion();
    }

    private void initializeUI() {
        edtInches = findViewById(R.id.edtInches);
        edtFeet = findViewById(R.id.edtFeet);
        edtMiles = findViewById(R.id.edtMiles);
        txtViewResult = findViewById(R.id.txtViewResult);
        chkboxDisplayMetres = findViewById(R.id.chkboxDisplayMetres);

        Button btnConvertMiles = findViewById(R.id.btnConvert);
        btnConvertMiles.setOnClickListener(this);

    }

    /**
     * This method uses to check whether these edit text have value or not
     * @return
     */
    private boolean isValid() {
        return edtInches.getText().toString().length() > 0 &&
edtMiles.getText().toString().length() > 0 && edtFeet.getText().length() > 0;
    }

    @Override
    public void onClick(View view) {
        String result;
        String metres = getResources().getString(R.string.metres);
        if (isValid()) {
            conversion.setData(Double.valueOf(edtInches.getText().toString())
,Double.valueOf(edtFeet.getText().toString()),
Double.valueOf(edtMiles.getText().toString()), chkboxDisplayMetres.isChecked());
            if (chkboxDisplayMetres.isChecked()) {
                result = String.format("%.2f %s", conversion.toValue(), metres);
```

```
            } else {
                result = String.format("%.2f cm", conversion.toValue());
            }
        } else {
            result = "";
            Toast.makeText(getApplicationContext(), "You have to enter a valid
value", Toast.LENGTH_SHORT).show();
        }
        txtViewResult.setText(result);
    }
}
```

6.2 MVP

Model - the same as MVC

View

```
public class MainActivity extends AppCompatActivity implements
View.OnClickListener, ConvertView {

    EditText edtFeet;
    EditText edtMiles;
    EditText edtInches;
    CheckBox chkboxDisplayMetres;
    TextView txtViewResult;
    ConvertPresenter presenter = new ConvertPresenter(this);

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        initializeUI();
        presenter.onCreate();
    }

    @Override
    protected void onPause() {
        super.onPause();
        presenter.onPause();
    }

    @Override
    protected void onResume() {
        super.onResume();
        presenter.onResume();
    }

    @Override
    protected void onDestroy() {
        super.onDestroy();
        presenter.onDestroy();
    }

    private void initializeUI() {
        edtInches = findViewById(R.id.edtInches);
        edtFeet = findViewById(R.id.edtFeet);
        edtMiles = findViewById(R.id.edtMiles);
        txtViewResult = findViewById(R.id.txtViewResult);
        chkboxDisplayMetres = findViewById(R.id.chkboxDisplayMetres);
        Button btnConvertMiles = findViewById(R.id.btnConvert);
        btnConvertMiles.setOnClickListener(this);
```

```
    }

    /**
     * This method uses to check whether these edit text have value or not
     * @return
     */
    private boolean isValid() {
        return edtInches.getText().toString().length() > 0 &&
edtMiles.getText().toString().length() > 0 && edtFeet.getText().length() > 0;
    }

    @Override
    public void onClick(View view) {
        if (isValid()) {

presenter.onConvertButtonClick(Double.valueOf(edtInches.getText().toString())
,Double.valueOf(edtFeet.getText().toString()),
Double.valueOf(edtMiles.getText().toString()), chkboxDisplayMetres.isChecked());
        }
    }


    @Override
    public void displayResult(String displayLabel) {
        txtViewResult.setText(displayLabel);
    }
}
```

View Interface

```
public interface ConvertView {
    void displayResult(String displayLabel);
}
```

Presenter

```
public class ConvertPresenter implements IPresenter {

    private ConvertView view;
    private Conversion model;

    public ConvertPresenter(ConvertView view) {
        this.view = view;
        this.model = new Conversion();
    }

    @Override
    public void onCreate() {
        model = new Conversion();
    }

    @Override
    public void onPause() {

    }

    @Override
    public void onResume() {

    }

    @Override
    public void onDestroy() {

    }
```

```
    // When the user select convert button, the presenter receives the actions from
    // View layer to set the data. Then, after computing successfully,
    // it invokes the callback to notify the view to update the UI based on the new
data
    @Override
    public void onConvertButtonClick(double inch, double feet, double mile, boolean
isMetres) {
        model.setData(inch, feet, mile, isMetres);
        String result;
        if (isMetres) {
            result = String.format("%.2f meters", model.toValue());
        } else {
            result = String.format("%.2f cm", model.toValue());
        }
        view.displayResult(String.valueOf(result));
    }
}
```

6.3 MVVM

Model – the same as MVC and MVP

View

layout

```
<?xml version="1.0" encoding="utf-8"?>
<layout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools">

    <data>
        <import type="android.view.View" />
        <variable name="viewModel"
type="com.example.swinburne.w2_conversion.viewmodel.ConvertViewModel" />
    </data>

    <TableLayout
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:layout_marginTop="8dp"
        tools:context="com.example.swinburne.w2_conversion.view.MainActivity">
        <TableRow >

            <EditText
                android:id="@+id/edtMiles"
                android:layout_width="match_parent"
                android:layout_height="wrap_content"
                android:layout_marginEnd="5dp"
                android:layout_marginStart="5dp"
                android:layout_weight="1"
                android:hint="Miles"
                android:inputType="numberDecimal"
                android:text="@={viewModel.mile}"/>

            <EditText
                android:id="@+id/edtFeet"
                android:layout_width="match_parent"
                android:layout_height="wrap_content"
                android:hint="Feet"
                android:inputType="numberDecimal"
                android:layout_marginStart="5dp"
```

```
                android:layout_marginEnd="5dp"
                android:layout_weight="1"
                android:text="@={viewModel.feet}"
                />

            <EditText
                android:id="@+id/edtInches"
                android:layout_width="match_parent"
                android:layout_height="wrap_content"
                android:hint="Inches"
                android:inputType="numberDecimal"
                android:layout_marginStart="5dp"
                android:layout_marginEnd="5dp"
                android:layout_weight="1"
                android:text="@={viewModel.inch}">
            </EditText>

        </TableRow>

        <TableRow>
            <Button
                android:id="@+id/btnConvert"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:text="@string/btn_convert"
                android:layout_weight="1"
                android:onClick="@{() -> viewModel.onConvertButtonClick()}"
                />
        </TableRow>

        <LinearLayout
            android:layout_marginTop="5dp"
            >
            <TextView
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:layout_marginStart="5dp"
                android:text="@string/result"
                android:textStyle="bold" />
            <TextView
                android:id="@+id/txtViewResult"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:layout_marginStart="5dp"
                android:text="@{viewModel.convertResult}"
                android:textStyle="bold"
                android:layout_weight="1"/>

            <CheckBox
                android:id="@+id/chkboxDisplayMetres"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:layout_marginEnd="5dp"
                android:text="@string/checkbox_metres"
                android:checked="@={viewModel.meters}"
                />
        </LinearLayout>

    </TableLayout>

</layout>
```

Activity

```
public class MainActivity extends AppCompatActivity {

    ConvertViewModel viewModel = new ConvertViewModel();

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        ActivityMainBinding binding = DataBindingUtil.setContentView(this,
R.layout.activity_main);
        binding.setViewModel(viewModel);
        viewModel.onCreate();
    }

    @Override
    protected void onPause() {
        super.onPause();
        viewModel.onPause();
    }

    @Override
    protected void onResume() {
        super.onResume();
        viewModel.onResume();
    }

    @Override
    protected void onDestroy() {
        super.onDestroy();
        viewModel.onDestroy();
    }
}
```

ViewModel

```
public class ConvertViewModel extends BaseObservable implements IViewModel {

    private Conversion model;

    public final ObservableField<String> convertResult = new ObservableField<>();

    public ConvertViewModel() {
        model = new Conversion();
    }

    @Override
    public void onCreate() {

    }

    @Override
    public void onPause() {

    }

    @Override
    public void onResume() {

    }

    @Override
    public void onDestroy() {
```

```
    }

    public void onConvertButtonClick() {
        String result;
        if (model.isMetres()) {
            result = String.format("%.2f meters", model.toValue());
        } else {
            result = String.format("%.2f cm", model.toValue());
        }
        convertResult.set(result);
    }


    @Bindable
    public String getInch() {
        return String.valueOf(model.getInch());
    }

    public void setInch(String inch) {
        if (!inch.isEmpty()) {
            if (Double.valueOf(inch) != model.getInch()) {
                model.setInch(Double.valueOf(inch));
                notifyChange();
            }
        }
    }

    @Bindable
    public String getFeet() {
        return String.valueOf(model.getFeet());
    }

    public void setFeet(String feet) {
        // Avoids infinite loops.
        if (!feet.isEmpty()) {
            if (Double.valueOf(feet) != model.getFeet()) {
                model.setFeet(Double.valueOf(feet));
                notifyChange();
            }
        }
    }

    @Bindable
    public String getMile() {
        return String.valueOf(model.getMile());
    }

    public void setMile(String mile) {
        if (!mile.isEmpty()) {
            if (Double.valueOf(mile) != model.getMile()) {
                model.setMile(Double.valueOf(mile));
                notifyChange();
            }
        }
    }

    @Bindable
    public Boolean getMeters() {
        return model.isMetres();
    }

    public void setMeters(Boolean meters) {
        if (model.isMetres() != meters) {
            model.setMetres(meters);
```

```
            notifyChange();
        }
    }
}
```