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

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, but 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 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

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.

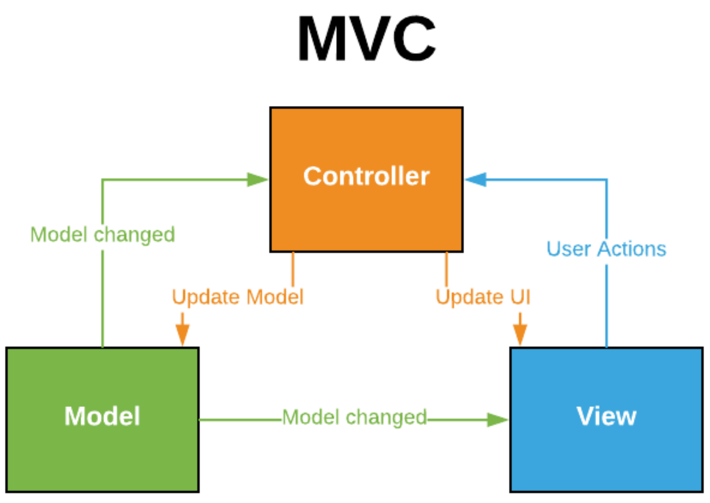


Figure 1: MVC pattern

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].

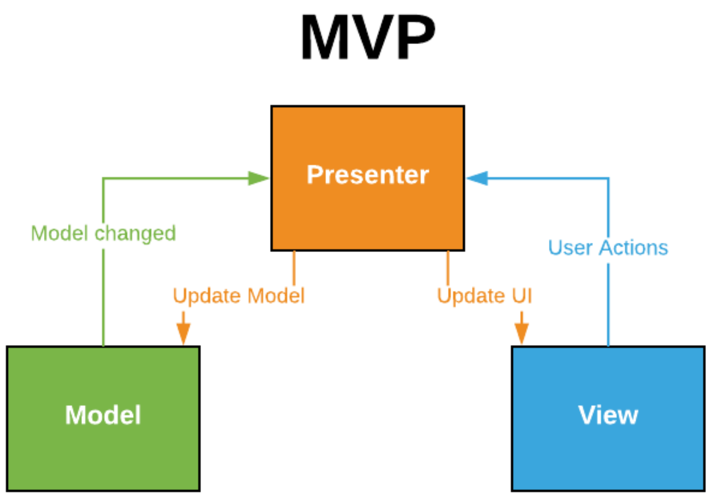


Figure 2: MVP architecture pattern

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.

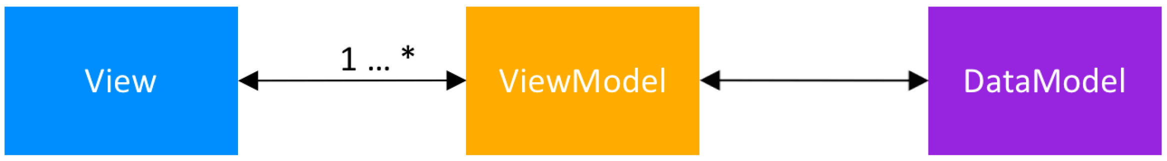


Figure 3: MVVM architecture pattern

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

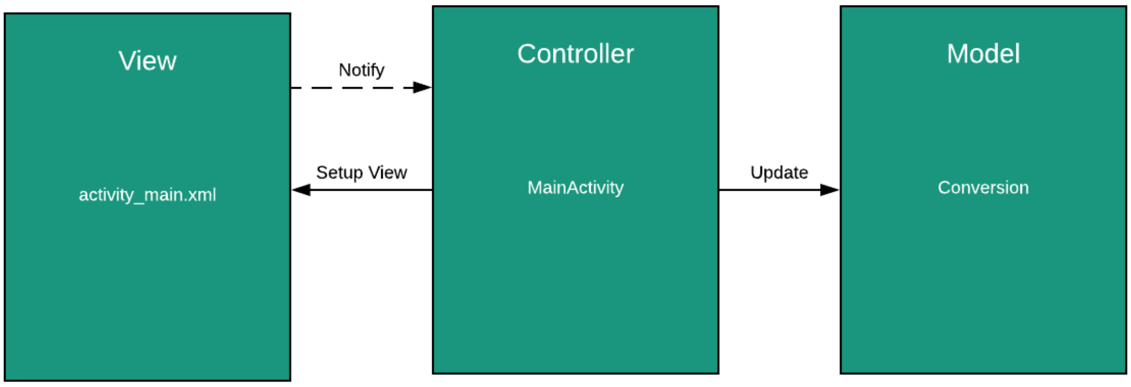
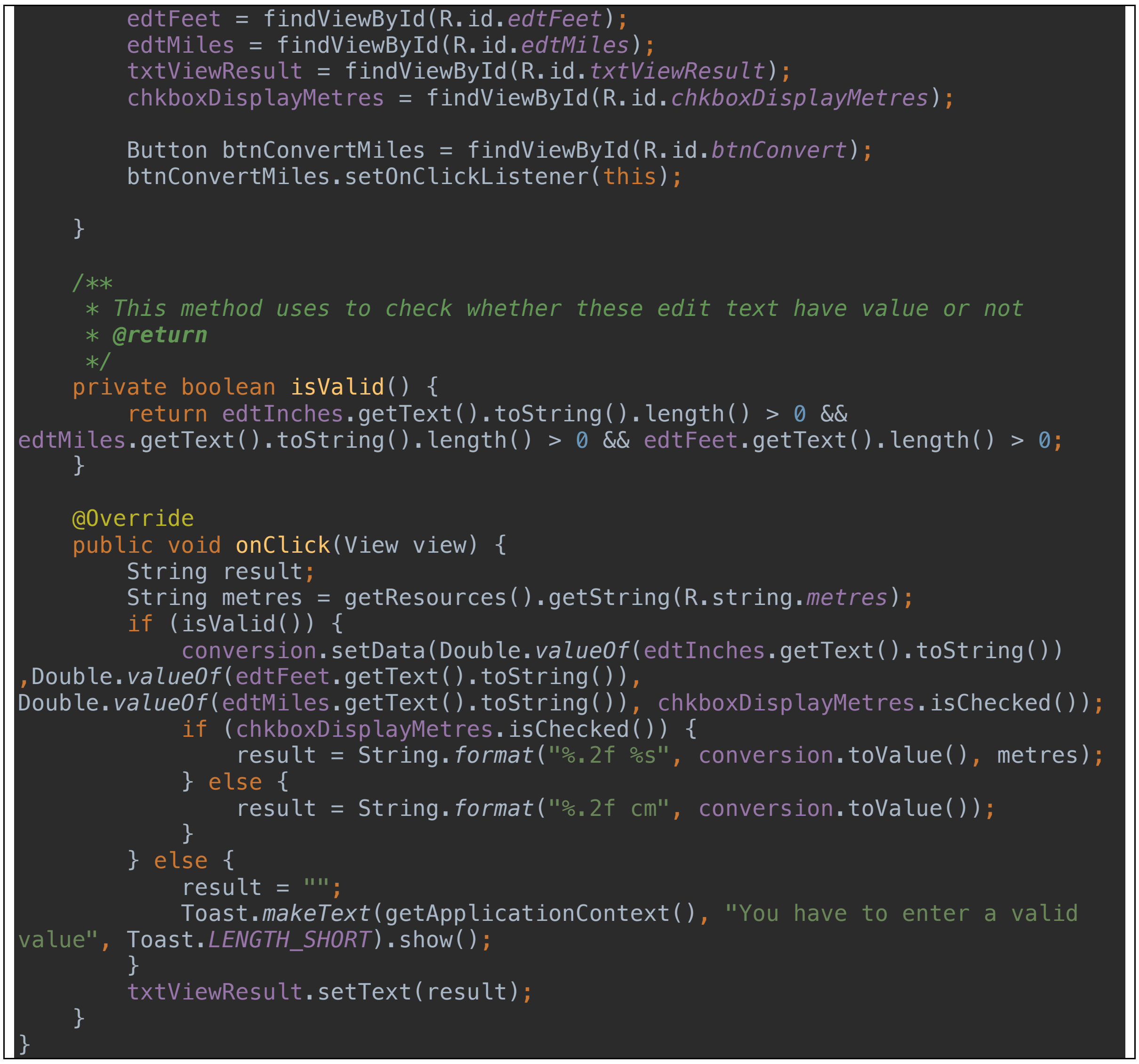


Figure 4: Apply MVC in Android

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

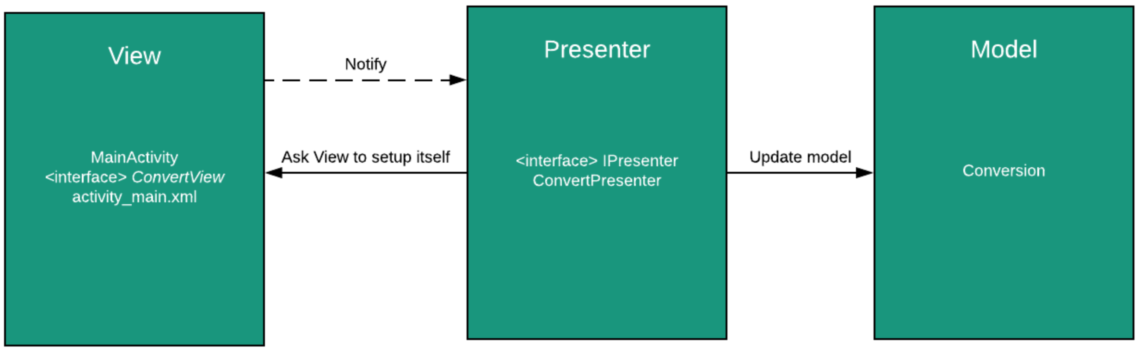
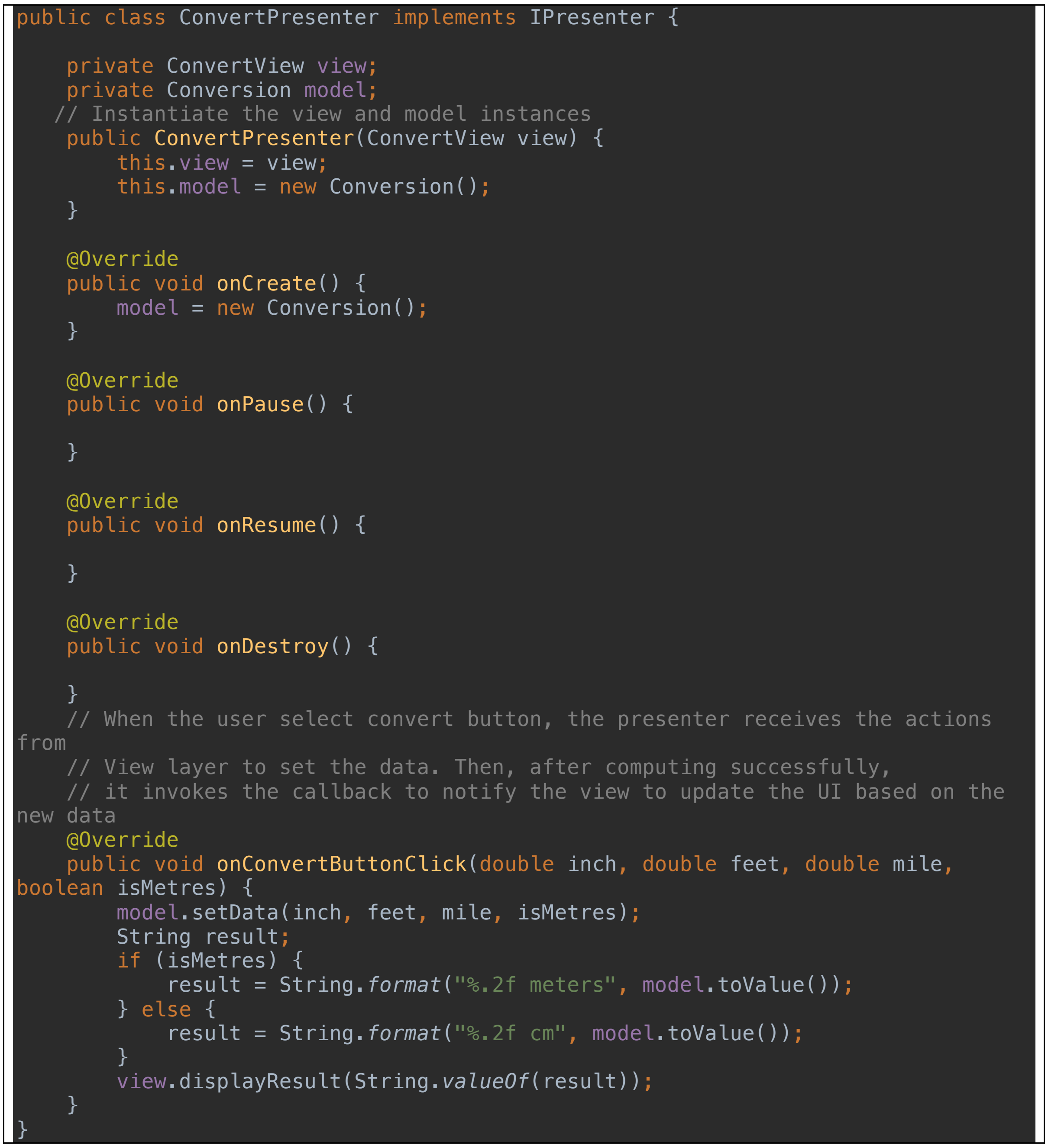


Figure 6: Apply MVP in Android

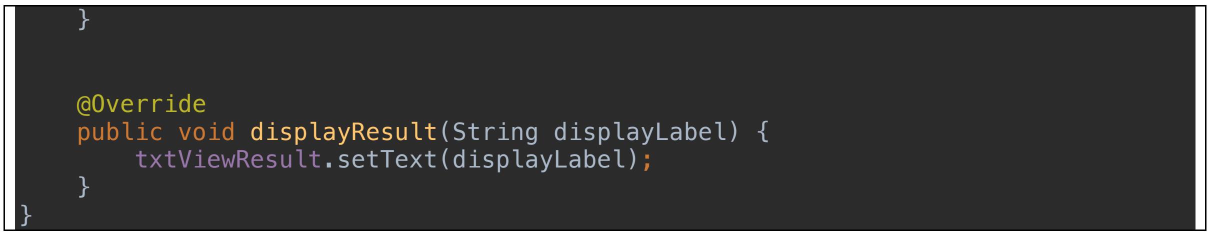
The design and implementation for MVP pattern are slightly different from MVC pattern. While the model layer is the same, View and Presenter are now communicating with each other via an interface. The View is now including the Activity/Fragment along with the xml layout and also, the View interface which 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));
    }
}
```

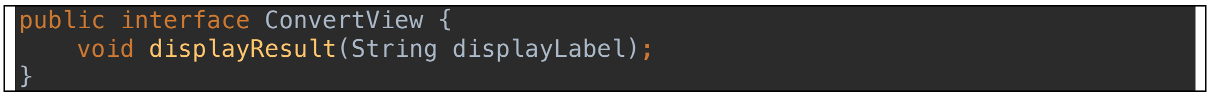
Figure 7: Presenter in practical

```java
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

In other words, to implement the MVP in Android application, developers must create an interface for each View layer (Activity or Fragment) and an interface for a Presenter. The interface of Presenter defines 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

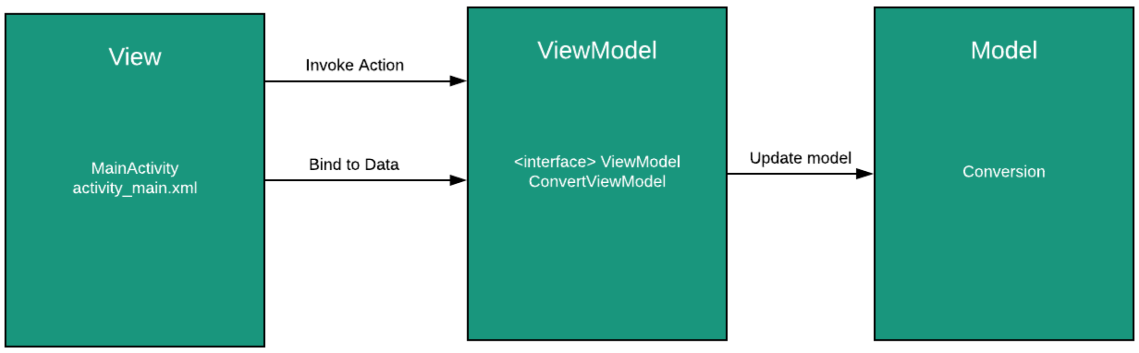


Figure 10: Apply MVVM in Android

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 set 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 full code version as well as the github to download them), in this part the evaluation is conducted to compare these three architectures in terms of testability, extensibility and maintenance the application. Starting with MVC, even though it is the easiest one to start learning Android development, it should not be implemented in real project because bla bla bla.

The second architecture is MVP, it is good but interface class increase throughout of time is a big challenges

The last architecture is MVVM, it is good, testable, extensible but the problem of it is

MVC	MVP	MVVM

3 DISCUSSION

4 CONCLUSION

5 REFERENCES

[1] https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/

[2] https://academy.realm.io/posts/eric-maxwell-mvc-mvp-and-mvvm-on-android/

[3] https://medium.com/upday-devs/android-architecture-patterns-part-1-model-view-controller-3baecef5f2b6

[4] https://android.jlelse.eu/android-architecture-2f12e1c7d4db

[5] https://upday.github.io/blog/model-view-viewmodel/

[6]https://blogs.msdn.microsoft.com/johngossman/2005/10/08/introduction-to-modelviewviewmodel-pattern-for-building-wpf-apps

6 APPENDIX