Application Design Pattern

Mobile App Programming

Design Pattern

- Term in Software Engineering
- General, reusable solution to commonly occurring problem within a given context in software design

- General solution, distinguishable with paradigm and algorithm

Design Pattern

- In this lesson, we will discuss about some examples of architectural pattern
  - MVC: Model-View-Controller
  - MVP: Model-View-Presenter
  - MVVM: Model-View-ViewModel

Disclaimer:

This lecture contents may be different from the design pattern with same name covered on other places such as Software Engineering lecture.

Design Pattern

- Why we need design pattern?
- Recap PA1
  - There are some listviews
  - Each listview have its own layout
  - Each listview have its own data
  - Each listview should show its data to user
- Easy to program?
  - What if there were teammates?

Design Pattern

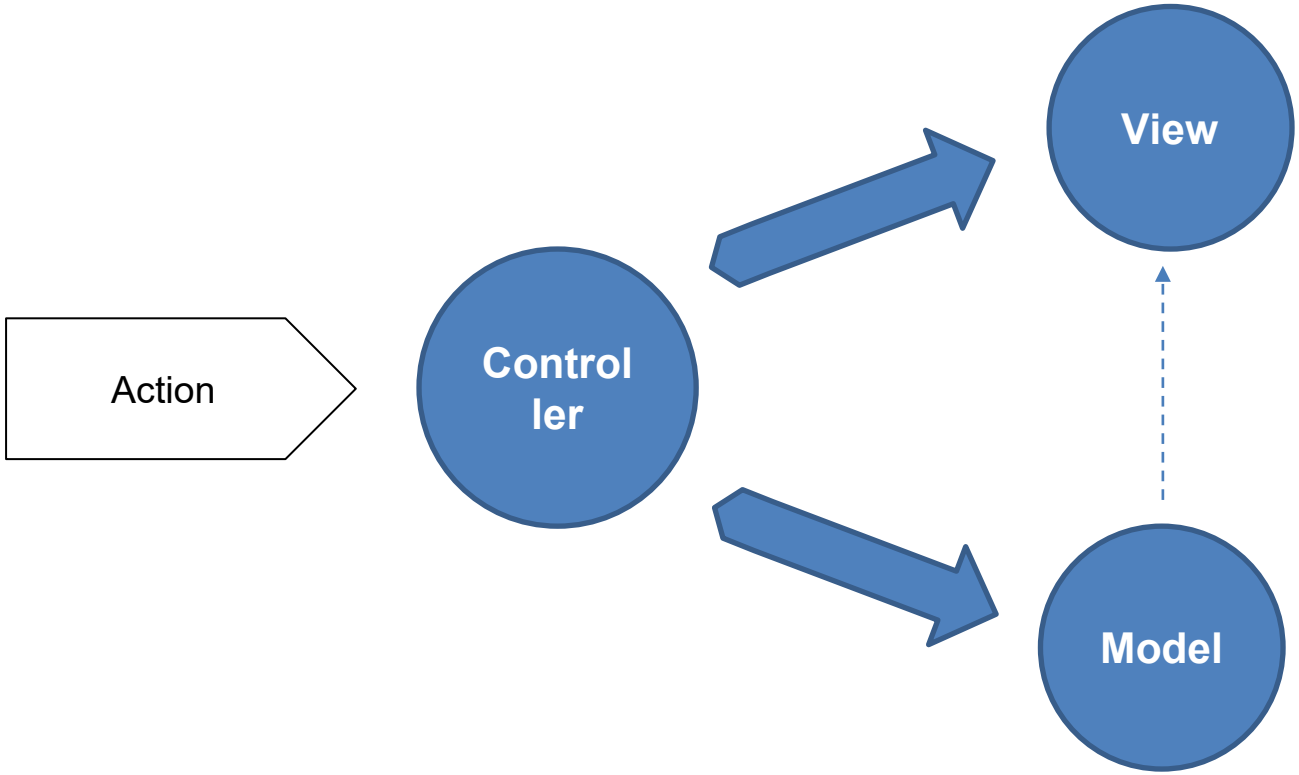
- Listview
  - Usually, it have its own layout.xml
  - Usually, it have its own Adapter.java
  - Usually, it is in ContainingActivity.java
  - Then, which point to handle incoming data?
  - Then, which point to …
- Regularize the process of refining, storing, presenting, … the data
  - Known approaches to problem in software engineering

MVC

- Model / View / Controller
  - View
    - UI shown to user
    - Receive data from Model and show to user
  - Model
    - Manage data
    - Not dependent to View
  - Controller
    - Process Action from user
    - Select View according to change of data in Model
    - Usually, Activity or Fragment in Android

MVC

- Model / View / Controller



MVC

- Model / View / Controller
  - View
    - Expression of Model
    - Not knowing anything, just showing
  - Model
    - Data + Status + Logic
    - Just processing the data
  - Controller
    - Receive Action from View then determine how to interact with Model
  - Originally, Model let View to update UI
    But in this lecture, Controller let View to update UI
    - MVC is hard to perfect-fit in Android

MVC

- Summarize the things to implement
  - View
    - **Show** data
  - Model
    - **Process** data
  - Controller
    - **Bind** View and Model

- Code may not be solidly distinct

MVC

- Simple application
  - Press button -> Count up

- View
  - activity_main.xml
- Model
  - MySimpleModel.java
- Controller
  - MainActivity.java

MVC

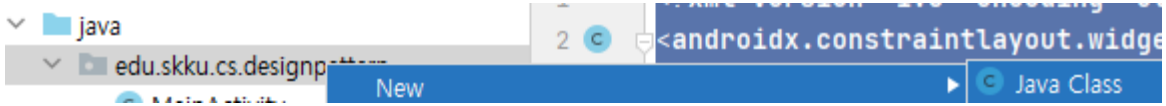
- View
  - activity_main.xml
  - Just showing data
  - Write activity_main.xml then Android will manage it

```
<Button
   android:id="@+id/button"
   android:layout_width="wrap_content"
   android:layout_height="wrap_content"
   android:text="Button"
   app:layout_constraintStart_toStartOf="parent"
   app:layout_constraintTop_toTopOf="parent" />
```

https://gist.github.com/devquint/195b7e133de1aacb4e6518f4a33372d0

MVC

- Model
  - Create MySimpleModel.java
  - Process data with (add1 method), (getValue method)




```
import java.util.Observable;

public class MySimpleModel extends Observable {
    private int value;

    public MySimpleModel(int initialValue){
        this.value = initialValue;
    }

    public void addOne(){
        this.value += 1;
        setChanged();
        notifyObservers();
    }

    public int getValue(){
        return this.value;
    }
}
```

https://gist.github.com/devquint/195b7e133de1aacb4e6518f4a33372d0

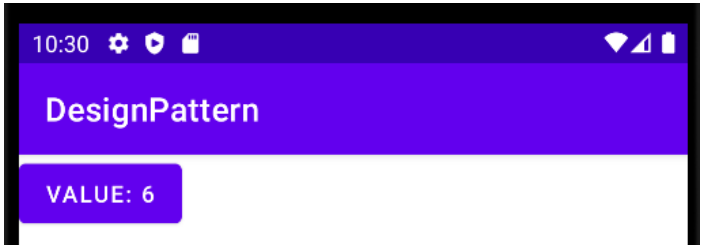
MVC

- Controller
  - MainActivity.java
  - Init View(setContextView) and Model(new MySimpleModel(0))
  - Get action from user which passed from View(new onClickListener…) then interact with Model properly(addOne)
  - Update View(setText) with data of Model when it should be changed(update from MySimpleModel)

https://gist.github.com/devquint/195b7e133de1aacb4e6518f4a33372d0

MVC

- Controller
  - MainActivity.java




```
import androidx.appcompat.app.AppCompatActivity;

import android.os.Bundle;
import android.widget.Button;

public class MainActivity extends AppCompatActivity {

    private MySimpleModel model;

    private Button btn;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        setContentView(R.layout.activity_main); // init View

        model = new MySimpleModel(0); // init Model
        model.addObserver((observable, o) -> {MainActivity.this.updateView();});

        btn = findViewById(R.id.button);
        updateView();
        btn.setOnClickListener(view -> {model.addOne();});
    }

    public void updateView(){
        int value = model.getValue(); // get value from Model
        btn.setText("Value: " + value); // change View
    }

}
```

https://gist.github.com/devquint/195b7e133de1aacb4e6518f4a33372d0

MVC

- Recap
  - View
    - **Show** data
    - Make .xml then Android do all
  - Model
    - **Process** data
    - Store and change data (+ API call, calculating, …)
  - Controller
    - **Bind** View and Model
    - Initialize both and handle events

https://gist.github.com/devquint/195b7e133de1aacb4e6518f4a33372d0

MVC

- Pros
  - Simple
  - Almost nothing to test with View
- Cons
  - Controller is quite dependent to Android API
  - Controller is quite dependent to View
    - View changed, then Controller may be changed
  - Controller do most of things
    - Bigger application? Long Controller code

https://gist.github.com/devquint/195b7e133de1aacb4e6518f4a33372d0

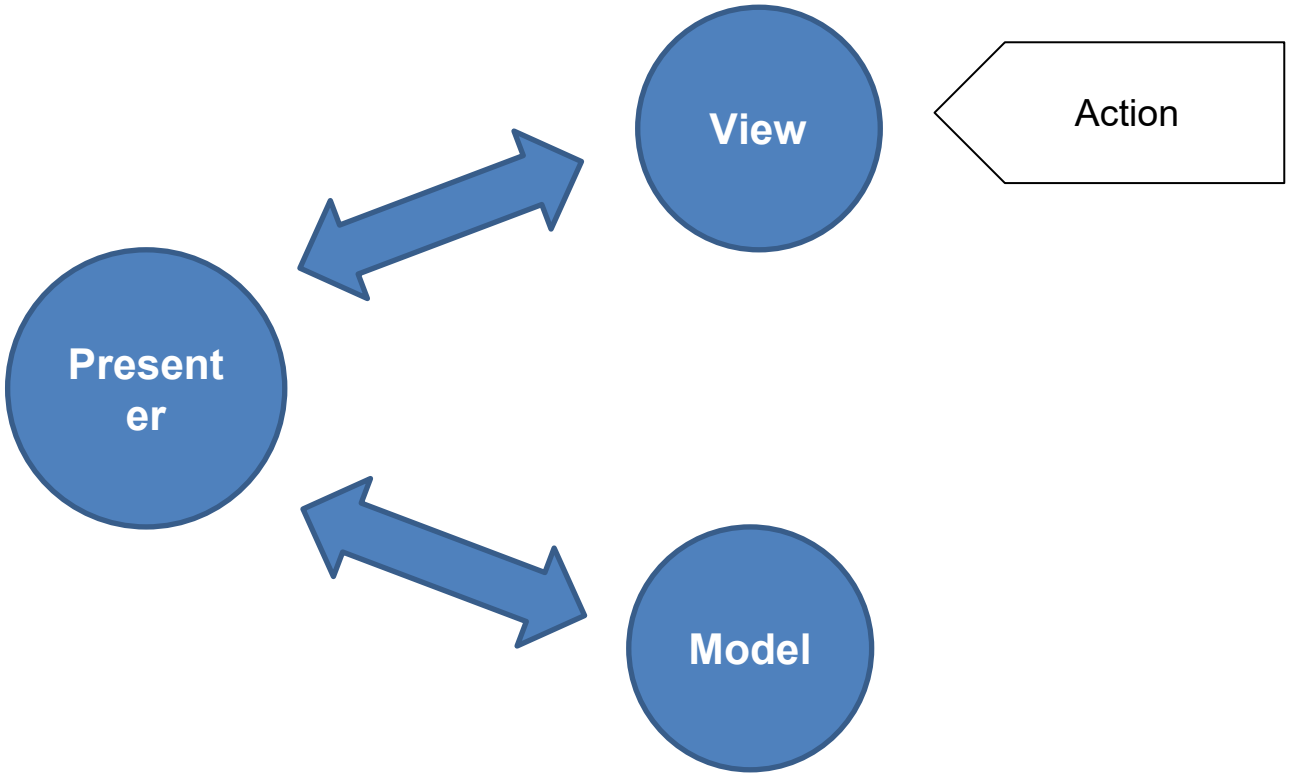
MVP

- Model / View / Presenter
  - Model is same
  - View is different
- View
  - Activity/Fragment is View in MVP while not in MVC
  - Change of UI is managed on View, with Interface in Presenter
- Presenter
  - Interconnect of View and Model
  - Presenter is Interface while Controller is normal class

MVP

- Model / View / Presenter
  - View
    - Expression of Model
    - Using interface of Presenter, manage itself
  - Model
    - Data + Status + Logic
    - Just processing the data
  - Presenter
    - Let View to interact with Model
    - According to changing data in Model, let View to know that it should be updated

MVP



MVP

- Not very different with MVC
- View will handle more things
- Controller->Presenter will handle less things

- MVC
  - Action from View, passed to Controller, Controller let Model to change value
- MVP
  - Action from View, View let Model to change value via/through Presenter

MVP

- Summarize the things to implement
  - View
    - **Show** data and manage UI
  - Model
    - **Process** data
  - Controller
    - Act as a **conduit** of View and Model

- Code may not be solidly distinct

MVP

- Simple application, same as previous
  - Press button -> Count up

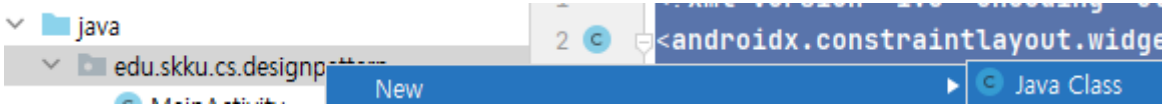
- View
  - activity_main.xml, MainActivity.java
- Model
  - MySimpleModel.java
- Presenter
  - MySimplePresenter.java

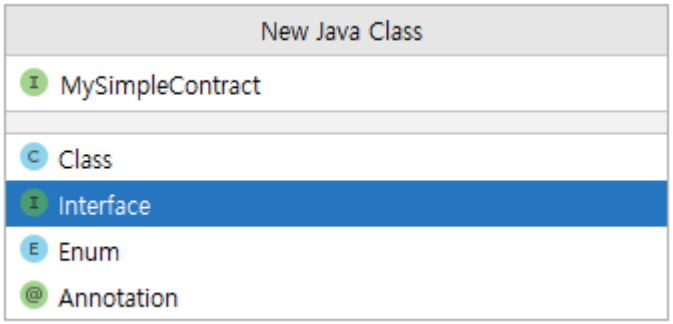
- Contract: specify what should be implemented, part of presenter
  - MySimpleContract.java

https://gist.github.com/devquint/09232498dc9f938a363998fcb6b73fc1

MVP

- Presenter
  - Serve the interface
  - To specify the restraint, we will make Contract first.
  - Make interface – MySimpleContract.java





https://gist.github.com/devquint/09232498dc9f938a363998fcb6b73fc1

MVP

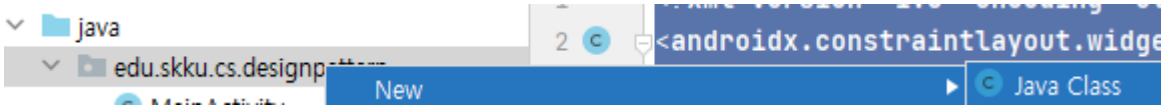
- Presenter
  - To specify the restraint, we will make Contract first

```
public interface MySimpleContract {
    interface ContractForView{
        void displayValue(int value);
    }

    interface ContractForModel{
        int getValue();
        void addOne(OnValueChangedListener listener);
        interface OnValueChangedListener{
            void onChanged();
        }
    }

    interface ContractForPresenter{
        void onAddButtonTouched();
    }
}
```

https://gist.github.com/devquint/09232498dc9f938a363998fcb6b73fc1

MVP

- Presenter
  - Then implement presenter class code: MySimplePresenter.java




```
public class MySimplePresenter implements MySimpleContract.ContractForPresenter,
        MySimpleContract.ContractForModel.OnValueChangedListener{

    private MySimpleContract.ContractForView view;
    private MySimpleContract.ContractForModel model;

    public MySimplePresenter(MySimpleContract.ContractForView view,
                             MySimpleContract.ContractForModel model){
        this.view = view;
        this.model = model;
    }

    @Override
    public void onAddButtonTouched() {
        model.addOne(this);
    }

    @Override
    public void onChanged() {
        if(view != null) view.displayValue(model.getValue());
    }
}
```

https://gist.github.com/devquint/09232498dc9f938a363998fcb6b73fc1

MVP

- Presenter
  - This make connection to Model and View

```
public class MySimplePresenter implements MySimpleContract.ContractForPresenter,
        MySimpleContract.ContractForModel.OnValueChangedListener{

    private MySimpleContract.ContractForView view;
    private MySimpleContract.ContractForModel model;

    public MySimplePresenter(MySimpleContract.ContractForView view,
                             MySimpleContract.ContractForModel model){
        this.view = view;
        this.model = model;
    }

    @Override
    public void onAddButtonTouched() {
        model.addOne(this);
    }

    @Override
    public void onChanged() {
        if(view != null) view.displayValue(model.getValue());
    }
}
```

https://gist.github.com/devquint/09232498dc9f938a363998fcb6b73fc1

MVP

- Model
  - Fix to fit with Contract

```
public class MySimpleModel implements MySimpleContract.ContractForModel{
    private int value;

    public MySimpleModel(int initialValue){
        this.value = initialValue;
    }

    @Override
    public int getValue(){
        return this.value;
    }

    @Override
    public void addOne(OnValueChangedListener listener) {
        this.value += 1;
        listener.onChanged();
    }
}
```

https://gist.github.com/devquint/09232498dc9f938a363998fcb6b73fc1

MVP

- View
  - Connect with Presenter

```
public class MainActivity extends AppCompatActivity
        implements MySimpleContract.ContractForView{

    private MySimplePresenter presenter;
    private Button btn;

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

        btn = findViewById(R.id.button);

        presenter = new MySimplePresenter(this, new MySimpleModel(0));
        presenter.onChanged();
        btn.setOnClickListener(view -> {presenter.onAddButtonTouched();});
    }

    @Override
    public void displayValue(int value) {
        btn.setText("Value: " + value); // update UI
    }
}
```

https://gist.github.com/devquint/09232498dc9f938a363998fcb6b73fc1

MVP

- Presenter is good to be independent with Android API
  - In example code, it will not use btn.setText(…) directly on Presenter
  - Just call method of View
    - Then View call btn.setText(…)

```
@Override
public void onChanged() {
    if(view != null) view.displayValue(model.getValue());
}
```

```
presenter = new MySimplePresenter(this, new MySimpleModel(0));
    presenter.onChanged();
    btn.setOnClickListener(view -> {presenter.onAddButtonTouched();});
}

@Override
public void displayValue(int value) {
    btn.setText("Value: " + value); // update UI
}
```

https://gist.github.com/devquint/09232498dc9f938a363998fcb6b73fc1

MVP

- Recap
  - View
    - **Show** data and communicate with Android API
    - Make .xml and editing them.
  - Model
    - **Process** data
    - Store and change data (+ API call, calculating, …)
  - Presenter
    - **Interconnect** View and Model
    - Make View to communicate with Model properly

MVP

- Pros
  - Unit test is easier than MVC
  - Presenter is now independent with Android API
    - (If it is implemented well)

- Cons
  - High dependency between View and Presenter
  - There still could be too much code in Presenter

MVVM

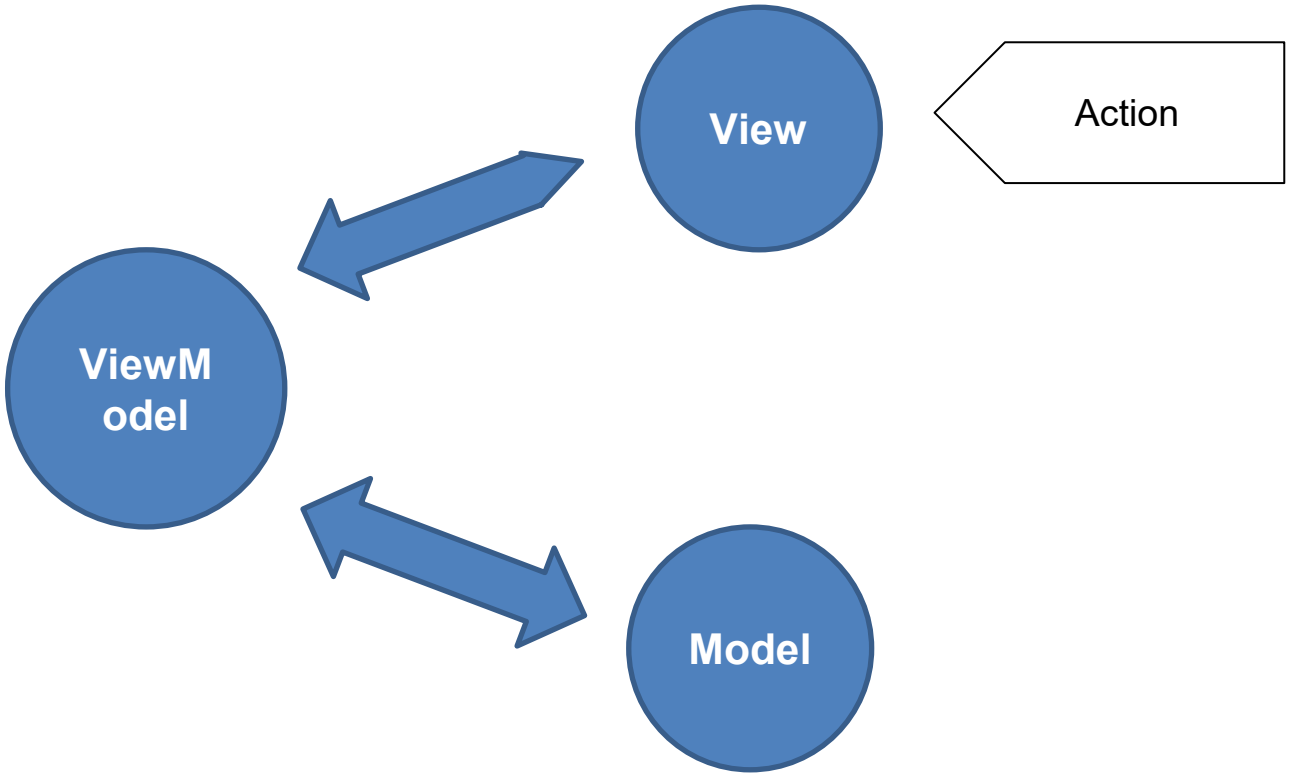
- Model / View / ViewModel
  - Model: same as MVC/MVP
  - View: some modification from MVP

  - View
    - Now update UI while observing ViewModel
    - It will use Data Binding
  - ViewModel
    - Independent with View, wrapping Model
    - Give Hook to View
      - with this, View can send event to Model

MVVM

- Model / View / ViewModel
  - View
    - Expression of Model
    - Passing event via hook of ViewModel
    - Refreshing UI by monitoring ViewModel
    - Data Binding
  - Model
    - Data + Status + Logic
    - Just processing the data
  - ViewModel
    - By hook, let View to manage Model

MVVM



MVVM

- Summarize the things to implement
  - View
    - **Let ViewModel** to work
  - Model
    - **Process** data
  - ViewModel
    - Manage View and Model

- Code may not be solidly distinct

MVVM

- Simple application, same as previous
  - Press button -> Count up

- View
  - Modify build.gradle(Module: app) to use data binding
  - activity_main.xml, MainActivity.java
- Model
  - MySimpleModel.java
- ViewModel
  - MySimpleViewModel.java

MVVM

- Pros
  - Only View could be changed for different UI
  - Unit test is easier since there is no dependency to View

- Cons
  - Quite much code should be added for new functionality
    - To display some other things, both ViewModel and View should be updated

Summary

- Design pattern
  - Needed for larger-scale, larger-team project

- MVC (Model View Controller)
  - Simple
  - Not well-fit for Android, too much dependencies
- MVP (Model View Presenter)
  - Better-fit for Android than MVC
  - Still hard to manage code complexity
- MVVM (Model View ViewModel)
  - With data binding, it is easy and flexible
  - Better to maintaining and unit testing
  - Coding well and maintaining well is hard

Lab – Week #9

- Make simple button to count application
  - Same as MVC, MVP

- You must use MVVM
- You must use data binding

- Compress to zip, and change file name
- Submit with name "<student_id>_w9.zip"

Lab – Week #9

- Enable data binding
  - build.gradle(module: app)

  – Do not forget to Sync Now

```
android {
    compileSdk 32

    defaultConfig {
        applicationId "edu.skku.cs.desig
        minSdk 26
        targetSdk 32
        versionCode 1
        versionName "1.0"

        testInstrumentationRunner "andro
    }

    dataBinding {
        enabled true
    }

    buildTypes {
        release {
            minifyEnabled false
            proguardFiles getDefaultProg
        }
    }
```

Lab – Week #9

- Model
  - MySimpleModel.java

```java
package edu.skku.cs.designpattern;

public class MySimpleModel{
    private int value;

    public MySimpleModel(int initialValue){
        this.value = initialValue;
    }

    public int getValue(){
        return this.value;
    }

    public void addOne() {
        this.value += 1;
    }
}
```

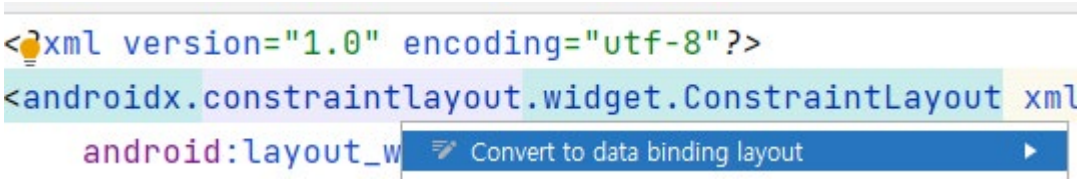
Lab – Week #9

- ViewModel
  - MySimpleViewModel.java
    - @Bindable
      - bindable on layout

```
import androidx.databinding.BaseObservable;
import androidx.databinding.Bindable;

public class MySimpleViewModel extends BaseObservable {
    private MySimpleModel model;

    public MySimpleViewModel(){
        model = new MySimpleModel( initialValue: 0);
        this.valueString = "Value: 0";
    }

    ////////////////////////////////////////////

    @Bindable
    private String valueString = null;

    public String getValueString(){
        return valueString;
    }

    public void setValueString(String valueString){
        this.valueString = valueString;
        notifyPropertyChanged(BR.valueString);
    }

    ////////////////////////////////////////////

    public void onAddingButtonClicked(){
        model.addOne();
        this.setValueString("Value: " + model.getValue());
    }
}
```

Lab – Week #9

- View
  - activity_main.xml
    - Alt+Enter on ConstraintLayout
      - Convert to data binding layout



    - Fill in data

```
<data>
    <variable
        name="mySimpleViewModel"
        type="edu.skku.cs.designpattern.MySimpleViewModel" />
</data>
```

Lab – Week #9

- View
  - activity_main.xml
    - Add button properties
      - @={field with getter/setter}
      - @{() -> function}

```
<Button
    android:id="@+id/button"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@={mySimpleViewModel.valueString}"
    android:onClick=
        "@{() -> mySimpleViewModel.onAddingButtonClicked()}"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent" />
```

Lab – Week #9

- View
  - activity_main.xml will be like

```xml
<?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"
    xmlns:bind="http://schemas.android.com/tools">

    <data>
        <variable
            name="mySimpleViewModel"
            type="edu.skku.cs.designpattern.MySimpleViewModel" />
    </data>

    <androidx.constraintlayout.widget.ConstraintLayout
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        tools:context=".MainActivity">

        <Button
            android:id="@+id/button"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="@={mySimpleViewModel.valueString}"
            android:onClick=
                "@{() -> mySimpleViewModel.onAddingButtonClicked()}"
            app:layout_constraintStart_toStartOf="parent"
            app:layout_constraintTop_toTopOf="parent" />
    </androidx.constraintlayout.widget.ConstraintLayout>
</layout>
```

Lab – Week #9

- View
  - MainActivity.java
    - Replace normal setContentView to databinding

```
public class MainActivity extends AppCompatActivity{

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        // comment out: setContentView(R.layout.activity_main);

        ActivityMainBinding activityMainBinding =
                DataBindingUtil.setContentView( activity: this, R.layout.activity_main);
        activityMainBinding.setMySimpleViewModel(new MySimpleViewModel());
        activityMainBinding.executePendingBindings();
    }

}
```

Lab – Week #9

- Summary
  - Implement Model: Just do simple
  - Implement ViewModel:
    - Before start, set dataBinding enabled (build.gradle)
    - Make model management and Bindable field(with getter and setter) and some methods(for button)
  - Implement View:
    - Make activity_main.xml to wrapped with data bind
      - Set data with ViewModel
      - Implement button with bind properties
    - Make MainActivity.java to use data binding

https://developer.android.com/codelabs/android-databinding?hl=ko