Session 2:
Design Patterns for Android

Karim Lekadir, PhD

Universitat de Barcelona
Dpt. Matemàtiques & Informàtica

Calendar

Week		Theory	Labs	Deliverables
1	16 February	Introduction to PIS	Introduction to Android	
2	23 February	Design pattern 1	Session 1	
3	2 March	Requirements	Session 2	
4	9 March	Examples requirements	Session 3	
5	16 March	Feedback deliverable 1	Feedback & support	
6	23 March	Design pattern 2	Feedback & support	1. Requirements + UI
7	30 March	Software testing	Feedback & support	
8	6 April	Week partials (not for PIS)	Feedback & support	
9	13 April	*Semana Santa*	Feedback & support	
10	20 April	Feedback deliverable 2	Feedback & support	
11	27 April	*Matefest-Infofest (no lectiu)*	Feedback & support	2. Design + demo
12	4 May	*Fira d'Empreses*	Feedback & support	
13	11 May	Feedback deliverable 3	Feedback & support	
14	18 May	Trial exam	Feedback & support	
15	25 May	Trial exam	Presentations	3. Final project

Design Patterns

1. General re-usable solution to a common problem in software design
2. They are not ready-to-code solutions (WHAT)
3. They provides templates and best practices for software design (HOW)
4. Increased flexibility and maintainability of the code
5. Proven solutions: Used by many !!

Design Patterns

Objects Structures Relationships & algorithms

Creational	Structural	Behavioral
• Factory Method	• Adapter	• Interperter
• Abstract Factory • Builder • Prototype • Singleton	• Adapter • Bridge • Composite • Decorator • Facade • Flyweight • Proxy	• Chain of Responsibility • Command • Iterator • Mediator • Momento • Observer • State • Strategy • Visitor

Design Principles

1. **S**ingle responsibility principle - Class has a single responsibility.
2. **O**pen/closed principle – Class should be open for extension, closed for modification.
3. **L**iskov substitution principle - Class can be replaced by any of its children.
4. **I**nterface segregation principle - Many client-specific interfaces are better than one general-purpose interface.
5. **D**ependency inversion principle - Depend upon abstractions, not concretions

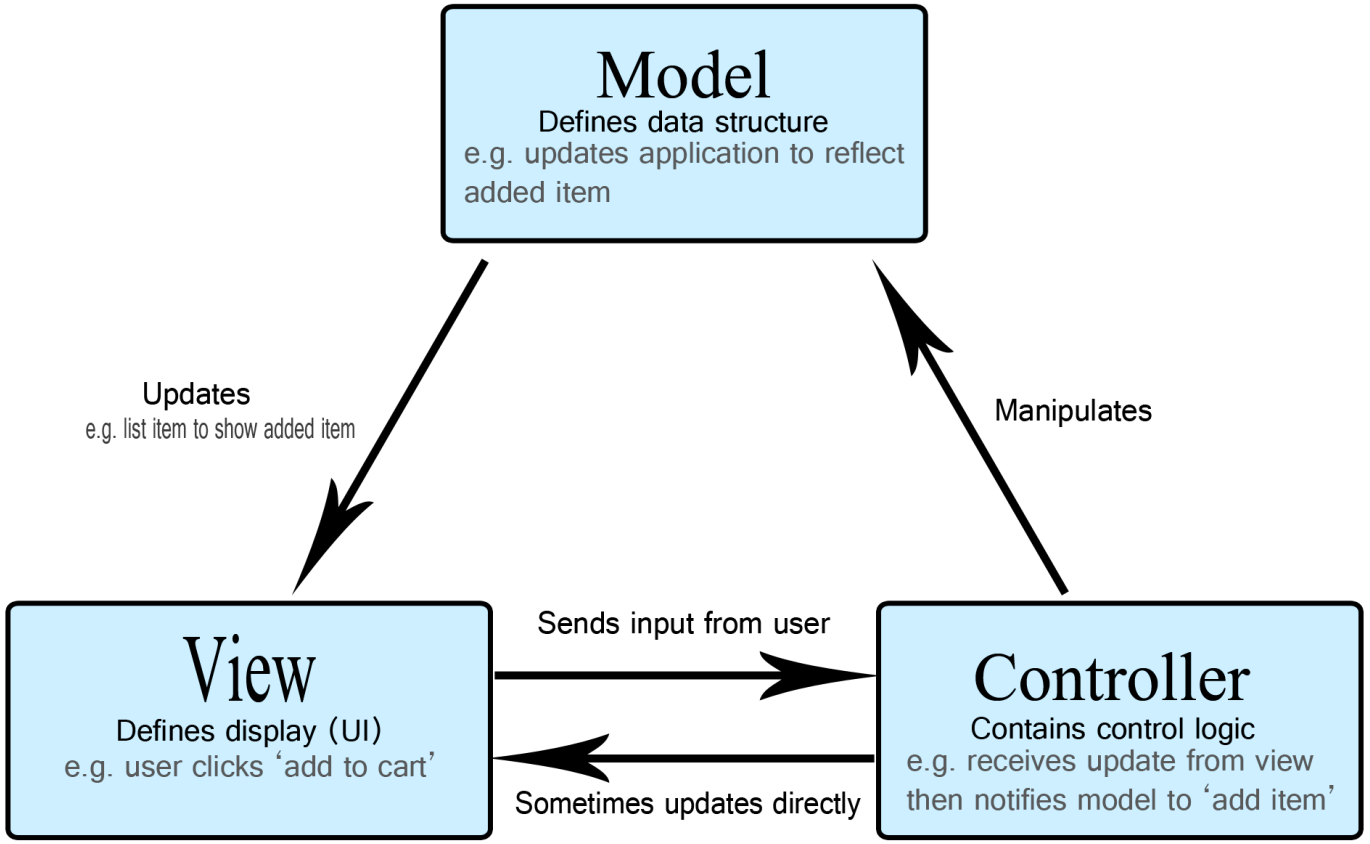
Content

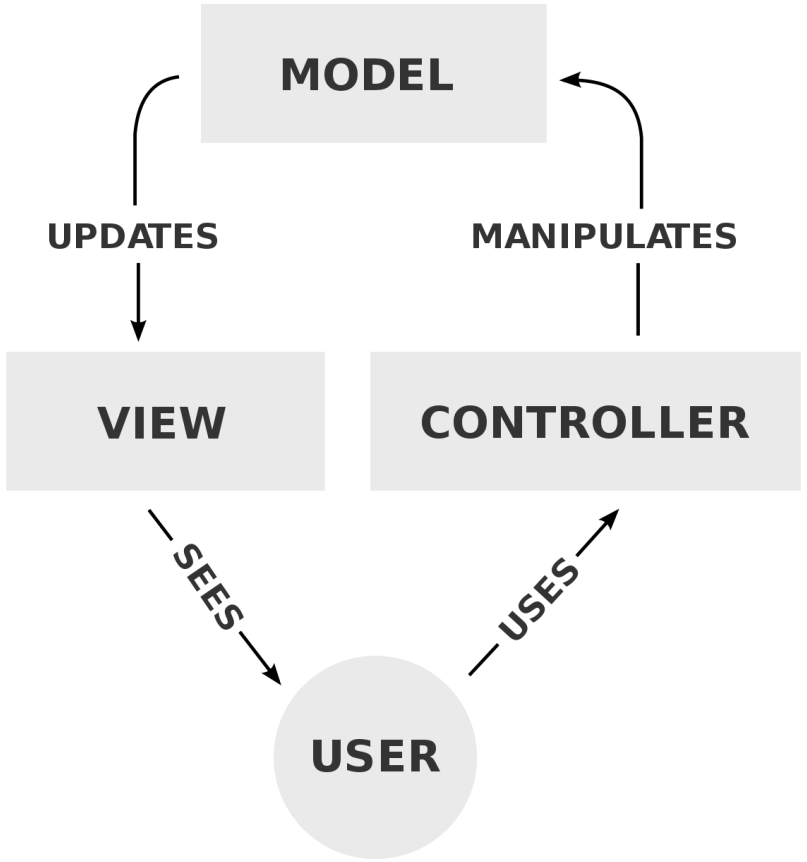
- Model–View–Controller (MVC)
- Model-View-ViewModel (MVVM)
- Example with the BMI App
- Questions / Answers

Model-View-Controller

- **Model**. Contains the data/information with which the system works. Provides the information to the user in **View**.
- **View**. Presents the user with the model's information.
- **Controller**. Responds to user actions, modifying the model when necessary. In addition, it communicates with the view to update with the latest model changes.

Model-View-Controller





Model-View-Controller

- In Android, there is no specific component to play the role of the **Controller**
- Communication between the **Model** and the **View** is done through the **Activity** or **Fragment**

➢ This violates the “single responsibility principle”, which means we have to run the entire **Activity** to test the logic.

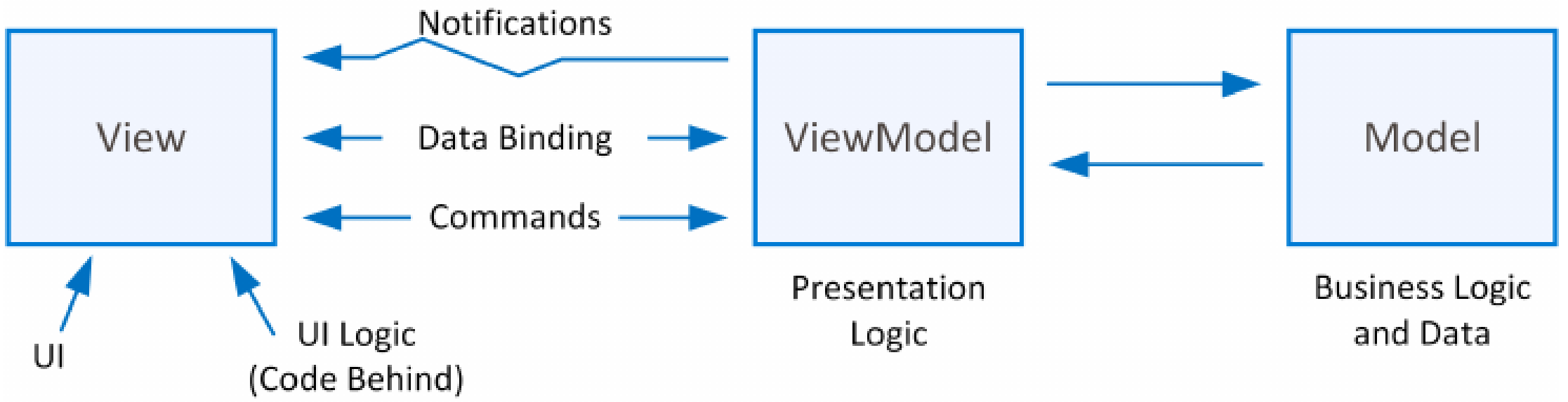
➢ Furthermore, these components are specific to Android. Implementing the logic in **Activity** creates dependencies to the eco-system.

Model-View-Viewmodel

- **Model**. Contains the data/information with which the system works. Provides the information to the user in **View**.
- **View**. Presents the user with the model's information.
- **Viewmodel**. Abstraction of the **View** exposing public properties and commands.

✓ The **View** directly binds to properties on the **Viewmodel** to send and receive updates. To function efficiently, this requires a *binding technology.*

Model-View-Viewmodel



1. View

- Activity, Fragment, Viewgroup, Layout, etc
- These elements have to be as simple as possible and fully independent of any logic

✓ They should be responsible of instantiating views (with getViewById)

✓ They transmit any response (e.g. from OnClick) to the Viewmodel and updates in the data based on **Observers**

1. View

MainActivity.java | activity_main.xml

```java
public class MainActivity extends Activity {

    // Instancias de objetos a usar
    private double valor_a, valor_b;
    //private double valor_bmi;
    private EditText op_a, op_b;
    private TextView resultado;

    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        // Asignamos los objetos
        this.op_a = (EditText) findViewById(R.id.op_a);
        this.op_b = (EditText) findViewById(R.id.op_b);
        this.resultado = (TextView) findViewById(R.id.resultado);
    }

    public void cSumar(View view) {
        if(this.op_a.getText().toString().length() > 0 && this.op_b.getText().toString().length() > 0) {
            this.valor_a = Double.parseDouble(this.op_a.getText().toString());
            this.valor_b = Double.parseDouble(this.op_b.getText().toString());
            this.resultado.setText(Double.toString((this.valor_a + this.valor_b)));
        }
    }
```

```java
public void BMI(View view) {
    if (this.op_a.getText().toString().length() > 0 && this.op_b.getText().toString().length()
        this.valor_a = Double.parseDouble(this.op_a.getText().toString());
        this.valor_b = Double.parseDouble(this.op_b.getText().toString());
        this.valor_bmi = ((this.valor_a * this.valor_a) / this.valor_b);
        if (this.valor_bmi < 25) {
            this.resultado.setText("Peso ideal");
        }
        if (this.valor_bmi >= 25) {
            this.resultado.setText("Sobrepeso");
        }
```

1. View


```
public class BMIActivity extends AppCompatActivity {

    private EditText op_a, op_b;
    private TextView resultado;
    private Button bmiButton;
    //declare our viewModel
    private BMIViewModel viewModel;

    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_bmi);
        initView();

    }
    private void initView(){
        // Instance the viewModel (update gradle dependencies if required)
        viewModel = new ViewModelProvider( owner: this).get(BMIViewModel.class);

        this.op_a = (EditText) findViewById(R.id.op_a);
        this.op_b = (EditText) findViewById(R.id.op_b);
        this.bmiButton = (Button) findViewById(R.id.bmi_buttom);
        this.resultado = (TextView) findViewById(R.id.resultado);

        bmiButton.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View view) {
                viewModel.BMI(op_a.getText().toString(),op_b.getText().toString());
            }
        });
        //Observer related stuff...
    }
```

1. The data/information (**Model**) is removed from the Activity
2. The logic is also removed from the Activity
3. The Activity has an instance of the **Viewmodel**
4. The data/information (**Model**) will be managed through the **Viewmodel**

2. Model

View

```
public class BMIActivity extends AppCompatActivity {

    private EditText op_a, op_b;
    private TextView resultado;
    private Button bmiButton;
    //declare our viewModel
    private BMIViewModel viewModel;

    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_bmi);
        initView();

    }
    private void initView(){
        // Instance the viewModel (update gradle dependencies if required)
        viewModel = new ViewModelProvider( owner: this).get(BMIViewModel.class);

        this.op_a = (EditText) findViewById(R.id.op_a);
        this.op_b = (EditText) findViewById(R.id.op_b);
        this.bmiButton = (Button) findViewById(R.id.bmi_buttom);
        this.resultado = (TextView) findViewById(R.id.resultado);

        bmiButton.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View view) {
                viewModel.BMI(op_a.getText().toString(),op_b.getText().toString());
            }
        });
        //Observer related stuff...
    }
```

Instance of Viewmodel

Model

```
package com.example.bmi_mvvm;

public class BMIDummyModel {

    public static String BMI(String op1, String op2) {
        double valor_a = Double.parseDouble(op1);
        double valor_b = Double.parseDouble(op2);
        double valor_bmi = ((valor_a * valor_a) / valor_b);
        if (valor_bmi < 25) {
            return "Peso ideal";
        }
        else return "Sobrepeso";
```

3. Viewmodel

BMIActivity.java | BMIDummyModel.java | BMIViewModel.java | activity_bmi.xml

```java
package com.example.bmi_mvvm;

import androidx.lifecycle.LiveData;
import androidx.lifecycle.MutableLiveData;
import androidx.lifecycle.ViewModel;

public class BMIViewModel extends ViewModel {
    //Mutable , allows assignation of a new result
    private MutableLiveData<String> resultado;

    //Constructor
    public BMIViewModel(){
        resultado = new MutableLiveData<>();
    }

    //public getter. Not mutable , read-only
    public LiveData<String> getResultado(){
        return resultado;
    }

    //comunicates user inputs and update the result in the viewModel
    public void BMI(String peso, String altura){
        resultado.setValue(BMIDummyModel.BMI(altura,peso));
    }
}
```

LiveData

- LiveData is a container of observable data
- Optimised for the life cycles of all components of the App
- For example, change in configurations (e.g. rotation): The Activity receives immediately the available data
- **MutableLiveData** is a subclass of LiveData. Using its setValue and postValue properties we can notify the UI when **onChange** is called.

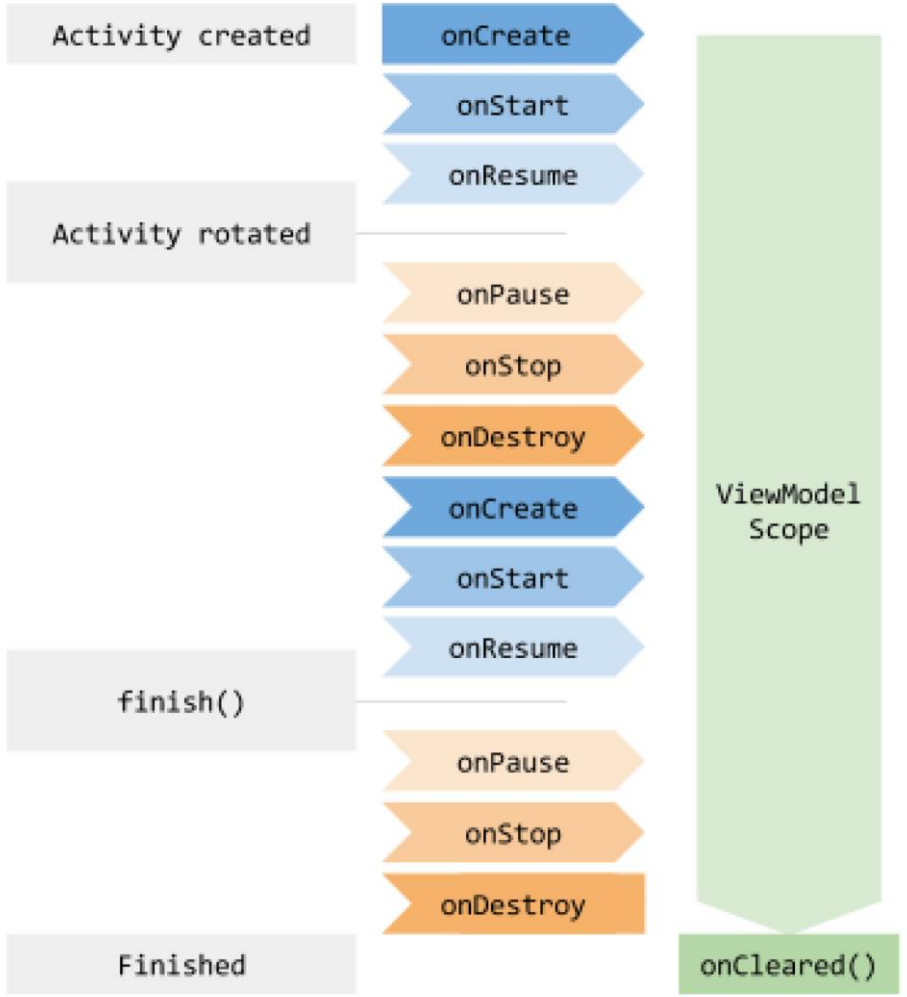
LiveData

```
    //Observer related stuff...
    //Observe changes in LiveData
    final Observer<String> observer = new Observer<String>() {
        @Override
        public void onChanged(String s) {
            resultado.setText(s);

        }
    };

    //Subscribe the activity to the observable
    viewModel.getResultado().observe( owner: this,observer);


}
```



Summary

- **View:** Activity or fragment, very simple
- **Model:** Separate class with all the information/data
- **Viewmodel**: Contains a copy of the **Model** based on **MutableLiveData**
- **View** has an instance of the **Viewmodel**
- Updates in the data are sent to **Viewmodel** using the **Observer** Activity

Advantages

1. **Modelview** is separate from the **View**
2. Thus, it is easier to test the logic of the program separately (no need to run the entire App/Activity)
3. **Viewmodel** resists any changes in the configuration of the **View** (e.g. rotation)
4. **View** only takes care of receiving information from the **Viewmodel** and update its elements/visualisations, thus reducing complexity

Deliverable

1. Design:
   - ✓ Model overview (structures, classes, relationships, etc)
   - ✓ Viewmodel overview
   - ✓ View overview
2. Demo:
   1. Description of the interfaces and functionalities (screenshots + text)
   2. Source code / project (ZIP)
   3. APK file

Question 1

What are design patterns?

A. Templates for object classes

B. Templates for architecture design

C. Common solutions to solve software design problems

D. Common solutions to transform requirements into software design

Question 2

In the Model-View-Controller pattern:

A. The Model and View components are not connected

B. The Model, View and Controller are connected

C. The Model and View are separated by the Controller

D. The View can send information to the Model

Question 3

In the Model-View-Viewmodel pattern:

A. The Model and View components are not connected

B. The Model, View and Viewmodel are connected

C. The View and Viewmodel are connected

D. The View cannot send commands to the Viewmodel

Question 4

Which of the SOLID principles of software design the Model-View-Viewmodel specifically achieves?

A. S: Separation of concern

B. O: Classes open for extension

C. I: Interfaces are segregated

D. All SOLID principles

Question 5

The View contains:

A. An instance of the Model

B. An instance of the Viewmodel

C. An instance of the Model and Viewmodel

D. An instance of the Viewmodel and Observer

Question 6

The Model can contain:

A. Android objects

B. An instance of the Viewmodel for communication

C. Only Java code

D. An instance of the View to show any updates on the interfaces

Question 7

The Viewmodel can contain:

A. An instance of the View for updating the interfaces

B. The MutableLiveData class for communication with the Model

C. Only Java code

D. The Observer class

Question 8

What is the role of the Observer in this design?

A. Observes changes in LiveData

B. Observes changes in the View

C. Observes changes in the Viewmodel

D. Observes the communication between the Viewmodel and Model