DA: Design Patterns

Applications for mobile devices - Theory - Unit 4

Didac Florensa Cazorla

Any: 2021-2022
Curs: 102386
Institut: University of Lleida (Campus Igualada)
Titulació: Bachelor's degree in Digital Interaction and Computing Techniques (GTIDIC)

Agenda

Introduction

Design Patterns

Observer Pattern

Model View Controller (MVC)

Model View Presenter (MVP)

Code Example

Homework

Introduction

Warm-up

- Dr. Jordi Mateo Fornés
- **Office**:
  - Office A.12 (Campus Igualada)
  - Office 3.08 (EPS Lleida)
- **Email**: jordi.mateo@udl.cat
- **Twitter**: https://twitter.com/MatForJordi
- **Github**: https://github/JordiMateoUdL

- Ph.D Dídac Florensa Cazorla
- **Office**:
  - Office A.12 (Campus Igualada)
  - Office 3.08 (EPS Lleida)
- **Email**: didac.florensa@udl.cat

Design Patterns

Getting started

> *Make it easy to reuse, If it is easy to reuse, people will. Create an environment that supports reuse. A pragmatic programmer, David Thomas and Andrew Hunt.*

Definition

Design patterns repsesent a solutions for recurring problems. They can be build blocks of a system's architecture, recurring solutions to design problems, even abstractions.

- **Creational patterns**: How you create objects
- **Structural patterns**: How you compose objects
- **Behavioral patterns**: How you coordinate object interactions.

They represent challenges, other developers already faced and prevent you from reinventing the wheel by showing you proven ways to solve those problems.

Which are the most used?

Design patterns should not be applied indiscriminately. Often they achieve flexibility and variability by introducing additional levels of indirection, and that can complicate a design and/or cost you some performance. A design pattern should only be applied when the flexibility it affords is actually needed, Erich Gamma, Design Patterns: Elements of Reusable Object-Oriented Software.

Creational patterns

- *Builder*
- Dependency Injection
- *Singleton*
- Factory

Structural patterns

- *Adapter*
- Facade
- Decorator
- Composite

Behavioral patterns

- Command
- **Observer**
- Strategy
- State

Observer Pattern

What is the Observer Pattern?

The observer is a behavioural design pattern. It specifies the communication between objects: observable and observers. An **observable** is an object which **notifies observers** about the **changes** in its **state**. It can be seen as a Many to Many dependencies.

Actors involved

- **Subject**: Defines the interface to register and unregister one or more observers.
- **Observer**: Defines the interface to the objects that need notification if *subject* status changes.
- **ConcreteSubject**: Stores the current status for **ConcreteObservers**. Sends a notification to **Oberservers** when its state change. Implements **Subject**.
- **ConcreteObserver**: Stores a reference to a ConcreteSubject, storing a consistent status with ConcreteSubject. Implements **Observer**.

Pattern diagram

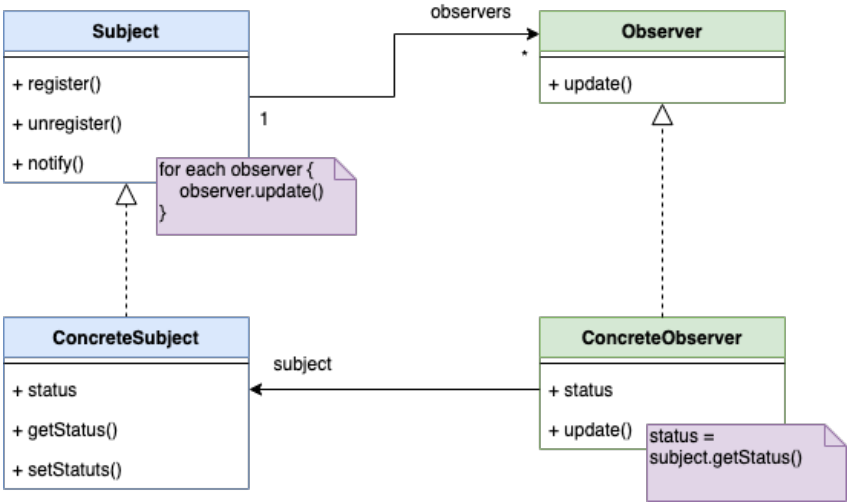

Figure 1: Schema of the obsever pattern

Where we need to use

- When a change in an object affects others, we don't know how many need to be changed.
- When an object needs to notify others without knowing the other object. The object will be **decoupled**.
- When we have layers where data can change in logic, the view shows the new data.

Why use

- We can modify subjects and observers independently.
- We can reuse subjects without observers and observers without subjects.

Model View Controller (MVC)

What is the pattern MVC?

MVC design pattern splits an application into three main aspects: *Model*, *View* and *Controller*. It forces a *separation* of concerns. The domain model and controller logic are *decoupled* from the user interface (view).

- **Model**: Represents a set of classes that describe the *business logic*, *business rules*, *data access operations* or *data model*.
- **View**: Represents the *UI components*. It is only responsible for *displaying the data.
- **Controller**: Responsible to *process incoming requests*.

Architecture

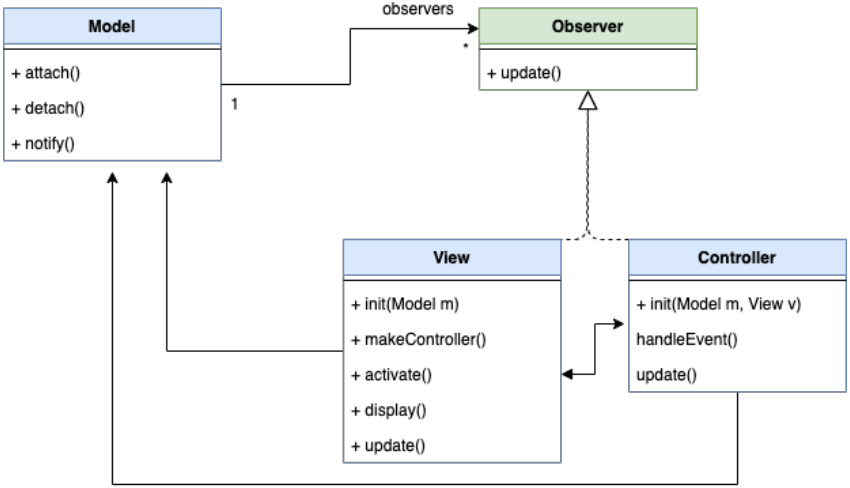

Figure 2: Schema of the Model View Controller

Processing Events

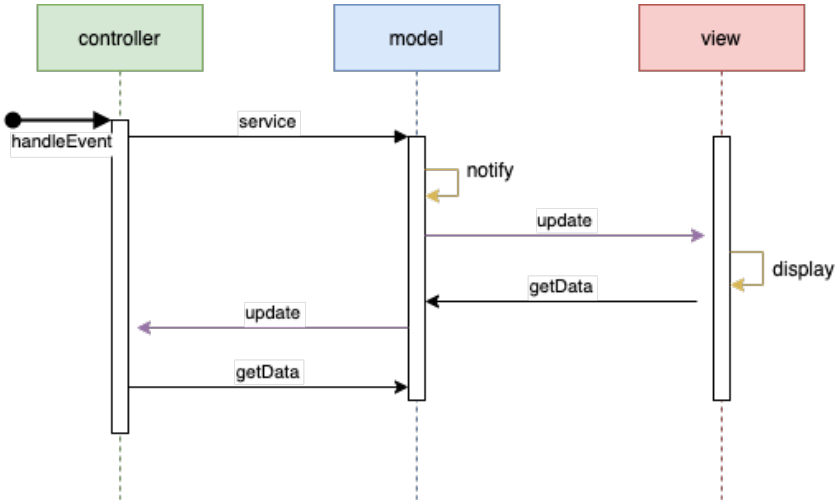

Figure 3: Sequence Diagram for processing events (MVC)

Init MVC

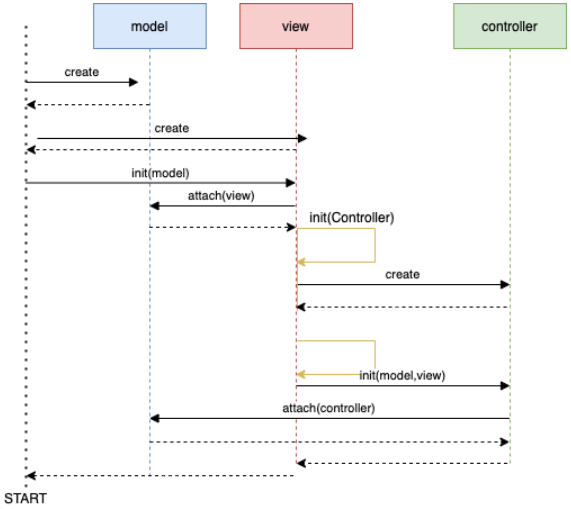

Figure 4: Sequence Diagram for starting (MVC)

MVC in Android

Android framework provides widgets that are capable of handling user inputs. The role of the Controller is gone.

Approach 1

- **Activities, Fragments, Views** represent the View in the MVC.
- **Controller and Model** should be separate classes -> they do not extend or use any Android class.
- **View and Controller** depend on each other. The View should implement an Interface, and the Controller should reference that and not the concrete class directly.

Approach 2

- **Controller and View** will be handled by the Activity or Fragment. It will act as an **observer**.
- **Model** will be a separate class that contains the data to be displayed. The operations on the data will be performed by functions of this **observable class** that will notify about changes.

Model View Presenter (MVP)

Model View Presenter (MVP)

Similar to MVC but **breaking** the **dependency** on *what* we have on **view**.

- **Model**: Represents a set of classes that describe the *business logic*, *business rules*, *data access operations* or *data model*.
- **View**: Represents the *UI components*: **layouts (xml)**, fragments or activities.
- **Presenter**: It *presents* the data from the model. Stores the data back to the model. Control all the behaviour managing the view. Any *interaction* between the model and the view is *handled* by the presenter.

Presenter

- Defines an interface in presenter class, to which it passes the required data. Activities and fragments implement this interface to render the data.
- The presenter manipulates the model and updates the view. The View and the model are entirely decoupled. Thus mocking the view facilitates unit testing.
- Has no relation to the views (Unlike MVC).
- The views invoke operations.
- Views are updated via interfaces.

Implementation

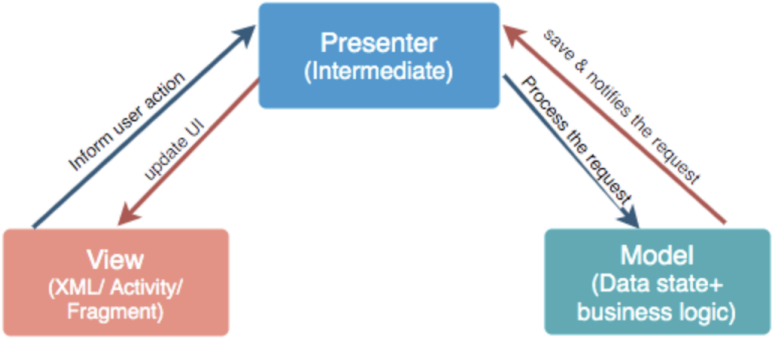

Figure 5: Schema of the Model View Presenter

Code Example

Code Example

DEMO: AgeApp

All the source code can be found in this github repo DAM-MVC-MVP.

Master branch

- Model-View-Controller using observer pattern. (Approach 1)

MVP branch

- Model-View-Presenter.

MVC (Approach 2)

- Not included in the repo (only in slides). *Happy Coding!!!!*

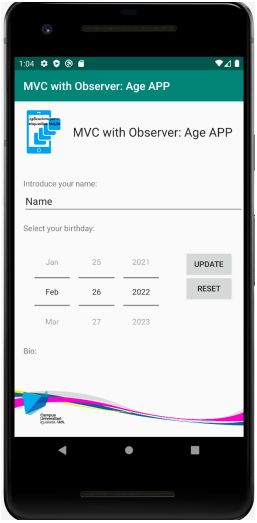

Figure 6: Screenshot of the demo app

AgeApp (Simplified Layout)

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout >
    <TextView
        android:id="@+id/textView_label_birthday"
        android:text="@string/introduce_your_name"
        app:layout_constraintTop_toBottomOf="@+id/editText_fullName" />
    <Button
        android:onClick="updateButtonOnClick"
        android:text="@string/update"/>
    <TextView
      android:id="@+id/textView_bio"/>
<androidx.constraintlayout.widget.ConstraintLayout>
```

AgeApp: UserModel

```
public class UserModel extends Observable {
    private String TAG = "UserModel";
    private String fullName;
    public void setFullName(String fullName) {
        Log.d(TAG,"Set fullName to the model");
        this.fullName = fullName;
        setChanged();
        Log.d(TAG,"Model is updated, let's notify observers");
        notifyObservers();
    }
    @NonNull @Override
    public String toString(){
        if (this.fullName.contentEquals("")){ return ""; } else {
            return "Hola, el meu nom és " + this.fullName +  ";
        }
    }
}
```

AgeApp: Activity (Approach 2)

```
public class MainActivity extends AppCompatActivity implements Observer {
    private EditText fullNameEditText;
    private TextView bioTextView;
    private UserModel model = new UserModel();

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        fullNameEditText = findViewById(R.id.editText_fullName);
        bioTextView = findViewById(R.id.textView_bio);
    }
```

AgeApp: Activity (Approach 2)

```java
public class MainActivity extends AppCompatActivity implements Observer {

    @Override
    public void onStart(){
        super.onStart();
        model.addObserver(this);
    }

    @Override
    public void onStop(){
        super.onStop();
        model.deleteObserver(this);
    }
```

AgeApp: Activity (Approach 2)

```
public class MainActivity extends AppCompatActivity implements Observer {

  @Override
  public void update(Observable o, Object arg) {
      String TAG = "MVCLog";
      Log.d(TAG,"Observer notifies changes -> update layout");
      bioTextView.setText(model.toString());
  }

  public String getUsername() {
        return fullNameEditText.getText().toString();
  }

  public void updateButtonOnClick(View v){
        model.setFullName(fullNameEditText.getUsername());
  }
```

Homework

Task A: Review the DEMO repository

- Ensure you understand the MVC.
- Ensure you understand the MVP.
- Design but not implement how this affects your current code.
- Ask all the questions needed to understand how it works.

This task is important as the base for understanding MVVM, which is the natural pattern that we suggest and recommend to use (Google and the teachers). Thus, this learning is crucial to move one to the GOD level (MVVM and data binding).

Task B: Project

Time to code and make progress for the Sprint II.

That's all

QUESTIONS?

About me

www — jordimateofornes.com
github — github.com/JordiMateo
twitter — @MatForJordi
gdc — Distributed computation group