Laboratorio di Applicazioni Mobili
Bachelor in Computer Science &
Computer Science for Management

University of Bologna

Architectural Components

Federico Montori
federico.montori2@unibo.it

Table of Contents

Architectural Design Patterns

In time, the development in Android has changed quickly

- Lack of architectural design patterns
- Different native languages
- Hybrid technologies
- Handling bindings between views and controllers is tedious
- A lot of boilerplate code…
- Adapting legacy stuff instead of redesigning from scratch

Architectural Design Patterns

Furthermore, well, you're on a smartphone, which means a lot more hassle:

- For example, you share a photo in your favorite social networking app
  - The app triggers a camera intent. The Android OS then launches a camera app to handle the request. So you leave the first app...
  - The camera app might trigger other intents, like launching the file chooser, which may launch yet another app.
  - Eventually, the user returns to the social networking app and shares the photo.
  - At any point, the user could be interrupted by a phone call or notification. After acting this, the user should resume the photo sharing process...
  - Keep in mind that the OS might kill some processes when needed

Architectural Design Patterns

Given such condition, we need a solid architectural decoupling that ensures component are not depending on each other.

- Model-View-Controller (MVC)
- Model-View-Presenter (MVP)
- Model-View-ViewModel (MVVM)

App Complexity
Decoupling

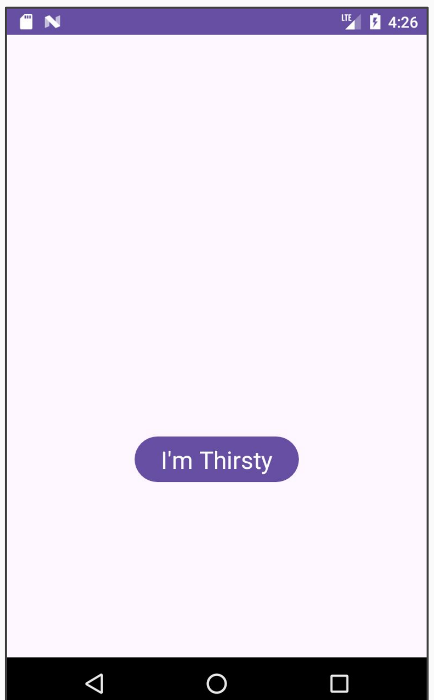
Architectural Design Patterns

- **Model**
  - The application data, with no knowledge about its interface. Handles the domain logic, with connection to databases and network layers.
- **View**
  - The UI, providing the means for visualizing the data in the model and to handle the interactions with the user.

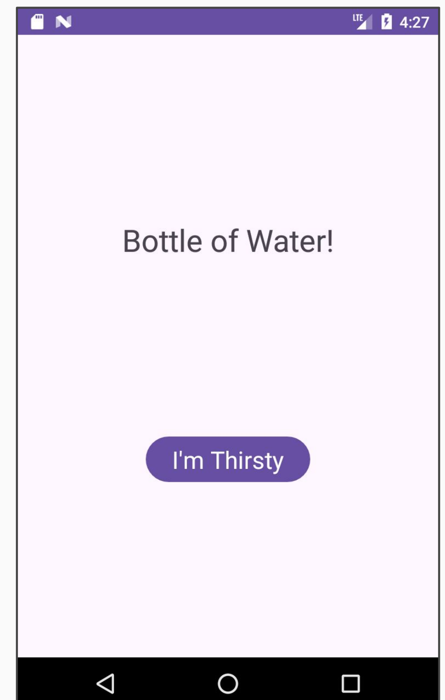
What about the business logic?

Architectural Design Patterns

We will see the differences hands-on with a tiny sample app.



This app simply simulates someone who wants a bottle of water. The **activity_main.xml** contains just a Button and a TextView. When the button is pressed, the TextView is populated.

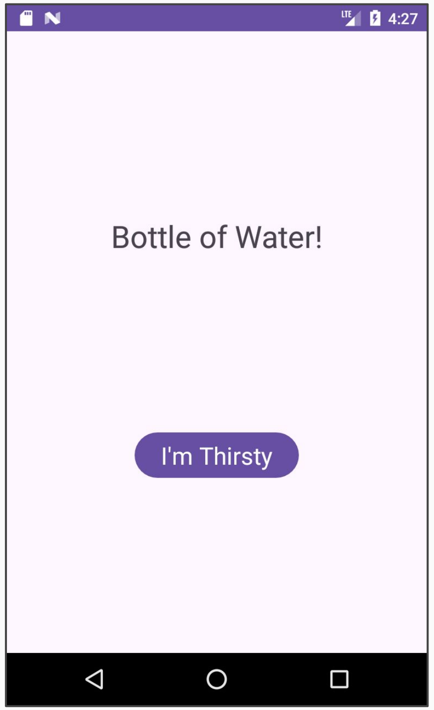


Credits go to Master Coding https://youtu.be/v4PbYeweaO4?si=gu6yWOw0hPDfP2t2

Architectural Design Patterns

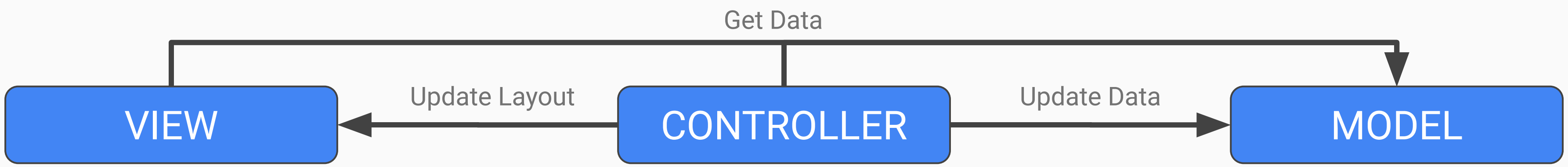
We are going to work with a very simple Model class, which simulates our DB and some placeholder domain logic.

```
class Model {
    private val data: String = "Water"

    fun getData(): String {
        return "Bottle of ${data}!"
    }
}
```

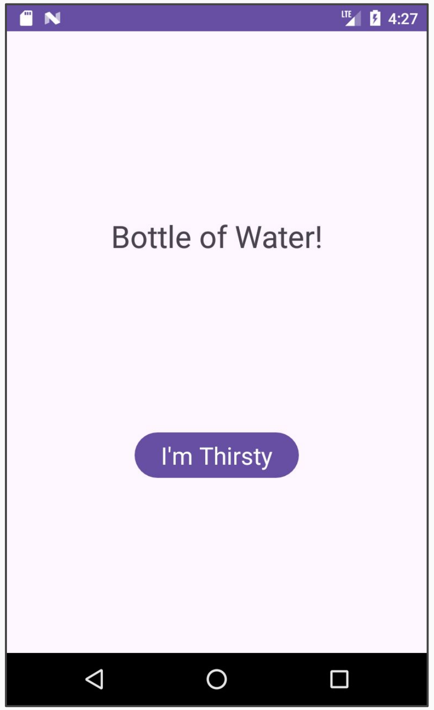


Model-View-Controller

"**<u>You</u>** go to a shop, **<u>you</u>** take the water from the fridge and the **<u>you</u>** go and pay for it."



- Controller is the **<u>active part</u>** containing the business logic.
- Model and View are almost passive.
- Easy to test the Model because it has no dependence.
  - Controller and View are heavily tied.
- Old pattern, suitable for small projects



Model-View-Controller

```
class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        val myModel: Model = Model()
        val textView = findViewById<TextView>(R.id.textView)
        val button = findViewById<Button>(R.id.button)
        button.setOnClickListener {
            textView.text = myModel.getData()
        }
    }
}
```

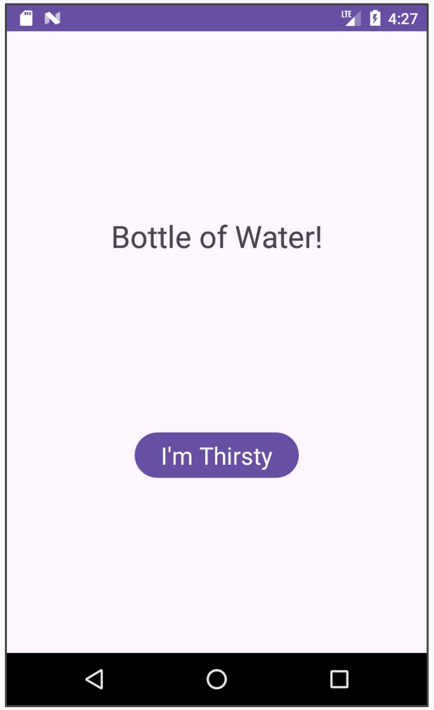
View: XML Layout file
Controller: Activity

Here, the coupling between View and Controller is very evident as they are almost the same thing.

Model-View-Presenter

"**You** go to a bar and ask for a bottle of water, then **the waiter** brings you the water."

- View is the **active part** containing the business logic.
- Presenter acts as a mediator, 1-to-1 with the View
- Easy to test the Model because of no dependence.
- Kind of easy to test the Presenter+Model if using an interface.
- Suitable for medium-sized projects



Model-View-Presenter

```
class Presenter (val appView: AppView) {
    private lateinit var model: Model
    private fun getModel(): Model {
        if (! ::model.isInitialized)
            model = Model()
        return model // Singleton pattern
    }

    fun getDataFromModel() {
        /* Call the interface and not the View */
        appView.onGetData(getModel().getData())
    }
}
```

```
interface AppView {
    fun onGetData(data: String) {
        /* By default does nothing */
    }
}
```

If using an interface, the Presenter just assumes that there is a View implementing it, with no clue about who it is. If not, then it is again a hard coupling.

Model-View-Presenter

```
class MainActivity : AppCompatActivity(), AppView {
   override fun onCreate(savedInstanceState: Bundle?) {
      super.onCreate(savedInstanceState)
      setContentView(R.layout.activity_main)

      val myPresenter: Presenter = Presenter(this)
      val button = findViewById<Button>(R.id.button)
      button.setOnClickListener { myPresenter.getDataFromModel() }
   }

   override fun onGetData(data: String) {
      val textView = findViewById<TextView>(R.id.textView)
      textView.text = data
   }
}
```

View: Activity
Presenter: Presenter

Here, the coupling between View and Presenter happens in one direction, even though one presenter needs exactly one View (because of its constructor).

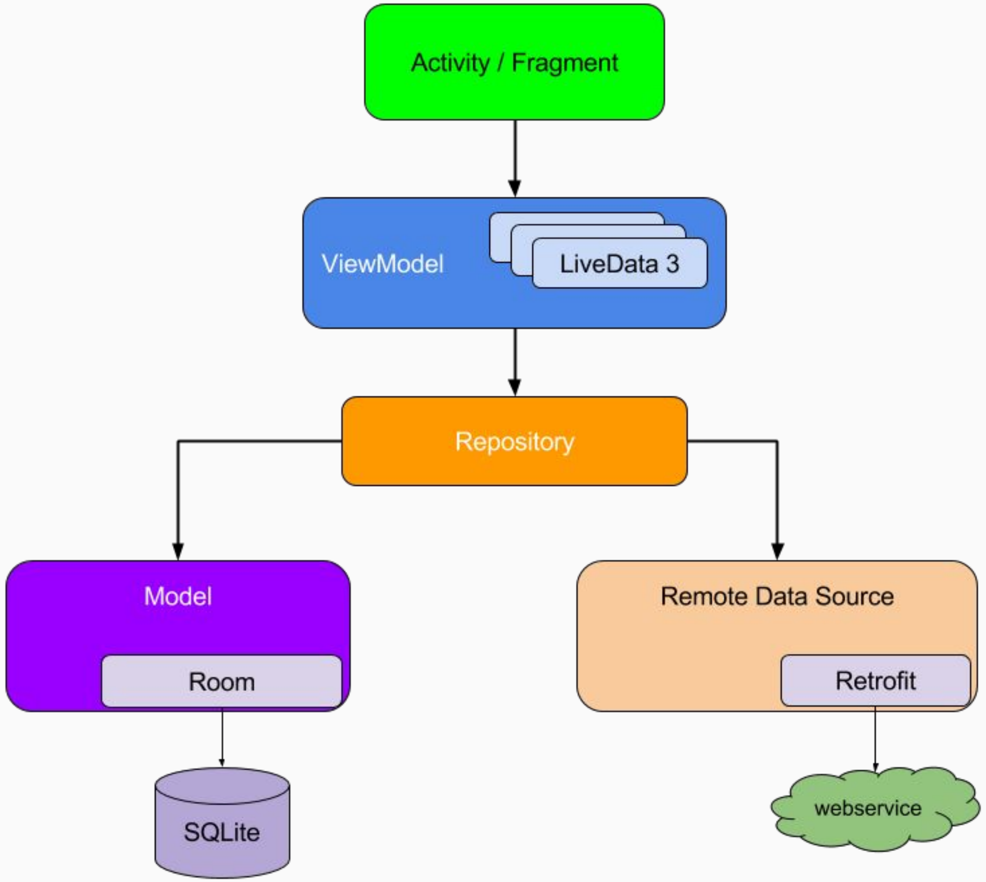
Model-View-ViewModel

MVVM is the recommended architectural design pattern for Android apps and it is industry-recognized.

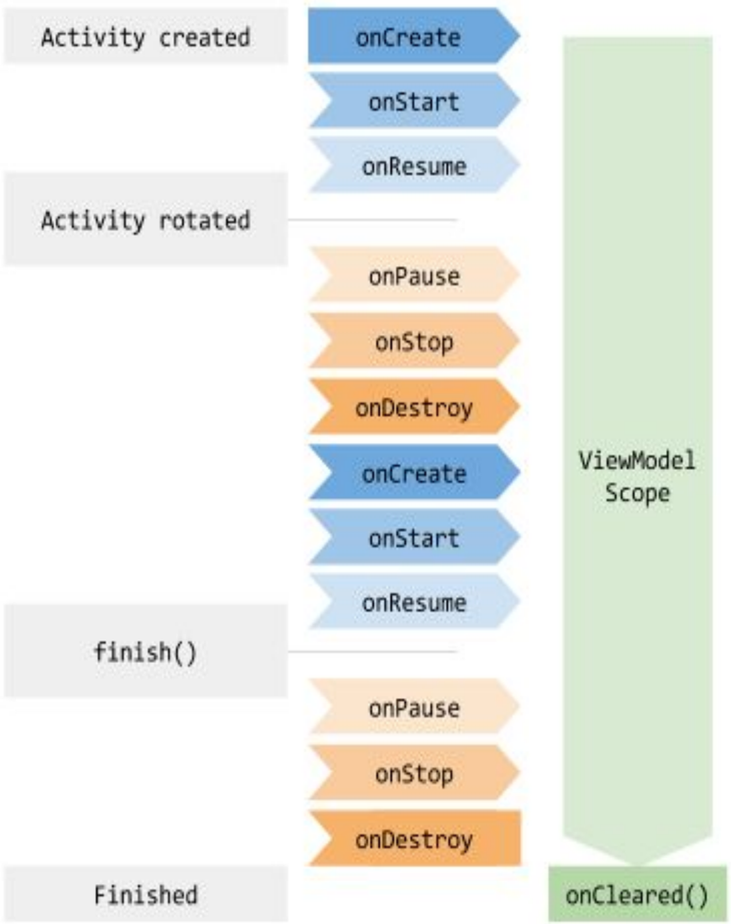
Before delving into its architectural details we will need to learn two primitives:

- ViewModel
- Observables and LiveData

Today our model will just be the "Repository" leaving the rest to the future…



ViewModel



A **ViewModel** is a component that stores UI-related data in a Lifecycle-aware way.

- It helps surviving seamlessly configuration changes
- If the activity is destroyed and re-created there is no need for saving instance state every time (which is instead suitable only for small data).
- Separates view data ownership (ViewModel) from UI controller logic (View).
  - One ViewModel per UI controller
  - Multiple UI controller per ViewModel

ViewModel

```
class MyViewModel: ViewModel() {
   private lateinit var value : String
   fun getValue(): String {
      /* Do stuff to retrieve the value */
      return value
   }
}
```

First extend the ViewModel helper class..

- A ViewModel is scoped to the lifecycle of the object passed to the ViewModelProvider (this request makes it sort of singleton).
- A ViewModel never references elements of the View, the reference should be one-way only.

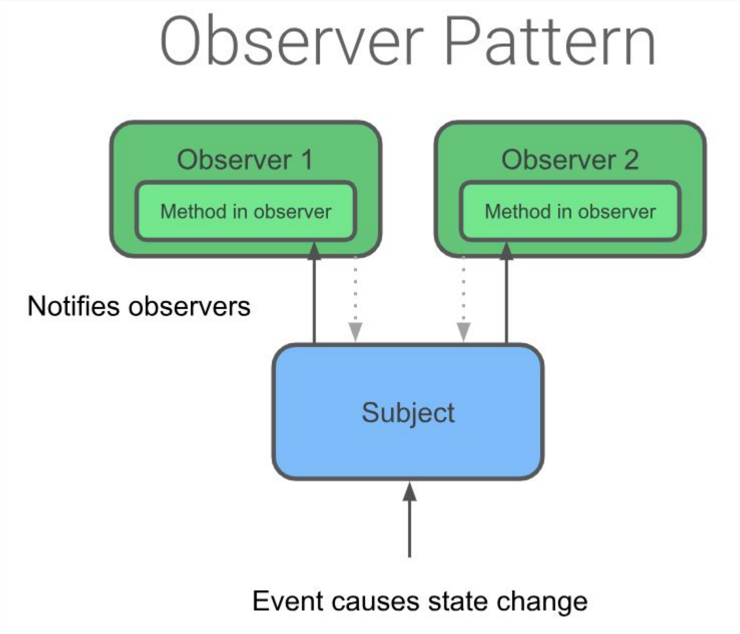
```
val myViewModel: MyViewModel =
      ViewModelProvider(this).get(MyViewModel::class.java)
textView.text = myViewModel.getValue()
```

LiveData

LiveData are based on the concept of **Observables**

- Observables are data classes that notify when changes on the observed data occur.
- they wrap existing data types

```
val firstName =
    ObservableField<String>()
val age =
    ObservableInt()
val items =
    ObservableArrayList<String>()
```



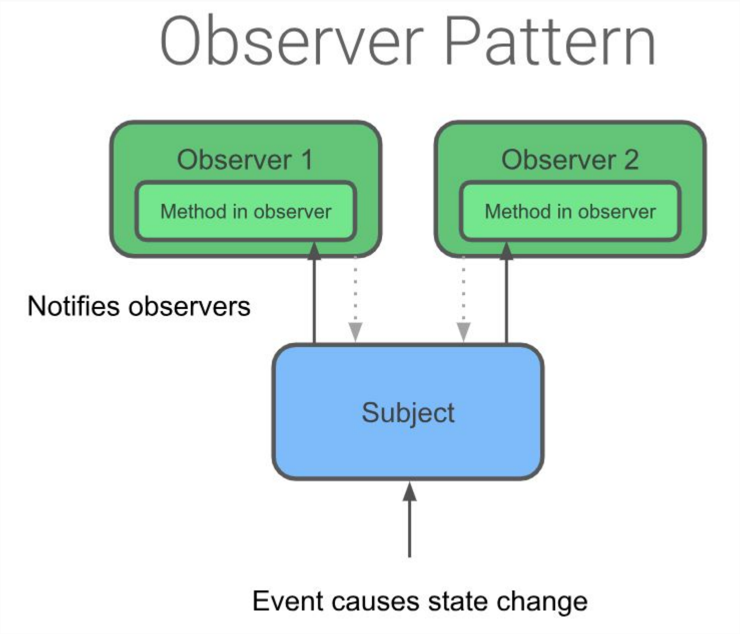
LiveData

The Observable object has a hidden list.

- Everytime another object subscribes to changes it is added to the list.
- Everytime a changes occur in the observed data field, all members of the list are notified by calling their callback function.

LiveData are Observables based on the concept of LifeCycle Awareness

- Let's leave observables for a second and see what these are…



LiveData

You can implement LifeCycle awareness by implementing an **Observer** to the **LifeCycle**:

Useful when the component needs to react to lifecycle changes.

```
class MyObserver : DefaultLifecycleObserver {
    override fun onResume
        (owner: LifecycleOwner) {
            /* Do stuff onResume */
    }

    override fun onPause
        (owner: LifecycleOwner) {
            /* Do stuff onPause */
    }
}

myLifecycleOwner.getLifecycle().
    addObserver(MyObserver())
```

LiveData

The function getLifecycle() can be called by a **LifeCycleOwner**
an object implementing the LifeCycleOwner interface, i.e. it has a Lifecycle (Activities, Services, Fragments...)

You can use powerful calls such as
lifecycle.getCurrentState().isAtLeast(STARTED)

You can create a class that implements the LifeCycleOwner interface

LiveData

LiveData are *lifecycle-aware observable components* that only notify subscribers that are in active state (i.e. RESUMED or STARTED).

- Useful for activities and fragments because they can observe data and not worry about their state.
- First of all, design your LiveData to contain the actual data (just like the observer, it is a wrapper).
- **MutableLiveData** can change (it has a setter), **LiveData** cannot
- Instantiate them in your **ViewModel**

```
val name: MutableLiveData<String> = MutableLiveData()
```

LiveData

LiveData are typically instantiated in your ViewModel, which means that the observer is located elsewhere (i.e. the Activity). It is typically good practice to return an immutable or a mutable LiveData to the class that observes:

```
class MyViewModel() : ViewModel() {

    /* Instantiated only the first time it is called */
    private val name: MutableLiveData<String> by lazy {  MutableLiveData<String>() }

    /* Returning an immutable reference to the observer */
    fun getNameImmutable(): LiveData<String> { return name  }
}
```

LiveData

Observer side:

You may want to start observe your LiveData in the Activity onCreate().
LiveData delivers updates to active observers when data changes.

```
val myViewModel: MyViewModel =
    ViewModelProvider(this).get(MyViewModel::class.java)

myViewModel.getNameImmutable().observe(this, Observer { newValue ->
        textView.text = newValue
    })
```

LifeCycleOwner

Observer is a SAM interface with the method **onChanged()**, called every time **name** changes and as soon as **observe** is called if there is a value already.

LiveData

LiveData values are updated by using:

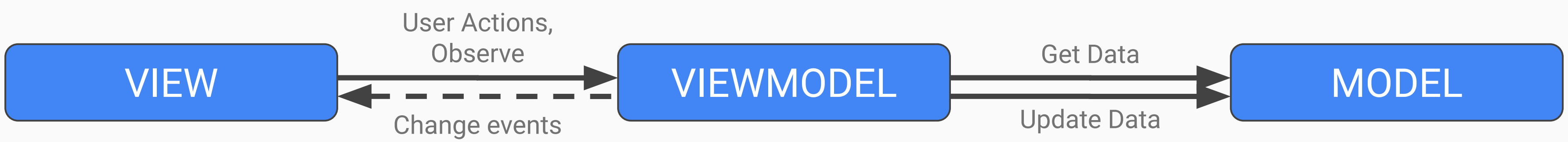
- **setValue**() if called from the main thread
- **postValue**() if called from a worker thread

```
name.postValue("John Doe")
```

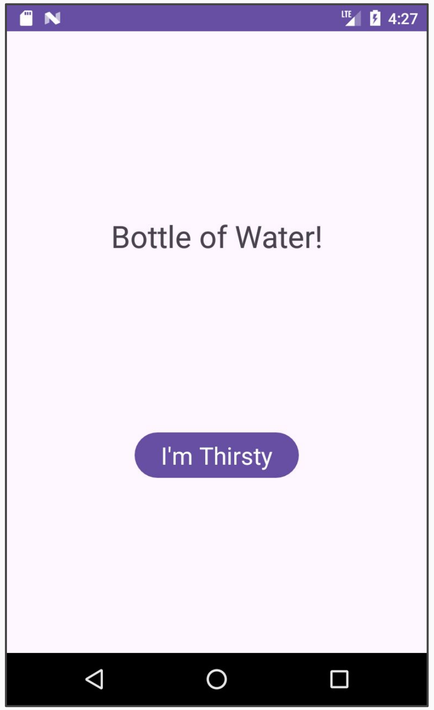
- Remember that setValue() and postValue() are only callable against a MutableLiveData.
- If you want to pass LiveData to a class not in charge of modifying it, then only pass LiveData type.
- Typically ViewModel updates LiveData, Activity only observes
  - or calls a method in the ViewModel to update the LiveData

Model-View-ViewModel

"**You** go to a vending machine and pay for a bottle of water, then **the machine** gives back the water."



- View is the **active part** containing the business logic.
- ViewModel contains LiveData, 1-to-many with the View
- Easy to test the Model because of no dependence.
- Easy to test the ViewModel because of no dependence.
- Suitable for large-sized projects



Model-View-ViewModel

```
class MyViewModel() : ViewModel() {

    private val model: Model by lazy { Model() }  // Singleton
    private val _liveValue: MutableLiveData<String> by lazy
        { MutableLiveData<String>() }
    /* Casts the private MutableLiveData with an immutable one */
    val liveValue: LiveData<String>
        get() = _liveValue

    fun getDataFromModel() {
        /* Set the LiveData value, no matter who is observing */
        _liveValue.value = (model.getData())
    }
}
```

View: Activity
ViewModel: ViewModel

The ViewModel does not even know if there is a View, or if there are multiple ones.

It just keeps updated its LiveData, no matter who is observing.

Model-View-ViewModel

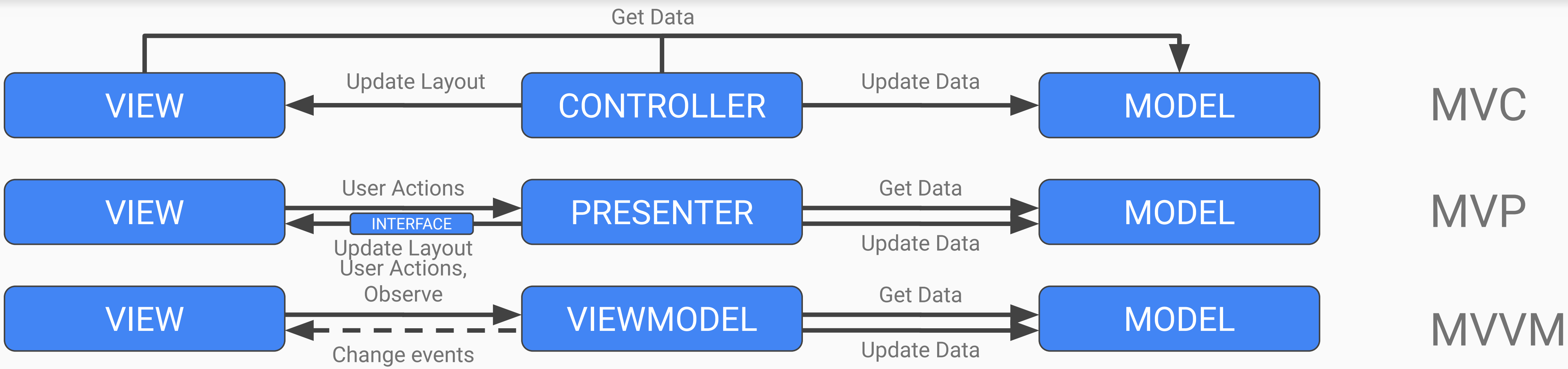
```
class MainActivity : AppCompatActivity(){

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        val textView = findViewById<TextView>(R.id.textView)
        val myViewModel: MyViewModel =
            ViewModelProvider(this)[MyViewModel::class.java]
        myViewModel.liveValue.observe(this)
            { newValue -> textView.text = newValue }
        val button = findViewById<Button>(R.id.button)
        button.setOnClickListener
            { myViewModel.getDataFromModel() }
    }
}
```

View: Activity
ViewModel: ViewModel

The ViewModel does not even know if there is a View, or if there are multiple ones.

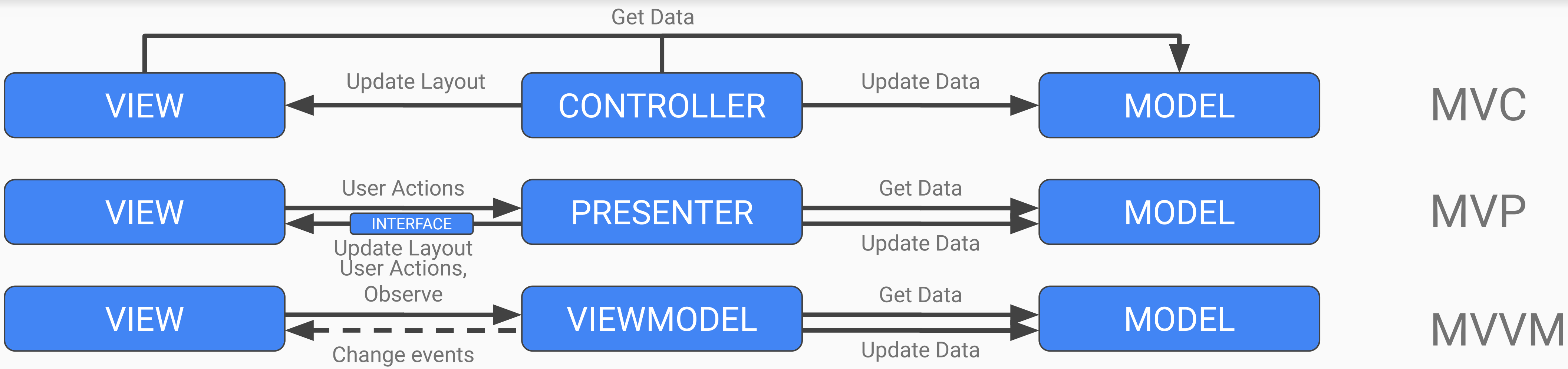
It just keeps updated its LiveData, no matter who is observing.

Comparison



	MVC	MVP	MVVM
Active Part	Controller	View	View
Testability	*M* only	*M-P* with mock View	*M-VM* unconstrained
View Constraints	Depends on *C*	Depends on *P* iface	Not dependent
Suitability	Small projects	Medium projects	Large projects

Comparison



Which one should you use?
as usual it depends, MVVM is the recommended pattern by Android. However, for a small project, MVC is much more immediate. If you work in team MVP is good, but MVVM offers easier extensibility in the future and less coupling between teams…

Others

LiveData and ViewModel are part of a bigger chunk of novelties that we will not explore. Here are the pointers:

For a tighter coupling between View elements and the UI controller we can also use:

- Data Binding
  - https://developer.android.com/topic/libraries/data-binding
- View Binding
  - https://developer.android.com/topic/libraries/view-binding

They both help in interacting declaratively with views (eliminating findViewById).

Questions?

federico.montori2@unibo.it