Architectural Components

Android mette a disposizione apposite librerie per l'attuazione dei *design pattern*, da cui ne deriva l'insieme dei benefici che li contraddistinguono. Rispetto ai comuni linguaggi di programmazione, l'ambiente di sviluppo circoscritto risulta essere più stringente su questa tematica, atteggiamento dovuto principalmente dalla volontà di mantenere al meglio delle proprie capacità una certa retrocompatibilità.

Da questo breve preambolo, nasce la necessità di organizzare e gestire il codice in maniera strutturale; richiesta che può essere unicamente soddisfatta mediante l'impiego dei *design pattern architetturali*. Inoltre è presente un ulteriore layer di difficoltà, poichè i prodotti sviluppati non dipendono da una netta suddivisione tra *client* e *server*; l'intero contesto promuove una diretta suddivisione dell'applicazione in *componenti*.

Ad esempio, la condivisione di una foto allocata all'interno di un dispositivo necessita un'elevato numero di azioni, le quali potrebbero dare vita a differenti tematiche di *dipendenza funzionale*; terminologia altamente pericolosa, dove un apparente errore potrebbe avere effetti disastrosi sull'intero sistema ideato.

Rispetto a quanto detto è fondamentale sviluppare una solida architettura di *decoupling*, in grado di assicurare che nessun componente sia strettamente dipendente ad altri. In relazione ad *Android* sono presenti tre pattern principali, denominati come segue:

- **Model-View-Controller**
- **Model-View-Presenter**
- **Model-View-ViewModel**

Tipicamente sono attuati qualora l'applicazione divenga sempre più complessa, sviluppati in base alle caratteristiche del componente che la *bussness logic* debba gestire.

Prima di specificare ognuno di essi, è bene soffermarsi sulle peculiarità comuni.

- **Model**, indica complessivamente le informazioni statiche, a lora volta denominate *domain logic*. Spesso, in ambiente di sviluppo *Android*, sono definite impropriamente *getter*
- **View**, la *User Interface*, ossia la modalità di gestione e di acquisizione dell'interazioni con l'utente
- **Bussness logic**, logica che governa l'acquisizione e visualizzazione a schermo dei dati, sezione sovrapposta tra *Model* e *View*

Per facilitare la comprensione dei *pattern architetturali* è identificata, a titolo esemplificativo, una semplice applicazione, la quale simula un utente desideroso di acquisire una bottiglia d'acqua. Il sistema è composto da una **activity_main.xml**, a sua volta suddivisa in una *textView* e in un *button*; elaborato l'evento *click* del bottone sarà popolata la *textView*.

Proseguendo, l'esempio sviluppa una classe elementare denominata *Model*, la quale imita il comportamento di un *database*, definita come segue.

```
class Model {
    private val data: String = "Water" // record all'interno del database fittizio
```

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

Pertanto, esistono tre metodi principali affinchè l'utente possa acquisire la bottiglia d'acqua; del tutto coincidenti rispetto ai tre pattern narrati precedentemente.

Model-View-Controller

In questa casistica il *Controller* è la **sezione attiva**, ossia contiene al suo interno la *bussness logic* di riferimento; mentre la *View* e il *Model* sono entità passive, non compiono alcuna mansione. Tipicamente si attua un paradigma simile qualora il progetto sviluppato sia di piccole dimensioni, garantendo un'estrema facilità di testing, dato che il *Model* non possiede alcuna dipendenza.

Tuttavia modifiche attuate al *Controller* potrebbero riflettersi rispettivamente sia sulla *View* che sul *Model*; si evince, nella sua complessità, di un sistema dipendente. Confrontando il *pattern* con l'esempio proposto, l'utente compie la totalità delle azioni, in cui:

User goes to shop, User takes the water and User pays it

Il *Controller*, il quale sviluppa la *bussness logic*, è posto all'interno di un'*activity*, affinchè possa compiere le funzioni richieste.

```
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()
        }
    }
}
```

Come da codice, è possibile osservare l'evidente grado di accoppiamento tra la *View* e il *Controller*, visualizzabili come un'unica entità; sicuramente una caratteristica simile non rispecchia l'obiettivo principale, ossia non è rispettata una suddivisione architetturale tra i vari componenti che possa isolarli da dipendenze reciproche.

Model-View-Presenter

Si tratta di una nuova concezione del paradigma, in cui la sezione attiva coincide con la *View*. Sicuramente, rispetto a quanto detto, è presente un livello astratto, in cui è incaricata un'entità di compiere determinate mansioni, senza che sia esplicitamente descritto il comportamento da attuare.

In un contesto simile varia anche l'esempio proposto, ossia l'atteggiamento detenuto dall'utente è differente, in cui:

User goes to the bar and asks for a glass of water, then the waiter brings him the water

Il cameriere simula il *Presenter*, colui che è incaricato di svolgere la specifica mansione, senza che l'utente sappia effettivamente quale sia il suo operato.

Proseguendo, il *Presenter* raffigura un mediatore sovrapposto alla *View*; per ogni *View* corrisponde un solo *Presenter* e viceversa. Il *pattern* descritto è sviluppato tipicamente per progetti di medie dimensioni, garantendo, anche in questo caso, un'elevata facilità di testing, dato che il *Model* non possiede alcuna dipendenza effettiva. Tuttavia, affinchè sia possibile garantire un *pattern architetturale* simile, occorre che sia sviluppata un'interfaccia posta tra il *Presenter* e la *View*, in maniera tale che sia garantito un tramite comunicativo.

```
class Presenter (val appView: AppView) {
        private lateinit var model: Model

        private fun getModel(): Model {
            if (! ::model.isInitialized)
                model = Model()

            return model
        }
    }

    fun getDataFromModel() {
        appView.onGetData(getModel().getData()) // Richiamata l'interfaccia non
           la View specifica
    }
}
```

Esempio implementativo di un Presenter

Rispetto alla sezione di codice riportata precedentemente, occorre stabilire due caratteristiche principali, contraddistinte come segue:

- All'interno del *costruttore* è attuato il pattern *Singleton* secondo il paradigma *lazy initialitation*, ossia l'istanziazione del *Model* è tardata fino a quando non sia strettamente necessario
- Il secondo metodo, denominato *getDataFromModel()*, da in pasto il *Model* istanziato all'interfaccia sovrapposta tra il *Presenter* e la *View*, garantendo in questo modo un'azione mirata di disaccoppiamento tra le entità

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

Esempio implementativo di un'interfaccia sovrapposta tra il Presenter e la View

Concludendo, l'approccio definito avrà delle ripercussioni sulla stessa logica attuata all'interno dell'attività; pertanto, di seguito, è riportato lo sviluppo della *MainActivity*.

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

```
        /* Presenter richiede il Contesto in input */
        val myPresenter: Presenter = Presenter(this)
        val button = findViewById<Button>(R.id.button)
        button.setOnClickListener { myPresenter.getDataFromModel() }
    }

    /* override della funzione, dato che estende l'interfaccia AppView */
    override fun onGetData(data: String) {
        val textView = findViewById<TextView>(R.id.textView)
        textView.text = data
    }
}
```

Esempio implementativo di una MainActivity che adotti il pattern MVP

Model-View-ViewModel

MVVM è il *pattern architetturale* raccomandato per lo sviluppo di una qualsiasi applicazione *Android*. Tuttavia, prima di definire concretamente le sue caratteristiche, è necessario stabilire alcuni punti cardine del discorso.

- **View-Model**, componente incaricato di memorizzare i dati relativi all'*User Interface*, mediante un approccio che rispetti il **Lifecycle** dell'*activities*. Per cui differenti sono i punti positivi di un componente simile, capace di ovviare ad azioni di salvataggio di istanze relative ad attività, qualora siano distrutte o ricreate, oppure separa la logica attuata all'*UI* dalla visualizzazione dell'informazioni.

  A livello implementativo sono definiti alcuni step a cui sottostare, dove si evidenzia:

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

  Nuovamente, nell'esempio di codice proposto è possibile notare una specifica peculiarità, in cui *MyViewModel* estende la classe *ViewModel*; ciò avviene poichè il *ViewModel* è attuato per osservare il *lifecycle* dell'oggetto passato in input al componente **ViewModelProvider**.

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

textView.text = myViewModel.getValue()
```

- **LiveData**, accomunati dallo stesso concetto su cui fondano gli *Observables*. Rappresentano classi di dati, le quali compiono *mansioni notificanti*, per tutte le entità che *osservino* le informazioni contenute in esse, qualora subiscano delle modifiche.

  In tale ambito oggetti *Observable* possiedono una lista nascosta, contenente l'insieme delle entità interessate ad acquisire feedback qualora determinati field dovessero essere

modificati. Contrariamente, i *LiveData*, come già descritto, sono *Observables* basati sul paradigma *LifeCycle Awareness*.

Il termine *LifeCycle Awareness* deriva dallo sviluppo e dall'implementazione di un *Observer* all'interno del *LifeCycle* degli oggetti.

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

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

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

Proseguendo, i *LiveData* sono componenti incaricati di notificare i propri *osservatori* solamente se posti all'interno di uno *stato attivo*; ciò potrebbe risultare piuttosto utile qualora le attività sviluppate siano orientate ai soli dati, senza che siano a conoscenza dello stato dei componenti interessati. Il meccanismo deve essere costruito affinchè possa contenere i dati correnti.

Tuttavia, i *LiveData* non possono essere modificati, ossia al loro interno non è presente alcun *setter* che possa svolgere una richiesta simile. Pertanto occorre implementare una *MutableLiveData*, in grado di poter variare il proprio contenuto anche in fasi successive alla propria istanziazione.

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

Nonostante il punto negativo descritto, tipicamente i *LiveData* sono istanziati all'interno del *ViewModel*, ne consegue che gli stessi osservatori consistano principalmente in molteplici *activities*, le quali acquisiscono il *LiveData* di riferimento, immutabile oppure modificabile a seconda della casistica.

Individuati gli elementi cardine su cui fonda il *pattern architetturale*, il discorso vira direttamente sulle peculiarità di **MVVM**.

La *View* è la parte attiva, contenente la *bussness logic*; *ViewModel* possiede al suo interno l'insieme dei *LiveData*, mentre il *Model* rappresenta la *parte passiva* dello schema. Si rinnova la facilità di testing anche per il pattern architetturale descritto, tipicamente sviluppato per applicazioni di grandi dimensioni.

Introdotto il nuovo paradigma, varia di conseguenza anche l'esempio proposto, in cui:

User goes to vending machine and pays for a water, then the machine gives back the water

Secondo una prospettiva generale, l'attitudine narrata si traduce in codice come segue.

```
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())
    }
}
```

La caratteristica principale del codice proposto consiste nell'estensione della classe *ViewModel*, dato che al suo interno sono contenuti tutti i *LiveData* interessati; la stessa entità mantiene aggiornati i propri dati, indipendentemente da chi stia osservando, mantenendo, in questo modo, un livello contenuto di *dipendencies*.

```
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() }
    }
}
```

All'interno della *MainActivity* si osserva perfettamente l'obiettivo del pattern, rendere l'*attività* totalmente estranea alla logica di acquisizione dei dati. È definita l'istanza *MyViewModel*, affinchè possano essere restituiti i dati aggiornati oppure modificati, su cui è attuato l'*observe*, in modo tale che possa associare i *LiveData* al contenuto della *textView*. Infine, mediante il *button*, avviene il refresh istantaneo dei dati del *Model*.