# Livedata e Observer Pattern

## Observer Pattern

Finora abbiamo interagito con il risultato di operazioni asincrone tramite i listener:

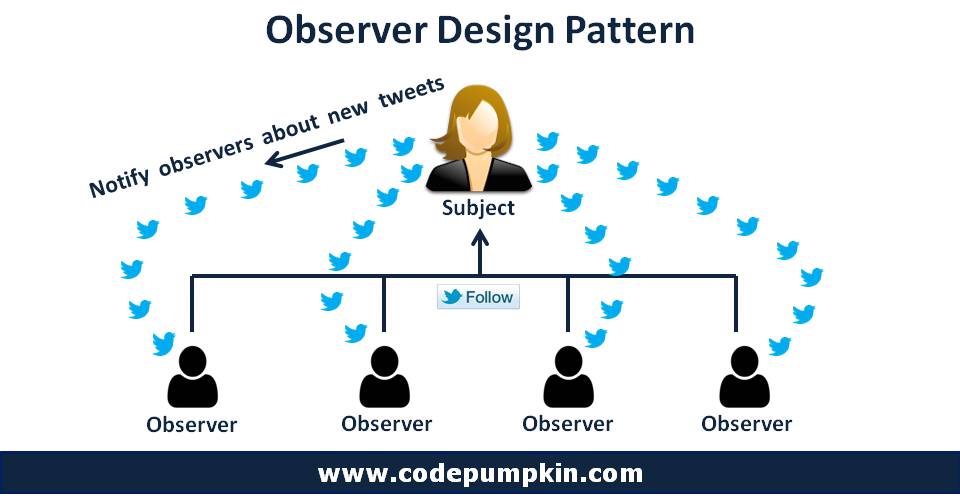
A screen shot of a computer program

Description automatically generated with low confidence

Ora vedremo come è possibile fare in modo di osservare direttamente i dati senza passare per una funzione intermedia, e fare in modo che la UI reagisca al cambio del valore di questi dati.

Ci si riferisce a questo concetto come **Reactive Programming**. Ci sono molti modi per implementarlo, noi vedremo l’**observer pattern.**

Observer Pattern definisce una dipendenza uno-a-molti tra gli oggetti in modo che quando un oggetto (Subject) cambia stato, tutti gli oggetti dipendenti da lui (Observers) vengono notificati e aggiornati automaticamente.



In questo pattern, ci sono molti **Observers** che osservano un particolare **Subject**. Gli osservatori vogliono essere informati con delle **notifiche** quando viene apportata una modifica al valore di Subject: effettuano dunque una registrazione a Subject. Quando perdono interesse, annullano la registrazione.

## LiveData

Android implementa l’observer pattern tramite **LiveData**

LiveData wrappa dei dati e li rende osservabili. Inoltre è Lifecycle-aware, si occupa quindi automaticamente di gestire le sottoscrizioni in base ai lifecycle.

Simile alle liste **LiveData è immutabile**, **MutableLiveData è mutabile**.

### Subject

Aggiungi livedata a MovieListViewModel:

A picture containing text, font, screenshot

Description automatically generated

* **\_movies** è un MutableLiveData, creato con il costruttore che ne identifica il tipo. Inizialmente, non ha valore.
* **movies** è invece un Livedata immutabile, esposto pubblico alla View.   
  View osserverà questa val per ottenere i dati dei film, e non chiamerà piu’ getMovies.
* **movies** non è nient’altro che un *cast* di \_movies a LiveData
* Rimuovi movieCache: useremo \_movies come cache ora.

Modifica getMovies:

A picture containing text, font, screenshot

Description automatically generated

* rimuovi il listener
* il risultato viene impostato come valore di movies

Ora il ruolo di getMovies è impostare il valore di \_movies.

getMovies non verrà più invocato dalla view, quindi facciamolo partire appena viene creato il viewmodel:

A picture containing text, font, screenshot, design

Description automatically generated

Infine, modifica getMovieById per far modo che si appoggi a \_movies come cache:

A picture containing text, font, screenshot, design

Description automatically generated

### Observer

In MovieListFragment cambia *getMovies* con l’osservazione di *movies,* tramite la funzione **.observe**

A screen shot of a computer program

Description automatically generated with low confidence

observe verrà lanciato **ogni volta che il valore di movies viene modificato.**

observe richiede un lifecycleOwner. Mentre in una Activity è possibile utilizzare l'activity stessa, in un fragment è necessario utilizzare *viewLifecycleOwner*.

Ricapitolando ora il flusso della nostra app:

* Quando viene creato il view model, viene lanciata la chiamata network di getMovies
* il fragment (lifecycleOwner) si registra come osservatore ai cambiamenti del Livedata movies
* la risposta arriva dal servizio e getMovies aggiorna il valore di movies
* appena il valore di movies viene modificato, il fragment riceve una notifica e aggiorna l'interfaccia

Questo farà sì che, indipendentemente da chi modificherà il valore di movies, la nostra interfaccia sarà sempre aggiornata.   
View osserva il valore di movies, e non sa chi o cosa lo modifica.

## setValue, postValue

Prima abbiamo impostato il valore di \_movies con un semplice *set*

A picture containing text, font, screenshot, line

Description automatically generated

Questa è la abbreviazione di Kotlin del setter *setValue.* Livedata però ha 2 setters:

* **setValue** imposta il valore in maniera **sincrona**
* **postValue** imposta il valore in maniera **asincrona**

Solitamente si usa postValue. SetValue è utilizzato solo nei casi in cui sia necessario mantenere una specifica sincronicità.

## Testare LiveData

Testare un oggetto osservabile può essere sfidante, perché il suo valore puo’ cambiare all’interno del test stesso.

Testeremo gli eventi in maniera **sincrona**, cosi’ da poter sempre ripetere il test e non [renderlo *flaky*](https://www.jetbrains.com/teamcity/ci-cd-guide/concepts/flaky-tests/).

Crea LiveDataTestUtil dentro la cartella di test e copia questa classe:

import androidx.annotation.VisibleForTesting  
import androidx.lifecycle.LiveData  
import androidx.lifecycle.Observer  
import java.util.concurrent.CountDownLatch  
import java.util.concurrent.TimeUnit  
import java.util.concurrent.TimeoutException  
  
  
@VisibleForTesting(otherwise = VisibleForTesting.NONE)  
fun <T> LiveData<T>.getOrAwaitValue(  
 time: Long = 2,  
 timeUnit: TimeUnit = TimeUnit.*SECONDS*,  
 afterObserve: () -> Unit = **{}**): T {  
 var data: T? = null  
 val latch = CountDownLatch(1)  
 val observer = object : Observer<T> {  
 override fun onChanged(o: T?) {  
 data = o  
 latch.countDown()  
 this@getOrAwaitValue.removeObserver(this)  
 }  
 }  
 this.observeForever(observer)  
  
 try {  
 afterObserve.invoke()  
  
 // Don't wait indefinitely if the LiveData is not set.  
 if (!latch.await(time, timeUnit)) {  
 throw TimeoutException("LiveData value was never set.")  
 }  
  
 } finally {  
 this.removeObserver(observer)  
 }  
  
 @Suppress("UNCHECKED\_CAST")  
 return data as T  
}

Questo ci permetterà di bloccare l’esecuzione del test finchè non riceviamo il risultato di livedata.

A screen shot of a computer program

Description automatically generated with low confidence

Utilizzeremo getOrAwaitValue come se fosse un normale getter.

## Esercizio

* Nella pagina di dettaglio, mostra i primi 6 attori presenti nel film.
* Dovrai integrare una nuova chiamata a TMDB:  
  <https://developer.themoviedb.org/reference/movie-credits>
* Utilizza coroutines e livedata.
* Effettua i test.