# E.1 - MVVM pattern

In questo capitolo affronteremo una serie di best practices che si possono applicare ad un ampio spettro di app. È necessario intenderle come delle raccomandazioni standard da seguire sempre. solo in casi particolari e dopo attenta analisi, possono essere adattate o ignorate.

Docs: <https://developer.android.com/topic/architecture>

In questo e nei prossimi capitoli lavoreremo su MovieMaster, un’app sul mondo del cinema e delle serie tv:

* Link progetto, utilizza il branch **start**

Finora abbiamo osservato metodi di composizione delle app, per creare funzionalità e testarle. In questo capitolo e nei prossimi, vedremo come è possibile strutturare un progetto in maniera tale che risulti sostenibile nel tempo, organizzato, e comprensibile dai colleghi che andranno a leggere il nostro codice.

Studieremo il **pattern architetturale MVVM,** che è l’abbreviazione di **MODEL/VIEW/VIEWMODEL**, l’attuale pattern consigliato per Android.

Lo scopo principale di questa architettura è di evitare [*god classes*](https://en.wikipedia.org/wiki/God_object) rispettando i principi di [*Separation of Concerns*](https://en.wikipedia.org/wiki/Separation_of_concerns) e [*Single Responsibility*](https://en.wikipedia.org/wiki/Single-responsibility_principle), e facilitare il testing.

Nel seguente schema generale dell’architettura, le frecce indicano le dipendenze dei vari Layers.

Diagram

Description automatically generated

In questa architettura:  
  
>>>>>>>>>>>>>>>>  
VIEW invierà eventi al VIEWMODEL, che in base ad essi richiederà dati a MODEL.   
  
<<<<<<<<<<<<<<<  
MODEL invierà dati a VIEWMODEL, che se necessario li elaborerà prima di inviarli a VIEW.

Questo pattern è chiamato **Unidirectional Data Flow (UDF)**

In UDF, lo **stato (dati)** scorre in una sola direzione. Gli **eventi** che modificano i dati scorrono nella direzione opposta.

L’idea di base dietro questo pattern è di rendere l’app guidata dallo stato dei dati. Sarà lo stato che verrà osservato dalla View, che reagirà in base a come esso cambia.

## Model

### Definizione

Il layer Model **è responsabile dei dati, e l’unico che può modificarli alla sorgente.** Riceve eventi ed invia dati a ViewModel.

Il Model utilizza classi chiamate **Repository** per interfacciarsi a ViewModel. Repository astrae la sorgente dati al resto dell’app, la quale non conosce niente della sorgente dati.

Solitamente all’interno del Model è contenuta la **Single** **Source of Truth**: una sorgente dati, che è proprietaria dei dati e che può modificarli. Spesso si tratta di un servizio remoto o di un database.

### Esempio

Apri MoviesRepository, attualmente vuoto:



Vedi che ha come dipendenza moviesDataSource, da cui otterremo i dati da inviare ai viewModels.

Aggiungi una nuova funzione:  
fun getMoviesResponse() = moviesDataSource.requestMovies()

Ora MoviesRepository sta astraendo la sorgente dati dal resto dell’app. Nel caso ci fossero piu’ sorgenti dati, si occuperebbe di organizzarle per fornire una risposta univoca alle richieste dei ViewModels.

* Puoi vedere queste modifiche nel branch *solution1\_repository*

## ViewModel

### Definizione

Il Layer ViewModel **è responsabile di esporre dati a View**. Riceve eventi e invia dati a View. Invia eventi e riceve dati da Model.

Il ViewModel mantiene i dati relativi alla View, che non vengono distrutti quando l’Activity o il Fragment viene distrutto e ricreato dal framework Android. Può elaborare i dati ricevuti da Model per allinearli alle specifiche necessità di View, ad esempio cambiando l’ordine degli elementi di una lista, o rendendo un testo maiuscolo.

Il ViewModel non conosce come vengono utilizzati i dati che invia a View, e non ha alcun riferimento alla UI. Non conosce come Model ottiene i dati.

### Esempio

Nel package UI, crea una nuova classe chiamata MovieListViewModel, che ha come parametro del costruttore il repository MoviesRepository, e che estende ViewModel



Crea ora una funzione che permette alla view di ottenere la lista di Film in ordine di data di uscita recente:



Così facendo ViewModel fornirà i dati a View, senza che essa sia a conoscenza di come vengono recuperati, o di quale sia la loro struttura originale.

## View

### Definizione

Il Layer View (Activity, Fragment) è **responsabile della presentazione e dell’interazione con l’utente**. Informa il ViewModel sulle azioni dell'utente e osserva lo stato, esposto dal ViewModel. In base allo stato, mostra la corretta UI.

View osserva i dati senza modificarli, e non conosce come vengono elaborati né come vengono ottenuti.

Il sistema Android può distruggere e ricreare Activities e Fragments in qualsiasi momento in base a determinate interazioni dell'utente, ad esempio la rotazione del device, o a causa di condizioni del sistema, come memoria insufficiente. Poiché questi eventi non sono sotto il tuo controllo, la logica decisionale sui dati è di solito contenuta nel ViewModel, che ha la capacità di mantenere il suo stato.

### Esempio

Ci affidiamo al framework android per recuperare il viewmodel nella view:



A questo punto potremo utilizzare la funzione che abbiamo creato per ottenere i dati:

Text

Description automatically generated

E’ ora possibile lanciare l’app per vedere una lista di cards con ID.

* Puoi vedere la soluzione nel branch *solution2\_viewmodel*

## Concetti di ownership

Doc: https://developer.android.com/topic/libraries/architecture/viewmodel/viewmodel-apis

Nella view (fragment) abbiamo ottenuto il viewmodel tramite *by viewModels*, una funzione che permette di definire lo **scope** del viewmodel.

Lo scope è un riferimento che viene utilizzato dal sistema Android per definire quando l’istanza di un viewModel deve essere distrutta.

Lo scope è un collegamento ad un *ViewModelStoreOwner*, un’interfaccia implementata dai *LifecycleOwner.*

Un viewmodel può essere **scoped to:**

* Activity
* Fragment
* NavigationGraph

Ognuno di questi elementi ha dei componenti interni che il sistema Android prende a riferimento quando deve effettuare operazioni di distruzione\ricostruzione, e che rimangono in memoria durante queste operazioni. Finchè anche questi riferimenti non verranno rimossi dal sistema, l’istanza del ViewModel resterà disponibile.

Queste sono le funzioni più comuni disponibili per lo scoping:

* **by viewModels()** : scoping al candidato più vicino. Se lanciato in un fragment, sarà scoping al fragment. Se lanciato in una activity, sarà scoping all’activity.
* **by activityViewModels()** : scoping all’activity *parent* di dove viene lanciato. Se lanciato in un fragment, sarà scoping all’activity che lo contiene.
* **by navGraphViewModels(R.id.nav\_graph)** : scoping al navigation graph fornito come parametro

Queste funzioni cercano in memoria un’istanza di quel viewmodel con quello scope.   
Se la trovano, forniscono un riferimento a quell’istanza.  
Se non lo trovano, creano in memoria l’istanza del viewModel e gli assegnano lo scope, e poi forniscono il riferimento.

Ad esempio, è possibile usare *by activityViewModels()* per condividere la stessa istanza di viewmodel tra piu’ fragments.

## Esercizio: shared viewmodel

Utilizza I dati disponibili nel moviesDataSource per popolare l’app con i seguenti presupposti:

* Il fragment lista deve avere palmeno immagine e titolo del film. Il layout riga deve essere semplice e contenere pochi dati utili.
  + La documentazione del MovieDB, il servizio da cui sono stati presi questi dati, è disponibile qui: <https://developer.themoviedb.org/docs>
* Il fragment dettaglio può contenere la maggior parte dei dati di Movie, in particolare il titolo, la locandina, e la descrizione. Mostra questi e altri dati in modo chiaro, e non mostrare dati senza che l’utente possa comprenderli (ad esempio, popularity stampato senza contesto)
* I fragment lista e dettaglio devono condividere la stessa istanza di viewmodel
* Nella navigazione, passa al fragment dettaglio solo l’id del film. Recupera il Movie corretto nel fragment di dettaglio, tramite il viewmodel.
  + Per rigenerare i file creati dai plugin di navigazione, come le directions o i navArgs, effettua un clean del progetto
* Feature: film preferiti.  
  Aggiungi a Movie una *var* *favourite: Boolean*   
  Aggiungi una stella nell’interfaccia di dettaglio, che puo’ essere cliccata per contrassegnare un film come preferito.   
  Anche la lista deve mostrare se un film è preferito oppure no, ma non deve essere modificabile dalla lista.  
  Non è importante (per ora) che i film preferiti permangano quando si chiude e riapre l’app
  + Per la stella, puoi importare una drawable nuova o utilizzare alcune disponibili nelle risorse android.R.drawable