# 08 Android Architecture Patterns

Un’applicazione è un pezzo di sw che gestisce dei dati secondo delle business rules.

* L’interfaccia utente dell’applicazione dipende dai dati interni
* Quando l’utente interagisce con l’applicazione, la GUI cambia in base all’azione, perché qualcosa è cambiato → c’è qualche evento → interfaccia cambia.

Se il data set è piccolo e le business rule sono semplici → solitamente no problem

* Ma se l’applicazione è grande → difficile mantenere un codice coerente
  + Bisogna pensare come strutturare l’applicazione in modo tale da riuscire a resistere ad un crescente numero di dati

## Architectural Patterns

**Architectural patterns** are a general, reusable resolution to a commonly occurring problem in conceiving software, within a given context:

* Alcuni pattern sono utili quando si opera in piccoli problemi semplici
* Architectural pattern vedono la nostra applicazione considerando problemi come
  + Deve essere **mantenuto** da diverse persone
  + Deve evolvere e quindi deve essere facile svolgere determinate azioni
  + Operational issue → questo deve essere garantito disponibile sempre?
    - **Availability** → sempre dispobnibile?
    - **Scalability** → deve essere capace di restistere a diverse quantità di user
  + **Business** **rule** → es: Login → applicazione deve registrare utente e mantenere dati
    - Farlo io → pw abbastanza sicura, pw
    - Prendere già fatto da altri

Le appliocazioni android sono costruite mettendo insieme una **serie di componenti** che sono gestiti direttamente dal OS

* Attività, servizi, content provider, boradcast receriver

### Componenti

**Components** can be **launched** individually and **out-of-order,** and can be destroyed at anytime by the user or by the system

* This may lead to unexpected sequences of events that makes designing and testing an app a difficult task

**Separazione dei concerns**:

* Bisogna evitare di storare dento l’activity o dentro un component direattamente delle informazioni riguardanti lo stato → andrebbero perse. Infatti, noi non creiamo proprietà nelle nostre attività.
* Bisogna evitare che un dato component dipenda dall’esistenza, in vita, di un altro component:
* Started service: fa iniziare attività e non si interessa più
* Bound service: finchè ci servel il servizio sarà vivo, garantisce android

*Da questo punto di vista, può essere problematico prendere dati da un remoto server, in quanto potrebbero arrivare quando le attività non sono più esistenti.*

Classi che estendono activity e Fragment devono avere a che fare con UI o interazioni di OS

* These classes are intended to be glue classes and can be destroyed at any moment by the user or by the OS

Il contenuto dell’UI fa riferimento a un modello, preferibilmente persistente:

* Utente non perde i dati se l’applicazione viene ditrutta
* L’applicazione continua a funzionare anche se ci sono problemi di connesione di rete

### **Stato Applicazione e dati:**

Dati dell’applicazione si **evolvono** quando interagisce utente

* Application state deve essere sempre consistente con le business rule dell’applicazione e azioni utente

*Ci sono alcuni dati che sono acceduti anche se non saranno modificabili direttamente.*

Dati vengono **salvati** in vari modi:

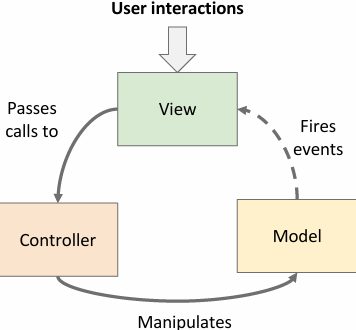
* Possono essere salvati in **variabili** che vengono perse se si chiude
* Alcune salvate **all’interno** del mio device
* Salvate in un **server**
  + Possono essere accedute da diversi dispositivi

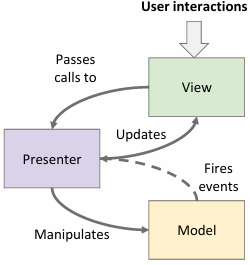
Lo **stato può essere**:

* Ui state → è il menù aperto?
* Form state → cosa è stato inserito nei vari input
* Application state → profilo personale, che si può editare
* Read-Only data → tutti gli articles nella sezione blog dell’applicazione

### **Vari tipi di architecture**

3 maggiori evoluzioni sono :

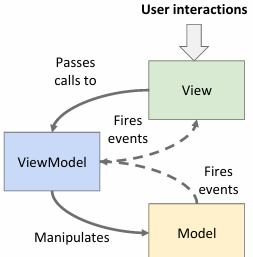
* **ModelViewController** 
  + Il più vecchio
  + Model → dove mantiene le informazioni
    - Contiene:
      * Dati
      * Metodi che permettono l’evoluzione dei dati secondo le regole
    - Può essere:
      * In un server remoto → proxy per remote server
        + Invochi metodi locali e questi si occupano di gestire i remoti
  + View → crea la view →GUI
    - Dipende dai dati di model
    - Colleziona le iterazioni dell’utente
  + Controller → manipola il model
    - Trasforma iterazioni utente in operazioni sul modello



* **Model-View-Presenter** 
  + Model
  + View: insieme di oggetti →GUI
  + Non si possono invocare metodi su una composition → non funzia
  + Presenter: model and view Manipulation
  + Difficile da usare con jetbacCompose, se vuoi update asincrono → so cazzi
* Model-View-ViewModel → attuale

## Model-View-ViewModel

* **Model**
  + Deve riportare eventi a viewModel
  + Diviso in due layer
    - **BusinessLogic**
      * Immagine che contiene testo, schermata, Carattere, diagramma

        Descrizione generata automaticamenteInterfaccia, insieme di metodi che ci garantisce che possiamo manipolare il model e ispezionarlo per vedere che contiene
    - BL deve **prendere informazioni**
      * **LocalDataSource** → database (SQLite)
        + Cache
        + Se ci sono modifiche devo propagare al server remoto
      * **Remote data Soource** → network
        + Dati reali, presi usando coroutine
* **View**
  + Quando ottiene una notifica → ridisegna immediamente
  + Insieme di composable function
* **ViewModel**:
  + Deve riportare eventi alla view
  + Garantisce che informazioni rilevanti che devono essere date alla view, sono sempre ok.
  + internamente contiene proprietà observable
    - Oggetti mutablestate
    - Flows
    - liveData
  + *il viewModel manipola il model*
  + *quando il model notifica il suo cambiamento, il viewModel deve aggiornarsi e se sono cambiate proprietà obervable, deve triggerare la view*

**Dunque**:

* *viewModel sente le notifiche della view e le manda al model*
* *il model ha business logic che è collegata alle sorgenti di dato*
* *notifica viewModel dove cambiano observaldata*
* *vireModel avvisa View che cambia i disegnini*

**Quali sono i vantaggi?**

* Clear separation of concerns as in MVC and MVP
* Higher decoupling, since there are no direct invocations from ViewModel to View
* Less synchronization problems
* Improved testability

**Svantaggi**:

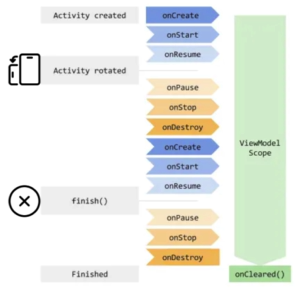
* Complessità elevata

*Bisogna definire quali informazioni salvare, come bisogna salvarlarle e come propagare le modifiche.*

### ViewModel

**ViewModel**: elemento che si trova tra View e Model. Deve riuscire a propagare gli aggiornamenti alla view in modo **lyfecycle conscious** in quanto la View può esistere ma può anche non esistere come conseguenza di uno spegnimento schermo/rotazione. Il viewModel sa che deve propagare informazione ad un qualcosa effimero.

Necessita di essere **creato dall’OS**, insieme di funzioni specifiche che garantiscono che una data istanza del viewModel viene creata in un dato istante di tenpo. In Compose possiamo passare un viewModel ad ogni Composable dicendo =ViewModel().

Il vm ha il suo **lifecycle bounded a quello dell’activity**. Una specifica istanza di un’attività può essere creata e successivamente distrutta ma se se una activity è distrutta ma non marcata come finita → viewModel continua ad essere vivo e quando l’attività sarà ricreata, lo stesso vm sarà usato.

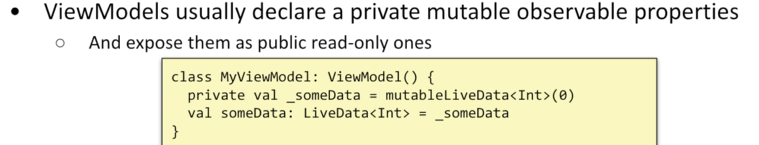
Quando l’attività viene distrutta e marcata come finita, viene invocato il metodo onCleared che accede alla struttura e rilascia ogni struttura e risorsa.

*Per essere usata come un viewModel, una classe deve estendere la classe astratta androidx.lyecycle.ViewModel o una sua sottoclasse. Una sottoclasse rilevante è AndroidViewModel.*

**Android viewModel subclass** contine un’istanza dellapplication object che permette di accedere al viewModel di accedere al fyleSytem.

Quando si **implementa un viewMode**l bisogna inserire:

* i **metodi** necessari per propgare la **l’iterazione fisica della GUI al model**
  + (bottono premuto → devo invocare metodo x su modello)
* un insieme di **proprietà Observavble che servono all View per mostrare i dati**
  + Observable → capacità di un oggetto di notificare i cambiamenti avvenuti sui suoi dati, agli altri.
    - View si ridisegna, recomposition in jetbackCompose.
    - Value : valore corrente in observable



Se dichiaro private → gli altri possono vedere ma non modificare, ViewModel riceve notifiche, valida e poi modifica lui.

Gli elementi del viewModel possono essere rappresentanti come:

* LiveData<T>
* State<T>
* StateFlow<T>
* SharedFlow<T>

Tutti questi hanno la Mutable..

Noi creiamo la variabile mutable privata e i setter.

LiveData: quando la tecnologia è ObjectBased

* Osservabile da oggetti che hanno lifecycle → capace di notificare agli altri i suoi cambiamenti
* Immagine che contiene testo, schermata, Carattere, linea

  Descrizione generata automaticamenteConosce il suo state

*LiveData può essere convertito a State usando .observeAsState()*

**QUALE OSSERVABLE USARE?**

* **LiveData<T>**: funziona senza problemi con gli oggetti View standard, poiché ereditano il lifecycle state dall’attività/fragment in cui sono ospitati
* **State<T>:** è destinato alle GUI di Jetpack Compose: una modifica del suo valore attiva automaticamente la ricomposizione, ma è complesso usarlo al di fuori di quel framework
* **SharedFlow<T>** e la sua sottoclasse **StateFlow<T>** fanno parte delle librerie di coroutine Kotlin e forniscono una soluzione generale all'osservabilità indipendente dal framework Android
  + StateFlow: puoi usarlo anche in model e può essere convertito a State usando .collectAsState

**Un viewModel deve essere sempre accessibile**. Ci sono alcune cose ch non devono essere nel viewModel:

* Non si può storare nel ViewModel:
  + Activity
  + Views derivate da activity
  + Fragment derivate da activity
  + Risorse
    - Es: posso storare nel viewModel l’id dell’icona ma non l’icona in se
    - Se salvo dentro il viewModel qualcosa ch eocntiene un riferimento all’attività, distruzione dell’attività fa casino
* Un ViewModel non può osservare i propri dati

Eccezione a questa regola è fatta dalla sottoclasse **AndroidViewModel** di viewModelClass:

* Speciale classe che **mantiene un reference all’application object** (prima cosa creata e ultima ad essere distrutta)
* Noi dobbiamo esser sicuri che il viewModel acceda al Model in un modo singleton

**Nota**: quando si crea un view → noi non abbiamo bisogno di accedere all’intero viewModel ma solo alle parti che ci interessano → abbiamo bisogno di accedere al Model in questa maniera e garantendo di avere una sorgente consistente.   
Per fare ciò storiamo nel nostro **application object**, la nostra copia del Model. **Android viewMode**l contiene la reference all’application object e quindi ispezionare le sue proprietà e quindi ottenere il Model.

**Note su ViewModel:**

Un ViewModel è la posizione naturale in cui è possibile sollevare lo stato della vista

* Questo dà la possibilità di mutare diverse proprietà come conseguenza di un singolo metodo di invocazione

I metodi che propagano le richieste di modifica al modello possono richiedere il passaggio a un thread diverso

* Android vieta, infatti, di eseguire operazioni a esecuzione prolungata dal thread principale

I ViewModel possono essere utilizzati anche come livello di comunicazione tra i diversi frammenti ospitati all'interno della stessa attività

* Questo disaccoppia lo scambio di dati permettendo ai frammenti di interagire senza necessariamente essere vivi allo stesso tempo

Tipicamente usiamo il view Model per hosting i nostri dati delle applicazioni → top del tree deve contenere le informazioni che flowano giù dove servono. Uno dei vantaggi del flow come comunicazione è il fatto che possiamo switchare ad un diverso thread molto facilmente (operazioni lunghe non su main thread)

Immagine che contiene testo, schermata, Carattere, numero

Descrizione generata automaticamente

→ nullabili

Immagine che contiene testo, schermata, Carattere, numero

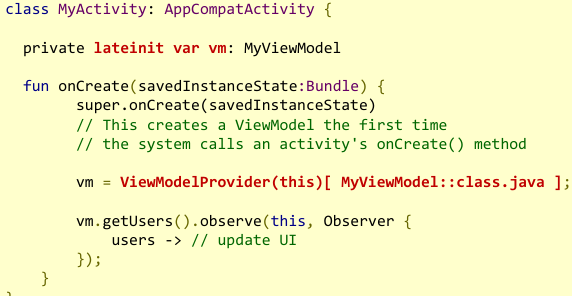
Descrizione generata automaticamente

Immagine che contiene testo, schermata, Carattere, numero

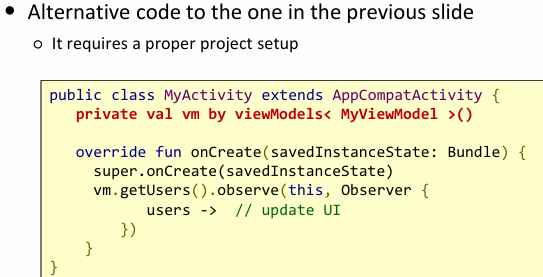
Descrizione generata automaticamente

Codice 8maggio

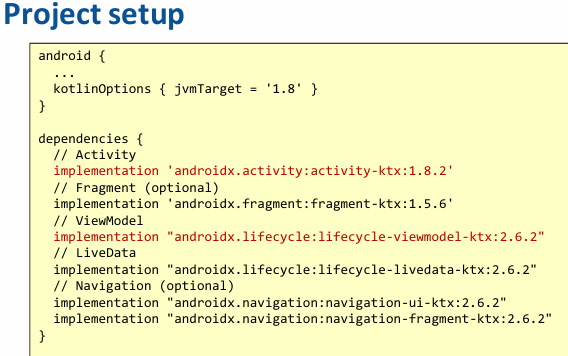
**Dietro il viewModel c’è una Factory**

Un viewModel deve essere **creato**:

→ se c’è un viewModel, perfect; altrimenti prova a crearlo



→ accetta il nome della classe che si vuole creare e la Factory



Funzione viewModels() → specifica che si prova a prendere da viewModel se c’è

**VITA**:

* viewModels() è connesso alle activity, vive finchè vive l’activity;
* viewModel può essere legato ad un singolo Fragment e che quindi esistere solo finchè il Fragment esiste.
* viewModel può essere legato ad un navigationGraph (es: di 3 activity) e che quindi rimane vivo finchè una delle tre è ancora viva.

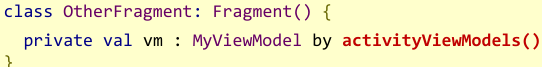
*ViewModel non vive quanto l’attività fisica dell’attività ma quanto quella lgoica.*

**Fragment:**

* può ottenere il proprio ViewModels → by viewModels()
* può porenderlo dall’attività → by actictivityViewModels()
* può prenderlo dal navigationGraph → by navGraphViewModels(…)

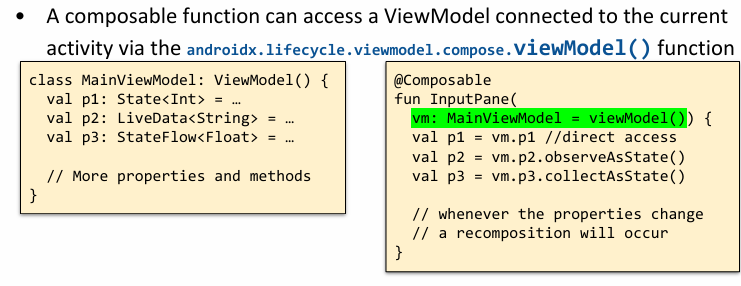
**nota**: A ViewModelFactorycan be provided to these functions in order to pass parameters to the ViewModel, if it has not been created, yet

*Condividere VM con fragments:*

Immagine che contiene testo, schermata, Carattere

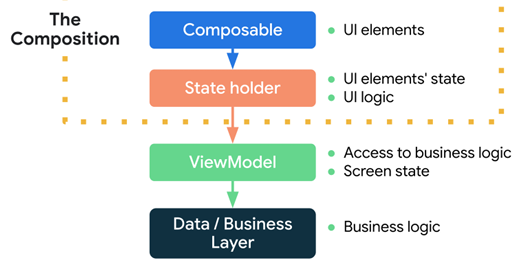
Descrizione generata automaticamente

In ogni composable noi abbiamo viewModel function



→ in ogni caso è triggerata recomposition e schermo ridisegnato

Nota: ci sono alcune informazioni rilevanti per viewModel e altre no:



→ in alcuni casi variabili di stato relative ad UI

→ metodi per accedere all business logic (es: cnt non può essere inferirore a 0) e seleziona dei pezzi dal model che ritiene importanti o che gli semplificano la vita.w

Adatto i dati presenti nel Modello ai metodi presenti nella presentation.

--post 35 → SALTATE DAL PROF