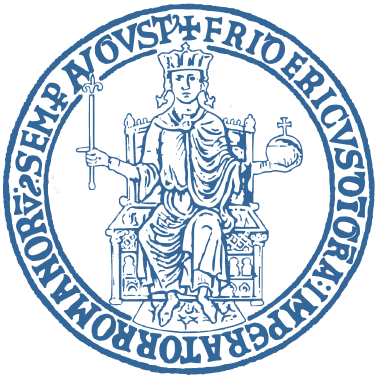

UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II

Corso di Laurea Magistrale in Ingegneria Informatica

PATTERN ARCHITETTURALI PER APPLICAZIONI MOBILI

Prof.ssa Anna Rita Fasolino

Aletto Andrea, M63/582

Coppola Bottazzi Umberto, M63/552

Introduzione

Durante il corso di Progettazione e Sviluppo dei Sistemi Software è stato dato ampio spazio alla trattazione dei pattern architetturali, che rappresentano delle soluzioni *ready-to-use* per costruire l'architettura software, ovvero ciò che può essere definito *fondamenta* di ogni sistema software, senza cui esso è destinato decadere nel tempo per mancanza di estendibilità, manutenibilità, portabilità, flessibilità, scalabilità, disponibilità e così via.
Uno dei pattern ampiamente seguiti ed utilizzati nella fase di progettazione di un sistema è il pattern Model-View-Controller, che separa il modello dei dati (che risiede nel package Model), dalla logica di gestione ed analisi degli stessi (che risiede nel package Controller), dal modo in cui tali astrazioni sono visualizzate (che risiede nel package View).
La forza di questo pattern si trova principalmente nel forte disaccoppiamento che esiste tra le entità, la logica di business e la logica di visualizzazione della UI, rendendo più o meno semplice la modifica di un package evitando che essa si ripercuota su tutto il sistema.
In tale soluzione, l'input dell'utente viene fornito tramite il Controller, il quale da un lato si preoccupa della gestione dei dati tramite i servizi offerti dal Model e dall'altro comanda la View di aggiornarsi opportunamente.

Con l'avvento dei dispositivi mobili, tuttavia, le responsabilità di ciascun package sono in parte variate, in quanto è la stessa View che, oltre a mostrare i cambiamenti, si occupa anche di porsi in ricezione degli eventi che può generare l'utente come input. Ciò ha fatto sì che nascessero nuovi pattern architetturali che meglio si adattano alle applicazioni mobili, ma che sono comunque visti come una derivazione del MVC. In particolare, quelli più diffusi sono il pattern Model-View-Presenter ed il pattern Model-View-ViewModel.

Nel corso dell'elaborato verrà approfondita la trattazione dei suddetti tre pattern architetturali in ottica di sviluppo app mobile e, successivamente, mostrata un'implementazione del pattern MVP.

I pattern MVC, MVP, MVVM

MVC - Model View Controller

Nel pattern architetturale MVC deployato in applicazioni mobile non è possibile distinguere in maniera netta il Controller e la View. Questo è dovuto al fatto che la View non è più solo un mezzo per notificare l'utente di un evento, ma è anche un componente attivo con il quale questi richiede l'esecuzione di un'operazione. In quest'ottica risulta difficile trovare una collocazione precisa per il gestore dell'interfaccia, il quale, in un'implementazione MVC di un'app mobile, farà da un lato le veci di View (poiché deve occuparsi di settare appropriatamente l'interfaccia) e dall'altro le veci di Controller (poiché deve sapere come gestire i dati offerti dal model, quando l'utente invia un comando tramite la stessa UI).
Per questi motivi si è ritenuto il pattern MVC non adatto allo sviluppo di applicazioni mobili.

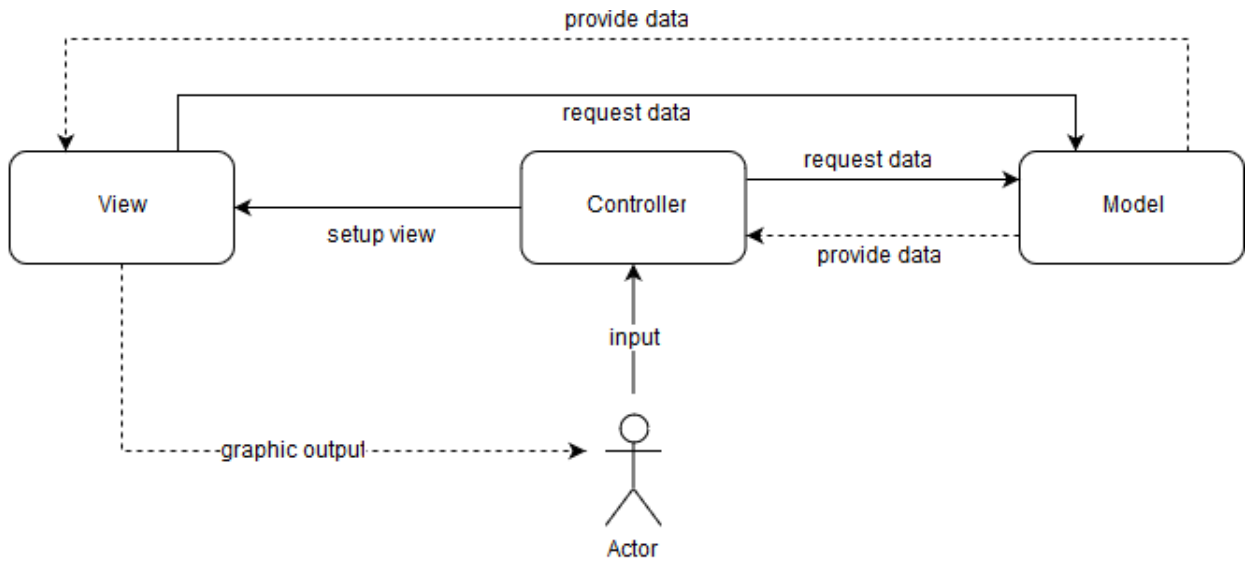

Figura 1: Model View Controller

È importante, ai fini di un confronto tra i tre pattern, andare ad individuare dapprima le responsabilità di ciascun package e, successivamente, analizzarne i pro e contro.

M Il package Model è responsabile di incapsulare i dati e di gestire opportunamente i servizi di retrieve e store degli stessi. Esso offre delle astrazioni semantiche ben distinte andando ad esibire una o più classi software utilizzabili tramite dei metodi.

V Il package View è responsabile di notificare l'utente della rappresentazione dei dati del modello, andando ad impostare l'interfaccia nella maniera opportuna, seguendo i cambiamenti e gli eventi che occorrono nel model.

C Il package Controller è responsabile di informare la View che occorre aggiornare uno o più elementi della stessa, sulla base del risultato dell'interazione con il Model e/o con l'utente.

Pro:

- MVC, in quanto utilizzato da anni, è ampiamente testato ed affermato;
- Separazione netta delle responsabilità.

Contro:

- Difficilmente testabile e manutenibile a causa dell'accoppiamento del codice del Controller e della View.

MVP - Model View Presenter

Nel pattern architetturale Model-View-Presenter si ritrovano i due package Model e View. Il primo è sostanzialmente identico al MVC, in quanto contiene le entità del sistema; la View, invece risulta essere più complessa: essa non è solo un contenitore di tutti gli elementi grafici della UI, ma è formata anche da quelle entità atte a controllare tali elementi. Un esempio su tutti è l'Activity di Android o il ViewController di iOS, che in un'implementazione MVC finirebbero per essere classificati come parte del package Controller.

In MVP, dunque, coloro i quali devono settare l'interfacciare vengono classificati come View, e in questo pattern architetturale si assume la View come un package che è privo di logica di business dell'app, in grado solo di richiedere un certo dato o di eseguire un comando fornito da un'entità esterna. In questo contesto si configura il ruolo del Presenter, il quale è *aware* del modo in cui i dati devono essere presentati e di *cosa* deve essere permesso o negato all'utente, e, ovviamente, si fa carico di interfacciarsi con il Model per richiedere o fornire dati.

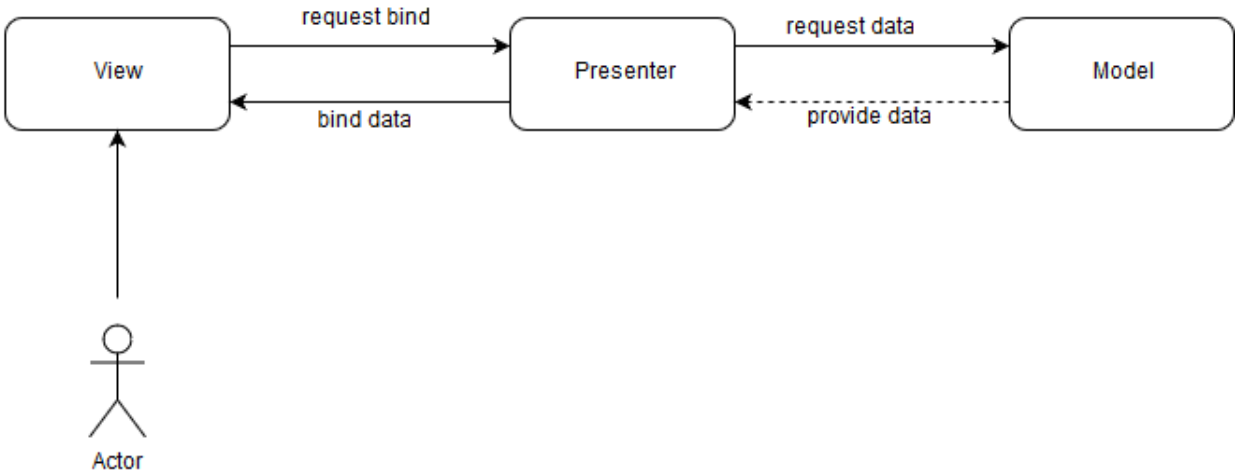

Figura 2: Model View Presenter

Dunque il ruolo del Presenter è, come nel pattern MVC, alla stregua di un Controller, ma con la peculiarità che esso non deve conoscere l'implementazione della UI. Esso si configura come un elemento di intermediazione tra il modello e l'interfaccia, fornendo ed utilizzando servizi offerti dagli altri package. In dettaglio, data la natura logic-free della View, il Presenter deve fornire a quest'ultima le informazioni necessarie per impostarsi in maniera adeguata. In altre

parole, al trigger di un evento dell'interfaccia, la View saprà come reagire solo "chiedendolo" al Presenter, il quale, disinteressandosi di come l'interfaccia è implementata, andrà a fornire alla View le istruzioni e i dati necessari per impostarsi adeguatamente.
Risulta, in questo contesto, utile chiarire subito i due vincoli fondamentali che esistono nell'implementazione della comunicazione tra View e Presenter:

1. Le azioni che dovrà eseguire il Presenter devono essere indipendenti dall'implementazione della UI, pertanto il Presenter stesso dovrà essere indipendente dalle classi utilizzate nel package View.

2. Sarà la stessa View ad impostarsi, su ordine del Presenter, utilizzando i gestori della UI; pertanto il Presenter dovrà inviare alla View solo i meri dati da visualizzare.

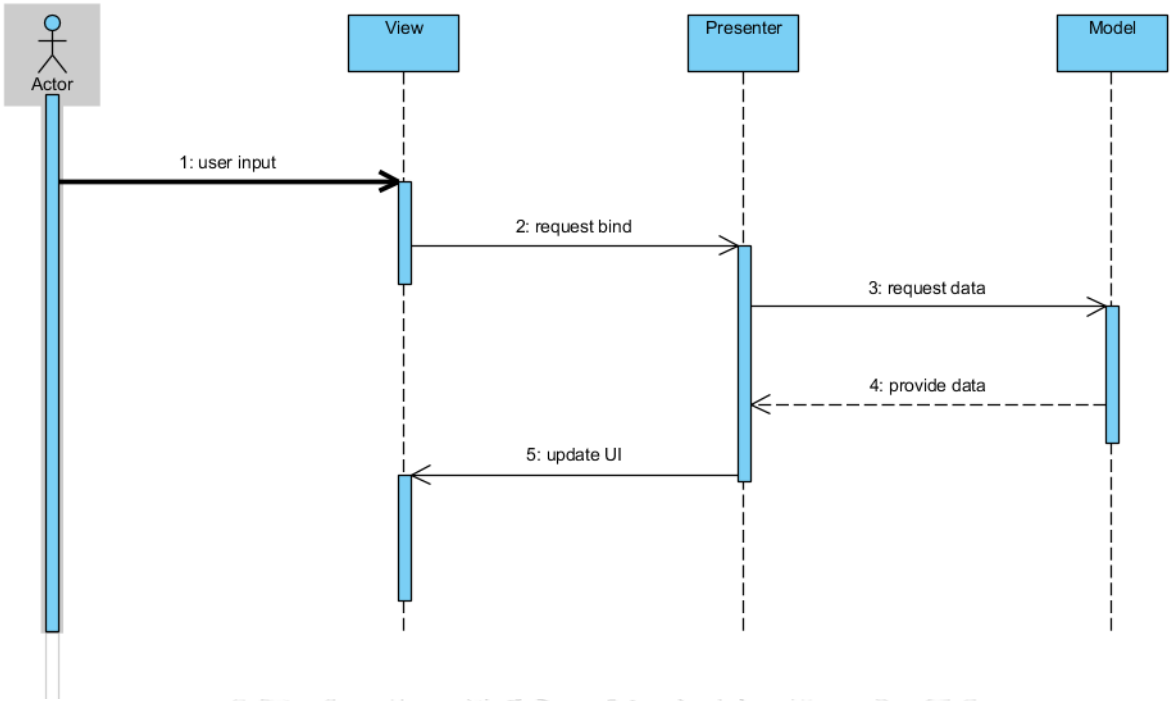

Figura 3: Interazione MVP

Analizzando ora le responsabilità, si noti che:

M Il Model risulta identico al MVC.

V La View è responsabile di impostare e instanziare tutti i componenti della UI.

P Il Presenter ha le stesse responsabilità del Controller in MVC, troncate, però, all'invio di un comando di aggiornamento della UI, eseguito tramite un'interfaccia che la View dovrà implementare.

Pro:

- Separazione netta delle responsabilità e mapping 1:1 nel contesto mobile;
- Migliore manutenibilità a fronte di un cambiamento di uno dei tre package.

Contro:

- Il test del Presenter è subordinato all'esistenza di una classe che implementi l'interfaccia della View, qualsiasi essa sia, non per forza la UI definitiva. Dunque è possibile, in questa architettura, andare a testare il Presenter impiegando una UI mock.
- Nel voler seguire pedissequamente il pattern, ogni evento della View dovrebbe richiedere l'esecuzione di un metodo del Presenter, aumentando notevolmente la dimensione del numero di linee di codice.
- Dovendo la View non essere a conoscenza del Model, il Presenter nel fornire i dati dovrà "scompattare" le informazioni di quest'ultimo. Al crescere delle informazioni mantenute dal Model cresce linearmente anche la quantità di parametri di ingresso/uscita delle funzioni di comunicazione.

MVVM - Model View ViewModel

Nel pattern architetturale Model-View-ViewModel viene inserita una nuova entità, il ViewModel, che si configura come un wrapper costruito intorno al Model. In questa architettura la View non è più logic-free, ma implementa la logica di gestione della UI basata sul valore dei dati del Model. Tuttavia, nell'ottica di non voler accoppiare View e Model, viene creata l'entità ViewModel che si occupa di fornire alla View un'astrazione Model-independent dei dati da rappresentare.

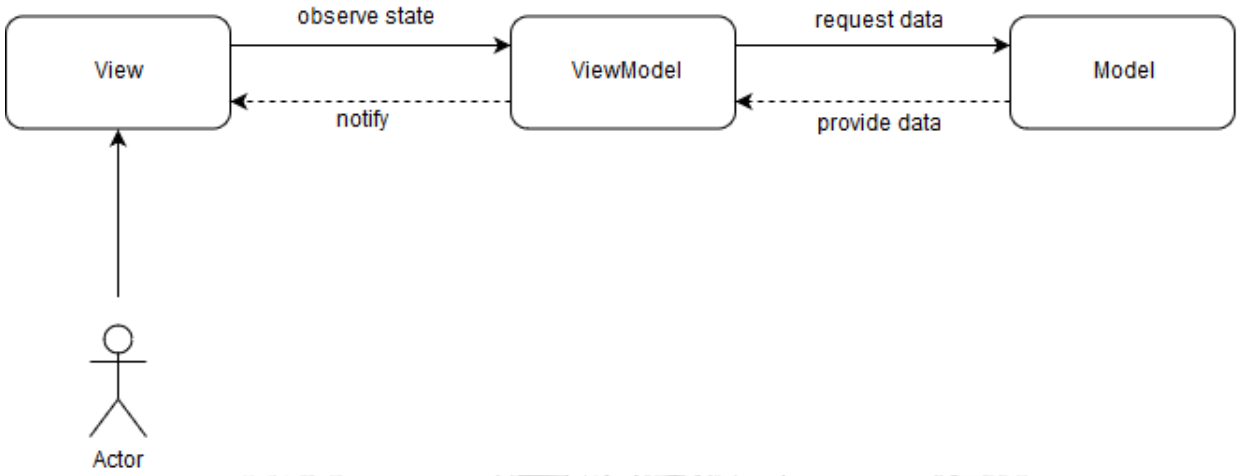

Figura 4: Model View ViewModel

Il ViewModel, dunque, conterrà una o più variabili che rappresentano, utilizzando dei tipi base, le informazioni estratte dal Model.

In questo contesto risulta estremamente utile l'utilizzo del DP Observer, rendendo le variabili del ViewModel di tipo observable e facendo in modo che gli osservatori siano le classi della View. Uno dei pattern di utilizzo, quindi, prevede che la View osservi le variabili del ViewModel e che quest'ultimo aggiorni tali variabili all'atto dell'interfacciamento con il Model.

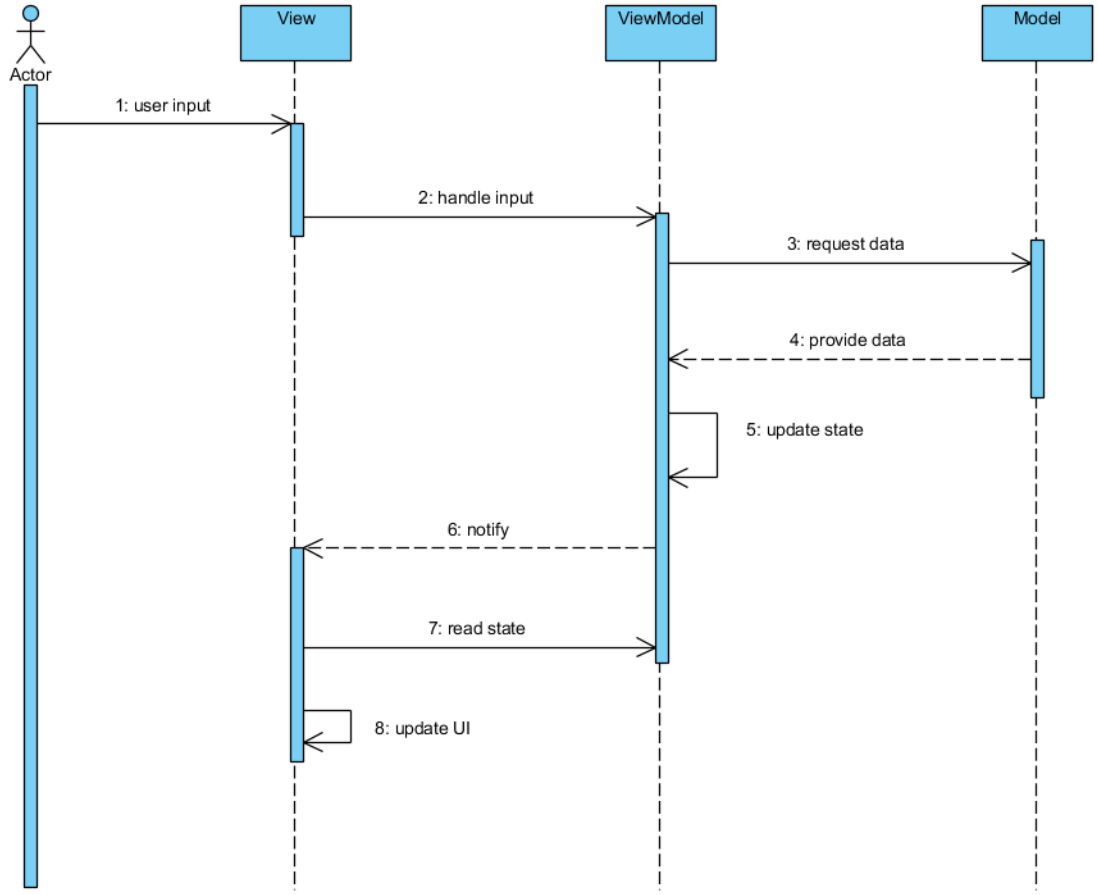

Figura 5: Interazione MVVM

Analizzando ora le responsabilità, si noti che:

M il Model risulta identico al MVC.

V La View è responsabile di impostare e instanziare tutti i componenti della UI e di implementare la logica di gestione della rappresentazione dei dati.

VM Il ViewModel ha la responsabilità di wrappare il Model e di fornire l'astrazione necessaria alla View per impostare la UI.

Pro:

- Testing semplificato, in quanto il ViewModel è testabile semplicemente verificando che nelle variabili di interfaccia con la View siano presenti gli opportuni dati del Model;
- La View è aware della logica di rappresentazione dei dati, dunque il progetto è generalmente composto da meno linee di codice.

Contro:

- La divisione dei compiti e delle responsabilità è più blanda rispetto a MVP ed MVC (la View è sia responsabile della UI che della logica di presentazione);
- Manutenibilità debole della View a causa del fatto che la grandezza dei package è sbilanciata.

Implementazione di MVP su Android

L'utilizzo di pattern architetturali per applicazioni mobile non è una pratica puramente teorica, ma è fortemente consigliata anche da Google in ambito, ovviamente, di sviluppo Android. Google ha di fatto pubblicato degli skeleton di progetti in cui è mostrata un'implementazione embrionale dei pattern MVP e MVVM. Nel seguito verrà analizzata l'implementazione del pattern MVP in Android utilizzando il linguaggio Java.

Dovendo la View e il Presenter utilizzare dei servizi offerti dalle due parti, è ragionevole dichiarare la lista di tali servizi in due interfacce, chiamate per l'appunto View e Presenter, che dovranno essere implementate dalle classi corrispettive.
Al fine di rendere tale trattazione più semplice e scorrevole, verrà ora presentato un semplice progetto che fa uso del pattern architetturale MVP. In particolare verrà realizzata un'app Android Java in cui alla pressione di un bottone verrà mostrato un nome casuale di persona in una TextView e sarà consentito all'utente di inserire un nuovo nome tramite una EditText. Tale progetto è disponibile clonando la seguente repo:

```
$git clone https://github.com/andreaaletto/AndroidMVPExample.git
```

Procedendo per package, sarà presente un Model che realizza l'astrazione di una lista di nomi e fornisce un metodo per estrarre un nome causale da essa ed un metodo per inserire un nuovo nome in essa; una View che contiene la MainActivity, il layout xml utilizzato e tutte le classi di elementi grafici utilizzate (due Bottoni, una TextView, una EditText ed un Toast); un Presenter che riceve richieste dalla View ed esegue operazioni sul Model prima di aggiornare la stessa. La comunicazione tra la View e il Presenter è regolamentata dalle due interfacce apposite.
La View deve essere in grado di:

1. accogliere i dati passati dal Presenter;
2. essere aggiornata dal Presenter a fronte del completamento del caricamento di un nuovo nome.

Il Presenter deve essere in grado di:

1. soddisfare la richiesta da parte della View di voler ricevere un nuovo nome;
2. soddisfare la richiesta da parte della View di voler inserire in lista un nuovo nome.

In figura è mostrato il class diagram dell'architettura MVP implementata.

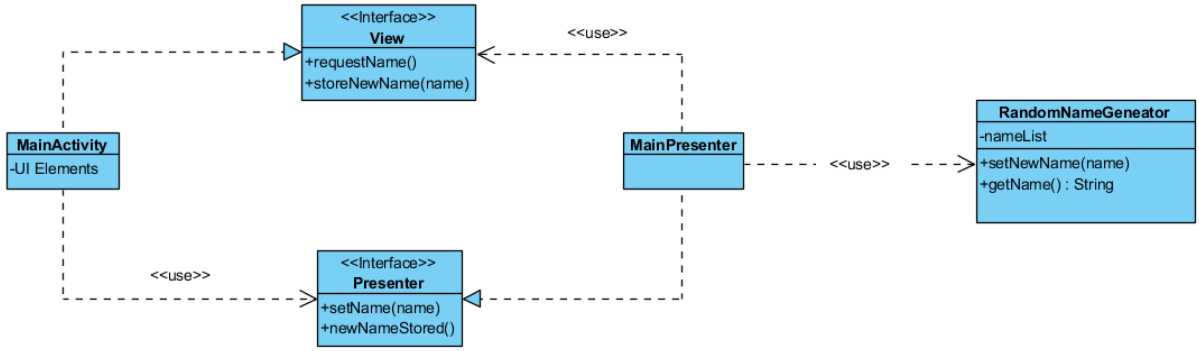

Figura 6: Modello architetturale

Risulta, in questo contesto, interessante mostrare l'implementazione delle funzioni chiave per la corretta implementazione del pattern. Innanzitutto occorre che le classi MainActivity e MainPresenter adottino ed implementino le interfacce relative; successivamente è necessario che la View possegga un riferimento ad un oggetto di tipo Presenter e che il Presenter possegga un riferimento ad un oggetto di tipo View. In questo modo l'Activity e il Presenter possono utilizzare vicendevolmente i servizi offerti.

Alla pressione del bottone "*Tell me a name*" la View deve aggiornare la TextView con un nome proveniente dal Model, fornitole dal Presenter, dunque l'implementazione della callback sul bottone sarà la seguente:

```
1 private void getNameCallback(){
2   presenter.requestName();
3 }
```

Il presenter riceve la richiesta e dovrà interagire col Model per soddisfarla; una volta ottenuto il valore ricercato dalla View, dovrà inviarlo ad essa utilizzando la funzione *setName* dell'interfaccia View. Il codice della funzione *requestName* è il seguente:

```
1 @Override
2 public void requestName() {
3   String name = model.getName();
4   view.setName(name);
5 }
```

La View, dunque, dovrà implementare la funzione utilizzata dal Presenter per inviarle il dato che ha richiesto. In tale funzione, quindi, la View conosce il valore da posizione nella TextView e quindi può settare quest'ultima. Il codice è il seguente:

```
1 @Override
2 public void setName(String name) {
3   this.nameTextView.setText(name);
4 }
```

La seconda delle due funzionalità implementate riguarda non più una lettura dal Model, bensì una scrittura. Come in precedenza, verrà dapprima mostrato il codice della callback del bottone "*Save this name!*":

```
1 private void setNameCallback(){
2   presenter.storeNewName(newNameEditText.getText().toString());
3 }
```

Dunque per soddisfare la richiesta dell'utente, la View si rivolge al Presenter fornendogli il nome scritto nella EditText e chiedendogli di intercedere con il Model al suo posto. Il Presenter, dunque, implementa il seguente servizio di *storeNewName* in cui in primo luogo aggiorna il model inserendo l'elemento in lista ed in secondo luogo notifica la View che l'operazione è andata a buon fine:

```
@Override
public void storeNewName(String name) {
  model.setNewName(name);
  view.newNameStored();
}
```

La View, dunque, dovrà implementare la funzione di *newNameStored* per sapere quando potrà notificare l'utente. Il corpo di questa funzione prevede la comparsa di un Toast message sulla UI:

```
@Override
public void newNameStored() {
  if(toast != null) toast.cancel();
  toast = Toast.makeText(this, newNameEditText.getText().toString() + "
      stored succesfully", Toast.LENGTH_SHORT);
  toast.show();
}
```

In figura sono mostrati due screenshot dell'applicazione realizzata.

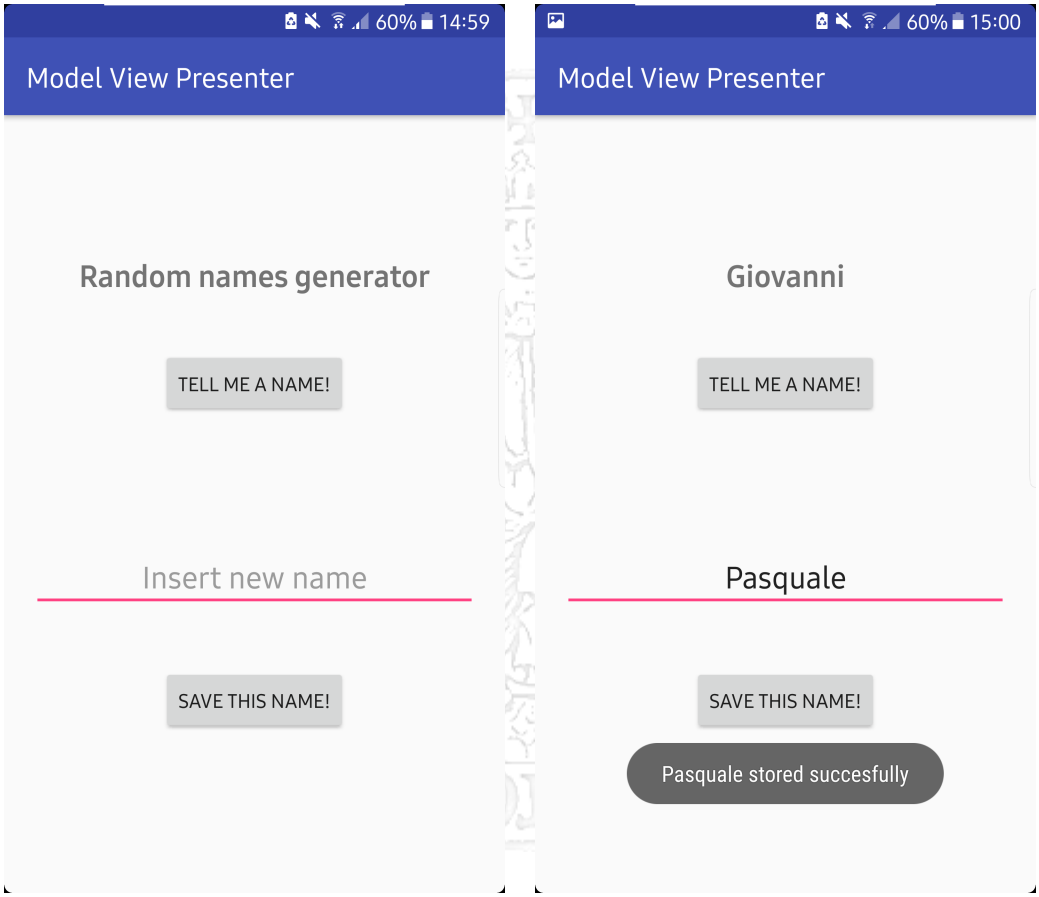

Figura 7: Screen app di esempio

Bibliografia

[1] Google. Android architecture blueprints [beta]. *Google*, 2016.

[2] Tian Lou. *A comparison of Android Native App Architecture MVC, MVP and MVVM*. PhD thesis, Aalto University, School of Science Master's Programme in ICT Innovation, 2016.

[3] Fatima Sabir M. Rizwan Jameel Qureshi. A comparison of model view controller and model view presenter. Technical report, Hailey College of Commerce, University of the Punjab, Lahore, Pakistan, 2013.

[4] Eric Maxwell. The mvc, mvp, and mvvm smackdown. *https://academy.realm.io/*, 2016.

[5] Mike Potel. Mvp: Model-view-presenter. the taligent programming model for c++ and java. *IBM*, 1996.