Relazione
Movie Magazine

Corso di Programmazione a oggetti
2014/2015

Gorbenko Ivan

Indice

1 Analisi

In questa sezione verranno descritti i requisiti dell'applicazione, suddivisi nei requisiti di base che il progetto dovrà sicuramente implementare, mentre le funzionalità extra sono brevemente descritte nel paragrafo 1.2. Le funzionalità richieste e quelle extra sono state suddivise con un preciso criterio. Le funzionalità richieste lavorano solo in modalità di lettura delle informazioni, e che quindi necessitano di essere realizzate prima, mentre le funzionalità extra aggiungeranno la possibilità di personalizzare l'esperienza con l'applicazione in quanto offriranno funzioni per creare, inserire e modificare contenuti all'utente.

1.1 Requisiti

L'applicazione che si vuole realizzare dovrà permettere di accedere a informazioni riguardanti il mondo del cinema. L'utente dovrà avere la possibilità di scegliere il criterio con cui visualizzare i film, come ad esempio i film attualmente più popolari, i più votati, prossimamente e attualmente nei cinema. Inoltre dovrà essere disponibile una funzionalità che permette di navigare il catalogo dei film, filtrarli e ordinarli su diversi parametri come ad esempio anno di uscita, genere e altro. L'utente, navigando il catalogo, potrà visualizzare i dettagli di un film da lui selezionato, ovvero, la trama, il cast, la media dei voti, il numero di voti del film e altri

dettagli. Nel dettaglio del film inoltre dovrà essere possibile, navigare l'elenco del cast, selezionare una persona per avere un dettaglio di quella persona, come ad esempio la sua biografia e filmografia. Infine sarà possibile fare ricerche nel catalogo sulla base del testo inserito dall'utente.

1.2 Requisiti extra

Le funzionalità extra verranno realizzate se rimarrà tempo disponibile, una volta completate quelle richieste.

Tra queste, prima verrà realizzata la modalità di accesso guest che permetterà solamente di votare i film e avere l'elenco dei film votati.

Successivamente, sempre tempo permettendo, sarà realizzata la funzione di login con il proprio account, che estenderà l'accesso in modalità guest, con le funzionalità che permetteranno di interagire con i dati associati con l'account. Gli utenti che utilizzeranno il proprio account, potranno votare i film (che in questo caso verranno registrati con il loro account personale), aggiungere e rimuovere film ai preferiti, aggiungere e rimuovere film dalla propria watchlist e infine creare le proprie liste di film con la possibilità di aggiungere e rimuovere film a queste.

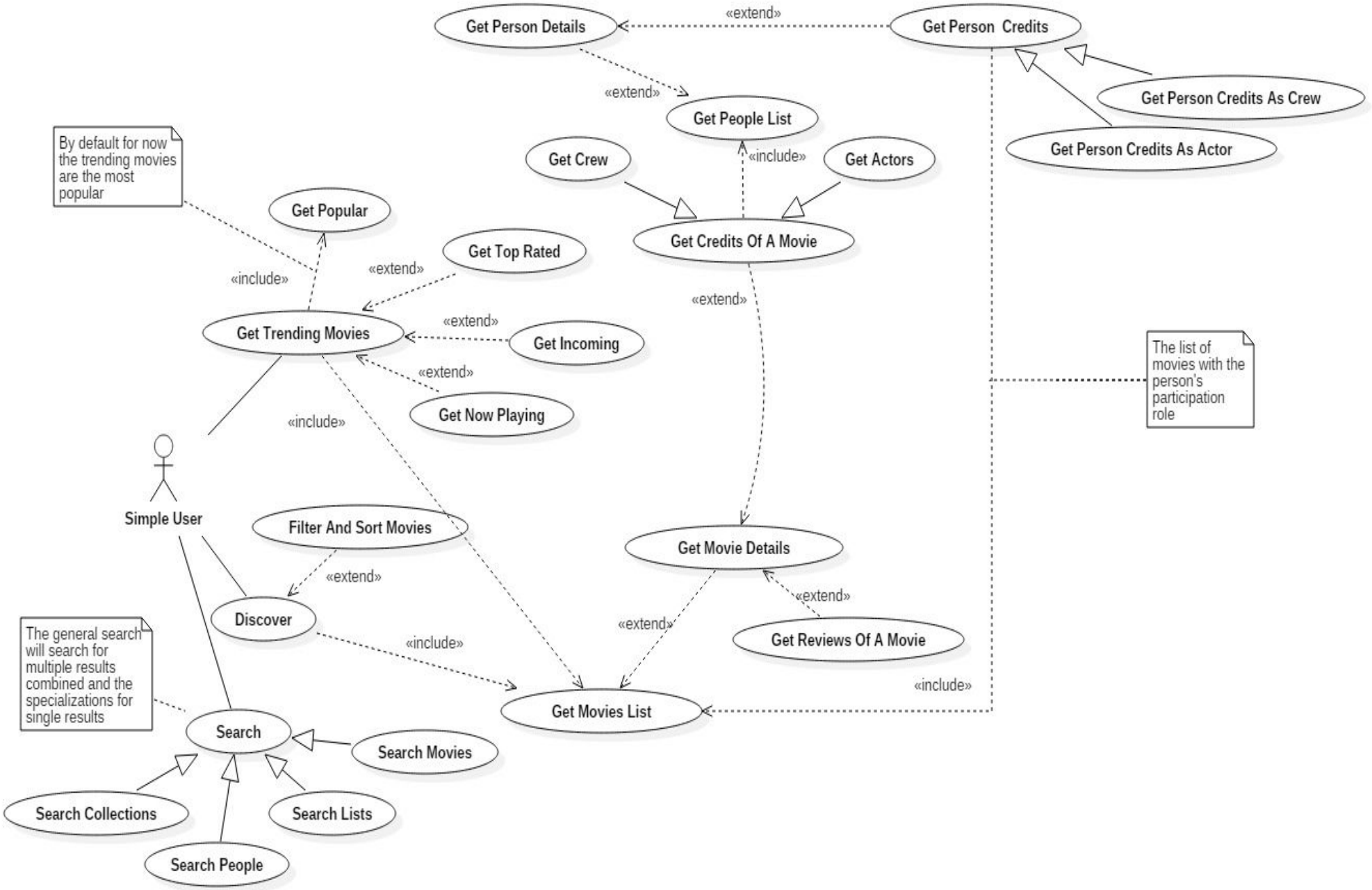
Le sessioni dell'utente dovrebbero persistere a fronte di diversi utilizzi dell'applicazione, indipendentemente dal tipo della modalità di accesso utilizzata dall'utente.

1.3 Casi d’uso

Nella figura 1, il diagramma dei casi d’uso mostra le funzionalità principali dell’applicazione. Il caso d’uso “Get Trending Movies” rappresenta la funzionalità che permetterà di mostrare all’utente i film in base al tipo di trend che sceglie. Mentre il caso d’uso “Discover” permetterà all’utente di sfogliare il catalogo dei film ed eventualmente ordinarli e filtrarli. L’ultimo caso d’uso “Search”, invece permette di eseguire ricerche miste ed eventualmente specifiche.

I casi d’uso rimanenti invece rappresentano l’evoluzione dell’applicazione a partire dai casi d’uso appena descritti. Ad esempio, indipendentemente dal fatto che l’utente stia guardando i trending film o sta semplicemente sfogliando il catalogo, si tratta comunque visualizzare un elenco di film e in entrambi i casi deve essere possibile avere un dettaglio del film, dal quale poi dovrà essere possibile anche avere un elenco delle persone che hanno partecipato al film, inoltre l’utente potrebbe volere avere un dettaglio di una persona e cosi via.

Il diagramma oltre a organizzare in blocchi le principali funzionalità, da anche un’idea di come potrebbe essere strutturata la navigazione dell’applicazione e di come riuscire a organizzare le funzionalità per riuscire a riutilizzarle.


Get Person Details
«extend»
Get Person Credits
Get Person Credits As Crew
Get Person Credits As Actor
«extend»
Get People List
«include»
Get Crew
Get Actors
By default for now the trending movies are the most popular
Get Popular
Get Credits Of A Movie
Get Top Rated
«include»
«extend»
«extend»
Get Trending Movies
«extend»
Get Incoming
«extend»
Get Now Playing
«include»
The list of movies with the person's participation role
Simple User
Filter And Sort Movies
Get Movie Details
«extend»
«extend»
Discover
«extend»
Get Reviews Of A Movie
«include»
«include»
The general search will search for multiple results combined and the specializations for single results
Get Movies List
Search
Search Movies
Search Collections
Search Lists
Search People

1.3 Problema

Vista la natura della piattaforma Android, non sarà semplice fare un design pulito. I componenti principali necessari per costruire le app Android, come le Activity, tendono a mescolare parti di codice che dovrebbero essere suddivisi tra Controller e View, complicando notevolmente lo sviluppo, l'estensione e la manutenibilità dell'applicazione, trasformando l'Activity in una god class.

Un altro problema legato alla piattaforma sarà come gestire la navigazione dell'applicazione visto che il sistema permette di avere una sola schermata attiva per volta. Sarà necessario mettere in campo una strategia di navigazione efficace e sempre consistente per quanto concerne l'ordine di navigazione delle schermate.

Il successivo problema da affrontare è quello del reperimento dei dati necessari all'applicazione. Si è scelto di utilizzare il servizio offerto da themoviedb.org (tmdb da ora in poi); un sito web mantenuto principalmente dalla comunità degli utenti per quanto riguarda i suoi contenuti. Le API in formato JSON, sono a libero accesso una volta registrati sul sito e fatta richiesta di una API key per utilizzarle. L'interfaccia [1] offerta, fortunatamente è ben documentata, tuttavia per il monte ore disponibile, l'applicazione utilizzerà solo una sua sottoparte data la sua notevole grandezza. Tuttavia non deve essere preclusa la possibilità di estendere l'applicazione per utilizzare tutte le funzionalità offerte dalle API.

Essendo la sorgente dei dati disponibile attraverso una connessione Internet, si cercherà per quanto possibile di minimizzare le richieste inviate quando non necessarie.

Il reperimento dei dati online va inoltre gestito correttamente, visto che un'applicazione Android di base ha solo il main Thread che è anche l'unico autorizzato a modificare lo stato delle componenti dell'interfaccia grafica, questo non deve essere bloccato in alcun modo, pena la terminazione forzata dell'applicazione da parte del sistema operativo.

Un possibile lavoro futuro potrebbe inoltre essere la possibilità di integrare nell'applicazione l'accesso anche alle informazioni relative alle serie televisive, visto che anche queste sono disponibili attraverso la stessa interfaccia.

2 Design

In questo capitolo verrà descritta la struttura generale dell'applicazione e i vari moduli che la compongono. Per semplificare l'esperienza utente, si cercherà di renderla il più simile possibile a quella del sito stesso, in quanto risulta consistente, ed efficace.

2.1 Architettura

Per strutturare l'architettura dell'applicazione si cercherà di sfruttare un modello di architettura basato su MVC, chiamato Android Passive MVC [2]. Il modello proposto rende innanzitutto il Model indipendente dalle View, Controller e le Activity, quindi il modello rimarrà completamente indipendente dalla piattaforma Android e sarà riutilizzabile anche da una applicazione Desktop così come qualsiasi altra piattaforma.

Il modello descritto tuttavia non da alcun suggerimento su come strutturare la navigazione, per realizzare tale funzionalità si utilizzerà in combinazione al modello Hierarchical MVC[2][3]. HMVC risulta difficile da applicare in Android perché prevede che siano i Controller, attraverso una chain of responsibility, a gestire la navigazione dell'interfaccia grafica. Per cercare di limitare e confinare il più possibile le dipendenze con Android, si sposterà la gestione della navigazione delle viste nei componenti di Android come i Fragment e le Activity.

Far gestire la navigazione alle viste implica che ogni vista debba conoscere le possibili viste a cui si può andare da essa. Le viste essendo dei Fragment, dovranno creare i loro Controller perché la vista è il primo componente della triade MVC che viene lanciato.

Utilizzando questo tipo di approccio, si creano delle coppie View-Controller, dove la View quando viene lanciata, crea il proprio Controller e gli chiede di caricare i dati, il Controller una volta che li mostra nella vista a cui è associato.

Nel modello HMVC, i Controller soddisfano le richieste di dati e di navigazione attraverso la catena di responsabilità, mentre nell'approccio adottato i Controller saranno responsabili solamente di gestire le richieste di dati da parte della vista a cui sono associati.

Spostando la responsabilità di gestire la navigazione sulle viste, comporta un altro problema: i Controller hanno comunque bisogno di comunicare tra loro ma non possono farlo direttamente perché in ogni momento ci sarà attiva solamente una vista alla volta, mentre tutte le viste precedenti a quella attualmente attiva, essendo dei Fragment potrebbero essere distrutte in qualunque momento e insieme a loro verrebbero distrutti anche i loro Controller. Per risolvere questo problema è necessario delegare la vista anche a fornire tutti i dati necessari alle sue prossime viste nel momento in cui dovranno essere mostrate che a loro volta li inoltreranno ai propri Controller quando li dovranno creare.

Le Activity rappresentano la vista padre per tutte le viste e saranno il loro contenitore poiché sono le Activity che hanno il controllo sui Fragment attraverso il

FragmentManager così come la gestione dello stack dei Fragment. I Fragment hanno modo di richiedere il riferimento all'Activity che li contiene, quindi i Fragment possono chiedere di mostrarne un altro. Quindi se una vista ha bisogno di mostrarne un'altra, costruisce il corrispondente Fragment passandogli i dati necessari al suo Controller, e chiede alla sua Activity di mostrarlo. Il diagramma in figura 2 mostra l'architettura generale dell'applicazione.

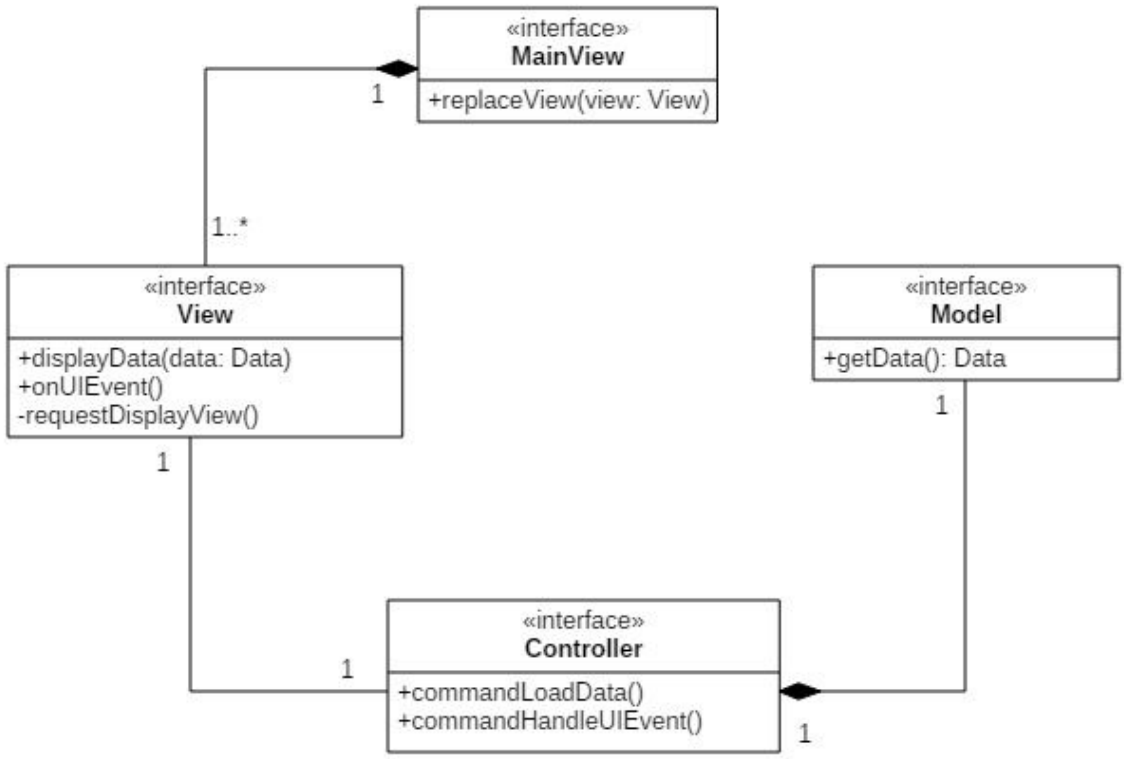


Figura 2: MainView rappresenta la vista padre e fa da contenitore a tutte le View, una View quando ha

bisogno di mostrare un’altra View, chiede alla propria MainView di mostrarla. Ogni View ha il suo Controller che ha il compito di interfacciarsi con il Model e di rispondere agli eventi della View.

2.2 Design dettagliato

In questa sezione verranno descritti singolarmente i componenti dell'applicazione e discusse le scelte di progettazione per ogni componente del pattern MVC.

2.2.1 Model

Il modello verrà realizzato con delle interfacce che rappresentano il modello dei dati di tmdb e le classi che le implementeranno saranno dei POJO che serviranno per la deserializzazione da JSON. Quindi il modello risulta abbastanza semplice da implementare e lo si può osservare nella figura 3 a pagina seguente.

Una particolare attenzione va prestata alle immagini. La documentazione delle API dice che l'url di ogni immagine è formato dal base url, una dimensione e un path relativo. Il servizio offre la stessa immagine in diverse dimensioni semplicemente cambiando la parte di dimensione dell'url. Ogni metodo che restituisce oggetti con immagini, contiene solo il path relativo dell'immagine. Ciò che si può notare in comune a ogni immagine è che hanno tutte lo stesso base url, delle dimensioni e un path relativo, tuttavia la parte della dimensione dell'url, dipende dal tipo dell'immagine. Si può osservare in figura 4 il diagramma delle classi che modella questi aspetti.

Le API hanno un metodo "configuration" che restituisce la configurazione delle immagini che comprende il base url da utilizzare e le dimensioni per ogni tipo di immagine.

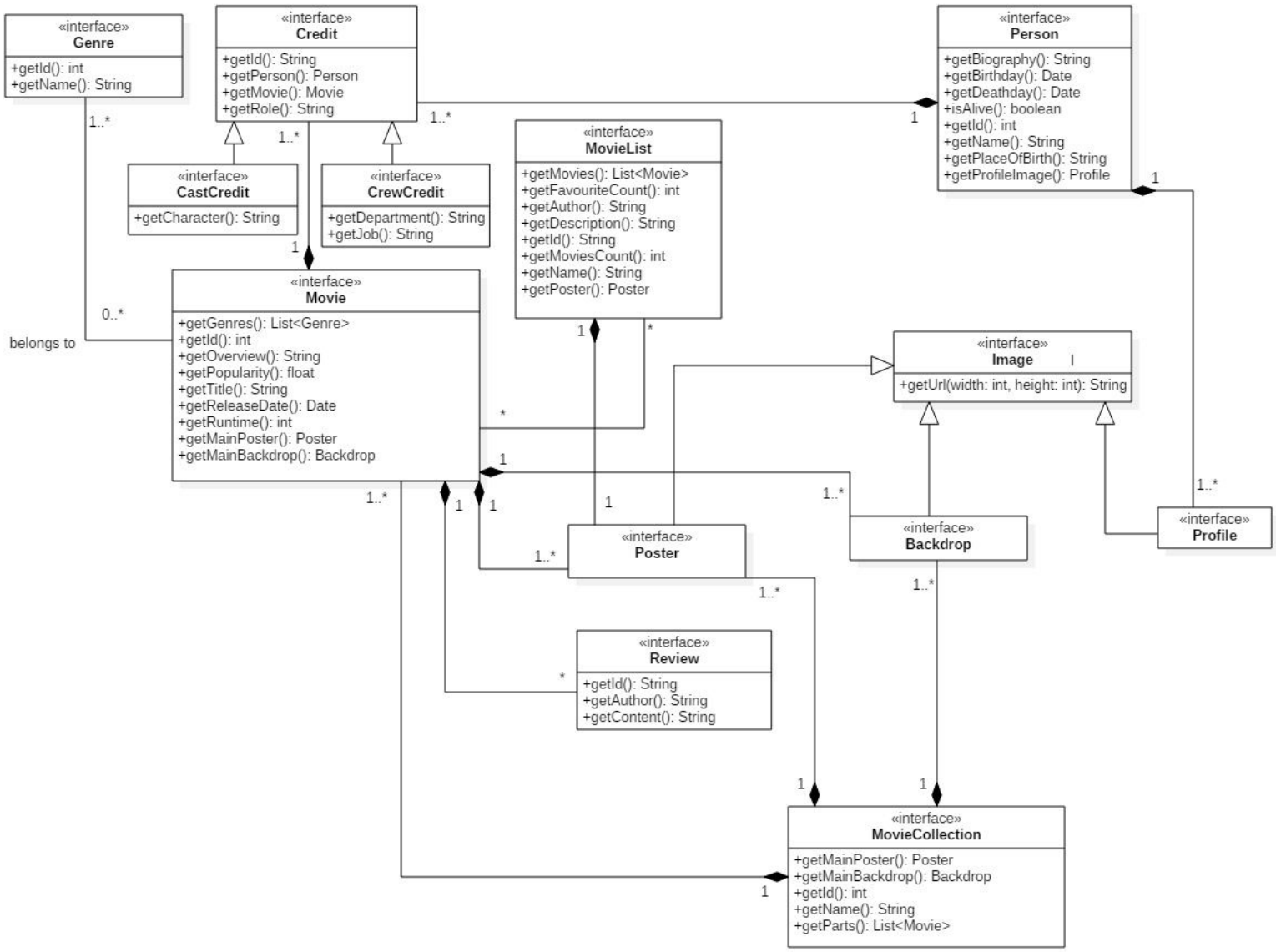


Figura 3: Dallo schema si può osservare che l'entità chiave sia Movie, alla quale possono essere associati dei Genre. Movie ha dei Poster e dei Backdrop che sono comunque immagini. Un Movie potrebbe avere delle Review, appartenere a una Collection che a sua volta ha dei Poster e dei Backdrop. Un Movie potrebbe essere in diverse MovieList, e un MovieList ha un Poster. Un Movie ha dei Credit che sono associati a loro volta a dei Person. Un Person ha dei Profile e dei Credit.

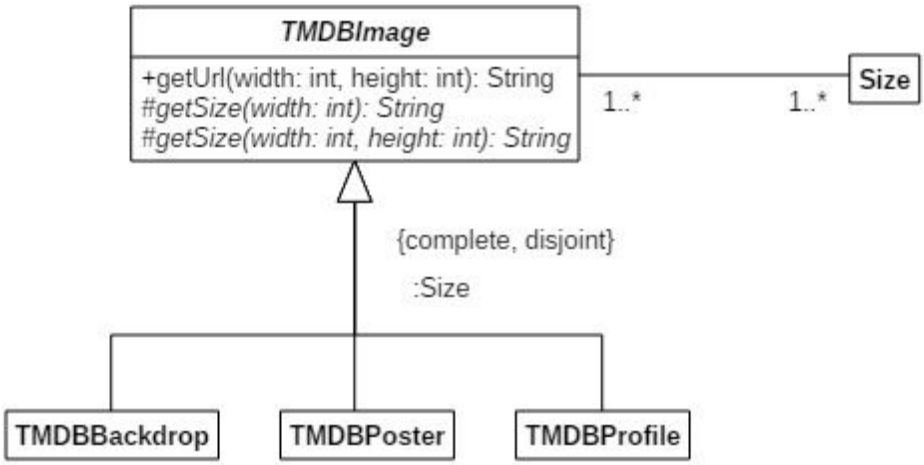


Figura 4: Ogni immagine deve poter costruire il suo url in base alla dimensione richiesta, però la dimensione dipende dallo specifico tipo. Quindi una classe astratta che rappresenta un'immagine può utilizzare il template method per prendere la giusta dimensione in base al tipo. Alle sottoclassi di TMDBImage sarà sufficiente implementare i metodi astratti getSize che restituiranno la dimensione. Sapendo che ogni immagine ha delle dimensioni che dipendono dal suo tipo, l'insieme delle dimensioni è stato partizionato in base al tipo con il meccanismo del powertype.

Per gestire questi dati è stata costruita la classe ImageConfigurationFactory che applicando il pattern factory method inizializza le dimensioni delle immagini. Inoltre ogni sottotipo di immagine ha sempre le stesse dimensioni, quindi queste possono essere condivise tra tutte le istanze di ogni sottotipo utilizzando il pattern flyweight, trasformando così la classe ImageConfigurationFactory in una factory di flyweight evitando un inutile spreco di memoria.

La deserializzazione dei modelli è stata gestita semplicemente costruendo dei wrapper degli oggetti del modello che ricalcano la forma della risposta per un particolare metodo remoto. Per deserializzare i modelli è stata utilizzata la libreria Gson [4]. I modelli essendo semplici, non richiedono di costruire deserializzatori custom, è sufficiente chiamare i campi degli oggetti deserializzati con lo stesso nome della corrispondente proprietà nella struttura JSON o annotare il campo con il nome se si vuole chiamare diversamente il campo.

Tuttavia c'è una situazione di ambiguità: la deserializzazione dei Credit dipende dal fatto che siano stati richiesti i Credit di un Movie o di un Person. Innanzitutto ogni Credit è riferito a un Movie e a un Person con il ruolo della persona nel film. Le API offrono due modi diversi per accedere ai crediti, dal lato di Movie si hanno i dati dei Credit e i dati delle Person a cui sono riferiti, per quel Movie, mentre dal lato di Person si hanno i dati dei Credit e i dati dei Movie a cui fanno riferimento. In questo caso è necessario un

deserializzatore custom perché in un caso vanno deserializzati i dati di un Person mentre nell'altro sono dei Movie. Questo problema verrà gestito con il pattern strategy perché è noto a priori il lato da cui si accede ai Credit, quindi è sufficiente selezionare la corretta strategia di deserializzazione quando si fa richiesta di Credit.

Alcuni metodi delle API restituiscono elenchi di oggetti, suddivisi per pagine. Visto che sarà necessario implementare la paginazione, è stata costruita la classe astratta PagedResult che contiene i dati necessari per implementare la navigazione per pagine. Quindi per ogni metodo che restituisce risultati a pagine, il corrispondente wrapper per deserializzarli dovrà estendere questa classe e aggiungere solo i campi che conterranno i particolari risultati.

2.2.2 View

Le viste dell'applicazione saranno suddivise in sezioni. Ad ogni sezione corrisponderà una Activity che avrà un contenitore per le viste di quella sezione. Ad ogni vista corrisponderanno uno o più Fragment. In figura 5 si può osservare lo schema delle principali interfacce che tutte le viste estenderanno.

Le viste in generale, sono perlopiù liste di elementi del modello, e dettagli degli stessi che verrano mostrati al click su un elemento della lista.

Le Activity di ogni sezione dovranno occuparsi di:

- ❏ mostrare la vista "home" della loro sezione all'avvio
- ❏ gestire il menu di navigazione laterale che permette di passare da una sezione ad un'altra
- ❏ gestire la navigazione avanti e indietro delle viste con il back stack dei Fragment

Un problema è sorto nel momento in cui è stata implementata la vista dei filtri utilizzando un DialogFragment: quando questo è aperto, se l'utente preme indietro, verrebbe rimossa anche la vista della sezione perché la pressione del pulsante indietro corrisponde a un pop dallo stack dei Fragment. Per risolvere questo problema è stato necessario aggiungere l'interfaccia StickyBackStack all'Activity della sezione in cui è possibile filtrare i film. Questa

interfaccia serve solamente a notificare l'Activity quando deve ignorare la pressione del pulsante indietro. In figura 6 è possibile vedere il diagramma che mostra le Activity utilizzate nell'applicazione.

Le viste dell'applicazione sono state implementate utilizzando dei Fragment. Sono stati realizzati dei Fragment che fanno da contenitore per le singole viste per realizzare delle viste tabbed. Come già detto in precendenza le viste sono semplici elenchi di elementi e dettagli degli stessi. In figura 7 sono mostrate le interfacce delle varie ListView. Le viste di dettaglio sono delle semplici interfacce che estendono View e aggiungono il metodo display che prende in ingresso un'interfaccia di un modello e lo mostra all'utente, ad esempio MovieDetailView mostra le informazioni di un film prendendo in ingresso un Movie.

I Fragment che implementano le interfacce delle varie ListView e *DetailView, vengono creati con il pattern factory method per imposizione di Android perché se vengono distrutti quando sono inattivi nel back stack devono poter essere ricreati in automatico. Un altro problema causato dalla natura dei Fragment è che i Controller che comandano i Fragment non possono avere dipendenze tra loro perché se un Fragment inattivo viene distrutto, verrebbe distrutto anche il suo Controller. Per ovviare a questo problema i factory method dei Fragment prendono in ingresso un intero che rappresenta il tipo del Controller che il Fragment deve istanziare. Nel momento in cui il Fragment verrà creato, dovrà istanziare il proprio Controller con l'id che gli è stato passato come

parametro, i dettagli di come ciò avverrà sarà descritto nel prossimo capitolo. Tuttavia alcuni factory method prendono in ingresso direttamente anche interfacce di Controller già istanziati perché in alcuni casi come viste tabbed da due tab è possibile comandare i due Fragment con un unico Controller. Ciò è possibile perché Android, a quanto pare, quando vengono utilizzate viste tabbed, appena viene visualizzato uno dei tab viene immediatamente precaricato anche il tab affianco. In previsione del possibile passaggio alla vista subito affianco, questa viene precaricata in modo anche da permettere l'effetto continuo che si vede quando si passa da un tab all'altro trascinando il dito nella direzione del prossimo tab.

Visto che tutti Fragment sono considerati come viste è stato conveniente costruire un Fragment di base che implementa l'interfaccia View per fornire tutte le operazioni di base. In figura 8 si può osservare il diagramma delle classi dei Fragment.

La navigazione da una vista all'altra è stata realizzata in maniera molto semplice: se siamo in una qualunque ListView, e l'utente clicca su un elemento della lista, il Fragment che realizza la ListView crea il corrispondente Fragment di dettaglio con l'id dell'elemento cliccato e richiede alla sua Activity di sostituire il Fragment attuale con quello che gli viene passato, aggiungendolo al back stack per poter sempre ritornare indietro.

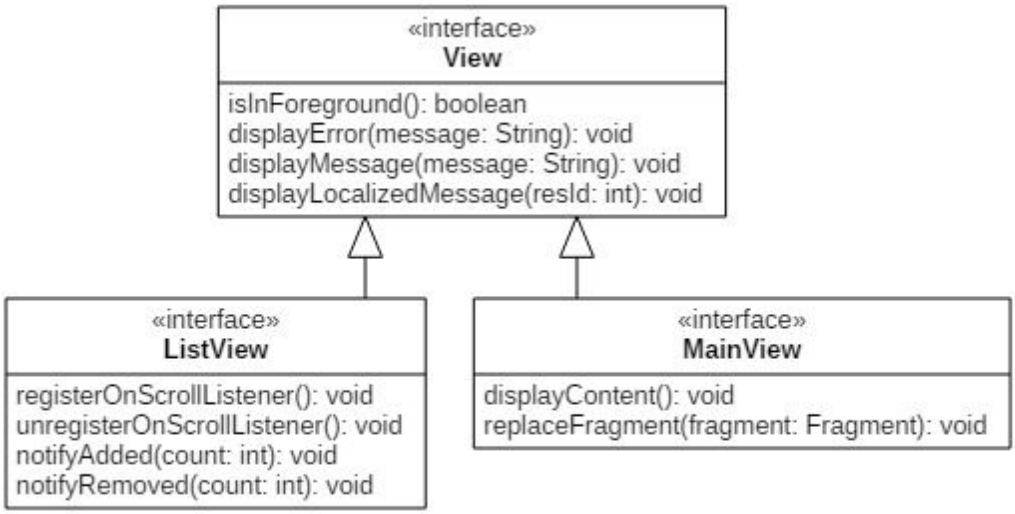


Figura 5: View rappresenta la radice della gerarchia di tutte le viste e fornisce solamente ciò che ogni vista dovrà fare. MainView invece verrà implementata dalle Activity che dovrà mostrare il contenuto della sua sezione e fornire la funzionalità di sostituire la vista attuale nel suo contenitore con la vista richiesta per realizzare la navigazione da una vista all'altra. ListView rappresenta l'interfaccia di base di tutte le viste che mostreranno una lista di elementi, i metodi per registrare e de-registrare uno scroll listener servono nel caso in cui gli elementi della lista sono fornite a pagine e quando l'utente arriva in fondo a una pagina, la vista dovrà chiedere al proprio controller di caricare altri elementi, se ce ne sono. I metodi notify* sono necessari a notificare alla vista che sono stati aggiunti/rimossi elementi alla lista ed è necessario il refresh della vista.

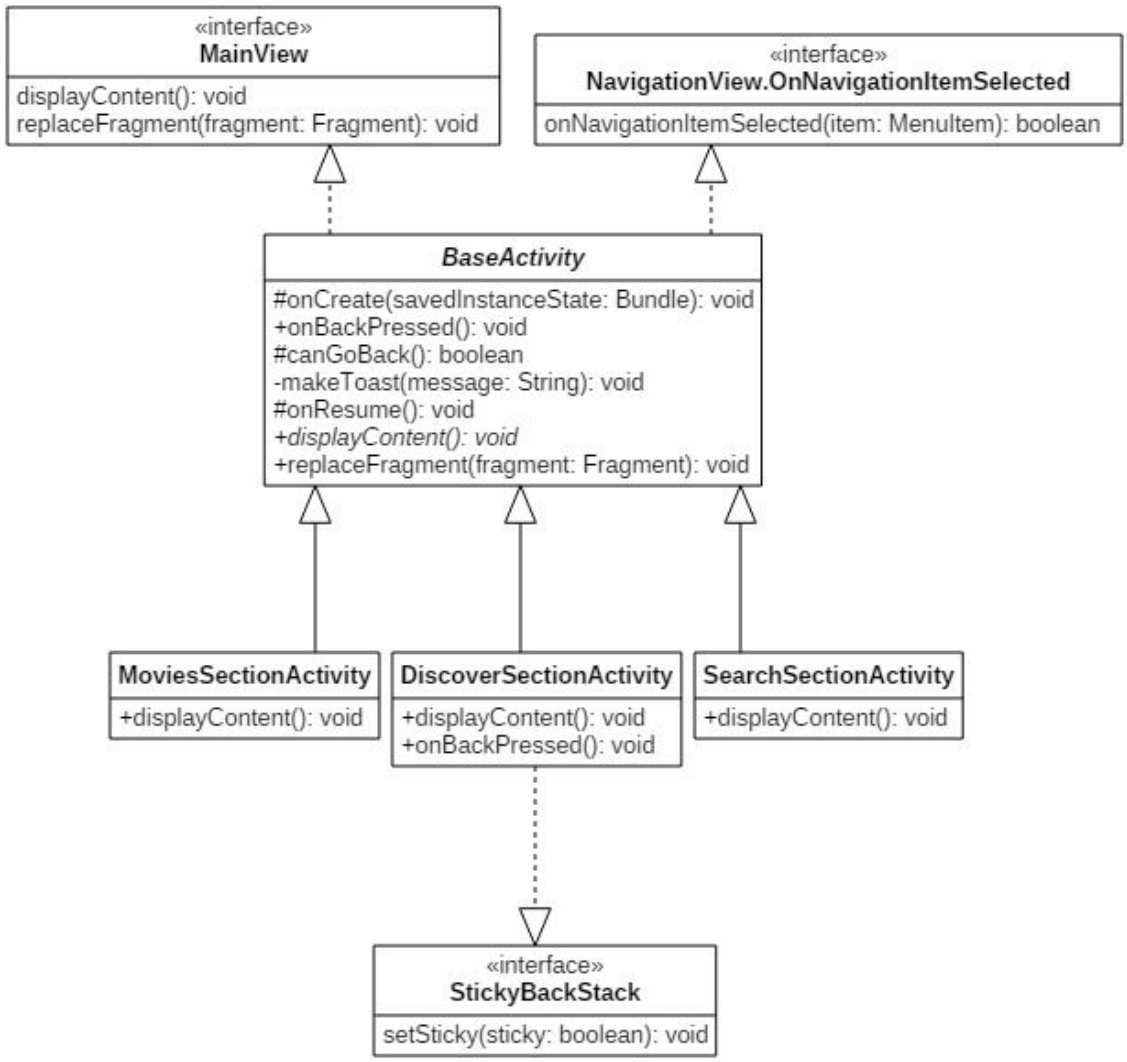


Figura 6: La classe astratta BaseActivity implementa tutte le funzionalità che le Activity di ogni sezione dovranno sicuramente fare. L'Activity di ogni sezione per mostrare la propria "home" dovrà implementare il solo metodo displayContent(). Ogni Activity dovrà mostrare la propria home appena diventa visibile all'utente, ed è stato applicato il template method pattern per realizzare ciò direttamente in BaseActivity in onResume(). Il menu di navigazione laterale, è gestito direttamente in BaseActivity perché non è altro che l'avvio di una delle sue sottoclassi. Le sottoclassi di BaseActivity rappresentano le sezioni dell'applicazione dove MoviesSectionActivity mostra i "trending movies", SearchSectionActivity permette di cercare inserendo testo e infine DiscoverSectionActivity dove è possibile filtrare i film.

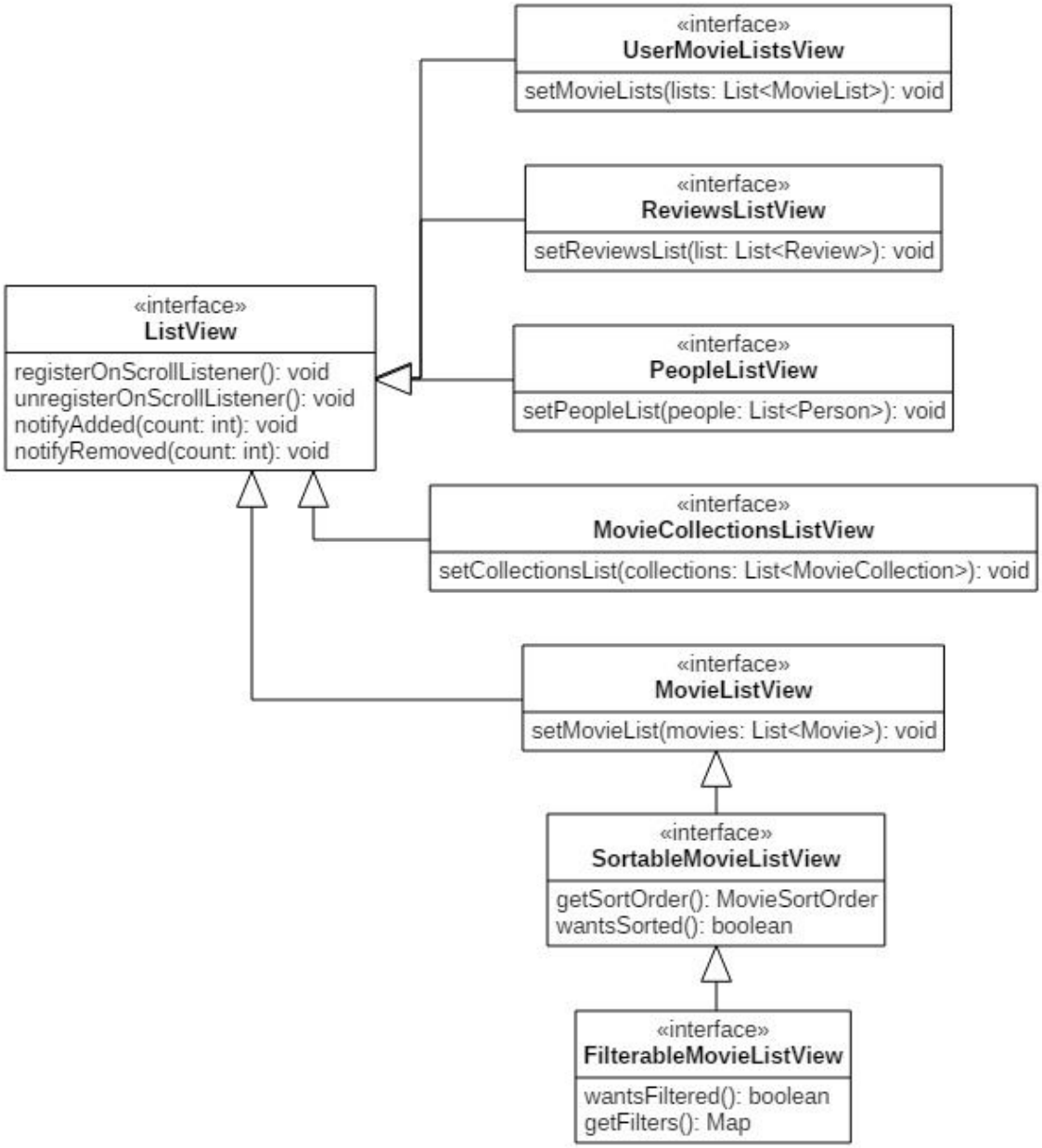


Figura 7: ListView è l'interfaccia di base da cui estendono tutte le viste che presentano elenchi di elementi. Ogni specializzazione contiene un metodo per impostare la lista degli elementi del modello da visualizzare. La lista viene impostata dal controller della vista una volta che gli elementi saranno scaricati e successivamente notificherà la vista che sono stati aggiunti/tolti degli elementi per aggiornarla. Le estensioni di MovieListView sono le viste che permetteranno di ordinare e filtrare la lista dei film visualizzati.

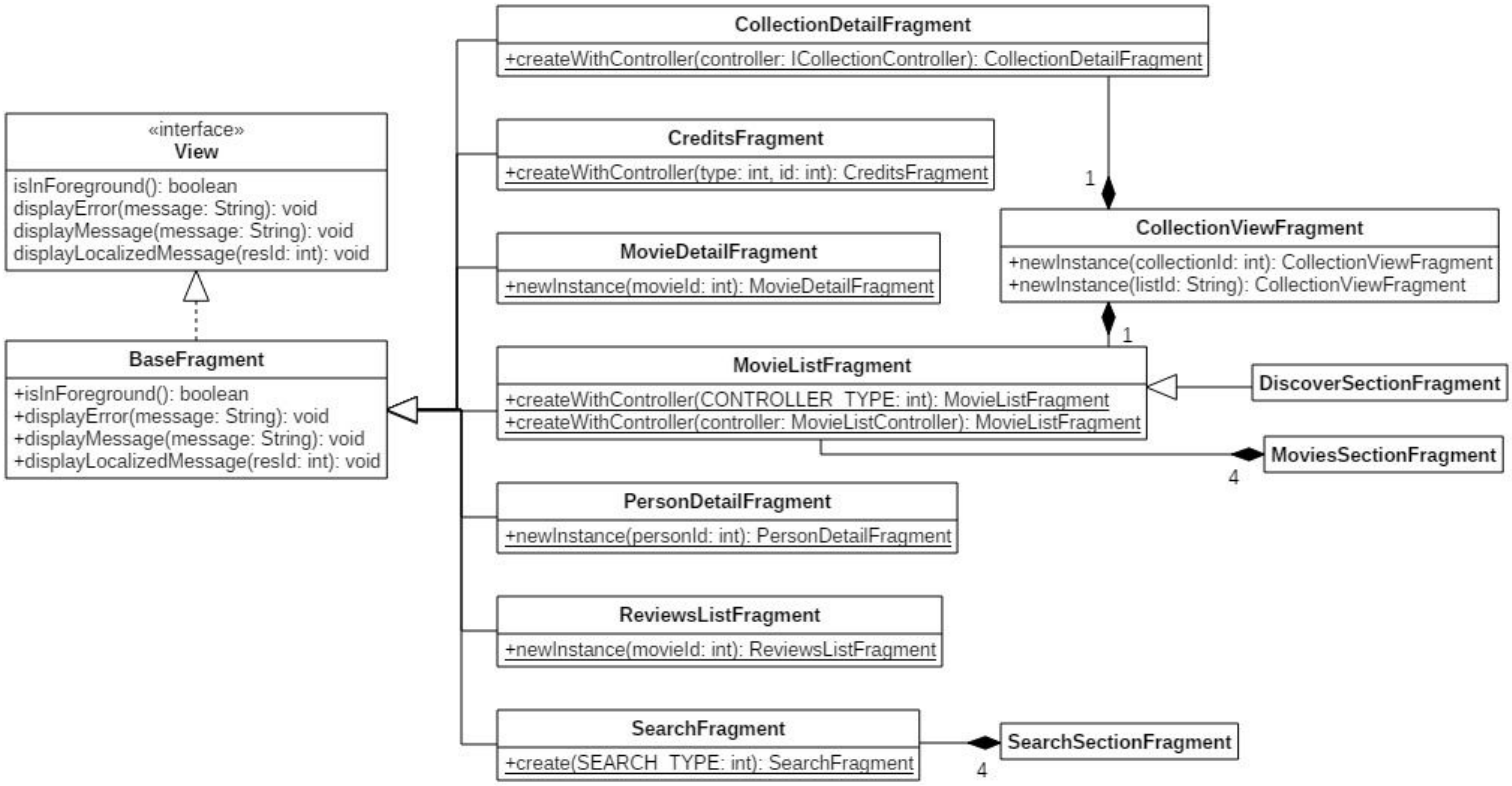


Figura 8: BaseFragment è la radice della gerarchia dei Fragment ed implementa l'interfaccia View. Ogni Fragment implementa la corrispettiva interfaccia della gerarchia di View o ListView. Per semplicità i Fragment che realizzano le viste tabbed saranno omessi perché semplicemente creano i Fragment in figura, con varie combinazioni di Controller. Tuttavia alcuni di questi Fragment realizzano le viste di 2 delle 3 sezioni dell'applicazione ed è necessario nominarli. MoviesSectionFragment mostra i trending film utilizzando 4 tab dove ad ogni tab corrisponde un MovieListFragment ed ognuno di essi viene comandato dallo stesso tipo di Controller anche se ognuno utilizza una strategia diversa per caricarli, ma questo verrà spiegato più in dettaglio nel prossimo capitolo. SearchSectionFragment realizza la sezione di ricerca con 4 tab dove ad ogni tab corrisponde un SearchFragment, in questo caso il tipo di Controller che SearchFragment deve creare è sempre lo stesso ma cambia la strategia di ricerca, nel senso di che cosa si cerca. DiscoverSectionFragment invece è un'estensione di MovieListFragment perché aggiunge la possibilità di poter filtrare e ordinare la lista di film ma il resto del comportamento è identico. CollectionDetailFragment permette di vedere i dettagli sia delle MovieCollection che delle MovieList, visto che entrambi i modelli sono una lista di film a cui sono associate delle informazioni, si è deciso di

evitare di fare 2 Fragment diversi. L'unico inconveniente che ha portato questa scelta è che l'id di una MovieList è una stringa mentre l'id di una MovieCollection è un intero, quindi è stato necessario distinguere i due utilizzando il giusto factory method.

CollectionViewFragment, uno dei contenitori per le viste tabbed, è un caso in cui è stato possibile far comandare più Fragment dallo stesso Controller perché come detto in precedenza se la vista è composta da 2 tab, questi vengono caricati subito entrambi. Il motivo per cui MovieListFragment e CollectionDetailFragment possono essere creati con dei Controller già istanziati è proprio questo: CollectionViewFragment utilizza un CollectionDetailFragment e un MovieListFragment come suoi 2 tab, quindi CollectionViewFragment deve creare il Controller delle 2 viste e creare le 2 viste con lo stesso Controller.

CreditsFragment invece nel suo factory method ha bisogno del tipo di Controller che deve utilizzare e di un id. Questo è stato necessario perché come detto in precedenza, ciò che si vede in questa vista in un caso sono Person con i loro ruoli se l'id passato è quello di un Movie, mentre nell'altro sono dei Movie con i ruoli di una persona se l'id passato è quello di un Person.

I rimanenti Fragment di dettaglio sono molto semplici: vengono creati con l'id dell'elemento di cui si vuole vedere il dettaglio.

2.2.3 Controller

Le viste dell'applicazione hanno bisogno di principalmente due tipi di Controller, uno che carica i dati delle viste di dettaglio e uno che carica gli elementi della ListView. Nel caso in cui gli elementi delle ListView vengano caricati a pagine, il Controller dovrà occuparsi anche di caricare altri elementi nel momento in cui l'utente scorre la lista fino in fondo. In figura 9 è possibile vedere le interfacce dei Controller appena descritti.

Ogni ListView o DetailView richiede di caricare modelli diversi, quindi per ognuna di esse è necessario fare un Controller in grado di caricare i modelli richiesti dalla particolare View. Per semplicità sarà omesso il diagramma dei Controller delle viste di dettaglio in quanto semplicemente estendono Controller ed aggiungono un metodo per legare il Controller alla View, ad esempio IMovieDetailController aggiunge il metodo addPersonDetailView in cui gli viene dato in ingresso la View in cui dovrà mostrare i dati. Stessa cosa avviene per i vari ListController delle ListView, tutte le estensioni di ListController aggiungono un metodo per legarsi alla sua ListView.

Come anticipato nel capitolo precedente, i Fragment dovranno creare il proprio Controller con l'eventuale parametro che gli viene passato. Questo tipo di approccio permette di nascondere l'effettiva implementazione dei Controller ai Fragment in quanto

utilizzeranno solamente l'interfaccia del Controller di cui ha bisogno. Per rendere possibile questo tipo di organizzazione, è stato utilizzato il pattern factory method e si è deciso di centralizzare la creazione dei Controller in due factory diverse, una per i Controller e una per i ListController. I factory method prendono in ingresso gli id dei modelli da caricare, eventualmente insieme ad un intero che rappresenta la strategia utilizzata internamente dal Controller.

Per semplificare lo sviluppo è stata utilizzata la libreria Retrofit[5] che implementa il facade pattern per mascherare le richieste http in semplici chiamate a metodi. Questa libreria, permette di creare dei facade attraverso la costruzione di interfacce, dove ogni metodo è annotato con i nomi dei metodi http da usare quando le chiamate verranno effettivamente eseguite, e i path relativi dei metodi remoti.

Per riuscire ad eseguire correttamente una richiesta attraverso uno dei metodi del facade, è necessario che il facade sia creato dalla classe omonima alla libreria che implementa l'abstract factory pattern per crearli. Tuttavia l'esecuzione di questi metodi remoti può avere buon fine solamente se l'abstract factory è stata opportunamente costruita e configurata con i componenti necessari alla deserializzazione degli oggetti restituiti dai particolari metodi remoti, ed eventualmente particolari configurazioni del client http utilizzato per eseguirli.

I metodi remoti utilizzati dall'applicazione sono stati raggruppati nelle interfacce facade esattamente come

sono raggruppati nella documentazione delle API di tmdb.

Un fattore da gestire con i metodi remoti utilizzati è che questi presentano diversi parametri opzionali che possono variare da metodo a metodo o possono essere anche in comune tra diverse categorie di metodi, un esempio di questi è il codice ISO 639-1 della lingua in cui si vuole ricevere il risultato. Un altro fattore da gestire è l'ambiguità della deserializzazione dei crediti, descritta nei capitoli precedenti.

L'abstract factory che crea i facade permette di costruirla agevolmente, configurandola con il builder pattern, al quale vanno sicuramente passati gli eventuali deserializzatori custom. Opzionalmente il builder può ricevere un http client se sono necessarie configurazioni particolari di client http.

Il client http utilizzato dalla libreria offre un meccanismo molto agevole per manipolare le richieste prima che vengano effettivamente eseguite, in modo da permettere l'eventuale modifica della stessa in maniera dinamica. Questo meccanismo è stato sfruttato per l'aggiunta dei parametri GET opzionali di vari metodi remoti.

Prendendo spunto da [6] si è utilizzato il factory method per incapsulare la costruzione e la configurazione dei facade nella classe ServiceFactory, che viene utilizzata dai Controller dell'applicazione.

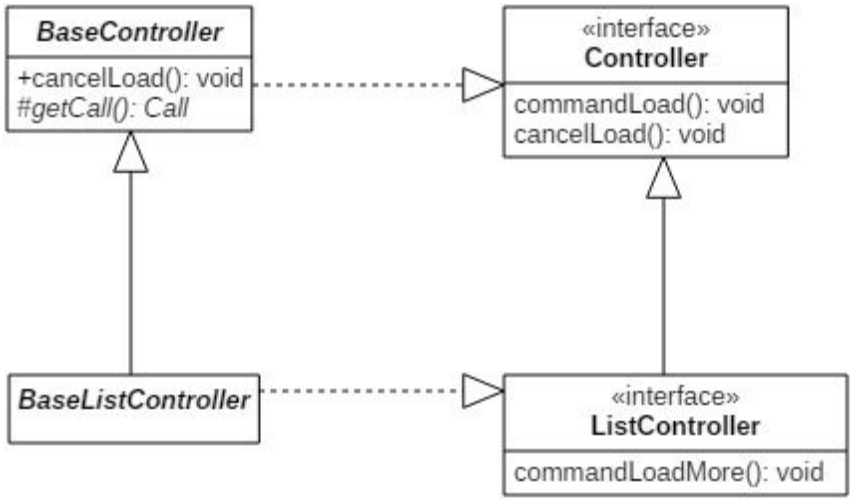


Figura 9: Il Controller di base deve occuparsi di caricare i dati della sua vista, non appena questa li richiede, eventualmente nel caso in cui la vista ha già richiesto il caricamento dei dati ma sta per diventare invisibile, può notificare al Controller di cancellare la richiesta di dati. ListController invece verrà utilizzato per comandare le ListView e nel caso in cui la lista sia paginata, l'eventuale pagina successiva verrà caricata nel momento in cui l'utente arriva in fondo alla ListView. BaseController è la radice della gerarchia delle classi dei Controller e visto che il metodo cancelLoad è necessario a tutti i Controller, questo è stato implementato applicando il template method. BaseListController invece non aggiunge metodi ma serve solamente per imporre l'implementazione dell'interfaccia ListController ed evitare che questa sia usata da sola.

3 Sviluppo

3.1 Testing automatico

Nello sviluppo dell'applicazione sono stati commessi diversi errori inizialmente. Prima di costruire dei semplici wrapper per la deserializzazione, venivano usati molti deserializzatori custom. Una volta costruiti i wrapper, l'unico deserializzatore rimasto in uso è quello dei crediti. Dal punto di vista del testing automatico sono stati testati solo questi wrapper.

Per verificare la correttezza delle richieste effettuate alle API e la corretta istanziazione dei Controller da parte dei Fragment, si è preferito utilizzare l'applicazione a mano perché era necessario anche verificare il corretto passaggio dei parametri da un Fragment all'altro, da cui dipende la corretta istanziazione dei Controller stessi.

L'applicazione è stata sviluppata principalmente testando su Nexus 5 (API 23) con qualche sporadico test su Oneplus One (API 23). Per non limitare la compatibilità dell'applicazione, sono state utilizzate le librerie di supporto di Android.

3.2 Note di sviluppo

Per evitare di scrivere il codice necessario per scaricare le immagini, è stata utilizzata la libreria Picasso[7] che oltre ad assolvere a questo compito in una linea di

codice, è in grado anche di fare caching delle immagini caricate.
Molti spunti su come si utilizza Retrofit sono stati presi dal blog [8] che mostra la varie funzionalità della libreria con diversi esempi.

Per realizzare il menu di navigazione laterale è stato preso spunto da [9].

4 Commenti finali

Non essendo considerato nel monte ore il lavoro fatto per Android, questo è stato trascurato da alcuni punti di vista che non precludono il corretto funzionamento basilare dell'applicazione.

Si vuole precisare che le parti Android trascurate saranno oggetto di lavori futuri. Di seguito sono elencate le parti che saranno sviluppate o migliorate:

❏ È stato duplicato il codice dei Fragment contenitori dei tab anche se è possibile costruire un unico contenitore dinamico in grado di adattarsi a tutte le viste tabbed. Ciò è stato fatto per il puro scopo di separare le varie viste e rendere più leggibile il codice in questa prima versione.

❏ Le viste in realtà mostrano solo una parte del modello utilizzato per velocizzare la costruzione dei layout di queste, tuttavia sarà sufficiente modificare solamente questi e ampliare di conseguenza le implementazioni delle viste.

❏ In alcuni casi possono capitare dei crash dell'applicazione se si preme il tasto home/multitasking e si ritorna all'applicazione. Questo è dovuto ad una non sempre corretta gestione del ciclo di vita dei Fragment/Activity. In alcuni casi addirittura si è notato che alcune callback delle macchine a stati dei Fragment/Activity, vengono eseguite più volte in sequenza prima di passare alla callback successiva, e

ciò ha costretto i Controller a dover fare più controlli prima di agire sull'interfaccia grafica.

L'architettura scelta per l'applicazione, permetterà di fare agevolmente le modifiche appena elencate perché saranno isolate nei Fragment o nelle Activity. L'estensione all'utilizzo completo delle API sarà semplice perché sarà sufficiente aggiungere coppie View-Controller ed integrare i Fragment che realizzano le View nello schema di navigazione utilizzato.

Bibliografia

[1] Travis Bell. The Movie Database API. http://docs.themoviedb.apiary.io/

[2] Karina Sokolova, Marc Lemercier, Ludovic Garcia. Android Passive MVC: a Novel Architecture Model for Android Application Development. *PATTERNS 2013, The Fifth International Conferences on Pervasive Patterns and Applications*, 27 maggio 2013, https://www.thinkmind.org/download.php?articleid=patterns_2013_1_20_70039

[3] Jason Cai, Ranjit Kapila and Gaurav Pal. HMVC: The layered pattern for developing strong client tiers. JavaWorld, 21 luglio 2000. http://www.javaworld.com/article/2076128/design-patterns/hmvc--the-layered-pattern-for-developing-strong-client-tiers.html

[4] Gson https://github.com/google/gson

[5] Retrofit https://github.com/square/retrofit

[6] Marcus Pöhls. Retrofit — Getting Started and Create an Android Client. https://futurestud.io/blog/retrofit-getting-started-and-android-client

[7] Picasso https://github.com/square/picasso
[8] Future Studio.
https://futurestud.io/blog/tag/retrofit
[9] Android Believe.
https://androidbelieve.com/navigation-drawer-with-swipe-tabs-using-design-support-library/