Relazione
Just in Train

Corso di Programmazione a oggetti 2014/15

Giunta Alberto
Mazzini Lisa

26 febbraio 2015

Indice

Capitolo 1 - Analisi

1.1 Requisiti

L'applicazione sviluppata vuole fornire all'utente un servizio veloce e semplice per tenersi aggiornato sull'andamento e sugli orari dei treni in circolazione sulla rete Trenitalia; oltre alle classiche funzioni di ricerca per numero treno o per tratta da percorrere, sarà possibile personalizzare l'esperienza, aggiungendo treni e tratte ai Preferiti, in modo da averli sempre a portata di mano e offrendo la possibilità di "pinnare" un treno (fare click sul pulsante "pin"), ovvero impostare una notifica che apparirà nel momento in cui il treno è vicino alla propria stazione di partenza. Inoltre l'applicazione offre servizi secondari, più di "svago", come la possibilità di sbloccare e collezionare Achievement al raggiungimento di determinati traguardi.

1.2 Problema

I principali problemi da affrontare saranno quelli che riguardano il procurarsi le informazioni e l'elaborarle in maniera tale da poterle inserire e visualizzare all'interno dell'applicazione, secondo una precisa struttura e gerarchia.
Ad esempio dovrà essere possibile poter cercare tratte e treni a piacere, visualizzarne i risultati, salvarli nei preferiti per poterli poi visualizzare più comodamente.
Inoltre l'applicazione dovrà dare la possibilità di impostare una notifica con le informazioni relative al treno selezionato, anche una volta che l'applicazione è stata chiusa.
Un ulteriore problema è permettere all'utente di salvare le sue preferenze e far sì che persistano nel sistema anche dopo la chiusura dell'applicazione, come ad esempio l'aggiunta ai preferiti e gli achievement sbloccati.

Capitolo 2 - Design

2.1 Architettura

Seppur il design degli elementi costitutivi di Android sia abbastanza vincolante, abbiamo cercato di implementare per quanto possibile il pattern architetturale MVC.
Grazie all'implementazione di questo pattern è possibile cambiare la View, ad esempio passando alle librerie grafiche di java o un'applicazione web, senza toccare il model né il controller. Tuttavia è necessario attenzione alla parte del networking, in quanto è stata usata una libreria che è strettamente collegata al lifecycle delle Activity e al concetto di Activity stesso; quindi i compiti che essa svolge (che sono compiti "da controller" che vengono innestati all'interno di Activity e/o Fragment) dovrebbero essere svolti da un altro componente, a seconda dell'environment che viene scelto.

Model

Dato che per ottenere le informazioni sfruttiamo le API REST di Trenitalia e la risposta è in formato JSON, abbiamo deciso di realizzare il Model sotto forma di classi POJO create seguendo in ogni sua parte la struttura del JSON.

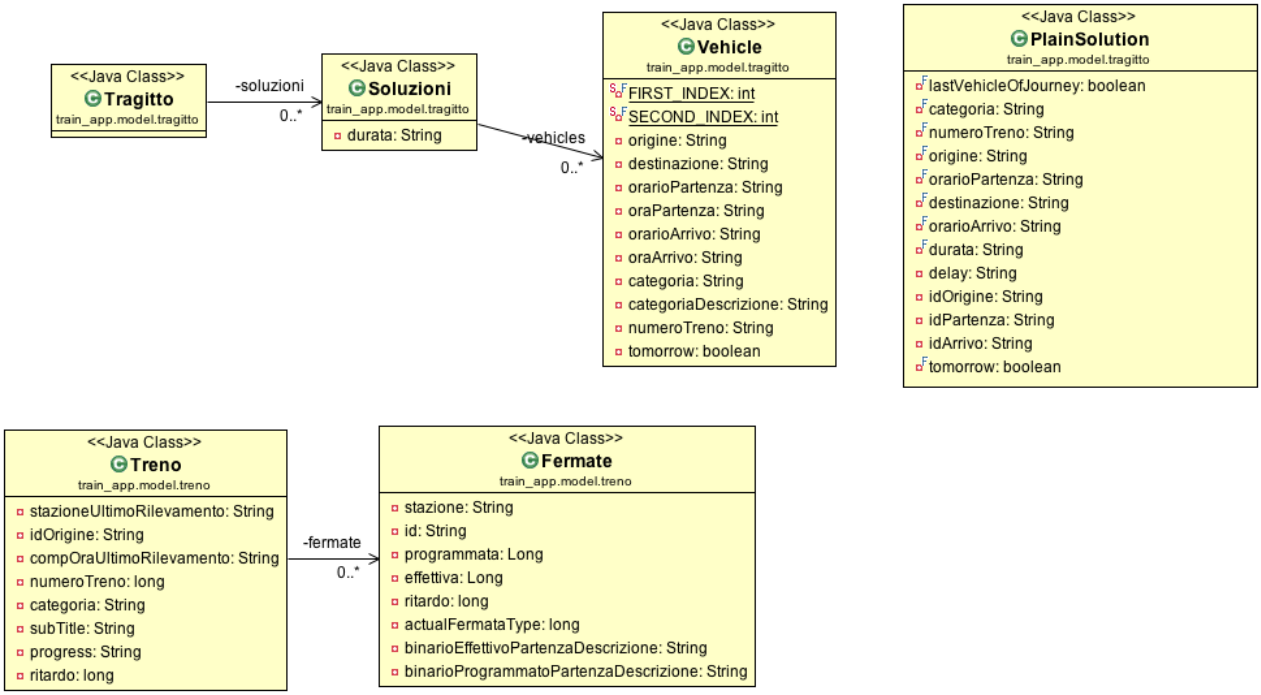


Figura 2.1: il model si divide essenzialmente in due parti principali, quella relativa al concetto di treno (Treno e Fermate) e quella relativa al tragitto (Tragitto, Soluzioni e Vehicle). Per quest'ultima si è deciso di creare un oggetto corrispondente per semplificare la relazione tra Soluzioni e Vehicle, unendoli in un unico concetto, la Plain Solution. Essendo classi composte sono di getter e setter, si è scelto di non creare interfacce.

View

La View è costituita da tre principali elementi, ovvero Activity, Fragment e Adapter, che lavorano in sinergia.

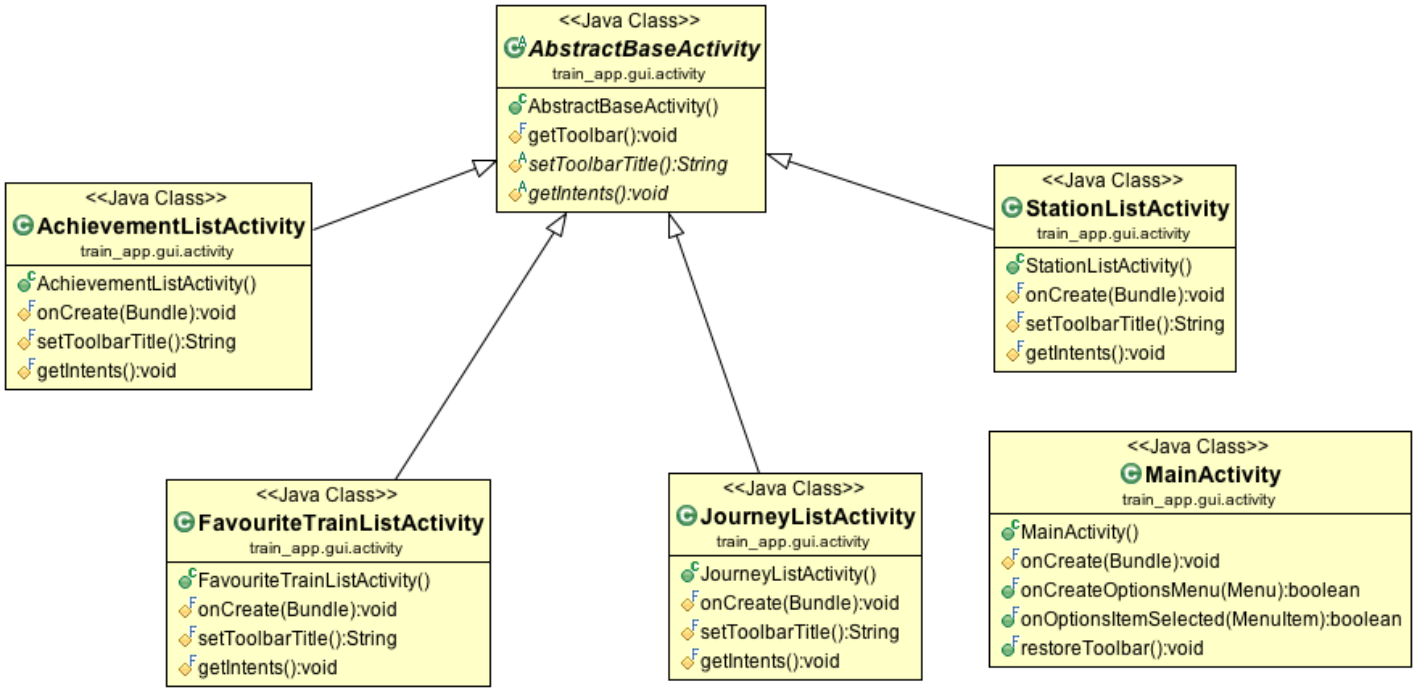


Figura 2.2: le Activity sono strutturate come in figura; fatta eccezione per la MainActivity, tutte estendono AbstractBaseActivity, perché tutte gestiscono la Toolbar e hanno bisogno di ricevere Intent. La MainActivity rappresenta un caso a parte, poiché non riceve nessun Intent e implementa una versione customizzata della Toolbar.

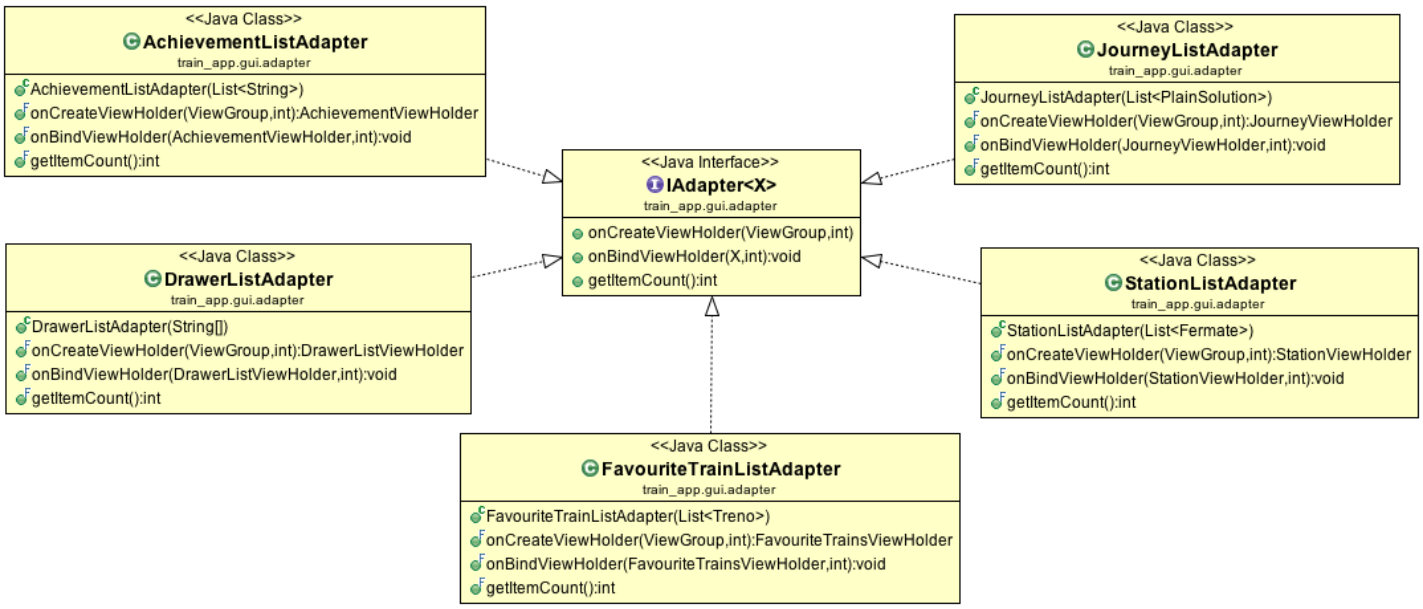


Figura 2.3: gli Adapter, che modellano il singolo elemento delle liste che compongono le RecyclerView, sono tutti implementazioni dell'interfaccia IAdapter<X>, dove il generico viene di volta in volta implementato con i ViewHolder del caso.

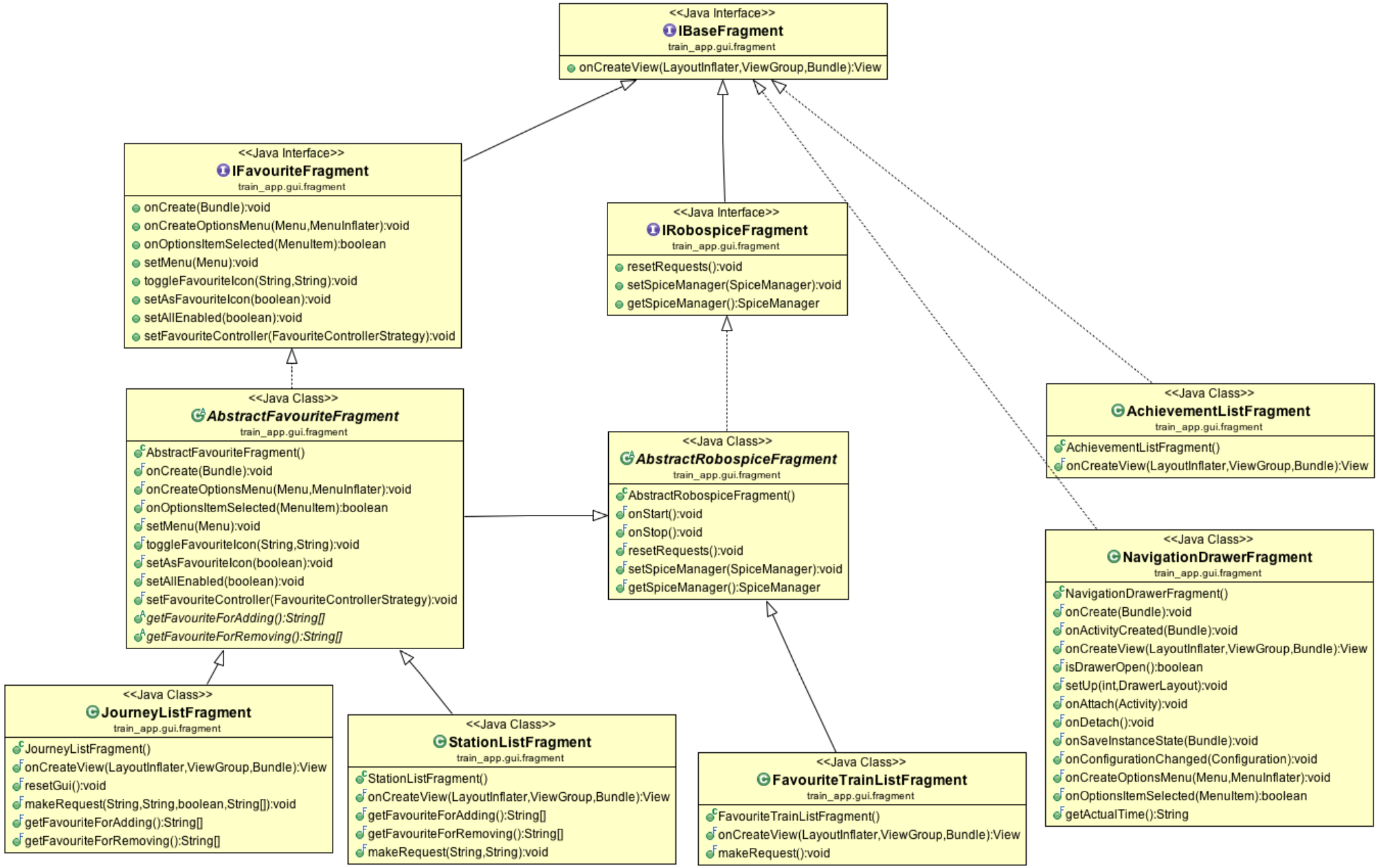

<<Java Interface>>
IBaseFragment
train_app.gui.fragment
onCreateView(LayoutInflater,ViewGroup,Bundle):View
<<Java Interface>>
IFavouriteFragment
train_app.gui.fragment
onCreate(Bundle):void
onCreateOptionsMenu(Menu,MenuInflater):void
onOptionsItemSelected(MenuItem):boolean
setMenu(Menu):void
toggleFavouriteIcon(String,String):void
setAsFavouriteIcon(boolean):void
setAllEnabled(boolean):void
setFavouriteController(FavouriteControllerStrategy):void
<<Java Interface>>
IRobospiceFragment
train_app.gui.fragment
resetRequests():void
setSpiceManager(SpiceManager):void
getSpiceManager():SpiceManager
<<Java Class>>
AbstractFavouriteFragment
train_app.gui.fragment
AbstractFavouriteFragment()
onCreate(Bundle):void
onCreateOptionsMenu(Menu,MenuInflater):void
onOptionsItemSelected(MenuItem):boolean
setMenu(Menu):void
toggleFavouriteIcon(String,String):void
setAsFavouriteIcon(boolean):void
setAllEnabled(boolean):void
setFavouriteController(FavouriteControllerStrategy):void
getFavouriteForAdding():String[]
getFavouriteForRemoving():String[]
<<Java Class>>
AbstractRobospiceFragment
train_app.gui.fragment
AbstractRobospiceFragment()
onStart():void
onStop():void
resetRequests():void
setSpiceManager(SpiceManager):void
getSpiceManager():SpiceManager
<<Java Class>>
AchievementListFragment
train_app.gui.fragment
AchievementListFragment()
onCreateView(LayoutInflater,ViewGroup,Bundle):View
<<Java Class>>
NavigationDrawerFragment
train_app.gui.fragment
NavigationDrawerFragment()
onCreate(Bundle):void
onActivityCreated(Bundle):void
onCreateView(LayoutInflater,ViewGroup,Bundle):View
isDrawerOpen():boolean
setUp(int,DrawerLayout):void
onAttach(Activity):void
onDetach():void
onSaveInstanceState(Bundle):void
onConfigurationChanged(Configuration):void
onCreateOptionsMenu(Menu,MenuInflater):void
onOptionsItemSelected(MenuItem):boolean
getActualTime():String
<<Java Class>>
JourneyListFragment
train_app.gui.fragment
JourneyListFragment()
onCreateView(LayoutInflater,ViewGroup,Bundle):View
resetGui():void
makeRequest(String,String,boolean,String[]):void
getFavouriteForAdding():String[]
getFavouriteForRemoving():String[]
<<Java Class>>
StationListFragment
train_app.gui.fragment
StationListFragment()
onCreateView(LayoutInflater,ViewGroup,Bundle):View
getFavouriteForAdding():String[]
getFavouriteForRemoving():String[]
makeRequest(String,String):void
<<Java Class>>
FavouriteTrainListFragment
train_app.gui.fragment
FavouriteTrainListFragment()
onCreateView(LayoutInflater,ViewGroup,Bundle):View
makeRequest():void

Figura 2.4: i Fragment sono gli elementi che contengono effettivamente i dati da mostrare all'utente, quindi abbiamo cercato di strutturarli al meglio possibile. L'interfaccia IBaseFragment modella il comportamento base di ogni Fragment; essa è poi estesa da IFavouriteFragment, che modella il comportamento dei Fragment che permettono l'aggiunta ai preferiti, e IRobospiceFragment, che invece fa riferimento alle classi che implementeranno i Listener per la connessione a Internet.
Queste interfacce sono poi implementate da classi astratte come AbstractFavouriteFragment o AbstractRobospiceFragment oppure in caso di Fragment particolari che non necessitano né preferiti né connessioni, direttamente da classi come AchievementListFragment e NavigationDrawerFragment.

Controller

Per quel che riguarda i componenti che fungono da Controller, non è stato possibile creare interfacce o astratte, perché ogni controller ha un comportamento differente in base alle funzioni che deve offrire, e piuttosto che creare un unico immenso Controller, che sarebbe quasi sicuramente risultato una God Class, si è scelto di frammentarli per funzionalità. Solo per quel che riguarda i Controller relativi all'aggiunta dei preferiti di treni e tratte, sono emersi comportamenti simili, e per questi è stato possibile creare una struttura più elaborata (si veda 2.2 Design dettagliato - Gestione dei preferiti).

2.2 Design dettagliato

Organizzazione dei package

I package sono strutturati in questo modo:

- achievement: package con tutte le classi relative agli achievement
- controller: package dove sono salvati tutti i controller
  - favourites: package con interfacce, astratte e classi dei controller per la gestione dei preferiti
- exception: package con le eccezioni create per l'applicazione
- gui: package con gli elementi che appartengono alla GUI
  - activity: package con tutte le Activity
  - adapter: package con tutti gli Adapter
  - fragment: package con tutti i Fragment
    - pickers: package per i DatePickerFragment e TimePickerFragment e relativa interfaccia
- model: package con gli elementi del Model e le classi di utilità come Constants e Utilities
  - tragitto: package con le classi relative alle tratte
  - treno: package con le classi relative ai treni
- network: package con le classi per la connessione a Internet (Request)
  - data: package con le classi per le richieste parziali al server
  - total: package con le classi per le richieste finali al server
- notification: package con le classi per la gestione della notifica

Gestione Network

JourneyAPI e TrainAPI sono interfacce che effettuano le richieste HTTP al server e restituiscono l'oggetto per cui si è fatto richiesta.
JourneyRestClient e TrainRestClient sono classi Singleton che definiscono dei REST Client; all'interno del metodo setupRestClient() viene legato Retrofit alla suddetta interfaccia API.
Le Request sono classi che estendono SpiceRequest di Robospice (si veda Note di Sviluppo) e attraverso il metodo loadDataFromNetwork() si occupano di restituire il risultato della richieste con eventuale elaborazione preliminare.
TrainDataRequest e JourneyDataRequest si occupano di ottenere dati necessari per eseguire le richieste successive.
TrainRequest, JourneyRequest e JourneyTrainRequest sono le richieste da cui si ottengono i dati veri e propri relativi a treni e tratte.
Le Request vengono lanciate dai Fragment e la risposta viene gestita all'interno di appositi Listener in forma di inner class (che per motivi di design relativi alla libreria usata non possono essere gestiti altrove che nella classe che ha lanciato la Request).
Ogni Listener implementa due metodi relativi alla gestione dell'avvenuta richiesta (onRequestSuccess() e onRequestFailure()), ma si è deciso di creare una sopraclasse astratta AbstractListener che gestisse in maniera comune il fallimento della richiesta.

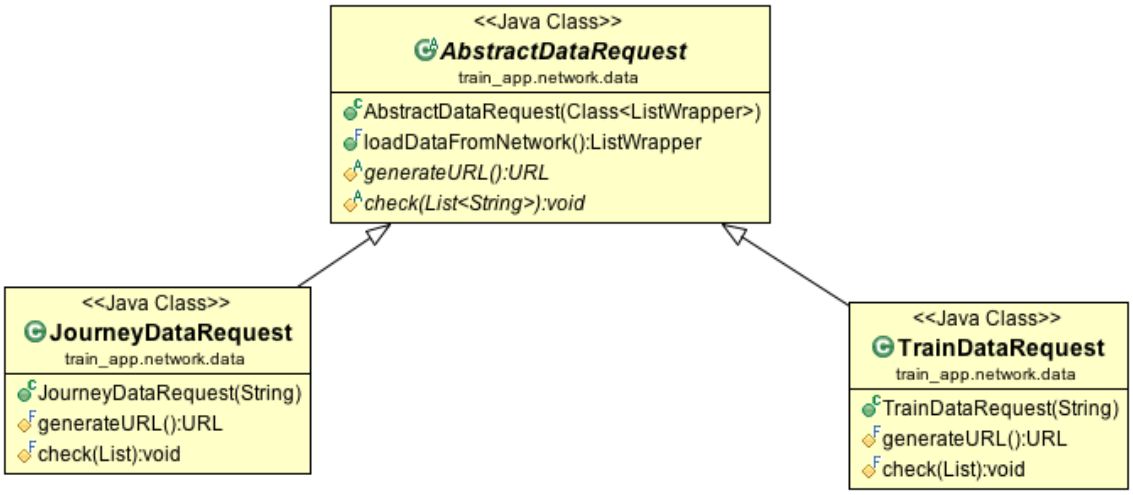


Figura 2.5: struttura di base delle Request.

Gestione della Notifica

Quando un utente fa “pin” su un treno, l’applicazione lo terrà informato sullo stato del treno in relazione alle stazioni di partenza e arrivo tramite una notifica.
Per gestire il meccanismo di invio della notifica alla barra delle notifiche, è stato necessario implementare un Service, ovvero un componente che lavori in “background” e non sia a diretto contatto con l’utente.
Esso si collega a Internet (infatti ha al suo interno un ResultListener) per ottenere informazioni riguardanti il treno scelto, e di conseguenza crea e lancia una nuova notifica che le mostra all’utente.
Se il treno è stato “pinnato” con un certo anticipo (prima di 15 minuti dal suo arrivo alla stazione desiderata) viene settato un AlarmManager, che mostrerà la notifica un quarto d’ora prima dell’arrivo del treno alla stazione di partenza.
Una volta che la notifica è comparsa, ha due pulsanti, “Aggiorna” e “Elimina” ;
questi sono associati all’invio di un PendingIntent alla classe ButtonListener, la quale estende BroadcastReceiver. Essa in base al tipo di PendingIntent che riceve, riavvia il Service, che quindi effettuerà una nuova connessione per avere i dati aggiornati (“Aggiorna”), o lo ferma eliminando anche la notifica (“Elimina”).
Si è scelto infatti di rendere la notifica fissa, quindi non rimovibile accidentalmente ma solo premendo il pulsante specifico, e inoltre si è impostata la sua priorità al massimo livello, per poterla visualizzare in cima allo stack delle notifiche e per motivi di compatibilità con alcune versioni di Android.

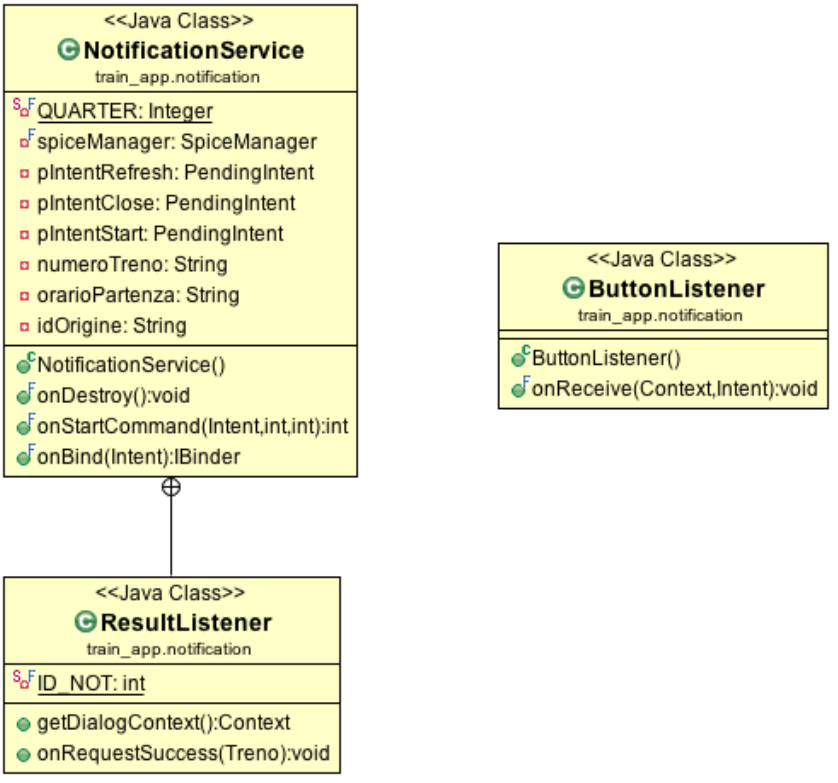
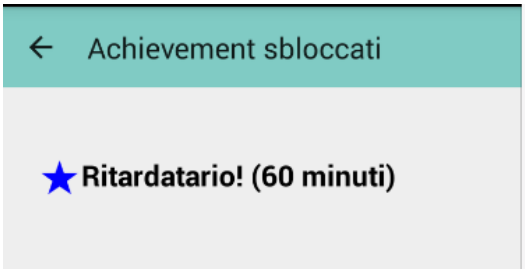


Figura 2.6 : classi principali che si occupano della gestione della notifica, ovvero il NotificationService e il relativo inner Listener, e il ButtonListener.

Gestione degli Achievement



Gli Achievement sono piccoli “traguardi” che l’utente può raggiungere durante l’utilizzo dell’applicazione.
Si è deciso di costruirli seguendo una struttura che rendesse il più semplice possibile la creazione di Achievement e livelli sempre nuovi, per eventuali sviluppi futuri.
La classe base di ogni achievement è la BasicAchievement, che implementa IAchievement. Essa ha al suo interno un oggetto di tipo Strategy che descrive il comportamento di ogni specifico achievement che verrà creato. Per creare un nuovo achievement (come ad esempio PinAchievement1 o DelayAchievement1) è sufficiente chiamare il costruttore della sopraclasse BasicAchievement, passando il valore iniziale del contatore, lo Strategy che descrive come aggiornarlo e quando sbloccarlo, e un Context.
Quest’ultimo è necessario perchè si è deciso di salvare i dati riguardanti gli Achievement all’interno di SharedPreferences che sono ottenibile tramite la chiamata getSharedPreferences() sul suddetto Context.

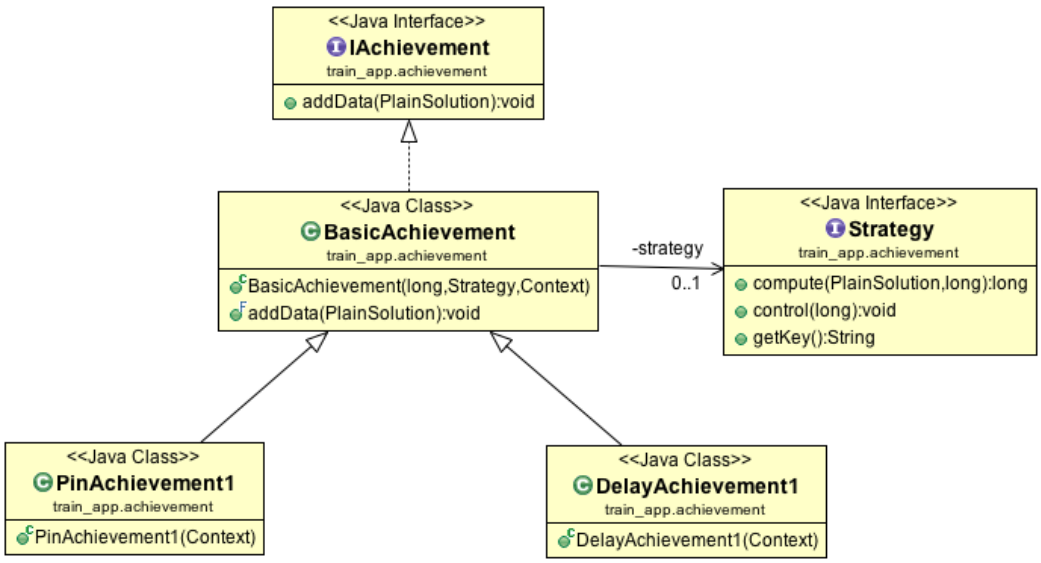


Figura 2.7 struttura base degli achievement.

Dato che gli achievement vengono calcolati in base a dati relativi a un treno su cui l’utente fa “pin”,
è stata creata la classe AchievementController, che viene istanziata all’interno del JourneyListAdapter, dove è gestita l’azione (e se in un futuro si volessero aggiornare i conteggi da un altro punto, sarebbe sufficiente istanziare il Controller).
AchievementController si occupa di aggiornare ogni achievement presente nell’applicazione in base al treno scelto, ed eventualmente di lanciare un’eccezione al momento dello sblocco di uno di essi, che nella view verrà mostrata sottoforma di Toast.

Nel file ACH_DATA_FILE verranno salvati l'achievement e il valore (ad esempio i minuti di ritardo accumulati o le volte che si ha "pinnato" un treno), mentre nel file ACH_STORE_FILE vengono registrati gli achievement già sbloccati. Quest'ultimo è necessario per sapere quali achievement non devono più mostrarsi all'utente al momento dello sblocco e per formare la lista di Achievement sbloccati all'interno dell'AchievementListActivity e Fragment.
L'AchievementListFragment contiene la lista di Achievement sbloccati. Essa viene generata tramite l'AchievementListController, che tramite il metodo computeAchievement() legge il file ACH_STORE_FILE e restituisce una lista di stringhe che descrivono gli Achievement sbloccati.

Gestione dei Treni

È possibile effettuare una ricerca per numero di treno all'interno del Navigation Drawer. Una volta inserito il numero e premuto il pulsante Cerca, si viene indirizzati alla StationListActivity e relativo Fragment. Qui avviene la connessione per scaricare i dati, la gestione degli errori, delle scelte multiple, e la loro eventuale visualizzazione. Una volta visualizzate le informazioni relative al treno, l'utente può scorrere la lista di stazioni, che saranno colorate diversamente in base al loro stato (azzurre se visitate, gialle se fermate straordinarie, rosse se fermate cancellate, bianche se non visitate). Tutto questo è gestito all'interno dello StationListAdapter, che gestisce l'aspetto di ogni elemento della RecyclerView del Fragment.
L'elaborazione dei dati scaricati avviene all'interno dello StationListController.
All'interno di questa schermata è possibile inoltre aggiungere ai preferiti il treno cercato, tramite la pressione della stellina in alto a destra. Ciò è possibile poichè lo StationListFragment estende l'AbstractFavouriteFragment.

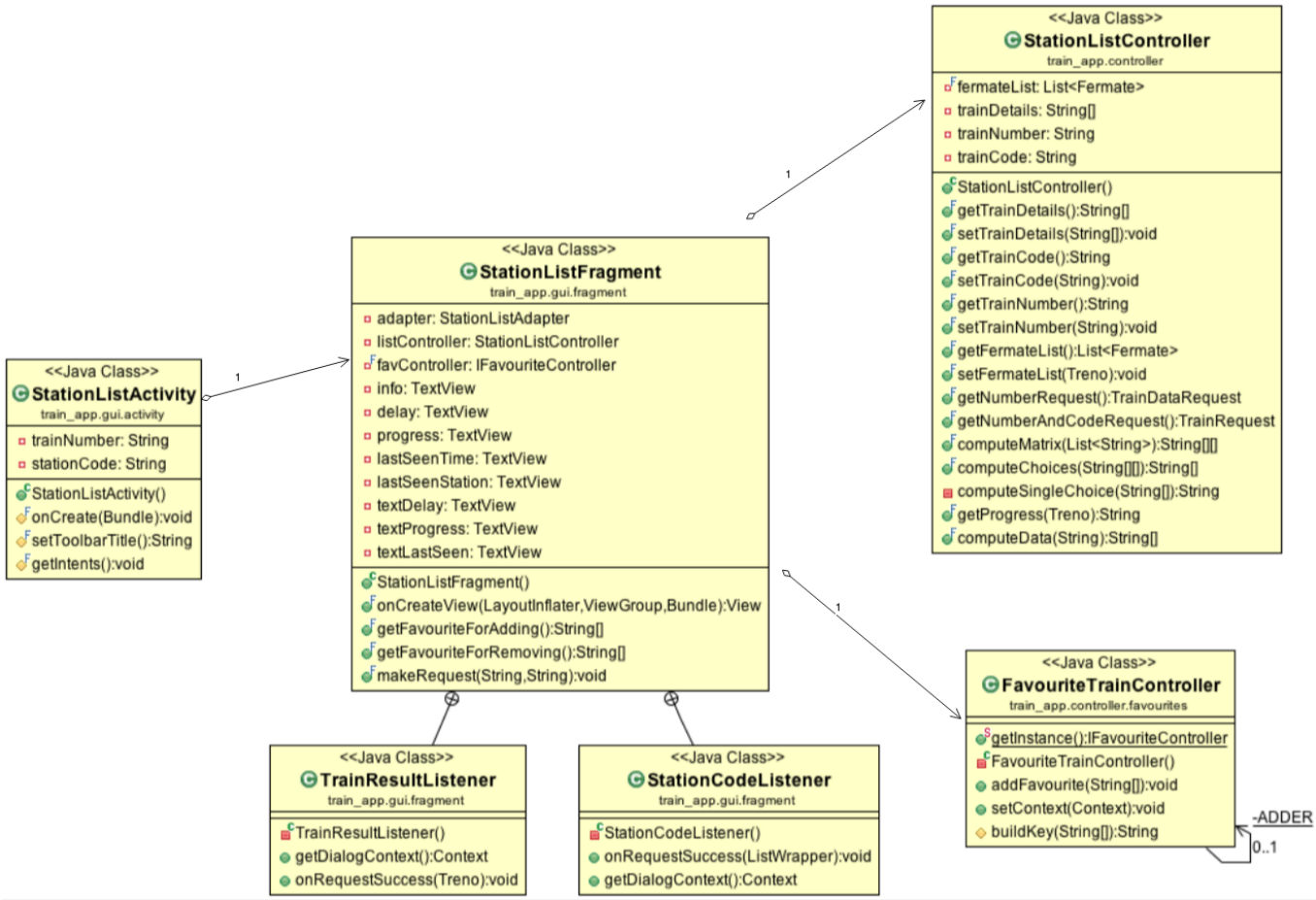
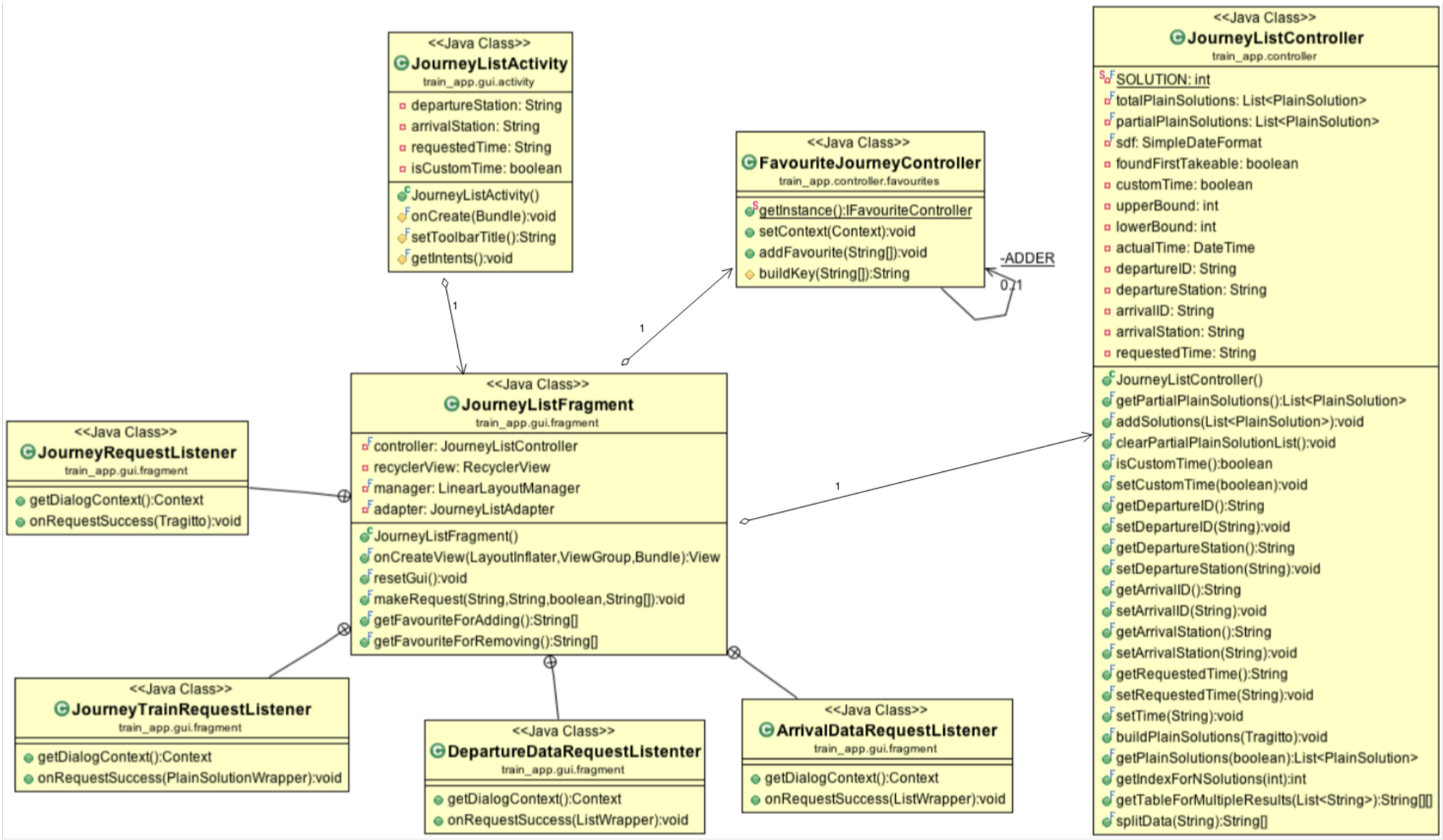


Figura 2.8: struttura base delle classi che si occupano della gestione dei treni.

Gestione delle Tratte

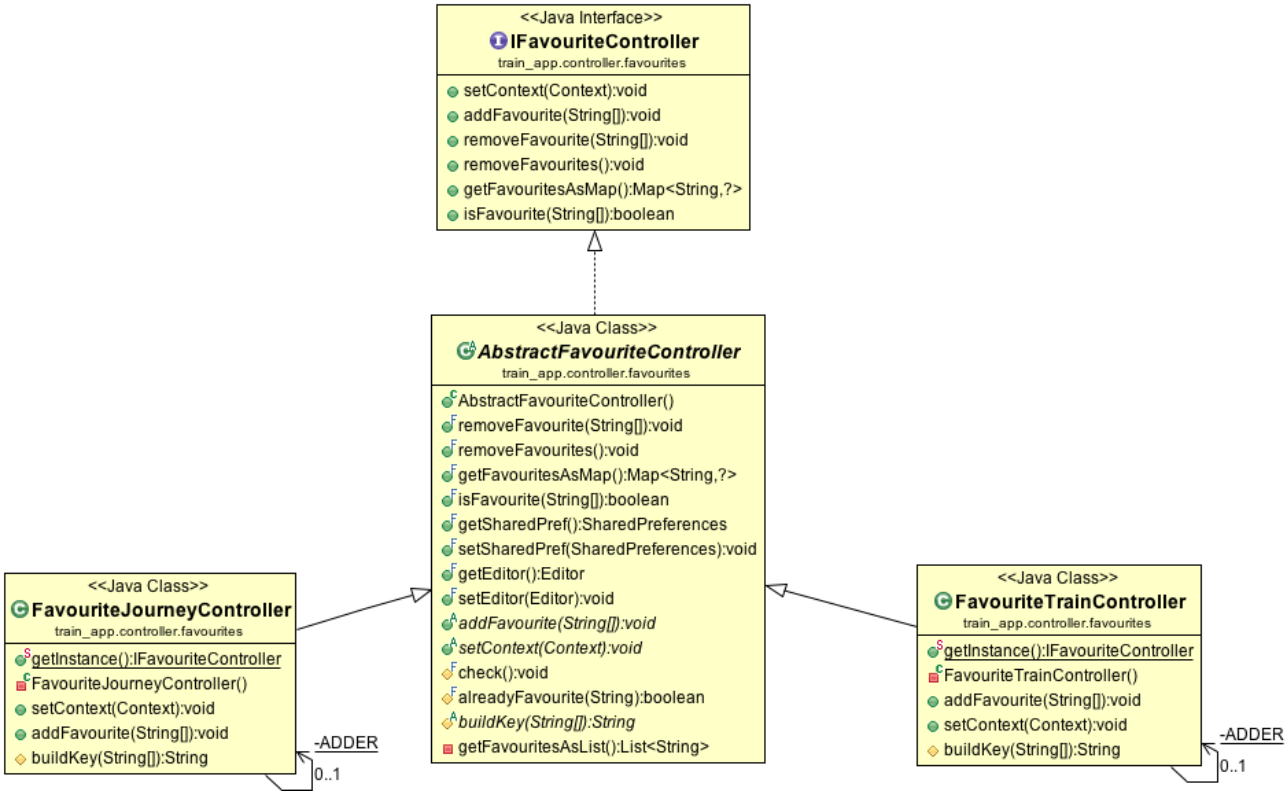
È possibile effettuare una ricerca per tratta (inserendo i nomi delle stazioni di partenza e arrivo) all'interno del Navigation Drawer.
Di default l'ora della ricerca è impostata ad un ora prima dell'ora attuale, per fornire anche le soluzioni che per eventuali ritardi o rallentamenti potrebbero essere ancora valide, ma essa può essere modificata liberamente dall'utente.
Una volta effettuata la ricerca viene visulizzata la schermata dei risultati, ovvero la JourneyListActivity e il relativo JourneyListFragment, che si occupa di scaricare i dati e gestire eventuali errori o scelte multiple.
Sarà poi possibile visualizzare le soluzioni, gestendo la visualizzazione di cambi, e permettendo all'utente di visualizzare più soluzioni (5 alla volta) semplicemente scorrendo la lista fino in fondo. Per ogni soluzione si noti l'applicazione esegue un'ulteriore ricerca, usando il numero del treno della soluzione in oggetto, per ottenerne il ritardo (informazione non disponibile altrimenti) e le informazioni vengono elaborate nella JourneyTrainRequest.
Cliccando su una soluzione si viene reindirizzati alla StationListActivity relativa al treno selezionato.
La RecyclerView su cui si basa questo fragment viene gestita dal JourneyListAdapter.
L'elaborazione dei dati scaricati, la trasformazione degli oggetti Soluzione con relativi Vehicles in oggetti di più semplice e intuitiva costruzione di tipo PlainSolution, e delle liste parziali di PlainSolution avviene invece all'interno del JourneyListController.


<<Java Class>>
JourneyListActivity
train_app.gui.activity
departureStation: String
arrivalStation: String
requestedTime: String
isCustomTime: boolean
JourneyListActivity()
onCreate(Bundle):void
setToolbarTitle():String
getIntents():void
<<Java Class>>
FavouriteJourneyController
train_app.controller.favourites
getInstance():IFavouriteController
setContext(Context):void
addFavourite(String[]):void
buildKey(String[]):String
-ADDER
0..1
<<Java Class>>
JourneyListController
train_app.controller
SOLUTION: int
totalPlainSolutions: List<PlainSolution>
partialPlainSolutions: List<PlainSolution>
sdf: SimpleDateFormat
foundFirstTakeable: boolean
customTime: boolean
upperBound: int
lowerBound: int
actualTime: DateTime
departureID: String
departureStation: String
arrivalID: String
arrivalStation: String
requestedTime: String
JourneyListController()
getPartialPlainSolutions():List<PlainSolution>
addSolutions(List<PlainSolution>):void
clearPartialPlainSolutionList():void
isCustomTime():boolean
setCustomTime(boolean):void
getDepartureID():String
setDepartureID(String):void
getDepartureStation():String
setDepartureStation(String):void
getArrivalID():String
setArrivalID(String):void
getArrivalStation():String
setArrivalStation(String):void
getRequestedTime():String
setRequestedTime(String):void
setTime(String):void
buildPlainSolutions(Tragitto):void
getPlainSolutions(boolean):List<PlainSolution>
getIndexForNSolutions(int):int
getTableForMultipleResults(List<String>):String[][]
splitData(String):String[]
<<Java Class>>
JourneyListFragment
train_app.gui.fragment
controller: JourneyListController
recyclerView: RecyclerView
manager: LinearLayoutManager
adapter: JourneyListAdapter
JourneyListFragment()
onCreateView(LayoutInflater,ViewGroup,Bundle):View
resetGui():void
makeRequest(String,String,boolean,String[]):void
getFavouriteForAdding():String[]
getFavouriteForRemoving():String[]
<<Java Class>>
JourneyRequestListener
train_app.gui.fragment
getDialogContext():Context
onRequestSuccess(Tragitto):void
<<Java Class>>
JourneyTrainRequestListener
train_app.gui.fragment
getDialogContext():Context
onRequestSuccess(PlainSolutionWrapper):void
<<Java Class>>
DepartureDataRequestListenter
train_app.gui.fragment
getDialogContext():Context
onRequestSuccess(ListWrapper):void
<<Java Class>>
ArrivalDataRequestListener
train_app.gui.fragment
getDialogContext():Context
onRequestSuccess(ListWrapper):void
1
1
1

Gestione dei Preferiti

Per la gestione dei preferiti si è scelto di creare un AbstractFavouriteFragment che implementi al suo interno i metodi per la gestione della pressione dell'apposito ImageButton presente nella Toolbar. Al suo interno è implementato un IFavouriteController che si occupa dell'aggiunta e rimozione dei preferiti, e a seconda della classe che poi estenderà l'astratta, sarà istanziato un FavouriteTrainController (per i treni preferiti) o un FavouriteJourneyController (per le tratte preferite). Entrambi sono Singleton, poiché i preferiti sono un concetto univoco in ogni punto dell'applicazione. Normalmente nei Fragment che hanno la gestione dei preferiti il titolo della Toolbar corrisponde alla ricerca selezionata ("RV 2129", oppure "Pesaro • Cesena"), ma a questo fa eccezione la MainActivity, che nella Toolbar implementa un menu Spinner. Da questo si possono scorrere le tratte preferite in precedenza, e selezionandone una i suoi dati verranno usati per lanciare la richiesta dal JourneyListFragment.
I treni preferiti invece sono mostrati all'interno della FavouriteTrainListActivity e relativo Fragment; questa schermata è gestita tramite il FavouriteTrainListController, che al suo interno lavora utilizzando un iteratore del keySet della mappa dei preferiti,da cui ottiene i dati per effettuare la richiesta, e una lista di treni preferiti che verrà restituita e mostrata tramite il FavouriteTrainListAdapter.

Figura 2.8: struttura e gerarchia delle classi che gestiscono treni e tratte favorite.



Aspetto Grafico

Per quel che riguarda l'aspetto grafico, abbiamo adeguato l'applicazione agli standard attuali, con l'eccezione di alcune funzionalità che abbiamo aggiunto di nostra iniziativa (si vedano i campi di ricerca all'interno del navigation drawer, cosa che si vede raramente fare, ma che secondo noi era la scelta migliore per incrementare l'usabilità finale dell'applicazione).
Nelle view delle liste di tratte, di stazioni o di treni favoriti, che sono le più complesse, si è cercato di non usare e innestare altri ParentLayout oltre a quello principale (RelativeLayout), per diminuire le computazioni necessarie e facilitare quindi il rendering delle suddette view quando mostrate a schermo.

Statistiche

Come spiegato successivamente, le statistiche non sono poi state implementate, ma la logica sarebbe stata quella di istanziare un nuovo fragment alla pressione di un Button all'interno dello StationListFragment, che avrebbe sostituito sostituito quest'ultimo, e in quel momento gli si sarebbe passata la List<Fermate> che ha al suo interno tutto il necessario per creare un grafico cartesiano con i nomi delle Stazioni nelle ascisse e i ritardi per ogni stazione nelle ordinate (dati che sono infatti già mostrati anche nella lista delle stazioni).

Pattern

Riassumendo, i pattern utilizzati all'interno dell'applicazione sono:

- Singleton: per i FavouriteController e le classi RestClient.
- Strategy: per gli Achievement e all'interno del metodo setFavouriteController() in AbstractFavouriteFragment.
- Template Method: all'interno di classi astratte come AbstractBaseActivity, AbstractFavouriteController, AbstractFavouriteFragment e AbstractListener.
- ViewHolder: pattern utilizzato all'interno degli Adapter, reso obbligatorio per utilizzare le RecyclerView.

Capitolo 3 - Sviluppo

3.1 Testing

L'applicazione è stata testata su diversi dispositivi durante il suo sviluppo.
I dispositivi utilizzati sono:

- Samsung S3 mini
- Motorola Moto G
- Nexus S
- Samsung S4

Su tutti i dispositivi citati, l'applicazione è risultata funzionante. Il livello minimo di SDK API è 16 (Android 4.1).

3.2 Divisione dei compiti e metodologia di lavoro

I compiti sono stati suddivisi nel seguente modo:

Lisa Mazzini:

- Gestione della ricerca dei treni (StationListActivity, Fragment, Controller e Adapter)
- Gestione dei treni preferiti (FavouriteTrainListActivity, Fragment, Controller e Adapter, e FavouriteTrainController)
- Gestione del "pin" e della notifica (NotificationService, ButtonListener)
- Gestione degli achievement (package "achievement")

Alberto Giunta:

- Gestione della ricerca delle tratte (JourneyListActivity, Fragment, Controller e Adapter)
- Gestione delle tratte preferite (MainActivity, FavouriteJourney Controller e FavouriteControllerStrategy)
- Gestione del NavigationDrawer (NavigationDrawerFragment)
- Realizzazione dei file xml per i layout dell'applicazione.

L'architettura di base è stata progettata insieme, come ad esempio la creazione di interfacce e classi astratte che sarebbero poi state estese da classi di entrambi. Da questo punto di partenza poi ognuno ha sviluppato indipendentemente i vari componenti.
Fa eccezione la View in cui alcune classi astratte e interfacce si sono rese necessarie nel momento in cui sono stati scelti alcuni componenti grafici piuttosto che altri, e le scelte di come modificare il comportamento precedentemente deciso, relative ad esempio all'AbstractBaseActivity, il package dei Pickers, l'AbstractFavouriteFragment e l'AbstractRobospiceFragment sono state fatte da Giunta, che era il componente che aveva in carico la parte grafica.

Si è collaborato per la realizzazione dei componenti necessari al collegamento internet e al parsing delle informazioni (si veda 2.2 Design dettagliato - Gestione Network), poichè è stato necessario fare ricerca e scegliere il modo più adeguato per farlo.
Le dipendenze fra le due parti sono state piuttosto chiare fin da subito, quindi non è stato necessario cambiare o modificare la struttura di base. Tuttavia durante lo sviluppo si è deciso di utilizzare elementi come il NavigationDrawer che ha reso necessario una rielaborazione della struttura dei Fragment, per renderlo il più "pluggabile" possibile anche in altri punti dell'applicazione un giorno che si lo volesse mostrare anche al di fuori della MainActivity.

Purtroppo non è stato possibile sviluppare all'interno del monteore la funzionalità delle Statistiche per l'andamento del treno, poichè, circa a metà del tempo concesso per il progetto, il sito da cui ricavavamo i dati tramite parsing HTML è stato dismesso (si veda 3.3 Note di sviluppo). Ciò ha reso necessario trovare mezzi diversi e di conseguenza cercare nuovi strumenti per il parsing e rivedere tutto il Model. Data l'imprevedibilità della cosa e delle conseguenze, è stato necessario sacrificare una funzionalità.

Il team ha usato il DVCS Mercurial, per tenere traccia di ogni cambiamento e feature che veniva aggiunta o modificata.

3.3 Note di sviluppo

Abbiamo deciso di non limitare la GUI a delle semplice Activity ma strutturarla in modo da inserire dei Fragment al loro interno, perchè questo rende il tutto molto più elastico e riutilizzabile (ad esempio si può usare uno stesso fragment all'interno di due activity diverse, si veda il caso della MainActivity e della JourneyListActivity).
Inoltre si è deciso di usare le RecyclerView al posto delle tradizionali ListView poiché queste sono state recentemente sviluppate e quindi saranno supportate presumibilmente per più tempo. Sono più performanti, offrono uno scrolling più fluido rispetto alle ListView.

Per come è strutturata l'applicazione, sarebbe stato opportuno utilizzare pattern come il Decorator per le Activity e i Fragment, tuttavia l'architettura del sistema Android non lo rendeva possibile se non tramite espedienti che avrebbero aumentato notevolmente la complessità del codice.

L'idea iniziale per ottenere i dati era quella di procedere con lo scraping del sito mobile.viaggiatreno.it, ma come si può notare visitandolo, esso non è più attivo in quanto sostituito da una nuova versione (www.viaggiatreno.it/viaggiatrenonew/index.jsp) che non utilizza più HTML in chiaro, come la versione precedente.
Ci siamo accorti di ciò verso i primi di Febbraio, ed è stato necessario cambiare strategia per l'ottenimento dei dati.
Analizzando quindi le richieste che venivano inoltrate dai portali ufficiali ai server di Trenitalia (sotto forma di API REST), abbiamo capito come formularne di nostre e analizzare le risposte.

Siamo passati quindi dall'uso della libreria di scraping JSOUP (http://jsoup.org) all'utilizzo della libreria Retrofit (http://square.github.io/retrofit/) con la quale è possibile generare oggetti Java a partire da JSON, tramite l'uso di classi POJO (che come detto in precedenza, sono quelle che costituiscono il Model). Abbiamo utilizzato i loro tutorial per capire come utilizzarla unitamente alla pagina http://inaka.net/blog/2014/10/10/android-retrofit-rest-client/ per l'implementazione vera e propria.

Per gestire i vari collegamenti Internet necessari, si è deciso di utilizzare la libreria Robospice (https://github.com/stephanenicolas/robospice) in alternativa al nativo AsynkTask perchè offre una migliore gestione delle richieste in relazione al lifecycle delle Activity.

Inoltre per gestire orari e date, si è scelto di utilizzare la libreria Joda-Time (http://www.joda.org/joda-time/) poichè offre una migliore gestione di questo tipo di dati.

I crediti per la classe AbstractEndlessRecyclerOnScrollListener e il suo funzionamento sono di https://gist.github.com/ssinss/e06f12ef66c51252563e.

L'implementazione delle componenti Android e le relazioni tra di esse sono frutto di studio delle soluzioni consigliate nelle linee guida di Android nel sito http://developer.android.com/index.html, così come di alcuni thread su StackOverflow.

Capitolo 4 - Commenti finali

4.1 Conclusioni

Il progetto è stato impegnativo, soprattutto per la scelta di svilupparlo per il sistema operativo Android, ambiente in cui entrambi i componenti del gruppo sono autodidatti. Tuttavia è sembrata la scelta migliore per poter rendere un giorno l'applicazione accessibile a tutti, pubblicandola sul Google Play Store.
Prima di fare ciò vogliamo sicuramente migliorare l'aspetto grafico e fare il modo di utilizzare la cache per i dati scaricati, in modo da non doverli scaricare ogni volta che viene fatta una richiesta.
In conclusione, siamo in gran parte riusciti a ottenere il risultato che ci eravamo prefissati, e ne siamo soddisfatti.

Appendice A - Guida utente

All'apertura, l'applicazione mostra i risultati per la ricerca delle tratte preferite; se non ci sono tratte preferite, la schermata appare vuota, altrimenti si possono scorrere tramite il menu spinner posto nella Toolbar.
Scorrendo da sinistra verso destra o premendo la lente di ingrandimento a sinistra nella Toolbar, si apre il NavigationDrawer da cui è possibile:

- Effettuare una ricerca per numero
- Effettuare una ricerca per tratta, giorno e ora
- Visualizzare i treni preferiti
- Eliminare tutti i treni preferiti
- Visualizzare gli achievement sbloccati

Per aggiungere o rimuovere un preferito è sufficiente premere la stellina posta in alto a destra nella Toolbar nella schermata del risultato della ricerca, sia per i treni che per le tratte.

Per impostare la notifica, è necessario premere il pulsante "pin" nei risultati per ricerca tratta; se il treno arriverà alla stazione desiderata entro 15 minuti, la notifica si imposterà subito, altrimenti verrà impostata un quarto d'ora prima dal suo arrivo.
La notifica sarà aggiornabile direttamente dalla barra delle notifiche e può essere cancellata solo tramite il pulsante "Elimina" e non tramite swipe.
Cliccando sulla notifica, si apre l'applicazione nella schermata che descrive l'andamento del treno.