SAPIENZA
UNIVERSITÀ DI ROMA

Progettazione e sviluppo di un'interfaccia utente per la gestione degli scambi nell'applicazione GeneroCity per Android

Facoltà di Ingegneria dell'Informazione, Informatica e Statistica

Corso di Laurea in Informatica in collaborazione con l'Università Telematica UNITELMA Sapienza - Teledidattica

Candidato

Domiziano Scarcelli
Matricola 1872664

Relatore

Prof. Emanuele Panizzi

Anno Accademico 2020/2021

Progettazione e sviluppo di un'interfaccia utente per la gestione degli scambi nell'applicazione GeneroCity per Android
Tesi di Laurea. Sapienza – Università di Roma

Questa tesi è stata composta con LaTeX e la classe Sapthesis.

Email dell'autore: scarcelli.1872664@studenti.uniroma1.it

Indice

Capitolo 1

Introduzione

1.1 Analisi di contesto

Secondo le analisi ISTAT del 2019, i problemi che preoccupano più le famiglie italiane sono: il traffico (39,2%), l'inquinamento dell'aria (37,7%), la difficoltà di parcheggio (37,4%), con un aumento di 1,7 punti percentuali rispetto all'anno precedente. Tale problema è concentrato nelle regioni del Nord.

La ricerca del parcheggio porta ad un ingente aumento dell'inquinamento dell'aria in quanto la maggior parte delle emissioni nocive dei mezzi a motore avviene a bassi regimi.

Questo è causato dal fatto che in città molto trafficate si possono utilizzare anche 45 minuti nel traffico alla ricerca di un parcheggio. [1]

Per *smart parking* si intende l'utilizzo di una tecnologia che consenta di individuare le piazzole di sosta disponibili all'interno di una determinata area, permettendo di creare una mappa dei vari parcheggi aggiornata in tempo reale. [2]

Un sistema di smart parking porterebbe quindi, se altamente diffuso, ad un elevato numero di benefici, tra cui:

- Riduzione del traffico causato dalle persone alla ricerca di parcheggio;
- Risparmio di carburante, e quindi denaro, da parte degli utenti;
- Risparmio di tempo occupato alla guida;
- Riduzione delle emissioni da CO2
- Riduzione dello *stress* degli utenti

Un modo per risolvere il problema dei parcheggi nelle città, e quindi implementare lo *smart parking*, è quello di creare delle infrastrutture *ad hoc* che permettano di capire quali parcheggi sono occupati e quali no tramite sensori e videocamere, in maniera da comunicare poi tale informazione tramite qualche canale, ad esempio un'applicazione per mobile, agli automobilisti interessati. [3]

Questa soluzione, anche se molto efficace, richiede non solo lo sviluppo di un'applicazione per la comunicazione tra l'infrastruttura e gli utenti, ma richiede soprattutto un intervento diretto da parte del singole città per quanto riguarda la vera e propria costruzione delle infrastrutture.

Attualmente la soluzione appena descritta viene implementate in parcheggi privati come garage, parcheggi di centri commerciali ecc.

1.2 GeneroCity

La soluzione proposta da GeneroCity è quella di realizzare tutte le funzionalità relative allo *smart parking* sopra citate senza però fare ricorso ad infrastrutture realizzate appositamente.

Tra le funzionalità che l'applicazione dovrebbe permettere è possibile trovare:

- Registrazione e rimozione automatica dei parcheggi effettuati dall'utente;
- Rilevazione autonoma della ricerca di un parcheggio da parte dell'utente;
- Possibilità di comunicare l'intezione di lasciare un parcheggio occupato ad un utente che ne ha bisogno;
- Visualizzazione di statistiche relative ai parcheggi effettuati;

Dal sito web ufficiale dell'applicazione, è possibile reperire una descrizione del sistema:

> GeneroCity è un'applicazione di smart parking per Android e iOS sviluppata dal Gamification Lab del Dipartimento di Informatica dell'Università degli Studi di Roma "La Sapienza".
>
> Lo scopo dell'applicazione è quello di facilitare lo scambio dei parcheggi all'interno di un'area urbana puntando sulla generosità degli utenti.
>
> L'applicazione consente inoltre di gestire le informazioni delle proprie automobili e di condividerle con i propri familiari. [4]

Gamification

Con il termine *gamification* si intende l'utilizzo di meccaniche ludiche all'interno di contesti non di gioco.

L'applicazione implementa un sistema di *gamification* volto a rendere gli utenti più partecipi, dando loro la possibilità di accumulare e spendere dei punti nell'effettuare scambi di parcheggio con gli altri utenti della *community*.

Un utente infatti può effettuare la ricerca di un posto solamente se ha abbastanza punti, denominati *GCoins*, a disposizione, che possono essere guadagnati dall'utente quando questo lascia il proprio parcheggio ad un altro utente.

Il sistema appena descritto è implementato nella versione dell'aplicazione di iOS ma non in quella Android, in quanto ancora nelle prime fasi di sviluppo.

1.2.1 Stato dell'applicazione all'inizio del tirocinio

All'inizio di questo tirocinio l'applicazione presentava solo una schermata principale, su cui veniva segnata la posizione del parcheggio corrente. Era tuttavia presente il modello della macchina a stati in grado di riconoscere lo stato dell'utente, e quindi di registrare e rimuovere automaticamente i parcheggi.

1.3 Obiettivi del tirocinio

Lo studente ha eseguito il tirocinio dal mese di Novembre 2021 fino a Marzo 2022. In tale periodo ha svolto i seguenti task:

- Progettazione e sviluppo, insieme ad altri due membri del team, di un'interfaccia utente per la visualizzazione di uno storico dei parcheggi effettuati dall'utente;
- Paginazione dei risultati presenti nello storico dei parcheggi.
- Progettazione e sviluppo delle schermate relative alla gestione dei match.
- Progettazione e sviluppo della schermata per modificare la posizione di un parcheggio.

1.4 Ciclo di progettazione dell'interfaccia utente

Con ciclo di progettazione di un sistema si intende quel processo volto a capire in che maniera strutturare il sistema che si vuole realizzare, partendo dal capire i bisogni degli utenti, fino ad effettuare dei prototipi dell'interfaccia.

Lo scopo di tale ciclo è quello di progettare un sistema che consenta la miglior esperienza utente possibile.

L'intero processo di progettazione dell'interfaccia utente (UI: User Interface) e dell'esperienza utente (UX: User Experience) può essere diviso in 5 fasi: [5] [6]

1. Definizione del prodotto
2. Ricerca
3. Analisi
4. Progettazione
5. Test di usabilità

Definizione del prodotto

Per definizione del prodotto si intende capire in cosa consisterà il prodotto, capire i suoi reali scopi e quindi perché è necessaria la sua realizzazione.

In questa fase i progettisti comunicano con le altre persone del team e con il committente in modo da ricavare dei requisiti, che serviranno poi nelle fasi successive

Ricerca

Durante questa fase viene effettuata una ricerca approfondita sull'utente e sul mercato.

Fasi della ricerca sono:

- *Individual in-depth interviews* (IDI)*:* consistono in interviste effettuate agli utenti e permettono di trovare informazioni come il target, i bisogni, le paure, le motivazioni e i comportamenti.

- *Competitive research*: consiste nell'identificare le opportunità del prodotto nella sua nicchia, studiando la competizione e comprendendo i punti di forza e di debolezza di tali prodotti.

Analisi

Nella fase di analisi si passa a capire non più "cosa" l'utente vuole, cosa pensa e di cosa ha bisogno, informazioni trovate nella fase di ricerca, ma si cerca di capirne il "perchè".

Vengono quindi prodotti una serie di oggetti:

- *User Personas*, ovvero delle rappresentazioni fittizie di personaggi che rappresentano differenti tipi di utenti che utilizzano il sistema.
- *User Stories*, ovvero uno strumento che consente di vedere l'interazione tra l'utente e il sistema dal punto di vista dell'utente. Spesso una *story* segue la struttura: "*Come [utente] voglio [obiettivo da raggiungere] in maniera tale da [motivazione]*"
- *Storyboards*, ovvero la connessione tra *User Personas* e *User Stories*, consiste in una storia (rappresentata testualmente o graficamente) dell'utente che interagisce con il sistema.

Progettazione

Una volta trovate le informazioni chiave relative all'utente e al mercato, è possibile progettare l'interfaccia utente del sistema.

Vengono quindi prodotti *wireframes* e prototipi che verranno usati poi per la fase di testing e la fase di implementazione.

Wireframe Un *wireframe* costituisce una bozza della prima rappresentazione dell'interfaccia, il cui scopo è quello di identificare la struttura dell'applicazione (applicazione intesa in maniera generica come il luogo dove viene visualizzata tale interfaccia utente) e mostrare la disposizione degli elementi nella pagina. [17]

Un wireframe può essere realizzato con penna e carta, ma esistono dei software *ad hoc* come *Adobe XD* e *Figma*.

I wireframe sono un modo veloce ed efficace per realizzare prototipi rapidi di interfacce, e possono essere utilizzati per misurare la funzionalità di quest'ultime.

Solitamente, soprattutto se progettati a mano, i wireframe consistono in una rappresentazione a bassa fedeltà (*lo-fi* in inglese) dell'interfaccia che si intende realizzare. Tale rappresentazione permette rapidità nella progettazione iniziale e negli eventuali cambiamenti futuri.

Una rappresentazione ad alta fedeltà, e quindi più ricca di dettagli e più simile all'interfaccia definitiva che verrà poi implementata sul sistema, è più impegnativa ed eventuali modifiche richiedono maggior tempo e sforzo. [18]

Il modello a bassa fedeltà di un'interfaccia utente definisce:

- L'organizzazione degli elementi sullo schermo;
- La sequenza dei passaggi che l'utente deve effettuare per concludere un *task*;

- Il modo tramite il quale l'utente interagisce con il sistema.

La principale proprietà di un wireframe a bassa fedeltà è il fatto che questo viene progettato senza tener conto dei dettagli estetici, ma viene data completa priorità alla struttura. In questo modo l'attenzione è incentrata completamente sullo scheletro della struttura dell'interfaccia.

Inoltre in questo modo gli utenti che effettueranno eventuali test di usabilità non verranno distratti da scelte estetiche non definitive, e i test riguarderanno solamente l'interazione dell'utente con il sistema.

Prototipo Mentre un *wireframe* è un artefatto statico, un prototipo è una bozza più avanzata dell'interfaccia che permette l'interazione da parte dell'utente. [19]

Un prototipo risulta utile per effettuare test di usabilità prima ancora che l'interfaccia venga implementata.

Test di usabilità con utenti

Alla base della progettazione di un'interfaccia utente vi è il concetto di usabilità, il cui scopo è quello di rendere l'interfaccia facile da comprendere, da ricordare e da usare, in maniera tale che comporti il minor sforzo cognitivo possibile da parte dell'utente.

Un'interfaccia poco usabile può far commettere delle azioni non volute dall'utente, e quindi rovinare l'esperienza di quest'ultimo.

Per test di usabilità si intende l'osservazione dell'interazione tra il sistema che si vuole testare ed un utente, a cui utente viene assegnato un o più compiti da svolgere, e vengono analizzati i suoi comportamenti durante tale svolgimento. Questi test sono una parte fondamentale della progettazione di un'interfaccia in quanto possono essere tenuti prima del vero e proprio sviluppo, visto che l'utente può interagire con un prototipo del sistema, e consentono di rilevare molti errori legati all'usabilità. [7]

1.5 Euristiche di Nielsen

Prima ancora di effettuare i test con gli utenti, è possibile valutare l'usabilità durante la progettazione dell'interfaccia osservando se quest'ultima soddisfi i principi stilati nelle 10 euristiche di Nielsen, indicate di seguito. [8]

Visibilità dello stato del sistema

Il sistema deve tenere aggiornato l'utente su cosa esso stia facendo, e quindi fornire dei *feedbacks*.

Esempi di *feedbacks* sono:

- Messaggi di errore;
- Icone, dei testi o dei bottoni meno saturi per mostrare che la funzione non è disponibile;
- Barre di caricamento per mostrare che una certa attività è in corso.

Corrispondenza tra sistema e mondo reale

Il sistema deve essere espresso in un linguaggio comune all'utente, con parole, frasi e concetti a lui familiari.

Essenziale è anche l'utilizzo di metafore, ovvero effettuare delle scelte progettuali che consentano all'utente di associare un immagine o un processo presente nell'interfaccia del sistema ad un qualcosa presente nella vita di tutti i giorni. In questa maniera la persona che viene messa di fronte ad una metafora ragiona per analogia.

Controllo e libertà

L'utente deve avere il controllo del sistema e deve potersi muovere in maniera agile all'interno di quest'ultimo.

È necessario quindi evitare:

- Procedure troppo lunghe;
- Percorsi predefiniti senza possibili scorciatoie;
- Azioni non volute dall'utente, come l'apertura di pagine non richieste.

Consistenza e standard

L'utente deve avere la sensazione di trovarsi sempre nello stesso ambiente. È quindi necessario che le convenzioni grafiche utilizzate siano valide all'interno di tutto il sistema.

Molto utile è anche l'utilizzo di elementi standard nel contesto in cui viene sviluppato il sistema.

Nell'ambito di sviluppo Android sarà quindi di aiuto attenersi alle linee guida del Material Design.

Prevenzione dell'errore

Il sistema deve evitare di porre l'utente in situazioni che possano portare questo a commettere degli errori. Bisogna quindi evitare che l'interfaccia presenti delle ambiguità.

Quando possibile è necessario dare la possibilità all'utente di annullare le proprie azioni.

È necessario chiedere conferma quando l'utente cerca di eseguire delle azioni distruttive o critiche.

Riconoscimento anziché ricordo

L'utilizzo del sistema deve evitare che l'utente abbia il bisogno di riscoprire ogni volta l'interfaccia.

I layout devono quindi essere semplici e schematici, simili tra di loro in maniera tale che l'utente possa riconoscere dei *pattern.*

Visto che la maggior parte degli utenti, al giorno d'oggi, utilizza applicazioni interattive su dispositivi mobili che, per costituzione, hanno delle dimensioni ridotte, è preferibile, ovunque sia necessario un inserimento di informazioni, dare all'utente

l'opportunità di scegliere e quindi di selezionare l'informazione corretta invece che inserirla tramite la tastiera.

Flessibilità d'uso

Offrire all'utente di utilizzare il sistema in maniera differente a seconda della sua esperienza. Offrendo quindi comandi semplici per i meno esperti e delle scorciatoie per i più esperti.

Design e estetica minimalista

Dare maggior importanza al contenuto quanto all'estetica.

È importante mantenere l'interfaccia minimalista in modo da evitare che l'utente venga distratto o confuso da elementi irrilevanti o raramente necessari.

Aiuto all'utente

Aiutare l'utente a riconoscere e diagnosticare un eventuale errore.

Bisogna quindi esprimere i messaggi di errore in linguaggio semplice e comprensibile, evitando codici.

Tali messaggi devono inoltre indicare in maniera precisa il problema e suggerire una soluzione.

Documentazione

Anche se il sistema dovrebbe essere usabile senza consultare una documentazione, quest'ultima dovrebbe essere disponibile, facile da leggere e strutturata in un insieme di passi comprensibili.

Capitolo 2

Tecnologie utilizzate

2.1 Version control

Il progetto è stato portato avanti da un team di persone, quindi per coordinare il lavoro e i relativi cambiamenti al codice effettuati di volta in volta, si è scelto di utilizzare il software di *version control* Git.

Tale software permette di tracciare i cambiamenti relativi ad un insieme di files, e viene utilizzato per coordinare il lavoro tra i programmatori nel team.

La gestione avviene all'interno di GitLab, ovvero un provider di servizi che permette di gestire i progetti che utilizzano il software di controllo di versione Git. Permette di creare delle *issues*, ovvero dei problemi che devono essere risolti, ed associare ad esse delle *merge request.*

Una *merge request* è un modo di controllare i cambiamenti effettuati nel codice presente in una *feature branch* (*branch* in cui si sta attualmente sviluppando una *feature* o risolvendo un *bug*) prima che questo venga unito al codice presente nella *branch* principale.

Il *branching* è un argomento fondamentale quando si parla di *version control* e di progetti che richiedono lo sviluppo contemporaneo di più persone all'interno di un *team.*

Il *branching*, in italiano ramificazione, consente agli sviluppatori di creare una copia del codice sorgente dell'applicazione, su cui lavorare per aggiungere una *feature* o per risolvere un *bug.* I cambiamenti effettuati sulla *feature branch* non impattano in nessun modo la *branch* principale.

Una volta che il bug è stato risolto, o la feature implementata, e una volta che sono stati effettuati i dovuti test, il codice della *feature branch* può essere inserito nella *branch* principale tramite l'operazione di *merge.*

2.2 Componenti fondamentali in un'applicazione Android

In questa sezione vengono documentate le classi che sono fondamentali al fine di sviluppare un'applicazione Android.

2.2.1 Activity

Una *activity* è una classe che permette di creare una finestra in cui è possibile specificare una determinata interfaccia grafica, il cui scopo è quello di permettere l'interazione tra il sistema e l'utente

Generalmente un'*activity* è a schermo intero, ma questa può essere visualizzata anche come una finestra fluttuante oppure può essere inserita in un'altra finestra. [9]

Dal momento in cui l'*activity* compare sullo schermo al momento in cui scompare, questa passa attraverso una serie di stati, in quella che viene chiamato *activity lifecycle*, o ciclo di vita di una *activity*.

Un'activity presenta una serie di *callbacks* che permettono alla activity stessa di capire che è avvenuto un cambio di stato all'interno dell'appicazione.

Queste funzioni, come ogni altra funzione in Java, possono essere *overwritten,* ovvero il loro comportamento può essere modificato, ed è quindi possibile specificare come l'activity debba comportarsi ad ogni cambio di stato. [10]

Una buona implementazione delle funzioni presenti nel ciclo di vita di una *activity* permette all'applicazione di essere performante e robusta, ad esempio può evitare che l'app:

- Vada in *crash* se l'utente, ad esempio, mentre utilizza l'applicazione, riceve una telefonata e quindi l'applicazione deve essere eseguita in background;
- Consumi delle risorse di sistema preziose quando non ce n'è bisogno;
- Perda i progressi effettuati dall'utente quando questo esce e rientra in un secondo momento;

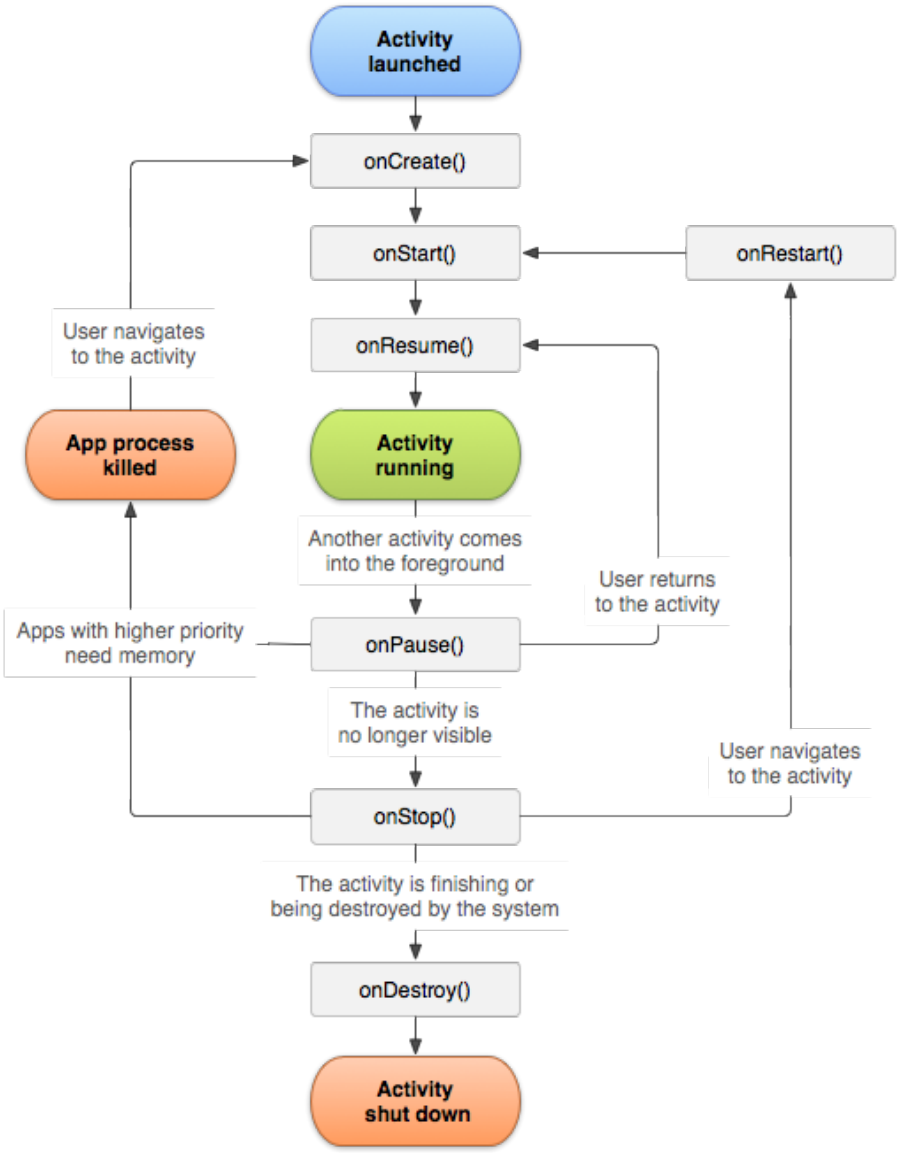

Figura 2.1. Ciclo di vita di una *activity*

Una *activity* implementa sei *callbacks* fondamentali:

- `onCreate`: viene chiamata quando il sistema crea l'activity, ed è all'interno di tale funzione che vengono effettuate le operazioni di inizializzazione che dovrebbero avvenire una sola volta durante l'intera vita dell'acivitiy;
- `onStart`: viene chiamata per rendere visibile l'activity all'utente, prima che essa venga eseguita in *foreground* e diventi interattiva.
- `onResume`: viene chiamata ogni volta che l'attività deve essere eseguita in *foreground*. L'attività rimane in questo stato finché non avviene un qualche evento che la porta in *background*, come può essere una chiamata o l'apertura di un'altra applicazione.

- `onPause`: viene chiamata ogni volta che si presenta un evento che interrompe la normale esecuzione dell'activity. Questo non vuol dire che l'activity stia per essere distrutta, tantomeno che essa non sia più visibile dall'utente. Quando l'activity tornerà ad essere eseguita in *foreground* verrà chiamata la funzione `onResume`;
- `onStop`: viene chiamata quando l'activity non è più visibile dall'utente. L'activity in stato di *stopped* è ancora presente in memoria. Da questo stato l'activity può tornare ad essere eseguita, e quindi verrà chiamata la funzione `onRestart`, oppure può essere distrutta, e quindi verrà chiamata la funzione `onDestroy`;
- `onDestroy`: viene chiamata prima che l'activity venga distrutta, ed è lo stato finale del ciclo di vita dell'activity.

2.2.2 Fragment

Un Fragment rappresenta una porzione dell'interfaccia utente dell'applicazione. Esso definisce e gestisce il suo layout, ha un ciclo di vita proprio e può gestire i suoi eventi di input.

Un fragment non può tuttavia esistere da solo, ma deve essere inserito all'interno di un'activity o di un altro fragment.

I fragments rendono l'applicazione più modulare e riusabile, in quanto permettono di dividere la UI in più parti gestibili separatamente e riutilizzabili all'interno di diverse activities. [11]

Dato che un fragment può esistere solo all'interno di una activity, il ciclo di vita di uno e dell'altra si innestano.

I cicli di vita del fragment quindi andranno di pari passo a quelli della activity in cui è inserito, e quindi il Fragment invocherà i metodi `onCreate`, `onStart` e `onResume`. [12] [20]

La classe fragment però possiede anche dei metodi propri, i più importanti sono:

- `onAttach`: viene chiamato quando il Fragment viene aggiunto ad un `FragmentManager` ed è collegato all'activity ospite;
- `onCreateView`: viene chiamato per restituire la `View` in cui è presente il layout dell'interfaccia. Tra i parametri di input riceve un `LayoutInflater` la cui utilità è quella dell'eleborazione di un layout definito in un file .xml;
- `onActivityCreated`: in questo momento la creazione dell'activity è completa ed il fragment potrà interagire con essa;
- `onDestroyView`: indica la rimozione dell'interfaccia utente dal fragment, e prepara quest'ultimo allo scollegamento dall'activity;
- `onDetach`: viene chiamato quando il frammento viene rimosso dal `FragmentManager` e quindi scollegato dall'activity ospite.

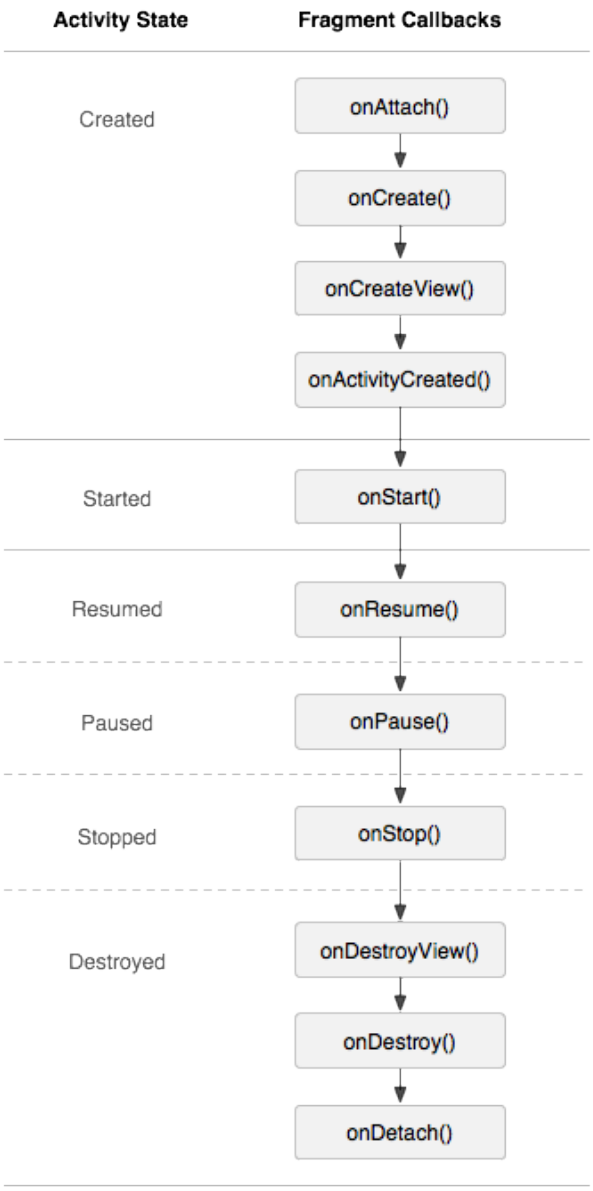

Figura 2.2. Ciclo di vita di un *fragment*

2.2.3 Intents

Un intent è un messaggio che viene usato da una componente dell'applicazione per richiedere un'azione da un'altra componente della stessa (o di un'altra) applicazione. [13]

Un intent può essere usato per:

- Avviare un'activity;
- Avviare un *service;*
- Inviare un messaggio in broadcast;

Esistono due tipi di *intents*:

- **Espliciti**: specificano quale componente verrà avviata per soddisfare l'intent. Un classico uso di un intent esplicito è per avviare un'activity a partire da un'altra activity;
- **Impliciti**: specificano solo un'azione generale, senza specificare la componente che andrà ad eseguire tale azione. Un esempio è quando si vuole effettuare un'azione tramite un'altra applicazione, e quindi si definisce l'azione, mostrando poi una finestra di dialogo in maniera tale che sia l'utente a scegliere con che applicazione eseguire tale azione.

Extras

Un intent porta con se delle coppie chiave-valore che possono essere inserite dalla componente che crea l'intent, e lette dalla componente che lo riceve.

2.2.4 Service

Un *service* è una componente, non dotata di interfaccia grafica, all'interno di un'applicazione che permette di eseguire un operazione a lungo termine in *background*. [14]

Una volta eseguito, un *service* può rimanere in esecuzione per un determinato lasso di tempo, anche in caso l'utente abbia cambiato applicazione.

Un *service* può essere:

- Foreground: esegue delle operazioni che possono essere notate dall'utente, ad esempio l'esecuzione di musica quando l'applicazione viene eseguita in background. Affinché il *service* funzioni, l'applicazione deve mostrare un notifica durante tutta l'esecuzione del *service*;
- Background: esegue delle operazioni che non sono direttamente osservabili dall'utente, ad esempio la conversione da indirizzo sotto forma di stringa in coordinate.

2.3 Gli stati dell'utente

L'applicazione implementa una macchina a stati finiti, il cui diagramma è mostrato di seguito.

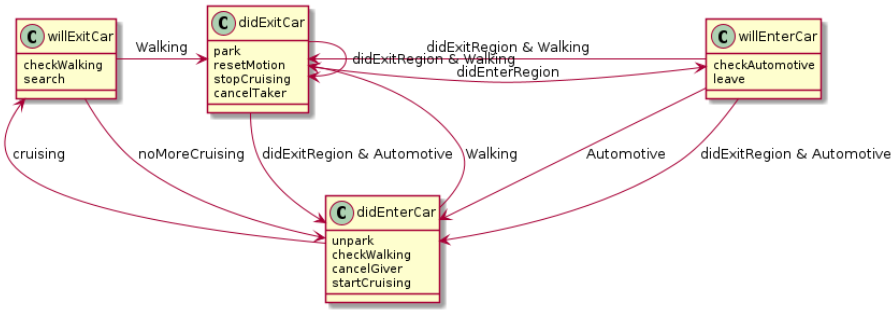

Figura 2.3. Macchina a stati

Gli stati in cui può essere l'utente sono i seguenti:

- `willExitCar`: l'utente ha parcheggiato e quindi sta per lasciare la macchina;
- `didExitCar`: l'utente ha parcheggiato dalla macchina ed è uscito da quest'ultima;
- `willEnterCar`: l'utente si sta avvicinando alla macchina a piedi e sta per entrare;
- `didEnterCar`: l'utente è entrato nella macchina parcheggiata e sta per uscire dal parcheggio.

2.4 Diagramma UML delle classi

Di seguito viene mostrato il diagramma UML delle delle classi, in cui vengono rappresentate le classi con cui il tirocinante ha interagito per l'implementazione delle funzionalità.

Le classi rappresentate da rettangoli a sfondo blu rappresentano le classi che sono state create durante il tirocinio, le altre rappresentano classi già esistenti nel progetto, che sono state inserite nel diagramma in quanto citate all'interno della relazione.

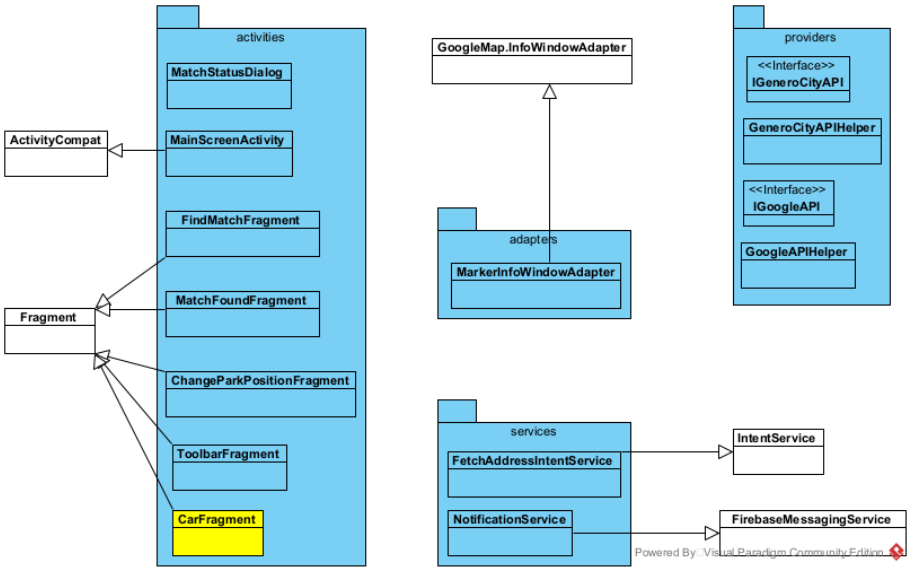

Figura 2.4. Diagramma UML delle classi

2.5 Librerie Android Utilizzate

2.5.1 Retrofit

Retrofit è una libreria Java che permette di gestire in maniera semplice chiamate e risposte ad API REST.

La libreria Retrofit necessità l'implementazione di tre tipologie di classi:

- Una classe per ogni oggetto che si vuole modellare, che verrà usata per la conversione da `json` ad oggetto java e viceversa;
- Un'interfaccia nella quale vengono definite le possibili API da chiamare;
- Una classe che implementa l'interfaccia definita nel punto precedente, ed in cui viene creato il Client Retrofit.

All'interno dell'interfaccia, ogni metodo rappresenta una possibile chiamata alle API. Vengono usate delle annotazioni del tipo `@GET("/me/")` per definire l'endpoint della chiamata alle API ed il metodo HTTP. [21]

All'interno della classe definita nel terzo punto, viene costruito il Client nella seguente maniera

```
Retrofit retrofit = new Retrofit.Builder()
    .baseUrl(GCEnv.SERVER_URL)
    .addConverterFactory(ScalarsConverterFactory.create())
    .addConverterFactory(GsonConverterFactory.create(new GsonBuilder()
        .excludeFieldsWithoutExposeAnnotation().create()))
    .client(client)
    .build();
return retrofit.create(IGeneroCityAPI.class);
```

2.5.2 Gson

`Gson` è una libreria Java (nel progetto integrata con Retrofit) che permette di convertire oggetti Java nella rappresentazione in oggetti `Json` e viceversa.

Tali oggetti Java modellano i vari elementi nel mondo reale trattati dal sistema. Gli oggetti rilevanti all'interno del lavoro svolto sono `Park` e `Match`. Visto che questi presentano un elevato numero di campi, di seguito verranno descritti quelli trattati maggiormente nel processo di sviluppo.

Park

Modella un parcheggio effettuato da un utente.

I campi più rilevanti sono:

- `parktype`: indica il tipo di parcheggio, che può essere sconosciuto, a spina, parallelo o a pettine;
- `parkid`: intero che rappresenta il codice univoco del parcheggio;
- `parklat`: latitudine del parcheggio;
- `parklon`: longitudine del parcheggio;
- `starttime`: timestamp formattato come stringa che indica l'istante in cui l'utente ha parcheggiato;
- `endttime` timestamp formattato come stringa che indica l'istante in cui l'utente ha lasciato il parcheggio;
- `takerid`: codice univoco dell'utente che ha trovato il parcheggio tramite una "ricerca posto". Vale 0 se non esiste;
- `giverid`: codice univoco dell'utente che ha lasciato il posto tramite un "lascia posto". Vale 0 se non esiste.

Match

Modella un match, e presenta i seguenti campi:

- `takerid`: codice univoco dell'utente Taker;
- `takercid`: codice univoco dell'auto con cui il Taker ha effettuato il match;
- `takerlat`: latitudine della posizione in cui il Taker intende trovare parcheggio;
- `takerlon`: longitudine della posizione in cui il Taker intende trovare parcheggio;
- `giverid`: codice univoco dell'utente giver
- `givercid`: codice univoco dell'auto con cui il Giver ha effettuato il match;
- `giverlat`: latitudine del parcheggio che il Giver intende lasciare;
- `giverlon`: longitudine del parcheggio che il Giver intende lasciare;
- `giverparkid`: codice univoco del parcheggio in cui è situata l'auto del giver;
- `parktype`: tipologia di parcheggio (si veda il campo omonimo in `park`)

- `schedule`: timestamp che indica l'orario previsto in cui il Taker dovrebbe arrivare dal giver;
- `status`: indica lo stato del match nel momento corrente, il quale può assumere i seguenti valori:
  - `waiting`: nel match è presente solamente il Taker o il Giver in attesa che il sistema trovi un utente con il ruolo opposto compatibile;
  - `running`: il match è stato effettuato, e quindi sono presenti sia Taker che Giver, ed è in corso di esecuzione;
  - `deleted`: il match, precedentemente in stato di `waiting` è stato annullato dal Taker o dal Giver;
  - `expired`: il match è scaduto perchè il Taker non ha raggiunto la posizione del Giver entro 15 minuti;
  - `success`: il match è stato completato con successo;
  - `unsuccess-giver`: il match, precedentemente in status di `running`, è stato annullato dal Giver;
  - `unsuccess-taker`: il match, precedentemente in status di `running`, è stato annullato dal Taker;
- `updated`: timestamp formattato come stringa che indica l'istante in cui l'oggetto match ha ricevuto un aggiornamento di un qualche campo;
- `historyid`: codice univoco del match.

Capitolo 3

Progettazione

3.1 Architettura del sistema

L'architettura utilizzata da GeneroCity è del tipo *client-server*.

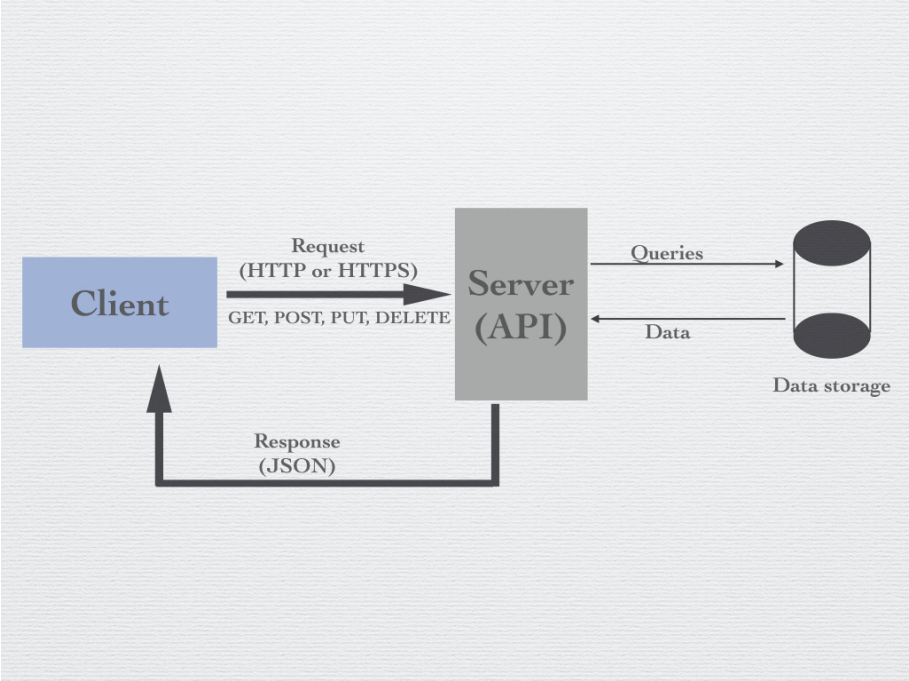

Figura 3.1. Diagramma di un'architettura client-server con API REST [15]

Il *client*, ovvero il dispositivo Android dell'utente, comunica con il *server* mediante delle chiamate alle API REST. La comunicazione permette al client l'accesso a delle risorse, ovvero a delle informazioni che il server decide di rendere disponibili per il client. [16]

La comunicazione avviene tramite il protocollo HTTP.

Una chiamata alle API REST contiene i seguenti elementi:

- ***Uniform Resource Identifier*** (*URI*): indirizzo che indica dove trovare la risorsa su Internet;
- **Metodo** `HTTP`: indica l'operazione che bisogna effettuare con la risorsa indicata dall'URI. I principali metodi sono quattro:
  - `GET`: permette di recuperare la risorsa, la quale verrà mostrata in uno specifico formato (solitamente XML o JSON);
  - `POST`: sostituisce la risorsa con un'altra risorsa, solitamente indicata nel body;
  - `PUT`: crea una nuova risorsa;
  - `DELETE`: elimina la risorsa indicata.
- `Header`: è una parte del messaggio che contiene delle informazioni chiave per entrambi il server ed il client. Tali informazioni possono essere ad esempio il formato della risposta, la chiave dell'API o l'indirizzo IP del server.
- `Body` (opzionale): viene utilizzato per inviare o ricevere ulteriori parti di dati.

3.2 Funzionalità minori

3.2.1 Implementazione della lista dei parcheggi

Lo scopo dell'implementazione dell'interfaccia di cui si andrà a parlare è quello di mostrare agli utenti uno storico dei parcheggi che sono stati effettuati.

Tali parcheggi vengono inseriti all'interno di una lista, implementata in Android mediante una `RecyclerView`, i cui i singoli elementi sono cliccabili.

Il click su un elemento che modella un singolo parcheggio porta ad una schermata contenente maggiori dettagli su tale parcheggio.

Figura 3.2. Storico parcheggi

Negli elementi che compongono la lista sono presenti tre dati:

- L'indirizzo, composto da via e civico, dove la macchina è stata parcheggiata
- L'orario e la data in cui la macchina è stata parcheggiata
- L'orario e la data in cui l'utente è uscito dal parcheggio

Il primo elemento della lista modella il parcheggio in cui l'auto dell'utente è correntemente situata. Essendo il parcheggio ancora in corso, non viene mostrato l'orario e la data in cui l'utente lo ha terminato.

L'implementazione della lista dei parcheggi è stata sviluppata insieme ad altri due membri del team di sviluppo, a differenza delle altre funzioni di cui si parlerà nel resto della relazione.

Paginazione dei risultati

Prima dell'implementazione della funzionalità che verrà descritta, lo storico mostrava solamente gli ultimi 16 parcheggi effettuati dall'utente.

Questo avviene in quanto la chiamata alle API per ottenere la lista dei parcheggi effettuati con una macchina (`/car/{cid}/park/`) richiede opzionalmente come *query parameter* un id del parcheggio.

Se questo parametro non viene inserito, l'API restituisce la lista degli ultimi 16 parcheggio effettuati, altrimenti restituisce la lista dei 16 parcheggi effettuati precedentemente al parcheggio passato come parametro.

Tramite questa logica è stato possibile paginare lo storico del parcheggi.

L'utente quando apre la schermata visualizza solo i 16 parcheggi più recenti. Quando effettua uno *scroll* ed arriva in fondo alla pagina, l'app effettua una chiamata alle API inserendo come parametro l'ultimo parcheggio della lista, in maniera da recuperare i 16 parcheggi antecedenti all'ultimo parcheggio mostrato.

Durante la richiesta viene mostrata una progress bar, che viene rimossa quando i risultati sono disponibili.

Una volta che ciò accade, vengono inseriti ulteriori 16 parcheggi, o meno se i parcheggi totali sono inferiori, all'interno dello storico.

L'utente può continuare ad effettuare uno *scroll* finché non visualizza tutti i parcheggi effettuati dalla macchina corrente.

3.2.2 Indicatori di scambio nel parcheggio

Figura 3.3. Logo di GeneroCity.

Aprendo lo storico dei parcheggi effettuati dall'utente, non era possibile capire a colpo d'occhio quali fossero i parcheggi effettuati tramite uno scambio con un altro utente, e quali fossero quelli effettuati senza uno scambio

Per ovviare a questa problematica, sono state inserite due icone alla destra di ogni elemento della lista dello storico dei parcheggi, che fanno capire all'utente la natura dello scambio (se presente). È possibile vedere una rappresentazione delle icone nella Figura 3.2

L'icona presente in Figura 3.3 è presente se il parcheggio è stato ottenuto mediante il completamento di un match; mentre il testo che indica "+100" è presente se il parcheggio è stato lasciato ad un utente mediante il completamento di un match.

3.3 Gestione dei matches

Nella seguente sezione verranno mostrate e documentate le interfacce che sono state realizzate ed il processo che ha portato alla loro realizzazione.

In particolare, sono state realizzate le schermate che permettono all'utente di:

- Comunicare al sistema l'intenzione di lasciare il posto ad un altro utente (diventare giver);
- Chiedere al sistema di trovare un posto disponibile da occupare (diventare Taker);
- Permettere di vedere se il sistema stia attualmente cercando un utente con ruolo opposto compatibile con cui scambiare il posto, ed eventualmente annullare tale ricerca;
- Permettere di vedere informazioni utili circa il match in corso, come la posizione del Taker e del Giver, il percorso tra il Taker ed il Giver ed il tempo stimato di arrivo.

Cos'è un match Quando un utente parcheggiato comunica all'app di voler lasciare il posto, egli diventa un Giver.

Quando invece un utente comunica all'app di voler cercare un posto disponibile da occupare, diventa un Taker.

Quando un Taker ed un Giver sono compatibili, viene effettuato un *match*. Quando si è all'interno di un match in corso, il Giver attende l'arrivo del Taker. Quando il Taker raggiunge il Giver, quest'ultimo può lasciare il posto, che verrà occupato dal Taker. In tal caso il match viene completato con successo.

Il Taker ha un tempo di 15 minuti per raggiungere il Giver prima che il match venga annullato.

3.3.1 Struttura delle schermate

Le schermate che sono state progettate hanno tutte la medesima struttura, rappresentata nella Figura 3.4.

Figura 3.4. Wireframe delle schermate implementate.

In alto è presente l'app bar, la quale contiene il nome dell'applicazione e dei pulsanti per effettuare delle azioni contestuali alla schermata in cui l'utente si trova.

Sottostante all'app bar è presente la mappa, ovvero un `Fragment` contenente una mappa fornita da Google su cui è possibile mostrare degli elementi.

Nella parte inferiore dello schermo è presente un `Fragment` che verrà sostituito a seconda della natura della schermata.

Il `Fragment` che viene mostrato all'apertura dell'applicazione è il `CarFragment` , già esistente all'inizio del tirocinio.

I metodi per effettuare le operazioni sui `Fragments` sono inseriti all'interno di una classe `FragmentUtility`.

Nei paragrafi successivi si parlerà proprio dei vari `Fragments` che sono stati sviluppati, e del modo in cui sono stati integrati all'interno del resto dell'interfaccia.

3.3.2 Registrare e rimuovere un parcheggio

L'applicazione registra un parcheggio quando una delle due condizioni si verifica:

- L'utente preme il pulsante `PARCHEGGIATA` nella schermata principale;
- L'utente è in stato di `DidExitCar`.

L'applicazione rimuove un parcheggio quando una delle due condizioni si verifica:

- L'utente preme il pulsante `IN USO` nella schermata principale;

- L'utente è in stato di `DidEnterCar`.

Il bottone per parcheggiare e rimuovere il parcheggio manualmente viene inserito in quanto il sistema potrebbe errare nella rilevazione automatica dello stato dell'utente (parcheggiato o non parcheggiato), e quindi all'utente viene data la possibilità di riallineare il sistema alla realtà, comunicando lo stato della macchina come "parcheggiata" o "in uso" mediante dei pulsanti dedicati.

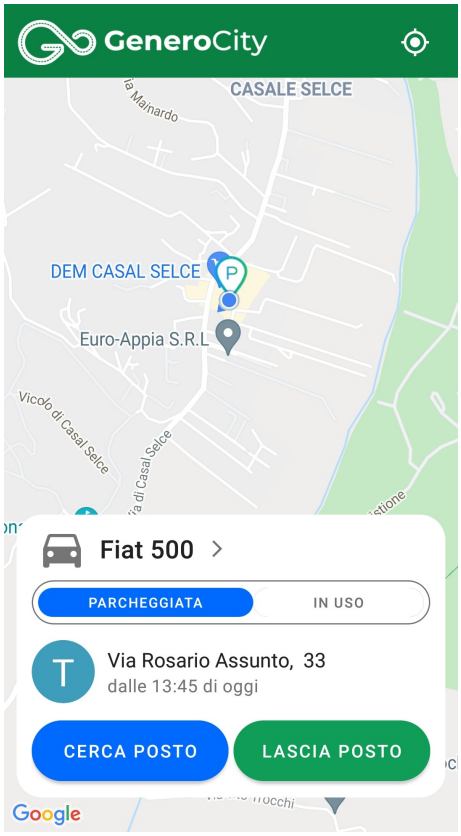

Figura 3.5. Interfaccia della pagina principale quando l'auto è parcheggiata.

In questo caso l'auto è parcheggiata e l'utente può premere sul pulsante `IN USO` per rimuovere tale parcheggio e comunicare all'applicazione che l'auto è attualmente in uso.

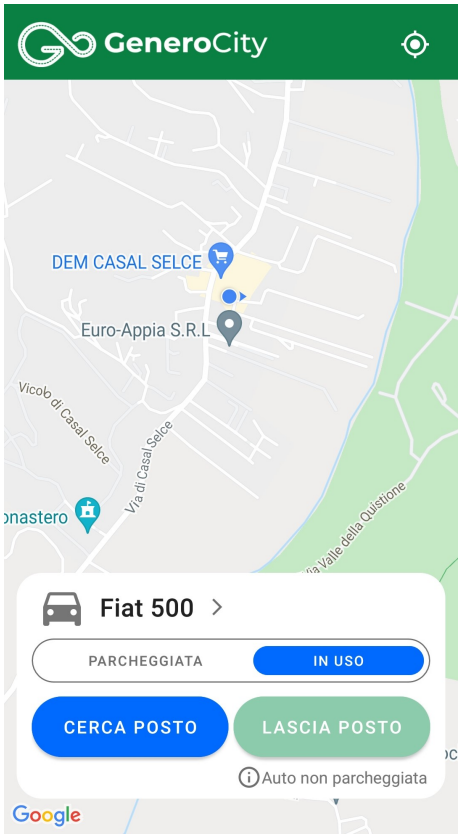

Figura 3.6. Interfaccia della pagina principale quando l'auto è in uso.

In questo caso l'auto è in uso, ma l'utente può premere il pulsante `PARCHEGGIATA` per parcheggiare l'auto nella sua posizione attuale.

Test effettuati

Per verificare se ci fossero dei problemi legati all'usabilità dell'interfaccia, sono stati effettuati dei test sul campo con degli utenti.

Tali utenti sono stati inseriti in un'ipotetica situazione in cui il sistema segnalava che la loro macchina fosse in uso, quando in realtà questa era parcheggiata. Si è quindi chiesto loro di comunicare al sistema il fatto che l'auto fosse parcheggiata nel luogo in cui ora loro si trovavano.

Agli stessi utenti è stato effettuato anche il test inverso, e quindi è stato detto loro di comunicare al sistema che la macchina non era realmente parcheggiata in un luogo, ma che fosse attualmente in uso.

Tutti i test sono andati a buon fine e quindi si è concluso che l'interfaccia fosse utilizzabile in maniera corretta.

3.3.3 Ricerca di un match

La classe `FindMatchFragment` modella un `Fragment` che viene mostrato all'utente quando è in corso una ricerca, da parte del sistema, di un utente compatibile con cui effettuare uno scambio.

Si accede a tale schermata in tre modi:

- Premendo il bottone “cerca posto” nella schermata inziale.
- Premendo il bottone “lascia posto” nella schermata iniziale.
- In caso di ripristino dello stato dell’*activity* dopo che questa è stata chiusa.

Il `FindMatchFragment` rimpiazza sempre il `CarFragment`.

Per evitare duplicazione di codice, il `Fragment` utilizza solo un layout `xml` il quale viene modificato a tempo di esecuzione a seconda del ruolo dell’utente, in maniera da mostrare le informazioni utili al Taker o al giver.

La *`view`* del `fragment` mostra all’utente i seguenti elementi:

- Una *label*, differente da Taker a Giver, che indica cosa l’utente stia cercando;
- Una *progress bar* circolare che mostra all’utente che il processo di ricerca è in corso;
- Un bottone per annullare la ricerca.

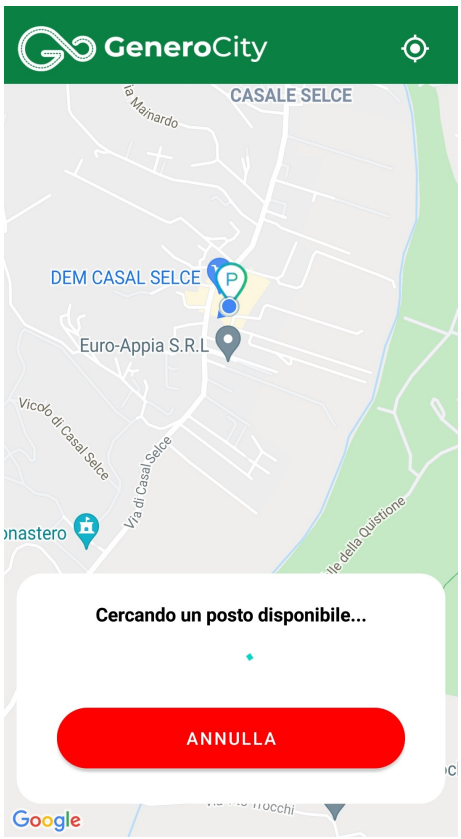

Figura 3.7. Interfaccia mostrata durante la ricerca di un Giver.

In caso del Taker, la label mostra la scritta “Cercando un posto disponibile”

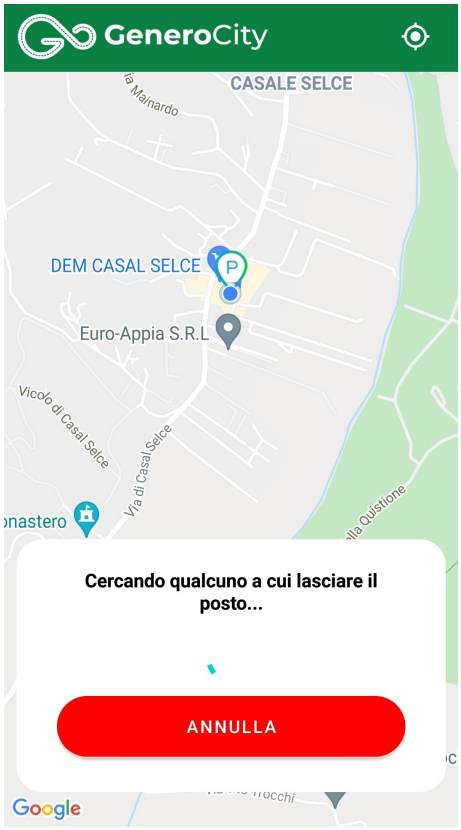

Figura 3.8. Interfaccia mostrata durante la ricerca di un Taker.

In caso del Giver, la label mostra la scritta "Cercando qualcuno a cui lasciare il posto".

3.3.4 Match effettuato

La classe `MatchFoundFragment` modella un `Fragment` che viene mostrato all'utente quando il sistema ha trovato un utente compatibile con cui effettuare uno scambio, e tale scambio è in corso di esecuzione.

La schermata può essere raggiunta dall'utente in tre modi:

- Quando il sistema, per un utente nel ruolo di Taker, trova un utente Giver compatibile;
- Quando il sistema, per un utente nel ruolo di Giver, trova un utente Taker compatibile;
- In caso di ripristino dello stato dell'*activity* dopo che questa è stata chiusa.

Nei primi due casi il `MatchFoundFragment` rimpiazza il `FindMatchFragment`.

Nel terzo caso la UI si trova nella schermata iniziale, dove il Fragment mostrato sotto la mappa è il `CarFragment`. Per questo, quando la UI verrà ripristinata, il `MatchFoundFragment` ripristina il `CarFragment`.

Anche qui è stato utilizzato un solo layout `xml` per evitare duplicazione di codice.

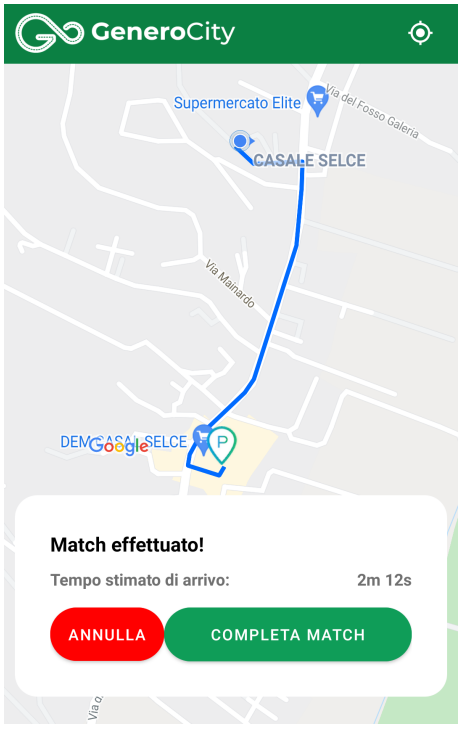

Figura 3.9. Interfaccia mostrata al Taker quando ha trovato un Giver compatibile.

Nel `Fragment` mostrato al Taker sono presenti quattro elementi:

- Una *label* che indica che un match è stato effettuato, e quindi che è presente un Giver compatibile pronto a cedere il posto auto;
- Una *label* che indica il tempo stimato circa l'incontro tra i due utenti;
- Un bottone con dicitura "Annulla" che permette di annullare il match in corso. Sia il Taker che il Giver verranno portati nella pagina principale dell'applicazione;
- Un bottone con dicitura "Completa match" che permette di segnalare il match come confermato. Sia il Taker che il Giver verranno portati nella pagina principale dell'applicazione;

Nella mappa sono presenti tre elementi:

- L'icona che rappresenta il punto in cui è parcheggiato il *giver,* ovvero il punto che il Taker dovrà raggiungere;
- L'icona che rappresenta la posizione aggiornata del Taker, ovvero dell'utente che visualizza tale schermata;
- Il percorso che il Taker deve effettuare per arrivare dal giver;

Percorso per arrivare dal giver

Il percorso mostrato sulla mappa, che indica la strada che il Taker deve percorrere per arrivare dal Giver mediante un'automobile, è stato ottenuto mediante le *Google Directions API.*

Il Taker, quando trova un Giver compatibile, effettua una chiamata alle Directions API all'url

`https://maps.googleapis.com/maps/api/directions/json`

in cui vengono inseriti come *query parameters*, i seguenti campi:

- `origin`: le coordinate (o indirizzo) del punto d'origine, ovvero l'attuale posizione del Taker;
- `destination`: le coordinate (o indirizzo) del punto destinazione, ovvero la posizione del Giver;
- `key`: la chiave per poter utilizzare le API.

Tale chiamata ritorna un oggetto json in cui sarà presente, in aggiunta ad altre informazioni, una codifica delle `Polylines` da inserire sulla mappa per mostrare il percorso tra i due punti.

Una `Polyline` è un segmento che mostra sulla mappa il collegamento tra due punti. La serie di `Polylines` che deve essere mostrata viene codificata da un algoritmo.

Listing 3.1. Lista di `Polylines` codificata

```
kox~Fq_kjALDGb@_@E]MMG]]qAuBq@eAY_@OOe@SqC}@yC}@gC}@o@KcGQqCIc@Ia
A]aGsC}E{BeBw@oAg@UCQYg@c@c@g@[g@ESAwA@aA?kCLsCHiAFQ[k@W_@W_@c@}@
_@aAs@eCWeBMe@GMaBgBeC}Be@]{A{@}@s@i@UiDeAeAO]Oc@WN{CJw@Z}@Ve@pAc
BJ[Da@AoBHm@d@}Ab@}@lAmBb@m@SQMAe@Fs@Ca@?y@R}@f@yAlA[NqBh@sB‘AQ@W
ID_@@SFSFI\\OLQHg@?UG[u@wAO]UkAQg@]c@o@k@oBkAq@e@SWk@}@{@{BMw@PoF
BgA?{@Es@CYG]u@G
```

Il messaggio viene decodificato in un vettore di oggetti `LatLng`, che verranno utilizzati per costruire le varie `Polylines` sulla mappa, che nell'insieme formeranno il percorso mostrato in Figura 3.9.

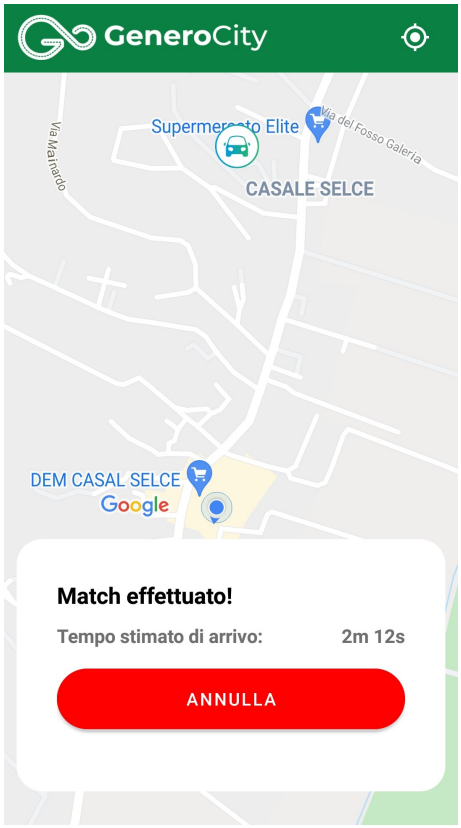

Figura 3.10. Interfaccia mostrata al Giver quando ha trovato un Taker compatibile.

Per quanto riguarda il `Fragment` mostrato al Giver, sono presenti tre elementi:

- Una *label* che indica che un match è stato effettuato, e quindi che è presente un Taker compatibile a cui cedere il posto;
- Una *label* che indica il tempo stimato circa l'incontro tra i due utenti;
- Un bottone con dicitura "Annulla" che permette di annullare il match in corso.

Nella mappa è presente il *marker* che rappresenta la posizione, aggiornata in tempo reale, del Taker.

Aggiornamento della posizione del taker

Quando il Taker comincerà a guidare dirigendosi verso il Giver, quest'ultimo vedrà sull'interfaccia il *marker* che rappresenta la posizione del Taker muoversi in tempo reale.

Il risultato viene raggiunto nella seguente maniera:

1. Siano T e G rispettivamente il Taker ed il Giver all'interno di un match in stato Running;
2. Il client di T effettua, ogni `5000ms`, una chiamata alle API con endpoint `"/car/{cid}/position"` dove `{cid}` è il codice univoco dell'auto con cui il Taker vuole parcheggiare. La chiamata alle API ha un metodo `HTTP POST`, e

richiede nel `body` un oggetto `json` con le informazioni relative alla posizione da inviare, che verranno documentate in dettaglio nella sottosezione 4.1.1.

3. Il server, alla ricezione di tale chiamata API, invia una notifica push al client del Giver, del tipo:

```
{x-gc-category:TAKERPOSITION, lat:42.2412, lon:12.4253, eta:2674}
```

4. Il client di G, alla ricezione di ogni notifica push, estrae la nuova posizione di T ed aggiorna il rispettivo marker sulla mappa tramite un'animazione della durata di `5000ms`.

Tempo previsto per l'incontro dei due utenti

Su entrambe le interfacce viene mostrata l'informazione relativa al tempo rimanente stimato prima che i due utenti si incontrino.

Il Taker, immediatamente prima di effettuare una chiamata alle API per comunicare la posizione aggiornata al server, come visto nel paragrafo precedente, effettua una chiamata alle *DistanceMatrix* Google APIs, all'url

```
https://maps.googleapis.com/maps/api/distancematrix/json
```

in cui vengono inseriti, come *query parameters* i seguenti valori:

- `origins`: le coordinate del punto d'origine, ovvero l'attuale posizione del Taker;
- `destinations`: le coordinate del punto destinazione, ovvero la posizione del Giver;
- `departure-time`: l'istante di partenza, impostato sempre su `now`;
- `key`: la chiave per poter utilizzare le API.

Nell'oggetto json che il server manderà nel `body` della risposta alla chiamata API, sono presenti tre valori:

1. Un valore che indica la distanza percorribile su strada, in metri, tra i due punti;
2. Un valore che indica la durata del viaggio, in secondi, senza considerare il traffico;
3. Un valore che indica la durata del viaggio, in secondi, considerando il traffico.

Viene preso in considerazione solo il terzo valore che, una volta ottenuto, viene mostrato sull'interfaccia del Taker. Quando il Taker, immediatamente dopo, effettua la chiamata per aggiornate la posizione (sezione 3.3.4), inserisce tale valore nel campo `eta` all'interno del `body` della richiesta.

Il Giver, alla ricezione della notifica vista in sezione 3.3.4, formatta il valore e lo inserisce nella sua interfaccia.

3.3.5 Annullamento di un match

Un match, che sia in fase di ricerca o in corso, può essere annullato dalla parte di entrambi gli utenti coinvolti, mediante la pressione del bottone "Annulla".

Figura 3.11. Messaggio mostrato in caso il match non sia andato a buon fine.

Annullamento di un match in fase di ricerca

In caso di annullamento di un match in fase di ricerca, l'utente viene riportato alla schermata iniziale e viene mostrato una `DialogWindow` (Figura 3.11) che comunica all'utente il corretto annullamento del match.

Annullamento di un match in corso

Visto che un match in corso comprende due utenti, l'annullamento da parte di un utente implica che il match venga annullato anche per il secondo utente.

Durante tutta la vita del frammento `MatchFoundFragment`, ovvero durante tutto il tempo in cui gli utenti sono all'interno di un match, un frammento di codice viene eseguito ogni 2 secondi, il cui scopo è quello di controllare se l'utente sia ancora in un match o meno. Questo avviene mediante una chiamata alle API.

Il flow dell'applicazione è quindi il seguente:

1. Siano T e G rispettivamente un Taker ed un Giver all'interno di un match
2. T annulla il match premendo sul pulsante "Annulla";
3. T viene riportato alla pagina principale e gli viene mostrata la *`DialogWindow`* che mostra i corretto annullamento del match (Figura 3.11);
4. Al più 2 secondi dopo dall'annullamento del match da parte di T, il codice che controlla lo stato del match riconosce che l'utente G non è più all'interno di un match;
5. G viene riportato alla pagina principale e gli viene mostrata la medesima schermata di corretto annullamento del match.

3.3.6 Completamento di un match

Un match può essere completato solamente se è in corso, e tale azione può essere effettuata solamente dal Taker, mediante la pressione del pulsante "Completa" nel `MatchFoundFragment`.

Il flow dell'applicazione è il seguente:

1. Siano T e G rispettivamente un Taker ed un Giver all'interno di un match;
2. T completa il match premendo sul pulsante "Completa"';

3. T viene riportato alla pagina principale e gli viene mostrata la *DialogWindow* che mostra il corretto completamento del match (Figura 3.12);
4. Al più 2 secondi dopo dall'annullamento del match da parte di T, il codice che controlla lo stato del match riconosce che il match che comprende G è stato completato;
5. G viene riportato alla pagina principale e gli viene mostrata la *DialogWindow* che mostra il corretto completamento del match (Figura 3.12;

Figura 3.12. Messaggio mostrato in caso il match sia stato completato con successo.

3.4 Modifica posizione parcheggio

Nello stato dell'applicazione prima dell'implementazione della modifica di cui si andrà a parlare, per l'utente non era possibile modificare la posizione di un parcheggio effettuato precedentemente.

3.4.1 Prima iterazione

Info Window

La `InfoWindow` è l'elemento che appare quando si clicca sopra un *marker* all'interno di una mappa. Tale elemento può essere personalizzato a piacere in maniera da contenere le informazioni che possono risultare più utili all'utente. Viene quindi aggiunta una funzionalità che permette all'utente di modificare la posizione del parcheggio in cui l'auto è attualmente parcheggiata.

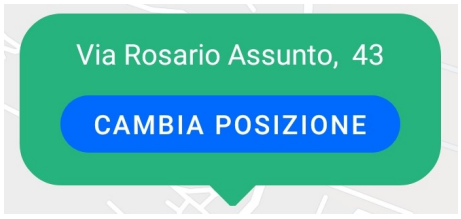

Figura 3.13. La `InfoWindow` relativa al marker del parcheggio

In tale elemento sono presenti tre elementi:

- L'indirizzo in cui è situato il *marker*;
- La distanza che separa la posizione del marker dalla posizione attuale dell'utente;

- Un pulsante che permette di modificare la posizione del *marker*;

UI per la modifica del parcheggio

Figura 3.14. Prima iterazione della schermata per la modifica del parcheggio.

L'utente può scegliere la nuova posizione del parcheggio muovendo la mappa. Il *marker* che segna la nuova posizione rimarrà sempre al centro dello schermo.

Si può accedere a tale schermata in due modi:

- Tenendo premuto a lungo sulla mappa;
- Premendo sul bottone mostrato in Figura 3.13 all'interno della `InfoWindow`.

La schermata presenta vari elementi:

- Un *fragment* in alto, sotto alla *app bar*, che indica che l'utente si trova all'interno della schermata per la modifica del parcheggio;
- Una label che indica la nuova posizione del parcheggio, ovvero la conversione da coordinate ad indirizzo del punto sulla mappa in cui è situato il *marker*.
- Un pulsante "Annulla" che consente di annullare la modifica del parcheggio. Una volta premuto tale pulsante l'utente verrà reindirizzato alla pagina principale ed il parcheggio non sarà modificato;

- Un pulsante “Conferma” che consente di completare la modifica del parcheggio. Una volta premuto tale pulsante l’utente verrà reindirizzato alla pagina principale, e la posizione del parcheggio sarà modificata con la nuova posizione scelta dall’utente.

Icona del marker

Figura 3.15. Marker di un parcheggio effettuato.

La Figura 3.15 mostra il marker che indica la posizione di un parcheggio sulla mappa.

Figura 3.16. Marker della potenziale nuova posizione del parcheggio.

La Figura 3.16 mostra l’icona presente al centro dello schermo quando l’utente si trova nella *activity* per modificare la posizione del parcheggio. In questo caso l’icona, molto simile a quella in Figura 3.15, presenta un’ombra sottostante in modo da far capire all’utente, mediante l’utilizzo di una metafora, che il marker non è “fissato” sulla mappa ma può essere spostato.

Test

Per verificare l’usabilità dell’interfaccia sono stati effettuati dei test con degli utenti.

Tali utenti sono stati inseriti in una situazione ipotetica, ed è stato chiesto loro di modificare la posizione del parcheggio nell’applicazione.

Gli utenti non sono riusciti a capire in che maniera raggiungere la schermata per modificare la posizione del parcheggio.

Una volta fatti entrare nella schermata desiderata, gli utenti sono riusciti a modificare il parcheggio, ma alcuni utenti hanno fatto difficoltà a trovare l’indirizzo comunicato.

Problemi rilevati

Dai test effettuati sono state rilevate le seguenti problematiche:

- Il *marker* del parcheggio non sembra cliccabile;
- La pressione lunga sulla mappa è un comando nascosto;

- Cercare la via spostandosi sulla mappa è utile per modifiche di distanze minori, ma è poco comodo in caso l'utente debba modificare il parcheggio in una posizione completamente differente.

3.4.2 Seconda iterazione

I primi due problemi individuati nella prima iterazione del prototipo sono stati risolti inserendo un `FloatingActionButton` nella pagina dei dettagli del parcheggio corrente. Visto che è possibile modificare solamente tale parcheggio, il `FloatingActionButton` non viene mostrato all'interno delle schermate di dettaglio dei parcheggi effettuati precedentemente.

Figura 3.17. Pagina dei dettagli di un parcheggio con `FloatingActionButton`

È stata presa questa decisione in quanto, nei test effettuati, gli utenti navigavano nella pagina contenente i dettagli di un parcheggio cercando un modo per modificare l'indirizzo.

Il terzo problema è stato risolto rendendo modificabile l'indirizzo presente nel `ChangeParkPositionFragment`. In questa maniera l'utente è sempre al corrente del nuovo indirizzo del parcheggio, ma può cliccare su di esso per modificarlo tramite tastiera.

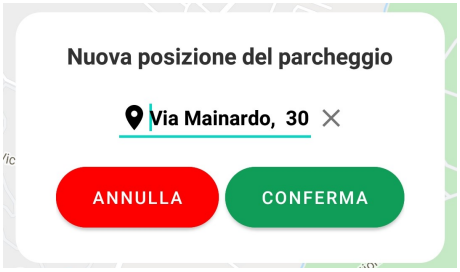

Figura 3.18. `ChangeParkPositionFragment` con indirizzo modificabile

Test

Sono stati effettuati ulteriori test utilizzando la seconda iterazione dell'interfaccia e gli utenti sono riusciti a modificare il parcheggio utilizzando il `FloatingActionButton` nella pagina dei dettagli del parcheggio corrente.

Una volta entrati nella schermata per la modifica del parcheggio, gli utenti hanno spostato la mappa manualmente quando si chiedeva loro di modificare la posizione del parcheggio in una posizione molto vicina a quella attuale, mentre hanno utilizzato il campo di testo editabile quando è stato chiesto loro di modificare la posizione del parcheggio in una via più distante dalla posizione attuale, e quindi più difficile da trovare muovendo manualmente la mappa.

Non sono stati trovati ulteriori problemi di usabilità, e quindi l'interfaccia può essere considerata definitiva.

Capitolo 4

Implementazione

In questo capitolo si discuteranno i dettagli implementativi di una funzionalità in particolare di quelle sviluppate. Si è scelta la funzionalità di aggiornamento della posizione del Taker in real-time sull'interfaccia mostrata al Giver.

4.1 Aggiornamento posizione Taker in real-time

Il flusso della funzionalità è stato descritto nella sezione 3.3.4.

4.1.1 Invio periodico della posizione al server

Per ripetere un'azione in maniera periodica ogni x secondi, all'interno del metodo `onCreateView()` del `MatchFoundFragment` viene inizializzato un oggetto `Runnable` il cui scopo è quello di eseguire la funzione per ottenere la posizione del Taker ed inviarla al server tramite una chiamata alle API.

Come è possibile vedere nel codice mostrato in Listing 4.1, il `Runnable` viene inserito all'interno di un `Handler`, e quindi eseguito immediatamente per la prima volta.

Listing 4.1. Inizializzazione del runnable

```
takerPositionUpdateHandler = new Handler();
takerPositionUpdateRunnable = this::startTakerPositionUpdate;
takerPositionUpdateHandler.post(takerPositionUpdateRunnable);
```

L'effettivo codice che viene utilizzato dal `Runnable` per ottenere la posizione ed inviarla al server è incapsulato all'interno del metodo `startTakerPositionUpdate()` mostrato in Listing 4.2.

Listing 4.2. Metodo per l'invio della posizione del Taker al server

```
private void startTakerPositionUpdate() {
    FusedLocationProviderClient fusedLocationProviderClient =
        LocationServices.getFusedLocationProviderClient(app);
    if (ActivityCompat.checkSelfPermission(app.getApplicationContext(),
        Manifest.permission.ACCESS_FINE_LOCATION) !=
        PackageManager.PERMISSION_GRANTED
```

```
            && ActivityCompat.checkSelfPermission(app.getApplicationContext
                (), Manifest.permission.ACCESS_COARSE_LOCATION) !=
                PackageManager.PERMISSION_GRANTED) {
        return;
    }
    fusedLocationProviderClient.getLastLocation().addOnCompleteListener(
        task -> {
        if (task.getResult() != null) {
            MatchUtils.updateTakerPosition(app, new LatLng(task.getResult()
                .getLatitude(), task.getResult().getLongitude()));
        }
        takerPositionUpdateHandler.postDelayed(takerPositionUpdateRunnable,
            LONG_TIMEOUT);
    });
}
```

Alla fine della chiamata API effettuata dal metodo, il `Runnable` viene aggiunto nuovamente all'`Handler` tramite il metodo `Handler.postDelayed`, che eseguirà il `Runnable` nuovamente dopo aver atteso un tempo specificato in `LONG_TIMEOUT`, che in questo caso è di `5000ms`.

Listing 4.3. Body della chiamata alle API per l'aggiornamento della posizione del Taker

```
{
  "lat": 41.89891375106414,
  "lon": 12.3247983291,
  "schedule": "2020-05-14T12:12:58+02:00",
  "eta": 65
}
```

Nel Listing 4.3 viene mostrato il `body` della chiamata alle API che il Taker effettua per aggiornare la posizione al server. Il body è un oggetto json contenente le seguenti informazioni:

- `lat`: la latitudine del Taker;
- `lon`: la longitudine del Taker;
- `schedule`: un `timestamp` formattato come stringa, il quale fa riferimento al campo omonimo all'interno dell'oggetto Match documentato in sezione 2.5.2, che indica quando il Taker intende occupare un posto;
- `eta`: il tempo previsto, indicato in secondi, prima dell'incontro tra il Taker ed il Giver.

4.1.2 Ricezione notifiche push

Nella sottosezione 4.1.1 si è discusso il modo in cui il client del Taker invia la posizione aggiornata al server. In questa sezione si parlerà invece del modo in cui il Giver ottiene la nuova posizione dal server e procede con l'aggiornamento sull'interfaccia.

Come già accennato, l'informazione relativa alla posizione del Taker viene inviata al client del Giver tramite una notifica push, tramite il servizio *Firebase Cloud Messaging* (*FCM*).

Per configurare il client alla ricezione di notifiche da parte di questo servizio, viene creata una classe `NotificationService` che estende `FirebaseMessagingService`.

Come mostrato in Listing 4.4, il `Service` viene aggiunto al file `AndroidMainfest.xml` per rendere consapevole l'applicazione della sua esistenza, e quindi per permetterne il corretto funzionamento.

Listing 4.4. Aggiunta del `Service` all'interno del file `AndroidMainfest.xml`

```
<service
    android:name=".services.NotificationService"
    android:exported="false">
    <intent-filter>
        <action android:name="com.google.firebase.MESSAGING_EVENT"/>
    </intent-filter>
</service>
```

Il primo passo per permettere la ricezione di notifiche push da parte del client è quello di inviare al server il Token di registrazione generato dal `FCM SDK` all'avvio dell'applicazione, e relativo all'istanza dell'applicazione stessa. La conoscenza da parte del server di questo Token è di fondamentale per rendere possibile la comunicazione uno-ad-uno.

Listing 4.5. Invio del Token al server

```
@Override
public void onNewToken(@NonNull String s) {
    super.onNewToken(s);
    //API call to update the push notification token
    app.me(false, me -> {
        app.api().updatePushToken(s).enqueue(new Callback<String>() {
            @Override
            public void onResponse(Call<String> call, Response<String>
                response) {
                Log.d(TAG, "onResponse: " + response.body());
            }

            @Override
            public void onFailure(Call<String> call, Throwable t) {
                Log.e(TAG, "onFailure: " + t.getMessage());
            }
        });
    });
}
```

Come mostrato in Listing 4.5, viene effettuato un *override* del metodo `onNewToken`, il quale viene chiamato dal client ogni volta che il `FCM SDK` ne genera uno nuovo.

Il Token viene inviato al server all'interno del `body` della chiamata alle API con `endpoint "/me/push/tokens/"` e metodo `HTTP POST`.

Viene poi effettuato un *override* del metodo `onMessageReceived` che viene chiamato ogni volta che il client riceve una notifica push dal server.

Listing 4.6. Override del metodo per gestire la ricezione di notifiche push

```
@Override
public void onMessageReceived(@NonNull RemoteMessage remoteMessage) {
```

```
    super.onMessageReceived(remoteMessage);
    Map<String, String> data = remoteMessage.getData();
    if (data.get("lat") != null && data.get("lon") != null) {
        LatLng takerPosition = new LatLng(Double.parseDouble(data.get("lat")
            ), Double.parseDouble(data.get("lon")));
        Intent intent = new Intent("TakerPosition");
        intent.putExtra("latLng", takerPosition);
        broadcaster.sendBroadcast(intent);
    }
}
```

All'interno di questo metodo viene creato un oggetto `LatLng` contenente le coordinate del Taker estratte dai campi `lat` e `lon` del messaggio della notifica.

L'oggetto contenente le coordinate viene inserito all'interno di un `Intent` che viene inviato al `LoacalBroadcastManager`.

Listing 4.7. Gestione dell'intent

```
private final BroadcastReceiver notificationReceiver = new
    BroadcastReceiver() {
    @Override
    public void onReceive(Context context, Intent intent) {
        Log.d("NotificationDebug", "NotificationReceiver data: " +
            Objects.requireNonNull(intent.getExtras()).getParcelable("latLng
            "));
        LatLng takerPosition = Objects.requireNonNull(intent.getExtras())
            .getParcelable("latLng");
        if (takerPositionMarker != null) {
            MarkerAnimationUtils.animateMarkerToGB(takerPositionMarker,
                takerPosition, new LatLngInterpolator.Linear());
        }
    }
};
```

In Listing 4.7 viene mostrato come viene gestita la ricezione dell'`Intent` inviato in Listing 4.6.

In particolare, viene estratto l'oggetto `LatLng` contenente le coordinate, e viene spostato il marker in tali coordinate mediante un'animazione fluida dalla durata di `5000ms`. [1]

La durata dell'animazione è stata impostata a `5000ms` in quanto questo è il tempo che passa tra una notifica push e l'altra. In questo modo il cambio di posizione ottenuto durante le varie notifiche push ottenuto sarà il più fluido possibile.

[1] Animazione permessa dalla libreria raggiungibile al seguente link.

Capitolo 5

Conclusioni

5.1 Sommario

Nel Capitolo 1 sono stati analizzati gli obiettivi che l'applicazione su cui si è svolto il lavoro vuole risolvere, ed in che modo promette di farlo.

Nel Capitolo 2 sono state mostrate le tecnologie utilizzate all'interno del tirocinio, insieme ai concetti fondamentali per lo sviluppo di applicazioni Android.

Nel Capitolo 3 è stata discussa la fase di progettazione della UI (User Interface) e della UX (User Experience). La prima è volta a comprendere come organizzare gli elementi mostrati sullo schermo in maniera da rendere l'applicazione facile da utilizzare; la seconda è volta a capire in che modo strutturare l'intero sistema per rendere l'esperienza che l'utente ha con quest'ultimo il più piacevole possibile.

Sono stati quindi mostrati i prototipi delle interfacce che sono state sviluppate, descrivendo le relative funzionalità, e sono stati documentati i test di usabilità effettuati, elencando i problemi individuati e le soluzioni che sono state proposte, fino all'arrivo di un'interfaccia completa.

Nel Capitolo 4 sono stati approfonditi i dettagli implementativi circa le interfacce e funzionalità progettate nel Capitolo 3, mostrando le parti di codice significative che hanno portato alla realizzazione delle funzionalità desiderate.

Il tirocinante, alla fine del lavoro, ha maturato le seguenti capacità:

- Progettazione di un prototipo di interfaccia;
- Implementazione di un'interfaccia all'interno di Android a partire da un prototipo;
- Sviluppo di funzionalità in ambito Android in linguaggio Java;
- Effettuare chiamate API e gestire le risposte.

5.2 Sviluppi Futuri

Dall'inizio del tirocinio alla fine di quest'ultimo all'interno dell'applicazione sono state inserite numerose funzionalità, tuttavia l'applicazione è ancora nelle prime fasi di sviluppo e quindi sono molteplici quelle che ancora possono essere implementate.

Le funzionalità principali a mancare per quanto riguarda la gestione dei match sono:

- La possibilità di un utente di cercare un posto in una zona differente da quella in cui si trova attualmente;
- La possibilità di un utente di cercare un posto in un orario differente da quello attuale, in modo da prenotarsi il posto per un momento futuro della giornata;
- La possibilità di un utente di essere Giver e Taker contemporaneamente.

A parte le prime due funzionalità che sono abbastanza autodescrittive, in questa sezione si descriveranno i dettagli e le complicazioni riguardanti la possibile implementazione della terza funzionalità.

Si ipotizzi che le prime due funzionalità elencate sopra già esistano all'interno dell'applicazione, e si ipotizzi la seguente situazione:

1. Sia A un utente attualmente parcheggiato;
2. A effettua un "Cerca posto" in un'altra zona;

L'utente diventa quindi un Taker e raggiunge il Giver B associato per occupare il suo posto. Essendo l'utente A però parcheggiato, ed essendo egli un Taker, non può effettuare un "Lascia posto" prima di raggiungere l'utente B.

Questo vuol dire che il posto da lui lasciato non verrà immediatamente occupato da un eventuale Giver C.

È possibile risolvere questo problema dissociando le azioni di "Cerca posto" e "Lascia posto", in modo che queste siano eseguibili indipendentemente l'una dall'altra.

Un utente A parcheggiato che intende effettuare un "Cerca posto", potrebbe quindi effettuare prima un "Lascia posto" per trovare un Taker B compatibile che occupi il suo attuale posto. Effettuerà il "Cerca posto" nell'attesa che il Taker arrivi nella sua posizione, in modo da trovarsi già un utente Giver C compatibile. Una volta che il Taker B sia arrivato nella posizione dell'utente A e questi si siano scambiati i posti, l'utente A raggiungerà l'utente C per scambiare il posto auto con lui.

Bibliografia

[1] Sistema statistico nazionale Istituto nazionale di statistica https://www.istat.it/storage/ASI/AnnuarioStatistico_2020/Asi_2020.pdf

[2] Progetto SPARTA: ridurre l'inquinamento grazie allo smart parking e all'analisi del traffico. https://www.greenvulcano.com/it/progetto-sparta-ridurre-linquinamento-grazie-allo-smart-parking-e-allanalisi-del-traffico/

[3] Smart parking: tutto ciò che bisogna sapere sul parcheggio intelligente https://www.backtowork24.com/news/smart-parking-tutto-cio-che-bisogna-sapere-sul-parcheggio-intelligente

[4] GeneroCity https://www.generocity.it/

[5] A Step by Step Guide to UI/UX Design Process https://aufaitux.com/blog/ui-ux-design-process/

[6] The UX Design Process: Everything You Need to Know https://xd.adobe.com/ideas/guides/ux-design-process-steps/

[7] Test di usabilità https://designers.italia.it/kit/test-usabilita/

[8] Nielsen e le sue 10 euristiche https://www.far.unito.it/usabilita/cap5.htm

[9] Activity https://developer.android.com/reference/android/app/Activity

[10] Activity Lifecycle https://developer.android.com/guide/components/activities/activity-lifecycle

[11] Fragments https://developer.android.com/guide/fragments

[12] Fragment Lifecycle https://developer.android.com/guide/fragments/lifecycle

[13] Intents https://developer.android.com/guide/components/intents-filters

[14] Services https://developer.android.com/guide/components/services

[15] What is RESTful API https://devdotcode.com/what-is-restful-api/

[16] REST API: Key Concepts, Best Practices, and Benefits https://www.altexsoft.com/blog/rest-api-design/

[17] Come creare un wireframe interattivo https://alfredoiannone.com/2018/05/02/come-creare-un-wireframe-interattivo/

[18] Il modello di un'interfaccia e il Wireframe Kit https://design-docs-francescozaia.readthedocs.io/en/revisione-capitolo-ui/doc/user-interface/il-modello-di-un-interfaccia-e-il-wireframe-kit.html

[19] Qual è la differenza tra Wireframe, Prototipo e Mockup? http://www.fabiomarasco.it/la-differenza-tra-wireframe-prototipo-e-mockup/

[20] Android Fragments: sperimentiamo il ciclo di vita https://www.devapp.it/wordpress/android-fragments-sperimentiamo-il-ciclo-di-vita/

[21] Using Retrofit 2.x as REST client https://www.vogella.com/tutorials/Retrofit/article.html