Alessandro Ravveduto alessandro.ravveduto@studio.unibo.it 0001020539
Alessandro Tomaiuolo alessandro.tomaiuol2@studio.unibo.it 0001030690

Physical Tracker App

Progetto LAM 2023-2024

Introduzione

Physical Tracker è un'applicazione sviluppata per il progetto del corso "Laboratorio di applicazioni mobili" A.A. 2023/2024 dell'università di Bologna.

Il progetto è focalizzato sulla realizzazione di un'applicazione Android che monitora le attività quotidiane dell'utente sfruttando i sensori integrati nel dispositivo mobile. Le attività registrabili includono: Camminata, Corsa, Riposo e Guida. Tali attività sono visualizzabili tramite un calendario interattivo e facile da utilizzare, il quale permette anche la visualizzazione delle attività degli altri utenti di cui si è importato il calendario. Una tra le principali feature è quella del Geofencing, ovvero un servizio basato sulla localizzazione che esegue un'azione pre-programmata rispetto a una barriera virtuale geografica. L'interfaccia grafica è ideata principalmente per un utilizzo in modalità Portrait; tuttavia, ogni schermata è ottimizzata anche per la visualizzazione Landscape. La navigazione è resa semplice ed intuitiva grazie ad una barra di navigazione posizionata in basso in modalità verticale e sul lato sinistro in modalità orizzontale. Infine, l'applicazione si adatta alla lingua del sistema, supportando sia quella Italiana sia Inglese.

Flusso di navigazione

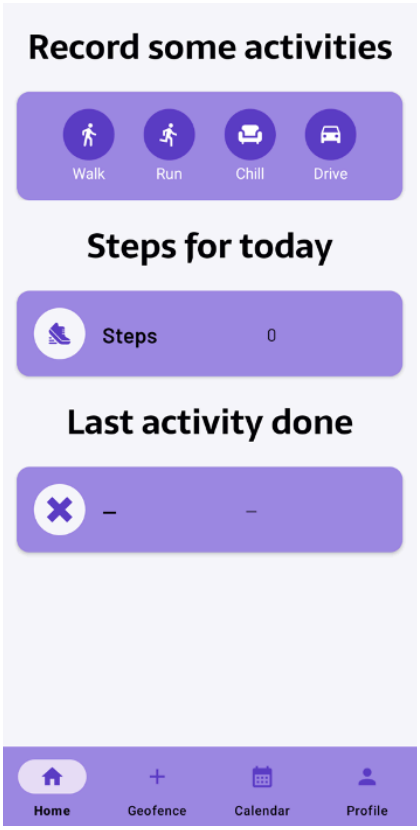
Front page

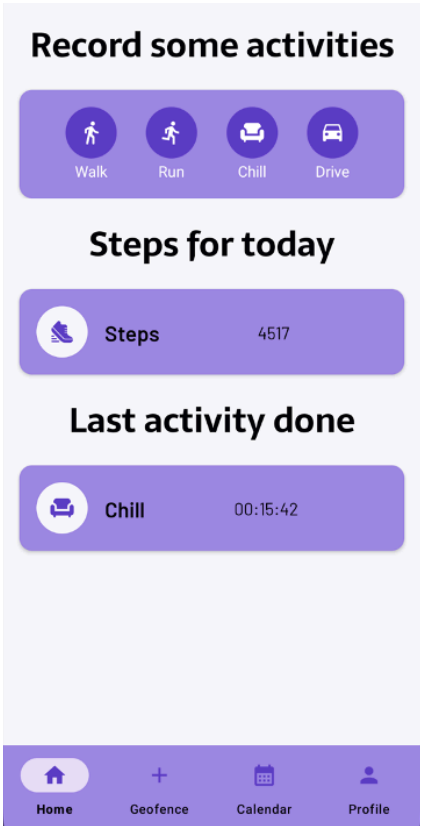
Quando apriremo l'applicazione per la prima volta verremo catapultati in una schermata in cui verrà richiesto all'utente di inserire il proprio nome. Questo identificativo verrà poi visualizzato nella sezione profilo e utilizzato per segnare le attività svolte.

Home

Dopo aver impostato il nome, l'utente verrà reindirizzato alla schermata principale, la Home. In questa sezione sono presenti diversi elementi grafici interattivi e altri componenti destinati alla visualizzazione di statistiche rilevanti. Nella parte superiore della schermata è presente una barra contenente quattro pulsanti, ciascuno associato ad un'attività specifica, che indirizzano l'utente ad una pagina dedicata per la registrazione dell'attività selezionata.

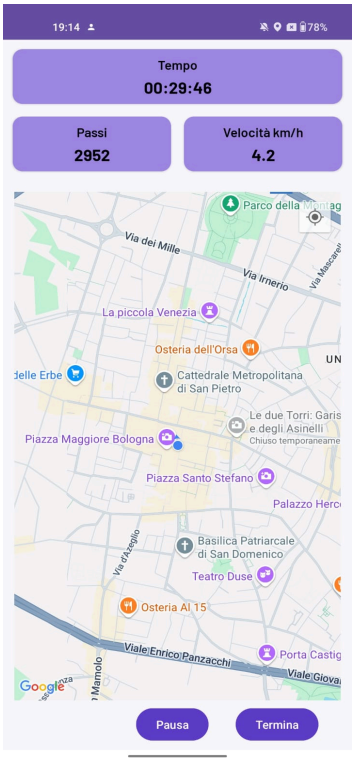
Al centro della schermata, si trovano due indicatori: la prima barra visualizza il numero dei passi effettuati nel giorno corrente, calcolato come somma dei passi registrati in tutte le attività della giornata; la seconda barra, situata immediatamente sotto, mostra l'ultima attività completata. Se l'utente ha appena installato l'app e non ha ancora registrato alcuna attività, viene visualizzata un'icona segnaposto in assenza di ulteriori informazioni, come illustrato in figura.

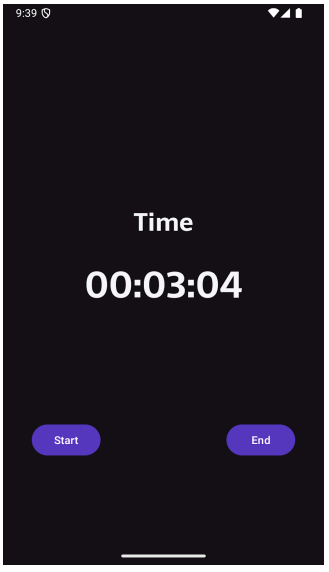


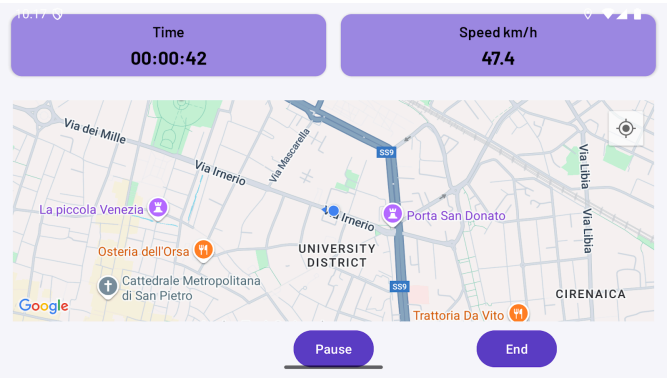


In fondo allo schermo troviamo la barra di navigazione che permette di muoversi tra le pagine dell'applicazione, di default sarà selezionata la Home.

Ritornando alla barra in alto, le quattro attività disponibili saranno Camminata, Corsa, Riposo e Guida. Ogni schermata, tranne le prime due, sono diverse tra loro e in ognuna si trovano le informazioni rilevanti per quell'attività, tra cui passi, velocità e ovviamente il tempo. Inoltre, sono sempre presenti i bottoni che permettono di avviare, mettere in pausa e terminare l'attività, che verrà poi salvata all'interno del database.





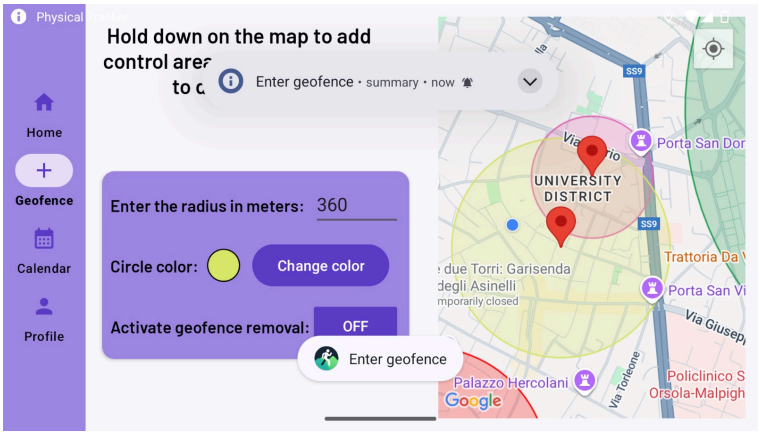


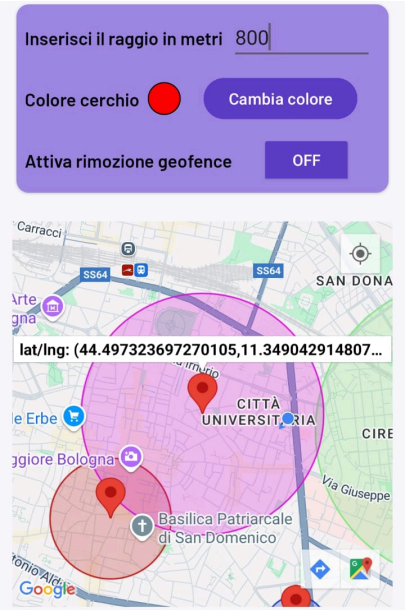
Geofence

La seconda schermata disponibile è dedicata alla funzionalità Geofence. Nella parte superiore è presente un'intestazione con istruzioni per l'utilizzo della mappa. L'utente può aggiungere un marker con un cerchio di raggio e colore selezionato tenendo premuto su un qualunque punto della mappa.

Disattivando l'opzione di rimozione dei geofence, è possibile cliccare su un marker per centrare la visualizzazione della mappa su di esso. Questo mostrerà una legenda con le coordinate esatte e in basso a destra due tasti che permettono l'apertura di Google Maps per raggiungere il luogo selezionato. Attivando invece la possibilità di rimuovere geofence sarà possibile eliminare le aree di controllo con un semplice tocco.

Per garantire la permanenza delle aree di geofence, queste verranno salvate nel database locale. Quando l'utente entra, esce o permane in una di queste aree gli verrà segnalato tramite una notifica e un messaggio toast. Il monitoraggio avviene in background, pertanto questa feature opera anche quando l'applicazione è chiusa.



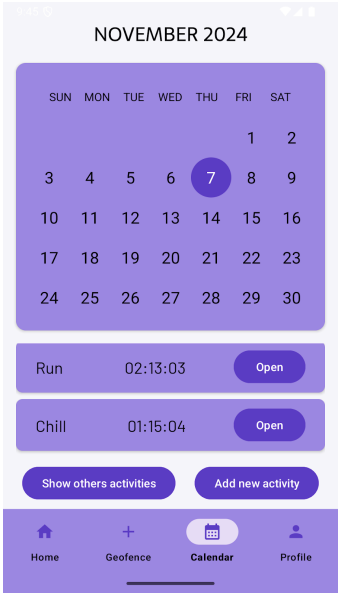


Calendario

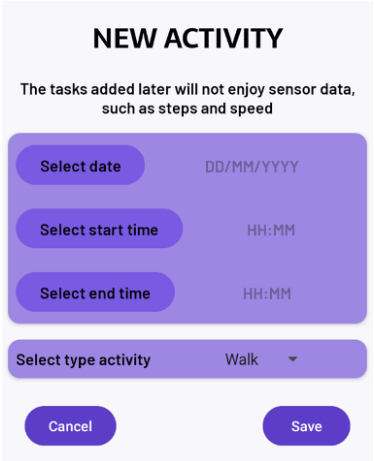
La terza schermata disponibile è il Calendario, che permette all'utente di monitorare sia le proprie attività sia quelle degli altri utenti con cui ha scambiato il calendario. È possibile scorrere tra i mesi con uno swipe laterale e selezionare il giorno desiderato tramite un tap. Se per quella data sono presenti attività, verranno visualizzate delle anteprime sotto al calendario, con informazioni chiave quali il tipo di attività e la durata. Inoltre, ogni anteprima include un pulsante che consente di accedere ad una pagina dedicata per visualizzare tutti i dettagli dell'attività.

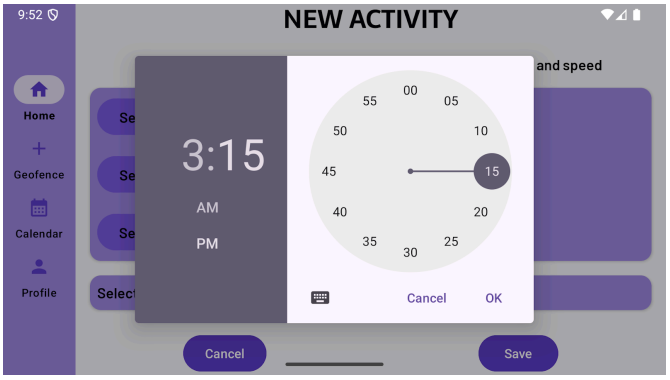
Nella parte inferiore dello schermo (in modalità Portrait) sono presenti due bottoni: a sinistra, il pulsante "Mostra attività altrui" consente, una volta selezionato, di visualizzare nella preview soltanto le attività importate degli altri utenti per il giorno selezionato. Come si può

notare dall'immagine, il testo cambierà automaticamente in "Mostra tue attività" per consentire all'utente di tornare alla visualizzazione delle proprie attività.

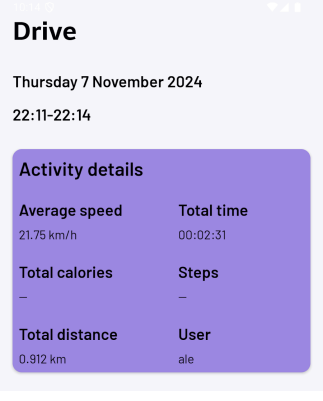


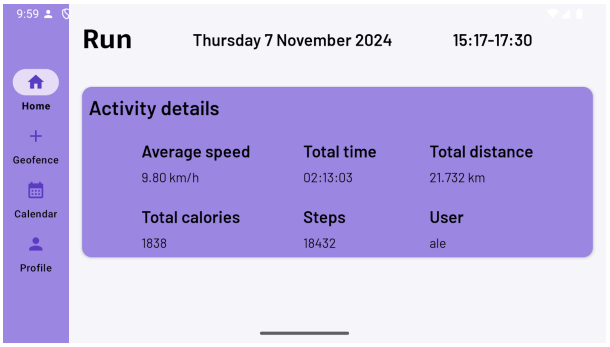
L'altro bottone "Aggiungi Attività" porta ad una schermata dedicata nella quale è possibile inserire un'attività a posteriori, nel caso in cui l'utente si sia dimenticato di registrarla. Si sottolinea che nel caso dell'inserimento di attività con questa modalità, informazioni aggiuntive come passi, velocità e calorie non saranno presenti; naturalmente vengono effettuati dei controlli per assicurarsi la validità degli intervalli temporali inseriti, in caso di errore verrà segnalato tramite dei messaggi toast.





Cliccando sul bottone presente di fianco ad ogni anteprima dell'attività sarà possibile visualizzare più nel dettaglio le informazioni su di essa.

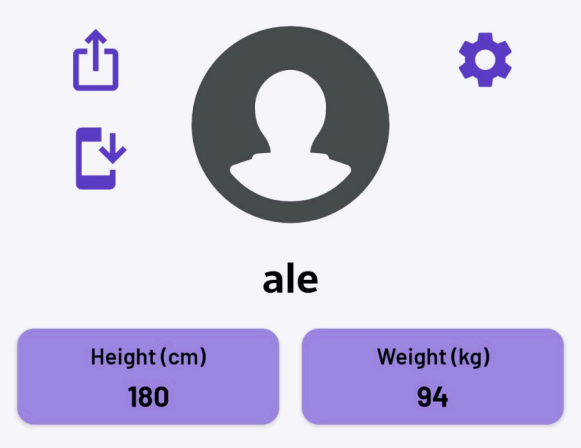


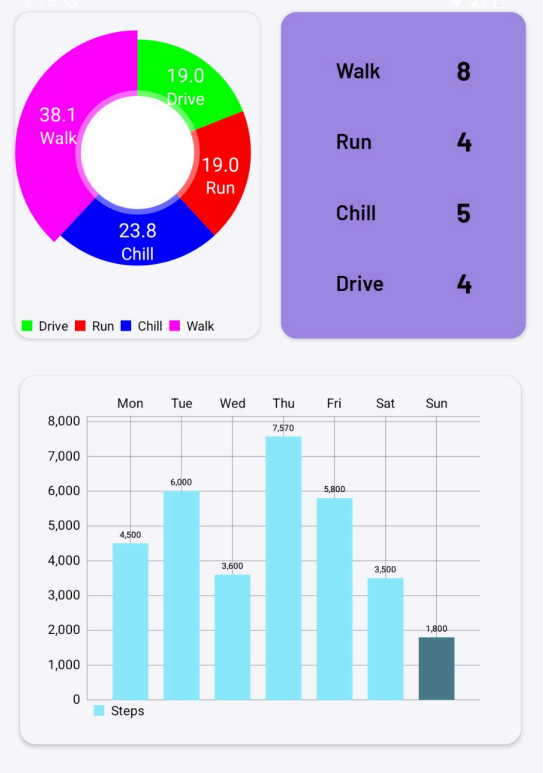


Profilo

L'ultima sezione disponibile è il Profilo, una schermata in cui l'utente può visualizzare in dettaglio statistiche personali, accedere ai comandi per esportare ed importare il calendario con altri utenti, oltre che ad un apposito bottone per raggiungere le impostazioni.

Più nel dettaglio, nella parte superiore della schermata sono indicati altezza e peso dell'utente, quest'ultima fondamentale per il calcolo delle calorie. Proseguendo verso il basso, sono presenti tre card: le prime due forniscono statistiche sul tipo e sulla frequenza delle attività svolte. In particolare, la prima contiene il grafico a torta che illustra le percentuali relative ai diversi tipi di attività, la seconda invece mostra i valori assoluti per ciascun tipo di attività. Al di sotto di queste due si trova l'ultima card al cui interno è presente un grafico a barre in cui sono riportati i passi effettuati quotidianamente nella settimana corrente.





Scelte implementative

Durante lo sviluppo dell'applicazione sono state prese le seguenti scelte implementative:

1. **Android Nativo**: La scelta di sviluppare nativamente su Android ha consentito di ottimizzare le prestazioni rispetto a un approccio multipiattaforma, migliorare l'integrazione con gli aggiornamenti del sistema operativo e avere pieno accesso alle API e ai servizi nativi.

2. **Architettura MVVM**: L'adozione del pattern Model-View-ViewModel (MVVM) ha facilitato la gestione del flusso di dati e migliorato la separazione delle responsabilità, favorendo una maggiore modularità e testabilità del codice.

3. **Google Maps SDK**: L'integrazione di questa SDK ha semplificato l'implementazione delle funzionalità di mappa nell'app, inclusa la gestione automatica degli eventi di geofencing e l'aggiornamento della posizione in tempo reale.

4. **kizitonwose.calendar**: Questa libreria, basata su RecyclerView, ha permesso l'implementazione di un calendario altamente personalizzabile e adattabile alle esigenze dell'applicazione.

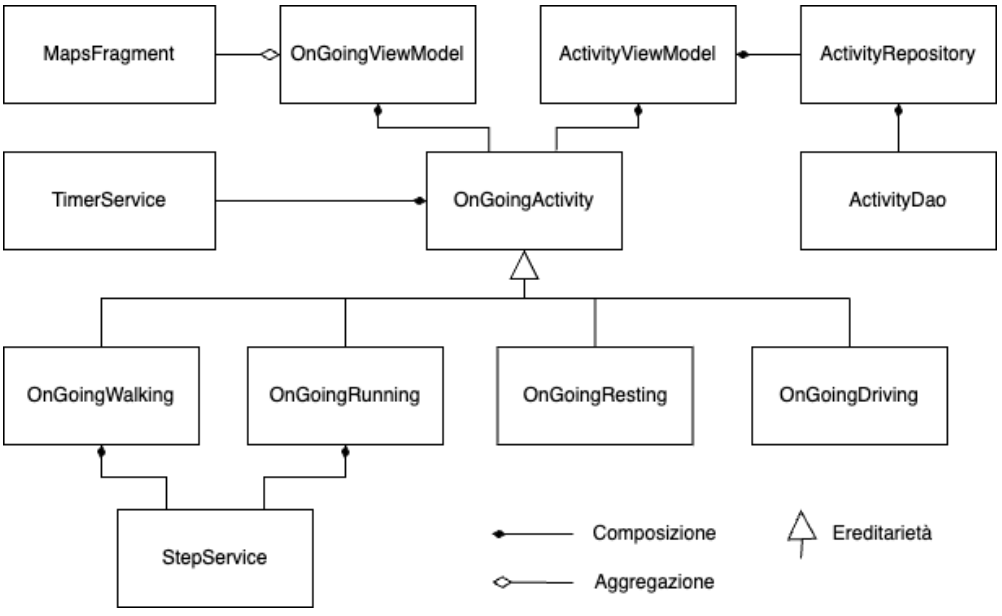
5. **MPAndroidChart**: Famosa libreria per la creazione e gestione dei grafici.

6. **AmbilWarna**: Per una personalizzazione avanzata, AmbilWarna è stata adottata come libreria per consentire agli utenti un'ampia scelta di colori, arricchendo l'esperienza d'uso.

7. **Room:** Una libreria che funge da abstraction layer, permettendo di sfruttare tutte le potenzialità del database MySQLite senza la preoccupazione di affrontare ogni aspetto nei dettagli.

Registrazione delle attività

La figura sottostante indica la struttura di classi e servizi realizzata per l'acquisizione dei dati dei sensori relativi alle attività e per la loro registrazione.



Segue una descrizione accurata delle funzionalità di ogni classe

Classe HomeFragment

La gestione della registrazione dell'attività inizia dalla classe HomeFragment.kt, la funzione responsabile dei Button che registrano attività è activityButtonListener, che viene associata come onClickListener ai bottoni medesimi.

La funzione crea un intent verso OnGoingPlaceHolder.kt, e a seconda dell'id del bottone cliccato, invia una stringa diversa come extra.

Classe OnGoingPlaceHolder

Questa classe, come dice il nome, funziona da placeholder. Il suo scopo è quello di indirizzare l'utente verso la classe che rappresenta l'attività da esso scelta

```
  val type = receivingIntent
  .getStringExtra(ActivityString.ACTIVITY_TYPE)

        when (type) {
            ActivityType.WALKING.toString() ->
{               startActivity(
                  Intent(this,
                  OnGoingWalking::class.java))
            }

            ActivityType.RUNNING.toString() ->
{               startActivity(
                  Intent(this,
                  OnGoingRunning::class.java))
            }

            ....
        }
}
```

Classi OnGoingActivity, OnGoingWalking, OnGoingRunning, OnGoingResting, OnGoingDriving

Come si evince dal grafico precedente, per evitare ridondanza nel codice e sfruttare l'ereditarietà delle classi di Kotlin, abbiamo deciso di creare OnGoingActivity.kt, che farà da classe padre per le classi che rappresentano le delle specifiche attività.

Nella OnGoingActivity.kt è dunque presente la logica delle feature che sono sfruttate da tutte le attività, come la comunicazione con TimerService.kt, classe che si occupa di gestire il service del timer. Inoltre, vengono inizializzati dei listeners su elementi che sono comuni a tutti i layout delle varie attività, come i bottoni di inizio e pausa. Infine, nella classe padre vengono istanziati e inizializzati due oggetti di tipo OnGoingViewModel e ActivityViewModel, permettendone l'utilizzo delle classi figlie.

Ogni attività ha la sua classe specifica e può presentare feature aggiuntive, in particolare:

- OnGoingWalking e OnGoingRunning
  Queste classi aggiungono la comunicazione con StepCounter, classe contenente l'accesso al sensore dei passi del telefono, fondamentale per registrarli durante le

attività. Tramite l'oggetto di tipo ActivityViewModel ereditato, questa classe comunica con MapsFragment, e ogni volta che viene aggiornata la posizione restituisce la velocità, che verrà memorizzata in un array, utilizzato al termine dell'attività per calcolare la velocità media.

- OnGoingDriving
  Questa classe è molto simile alle classi precedenti ma più snella siccome non necessita del sensore dei passi.

- OnGoingResting
  Questa classe rappresenta l'attività più basilare, contenente soltanto il timer e la possibilità di metterlo in pausa e riavviarlo.

Classe MapsFragment

In ogni attività o fragment in cui sia presente una mappa opera la classe MapsFragment. Essa è responsabile della comunicazione con Google Maps SDK e racchiude la logica necessaria per ricevere gli aggiornamenti della posizione, questi verranno utilizzati per animare la mappa e stimare la velocità dell'utente in movimento.

Classe ActivityViewModel

Data la necessità di comunicare con il database per quanto riguarda la registrazione delle attività, è stata ideata la classe ActivityViewModel, la quale contiene i metodi per comunicare con una repository associata al database.

Classi ActivityDatabase, SpecialActivities e ActivityDAO

Il database delle attività contiene cinque tabelle dove la prima, base_activity_table, contiene le entità di tipo BaseActivity, classe in cui trovano posto i principali attributi di ogni attività.

Nella classe SpecialActivities troviamo quattro entità che rappresentano ognuna le rispettive attività specifiche e a cui è associata una tabella nel database. Le entità specifiche sono associate alla propria BaseActivity tramite una *foreignKey* che collega l'*id* della BaseActivity all'*activityId* della relativa classe.

Siccome la libreria Room non è in grado di gestire in modo autonomo tipi complessi come LocalTime e LocalDate, è stato creata la classe TimeConverter con lo scopo di convertire tali tipi in stringhe, rendendo più facile il salvataggio di questi dati nel db; verranno poi recuperati a run-time tramite il parsing da stringe a LocalTime e LocalDate.

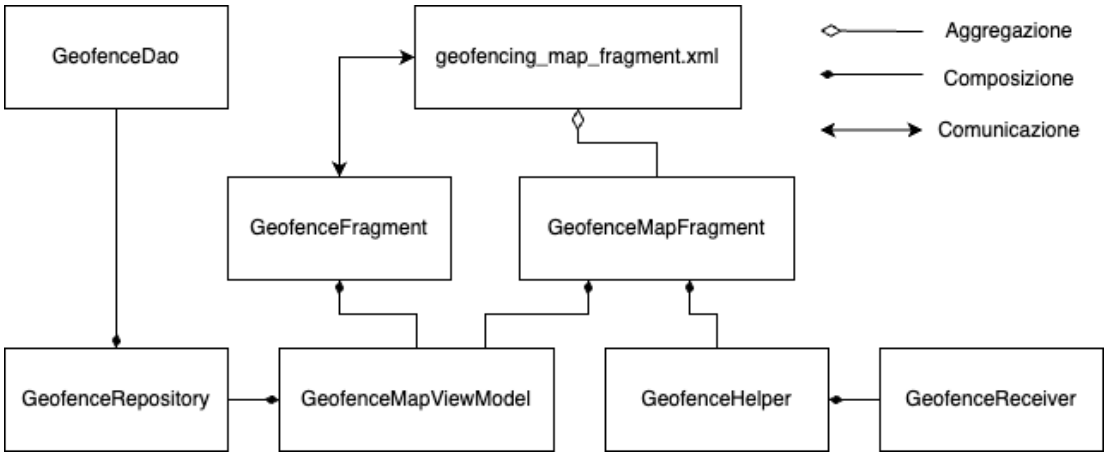
```kotlin
class TimeConverter {
    @TypeConverter
    fun fromDate(value: String?): LocalDate? {
        return value?.let { LocalDate.parse(it)
}   }

    @TypeConverter
    fun dateToString(date: LocalDate?): String?
{       return date?.toString()
    }
    //analogo con LocalTime
}
```

Per ottenere le informazioni dell'attività in una sola classe, è stata creata la classe ActivityJoin, una nested class che contiene una baseActivity e tutti i tipi di attività speciale, dove solo un tipo sarà diverso da null. Le funzioni nell'interfaccia DAO che ritornano questa classe, sono annotate con @Transaction.

L'ActiviyRepository comunica con l'interfaccia ActivityDao, la quale racchiude le operazioni di accesso ai dati (DAO) per le classi BaseActivity e le classi presenti in SpecialActivities.kt.

Geofencing

L'immagine seguente mostra la struttura utilizzata per la gestione dei geofence.



Classe GeofenceFragment

La classe GeofenceFragment è piuttosto semplice ed è responsabile per l'interazione tra l'utente e alcuni componenti della pagina in questione, come i bottoni per modificare il raggio e il colore dell'area e l'attivazione del selettore di eliminazione dei Geofence.

Classe GeofenceMapViewModel

La classe GeofenceMapViewModel è fondamentale sia per GeofenceFragment che per GeofenceMapFragment in quanto agisce da ponte tra le due e contiene i metodi necessari per la gestione del salvataggio ed eliminazione delle aree di geofence dal database. Per assolvere questi compiti il viewModel in questione fa uso della classe GeofenceRepository e a sua volta della GeofenceDAO.

Classe GeofenceMapFragment

GeofenceMapFragment racchiude la core-logic dietro la creazione, eliminazione e monitoraggio delle aree di geofencing. In particolare, per aggiungere e rimuovere le aree di controllo vengono adoperati i metodi mostrati nelle seguenti figure. Si può notare come vengano utilizzati oggetti quali il GeofenceClient fornito dal SDK di Google ed existingGeofence. Quest'ultimo è di tipo MutableMap: come chiave usa una stringa, ottenuta utilizzando la data class GeofenceKey, in tal modo ci si assicura che ogni Geofence sia unico e che non ce ne siano due identici; come valore contiene la coppia Circle e Marker, oggetti ottenuti a run-time necessari per la visualizzazione ed eliminazione delle aree. L'oggetto existingGeofence permette di sincronizzare quello che avviene nella mappa a seguito delle operazioni dell'utente con ciò che viene memorizzato nel database.

```
private fun populateMap(geofenceListFromDb: List<GeofenceInfo>) {
        var position :LatLng
        if (geofenceListFromDb != null) {
            for (geofence in geofenceListFromDb) {
                if (!existingGeofence.containsKey(geofence.id)) {
                    position = LatLng(
                      geofence.latitude,
                      geofence.longitude)

                    displayGeofence(
                      position,
                      geofence.color,
                      geofence.radius,
                      geofence.id)
                }
            }
        }
    }
```

```
private fun deleteGeofence(geofenceId2remove: String, marker: Marker) {
        // Removing the geofence from the client
        geofencingClient.removeGeofences(listOf(geofenceId2remove))
            .addOnSuccessListener {
                // Removing the circle and marker from the map
                val circle2remove = getCircleByGeofenceId(geofenceId2remove)
                marker.remove()
                circle2remove?.remove()

                // Removing the geofence from the view model
                geofenceMapViewModel.removeGeofenceInfo(geofenceId2remove)

                // Removing the geofence from the existing geofence list
                existingGeofence.remove(geofenceId2remove)

                // Notify the user the successful removal
                Toast.makeText(...).show();
            }
            .addOnFailureListener {
                // Notify the user the successful removal
                Toast.makeText(...).show();
            }

    }
  return go(f, seed, [])
}
```

Classi NotificationHelper, GeofenceHelper e GeofenceReceiver

Ai fini di una migliore leggibilità e manutenibilità del codice abbiamo deciso di utilizzare queste tre classi per delegare alcune operazioni come la creazione delle GeofenceRequest e tramite la funzione getPendingIntent si associa ad ogni geofence un pendingintent, questo verrà attivato dal sistema qualora si verifichi un evento legato al geofence (come entrata o uscita dall'area). Inoltre, grazie all'utilizzo dei pendingintent si garantisce che tali notifiche vengano inviate anche quando l'applicazione non è eseguita in primo piano. Per quanto riguarda la classe GeofenceReceiver, questa estende la classe nativa

BroadcastReceiver e permette di ricevere e smistare gli intent in base al geofenceTransition di ogni geofenceEvent, dopo di che viene inviata la relativa notifica utiilzzando un'istanza della classe NotificationHelper.

Interfaccia Grafica

Per la schermata principale dell'applicazione, è stata adottata una singola Activity che utilizza diversi fragment per gestire le varie sezioni. Questo approccio ha permesso di mantenere il contenuto sempre aggiornato senza la necessità di ricreare l'intera Activity, migliorando l'esperienza utente in termini di fluidità e continuità.
Grazie all'utilizzo di LiveData e ViewModel, la gestione dei cambi di configurazione, come la rotazione dello schermo, risulta stabile.
La funzione che gestisce il cambio dei fragment all'interno della MainActivity è la changeFragment, questa rende possibile passare da un fragment all'altro tramite la navbar senza che quest'ultimo venga ogni volta ricreato, ripescandolo così dal BackStack.

```
private fun changeFragment(fragment: Fragment, tag: String) {
        val currentFragment = supportFragmentManager
        .findFragmentById(R.id.fragmentContainerView)

        if (currentFragment != null &&
            currentFragment.tag == tag) {
            return
        }
        val storedFragment = supportFragmentManager
                            .findFragmentByTag(tag)

        if ( storedFragment != null ) {
            supportFragmentManager.beginTransaction().run {
                if (currentFragment != null) {
                    detach(currentFragment)
                }
                attach(storedFragment)
                commit()
            }
        } else { supportFragmentManager.beginTransaction().run
{               setReorderingAllowed(true)
                if (currentFragment != null) {
                    detach(currentFragment)
                }
                add(R.id.fragmentContainerView,fragment,tag)
                addToBackStack(null)
                commit()
            }
        }
    }
```

Per la gestione delle pagine riguardanti le attività in corso, la schermata di resoconto delle attività e la frontpage, sono state create delle Activity per ognuna, siccome sono logicamente differenti dalla MainActivity.

Notifiche

L’applicazione integra un sistema di notifiche per garantire all'utente un monitoraggio costante delle proprie attività. Un WorkManager esegue un controllo ogni 15 minuti, verificando se l'utente non ha svolto attività recenti e, in caso di inattività superiore ad una certa soglia, invia una notifica per incoraggiare l’utente a svolgere e registrare nuove attività. Inoltre, se un’attività è in corso e l’app è in background, una notifica specifica ricorda all’utente che l’attività è ancora attiva. Inoltre, sono presenti notifiche anche per il geofencing, che informano l'utente sugli eventi di entrata, uscita o permanenza in un'area predefinita, come discusso nelle sezioni precedenti. Infine, una notifica avvisa l'utente che la posizione è attualmente tracciata per abilitare il monitoraggio geofence.