SmartTracker: aggiornamenti periodici sulla posizione in background attivabili da SMS

Andrea Latanza, matricola 290219
Guglielmo Sisti, matricola 289073
Alexey Zababurin, matricola 290534

Anno Accademico 2017-2018

Indice

1 Introduzione

1.1 Abstract

Rivolta principalmente a genitori, coppie o amici che necessitano di conoscere la posizione dei propri cari, l'applicazione permette di richiedere aggiornamenti sulla posizione del dispositivo sulla quale è installata, anche quando è in background, e da remoto tramite SMS. L'app è tuttavia utilizzabile anche tramite la semplice ed intuitiva interfaccia grafica, dalla quale è possibile modificare alcune impostazioni, per personalizzare l'esperienza e visualizzare una mappa per consultare lo storico delle posizioni, disponibili all'indirizzo `http://smarttracker.ddns.net`.

1.2 Obiettivi

Obiettivo principale del progetto è fornire agli utenti un'app semplice da utilizzare che consenta di ottenere aggiornamenti periodici sulla posizione del dispositivo sulla quale è installata, anche quando la stessa è in background. È possibile richiedere o rimuovere gli aggiornamenti sulla posizione nei seguenti modi:

- Tramite interfaccia grafica, mediante uno `Switch`;
- Tramite SMS, qualora venisse superato il processo di validazione.

2 Documento di analisi

2.1 Ottenimento aggiornamenti posizione in background

Al fine di ricevere aggiornamenti periodici sulla posizione, si è scelto di ricorrere all'API `FusedLocationProviderClient`, seguendo le raccomandazioni di Google. Quest'ultima è il punto d'ingresso per interagire con tutti i servizi di posizione (GPS, reti mobili e Wi-Fi) e, rispetto alla libreria inclusa in Android, `LocationServices`, consente di ridurre il codice boilerplate e diminuire le possibili insidie, oltre che scegliere automaticamente il servizio di posizione migliore e ottimizzare l'utilizzo della batteria. L'API è disponibile ai dispositivi con il Play Store installato che ammontano ad oltre il 93% [1]del totale.
Data la necessità di ottenere aggiornamenti anche in background ed a distanza ravvicinata (5 secondi), si è scelto di ospitare il
`FusedLocationProviderClient` in una classe che estende `Service`,
`LocationUpdatesService`.
Attraverso una classe appositamente realizzata, che funge da mediatrice tra l'interazione tramite interfaccia grafica e quella remota, il Singleton `ServiceManager`, è possibile ricevere e rimuovere gli aggiornamenti posizione tramite i due metodi *requestLocationUpdates()* e *removeLocationUpdates()*.
Per poter funzionare effettivamente quando l'app non è visibile e rispettare i limiti di esecuzione in background introdotti in Android Oreo, qualora la singola `Activity` dell'app andasse in background, per continuare a ricevere aggiornamenti, il servizio verrà promosso ad un Foreground Service. L'utente verrà informato dell'attività in background tramite una notifica persistente, appositamente progettata per riflettere lo stato degli aggiornamenti posizione.

[1]Aggiornato al 2014, fonte Sundar Pichai, CEO Google: https://youtu.be/wtLJPvx7-ys?t=4410

2.2 Controllo remoto tramite SMS

Alla ricezione di un SMS, il framework di Android invia un broadcast di sistema contenente un `Intent` che deve essere ricevuto utilizzando un `BroadcastReceiver`. A questo scopo si è estesa la classe `BroadcastReceiver` creando `SmsReceiver`.
In **SmartTracker** è possibile impostare alcune preferenze, come ad esempio la possibilità di attivare i comandi a distanza solo da un numero di telefono specifico o da tutti. Per questo all'interno di `SmsReceiver` sono presenti alcuni metodi per verificare se il mittente dell'SMS è autorizzato, e se l'SMS stesso corrisponde al messaggio di attivazione o disattivazione degli aggiornamenti posizione, anch'essi personalizzabili.
Se l'SMS corrisponde al messaggio d'attivazione per richiedere o rimuovere gli aggiornamenti posizione, si otterrà l'istanza del singleton `ServiceManager` tramite il metodo statico *getInstance()* e, rispettivamente, verrà invocato il metodo *requestLocationUpdates()* o *removeLocationUpdates()*. La classe `ServiceManager` è l'unico punto d'ingresso per `LocationUpdatesServices`.

Validazione internazionale del numero di telefono Per controllare i numeri di telefono, si è ricorsi alla libreria di Google **libphonenumber**, molto potente e robusta. **SmartTracker** considera due numeri di telefono come uguali qualora si verifichino le seguenti condizioni: il prefisso internazionale, il numero significativo nazionale, la presenza di uno zero iniziale per numeri italiani ed altre estensioni presenti siano uguali.

2.3 Archiviazione su server e visualizzazione da pagina web

Le posizioni che `LocationUpdateService` ottiene devono essere inviate al server per poter essere archiviate e consultate. Come Client HTTP si è scelto **Retrofit**, consideranto dagli esperti di Android e da Google stessa come il più affidabile. Il metodo che invia la richiesta è *addLocation()*, contenuto nell'interfaccia `ApiService`. **Retrofit** utilizza un'interfaccia come definizione degli endpoints. La richiesta è effettuata mediante `UploadWorker`, che estende `ListenableWorker` incluso nell'API **WorkManager**. Quest'ultima rende facile programmare compiti differibili ed asincroni che dovrebbero essere eseguiti anche se l'app viene chiusa ed il dispositivo viene riavviato. Pertanto, le posizioni da inviare al server non verranno mai perse qualora non ci fosse connessione, ma verrebbero archiviate e reinviate alla riconnessione.
Il server è hostato in-house su piattaforma **Apache** e per la comunicazione tra pagina web e database **MySQL** viene usato **PHP**.
Per l'implementazione della mappa sono stati utilizzati Google Maps ed al linguaggio di scripting **JavaScript**.

2.4 Interfaccia grafica chiara ed intuitiva

L'interfaccia grafica di **SmartTracker** segue le linee guida del **Material Design**, linguaggio visivo di casa Google che unisce i principi classici del buon design con l'innovazione della tecnologia e della scienza. Per questo motivo l'app avrà un aspetto familiare alle app native di Google e sarà chiara e semplice da usare, elemento di fondamentale importanza per le interfacce utente.
Per quanto concerne l'architettura, si è seguito il principio del **single Activity, multiple Fragments**: è presente una sola `Activity` che ospita i vari `Fragment` che compongono l'app.
La navigazione all'interno dell'app è stata implementata tramite l'API **Navigation**.
Per separare la gestione dei dati dalla loro rappresentazione, essi sono contenuti in `MainViewModel` che estende `ViewModel`. Questo permette di creare `Fragment` che si occupano solo di mostrare i dati e non, come spesso avviene, creare **God Objects**. Risultato di ciò è un codice più snello, robusto e facile da leggere e manutenere. Per ogni azione che l'utente può compiere, ogni possibile stato successivo è gestito e l'utente è sempre informato delle azioni che compie. Per esempio, al primo avvio dell'app in una versione di Android che richiede la gestione dei permessi al runtime, **SmartTracker** guiderà l'utente nella gestione degli stessi e notificherà lo stato attuale dei permessi con una breve spiegazione.
Inoltre l'app usa l'API **Data Binding**, che consente di associare i componenti dell'interfaccia utente contenuti nel layout ai dati nell'app utilizzando un formato dichiarativo anziché programmatico.

2.5 Metodologie di sviluppo

Come modello di sviluppo si è deciso di ricorrere al modello incrementale, le cui fasi sono ripetute più volte finché la valutazione del prodotto è soddisfacente. L'app è stata rilasciata internamente molteplici volte, ognuna delle quali con funzionalità aggiuntive, fino alla release finale, il cui codice sorgente è disponibile sotto Apache License, Version 2.0 all'indirizzo `https://github.com/yankarinRG/SmartTracker`.
Al fine di realizzare un prodotto solido ed affidabile, si è ricorsi quasi esclusivamente a librerie esterne di Google. Tra queste, citiamo **Android Architecture Components**, collezione di librerie che aiutano a sviluppare app robuste, testabili e mantenibili. Questi componenti aiutano a seguire le best practices, liberando dalla scrittura di codice boilerplate e semplificano le attività complesse.
Particolare attenzione è stata prestata alla scrittura di un codice elegante e piacevole da leggere; di conseguenza è stato seguito lo stile di scrittura dell'Android Open Source Project (AOSP).
Per la generazione dell'apk, al fine di minimizzare la dimensione di quest'ultimo, sono state attivate le opzioni di ottimizzazione e offuscazione del bytecode tramite **ProGuard**.
L'ambiente di sviluppo utilizzato per il lato client è **Android Studio**, mentre per il lato server si è optato per l'editor di testo **Visual Studio Code**.

Requisiti minimi Sistema operativo Android 4.1 Jelly Bean, Google Play Services installato, hardware GPS ed accesso ad internet.

3 Documentazione UML

3.1 Diagramma di classe

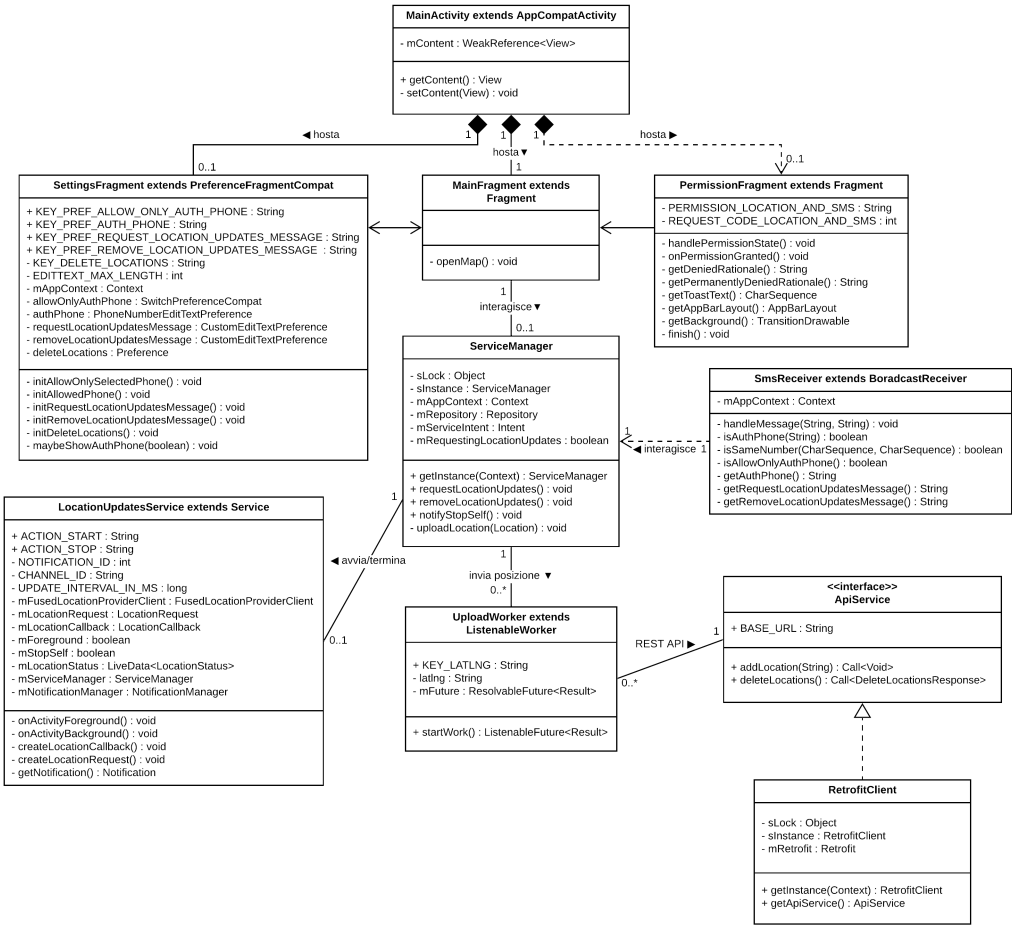

Figura 1: Il diagramma di classe chiarisce la relazione tra le varie classi che compongono **SmartTracker**. Per chiarezza, si è scelto di escludere le classi d'aiuto e che realizzano funzioni aggiuntive, lasciando quelle essenziali che rispettano i requisiti imposti.

3.2 Diagramma del caso d'uso

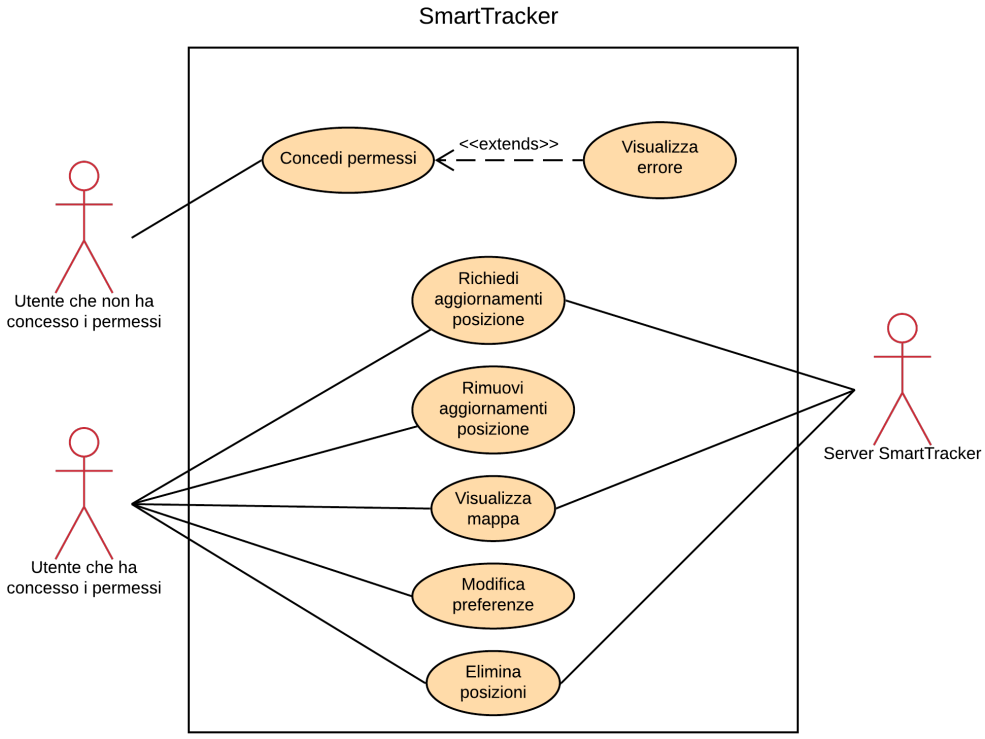

Figura 2: Il diagramma del caso d'uso evidenzia, in maniera astratta, le funzionalità e le azioni che l'utente può svolgere in **SmartTracker**. Si precisa che, al concedimento delle autorizzazioni necessarie, un utente che non ha concesso i permessi diventa un utente che ha concesso i permessi.

3.3 Diagramma di sequenza

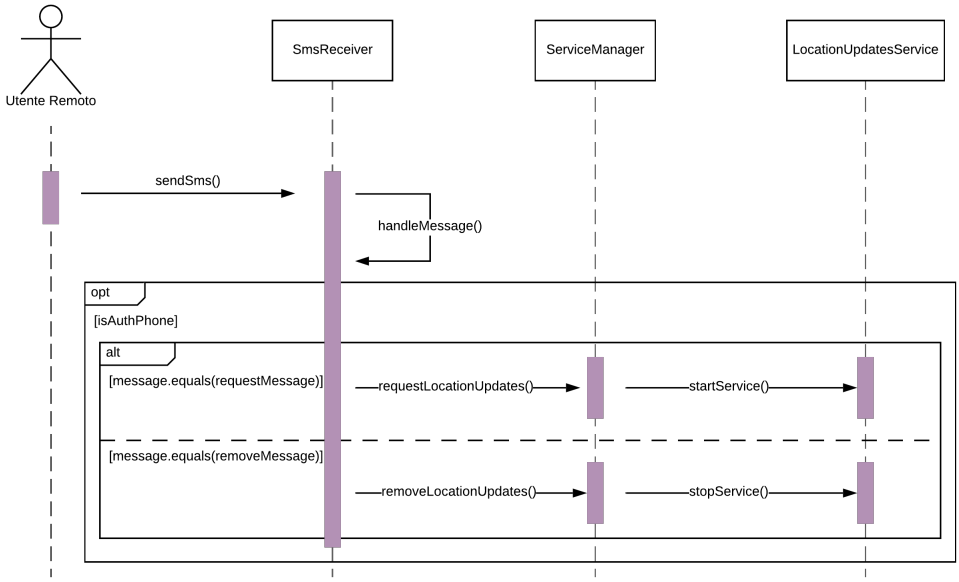

Figura 3: Il diagramma di sequenza mostra le operazioni che `SmsReceiver` effettua alla ricezione di un SMS, in base al mittente ed al contenuto dello stesso.

3.4 Diagramma di collaborazione

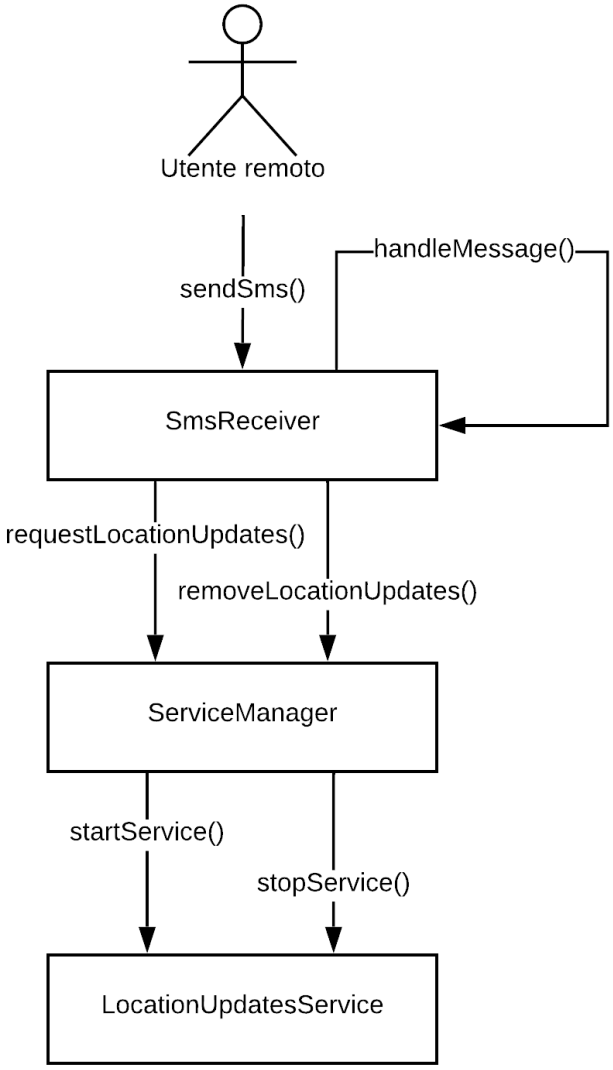

Figura 4: Il diagramma di collaborazione mostra l'interazione delle classi alla ricezione di un SMS.

3.5 Diagramma di stato

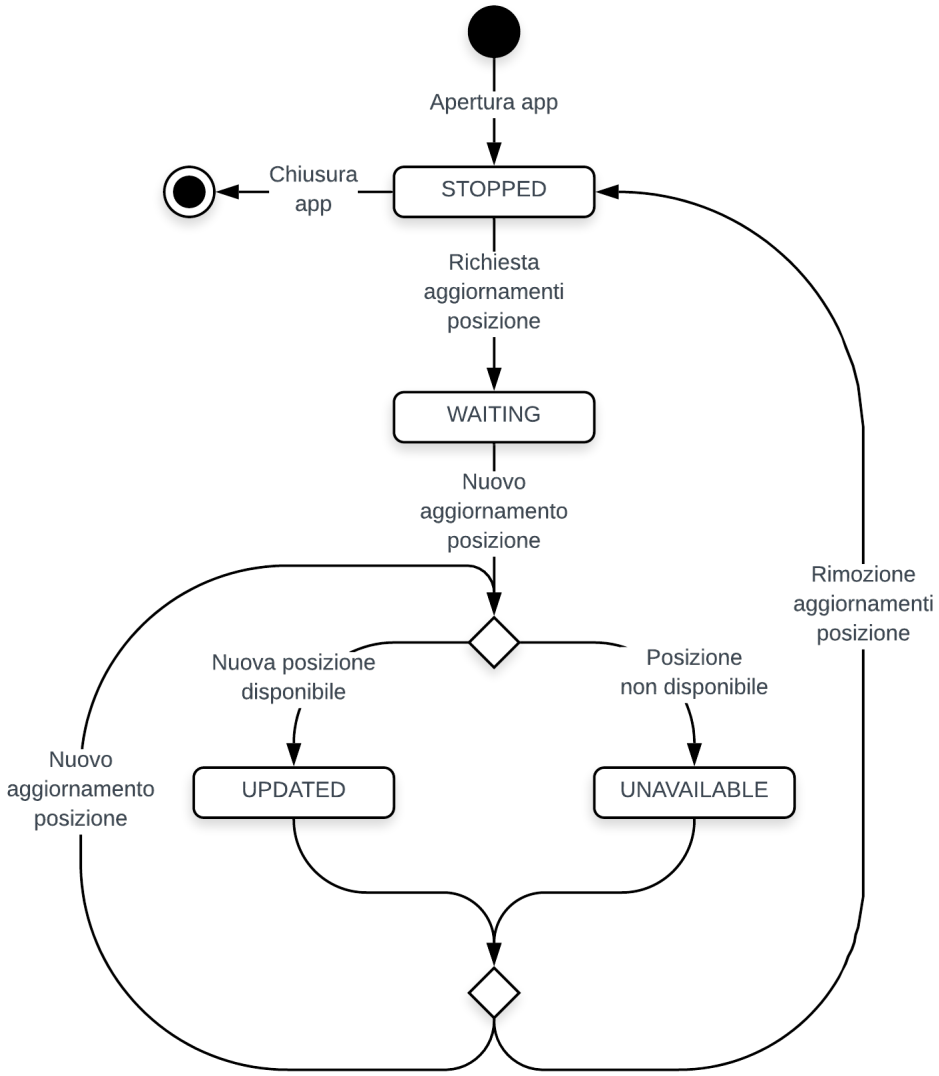

Figura 5: Il diagramma di stato fa vedere come varia `LocationStatus`, classe che rappresenta lo stato corrente degli aggiornamenti sulla posizione, utilizzata per il corretto aggiornamento dell'interfaccia grafica.

3.6 Diagramma di attività

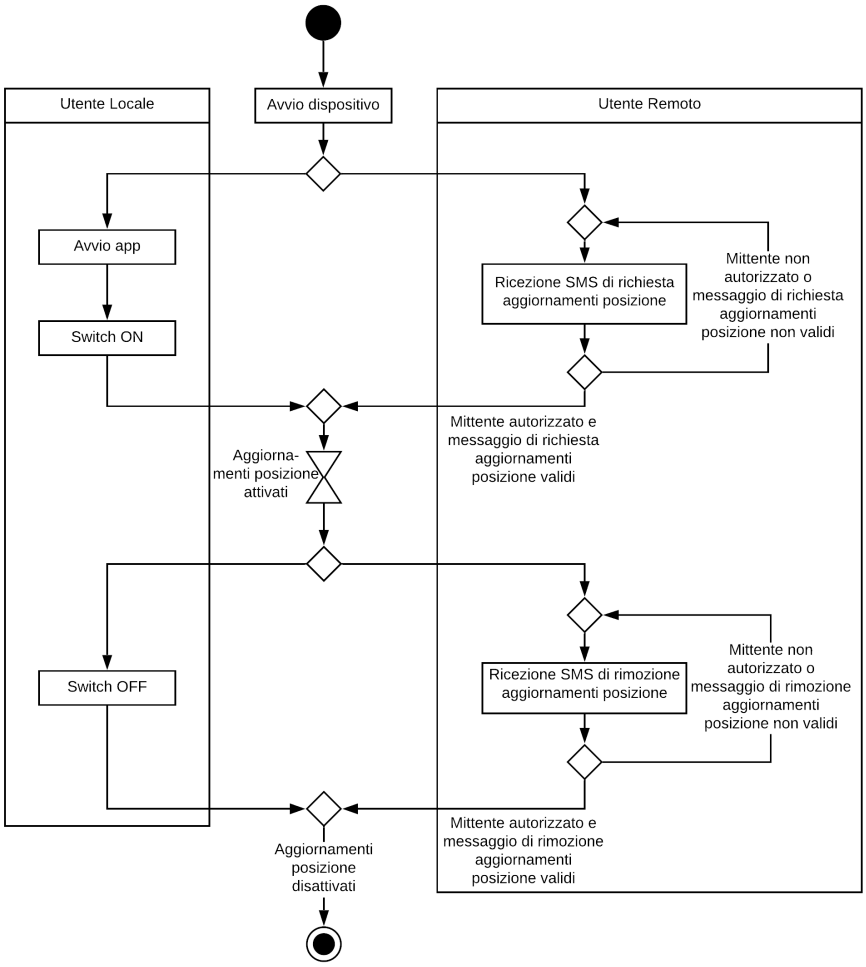

Figura 6: Il diagramma di attività evidenzia il normale utilizzo dell'app: gli aggiornamenti sulla posizione vengono attivati; per un po' di tempo, a cadenza periodica, vengono ottenuti nuovi aggiornamenti; successivamente, gli aggiornamenti vengono disattivati. Si noti come sia possibile richiedere o rimuovere gli aggiornamenti secondo le modalità precedentemente dette.

4 Casi di test funzionali

Si è scelto di limitare la lunghezza massima dei messaggi di attivazione e disattivazione degli aggiornamenti posizione a 20 caratteri. Questo vincolo, del tutto arbitrario, è stato imposto per facilitare l'esperienza utente, cosicché quest'ultimo dovrà ricordare un messaggio breve ed efficacie. Per ovvi motivi, il messaggio di richiesta aggiornamenti posizione deve essere diverso, senza considerare la sensibilità alle maiuscole, dal messaggio di rimozione aggiornamenti posizione. Qualora l'utente provasse ad inserire un messaggio uguale all'altro, verrà informato dell'errore. Lo stesso vale se si tenta di inserire un messaggio vuoto.

Di seguito, una tabella che mostra i possibili inserimenti di uno dei due messaggi di attivazione.

n.	Messaggio	Validità
1	START TRACKING	V
2	STOP TRACKING	V
3	precipitevolissimevolmente	NV
4	ϵ[2]	NV
5	`null`	NV

Dove abbiamo come casi estremi:

Lunghezza messaggio	0	1..20	21..`Integer.MAX_VALUE`
Validità	NV	V	NV

[2]Nella teoria dei linguaggi formali indica la parola (stringa) vuota.

5 Design pattern

Come consigliato dalla guida all'architettura dell'app di Google, l'applicazione è stata sviluppata implementando il pattern architetturale **Model-View-ViewModel**. Il fulcro del funzionamento di questo pattern è la creazione di un componente, il `ViewModel` appunto, che rappresenta tutte le informazioni e i comportamenti della corrispondente `View`. La `View` si limita infatti, a visualizzare graficamente quanto esposto dal `ViewModel`, a riflettere in esso i suoi cambi di stato oppure ad attivarne dei comportamenti. Inoltre, è stato aggiunto il pattern Repository. La combinazione di questi design pattern, uniti alla classe `LiveData` utilizzata in `Repository`, hanno facilitato l'implementazione dell'interfaccia grafica ed il corretto funzionamento di essa in ogni situazione. Grazie a quest'architettura, viene seguito il single responsibility principle ed il principio di separation of concerns. Queste best practices rendono il codebase più solido, più testabile, meno soggetto agli errori e favoriscono il riuso del codice.

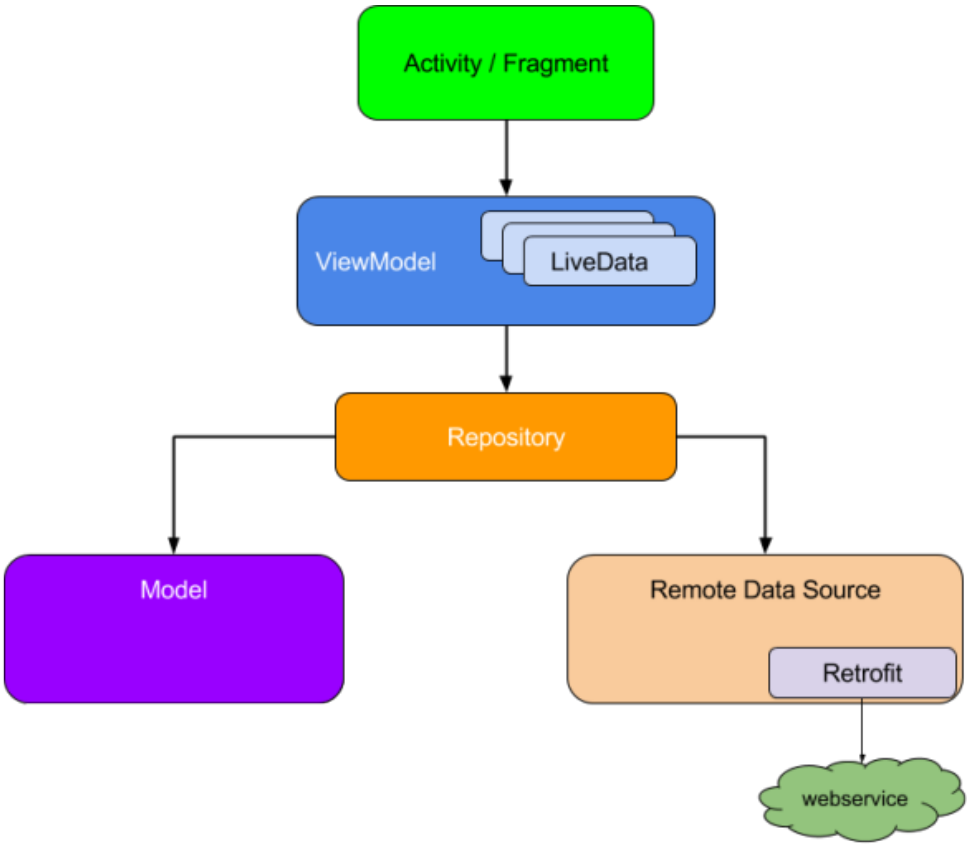

Figura 7: Il seguente diagramma mostra come i componenti interagiscono tra di loro.

Glossario

Best practices Insieme di tecniche o metodologie che, attraverso l'esperienza e la ricerca, hanno dimostrato di condurre in modo affidabile al risultato desiderato. 8, 16

Boilerplate Sezione di codice riutilizzata con piccole modifiche, spesso riferita a lunghe sezioni di codice che svolgono lavori minimali. 4, 8

Design pattern Soluzione generale e riutilizzabile per un problema che si verifica comunemente in un dato contesto nella progettazione del software. 16

Endpoint URL a cui è possibile accedere al servizio da un'applicazione client. 6

Framework Architettura logica di supporto, che è spesso un'implementazione logica di un particolare design pattern, su cui un software può essere progettato e realizzato, spesso facilitandone lo sviluppo da parte del programmatore. 5

God Object Anti-pattern di oggetti o classi che conoscono troppe informazioni o svolgono troppi compiti. 7

LiveData Contenitore di dati osservabili. Altri componenti dell'app possono monitorare le modifiche agli oggetti usando questo supporto senza creare percorsi di dipendenza espliciti e rigidi tra di loro. Il componente `LiveData` rispetta anche il ciclo di vita dei componenti dell'app, quali `Activity`, `Fragment` e `Service`, ed include una logica di pulizia per evitare perdite di oggetti ed un consumo eccessivo di memoria. 16

Repository Classe che gestise le operazioni sui dati e fornisce un'API pulita in modo che il resto dell'app possa recuperare facilmente questi dati. Sanno da dove ottenere i dati e quali chiamate API eseguire quando vengono aggiornati i dati. È possibile considerare i Repository come mediatori tra diverse origini dati, come modelli persistenti, servizi web e cache. 16

Separation of concerns Principio di progettazione per cui ogni modulo, classe o funzione dovrebbe avere la responsabilità su una singola parte della funzionalità fornita dal software e tale responsabilità dovrebbe essere interamente incapsulata dalla classe. 16

Single responsibility principle Principio di progettazione per separare un programma in sezioni (classi) distinte, in modo che ogni sezione risolva un problema separato. 16

ViewModel Fornisce i dati per un componente UI specifico, ad esempio un `Fragment` o `Activity`, e contiene la logica aziendale di gestione dei dati per comunicare con il modello. `ViewModel` può chiamare altri componenti per caricare i dati e inoltrare le richieste degli utenti per modificare i dati e non conosce i componenti dell'interfaccia utente, quindi non è interessato dalle modifiche alla configurazione, come la ricreazione di un'attività durante la rotazione del dispositivo. 7, 16