SAPIENZA
UNIVERSITÀ DI ROMA

Sviluppo di un sistema per rilevare spostamenti in auto tramite il Bluetooth degli smartphone

Facoltà di Ingegneria dell'Informazione, Informatica e Statistica
Corso di Laurea in Informatica

Antonio Pietro Romito
Matricola 1932500

Relatore
Prof. Emanuele Panizzi

Anno Accademico 2023/2024

Sviluppo di un sistema per rilevare spostamenti in auto tramite il Bluetooth degli smartphone

Relazione di tirocinio. Sapienza Università di Roma

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

Email dell'autore: antormt@gmail.com

Sommario

L'elaborato descrive lo sviluppo e l'integrazione di un sistema per rilevare gli spostamenti in auto tramite il Bluetooth degli smartphone nell'applicazione GeneroCity, un'applicazione di smart parking progettata dal Gamification Lab della Sapienza Università di Roma. Questo sistema, chiamato sensore Bluetooth, è in grado di rilevare automaticamente quando l'utente è alla guida di un'auto, sfruttando le interazioni implicite e analisi contestuale dei dati forniti dal Bluetooth degli smartphone, ed è stato sviluppato utilizzando le API di Android per la gestione delle connessioni Bluetooth. Il sistema è in grado di riconoscere se l'utente è alla guida con una certa confidenza contribuendo al riconoscimento dello stato dell'utente tramite un algoritmo di media pesata tra vari sensori. I test effettuati hanno dimostrato l'affidabilità del sensore nello scenario reale, confermandone l'efficacia e la corretta integrazione con l'app GeneroCity. Futuri sviluppi prevedono l'introduzione di modelli di machine learning per migliorare la determinazione dello stato dell'utente e l'implementazione di servizi esterni per il riconoscimento dei dispositivi Bluetooth.

Indice

Capitolo 1

Introduzione

GeneroCity è un'applicazione di smart parking[1], in sviluppo per smartphone Android e iOS, realizzata dal Gamification Lab del Dipartimento di Informatica della Sapienza Università di Roma. Lo scopo dell'applicazione è quello di facilitare la ricerca del parcheggio all'interno di un'area urbana [3].

Figura 1.1 Logo di GeneroCity

Il mio contributo a questo progetto è stato quello di realizzare un modulo software che, analizzando i dati forniti dal Bluetooth degli smartphone Android, rilevi automaticamente quando l'utente è alla guida della propria auto. Di seguito si spiegherà come questo scopo viene raggiunto attraverso il lavoro coordinato di vari **sensori**, tra cui quello Bluetooth da me realizzato.

1.1 I sensori e il loro utilizzo per ottenere interazioni implicite

Una delle caratteristiche principali dell'applicazione è l'utilizzo di interazioni implicite.

> **Implicit human computer interaction** is an action, performed by the user that is not primarily aimed to interact with a computerized system but which such a system understands as input. [7]

Con questa espressione si intende un tipo di interazione uomo-macchina che non richiede dei comandi espliciti da parte dell'utente, ma utilizza il contesto in cui quest'ultimo agisce come input per l'elaborazione [5]. Questo tipo di approccio è fondamentale per garantire la sicurezza degli utenti finali che utilizzeranno l'applicazione

[1] lo smart parking è una strategia che utilizza tecnologie digitali con l'obbiettivo di utilizzare il minor numero di risorse possibili (carburante, tempo, spazio) per ottenere un processo di sosta dei veicoli più veloce, facile e ottimizzato. [4]

durante la guida. Le interazioni implicite vengono ottenute grazie all'implementazione di moduli chiamati sensori.

I sensori

In GeneroCity un sensore è un modulo software che, analizzando il contesto in cui agisce l'utente in uno specifico istante, determina l'azione compiuta da quest'ultimo. In particolare ciascun sensore calcola un valore reale compreso tra 0 e 1, detto **confidenza**, il quale rappresenta il grado di sicurezza con con cui il sensore ha effettuato la rilevazione dell'azione di guida (dove 0 rappresenta una sicurezza minima e 1 massima). Più specificamente:

- un valore compreso tra 0 e 0,5 (stato *walking*) indica che l'utente non sta guidando;
- il valore 0,5 (stato *unknown*) denota che il sensore non è in grado di inferire lo stato dell'utente;
- una confidenza compresa tra 0,5 e 1 (stato *automotive*) segnala che l'utente sta guidando.

Per analizzare il contesto e calcolare la confidenza ogni sensore può adottare svariati approcci: esso si può basare sulle rilevazioni di un vero e proprio sensore hardware presente nello smartphone (come quelli di movimento o ambientali), oppure sullo stato di una specifica componente del dispositivo (ad esempio la batteria, il display, ecc.) o può fare uso della connettività Wi-Fi e Bluetooth o ancora utilizzare altre tecnologie e protocolli specifici, come la geolocalizzazione attraverso GPS. Lo stato effettivo dell'utente sarà poi determinato analizzando il risultato ottenuto da tutti i sensori in un preciso istante di tempo.

Una linea guida importante per lo sviluppo di un sensore è che esso si avvalga di una sola di queste tecnologie o componenti in modo da effettuare la computazione in maniera indipendente dagli altri: ad esempio non succederà mai che il sensore Bluetooth effettui una scansione dei dispositivi vicini quando il GPS rileva che l'utente si sta muovendo ad una certa velocità. Questo perché eventuali correlazioni tra sensori differenti verranno prese in considerazione dal sistema che, facendo uso di apprendimento automatico, riconoscerà questi legami ed effettuerà il calcolo dello stato dell'utente tenendone conto.

Capitolo 2

L'utilizzo di Android per lo sviluppo del sensore Bluetooth

Prima di sviluppare il sensore si è scelto di creare una piccola applicazione a sé stante, con lo scopo di prendere dimestichezza con le librerie di Android per Java, nello specifico con le sue funzionalità riguardanti la connettività via Bluetooth [1]. Ciò è stato fatto al fine di trovare una strategia efficiente da adottare nello sviluppo del sensore finale in GeneroCity, dato che essa contiene molte altre funzionalità e di conseguenza sarebbe stato più difficoltoso effettuare questo tipo di sperimentazione al suo interno. Questa applicazione permette l'esecuzione di tre task, la cui implementazione è stata ritenuta un buon punto di partenza per lo sviluppo del sensore Bluetooth. Esse sono:

- la rilevazione del cambiamento dello stato del modulo bluetooth dello smartphone, più specificamente la sua accensione e lo spegnimento;
- l'ottenimento dei dati riguardanti i dispositivi che vengono connessi al Bluetooth;
- l'esecuzione di scansioni per trovare dispositivi nelle vicinanze.

In questo capitolo verranno quindi illustrati i vari componenti dell'applicazione e come essi implementano le suddette task consentendone la visualizzazione dei risultati.

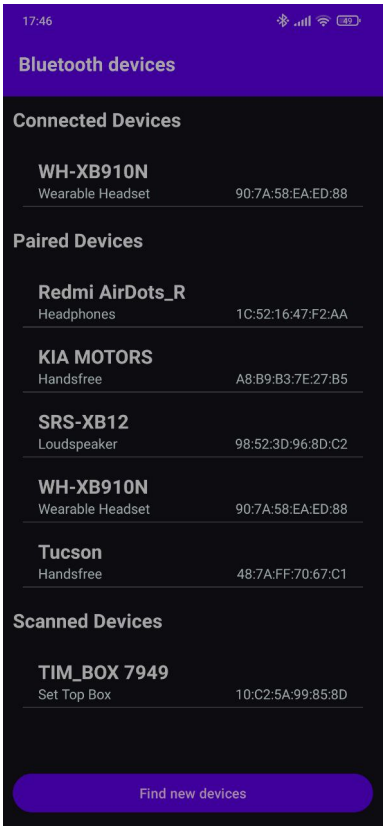


Figura 2.1 Interfaccia dell'applicazione

2.1 I Broadcast Receiver

Il primo ostacolo che si è presentato nella realizzazione dell'applicazione è stato l'impossibilità di ottenere la lista dei dispositivi connessi al Bluetooth in maniera sincrona tramite le funzionalità messe a disposizione dalle API[1] di Android. Infatti il sistema operativo non permette di essere interrogato direttamente dai singoli processi, piuttosto è esso a notificarli quando avvengono degli eventi di sistema, inviando dei messaggi denominati *system broadcast*. Le applicazioni potranno quindi registrarsi per ricevere messaggi relativi ad eventi specifici, ad esempio, quando avviene l'accensione e lo spegnimento della modalità aereo, il sistema manderà un system broadcast a tutte le app registrate per ricevere questo evento. Questa registrazione avviene attraverso la creazione di oggetti la cui classe estende quella dei *BroadcastReceiver*.(2)

La classe sopracitata espone un metodo astratto, denominato *onReceive*, il quale deve essere implementato per definire il comportamento adottato dal programma quando viene ricevuto un system broadcast. Questo metodo ha come parametri il contesto, vale a dire lo stato dell'applicazione al momento in cui è avvenuto l'evento, e l'intent, ossia una rappresentazione dell'evento per cui è scaturita la notifica.

```
1 public abstract void onReceive(Context context, Intent intent);
```

Per attuare l'effettiva registrazione al fine di ricevere i system broadcast attraverso un broadcast receiver è necessario utilizzare il metodo *registerReceiver* della classe Context. Per far ciò bisogna inoltre dichiarare quali tipologie di eventi esso ascolta, creando un *IntentFilter*. Viene riportato qui sotto un esempio di codice dove viene registrato un broadcast receiver:

```
1 BroadcastReceiver br = new MyBroadcastReceiver();
2 IntentFilter filter = new IntentFilter(APP_SPECIFIC_BROADCAST);
3 context.registerReceiver(br, filter);
```

Tutti i broadcast receiver definiti nell'applicazione si occuperanno solamente di catturare l'evento, sarà poi un metodo astratto, specifico per ognuno di essi, ad essere esteso per implementare l'algoritmo scaturito da esso. Questi metodi saranno estesi direttamente alla creazione dei receiver nella classe Controller, la quale verrà discussa nel paragrafo 2.2. Inoltre tutti i receiver intercettano gli eventi anche quando l'applicazione è in background, pertanto il funzionamento dell'app è inalterato quando essa è in esecuzione e l'utente non la sta utilizzando direttamente. Di seguito verranno elencati i broadcast receiver implementati, riportandone il codice del metodo *onReceive*, e per ciascuno di essi verranno discussi i *system broadcast* che riceveranno.

2.1.1 Lo StatusChangeReceiver

Lo StatusChangeReceiver è il broadcast receiver più semplice ed è stato definito per rilevare l'accensione e lo spegnimento del Bluetooth del dispositivo. Questo receiver

[1] Una API, acronimo di Application Software Interface, è un tipo di interfaccia software che permette ad un programma di offrire servizi e funzionalità ad altri software.

si occuperà di intercettare gli eventi che cambiano lo stato della radio Bluetooth, ossia del tipo *BluetoothAdapter.ACTION_STATE_CHANGED*.

```
@Override
public void onReceive(Context context, Intent intent) {
    final String action = intent.getAction();

    if (action != null &&
    ↪  action.equals(BluetoothAdapter.ACTION_STATE_CHANGED)) {
        final int state =
        ↪  intent.getIntExtra(BluetoothAdapter.EXTRA_STATE,
        ↪  BluetoothAdapter.ERROR);
        switch (state) {
            case BluetoothAdapter.STATE_OFF:
                onStatusChange(context, false);
                break;
            case BluetoothAdapter.STATE_TURNING_OFF:
                Toast.makeText(context, "Turning bluetooth off...",
                ↪  Toast.LENGTH_SHORT).show();
                break;
            case BluetoothAdapter.STATE_ON:
                onStatusChange(context, true);
                break;
            case BluetoothAdapter.STATE_TURNING_ON:
                Toast.makeText(context, "Turning bluetooth on...",
                ↪  Toast.LENGTH_SHORT).show();
                break;
            default:
                break;
        }
    }
}
```

Quando il Bluetooth viene acceso o spento si esegue il metodo astratto *onStatusChanged*, il quale ha come parametro un booleano che indica lo stato del Bluetooth (true quando è acceso e false se spento). Questo metodo sarà quindi esteso per definire cosa l'applicazione dovrà fare quando ciò avviene.

```
public abstract void onStatusChange(Context context, boolean
↪  bluetoothStatus);
```

2.1.2 Il FoundDeviceReceiver

Questo broadcast receiver si occupa di ricevere l'evento riguardante la scoperta di un nuovo dispositivo Bluetooth quando viene effettuata la scansione dei dispositivi nelle vicinanze. Esso riceverà le notifiche degli eventi di tipo *BluetoothDevice.ACTION_FOUND* e si occuperà di prelevare le informazioni riguardanti

dispositivi rilevati attraverso la funzione di utilità *BluetoothDeviceUtils.map*. Essa prende in input il dispositivo Bluetooth e restituisce un oggetto che ne contiene le informazioni di interesse per l'applicazione (indirizzo MAC, nome del dispositivo e classe bluetooth).

```
@Override
public void onReceive(Context context, Intent intent) {
    String action = intent.getAction();

    if (action == null
            || !action.equals(
            android.bluetooth.BluetoothDevice.ACTION_FOUND
        )) {
        return;
    }

    android.bluetooth.BluetoothDevice device;
    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.TIRAMISU) {
        device = intent.getParcelableExtra(
                android.bluetooth.BluetoothDevice.EXTRA_DEVICE,
                android.bluetooth.BluetoothDevice.class
        );
    } else {
        device = intent.getParcelableExtra(
                android.bluetooth.BluetoothDevice.EXTRA_DEVICE
        );
    }
    if (device == null) {
        return;
    }

    BluetoothDevice newDevice = BluetoothDeviceUtils.map(device);
    onDeviceFound(newDevice);
}
```

Come si può vedere dal codice, una volta ottenuti i dati del dispositivo viene chiamato un metodo astratto denominato *onDeviceFound* che verrà esteso per salvare il dispositivo in una struttura dati.

```
public abstract void onDeviceFound(BluetoothDevice device);
```

2.1.3 Il ConnectionReceiver

Gli eventi di tipo *BluetoothDevice.ACTION_ACL_CONNECTED* e *BluetoothDevice.ACTION_ACL_DISCONNECTED* sono gestiti da questo receiver per ottenere le informazioni sui dispositivi che vengono connessi e disconnessi dal Bluetooth. Quan-

do ciò avviene verrà utilizzata la funzione di mappatura *BluetoothDeviceUtils.map*, citata precedentemente, per prelevare le informazioni di interesse dai dispositivi.

```
@Override
public void onReceive(Context context, Intent intent) {
    String action = intent.getAction();

    if (action == null
        || !action.equals(
            android.bluetooth.BluetoothDevice.ACTION_ACL_CONNECTED
        ) && !action.equals(
            android.bluetooth.BluetoothDevice
                .ACTION_ACL_DISCONNECTED
        )) {
        return;
    }

    android.bluetooth.BluetoothDevice device;
    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.TIRAMISU) {
        device = intent.getParcelableExtra(
                android.bluetooth.BluetoothDevice.EXTRA_DEVICE,
                android.bluetooth.BluetoothDevice.class
        );
    } else {
        device = intent.getParcelableExtra(
                android.bluetooth.BluetoothDevice.EXTRA_DEVICE
        );
    }

    if (device == null) {
        return;
    }

    BluetoothDevice newDevice = BluetoothDeviceUtils.map(device);

    if (action.equals(android.BluetoothDevice.ACTION_ACL_CONNECTED))
    ↪ {
        onDeviceConnection(context, newDevice);
    } else {
        onDeviceDisconnection(context, newDevice);
    }
}
```

Infine viene invocato il metodo astratto *onDeviceConnection* in caso di connessione del dispositivo oppure *onDeviceDisconnection* se si tratta di una disconnessione. Questi metodi verranno estesi per tenere aggiornata una struttura dati che mantiene in memoria i dispositivi connessi via Bluetooth.

```
public abstract void onDeviceConnection(Context context,
↪ BluetoothDevice device);
public abstract void onDeviceDisconnection(Context context,
↪ BluetoothDevice device);
```

2.1.4 L'ImplicitConnectionReceiver

La creazione di questo receiver deriva dal fatto che tutti quelli descritti precedentemente vengono registrati quando viene avviata l'applicazione, pertanto, se ci sono dei dispositivi Bluetooth già connessi prima dell'avvio, il ConnectionReceiver non è in grado di rilevarli. L'ImplicitConnectionReceiver nasce per risolvere questo problema, difatti esso è una sottoclasse di ConnectionReceiver e funzionerà anche quando l'applicazione è completamente chiusa: esso viene registrato nel manifest[2] dell'applicazione e sarà quindi il sistema operativo a *svegliarla* se essa non è in esecuzione, così da permettere l'esecuzione di codice in risposta ad una connessione o disconnessione di un dispositivo Bluetooth. Segue la porzione di manifest dove viene dichiarato il receiver specificandone gli intent per cui esso sarà notificato.

```
<receiver
    android:name="receivers.ImplicitConnectionReceiver"
    android:exported="true">
    <intent-filter>
        <action
            android:name=
            ↪ "android.bluetooth.device.action.ACL_CONNECTED"/>
        <action
            android:name=
            ↪ "android.bluetooth.device.action.ACL_DISCONNECTED"/>
    </intent-filter>
</receiver>
```

Esso essendo di tipo ConnectionReceiver eredita anche il metodo onReceive di quest'ultimo, tuttavia deve fornire un'implementazione dei suoi metodi astratti. Nel caso della connessione di un dispositivo il metodo *onDeviceConnection* si occuperà di caricare l'insieme dei dispositivi connessi in quel momento da un file, di aggiungere il dispositivo connesso in quel momento e di salvare l'insieme aggiornato sul file.

```
@Override
public void onDeviceConnection(Context context, BluetoothDevice
↪ device) {
    Set<BluetoothDevice> connectedDevices =
    ↪ loadConnectedDevices(context);
    connectedDevices.add(device);
```

[2]Il manifest Android è un file che definisce la struttura, le funzionalità e i requisiti di un'applicazione Android.

```
5       saveConnectedDevices(context, connectedDevices);
6   }
```

Per quanto riguarda il metodo *onDeviceDisconnection* viene fatta pressoché la medesima cosa: viene caricato l'insieme dei dispositivi, rimosso il dispositivo disconnesso e salvato l'insieme aggiornato.

```
1   @Override
2   public void onDeviceDisconnection(Context context, BluetoothDevice
    ↪  device) {
3       Set<BluetoothDevice> connectedDevices =
        ↪  loadConnectedDevices(context);
4       connectedDevices.remove(device);
5       saveConnectedDevices(context, connectedDevices);
6   }
```

Grazie a questo receiver, il Controller, non appena verrà istanziato all'avvio dell'applicazione, potrà ottenere i dispositivi connessi prima dell'avvio, in quanto verranno caricati dal file da esso aggiornato.

2.2 Il Controller

Il Controller è l'unità centrale dell'applicazione, infatti esso si occupa di mantenere gli insiemi dei dispositivi Bluetooth accoppiati al telefono[3], quelli rilevati dalle scansioni e infine i dispositivi connessi. Esso implementa due design pattern[4]: il **singleton pattern**, il quale consente alla classe Controller di avere un'unica istanza accessibile da qualunque punto del codice, e l'**observer pattern** che permette ad altri oggetti di registrarsi per venire notificati dal Controller quando esso cambia il suo stato. In particolare lo stato del Controller è rappresentato dai suoi seguenti attributi:

- *isBluetoothEnabled*, un flag che indica se il modulo Bluetooth del dispositivo è acceso o spento;
- *pairedDevices*, l'insieme dei dispositivi Bluetooth accoppiati allo smartphone;
- *scannedDevices*, l'insieme dei dispositivi Bluetooth trovati durante l'ultima scansione;
- *connectedDevices*, l'insieme dei dispositivi Bluetooth connessi allo smartphone.

Inoltre è presente una lista di listener, ossia degli oggetti che verranno notificati dal Controller ogni qualvolta esso cambierà stato. Nello specifico si tratta di una

[3] Con accoppiamento Bluetooth si intende il pairing, ossia il processo in cui due dispositivi Bluetooth effettuano la prima connessione e si scambiano le chiavi di sicurezza, le quali verranno memorizzate per permettere di effettuare rapidamente le connessioni successive.

[4] Un design pattern, traducibile in schema di progettazione, nell'ambito dell'ingegneria del software è una soluzione generale ad un problema ricorrente.

lista di *Runnable*, ossia un'interfaccia funzionale[5] il cui metodo non prende input né produce output, ma esegue semplicemente una sequenza di istruzioni. Essi verranno eseguiti sequenzialmente uno dopo l'altro ogni volta che verrà chiamato il metodo *notifyListener*, cosa che avverrà ad ogni cambiamento di stato del Controller.

Come detto in precedenza il Controller utilizza i broadcast receiver sopradescritti in modo da poter aggiornare il suo stato in risposta agli eventi di sistema. Tutti i receiver verranno creati come costanti della classe Controller e saranno registrati nel suo costruttore. Nello specifico vengono estese in maniera anonima le loro classi in modo da fornire un'implementazione dei metodi astratti. Di seguito verrà descritto come gli aggiornamenti di stato avvengono attraverso questi metodi.

2.2.1 Stato del modulo Bluetooth (accensione e spegnimento)

Per rilevare l'accensione e lo spegnimento del Bluetooth dello smartphone, il Controller crea uno StatusChangeReceiver fornendo un'implementazione del metodo onStatusChange. Esso ha il compito di assegnare il nuovo valore al flag isBluetoothEnabled del controller sulla base del parametro booleano bluetoothStatus, il quale sarà true se il bluetooth è acceso o false altrimenti. In caso di spegnimento, verrà svuotato l'insieme dei dispositivi connessi. Infine verranno notificati i listener del cambiamento.

```
1  private final StatusChangeReceiver statusChangeReceiver = new
   ↪  StatusChangeReceiver() {
2
3          @Override
4          public void onStatusChange(Context context, boolean
           ↪  bluetoothStatus) {
5              isBluetoothEnabled = bluetoothStatus;
6              if (!bluetoothStatus) {
7                  connectedDevices.clear();
8              }
9              notifyListeners();
10         }
11 };
```

2.2.2 Connessione e disconnessione di un dispositivo

Allo stesso modo verrà creato un ConnectionReceiver per aggiornare l'insieme dei dispositivi connessi quando uno di essi si connette o disconnette. Per raggiungere questo scopo, vengono quindi implementati i metodi onDeviceConnection e onDeviceDisconnection: nel primo viene aggiunto il dispositivo ottenuto all'insieme dei dispositivi connessi, mentre nel secondo il dispositivo viene rimosso. In entrambi i casi vengono notificati i listener.

[5] In Java un'interfaccia funzionale è un'interfaccia che contiene un solo metodo astratto. Le interfacce funzionali permettono di utilizzare il paradigma di programmazione funzionale, facendo in modo di rappresentare delle funzioni come oggetti e potendole quindi passare come parametri ad altre funzioni.

```
private final ConnectionReceiver connectionReceiver = new
↪  ConnectionReceiver() {

    @Override
    public void onDeviceConnection(Context context, BluetoothDevice
    ↪  device) {
        connectedDevices.add(device);
        notifyListeners();
    }

    @Override
    public void onDeviceDisconnection(Context context,
    ↪  BluetoothDevice device) {
        connectedDevices.remove(device);
        notifyListeners();
    }
};
```

2.2.3 Scoperta di un nuovo dispositivo durante la scansione

Quando invece verrà effettuata la scansione dei dispositivi Bluetooth nelle vicinanze, sarà il FoundDeviceReceiver a essere notificato ogni volta che ne viene trovato uno. Viene infatti implementato il suo metodo onDeviceFound per aggiungere il dispositivo all'insieme dei dispositivi trovati tramite la scansione e successivamente vengono notificati i listener del controller.

```
private FoundDeviceReceiver foundDeviceReceiver = new
↪  FoundDeviceReceiver() {

    @Override
    public void onDeviceFound(BluetoothDevice device) {
        scannedDevices.add(newDevice);
        notifyListeners();
    }
};
```

2.3 La presentazione dei dati

L'ultimo componente di quest'applicazione è l'interfaccia utente che permetterà la visualizzazione dispositivi amministrati dal Controller e di far partire le scansioni Bluetooth attraverso un bottone. Essa mostrerà i dispositivi divisi in tre elenchi differenti: uno per i dispositivi connessi, uno per quelli accoppiati e infine uno per quelli trovati via scansione. Queste liste sono amministrate da tre oggetti diversi tutti appartenenti alla classe *BluetoothDeviceAdapter*, la quale si occupa di convertire

gli elementi di una lista di dispositivi in un componente grafico che ne riporta le relative informazioni, come visibile in figura 2.1. Inoltre in questa classe è presente un metodo astratto, denominato *updateDevices*, che si occupa di aggiornare la lista dei dispositivi dell'adapter:

```
1 public abstract void updateDevices();
```

Esso sarà implementato alla creazione dei vari BluetoothDeviceAdapter in modo da ottenere dal Controller la lista dei dispositivi utili per l'adapter e successivamente notificarlo del cambiamento. Così facendo esso renderizzerà nuovamente i suoi componenti grafici con i nuovi dati. Poiché l'implementazione è pressoché uguale per tutti gli adapter, si riporta come esempio solamente il codice di quello utilizzato per aggiornare i dispositivi connessi:

```
1 BluetoothDeviceAdapter connectedDeviceAdapter = new
  ↪ BluetoothDeviceAdapter(this) {
2     @Override
3     public void updateDevices() {
4         BluetoothController controller = BluetoothController
5             .getInstance(MainActivity.this);
6         devices = controller.getConnectedDevices();
7         notifyDataSetChanged();
8     }
9 };
```

L'unica differenza che avranno gli altri BluetoothDevice adapter consiste nella lista di dispositivi che essi richiederanno al Controller: per ottenere la lista aggiornata l'adapter che si occupa dei dispositivi accoppiati utilizza il metodo *getPairedDevices* del Controller, quello che si occupa dei dispositivi scansionati il metodo *getScannedDevices*, mentre, come visto sopra, quello relativo ai dispositivi connessi usa *getConnectedDevices*.

Infine i metodi updateDevices di ogni BluetoothDeviceAdapter saranno registrati come listener del Controller. In tal modo, ogni qualvolta viene aggiornato lo stato di quest'ultimo e vengono notificati, i listener questi metodi saranno eseguiti così da avere un riscontro grafico dell'aggiornamento avvenuto.

```
1 controller.addListener(pairedDeviceAdapter::updateDevices);
2 controller.addListener(scannedDeviceAdapter::updateDevices);
3 controller.addListener(connectedDeviceAdapter::updateDevices);
```

Ora che sono stati definiti tutti i componenti si può fornire un esempio del flusso di aggiornamento dell'applicazione, nello specifico esso verrà descritto in seguito alla connessione di un dispositivo Bluetooth: quando ciò accade il ConnectionReceiver viene notificato ed esegue il metodo onDeviceConnection, che si occupa di aggiungere il dispositivo all'insieme dei dispositivi connessi del Controller. Una volta fatto ciò, a seguito del cambiamento di stato, verranno eseguiti sequenzialmente tutti i listener registrati. Tra di essi è presente il metodo updateDevices del connectedDeviceAdapter

il quale interrogherà il Controller per ottenere la lista aggiornata dei dispositivi connessi e farà partire la procedura di rendering grafico della lista, utilizzando i nuovi dati ottenuti. In maniera equivalente ciò avverrà per tutte le altre liste quando accade un evento che aggiorna uno degli insiemi dei dispositivi. Invece, per quanto riguarda il cambiamento di stato del modulo Bluetooth del telefono, se il Bluetooth viene spento, le liste saranno rimpiazzate da un messaggio che intima all'utente di accenderlo e saranno nuovamente mostrate quando esso viene acceso.

2.4 Testing

Di pari passo con lo sviluppo di questa applicazione è stato verificato il suo funzionamento su più smartphone e, al termine di esso, è stato possibile rilevare le connessione di dispositivi Bluetooth sia con l'applicazione in foreground che in background ed effettuare scansioni per trovarne di nuovi, il tutto con successo e senza incorrere in nessun tipo di errore. Inoltre qualche test è stato effettuato utilizzando il Bluetooth di alcune autoradio ed è stato notato che tutte le macchine prese in esame sono rilevabili via scansione Bluetooth esclusivamente quando viene effettuata la procedura per accoppiare un nuovo smartphone ad esse. L'impossibilità trovare le macchine nelle vicinanze attraverso la scansione rende l'utilizzo di questa funzionalità pressoché inutile in GeneroCity, di conseguenza si è scelto di non implementarla nel sensore Bluetooth. A questo punto si è potuto procedere con la realizzazione del sensore Bluetooth andando ad integrare in GeneroCity un sistema basato su quanto progettato per quest'applicazione

Nel prossimo capitolo verrà illustrata Generocity e in particolare il suo sistema di sensori per poi descrivere, nel capitolo 4, l'integrazione del nuovo sensore Bluetooth.

Capitolo 3

GeneroCity

In questo capitolo verrà descritta l'applicazione di GeneroCity e in particolar modo come i sensori operano in essa. Verrà prima delineata la sua architettura generale per poi approfondire i dettagli implementativi riguardanti l'integrazione dei sensori nell'applicazione per Android.

3.1 L'architettura dell'applicazione

L'applicazione di Generocity è basata su un'architettura di tipo client-server[1] ed è quindi divisa in due componenti principali tra loro interconnesse: il *backend* e il *frontend*. Queste due componenti comunicano utilizzando il protocollo HTTP[2].

3.1.1 Backend

Il backend è il software che si occupa dell'effettiva elaborazione dei dati e di gestire il database al fine di immagazzinarli e accedere ad essi. Esso espone delle funzionalità attraverso una web API, definendo come le altre componenti software debbano interagire con il server senza la necessità di condividere il codice sorgente, ma effettuando delle richieste HTTP a degli specifici URL chiamati endpoint. In particolare il backend di GeneroCity adotta una filosofia REST, la quale definisce una serie di principi chiave per la progettazione di un'API.

Figura 3.1 Comunicazione tramite API REST

[1] Il sistema client-server è un'architettura di rete dove un dispositivo terminale o client si connette ad un server per usufruire di un servizio.

[2] HTTP, acronimo di HyperText Transfer Protocol, è il principale protocollo di livello applicativo usato per il trasferimento di dati sul web.

3.1.2 Frontend

Il frontend invece è il programma con cui l'utente finale interagisce direttamente, in questo caso attraverso un'interfaccia grafica. Questa componente invierà le richieste agli endpoint definiti dal backend per inviare e ottenere dei dati. Esistono due versioni differenti del frontend di Generocity: un'applicazione per Android scritta in Java e una in Swift per iOS. Si è scelto di sviluppare due applicazioni in maniera nativa anziché utilizzare un approccio cross-platform[3] a causa dell'estensivo utilizzo di funzionalità di sistema che verrà fatto dai sensori.

La loro gestione, infatti, avviene interamente nel frontend e richiede che il loro funzionamento sia ben coordinato per garantire che lo stato dell'utente sia continuamente aggiornato. Di seguito verranno analizzati i dettagli implementativi riguardanti questo aggiornamento.

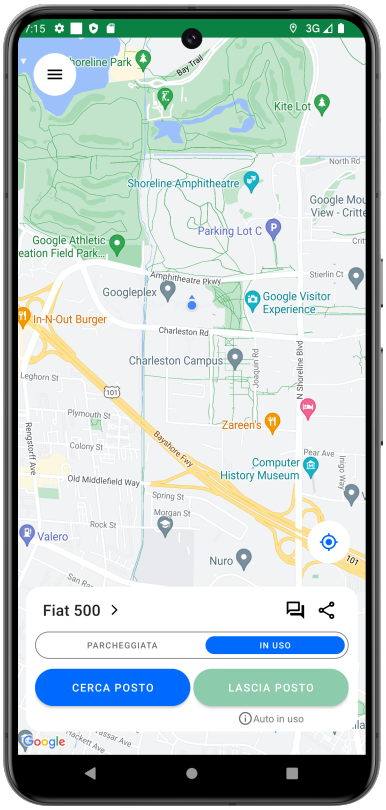


Figura 3.2 GeneroCity Android

3.2 Il flusso di aggiornamento della confidenza dei sensori

Dato che in GeneroCity i sensori presenti sono molteplici, un fattore importante è la loro coordinazione: lo stato dell'utente sarà determinato basandosi sulla confidenza di ogni sensore ed è quindi importante che esso sia aggiornato ogni qualvolta un sensore cambia la sua confidenza. Per sincronizzare il lavoro dei sensori è stato quindi fondamentale definire delle funzionalità comuni a tutti i sensori. Questo è stato possibile sfruttando il paradigma di programmazione orientata agli oggetti utilizzato da Java, in particolare è stata definita una classe astratta[4] che tutti i sensori ereditano.

3.2.1 La classe astratta Sensor

Ogni sensore eredita dalla classe Sensor i seguenti attributi: il *nome* univoco che identifica il sensore, la *versione* del sensore, il *peso* (rappresentato come numero reale) che ha il sensore nel calcolo dello stato dell'utente.

Inoltre, la confidenza calcolata da ogni sensore è mantenuta in uno storico tramite una mappa ad albero[5] che adotta come chiave lo Unix Timestamp, che indica il

[3] Con cross-platform si intende un'applicazione che viene sviluppata una sola volta con un unico codice sorgente, ma può essere eseguita su diversi sistemi operativi. Differisce dall'approccio nativo, il quale prevede l'utilizzo di un linguaggio di programmazione specifico per la piattaforma utilizzata.

[4] Nella programmazione orientata agli oggetti una classe astratta è una classe che non può essere istanziata, la quale definisce delle funzionalità di base per tutte le sue sotto classi.

[5] Una treemap è una struttura dati che implementa le funzionalità di normale una normale mappa, ossia immagazzina delle coppie (chiave, valore), e inoltre mantiene l'ordinamento delle coppie basato sull'ordinamento naturale delle chiavi.

momento in cui il valore è stato calcolato, ovvero il numero di secondi trascorsi dalla mezzanotte del 1° Gennaio 1970.

Di seguito vengono riportati i principali metodi esposti da questa classe.

getStatus

```
public abstract double getStatus(Calendar timestamp);
```

Esso ha come parametro un Timestamp e restituisce la confidenza del sensore, calcolata nell'istante più vicino a quello passato come parametro. Questo è l'unico metodo astratto ed è quindi il metodo principale che i sensori concreti dovranno implementare.

update

```
public void update(Calendar timestamp, SensorData sensorData) {
    collect(timestamp, sensorData);
    GCSensorConstants.onUpdate(timestamp);
}
```

Il metodo update riceve in input un Timestamp e un oggetto di un tipo da noi definito, denominato SensorData, contenente i dati inviati dal sensore. Esso si occupa di inviare i dati raccolti al server attraverso il metodo *collect* e di innescare il ricalcolo dello stato dell'utente. Quando viene eseguito il metodo update viene notificata l'unità centrale: una classe statica chiamata SensorConstants. Essa, attraverso il metodo *onUpdate*, richiede lo stato di ogni sensore (utilizzando il metodo getStatus) e calcola il nuovo stato dell'utente attraverso il metodo *compute*. Attualmente questo stato viene rappresentato come la media pesata delle confidenza di ogni sensore in uno specifico momento.

GCSensorConstants.onUpdate

```
static void onUpdate(Calendar time) {
    if (updating) {
        return;
    }

    updating = true;
    double computed = compute(time);
    computeHistory.put(time.getTimeInMillis(), computed);
    updating = false;
}
```

GCSensorConstants.compute

```
private static double compute(Calendar time) {
    double sum = 0.0;
    double wei = 0.0;

    double confidence;
    for (GCSensorInterface module : sensors) {
        confidence = module.getStatus(time);

        sum += (confidence - 0.5d) * module.weight;
        wei += module.weight;
    }

    return sum / wei + 0.5d;
}
```

3.2.2 L'invio di dati al server

Come anticipato, ogni qualvolta viene aggiornata la confidenza di un sensore, il metodo update richiede in input un oggetto di tipo SensorData che rappresenta lo stato del sensore al momento dell'aggiornamento. Ciò accade perché, prima che venga ricalcolata la confidenza media, viene chiamato il metodo *collect* che serializza l'oggetto in formato JSON e lo invia al server tramite una richiesta HTTP all'endpoint in figura 3.3. Esso si occuperà di memorizzare i dati forniti dal sensore nel database. Questi dati vengono mantenuti sia per tenere traccia del corretto funzionamento dei sensori, sia perché verranno utilizzati per l'allenamento di un modello di Machine Learning che verrà implementato in futuro in GeneroCity per determinare lo stato dell'utente.

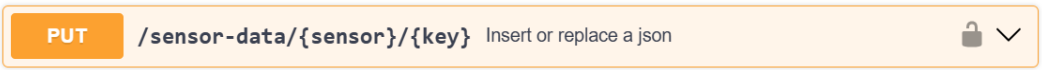


Figura 3.3 L'endpoint chiamato per memorizzare i dati dei sensori sul database

collect

```
public void collect(Calendar timestamp, SensorData sensorData) {
    long tsMillis = timestamp.getTimeInMillis();
    if (tsMillis - lastApiTimestamp <= getApiCooldownMs()) {
        return;
    }

    lastApiTimestamp = tsMillis;
```

```
    String key = ver + "-" + app.config().getMyId() + "-" +
    ↪ timestamp.getTimeInMillis();
    app.api().setSensorData(name, key,
    ↪ sensorData).enqueue(updateCall);
}
```

La classe SensorData serializza le seguenti informazioni:

- *datetime*: la data e l'ora in cui è avvenuto l'aggiornamento seguendo il formato definito nello standard RFC3339[6];
- *confidence*: la confidenza calcolata dal sensore al momento dell'aggiornamento;
- *action*: l'azione che il sensore ha rilevato (*walking*, *unknown* o *automotive*);
- *data*: alcuni dati aggiuntivi specifici del sensore che ha effettuato la rilevazione, i quali indicano lo stato in cui si trovava il sensore. Ad esempio per il sensore Wi-Fi questi dati possono essere relativi alla rete a cui il dispositivo è connesso, mentre per il sensore GPS le coordinate geografiche in cui l'utente si trova.

Di seguito un esempio di body inviato dal sensore GPS attraverso una richiesta HTTP:

```
{
    "action":"walking",
    "confidence":0.25436,
    "data": {
        "latitude": 46.70047381187384,
        "longitude": 5.05308332812501,
        "precision": 7.30493215740106
     },
    "datetime":"2024-10-21T15:43:24.525+02:00"
}
```

Come si può notare il campo data è costruito sulla base dello stato del sensore, in particolare, in questo caso, sono presenti le coordinate geografiche in a cui si trova il dispositivo.

In aggiunta a questi dati, nel path della richiesta (figura 3.3) vengono anche mandati il nome del sensore utilizzato ed una chiave univoca usata per identificare la richiesta nel database. Essa è costruita utilizzando la versione del sensore, l'ID dell'utente autenticato nell'applicazione ed il Timestamp del momento in cui viene inviata la richiesta. Inoltre, onde evitare un sovraccarico del server, è stato definito un tempo di *cooldown* tra le richieste: ogni volta che viene notificato un aggiornamento da parte del sensore, se tale tempo non è trascorso dall'ultima richiesta, allora la richiesta non verrà inviata ma viene comunque ricalcolata la nuova media.

Capitolo 4

Integrazione del sensore bluetooth in Generocity

In questo capitolo verrà spiegato come è stato integrato il sensore Bluetooth nel sistema precedentemente descritto. Esso si basa sulla struttura illustrata nel capitolo 2 a cui verranno aggiunti degli algoritmi per rilevare la connessione di automobili e per il calcolo della confidenza. Tutti i file riguardanti il nuovo sensore sono stati aggiunti in un package[1], denominato *it.uniroma1.di.generocity.sensors.bluetooth*, il quale a sua volta è all'interno di *it.uniroma1.di.generocity.sensors* che raccoglie tutto il codice sorgente relativo a tutti i sensori. I suddetti file sono poi stati organizzati a loro volta nei package qui riportati:

- *presentation*, il quale contiene tutte le classi utili alla presentazione dei dati e all'interfaccia utente;
- *receivers*, al cui interno sono definiti tutti i broadcast receiver utilizzati;
- *utils*, che contiene una classe di utilità e delle classi utilizzate per rappresentare i dispositivi Bluetooth sotto forma di oggetti.

Inoltre direttamente in *it.uniroma1.di.generocity.sensors.bluetooth* sono presenti due classi: il Controller, che, pur mantenendo il suo scopo descritto nel capitolo 2, è stato leggermente modificato, e la classe BluetoothSensor, la quale fungerà da interfaccia tra il Controller ed il modulo dei sensori calcolando e aggiornando la confidenza del sensore Bluetooth.

4.1 Le modifiche attuate al Controller

Come per l'applicazione sviluppata precedentemente, il Controller assume un ruolo centrale per il sensore Bluetooth, difatti è qui che verrà mantenuto lo stato del sensore. Allo stesso modo in esso è presente una lista di listener che verranno notificati ad ogni cambiamento di stato. Quest'ultimo è rappresentato dai seguenti attributi:

[1] In Java un package è una sorta di contenitore che ha lo scopo di raggruppare classi, interfacce ed enumerazioni logicamente correlate.

- *isBluetoothEnabled*, un flag che, come nell'applicazione separata, indica se il Bluetooth è acceso o spento;
- *cars*, una mappa che mantiene i dati dei dispositivi Bluetooth delle ultime 10 macchine connesse allo smartphone;
- *connectedDevices*, l'insieme dei dispositivi connessi.

A differenza del primo Controller descritto non sono presenti gli insiemi riguardanti i dispositivi accoppiati e quelli rilevati durante le scansioni, in quanto, come detto in precedenza, l'idea di effettuare la scansione dei dispositivi nelle vicinanze è stata scartata. Per quanto riguarda l'insieme dei dispositivi accoppiati si è scelto di sostituirlo con la mappa cars, così da mantenere in memoria solo i dispositivi connessi precedentemente rilevati come macchine, il tutto per accelerare il processo di rilevamento di un'automobile come verrà descritto nel paragrafo 4.1.1. Questa mappa associa l'indirizzo MAC[2] del dispositivo all'oggetto contenente le sue relative informazioni e consente la memorizzazione di dieci dispositivi al massimo, rimpiazzando quello connesso meno recentemente in caso si provasse ad inserirne un undicesimo. Inoltre la mappa viene salvata nella memoria dello smartphone attraverso le SharedPreferences[3] in modo da renderla persistente in memoria anche dopo la chiusura dell'app.

Infine il Controller, come il suo corrispondente nell'altra applicazione, utilizza i broadcast receiver per ascoltare gli eventi di sistema e aggiornare il suo stato di conseguenza. Esso registra uno StatusChangeReceiver esattamente come descritto nel paragrafo 2.2 in modo da rilevare l'accensione e lo spegnimento del Bluetooth. A differenza di questo receiver, il ConnectionReceiver invece è stato leggermente modificato per permettere di verificare se i dispositivi connessi sono macchine o meno, mantenendo la sua normale funzione di inserire questi dispositivi nell'insieme dei dispositivi connessi e rimuoverli da esso quando si disconnettono. L'ultimo broadcast receiver implementato è l'ImplicitConnectionReceiver anche esso senza modifiche, il Controller però, al momento della sua creazione, non caricherà solamente i dispositivi connessi dal file, ma, per ognuno di essi eseguirà l'algoritmo per verificare se si tratta di un'automobile. I dettagli di questo algoritmo saranno discussi nel prossimo paragrafo. L'unico broadcast receiver non presente nel nuovo controller è il FoundDeviceReceiver in quanto il suo utilizzo è limitato alle scansioni Bluetooth.

4.1.1 L'algoritmo per l'identificazione di automobili

Come detto in precedenza, quando viene connesso un dispositivo viene attuata una procedura per verificare se si tratta del Bluetooth di un'autoradio o di altoparlanti di una macchina. Tale procedura è implementata nel metodo *onDeviceConnection* del ConnectionReceiver che, allo stesso modo dell'app sviluppata precedentemente, si occupa di rilevare le connessioni dei dispositivi Bluetooth. Viene qui riportato il codice del metodo:

[2] L'indirizzo MAC è in codice di 48 bit associato univocamente ad ogni dispositivo di rete.

[3] Un oggetto di tipo SharedPreferences punta ad un file contenente coppie chiave-valore e fornisce dei metodi per scrivere e leggere quest'ultimo.

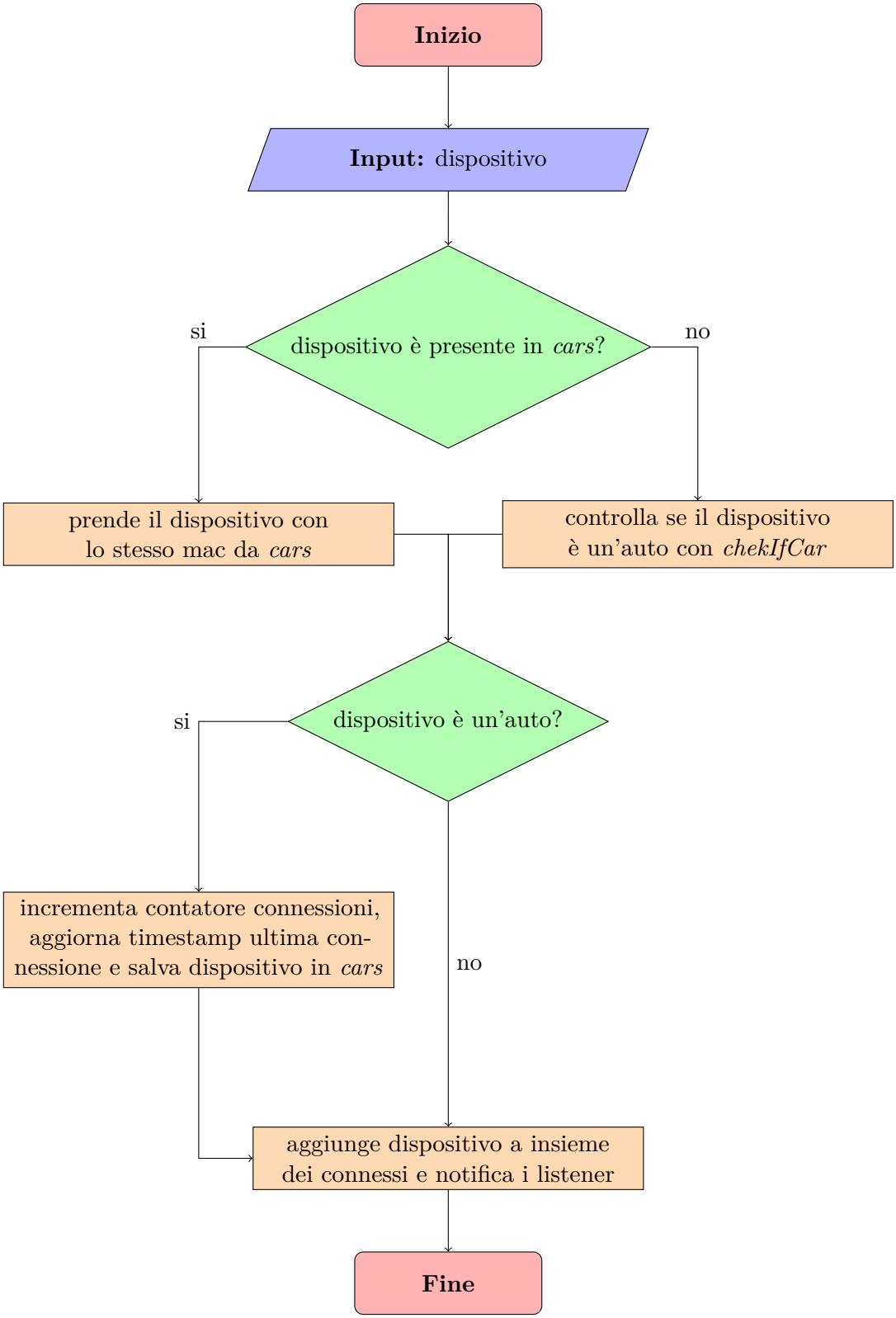
```java
@Override
public void onDeviceConnection(Context context, BluetoothDevice
↪  device) {
    BluetoothDevice car = getCar(device.getMacAddress());
    if (car != null) {
        device = car;
    } else {
        device.checkIfCar();
    }

    if (device.isCar()) {
        device.saveConnection();
        BluetoothController.this.saveCar(context, device);
    }

    connectedDevices.add(device);
    notifyListeners();
}
```

Come prima cosa il metodo cerca di ottenere dalla mappa *cars* il dispositivo connesso e, nel caso esso non sia presente nella struttura dati, verrà utilizzato il metodo *checkIfCar* della classe dei dispositivi Bluetooth, il quale effettua il vero e proprio controllo al fine di determinare se si tratta di un'auto o meno. Nel caso in cui il dispositivo connesso è un'automobile, a prescindere se fosse già salvata o meno, viene chiamato un metodo denominato *saveConnection*, il quale si occupa di aggiornare il numero di connessioni effettuate da quel dispositivo e la data e ora della sua ultima connessione. Successivamente verrà salvata l'auto nella mappa in modo da aggiungere il dispositivo o aggiornare i suoi dati se già presente. Infine verrà aggiunto il dispositivo nell'insieme dei dispositivi connessi e verranno notificati i listener del Controller. Di seguito viene riportato il diagramma di flusso che descrive le istruzioni eseguite al momento della connessione di un nuovo dispositivo.

Figura 4.1 Diagramma di flusso del metodo onDeviceConnection



Come già accennato, il metodo *checkIfCar*, utilizzato quando si connette un dispositivo non presente nella mappa *cars*, si occupa di analizzare le informazioni del dispositivo per inferire se si tratta di un'automobile. Esso andrà a modificare il flag *isCar* presente negli oggetti con cui vengono rappresentati i dispositivi Bluetooth. Per far ciò viene prima controllato se la classe Bluetooth, ossia un codice che identifica la tipologia del dispositivo Bluetooth, è una delle classi di tipo audiovideo e, in caso contrario, l'esecuzione del metodo viene terminata lasciando l'attributo *isCar* del dispositivo a false.

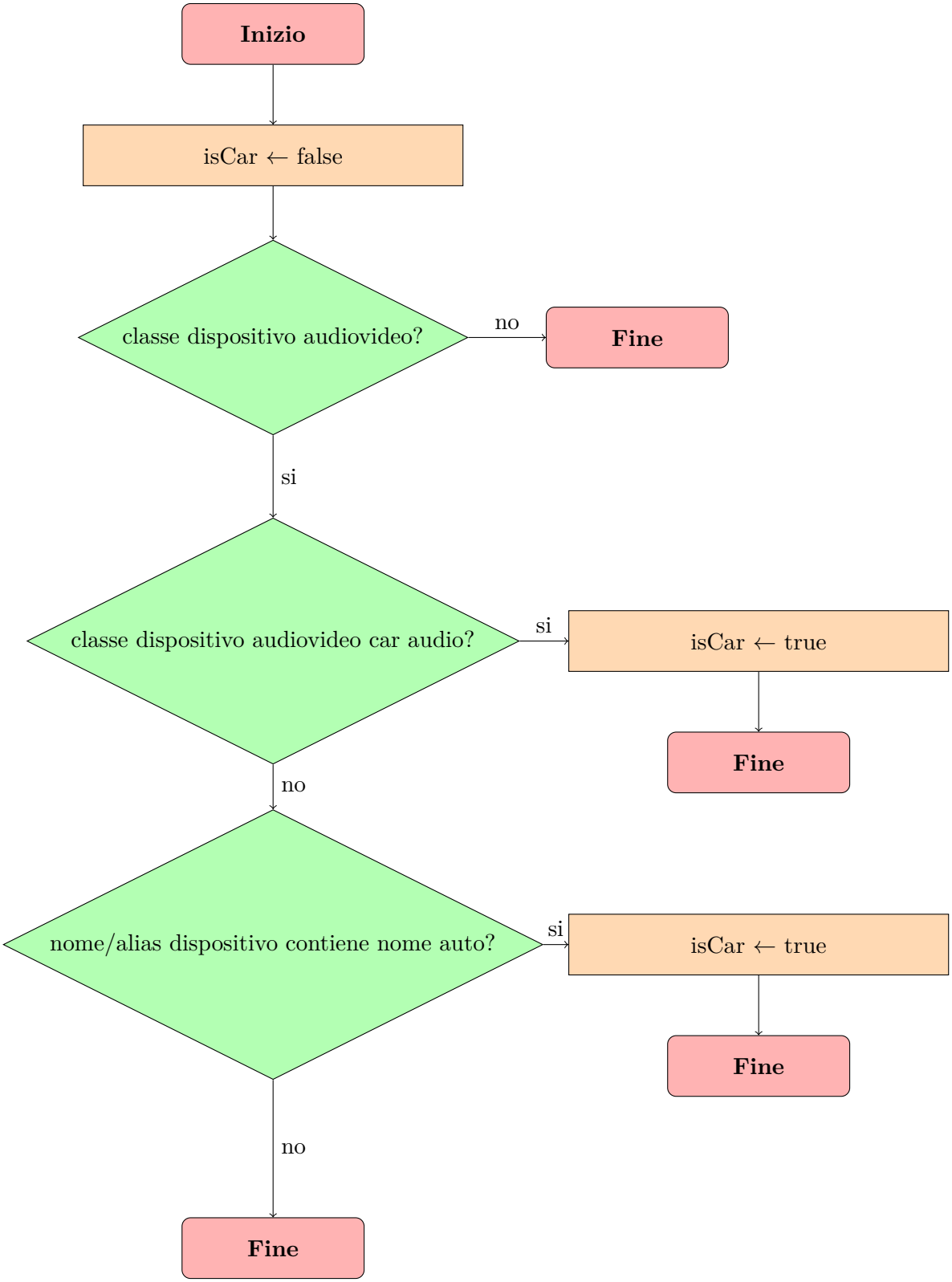
Tutte le autoradio, infatti, utilizzano la categoria audiovideo, in quanto sono dispositivi audio e anche video se presentano uno schermo e, quando il dispositivo connesso è di questo tipo, la sua classe viene controllata più dettagliatamente. Ciò viene fatto per verificare se si tratta di un dispositivo audiovideo car audio, categoria che identifica specificamente i sistemi audiovideo delle automobili.

Purtroppo però si è notato che la maggior parte dei produttori di questi sistemi non utilizza questa classe per categorizzarli, bensì delle classi più generiche che fanno comunque parte della macrocategoria audiovideo, le quali però non sono utilizzate solamente dalle auto, come ad esempio le classi handsfree device oppure loudspeaker. Per ovviare a questo problema, nel caso il dispositivo non sia di tipo car audio, viene controllato il suo nome oppure il suo alias. Nello specifico viene verificato tramite un apposito metodo se in queste stringhe sono contenuti nomi di note marche o modelli di automobili, che sono salvati nell'applicazione. Se un dispositivo ricade in uno di questi casi, ossia utilizza la classe audiovideo car audio oppure il suo nome o il suo alias contengono il nome di una marca o modello di automobile, allora al flag *isCar* del dispositivo viene assegnato il valore true. L'implementazione del metodo è la seguente:

```
1  public void checkIfCar() {
2      isCar = false;
3      if (bluetoothMajorClass !=
       ↪  BluetoothClass.Device.Major.AUDIO_VIDEO) {
4          return;
5      }
6
7      if (bluetoothClass ==
       ↪  BluetoothClass.Device.AUDIO_VIDEO_CAR_AUDIO
8              ||
               ↪  BluetoothDeviceUtils.stringContainCarName(deviceName)
9              || BluetoothDeviceUtils.stringContainCarName(alias)) {
10         isCar = true;
11     }
12 }
```

Di seguito viene invece riportato un diagramma di flusso che descrive il processo decisionale effettuato dal metodo *checkIfCar*.

Figura 4.2 Diagramma di flusso del metodo checkIfCar



4.2 La classe BluetoothSensor

La vera novità rispetto alla precedente app è la classe BluetoothSensor la quale estende la classe Sensor descritta nel paragarafo 3.2.1. Essa infatti è stata definita per interfacciare il Controller con l'intero sistema dei sensori, poter calcolare la confidenza del sensore Bluetooth e innescare il calcolo dello stato dell'utente. La classe BluetoothSensor eredita infatti tutti gli attributi della classe sensor, tra cui il nome, che sarà inizializzato alla stringa "bluetooth", il peso, con il valore di 1.0, e lo storico della confidenza calcolata. Oltre agli attributi vengono ereditati anche tutti i metodi che consentono di notificare l'unità centrale dei sensori quando viene calcolata una nuova confidenza e di inviare lo stato del sensore al server. Uno dei metodi ereditati è il metodo astratto *getStatus*, utilizzato dall'unità centrale per interrogare i sensori circa la loro confidenza in un preciso istante temporale. Inoltre, in quanto astratto, è stato necessario fornire un'implementazione la quale viene qui riportata:

```
1  public double getStatus(Calendar timestamp) {
2      Map.Entry<Long, Double> closestStatus =
       ↪  closestStatusTo(timestamp);
3      return closestStatus == null ? 0.5d : closestStatus.getValue();
4  }
```

Il metodo individua nello storico la confidenza calcolata nel momento più vicino al timestamp ricevuto come input e la restituisce. Esso è infatti utilizzato dall'unità centrale per richiedere a tutti i sensori la loro confidenza, allo scopo di calcolarne la media in uno specifico momento.

Il fulcro della classe è però il metodo *onBluetoothUpdate* il quale viene registrato come listener del Controller nel costruttore: così facendo il BluetoothSensor sarà notificato ogni volta che il Controller effettua un cambiamento nel suo stato e questo metodo sarà eseguito. Nello specifico il metodo si occuperà di calcolare la nuova confidenza, che se diversa dall'ultima volta che è stata calcolata verrà aggiunta allo storico, e di eseguire il metodo *update* al fine di inviare i dati al server e far ricalcolare lo stato dell'utente dall'unità centrale, come illustrato nel paragrafo 3.2.1.

Si riporta la sua implementazione di seguito.

```
1  private void onBluetoothUpdate() {
2      Object extraData = getExtraData();
3
4      double currentConfidence = calculateConfidence();
5      Calendar calendar = Calendar.getInstance();
6
7      if (lastConfidence != currentConfidence) {
8          sensorHistory.put(calendar.getTimeInMillis(),
           ↪  currentConfidence);
9          update(calendar, new SensorData(calendar, currentConfidence,
           ↪  extraData));
10     } else if (collectAllData) {
```

```
11          collect(calendar, new SensorData(calendar,
            ↪  currentConfidence, extraData));
12      }
13      lastConfidence = currentConfidence;
14  }
```

Inoltre, come si può notare nel codice, se il flag *collectAllData* è impostato a true allora i dati del sensore saranno inviati al server tramite il metodo *collect* anche quando la confidenza non è cambiata rispetto all'ultima esecuzione. Nei prossimi paragrafi si spiegherà come viene calcolata la confidenza e successivamente come sono costruiti i dati inviati al server.

4.2.1 Il calcolo della confidenza

La confidenza, come descritto nel capitolo 1, è un valore reale compreso tra 0 e 1 che indica con probabilità crescente che l'utente sta guidando, dove 0 rappresenta con massima sicurezza che l'utente non è alla guida, 0.5 l'incertezza e 1 la certezza che l'utente stia guidando.

Nel sensore Bluetooth il valore della confidenza è calcolato sulla base dello stato del Controller, il quale viene modificato in risposta agli eventi di sistema. Difatti la prima cosa che il metodo *calculateConfidence* effettua è quella di ottenere l'istanza del Controller in modo da verificare sequenzialmente tutti i componenti del suo stato. Come prima cosa viene controllato se il Bluetooth del dispositivo è spento: in tal caso la confidenza restituita sarà pari a 0.5 in quanto il sensore non è in grado di ottenere dati. Se invece il Bluetooth è acceso viene presa in esame la lista dei dispositivi connessi nell'istante in cui viene effettuata la computazione. Nello specifico se essa è vuota verrà restituito il valore 0, dato che sicuramente il dispositivo non è connesso ad un'automobile, non essendo connesso ad alcun dispositivo, altrimenti si controllerà che almeno uno dei dispositivi nella lista sia stato identificato come auto al momento della connessione. Se così non fosse allora la confidenza sarà pari a 0.1 in quanto non si può essere completamente sicuri che uno fra i dispositivi collegati non sia un'automobile. Se invece tra essi è presente almeno un'auto viene ricavato il numero più alto di connessioni effettuate tra i dispositivi e viene restituita una confidenza compresa tra 0.75 e 1 in base al numero di connessioni. Più precisamente, la confidenza sarà pari a 0.75 sommato ad un bonus calcolato nella funzione *computeConnectionScore*.Di seguito si riporta l'implementazione di questo algoritmo.

```
1   private double calculateConfidence() {
2       BluetoothController controller =
        ↪  BluetoothController.getInstance(app);
3
4       // If the bluetooth is disabled, return 0.5 (unknown)
5       if (!controller.isBluetoothEnabled()) {
6           return 0.5d;
7       }
```

```
8
9       // Bluetooth is enabled
10
11      List<BluetoothDevice> connectedDevices =
        ↪  controller.getConnectedDevices();
12
13      // If no device is connected return 0.0 (walking)
14      if (connectedDevices.isEmpty()) {
15          return 0.0d;
16      }
17
18      // There are connected devices
19
20      // If no cars are connected return 0.1 (probably walking, we
        ↪  are not a 100% sure that the device is not a car)
21      if (connectedDevices.stream().noneMatch(BluetoothDevice::isCar))
        ↪  {
22          return 0.1d;
23      }
24
25      // There are cars connected, so get the maximum connection
        ↪  count
26      int maxConnectionCount = connectedDevices.stream()
27              .filter(BluetoothDevice::isCar)
28              .map(BluetoothDevice::getConnectionCount)
29              .reduce(Integer::max)
30              .orElse(0);
31
32      // Return a value between 0.75 and 1 based on the maximum
        ↪  connection count
33      return 0.75d + computeConnectionScore(maxConnectionCount);
34  }
```

La funzione *computeConnectionScore* si occupa di calcolare un punteggio che varia tra 0 e 0.25 sulla base del numero di connessioni effettuate da uno smartphone verso una specifica automobile. In particolare viene restituito un valore che cresce esponenzialmente all'aumentare delle connessioni, questo perché più un'auto viene connessa più è probabile che essa sia utilizzata spesso dall'utente ed è quindi più probabile che quest'ultimo sia alla guida della vettura. Questo bonus viene calcolato attraverso la seguente funzione matematica:

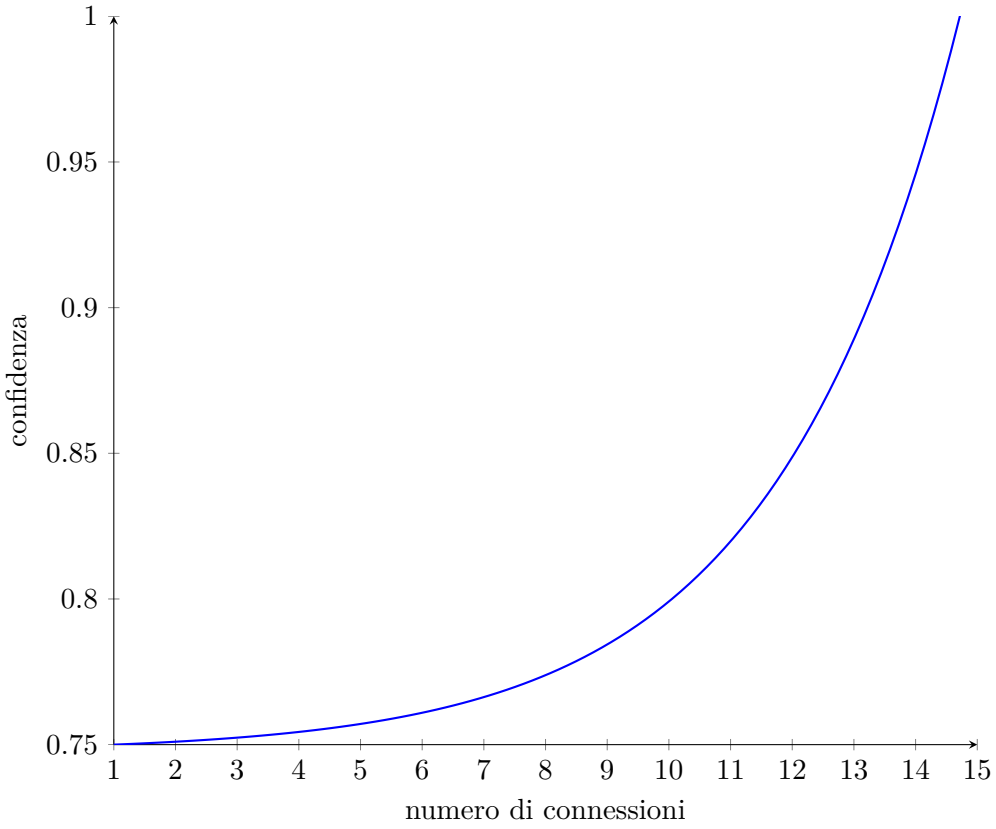
$$connectionScore(connectionCount) = \frac{25}{10000} * (1.4^{connectionCount-1} - 1)$$

Essa è stata scelta poiché ha un andamento che porta la confidenza ad aumentare leggermente a seguito delle prime connessione e ad esplodere esponenzialmente superate le dieci. Il numero di connessioni viene limitato a 15, pertanto superato

questo valore sarà sempre restituito il massimo incremento possibile e di conseguenza ad ogni connessione successiva la confidenza sarà sempre pari a 1.

In figura 4.3 viene tracciata la funzione che determina la confidenza restituita sulla base del numero di connessioni del dispositivo.

Figura 4.3 Incremento della confidenza basato sul numero di connessioni



Il codice qui riportato, invece, stabilisce come viene calcolato il connection score: viene restituito 0 se il numero di connessioni è pari o inferiore a zero e invece restituisce 0.25 quando il numero di connessioni è maggiore o uguale al massimo, ossia 15. Quando invece il numero di connessioni è compreso tra questi due valori viene calcolato l'incremento utilizzando la funzione matematica prima descritta.

```
1 private double computeConnectionScore(int connectionCount) {
2     if (connectionCount <= 0) return 0.0d;  // No connections
      ↪  (should not happen)
3     if (connectionCount >= MAX_CONNECTIONS) return 0.25d;  //
      ↪  Maximum score reached
4     // connectionCount between 1 and MAX_CONNECTIONS => use the
      ↪  function
5     return 0.0025 * (Math.pow(EXPONENTIAL_BASE, connectionCount - 1)
      ↪  - 1);
6 }
```

4.2.2 I dati inviati al server


```
{
   "action":"automotive",
   "confidence":0.75436,
   "data":{
      "connected":[
         {
            "alias":"My Car",
            "bluetooth_class":"Handsfree",
            "connection_count":4,
            "device_name":"Fiat Punto",
            "is_car":true
         }
      ],
      "bluetooth_enabled":true
   },
   "datetime":"2024-10-21T15:43:24.525+02:00"
}
```


Come si può notare dal codice del metodo *onBluetoothUpdate* (paragrafo 4.2), sia *update* che *collect* prendono in input un oggetto di tipo SensorData, il quale ha lo scopo di serializzare i dati relativi allo stato del sensore, come riportato nell'esempio qui sopra, per inviarli al server tramite la web API definita dal backend (paragrafo 3.2.2). Come per tutti i sensori, vengono serializzati la confidenza, l'azione compiuta dall'utente e il timestamp che indica il momento in cui è stato effettuato il calcolo. In aggiunta a ciò, nel campo data il sensore Bluetooth serializza il suo stato, indicando se il Bluetooth dello smartphone è acceso o meno e in caso affermativo riportando la lista dei dispositivi ad esso connessi. Inoltre per ogni dispositivo viene serializzato il nome, l'alias, la classe Bluetooth, il numero di connessioni effettuate con il dispositivo e un campo denominato *is_car* che indica se si tratti di un'automobile o meno.

Figura 4.4 Interfaccia utente del sensore Bluetooth

4.3 La presentazione dei dati

Come per la prima applicazione sviluppata è stata creata un'interfaccia utente (visibile in figura 4.4) accessibile dal menu laterale di GeneroCity. Essa consente di visualizzare le auto salvate nella

mappa *cars* e i dispositivi connessi, il tutto in un unica lista ordinata in modo da avere i dispositivi connessi in alto e successivamente tutti gli altri ordinati in base al numero di connessioni effettuate. Di conseguenza, le automobili con più connessioni appariranno più in alto nella lista. Inoltre per ogni dispositivo è stata aggiunta un'icona che aiuta l'utente a distinguere se si tratta di un'auto o meno. Tutto ciò è stato implementato utilizzando un *BluetoothDeviceAdapter* in maniera simile a quanto descritto nel paragrafo 2.3. È stato quindi definito il metodo onDeviceUpdate in modo da fondere la lista dei dispositivi connessi e quella delle automobili, entrambe ottenute dal Controller, in un unica lista ordinata come descritto in precedenza. Questo metodo viene quindi registrato come listener del Controller in modo che, ogni volta che esso cambia stato, venga aggiornata l'interfaccia.

La definizione del BluetoothDeviceAdapter viene riportata qui sotto.

```
1  BluetoothDeviceAdapter bluetoothDeviceAdapter = new
   ↪  BluetoothDeviceAdapter(this) {
2      @Override
3      public void updateDevices() {
4          BluetoothController controller =
           ↪  BluetoothController.getInstance(
5              BluetoothSensorActivity.this
6              );
7
8          Comparator<BluetoothDevice> deviceComparator =
           ↪  Comparator.comparing(
9                          (BluetoothDevice bluetoothDevice) ->
                           ↪  bluetoothDevice.isConnected(context),
10                         Comparator.reverseOrder()
11                 )
12                 .thenComparing(BluetoothDevice::getConnectionCount,
                   ↪  Comparator.reverseOrder())
13                 .thenComparing(BluetoothDevice::getLastConnection,
                   ↪  Comparator.reverseOrder());
14
15         devices = Stream.concat(controller.getCars().stream(),
           ↪  controller.getConnectedDevices().stream())  // merge
           ↪  cars and connected devices
16                 .distinct()  // remove duplicates
17                 .sorted(deviceComparator)  // sort by connection
                   ↪  status, connection count and last connection
18                 .collect(Collectors.toList());
19
20         notifyDataSetChanged();
21     }
22 };
```

4.4 Il testing del sensore

Sia durante lo sviluppo del sensore sia successivamente sono state effettuate delle prove con smartphone differenti e con alcuni modelli di macchine provviste di autoradio con Bluetooth, ciò allo scopo di verificare il corretto funzionamento del sensore Bluetooth. Le prove sono state effettuate tutte in sicurezza, poiché per questo sensore è necessario solamente instaurare una connessione tra l'auto e uno smartphone, senza la necessità di mettersi effettivamente alla guida.

4.4.1 L'identificazione di automobili

I primi test che sono stati effettuati sono serviti a verificare il corretto funzionamento dell'algoritmo per identificazione di diversi tipi di dispositivi tramite le informazioni ricavate dal sensore Bluetooth. In particolare è stato importante controllare che le autovetture venissero rilevate come tali, indipendentemente se attraverso la classe Bluetooth o il nome del dispositivo.

Per effettuare questi test sono stati utilizzati diversi dispositivi allo scopo di verificarne il funzionamento su versioni di Android e hardware differenti. Gli smartphone utilizzati sono i seguenti: Xiaomi mi 9 Lite, OnePlus 6 e Xiaomi Note 11 Pro. Questi non hanno presentato differenze sostanziali durante la fase di testing. Inoltre, è stato possibile avere un riscontro dei risultati dei test attraverso l'interfaccia utente descritta precedentemente, visualizzando i log forniti dall'applicazione e analizzando i dati che sono stati inseriti nel database attraverso l'utilizzo della web API.

Prima di tutto sono stati effettuati test su diverse automobili con sistema Bluetooth integrato, per verificare che fosse indicata correttamente la classe Bluetooth relativa alle automobili (`BluetoothClass.Device.AUDIO_VIDEO_CAR_AUDIO`). Dalle numerose vetture testate è emerso che, come precedentemente accennato, la classe del dispositivo ha sempre un identificativo più generico, ad esempio `handsfree` oppure `loudspeaker`. Per questo motivo si è deciso di implementare la rilevazione tramite nome o alias del dispositivo, dato che, anche se la classe è risultata generica, il nome dei sistemi integrati nelle auto ha sempre presentato informazioni sull'auto come il nome del modello o del produttore. Da qui si è presa la decisione di creare un piccolo database contenente il nome dei maggiori produttori di automobili, in modo da consentire l'identificazione tramite esso. Ciò permette anche di identificare dispositivi non integrati all'interno dell'auto, come casse Bluetooth o autoradio di terze parti, dato che essendo pensati per automobili presentano un alias coerente ("car", "auto", ecc.).

Dopo aver implementato le modifiche per effettuare l'identificazione tramite il nome, si sono eseguiti nuovamente dei test e si è osservato che il sensore è in grado di determinare automaticamente quando l'utente si connette ad un automobile.

4.4.2 Verifica del valore di confidenza

L'aspetto che si è voluto testare successivamente è l'aumento della confidenza coerentemente al numero di connessioni effettuate verso la stessa auto, allo scopo di rilevare con maggiore probabilità lo stato guida dell'utente. Infatti è importante che

l'applicazione sia in grado di riconoscere quando l'utente si trova effettivamente alla guida e non sia un passeggero. Chiaramente il sensore Bluetooth da solo non può discernere questi due casi, quindi si è scelto di basarsi sul numero di connessioni per inferire che, quando il dispositivo è connesso ad un auto utilizzata frequentemente, l'utente stia guidando.

Anzitutto si è testato un approccio lineare, ovvero aumentando la confidenza di un valore costante ogni qualvolta una nuova connessione con la stessa vettura fosse stata rilevata. Dai test è emerso che diverse situazioni, come l'utilizzo di servizi di car sharing o di uso sporadico di un automobile come passeggero, la confidenza restituita fosse troppo elevata portando a falsi positivi. Si è quindi deciso di utilizzare una funzione esponenziale in modo da incrementare leggermente la confidenza per le prime connessioni, mentre per un numero più elevato restituire una confidenza più alta. Questo approccio si è rivelato più affidabile nei successivi test.

Capitolo 5

Conclusione

Il lavoro descritto in questa tesi è consistito nello sviluppo e integrazione di un sensore Bluetooth all'interno dell'applicazione GeneroCity, con l'obiettivo di rilevare in modo affidabile quando un utente è alla guida di un veicolo. Questo modulo software, congiuntamente agli altri sensori, contribuirà a rendere l'esperienza degli utenti più sicura e comoda, riducendo la necessità di interazioni manuali durante la guida, facilitando così lo scambio di parcheggi con altri utenti che utilizzano l'applicazione. L'implementazione del sensore ha richiesto un'analisi approfondita delle API Android e la creazione di una strategia per riconoscere la connessione degli smartphone con delle automobili. Infine il sensore sviluppato ha dimostrato di essere efficace nei test effettuati, rilevando correttamente la connessione a dispositivi compatibili con il contesto di guida, come le autoradio.

5.1 Sviluppi futuri

Per migliorare ulteriormente il sistema, sono attualmente in sviluppo presso il Gamification Lab modelli di machine learning per il calcolo dello stato dell'utente, così da sostituire in futuro la media pesata della confidenza dei vari sensori con algoritmi più sofisticati. Il sensore da me sviluppato, oltre agli scopi descritti precedentemente, verrà utilizzato insieme agli altri sensori anche per raccogliere dati utili all'allenamento di questi modelli. Inoltre, sarebbe utile implementare un servizio esterno per verificare se il nome del dispositivo Bluetooth corrisponda a modelli di veicoli conosciuti, incrementando così l'accuratezza del riconoscimento e adattandosi a veicoli nuovi. Questo servizio potrebbe essere implementato ad esempio attraverso un nuovo endpoint esposto dalla web API di GeneroCity. In questa maniera sia l'applicazione Android sia quella iOS potrebbero farne uso, in modo da effettuare in modo centralizzato questa computazione.

L'utilizzo del sistema di rilevamento basato su sensori intelligenti come quello descritto in questa tesi per ottenere interazioni implicite rappresenta una direzione promettente per lo sviluppo di GeneroCity e più in generale nel campo della mobilità urbana.

Bibliografia

[1] Android bluetooth connectivity guide and documentation. https://developer.android.com/develop/connectivity/bluetooth.

[2] Broadcasts overview. https://developer.android.com/develop/background-work/background-tasks/broadcasts.

[3] Generocity, 2021. https://www.generocity.it/.

[4] Che cos'è lo smart parking: le soluzioni e le tecnologie per il parcheggio intelligente. https://www.economyup.it/mobilita/che-cose-lo-smart-parking-le-soluzioni-e-le-tecnologie-per-il-parcheggio-intelligente/, 2020.

[5] What is implicit interaction. https://www.igi-global.com/dictionary/implicit-interaction/13953, 2011.

[6] C. Newman G. Klyne. Date and time on the internet: Timestamps. Technical report, Clearswift Corporation, Sun Microsystems, 2002. https://www.rfc-editor.org/rfc/rfc3339.

[7] Albrecht Schmidt. Implicit human computer interaction through context. *Personal technologies*, 4:191–199, 2000.