Projet Trottinette Électrique Connectée

Revue Finale

Trottinette Électrique Tout Terrain(TTE) SXT 1000 XL

Version 1.0

HACHETTE Alexandre (IR)

Introduction

Expression du besoin

L'exploitant veut développer un système embarqué sur une trottinette électrique équipée de capteurs afin d'assister l'utilisateur sur son trajet. Un accessoire est monté sur le guidon pour permettre d'y poser un terminal mobile et d'accéder en temps réel aux données de la trottinette qui sont transmises par la trottinette.

Présentation générale du projet

Nous devrons donc :

- Acquérir les données de fonctionnement de la trottinette
- Transmettre les données de fonctionnement de la trottinette via une communication sans fil
- Visualiser les données de fonctionnement reçues de la trottinette, la durée d'utilisation et l'autonomie sur le terminal mobile
- Géolocaliser la trottinette et la visualiser sur une carte de l'écran du terminal mobile
- Enregistrer les données de fonctionnement de la trottinette sur une carte SD (en option)
- La régulation de la vitesse et/ou l'arrêt de la trottinette (en option)
- Protection contre le vol (en option)

Les données de la trottinette seront transmises grâce à une communication Bluetooth entre le terminal mobile et la trottinette.

Les tâches à réaliser seront réparties sur 2 étudiants :

- Étudiant EC :
  HILLION Alexis devra acquérir des capteurs, établir un protocole de communication avec le terminal mobile, transmettre les données de fonctionnement vers le mobile.

- **Étudiant IR :**
  HACHETTE Alexandre devra établir le protocole de communication avec la TTE[1], réceptionner les données de fonctionnement de la TTE, visualiser les données de fonctionnement de la TTE et la durée d'utilisation sur l'écran du terminal mobile, calculer et afficher l'autonomie pour un parcours, afficher et actualiser la carte avec la géolocalisation de la TTE.

[1] TTE : Trottinette tout Terrain Électrique

Contraintes matérielles et logicielles

Les ressources matérielles mises à disposition pour mener à bien le projet sont :

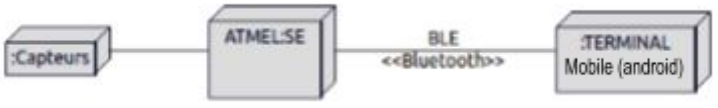
Désignation	*Caractéristiques techniques*	*Acquisition*	*Existant*
SXT 1000 XL	Trottinette Electrique Tout Terrain (TTE) Plomb 48V 12Ah de la marque SXT		✖
SE ATMEL	Carte de développement ATMEL (ou équivalente)		✖
TERMINAL	Terminal mobile sous Android		✖
BLE	Module *Bluetooth Low Energy*		✖
CAPTEURS	Ensemble de capteurs à définir	✖	
SD	Carte (*Secure Digital*) carte mémoire amovible de stockage de données numériques minimum 1GO (en option)		✖

Ainsi que les logiciels sont :

Environnement de développement	© ATMEL Studio
Systeme d'exploitation du terminal mobile	Android
Système de gestion de bases de donnée relationnelles	SQLite3
Gestion et administration de bases de donnees	Sqliteman ou SQLiteManager
Atelier de génie logiciel	Bouml 7.4
Logiciel de gestion de versions	subversion
Generateurs de documentation	Doxygen version 1.8.11
Gestionnaire de projet	Planner (version 0.14.5)

Diagramme de déploiement

Le SE (Système Embarqué) est construit autour d'une carte Atmel sur laquelle est reliée les différents capteurs pour l'acquisition des données de fonctionnement (partie EC).



Le terminal mobile (partie IR) fonctionne sous Android et communique avec le SE par une communication Bluetooth.

Cela implique qu'une communication est nécessaire et qu'il nous faut un protocole de communication entre la trottinette et notre terminal mobile.

Présentation d'Android

Android est un système d'exploitation Open Source de la société Google pour terminal mobile (téléphone, tablette, …).

Pour développer des applications pour ce système, il est nécessaire de disposer du **SDK** (*Software Development Kit*) Android et d'un environnement de développement. Pour le projet, nous utiliserons **Android Studio**. Le langage de programmation utilisé est **Java**.

Activité

Une **activité** est la composante principale d'une application Android. Elle représente à la fois le code et les interactions avec l'interface graphique.

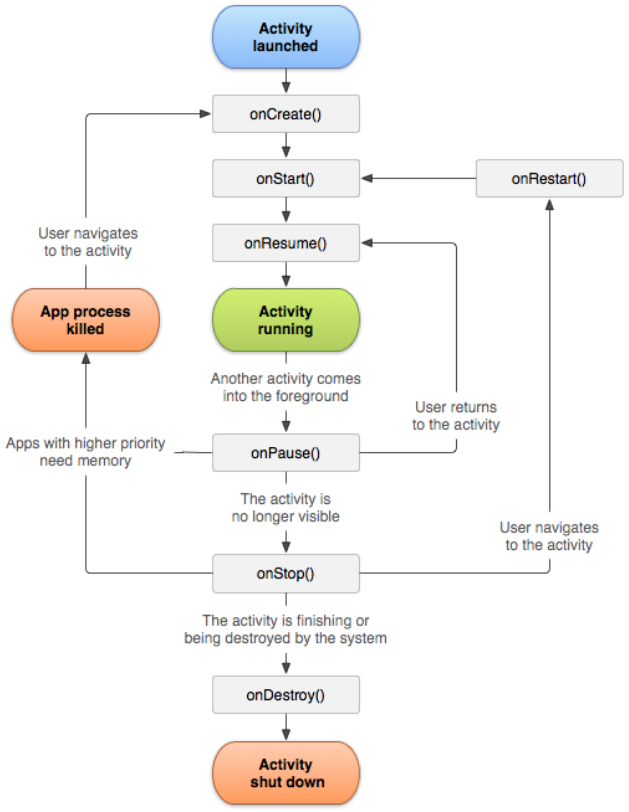
On appellera donc les différentes fenêtres d'une application une **activité**. Une activité prend tout l'écran et il ne peut donc y en avoir qu'une à la fois. Dans notre programme, nous ne disposons que d'une seule activité : **MainActivity**.

Une activité est associée à un fichier ressource XML qui fournit l'aspect graphique de l'interface sous forme de *View*. Une activité permet aussi de contenir des information sur l'état actuel de l'application, ces information s'appellent le *context*.

Pour créer une activité principale, il suffit de déclarer une nouvelle classe héritant d'une classe mère **AppCompatActivity** (ou **Activity**) :

```
public class MainActivity extends AppCompatActivity
{
      // ...
}
```

Le point d'entrée d'une application Android est la méthode **onCreate()** (équivalent du main() dans d'autres environnements) d'une activité. Une application Android a un cycle de vie décrit dans le schéma suivant :



Notre activité passe donc par une phase onCreate sans faire de onStart car nous n'utilisons pas de système de onPause et onStop, elle continue sur un onResume a ce moment l'activité est fonctionnel l'utilisateur peut cliquer sur les boutons et sélection des objet dans les liste qui lui sont dédiés l'application se ferme après un onDestroy.

Resources XML

En Android des layouts en XML (*Extensible Markup Language*) nous permettent de gérer le côté graphique de notre application. Pour déclarer des ressources, on passe très souvent par le format XML.

Le XML est un langage de balise simple qui ressemble au HTML. Les langage de balise s'oppose au langage de programmation tel que le java ou le C++ car il ne donne pas d'ordre à l'ordinateur pour effectuer des calculs mais ont juste pour objectif de mettre en forme l'information de façons à ce que l'on puisse la lire. Pour mettre en forme notre information on utilise des balises. Par exemple:

En android 5 types de ressources sont utilisées :

- **Drawable** qui contient toutes les images et fichiers de dessins.
- **Layout** qui contient nos interfaces graphique et nos différente vues.
- **Menu** qui contient toutes les déclarations d'éléments pour confectionner des menus.
- **Raw** qui contient tout les ressources de format.
- **Values** qui contient les valeur pour les chaînes de caractère, les dimension ou encore les couleur …

La classe R

La classe R est une classe nécessaire pour accéder a nos ressources XML depuis notre code en java. Elle est situe dans le fichier R.java .

Cette classe n'aura pas besoin d'être instancié comme nous allons le voir après.

La classe layout

La classe layout est un classe dite **interne** car elle a été déclaré dans une autre classe.
Pour y accéder, il faut donc faire référence à la classe qui la contient de ce cas si la classe R.
la classe layout est de type public static final et a le nom layout.

- Public car elle doit être accessible par tout le monde sans restriction.
- Static, car elle ne dépend pas de l'instanciation de la classe R (nous ne somme pas obligé de créer un objet de type R)
- Final signifie que l'on ne peut pas avoir de classe dérivé de layout.

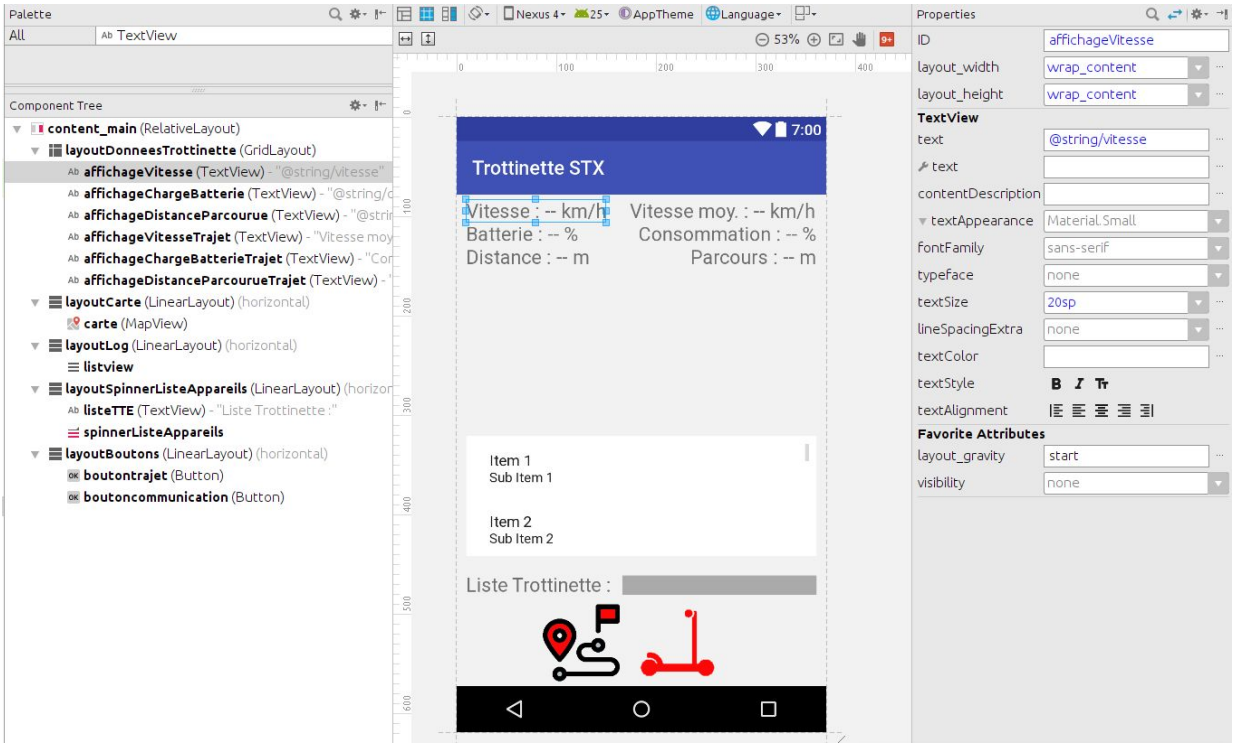
Le fichier **content_main_ihm.xml** décrit la structure de l'IHM principale en définissant les ressources graphiques :

Dans la ressource content_main_ihm.xml

```
[...]
   <GridLayout
       android:id="@+id/layoutDonneesTrottinette"
       android:layout_width="match_parent"
       android:layout_height="wrap_content"
       android:layout_marginBottom="5sp"
       android:layout_marginLeft="10dp"
       android:layout_marginRight="10dp"
       android:layout_marginTop="5sp"
       android:columnCount="2"
       android:rowCount="3">

       <TextView
           android:id="@+id/affichageVitesse"
           android:layout_width="wrap_content"
           android:layout_height="wrap_content"
           android:layout_column="0"
           android:layout_gravity="start"
           android:layout_marginRight="20dp"
           android:layout_row="0"
           android:text="Vitesse : -- km/h"
           android:textSize="20sp"/>
       [...]
   </GridLayout>
[...]
```

Ici, le **TextView** est placé dans un **GridLayout** qui permet un positionnement en grille. Le TextView permet l'affichage d'un texte dans l'IHM, ici l'id **affichageVitesse** affichera la vitesse de la trottinette en km/h :



Lien entre XML et Activité

Pour faire le lien entre le layout et le code en java on utilise la ligne suivant dans le fichier XML :

```
tools:context="com.example.iris.myapplication.MainActivity"
```

Ensuite dans notre code Java on pourra utiliser les ressources que nous avons placées dans le layout à l'aide de leur **id**.

```
[...]
private TextView affichageVitesse;
[...]
affichageVitesse = (TextView) findViewById(R.id.affichageVitesse);
[...]
affichageVitesse.setText("Vitesse : " + trame.getVitesse() + " km/h");
[...]
```

Nous pouvons grâce à ce morceau de code **modifier le texte** que contient notre label en ajoutant la valeur que nous recevons lors de la transmission des données par notre trottinette.

Thread

Un *thread* est un fil d'exécution dans un programme. Il permet d'exécuter du code en parallèle.

Le thread UI (*User Interface*) est le fil d'exécution d'une activité. Il est responsable de l'affichage et des interactions avec l'utilisateur et surtout c'est le seul thread qui doit modifier l'affichage.

Par contre, on ne peut pas effectuer des traitements consommateurs en temps dans le thread UI car celui-ci se "figerait" et ne répondrait plus aux actions de l'utilisateur. Si une activité réagit en plus de cinq secondes, elle sera tuée par l'ActivityManager d'Android qui la considérera comme morte.

Il faudra donc créer et exécuter des threads d'arrière-plan pour les traitements lourds. Nous utilisons dans notre programme un thread notamment pour la réception de trame et pour la connexion et déconnexion des modules bluetooth.

En Java, il y a plusieurs façons de créer et exécuter un thread. Le principe de base revient à dériver (*extends*) une classe **Thread** ou à implémenter (*implements*) l'interface **Runnable** et à écrire le code du thread dans la méthode run(). Ensuite, on appellera la méthode start() pour démarrer le thread et la méthode stop() pour l'arrêter.

Le thread UI (*User Interface*) de l'activité principale est le seul thread qui peut modifier l'affichage. Il faudra donc mettre en place une communication en threads pour interagir avec l'IHM. On utilisera les services de la classe **Handler**.

Handler

Lorsqu'un thread doit interagir avec l'IHM, on utilisera la classe **Handler** pour communiquer avec le thread UI qui a la responsabilité des interactions avec l'IHM.

Le Handler est associé à l'activité principale (qui le déclare) et travaille au sein du thread UI. Le handler se chargera de mettre à jour l'IHM. Le thread peut communiquer avec cet handler au moyen de messages :

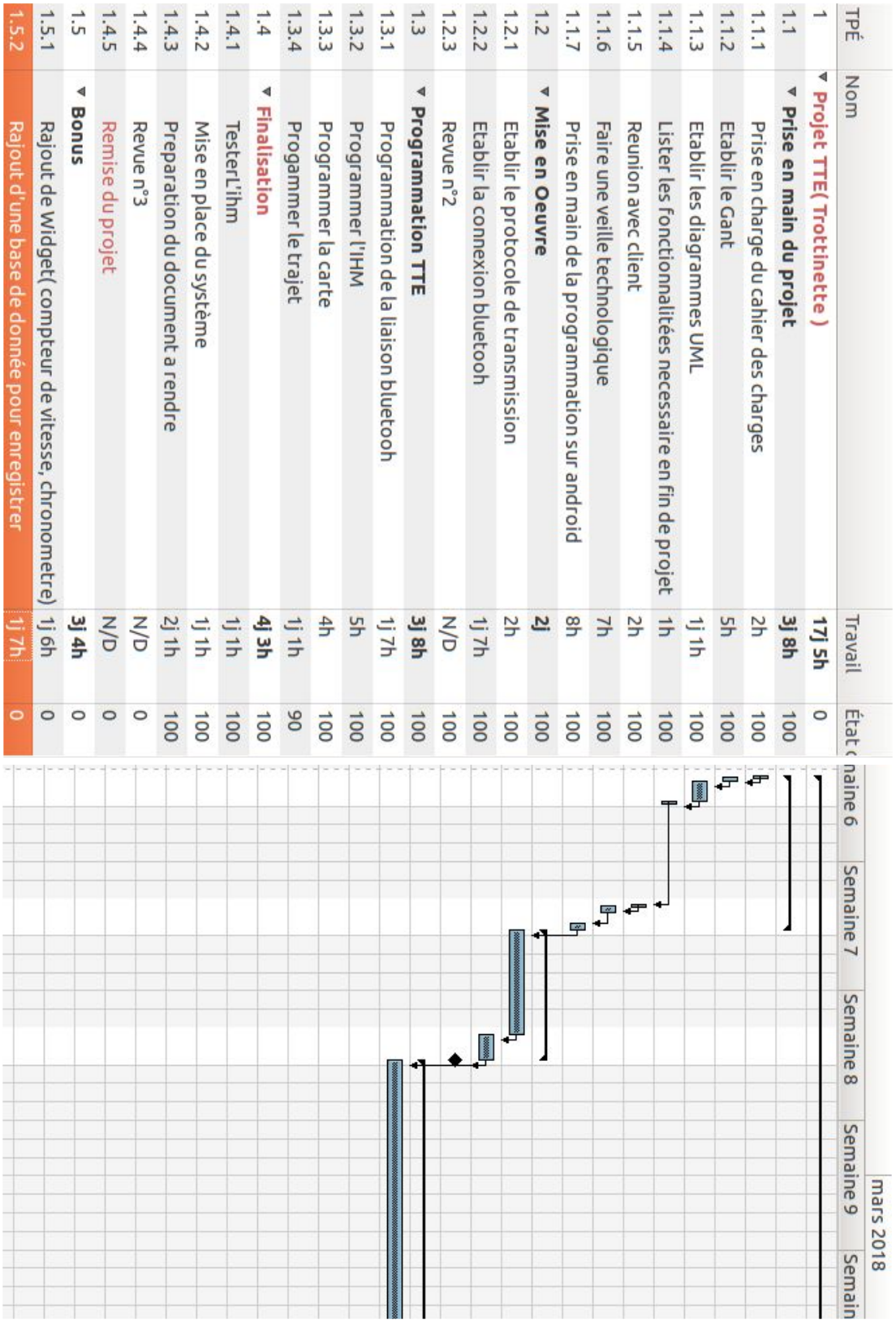
- le thread récupère un objet **Message** du Handler par la méthode Message.obtain(). Il peut ensuite ajouter des données en utilisant un objet **Bundle** ;
- le thread envoie le message au Handler en utilisant la méthode **sendMessage()** ;
- La présence d'un message déclenchera l'exécution de la méthode **handleMessage()** du Handler qui lui permettra de traiter le message et les données contenues.

Dans le thread TReception	Dans l'activité MainActivity

`Handler gestionnaire;`
`…`
 	
`Message message = Message.obtain();`
`message.what = PeripheriqueBluetooth.CODE_CONNEXION;`
`Bundle paquet = new Bundle();`
`paquet.putString("nom",getNom());`
`paquet.putString("adresse",getAdresse());`
`paquet.putInt("etat",CODE_CONNEXION);`
`paquet.putString("donnees","");`
`message.setData(paquet);`
 	
`gestionnaire.sendMessage(message);`
	`final private Handler gestionnaire = new Handler()`
	`{`
	`  public void handleMessage(Message msg)`
	`{`
	`    Bundle b = msg.getData();`
	
	`    switch(b.getInt("etat"))`
	`    {`
	`      case PeripheriqueBluetooth.CODE_CONNEXION:`
	`        // ...`
	`        break;`
	`      case PeripheriqueBluetooth.CODE_RECEPTION:`
	`        // ...`
	`        break;`
	`      case PeripheriqueBluetooth.CODE_DECONNEXION:`
	`        // ...`
	`        break;`
	`      default:`
	`    }`
	`  }`
	`};`

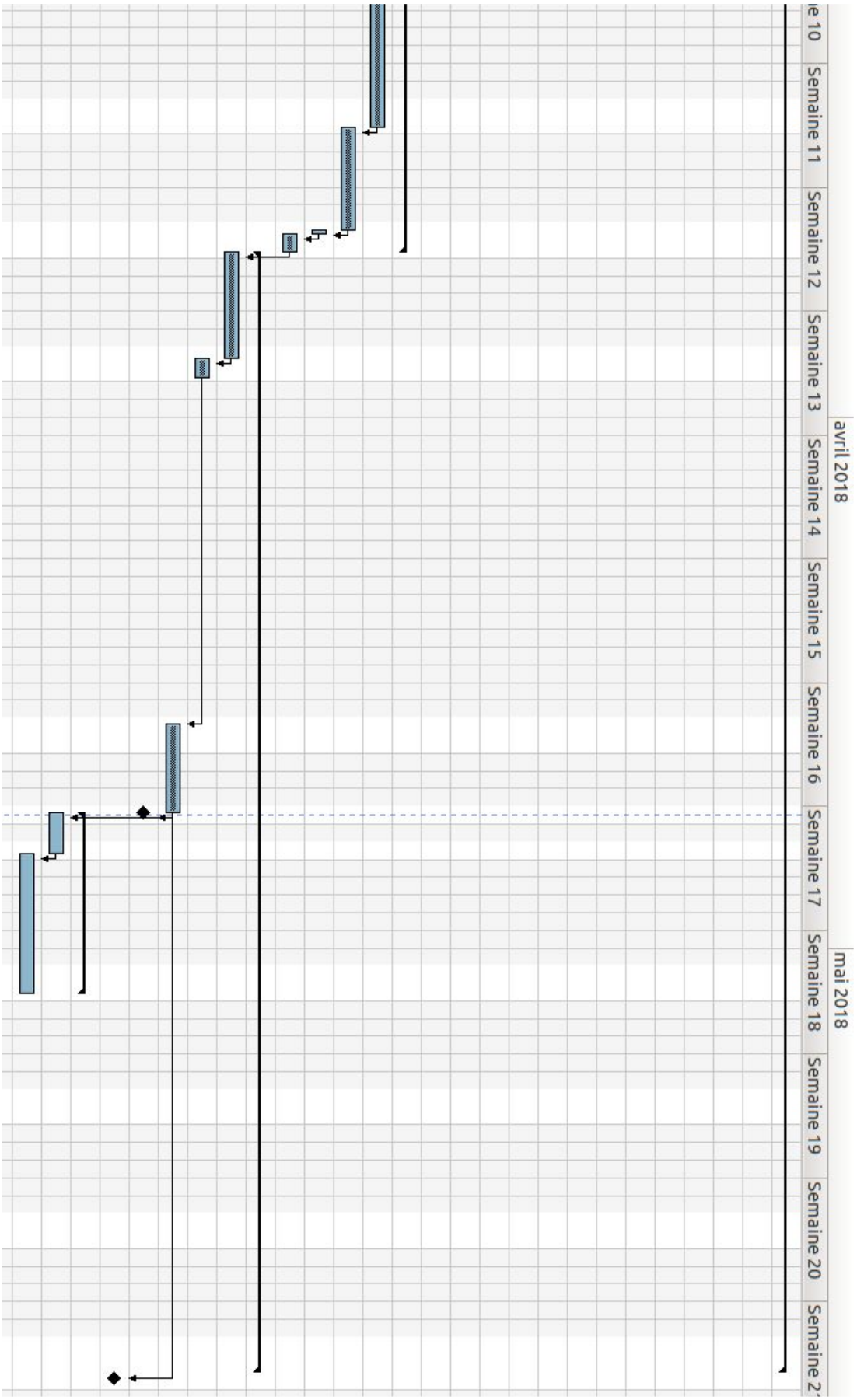
Planification prévisionnelle et tâches à réaliser

Pour organiser notre travail nous avons donc eu recours à un diagramme de Gantt[2] :



TPÉ	Nom	Travail	État
1	**Projet TTE(Trottinette)**	**17j 5h**	0
1.1	**Prise en main du projet**	**3j 8h**	100
1.1.1	Prise en charge du cahier des charges	2h	100
1.1.2	Etablir le Gant	5h	100
1.1.3	Etablir les diagrammes UML	1j 1h	100
1.1.4	Lister les fonctionnalitées necessaire en fin de projet	1h	100
1.1.5	Reunion avec client	2h	100
1.1.6	Faire une veille technologique	7h	100
1.1.7	Prise en main de la programmation sur android	8h	100
1.2	**Mise en Oeuvre**	**2j**	100
1.2.1	Etablir le protocole de transmission	2h	100
1.2.2	Etablir la connexion bluetooh	1j 7h	100
1.2.3	Revue n°2	N/D	100
1.3	**Programmation TTE**	**3j 8h**	100
1.3.1	Programmation de la liaison bluetooh	1j 7h	100
1.3.2	Programmer l'IHM	5h	100
1.3.3	Programmer la carte	4h	100
1.3.4	Progammer le trajet	1j 1h	90
1.4	**Finalisation**	**4j 3h**	100
1.4.1	Tester L'ihm	1j 1h	100
1.4.2	Mise en place du système	1j 1h	100
1.4.3	Preparation du document a rendre	2j 1h	100
1.4.4	Revue n°3	N/D	0
1.4.5	Remise du projet	N/D	0
1.5	**Bonus**	**3j 4h**	0
1.5.1	Rajout de Widget(compteur de vitesse, chronometre)	1j 6h	0
1.5.2	Rajout d'une base de donnée pour enregistrer	1j 7h	0

[2] Diagramme de Gantt : c'est un outil permettant de visualiser dans le temps les diverses tâches composant un projet.


Semaine 11
Semaine 12
Semaine 13
Semaine 14
Semaine 15
Semaine 16
Semaine 17
Semaine 18
Semaine 19
Semaine 20
avril 2018
mai 2018

Dans ce diagramme de gantt on peut observer que nous avons divisé notre travail en 5 itérations :

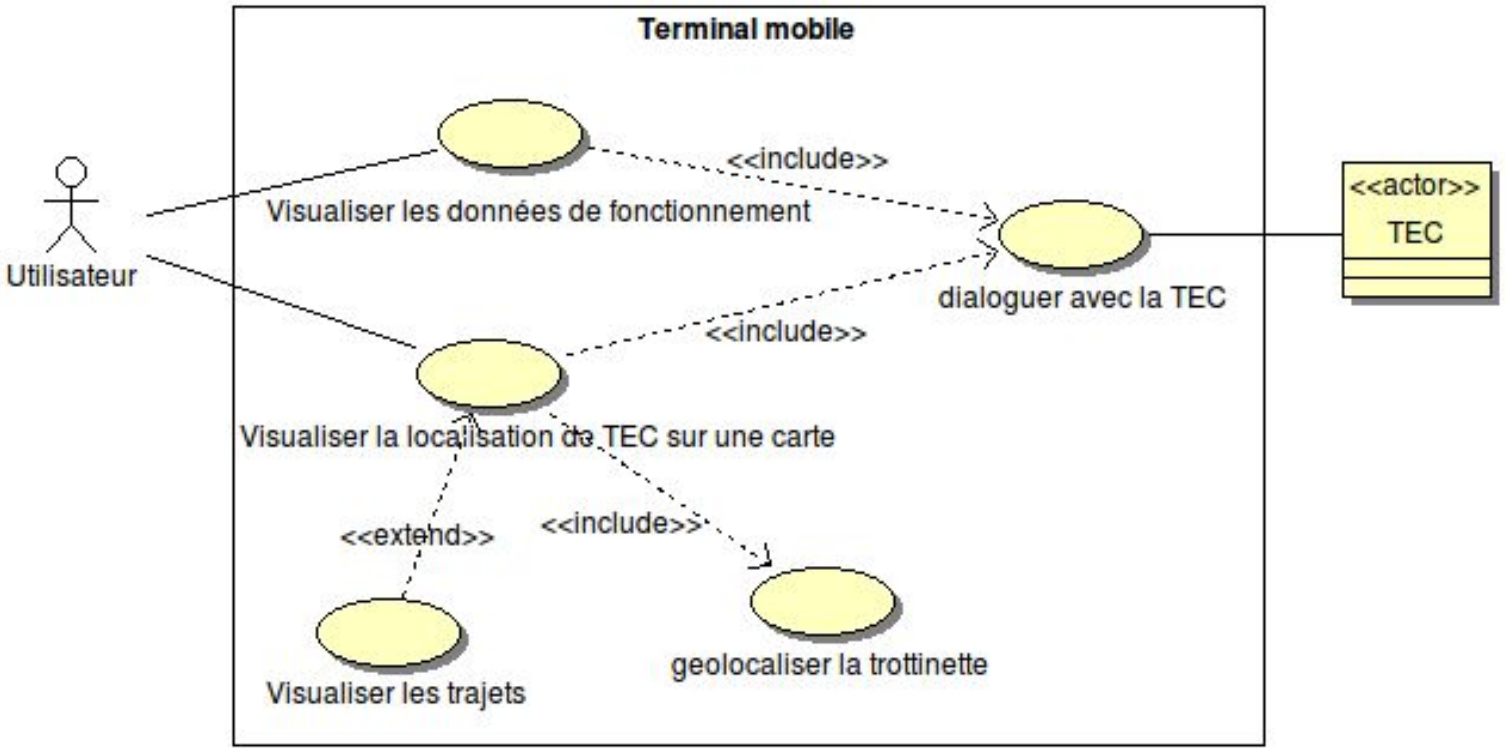
- Prise en main du projet
- Mise en oeuvre
- Programmation TTE
- Finalisation
- Bonus

Avec une première itération d'analyse pour mieux connaître le projet et analyser le temp de travail à fournir sur chaque fonctionnalité de notre application. Ensuite nous avons effectué une étape importante pour établir le protocol de communication et la connexion bluetooth qui devait nous emmener a la 2eme revue de projet. Nous pouvions alors attaquer la face de programmation pour établir la connexion bluetooth.

Les tâches à réaliser par l'étudiant HACHETTE Alexandre sont :

- Mettre en oeuvre un protocole de communication avec la TTE
- Réceptionner les données de fonctionnement de la TTE
- Visualiser les données de fonctionnement de la TTE et la durée d'utilisation
- Calculer l'autonomie pour un parcours et l'afficher
- Afficher et actualiser la carte avec la géolocalisation de la TTE

Diagramme de cas d'utilisation



Une communication Bluetooth entre la trottinette et le terminal mobile est nécessaire pour pouvoir visualiser les données de fonctionnement de la TEC[3]. La visualisation de sa localisation nécessite une géolocalisation par le terminal mobile. L'utilisateur pourra éventuellement visualiser son trajet.

A partir du terminal mobile, grâce à l'application, l'utilisateur pourra donc :

- Visualiser les données de fonctionnement (vitesse en km/h, charge de la batterie en % et la distance parcourue en mètres)

- Visualiser la position de sa trottinette sur une carte

- Visualiser son trajet

L'utilisateur pourra lorsqu'il active le mode trajet visualiser les données et le tracé de son trajet depuis le début de l'activation du bouton. Il ne pourra voir que le tracé qu'il a effectué depuis que le bouton est activé.

[3] TEC : Trottinette Électrique Connectée (la TTE avec le système développé)

Étude préliminaire

IHM

Affiche les données du parcours depuis l'activation du bouton trajet

Affiche les données de la trottinette lors de la connexion avec celle-ci

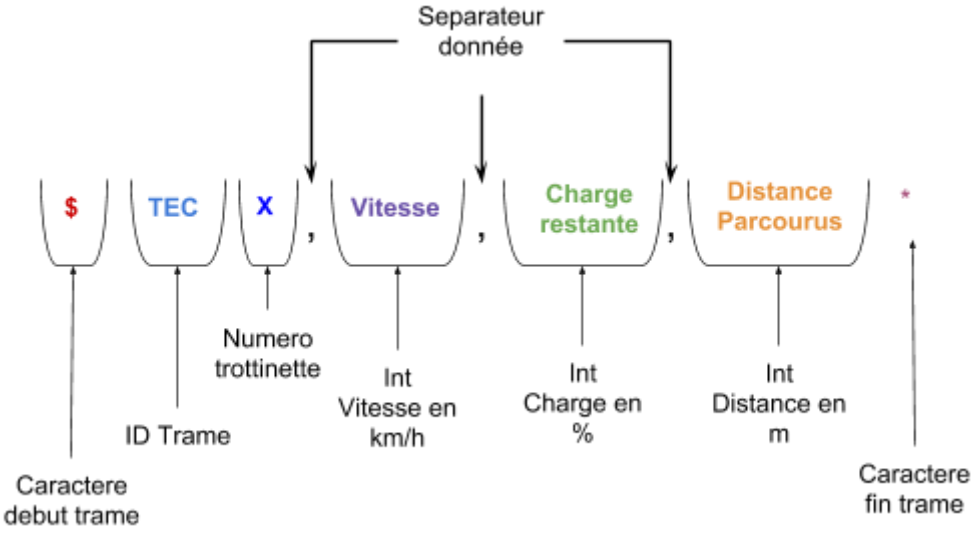
Position de la trottinette

Affiche les trames et la données de la TTE (mode technicien)

Listes des périphériques bluetooth appareillable

Protocole de Transmission

Trame de données



La trame reçu doit donc contenir:

- **Un caractère de début** : $
- **Un type de protocole** : TEC
- **Un numéro de trottinette** : X
- **La vitesse :** en km/h
- **La charge :** en %
- **La distance parcourue :** en m
- **Un caractère de fin de trame :** *

Exemple de trames :

Trames Reçues
$TEC1,25,70,1525*
$TEC1,28,69,1750*

Décodage :

TEC1	25	70	1525
TEC1	28	69	1750
ID TTE	Vitesse (km/h)	Charge restante (%)	Distance parcourus (m)

Pour traiter cette trame, on utilisera un type **String** afin d'utiliser une fonction **split()** qui permet de mettre chaque donnée dans un tableau.

Les trames seront envoyées toutes les 1000ms (1s) par défaut. La périodicité est fixée par l'étudiant EC.

On peut donc établir la fonction VerifierTrame() puis DecoderTrame() de la classe TrameTTE.

- VerifierTrame()

VerifierTrame() a pour but de vérifier si la trame est bien conforme au protocole de transmission.

VerifierTrame()

```
private Boolean VerifierTrame(String trame) throws InterruptedException
{
   if(trame.length() != 0)
   {
       if(trame.startsWith(DEBUT_TRAME))
       {
           if(trame.startsWith(DEBUT_TRAME + TYPE_PROTOCOLE))
           {
               if(trame.contains(FIN_TRAME))
               {
                   return true;
               }
               else
                   Log.e("<TrameTTE>","<FiltrerTrame> Trame non valide pas
de caractere de Fin");
           }
           else
               Log.e("<TrameTTE>","<FiltrerTrame> Trame non valide mauvais
type de trame");
       }
       else
           Log.e("<TrameTTE>","<FiltrerTrame> Trame non valide caractere
de debut different");
   }
   else
   {
       Log.e("<TrameTTE>","<FiltrerTrame> Trame vide");
   }
   return false;
}
```

Exemple : *$TEC1,25,70,1525**

On va successivement vérifier :

- si la trame est vide ou non :

```
if(trame.length() != 0)
{
    [...]
}
else
{
    Log.e("<TrameTTE>","<FiltrerTrame> Trame vide");
}
```

- Si la trame possède le caractère de début $:

```
if(trame.startsWith(DEBUT_TRAME))
{
    [...]
else
    Log.e("<TrameTTE>","<FiltrerTrame> Trame non valide caractere de debut
different");
```

- Si la trame est du type TEC :

```
if(trame.startsWith(DEBUT_TRAME + TYPE_PROTOCOLE))
{
    [...]
}
else
    Log.e("<TrameTTE>","<FiltrerTrame> Trame non valide mauvais type de
trame");
```

- Si la trame possède le caractère de fin * :

```
if(trame.contains(FIN_TRAME))
{
    return true;
}
else
    Log.e("<TrameTTE>","<FiltrerTrame> Trame non valide pas de caractere
de Fin");
```

- DecoderTrame()

Avant de décoder les données de la trame, on vérifie si elle possède le nombre correct de champs :

DecoderTrame()

```
public boolean DecoderTrame(String trame)
{
   trame = trame.substring(0, trame.length()-1);
   List<String> champs = new
ArrayList<String>(Arrays.asList(trame.split(",")));

   if(champs.size() == 4)
   {
       int position = 0;
       Vitesse = Integer.parseInt(champs.get(++position));
       ChargeBatterie = Integer.parseInt(champs.get(++position));
       DistanceParcourue = Integer.parseInt(champs.get(++position));
       String id = trame.substring(champs.get(0).indexOf("C") + 1,
champs.get(0).length());
       IdTEC = Integer.parseInt(id);

       return true;
   }
   return false;
}
```

Exemple : *$TEC1,25,70,1525**

On commence par enlever le caractère de fin de trame avec :

```
trame = trame.substring(0, trame.length()-1);
```

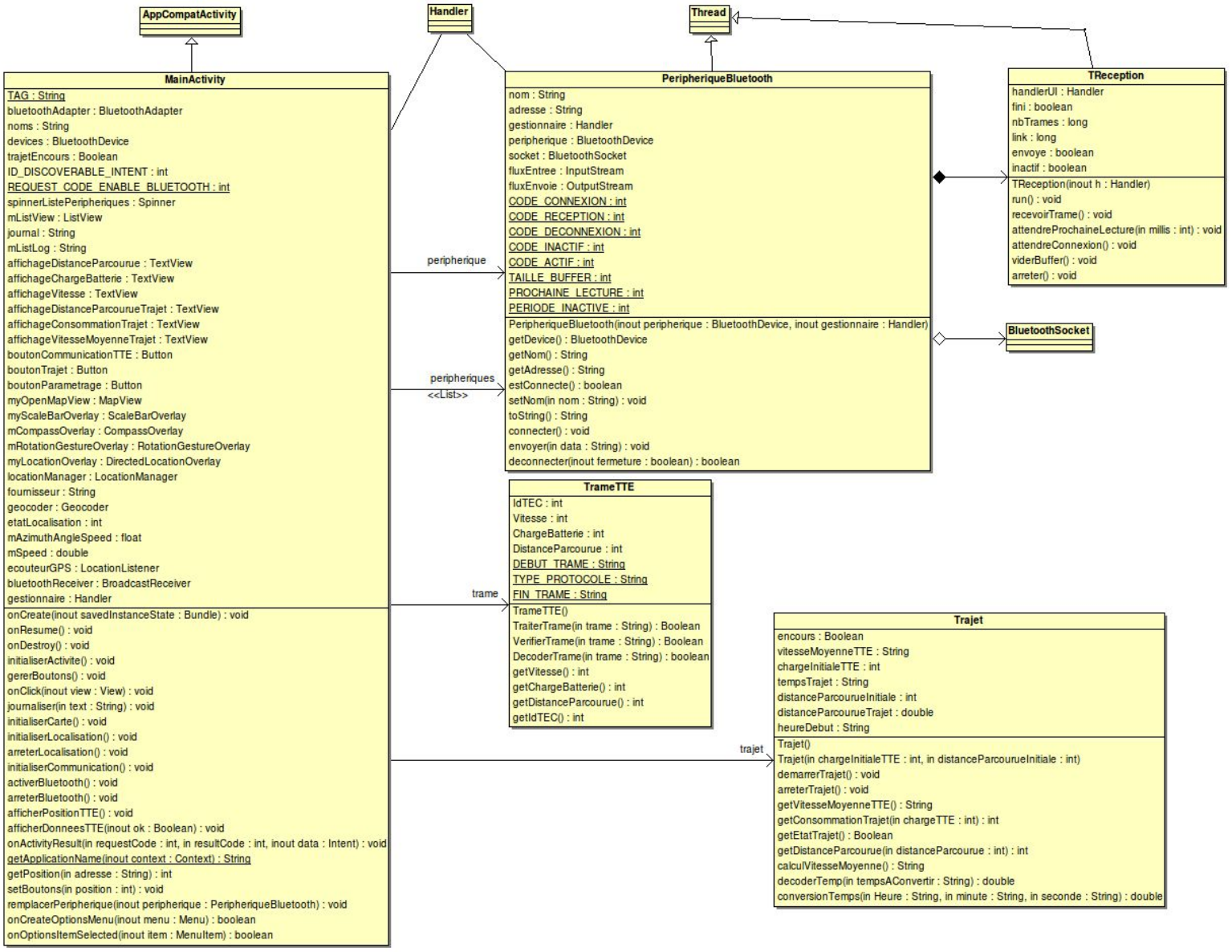
Puis on utilise les fonctions associés à la classe String, telles que split() qui permet de diviser une donnée délimitée par des virgules :

```
List<String> champs = new
ArrayList<String>(Arrays.asList(trame.split(",")));
```

On termine par extraire chaque donnée :

```
if(champs.size() == 4)
{
   int position = 0;
   Vitesse = Integer.parseInt(champs.get(++position));
   ChargeBatterie = Integer.parseInt(champs.get(++position));
   DistanceParcourue = Integer.parseInt(champs.get(++position));
   String id = trame.substring(champs.get(0).indexOf("C") + 1,
champs.get(0).length());
   IdTEC = Integer.parseInt(id);
   return true;
}
```

Diagramme de classes



- Rôle des classes

- La classe TReception

Elle permet d'écouter les connexions bluetooth sur le terminal et d'envoyer les différents codes et les données en cas de réception. Il est donc nécessaire que cette classe dispose de son propre fil d'exécution elle hérite donc de la classe Thread[4].

- La classe PeripheriqueBluetooth

Cette classe est aussi mise dans un thread pour avoir accès à ces fonctionnalités la connexion au bluetooth, la deconnexion et si besoin l'envoie de trame. Elle permet la création de module bluetooth sur notre terminal pour la connexion avec eux. Elle est reliée à la classe MainActivity grâce à un Handler[5].

- La classe TrameTTE

Cette classe permet de gérer l'analyse de la trame pour identifier si c'est une trame TEC a la suite de cela elle décode la trame pour stocker les données de la TEC qui sont ensuite envoyées par le handler au MainActivity.

- La classe Trajet

La classe trajet permet de gérer les données d'un trajet lorsque nous appuyons sur le bouton trajet a partir de ce moment est calcule la vitesse moyenne la distance parcours et le taux de décharge de la batterie durant le trajet.

Une fonction est offerte avec la classe Localisation qui permet d'update on peut donc modifier les paramètres pour afficher la position de la trottinette.

Dans l'activité MainActivity

```
private void initialiserLocalisation()
{
     [...]
     if (fournisseur != null)
     {
      [...]
       // on configure la mise à jour automatique : au moins 1 mètres et 5
secondes
       locationManager.requestLocationUpdates(fournisseur, 5000, 1,
ecouteurGPS);
   }
}
```

[4] Thread : Fil d'exécution en parallèle de celui du main
[5] Handler : Utiliser pour communiquer entre un thread et le main, il met aussi à jour l'IHM.

- La classe MainActivity

Cette classe représente notre point d'entrée dans le programme. C'est ici que les objets sont instanciés. Elle a pour rôle d'afficher l'IHM. Une liste déroulante permet de sélectionner le module bluetooth auquel nous voulons nous connecter. A la selection d'un module dans cette liste déroulante un périphérique sera alors créé et lors de la connexion ou déconnexion les information techniques seront affichées dans une "console".

Dans l'Activité MainActivity

```
ArrayAdapter<String> adapter = new ArrayAdapter<String>(this,
android.R.layout.simple_spinner_item, noms);
adapter.setDropDownViewResource(android.R.layout.simple_spinner_dropdown_it
em);
spinnerListePeripheriques.setAdapter(adapter);

spinnerListePeripheriques.setOnItemSelectedListener(new
AdapterView.OnItemSelectedListener()
{
   public void onItemSelected(AdapterView<?> arg0, View arg1, int
position, long id)
   {
       peripherique = peripheriques.get(position);
       setBoutons(position);
   }
[...]
});
```

Dans cette classe on retrouve aussi le système de handler précédemment introduit.

Les instanciation de nos ressources pour l'affichage de nos données :

Dans l'Activité MainActivity

```
boutonTrajet = (Button) findViewById(R.id.boutontrajet);
spinnerListePeripheriques = (Spinner)
findViewById(R.id.spinnerListeAppareils);
affichageDistanceParcourue = (TextView)
findViewById(R.id.affichageDistanceParcourue);
affichageChargeBatterie = (TextView)
findViewById(R.id.affichageChargeBatterie);
affichageVitesse = (TextView) findViewById(R.id.affichageVitesse);
```

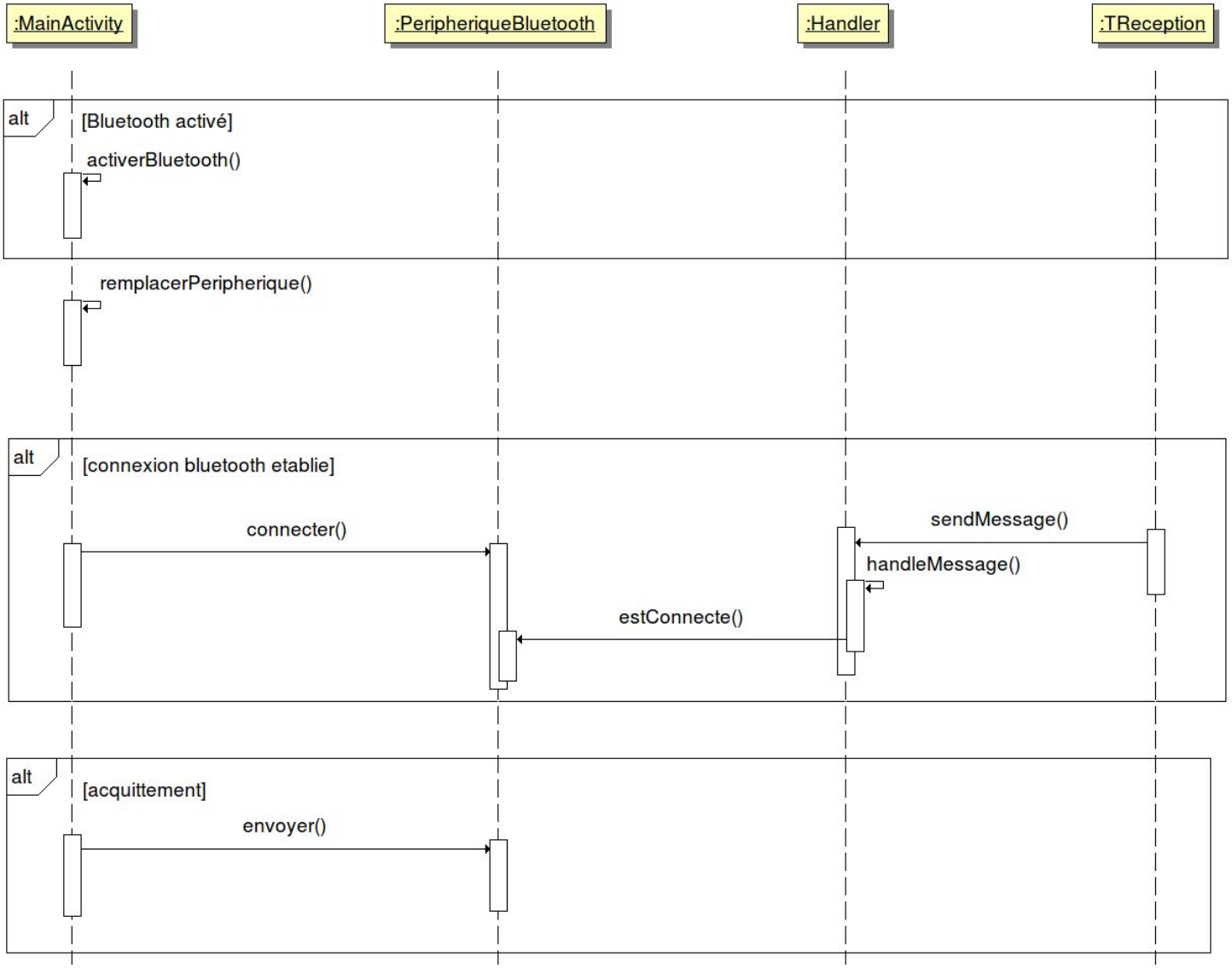
Étude détaillée

Scénarios

- Le cas d'utilisation "Dialoguer avec la TEC"

La récupération des données de fonctionnement de la trottinette nécessite un dialogue via le Bluetooth.

- Diagramme de séquence



On remarque sur ce diagramme un échange d'un message depuis TReception qui est un Thread. Ce message contient donc :

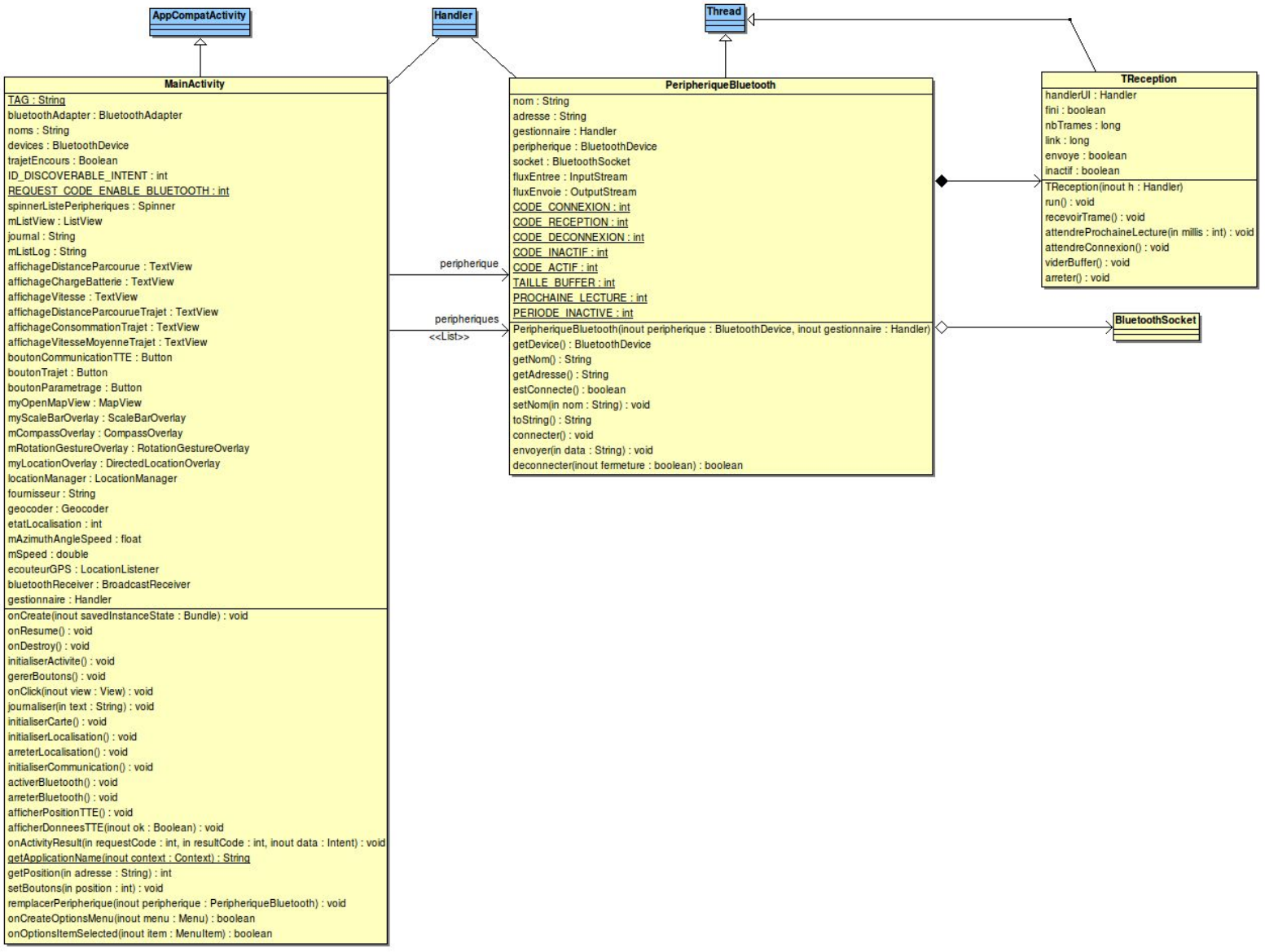
- Le **nom du module**
- Son **adresse MAC**
- Un **etat (connection,deconnection ou reception)**
- Et **la données**

Dans la classe TReception

```
Message msg = Message.obtain();
Bundle b = new Bundle();
b.putString("nom", getNom());
b.putString("adresse", getAdresse());
b.putInt("etat", CODE_RECEPTION);
b.putString("donnees", data);
msg.setData(b);
handlerUI.sendMessage(msg);
```

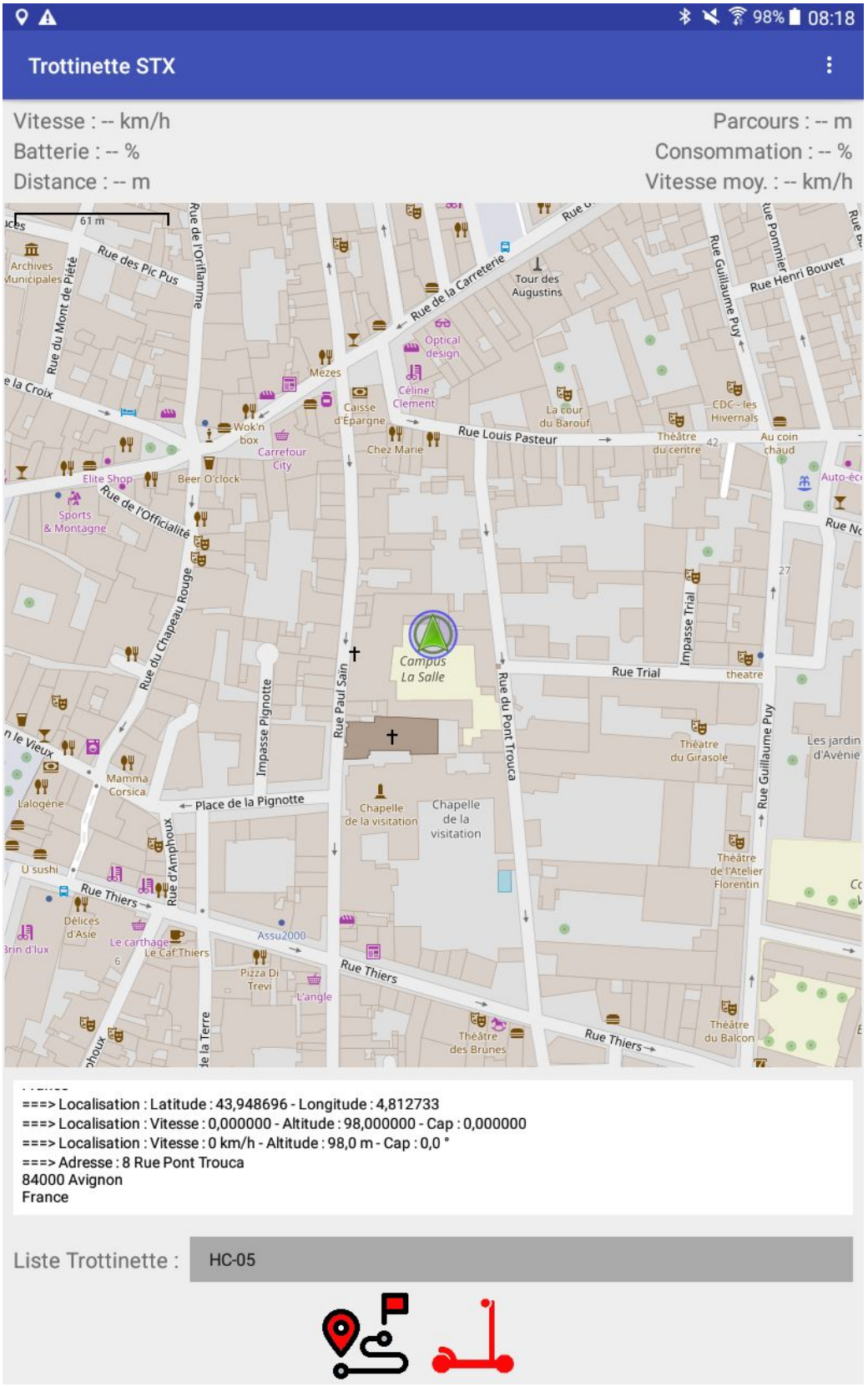
Ce message est ensuite empaqueté dans un bundle pour le transfert au main activity via le handler.

- Diagramme de classes partiel



- Tests

Notre application au demarrage :



Dans cette partie notre test a pour but d'observer une connexion vers un module bluetooth ici nous avons filtré les modules HC-05 :

Code du filtrage

Dans l'activite MainActivity

```
for (BluetoothDevice blueDevice : devices)
{
   if((blueDevice.getName().equals("HC-05")))
   {
       peripheriques.add(new PeripheriqueBluetooth(blueDevice,
gestionnaire));
       noms.add(blueDevice.getName());
   }
}
```

Si notre filtre n'est pas actif on a alors une liste beaucoup plus large de tous les modules appareillés :

A partir de cette liste, nous sélectionnons le module bluetooth que nous voulons connecter.

Dans la console du technicien, les informations du module sont affichées :

On y retrouve le **nom du module**, **son adresse mac** et **son état.**

Nous allons ensuite pouvoir nous connecter grâce au bouton suivant :

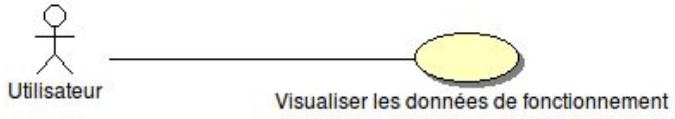
Le module est alors en cours de connexion et on peut observer dans la console :

Lorsqu'il est connecté on a l'affichage du périphérique sélectionné avec son nouvel état, le bouton de connexion change également de couleur pour le signifier à l'utilisateur :

```
===> Connexion HC-05 [98:D3:31:F5:6E:7A] ok
===> Périphérique sélectionné HC-05 [98:D3:31:F5:6E:7A] : connecté
```

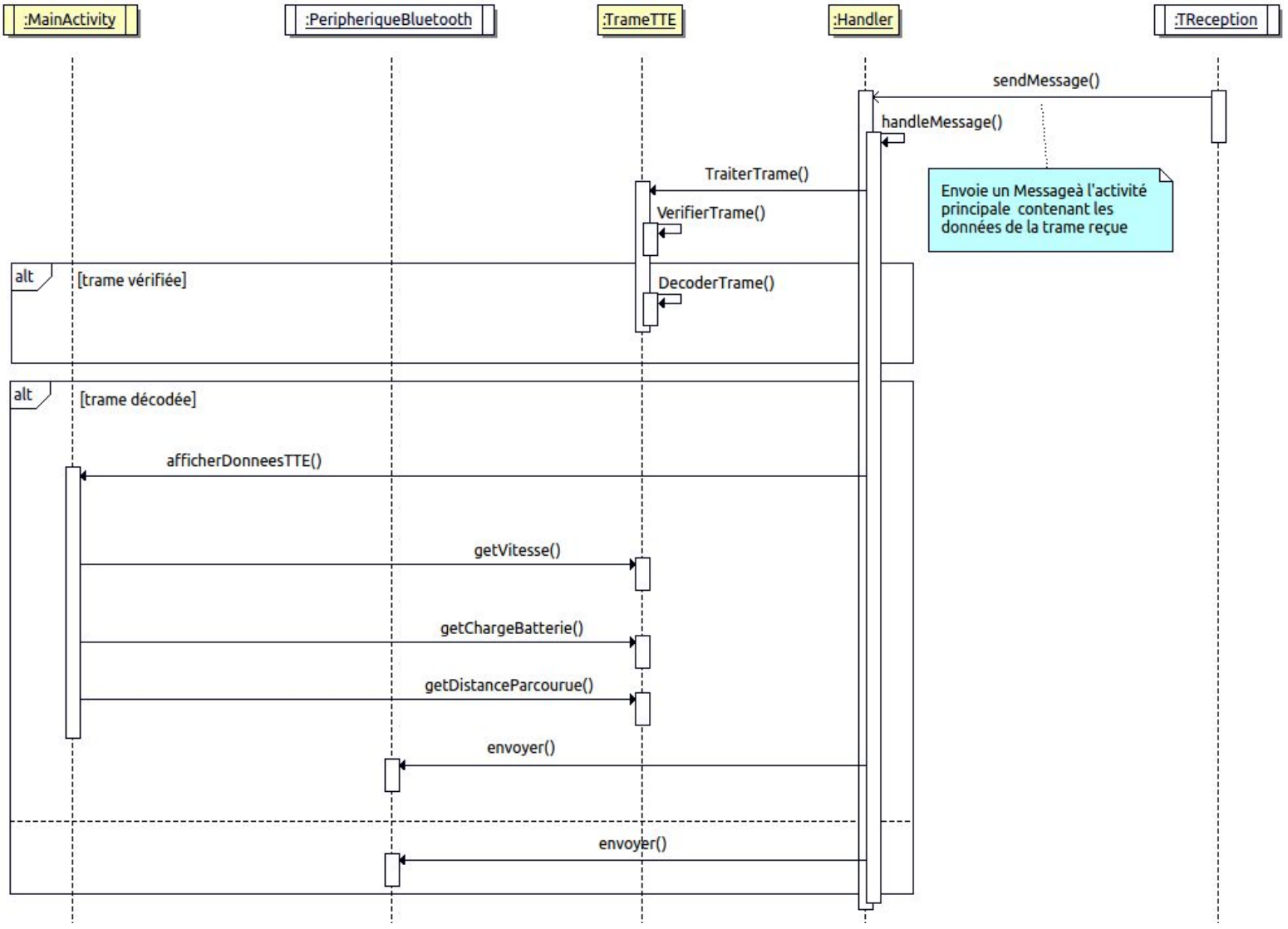
On obtient :

- Le cas d'utilisation "Visualiser les données de fonctionnement"



L'utilisateur doit pouvoir visualiser les données de fonctionnement de la trottinette. Pour cela il est nécessaire qu'une connexion Bluetooth soit établie.

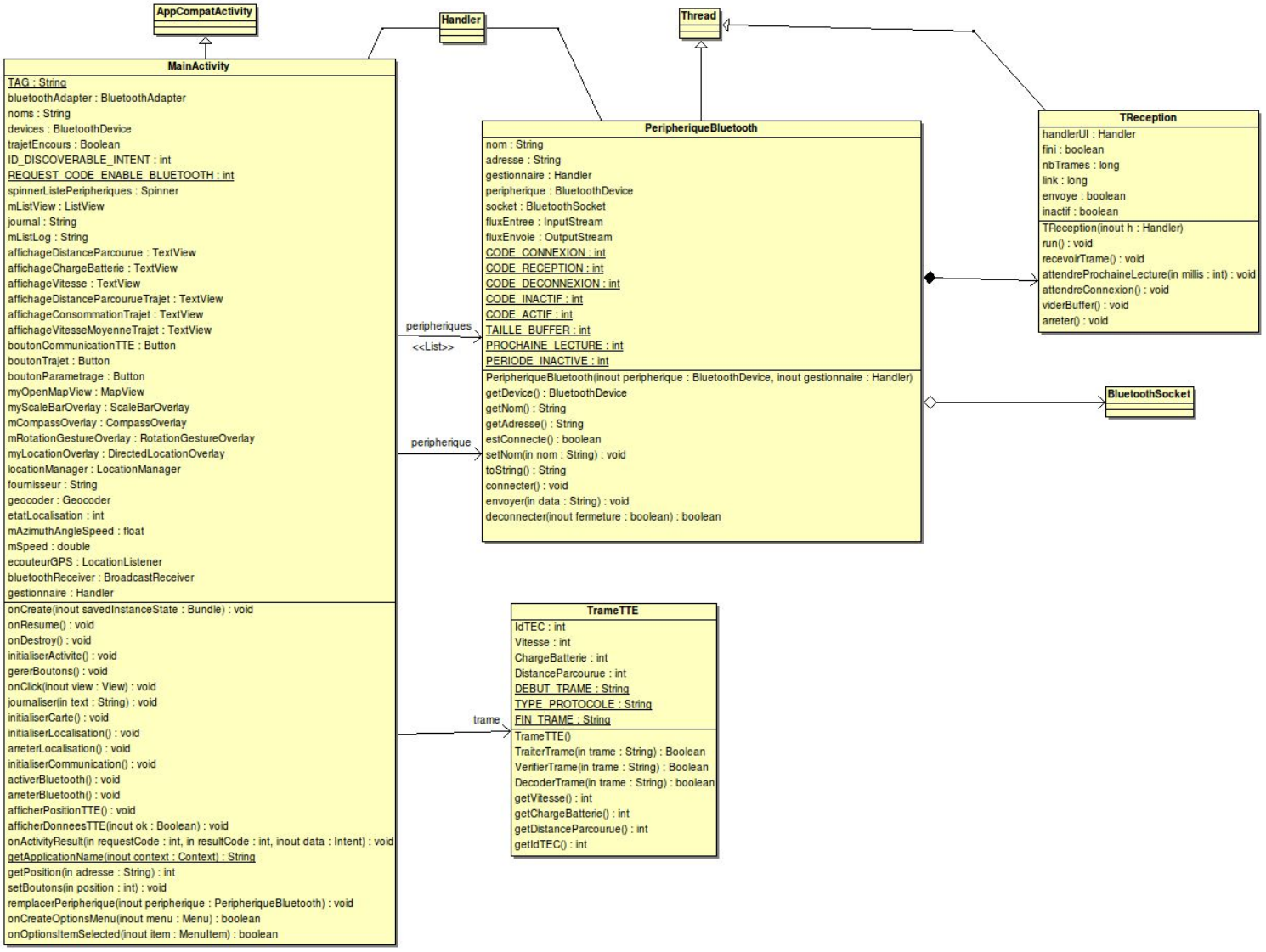
- Diagramme de séquence



La réception de nouveau message est fixé à 1s et nous permet de recevoir une trame qui est ensuite vérifiée pour savoir si elle est du bon type avant qu'elle soit décodée pour récupérer les données de notre trottinette.

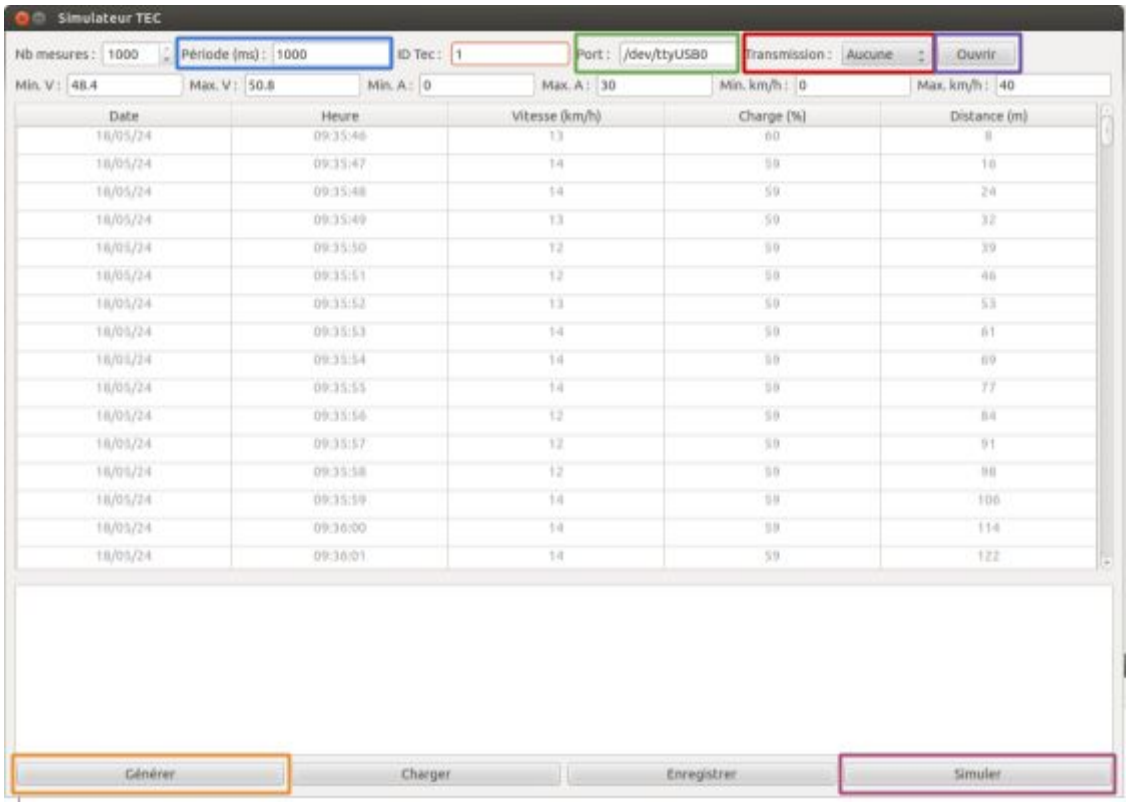
On remarque dans le diagramme qu'une trame est échangée entre MainActivity et TReception à l'aide du handler. Elle est ensuite traitée par la classe TrameTTE de telle sorte que la donnée soit affichable dans des TextView de notre IHM. La trame est un String et contient la vitesse, la charge de la batterie et la distance parcourue par la trottinette.

- Diagramme de classes partiel



- Tests

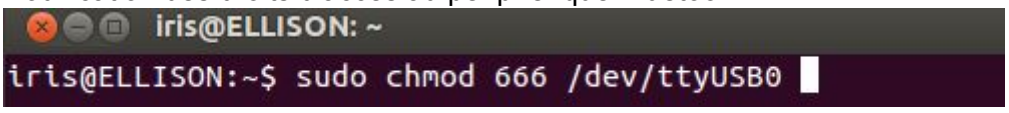
L'objectif de ce test est de visualiser les données de fonctionnement de la trottinette
Après la connexion au bluetooth, nous activons notre simulateur de trame :



Ce simulateur nous permet de générer des trames depuis un module bluetooth que nous avons relié à un PC. On effectuer les réglages suivants :

- **La période en ms d'envois des trames**
- **Le port sur lequel notre module bluetooth est branché**
- **Le type de transmission que nous voulons**

Modification des droits d'accès au périphérique Bluetooh :



On est alors prêt à envoyer nos trames vers notre application à l'aide du bouton Générer puis Simuler.

On obtient :

Notre trame a donc été traitée de telle sorte à ce que nous puissions récupérer la vitesse la charge de la batterie et la distance parcourue.

Lorsqu'une trame est envoyée par notre simulateur, elle arrive à notre TReception qui va **signifier à l'activité principale** grâce à un **code reception** qu'une **donnée** est disponible :

Dans la classe TReception

```
private void recevoirTrame()
{
    [...]
    String datas = new String(rawdata);
    System.out.println("<Bluetooth> Reception " + datas);
    StringBuffer d = new StringBuffer(datas);
    Message message = Message.obtain();
    Bundle paquet = new Bundle();
    paquet.putString("nom", getNom());
    paquet.putString("adresse", getAdresse());
    paquet.putInt("etat", CODE_RECEPTION);
    paquet.putString("donnees", datas);
    message.setData(paquet);
    handlerUI.sendMessage(message);
    [...]
}
```

Dans l'activité principale MainActivity, nous faisons alors appel à une classe TrameTTE ou la trame va être **vérifiée** et si elle est bonne **l'affichage sera activé**

Dans l'activite MainActivity

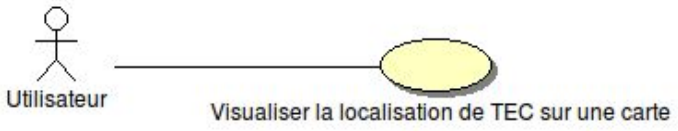
```
case PeripheriqueBluetooth.CODE_RECEPTION:
   [...]
           if(trame.TraiterTrame(donnees))
           {
                afficherDonneesTTE(true);
                [...]
           }
   [...]
   break;
```

Dans la classe trameTTE

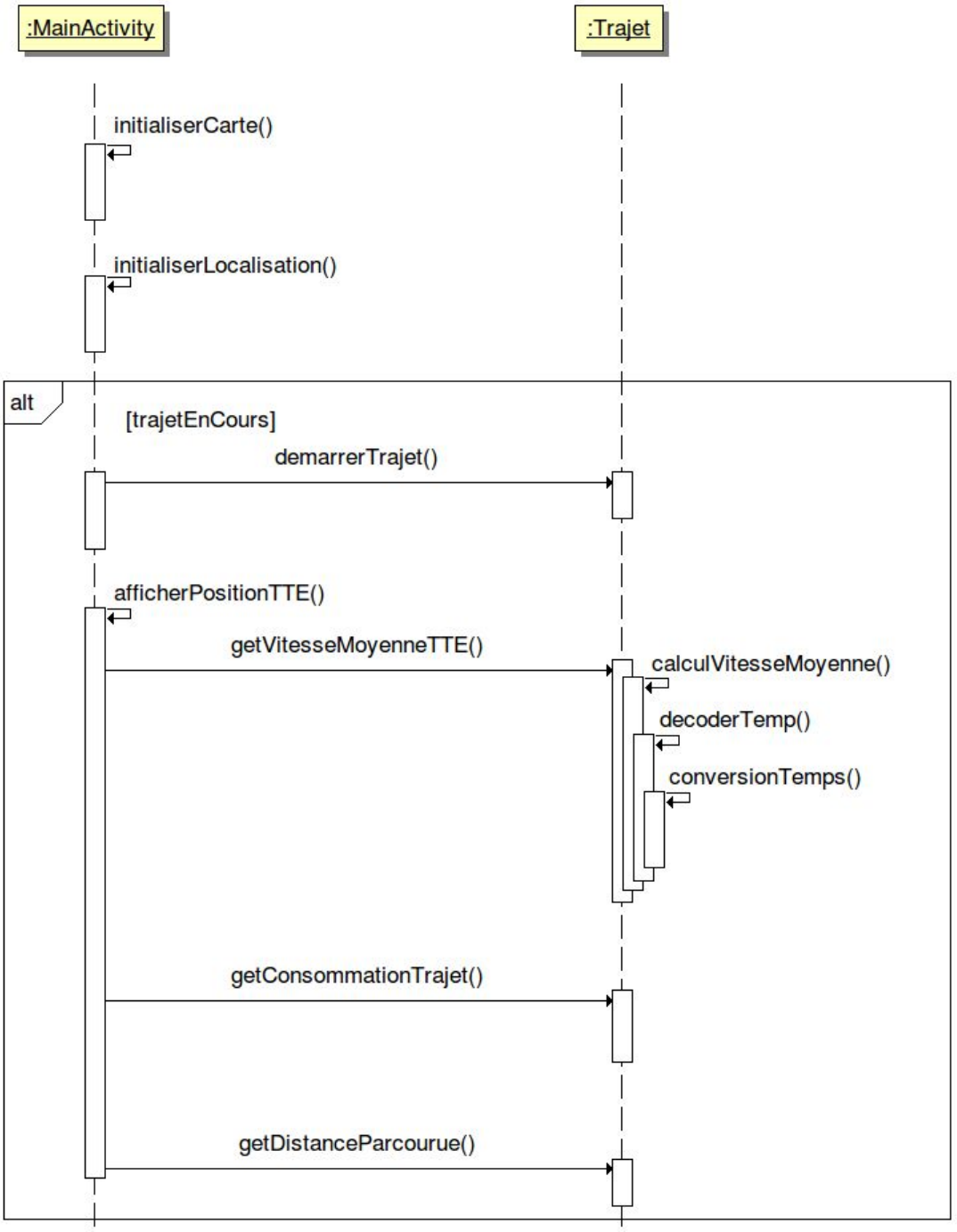
```
public Boolean TraiterTrame(String trame) throws InterruptedException
{
   if(VerifierTrame(trame))
   {
       if(DecoderTrame(trame))
           return true;
   }
   return false;
}
```

- Le cas d'utilisation "Visualiser la localisation de TEC sur une carte"

Dans cette partie l'objectif est de pouvoir visualiser la localisation de la trottinette et le trajet effectué depuis l'activation du bouton pour cela il est nécessaire qu'une connexion bluetooth soit établie.



- Diagramme de Séquence



Initialisation

Nous devons en premier temps, pour cette activité, initialiser nos variables ce qui est effectué dans initialiserCarte() puis on continue avec la fonction initialiserLocalisation() :

Dans l'activite MainActivity

```
private void initialiserLocalisation()
{
   [...]
   if (fournisseur != null)
   {
      [...]
       Location localisation =
locationManager.getLastKnownLocation(fournisseur);
       if(localisation != null)
       {
           // on notifie la localisation
           ecouteurGPS.onLocationChanged(localisation);
       }

       // on configure la mise à jour automatique : au moins 1 mètres et 5
secondes
       locationManager.requestLocationUpdates(fournisseur, 5000, 1,
ecouteurGPS);
   }
}
```

On peut donc voir que si nous recevons un "signal gps" nous allons "regarder" la dernière position connue et si il y en a une, on installe le gestionnaire pour l'écoute :

```
ecouteurGPS.onLocationChanged(localisation);
```

Cette ligne signifie que l'on a placé en quelque sorte une mis a jour automatique lors de changement qui appellera la fonction onLocationChanged(). Elle permet de changer la visualisation sur la carte.

Demander la mise à jour de la carte ce qui est effectué par cette ligne :

```
locationManager.requestLocationUpdates(fournisseur, 5000, 1,
ecouteurGPS);
```

- 5000 represente donc le nombre de millisecondes.
- 1 représente le nombre de mètres.

Ces paramètres règlent la précision du changement de position.

AfficherPositionTTE()

On a ensuite recours à afficherPositionTTE() qui va permettre d'afficher la position de notre TTE en signifiant à notre ecouteurGPS que la localisation de la trottinette a changé.

Dans l'Activité MainActivity

```
private void afficherPositionTTE()
{
   [...]
   if (locationManager != null)
   {
       if (fournisseur != null)
       {
           Location localisation =
locationManager.getLastKnownLocation(fournisseur);
           if (localisation != null)
           {
               // on notifie la localisation
               ecouteurGPS.onLocationChanged(localisation);
           }
       }
   }
}
```

Notre ecouteurGPS aura alors recours dans sa fonction onLocationChanged à **l'affichage sur la carte** de la position de la trottinette.

Dans l'activité MainActivity

```
LocationListener ecouteurGPS = new LocationListener() {
   @Override
   public void onLocationChanged(Location localisation)
   {
       [...]
       GeoPoint nouvelleLocalisation = new GeoPoint(localisation);
       [...]
       myOpenMapView.getController().animateTo(nouvelleLocalisation);
       myOpenMapView.setMapOrientation(-mAzimuthAngleSpeed);
       [...]
   }
   [...]
};
```

Ensuite intervient la partie du trajet, nous allons alors avoir besoin d'afficher ces données pour cela une classe Trajet a été implémenté et 3 fonctions pour accéder aux données sont nécessaire.

- getVitesseMoyenneTTE()

Dans le cas ou nous voulons récupérer la vitesse moyenne de notre trottinette, on va la calculer puis la retourner.

Dans la classe Trajet

```
public String getVitesseMoyenneTTE()
{
   vitesseMoyenneTTE = calculVitesseMoyenne();
   return vitesseMoyenneTTE;
}
```

↳ calculVitesseMoyenne()

Dans cette partie on introduit le besoin d'avoir le **temps depuis le début du trajet** ce qui va nous permettre de **calculer la vitesse moyenne**

Dans la classe Trajet

```
private String calculVitesseMoyenne()
{
   [...]
   if(heureDebut.length() > 0)
   {
       double dureeTrajet = decoderTemp(heureActuelle) -
decoderTemp(heureDebut);
       tempsTrajet = String.format("%.2f",dureeTrajet);
       double vitesseMoyenne = (distanceParcourueTrajet/1000) /
dureeTrajet;
       vitesseMoyenneTTE = String.format("%.2f",vitesseMoyenne);
       return (vitesseMoyenneTTE);
   }
   return "0";
}
```

↳ decoderTemp()

Cette dernière partie a pour objectif de séparer les **heures, minutes et secondes** de notre **temps à convertir** pour pouvoir effectuer sa conversion en double pour le calcul de la vitesse moyenne.

Dans la classe Trajet

```
private double decoderTemp(String tempsAConvertir)
{
   [...]
   for (int i=0; i < tempsAConvertir.length() ;i++)
   {
       [...]
       else
       {
           if (position == 0)
           {
               heure = heure + tempsAConvertir.charAt(i);
           } else if (position == 1)
           {
               minute = minute + tempsAConvertir.charAt(i);
           } else
           {
               seconde = seconde + tempsAConvertir.charAt(i);
           }
       }
   }
   return (conversionTemps(heure,minute,seconde));
}
```

↳ conversionTemps()

Cette dernière fonction permet de **convertir notre temps** en double pour le calcul.

Dans la classe Trajet

```
private double conversionTemps(String Heure, String minute, String
seconde)
{
   double temps = 0;
   Double heureConversion;
   Double minuteConversion;
   Double secondeConversion;

   heureConversion = Double.parseDouble(Heure);
   minuteConversion = Double.parseDouble(minute) / 60;
   secondeConversion = Double.parseDouble(seconde) / 3600;
   [...]
   temps = heureConversion + minuteConversion + secondeConversion;
   return temps;
}
```

- getConsommationTrajet()

Ici on cherche à récupérer la charge depuis le début de notre trajet on soustrait donc **la charge initiale de la trottinette lors de l'activation du mode trajet** et on y soustrait la **charge actuelle de la batterie**.

Dans la classe Trajet

```
public int getConsommationTrajet(int chargeTTE)
{
   return chargeInitialeTTE - chargeTTE;
}
```

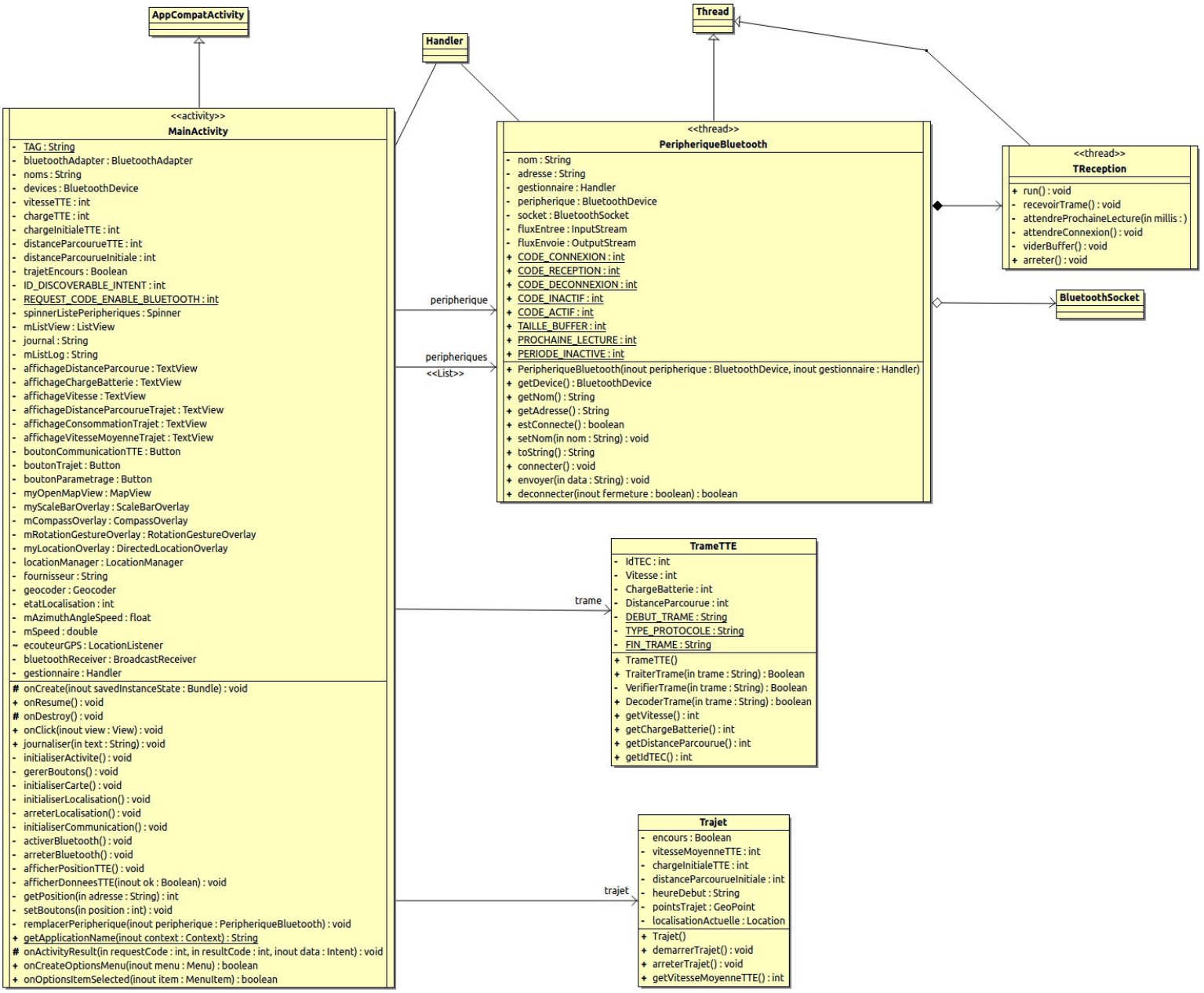
- getDistanceParcourue()

On cherche à récupérer la distance parcourue depuis le début de notre trajet on soustrait donc la **distance parcourue initialement** par la **distance parcourue** actuellement.

Dans la classe Trajet

```
public int getDistanceParcourue(int distanceParcourue)
{
   distanceParcourueTrajet = distanceParcourue -
distanceParcourueInitiale;
   return (distanceParcourue - distanceParcourueInitiale);
}
```

- Diagramme de classes partiel



- Tests

L'objectif de ce test est de visualiser la localisation de la trottinette ainsi que pouvoir activer un mode trajet pour visualiser le parcours a effectué.

Pour le mode trajet, il est impératif que notre module bluetooth soit connecté à la trottinette.

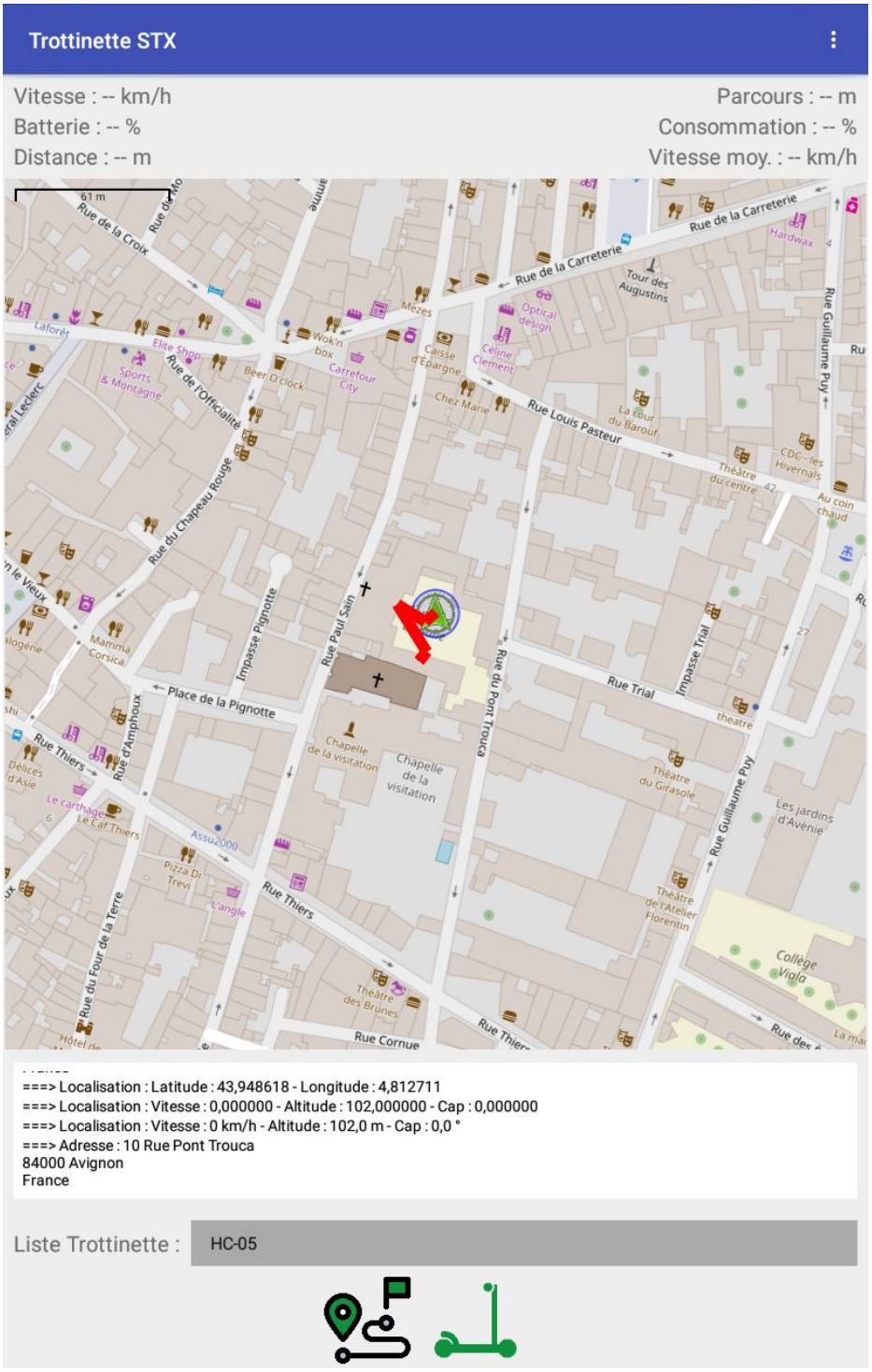
Nous pouvons maintenant activer le mode trajet à l'aide du bouton :

Le trajet passera alors en mode actif :

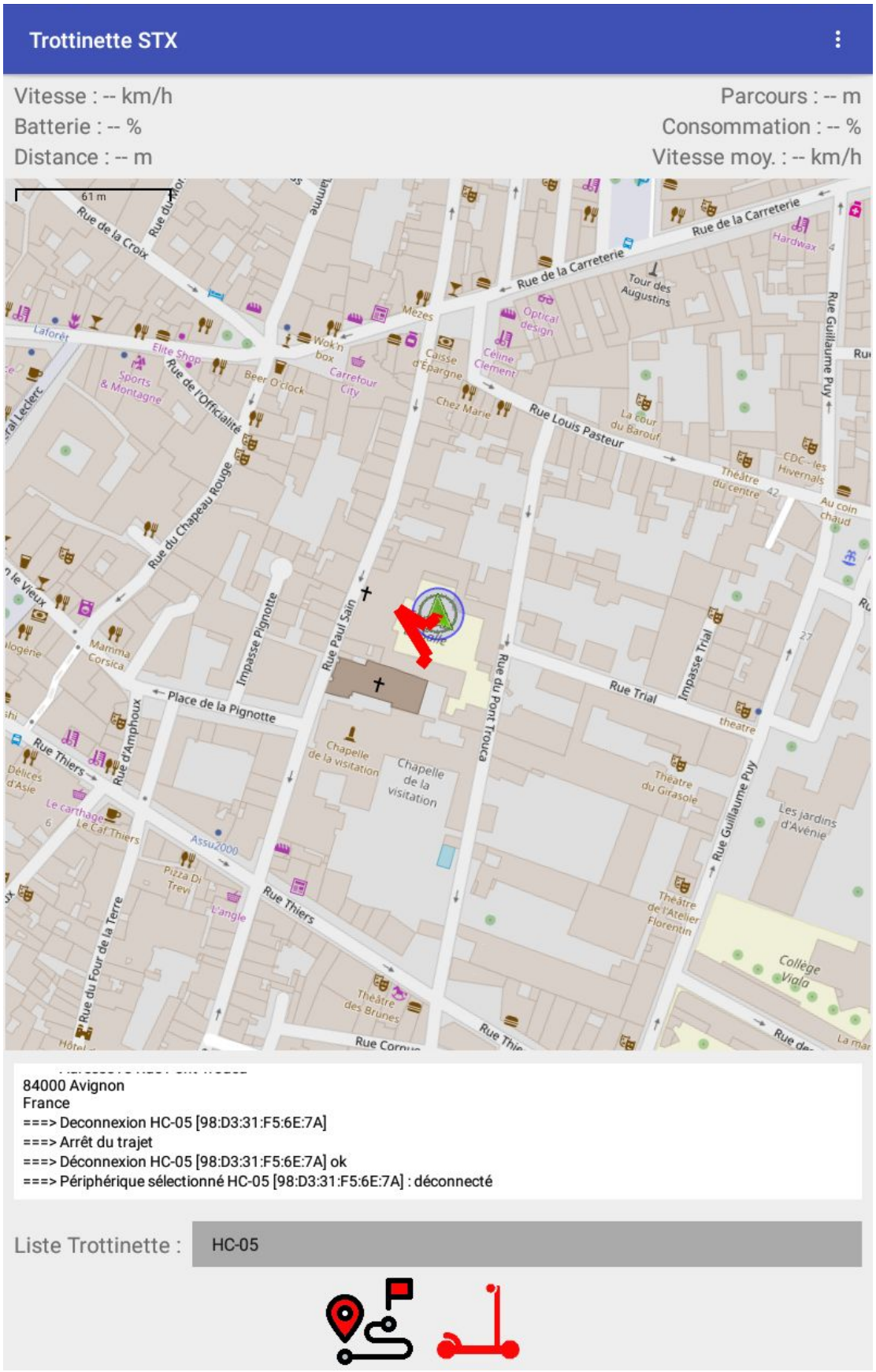
Le mode trajet nous permet d'obtenir les données de notre trajet comme vu précédemment depuis le code.

```
Vitesse moy : 27,00 km/h
Consommation : 0 %
Parcours : 23 m
```

Une fonctionnalité a été ajouté qui est l'affichage de notre trajet en cours à l'aide d'une ligne rouge selon les déplacements depuis la dernière position.



Une fois le mode trajet activé, si nous déconnectons le module bluetooth, le trajet s'arrête instantanément.



Tests de validation

Test de validation		
Nom du test	Oui	Non
Le protocole de communication avec la TTE est spécifié et mis en oeuvre	X	
La réception des données de fonctionnement de la TTE est effective	X	
La visualisation des données de fonctionnement de la TTE et la durée d'utilisation est fonctionnelle	X	
L'autonomie pour un parcours est calculée et affichée	X	
La carte avec la géolocalisation de la TTE est affiché et actualisée périodiquement	X	

Conclusion

- Nous pouvons faire des améliorations graphique comme un compteur de vitesse.
- Créer une base de données pour stocker nos trajet et les données de ceci.
- Saisir une destination et crée un trajet pour y arriver.

Glossaire

TTE : Trottinette tout Terrain Électrique
TEC : Trottinette Électrique Connectée (la TTE avec le système développé)
Diagramme de Gantt : c'est un outil permettant de visualiser dans le temps les diverses tâches composant un projet.
Thread : Activité en multitâche
Handler : Gestionnaire pour passer sur une autre activité