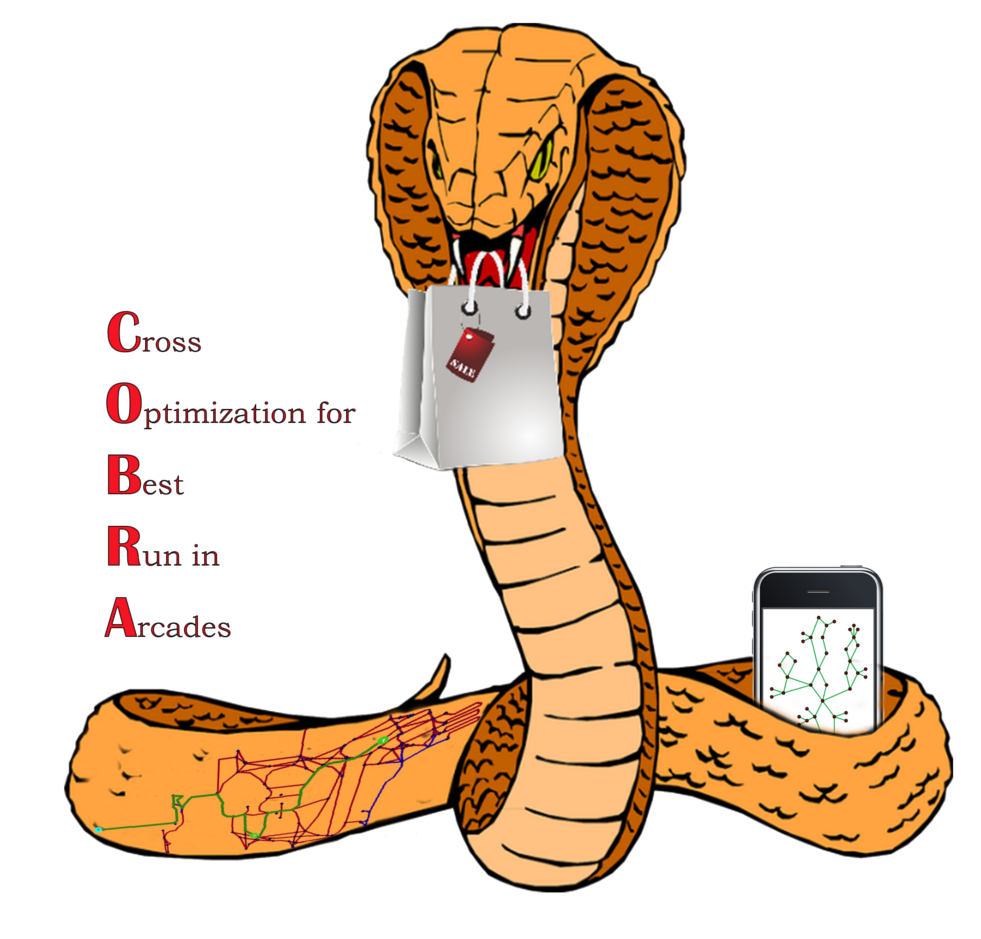
**MANUEL DE DEVELLOPEMENT DE L’APPLICATION C.O.B.R.A.**



Sommaire :

[1. Modélisation : 3](#_Toc450902250)

[1.1 Cas d’utilisation : 3](#_Toc450902251)

[1.2 Diagramme de classes : 4](#_Toc450902252)

[1.3 Diagramme d’activité : 5](#_Toc450902253)

[1.4 Diagramme de séquence : 6](#_Toc450902254)

[2. Organisation et fonctionnement du code : 7](#_Toc450902255)

[2.1 Organisation général : 7](#_Toc450902256)

[2.2 Fonctionnement des différentes parties : 7](#_Toc450902257)

[2.2.1 MainActivity.java : 7](#_Toc450902258)

# Introduction :

Nous allons vous présenter dans ce manuel l’organisation et le fonctionnement du code de notre application C.O.B.R.A. qui est une application mobile d'aide au déplacement dans le centre commercial des Arcades situé à Noisy-le-Grand qui a pour caractéristiques notamment d’être multi-niveau.

Le code de notre application est disponible sur git au lien suivant :

<https://github.com/hbaltz/projetDev.git>

Le code final se situe dans 2-Android\Assemblee\AndroidProjet

En annexe vous trouverez la liste de l’ensemble des variables globales avec leur utilité.

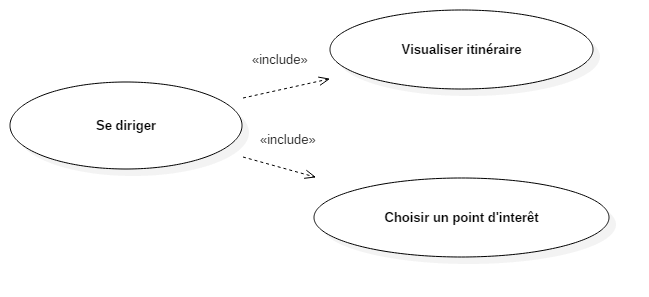
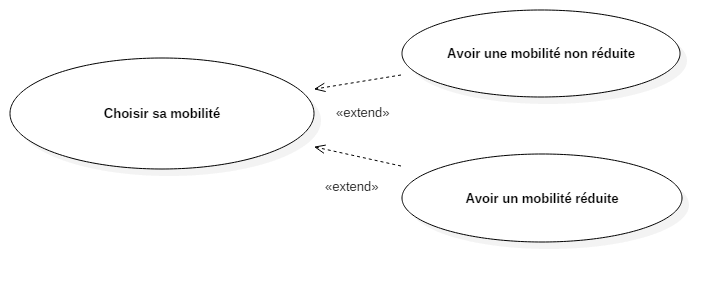
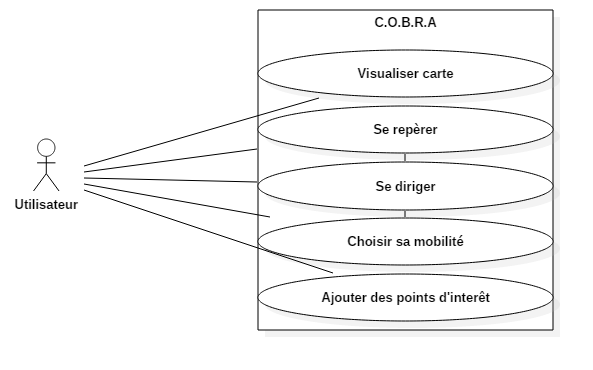
# Modélisation :

## Cas d’utilisation :

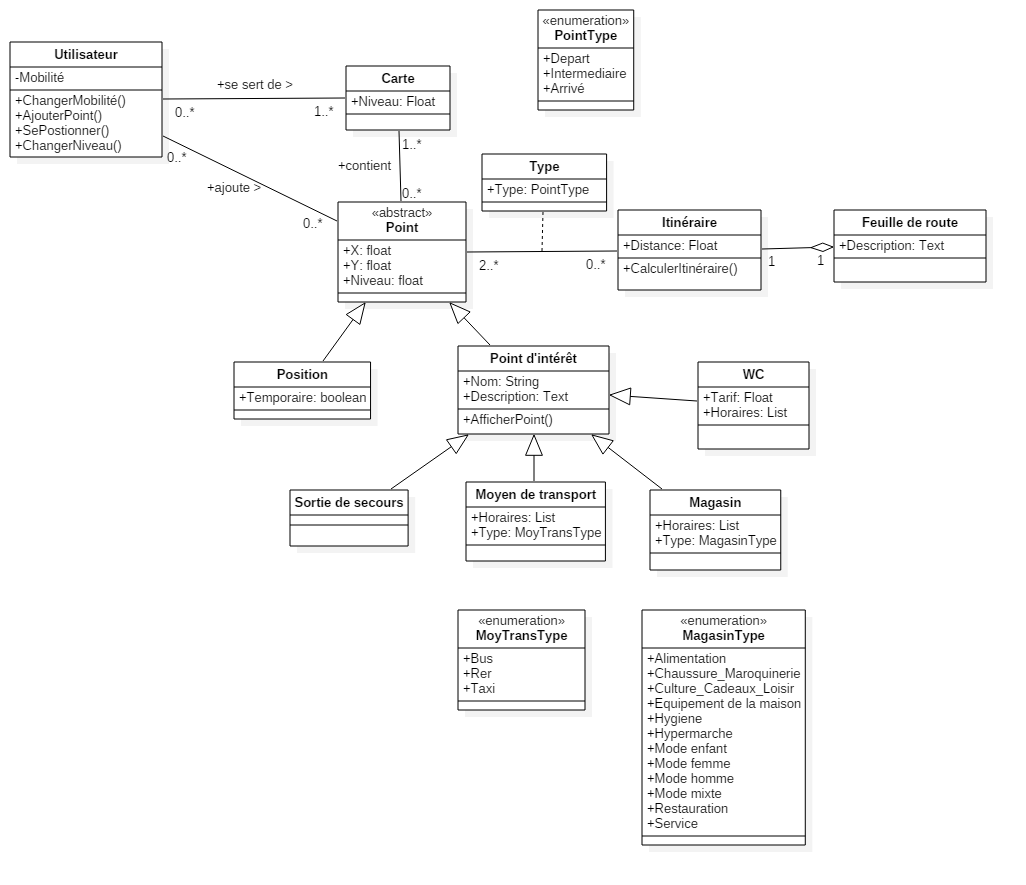
Notre application doit aider un utilisateur à effectuer différentes tâches dont les principales sont listées ci-dessous :

* Visualiser la carte du centre commercial
* Se repérer dans ce centre
* Se diriger à l’intérieur des Arcades
* Choisir sa mobilité, c’est-à-dire si il a une mobilité réduite (handicap, poussette, caddie) ou non
* Ajouter des points d’intérêt

Vous trouverez ci-dessous les diagrammes de cas d’utilisation représentant les fonctionnalités principales de notre application COBRA (Cross Orientation for Best Run in Arcades) :



## Diagramme de classes :



Nous avons identifiés 6 classes principales :

* Utilisateur
* Carte
* Point (Dont position et point d’intérêt est une classe enfant)
* Point d’intérêt (Dont WC, sortie de secours, moyen de transport, magasin sont des classes enfants)
* Itinéraire
* Feuille de route

Nous sommes partis de données préexistantes notre modélisation s’est donc basée sur ses données, notamment pour les classes Point et Point d’intérêt.

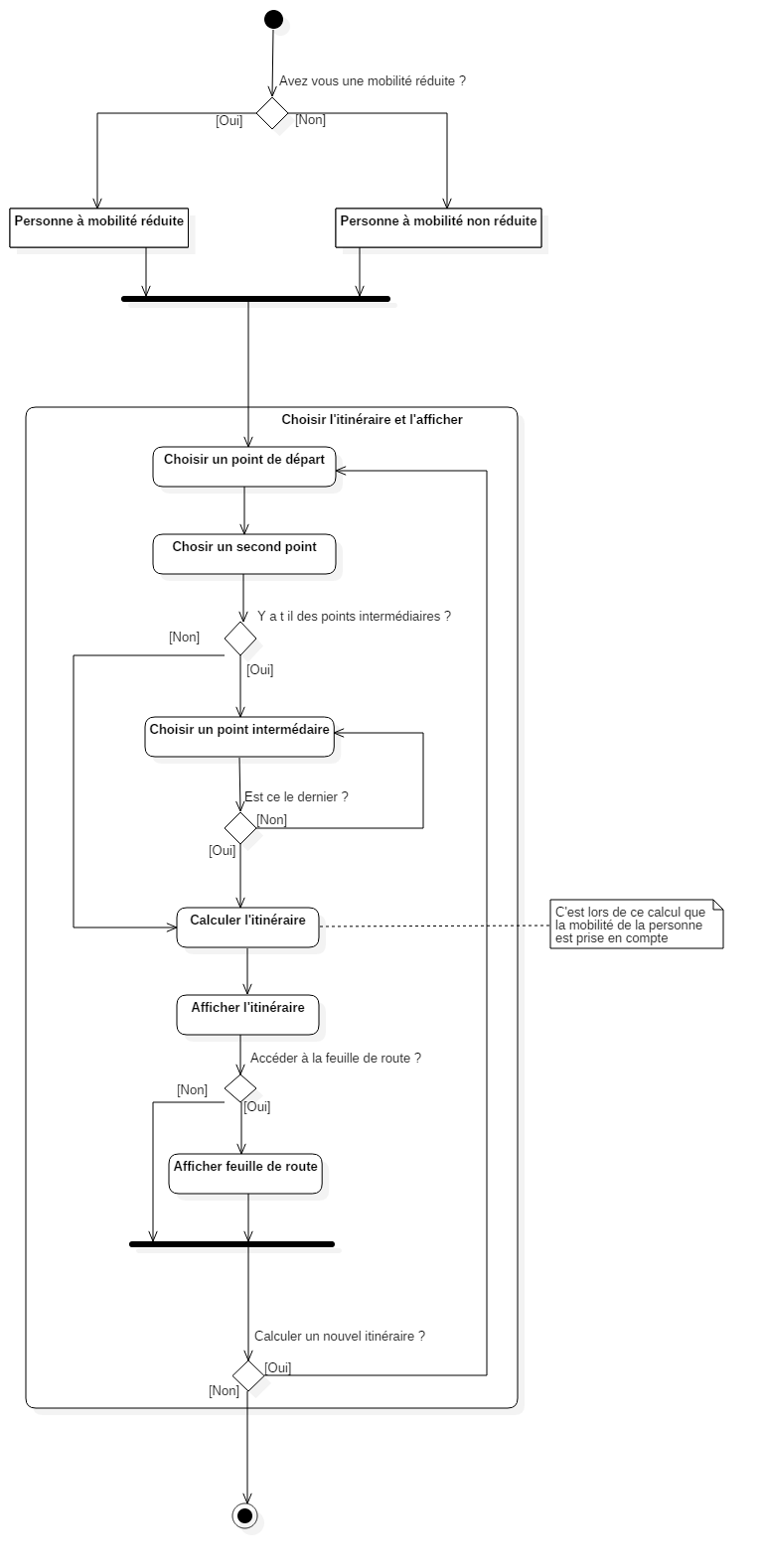
Les classes Itinéraires et Feuille de routes servent à afficher les informations nécessaires à l’Utilisateur pour se déplacer. (Feuille de route n’étant pour l’instant pas utilisée dans notre application).

La classe Carte sert à la gestion du multi-niveau, en effet le but de notre projet est de gérer un itinéraire sur du multi-niveau.

Enfin la classe Utilisateur sert à gérer la mobilité de ce dernier (s’il peut ou non monter des escaliers).

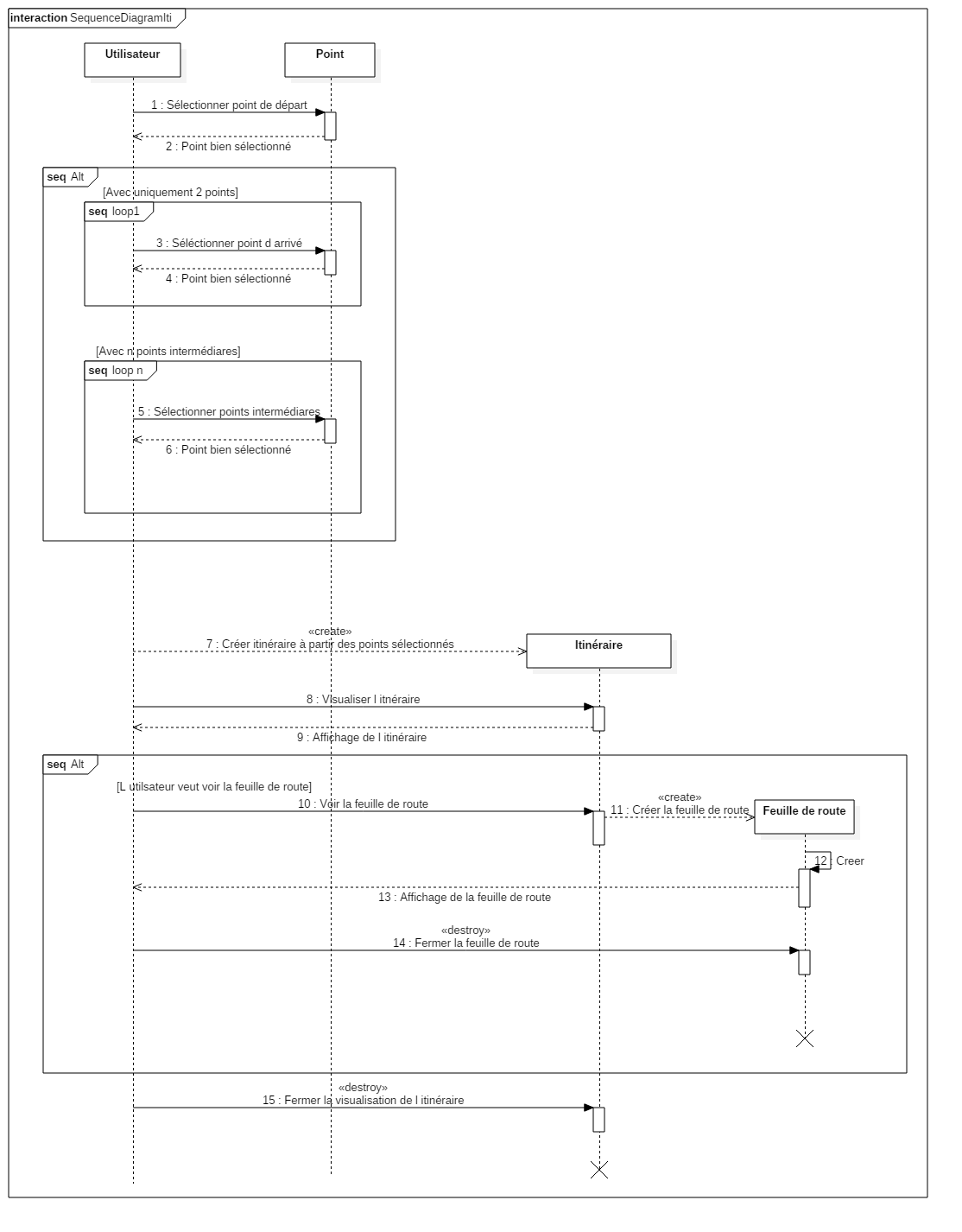
## Diagramme d’activité :

Ce diagramme présente le déroulement de la création d’un itinéraire sur notre application



Notez cependant que le choix de point intermédiaire et l’affichage de feuille de route n’est pas gérer dans la version actuelle de notre code.

## Diagramme de séquence :



Dans notre application, nous commençons par choisir un point de départ, puis nous choisissons le point d’arriver, (le choix de point intermédiaire n’étant pas encore implémenter). A ce moment le calcul d’itinéraire est effectué. Puis la feuille de route est créée et affichée (Non implémentée dans la version actuelle de notre application).

# Organisation et fonctionnement du code :

## Organisation général :

Nous allons vous présenter ici l’organisation général du code, le fonctionnement de chacune des parties décrites sera développée par la suite.

* Dans AndroidProjet\app\src\main\java\com\example\formation\androidprojet\_v1 nous avons les fichiers java qui gèrent le fonctionnement de notre application:
  + MainActivity.java : gère la majeure partie fonctionnement de notre application (récupération de la base de données, calcul et affichage d’itinéraire par niveaux, formulaire avec autocomplétion).
  + Choix.java, Listetype.java, Listemagasin.java : servent à gérer le formulaire par choix du type de magasin (et le point de départ par QrCode).
* Dans AndroidProjet\app\src\main\res\layout nous avons les fichiers liées à l’affichage des différentes activités, les différents fichiers correspondent aux différents fichiers java décrits ci-dessus.

## Fonctionnement des différentes parties :

### MainActivity.java :

#### 2.2.1.a OnCreate() :

* Nous commençons par gérer les éléments graphiques. La gestion des différentes parties s’effectue toujours de la même façon, nous allons expliquer le déroulement de la gestion d’un élément graphique complexe (une zone de saisie d’autocomplétion), les autres se gérant de la même façon :
* // Nous ouvrons ici un objet de AutoCompleteTextView défini dans le layout:
* textViewDep = (AutoCompleteTextView)  
   findViewById(R.id.*dep\_magasin*);
* // Nous récupèrons la valeur dans string.xml qui correspond au texte que l’on souhaite affiché par défaut dans la zone de saisie
* String depTxt = getResources().getString(R.string.*dep*);
* // Nous mettons le texte par défaut dans le zone de saisie
* textViewDep .setHint(depTxt);
* // Nous commençons la recherche automatique dès la première lettre écrite
* textViewDep .setThreshold(1);
* // Nous associons à la zone de saisie un listener
* textViewDep .setOnItemClickListener(new BoutonSaisieAutomatiqueDepListener());
* Nous ouvrons ensuite l’intégralité des cartes de fond qui seront affiché dans notre application et nous mettons leur visibilité à false pour les cache, ainsi les images sont toutes chargés et il suffit de changer la visibilité pour afficher les images de fond. Nous ajoutons de plus une couche graphique par-dessus les images de fond c’est à cet endroit que nous dessinerons nos points et notre itinéraire :
* // Retrieve the map and initial extent from XML layout  
  mMapView = (MapView) findViewById(R.id.*map*);  
  // Mise en place des fonds et visibilité = false :  
  mMapView.addLayer(mTileLayer0);  
  mTileLayer0.setVisible(false);  
  // Ajout couche graphique :  
  mMapView.addLayer(mGraphicsLayer);
* Nous gérons ensuite la définition des marqueurs des points d’arrivés et de départ :
* // Création symbole point départ/arrivé :  
  marqueur = getResources().getDrawable(R.drawable.*ic\_action\_marqueur*);  
  symStop = new PictureMarkerSymbol(marqueur);
* Enfin nous récupérons les informations dans la base de données à l’aide de la fonction accesBdd() dont le fonctionnement sera détaillée par la suite

#### 2.2.1.b onActivityResult(int requestCode, int resultCode, Intent intent) :

Fonction qui sert à la gestion des informations reçu de la part du QrCode et du formulaire

* Nous commençons par gérer les informations reçues lorsque qu’un QrCode est scanné. / Nous utilisons la classe IntentIntegrator et sa fonction parseActivityResult pour parser le résultat du scan.
* IntentResult scanningResult = IntentIntegrator.*parseActivityResult*(requestCode, resultCode, intent);

Si le résultat est non nul, alors nous traitons les informations obtenus par le scan du QrCOde. Nous récupérons le contenu, le format du code barre, et nous affichons le résultat dans nos TextView. Ensuite nous testons si le contenu de notre QrCode correspond à une valeur connue et si oui on ajoute pour le cas où il s’agit du Qr Code 01 un point sur la carte (ce point n’est pour l’instant pas ajouté au stop).

* // Nous récupérons le contenu du code barre :  
  String scanContent = scanningResult.getContents();  
  // Nous récupérons le format du code barre :  
  String scanFormat = scanningResult.getFormatName();
* if(scanContent.equals( "QR code 01" ) ){  
   // On marque la geometrie du QR code sur la carte  
   // Rappel, on teste avec le magasin "La grande recre"  
   Geometry projection = geomen.*project*(geom\_QR\_code, WKID\_RGF93, mapRef);  
   mGraphicsLayer.addGraphic(new Graphic(projection, new SimpleMarkerSymbol(Color.*RED*, 10, STYLE.*CROSS*)));}
* Nous gérons ensuite les informations renvoyé par le formulaire (normalement point de départ et point d’arrivé). Nous commençons par récupérer le nom des magasins sélectionnés dans le formulaire. Si il y a deux stops ou plus nous les réinitialisons. Nous récupérons à l’aide de leurs noms les géométries des magasins sélectionnés et nous les ajoutons au stop. Nous lançons ensuite le calcul d’itinéraire.
* // Récupération des noms du magasin de départ et d'arrivée  
  final String mag\_dep = intent.getStringExtra("Depart");
* final String mag\_arr = intent.getStringExtra("Arrivee");
* // Nous comptons le nombre de points présents dans mStops :  
  int tStop = mStops.getFeatures().size();
* // Si il y en a plus de deux nous réinitialisons les stops :  
  if( tStop >=2 ) {  
   mStops.clearFeatures();  
   clearAffich();} // Fonction expliquée plus bas
* // Nous retrouvons les points de départ et d'arrivé à l'aide de leurs noms dans la liste de magasin :  
  Geometry ptDep = trouverPtSel(mag\_dep, true);  
  depart = geomen.*project*(ptDep, WKID\_RGF93, mapRef);  
  ajouterPoint(depart, symStop);
* // Nous effectuons la même chose pour l’arrivée (non écrit ici)
* // On récupére à nouveau le nombre de stop :  
  tStop = mStops.getFeatures().size();  
  // Si on a 2 stops on calcule et on affiche l'itinéraire :  
  if( tStop >= 2) {  
   calculerIti(mapRef);  
   afficherPpv(mapRef);}

#### 2.2.1.c checkedListener :

Cette classe défini un listener utilisé pour modifié la valeur de estRestreint (mobilité réduite ou non), elle ne possède donc qu’une méthode onClick qui change cette valeur

* // Si la checkbox est cochée on met le booléen estRestraint à true, sinon à false  
  if (((CheckBox) v).isChecked()) {estRestreint = true;}  
  else {estRestreint = false;}

#### 2.2.1.d BoutonEtageListener :

Cette classe défini un listener sur une liste déroulante à choix, plus précisément sur le choix de l’étage. Cette classe implémente deux méthodes onItemSelected(AdapterView<?> parent, View view, int position, long id) et onNothingSelected(AdapterView<?> arg0).

* Nous ne faisons rien dans onNothingSelected.
* Dans onItemSelected, nous récupérons l’objet sélectionné et en fonction nous changeons les variables globales de visibilité des images de fond. Puis nous n’affichons que l’étage sélectionné et à l’aide de la fonction afficherIti(), nous affichons l’itinéraire au niveau sélectionné. Voici pour exemple ce qu’il se passe lorsque l’étage 0 est sélectionné.
* // On selecting a spinner item  
  String etageSelec = parent.getItemAtPosition(position).toString();  
    
  // Nous récupèrons les noms des étages qui sont stockés dans ressources.strings.values  
  String[] nom\_etage = getResources().getStringArray(R.array.*etage\_array*);  
    
  // Test suivant la sélection de l'utilisateur:  
  if (etageSelec.equals(nom\_etage[0])) {  
   etgsSelected = false;  
   etg0Selected = true;  
   etg1Selected = false;  
   etg2Selected = false;}
* // Nous affichons l'étage sélectionné :  
  mTileLayer.setVisible(etgsSelected);  
  mTileLayer0.setVisible(etg0Selected);  
  mTileLayer1.setVisible(etg1Selected);  
  mTileLayer2.setVisible(etg2Selected);

#### 2.2.1.e BoutonQRcodeListener :

Cette classe défini le listener utilisé sur le bouton de scan du QrCode, elle implémente une méthode onClick(View v). Cette méthode lance le scan par QrCode à l’aide d’un IntentIntegrator :

* // Nous lançon le scanner au clic sur notre bouton  
  IntentIntegrator integrator = newIntentIntegrator(MainActivity.this);  
  integrator.initiateScan();

#### 2.2.1.f BoutonSaisieAutomatiqueDepListener et BoutonSaisieAutomatiqueARrrListener :

Nous devons définir deux classes de listener pour les zones de saisie automatique de magasins (l’un pour le départ, l’autre pour l’arrivée), en effet récupérer facilement l'id de l’AutoCompleteTextView sur laquelle nous cliquons. Ces classes implémentent la méthodeonItemClick(AdapterView<?> parent, View view,int position, long id). Cette méthode fonctionne similairement dans les deux classes.