

Pick'UP - Conception détaillée

02.04.2021

Nel BOUVIER
Kesly GASSANT
Aleryc SERRANIA

Imad ROUAMI
Niels DE BARBANSON

Loïc DUBOIS-TERMOZ
Romain CHIKIROU

| | |
|---|----------|
| Introduction | 2 |
| Architecture générale | 3 |
| Structure de données | 3 |
| Models | 5 |
| Architectures spécifiques | 8 |
| Import XML | 8 |
| Algorithmes de calcul de chemin | 8 |
| Déterminer le chemin le plus court entre chaque points | 8 |
| Déterminer le circuit hamiltonien le plus court entre les points de passage | 9 |
| Calcul du crochet lorsqu'on ajoute une livraison | 10 |
| IHM | 10 |
| Les vues | 10 |
| Etat de la vue | 11 |

Introduction

Pick'Up est une application développée dans le cadre du projet Optimod'Lyon porté par le Grand Lyon. Ce projet vise à proposer de nouveaux services pour la mobilité urbaine (exemple : récolte et fourniture d'information en temps réel sur l'état du réseau urbain).

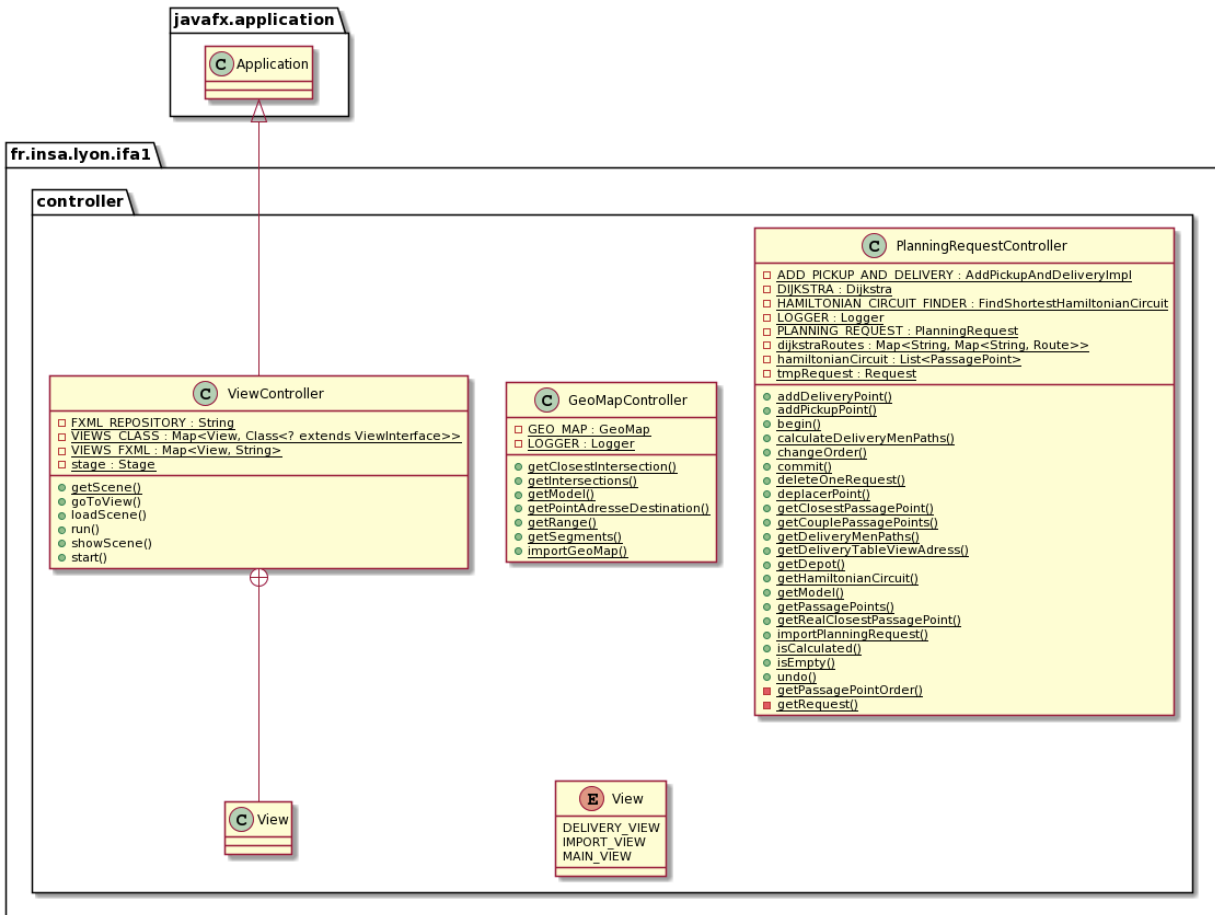
L'objectif de l'application Pick'Up est de permettre à des sociétés de livraison de préparer leurs tournées. Celle-ci est spécialisée dans le service de P&D (Pickup & Delivery) à vélo.

Ce document vise à présenter la conception détaillée de l'application Pick'Up. On présentera dans un premier temps l'architecture générale, puis les architectures spécifiques.

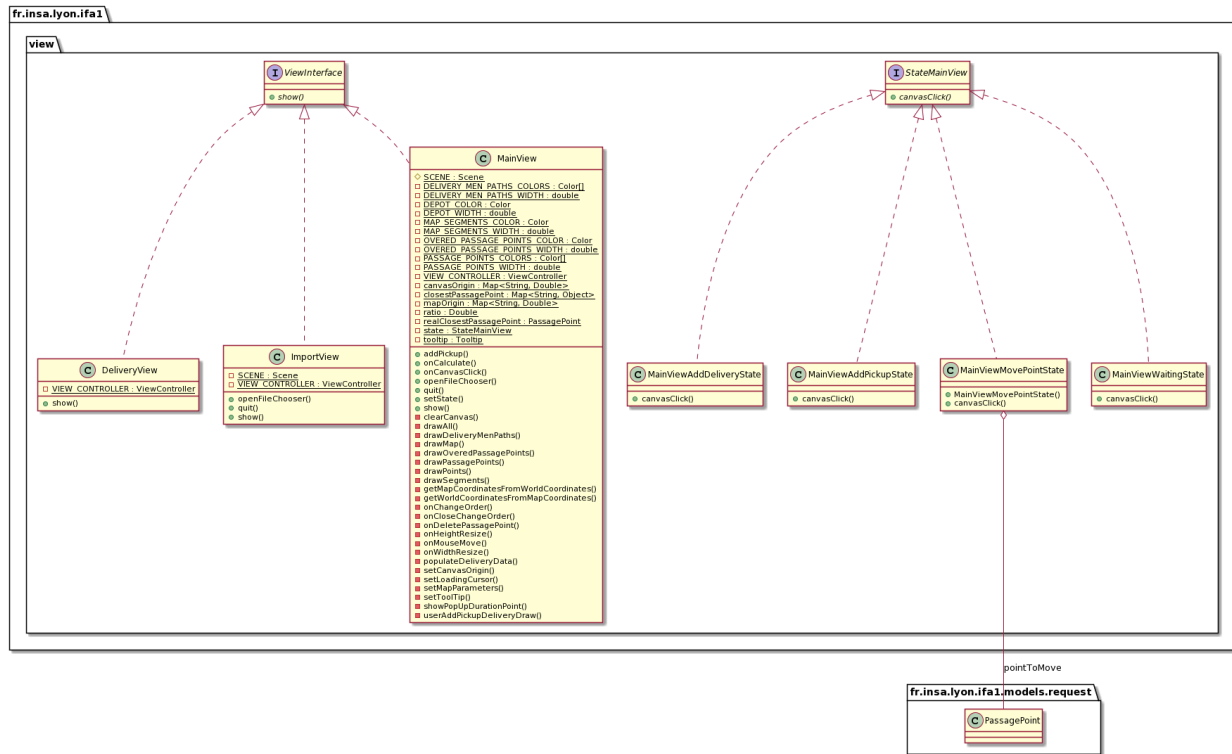
Architecture générale

1. Structure de données

CONTROLLER's Class Diagram

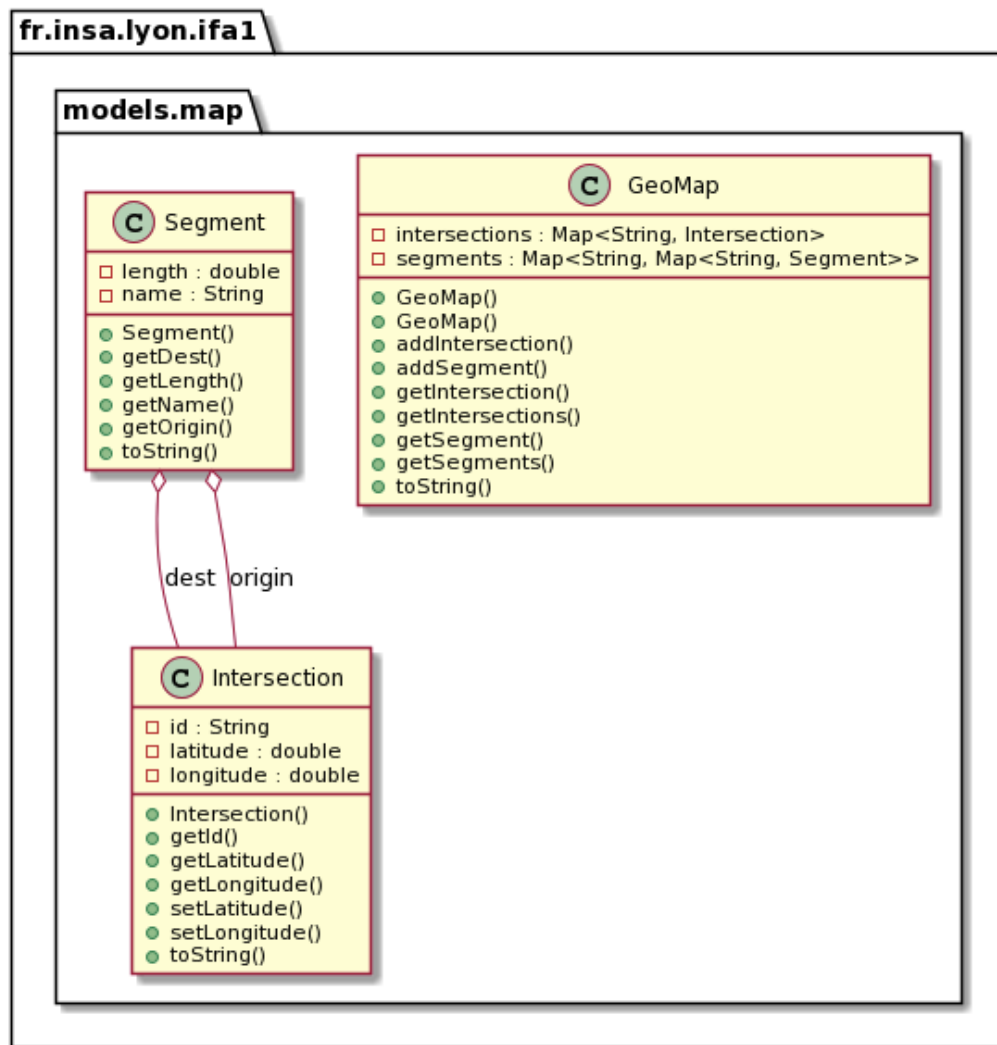


VIEW's Class Diagram

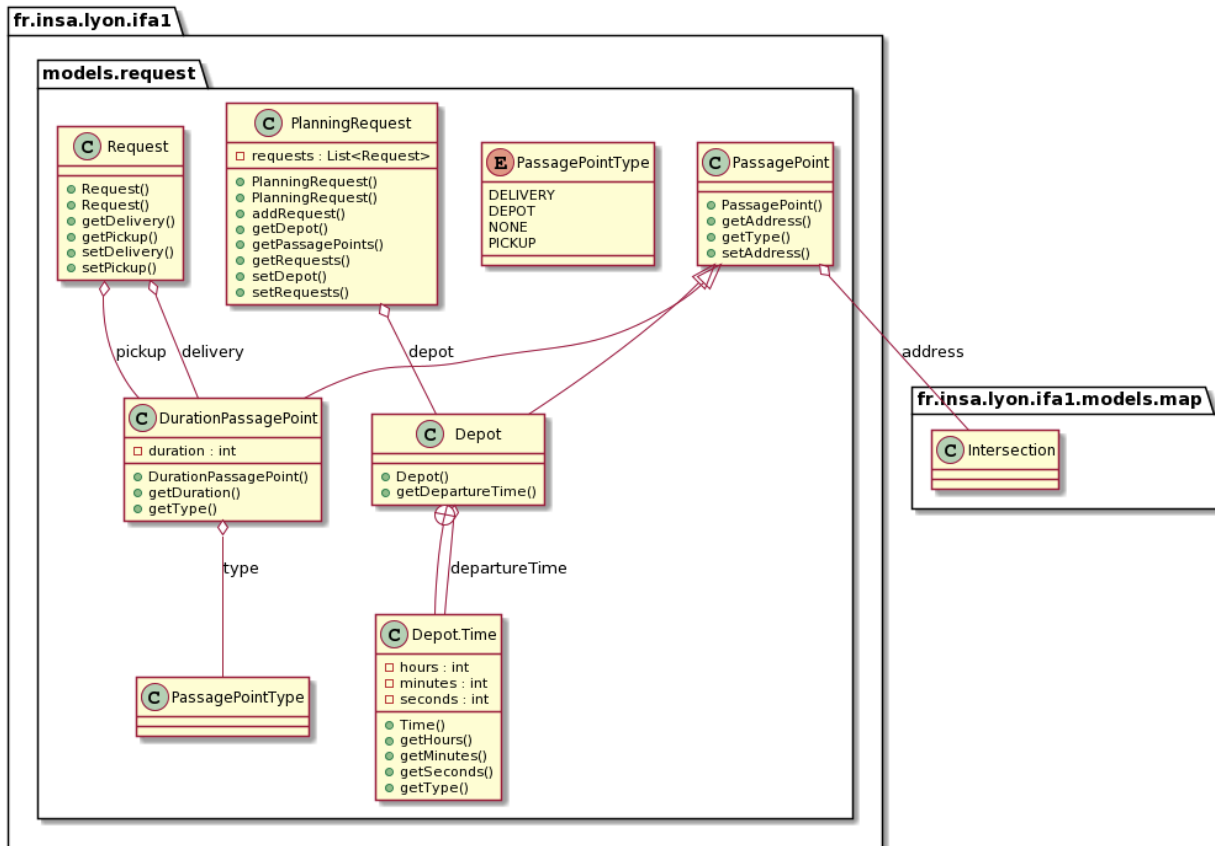


Models

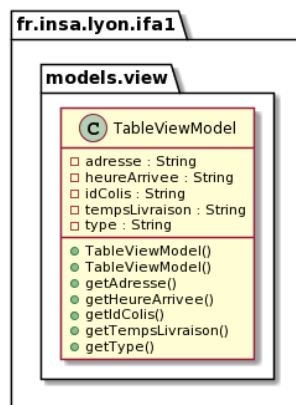
MAP's Class Diagram



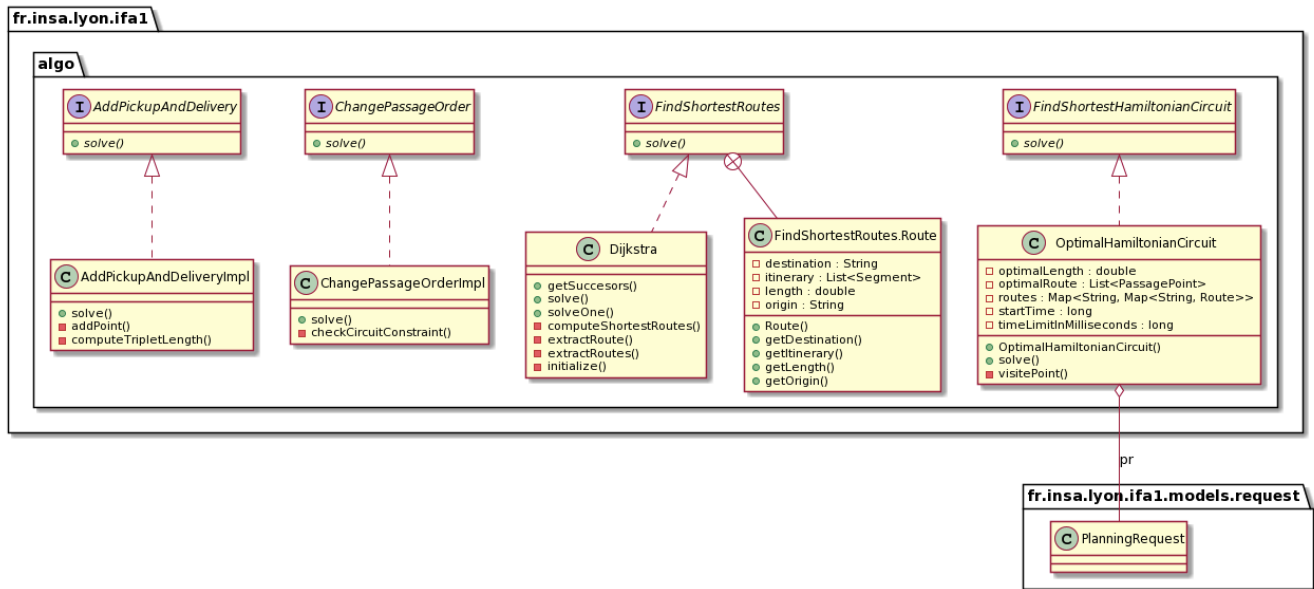
REQUEST's Class Diagram



VIEW's Class Diagram



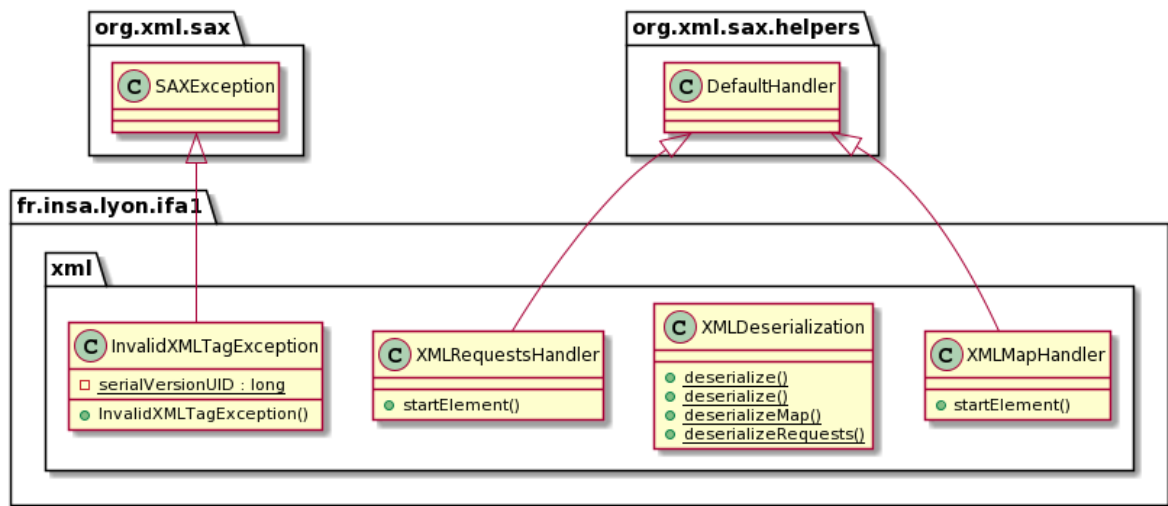
ALGO's Class Diagram



Architectures spécifiques

1. Import XML

XML's Class Diagram



PlantUML diagram generated by SketchIt! (<https://bitbucket.org/pmesmeur/sketch.it>)
For more information about this tool, please contact philippe.mesmeur@gmail.com

XMLDeserialization est le point d'entrée pour lancer la désérialisation d'un fichier XML map.xml (méthode deserializeMap) ou requests.xml (deserializeRequests). La désérialisation se fait à l'aide de la librairie SAX. SAX est mieux adapté à JAX dans notre cas puisqu'il parcourt le fichier progressivement sans stocker les lignes progressivement contrairement à JAX qui stocke intégralement l'arbre défini (ce qui peut être bloquant pour des fichiers de tailles trop conséquentes).

XMLRequestsHandler et XMLMapHandler permettent l'interprétation de leur type de fichier respectif.

2. Algorithmes de calcul de chemin

a. Déterminer le chemin le plus court entre chaque points

Deux choix d'algorithmes :

- **Algorithme de Dijkstra** : permet de calculer les chemins les plus courts entre un point et toutes les intersections du plan de la ville
 - Avantages : Uniquement le calcul des chemins nécessaires à la création du circuit hamiltonien le plus court

- Inconvénients : Lors de l'ajout ou du déplacement de points de passage, on est obligé de recalculer les chemins les plus courts entre ce nouveau point et tous les autres
- Complexité temporelle (Big O notation) : $O((I + T).log(I).P)$ avec :
 - I : le nombre d'intersections
 - T : le nombre de tronçons
 - P : le nombre de points de passage
- **Algorithme de Floyd-Warshall** : permet de calculer les chemins les plus courts entre toutes les intersections
 - Avantages : Puisque tous les chemins les plus courts entre toutes les intersections ont été calculés, aucun calcul supplémentaire n'est nécessaire pour l'ajout ou le déplacement d'un point de passage. Algorithme plus facile à implémenter que Dijkstra.
 - Inconvénients : Complexité temporelle élevée
 - Complexité temporelle (Big O notation) : $O(I^3)$ avec :
 - I : le nombre d'intersections

Le nombre d'intersections étant extrêmement élevé pour la largeMap et le nombre de modifications sur les points de passage étant jugées peu élevées en comparaison de ce nombre d'intersections, les avantages de l'algorithme de Floyd-Warshall nous ont paru assez peu significatifs.

b. Déterminer le circuit hamiltonien le plus court entre les points de passage

Pour cette itération (n^2) :

- **Recherche optimale** avec limitation du temps de recherche pour éviter le temps de calcul de complexité factorielle (pour 18 points de passages, on doit chercher le circuit le plus court parmi $18! = 6\,402\,373\,705\,728\,000$ circuits...)
- Avantages : Trouve le chemin optimal. Tests complets contrairement à une heuristique.
- Inconvénients : Complexité temporelle beaucoup trop élevée

Nous n'avons pas encore eu le temps d'explorer des heuristiques lors de l'itération n^2 , mais les résultats donnés avec la largeMap et les 9 requests donnent des résultats satisfaisants avec la limitation du temps.

Un problème qui se pose avec les heuristiques est la conception de tests unitaires : les tests sont bien plus complexes à réaliser et dépendent surtout de l'heuristique à implémenter. La seule partie stable que l'on peut tester est si notre heuristique passe bien par tous les points de passage en respectant l'ordre de précedence des Pickups et Deliveries.

c. Calcul du crochet lorsqu'on ajoute une livraison

Lors de l'ajout du pick up, nous utilisons l'algorithme de Dijkstra pour calculer les chemins les plus courts entre ce point et les autres. Ainsi nous trouvons le point de passage le plus proche du nouveau point de pick up. Nous essayons d'ajouter le point de pick avant et après ce point de passage et recalcul le circuit pour les deux cas de figure et sélectionne la meilleure.

L'ajout du delivery se fait de la même façon mais en prenant en compte la contrainte de précedence pour le l'ancien point de pick up.

Cet algorithme ne trouve pas le chemin optimal puisqu'on traite le point de delivery après avoir traiter l'ajout du point de pick up.


Nb : Pour la suppression d'une livraison, le circuit n'est pas recalculé, nous utilisons les chemins le plus courts calculés avec l'Algorithme de Dijkstra auparavant. Exemple : si l'on a un circuit A -> B -> C -> D -> E -> F, et supprimons les points de passage C et D (correspondant à une livraison). Nous utilisons le chemin le plus court entre B et E.

3. IHM

a. Les vues

L'application est composée de 2 vues :

- ImportView : Vue d'import de la carte de lyon
- MainView : Vue principale de l'application qui affiche la carte et les données des fichiers XML



Les composants statiques de ces ihm ont été implémentés via des fichiers fxml. Ces fichiers décrivent dans un formalisme particulier comment la vue doit être affichée. Un fichier FXML est lié à une classe qui va gérer l'aspect dynamique de l'ihm et l'interaction avec l'utilisateur. Chaque interaction entre l'utilisateur et l'ihm va alors nécessiter l'appel à des contrôleurs particuliers.

La vue principale est composée de plusieurs canvas qui vont gérer l'affichage de la carte, des points de livraisons ainsi que tout élément interactif. A chaque traitement : ajout d'un point de livraison, déplacement d'un point, calcul d'un trajet etc ... Les différents canvas sont mis à jour avec les données calculées et retournées par les contrôleurs.

Les vues ont été réfléchies de manière à ce qu'elles soient responsives en utilisant des layout adéquats. Lors du redimensionnement de la fenêtre applicative, le canva est redessinée de manière à ce qu'il s'adapte à l'écran.

b. Etat de la vue

Nous avons mis en place le design pattern State afin de gérer facilement les différents états de la vue principale. On dénote 4 états :

- WaitingState : État courant de la vue
- AddPickupState : État dans lequel l'utilisateur ajoute un point de livraison de type "pickup"
- AddDeliveryState : État dans lequel l'utilisateur ajoute un point de livraison de type "delivery"
- MoveStateDelivery : État dans lequel l'utilisateur déplace un point de livraison

Ces différents états implémentent une méthode qui gère l'affichage de la vue en fonction des besoins utilisateurs. Certains composants de la vue seront ainsi indisponible ou vont interagir de manière différentes avec l'utilisateur.