

ECOLE POLYTECHNIQUE DE L'UNIVERSITÉ FRANÇOIS RABELAIS DE TOURS

Département Informatique

64 avenue Jean Portalis

37200 Tours, France

Tél. +33 (0)2 47 36 14 14

[polytech.univ-tours.fr](http://polytech.univ-tours.fr)

## Projet Recherche & Développement 2018-2019

# Automate pour logiciel d'affectation de véhicules en temps réel

**Entreprise**

Keolis

**Tuteur entreprise**

Sylvain FABRE

**Étudiant**

Victor COLEAU (DI5)

**Tuteur académique**

Yannick KERGOSIEN

30 mars 2019



# Liste des intervenants

## Entreprise

Keolis Impasse de Florence, 37200 Tours <a href="http://www.keolis.com/">www.keolis.com/</a>
--

Nom	Email	Qualité
Victor COLEAU	<a href="mailto:victor.coleau@etu.univ-tours.fr">victor.coleau@etu.univ-tours.fr</a>	Étudiant DI5
Yannick KERGOSIEN	<a href="mailto:yannick.kergosien@univ-tours.fr">yannick.kergosien@univ-tours.fr</a>	Tuteur académique, Département Informatique
Sylvain FABRE	<a href="mailto:sylvain.fabre@keolis.com">sylvain.fabre@keolis.com</a>	Tuteur entreprise



# Avertissement

Ce document a été rédigé par Victor Coleau susnommé l'auteur.

L'entreprise Keolis est représentée par Sylvain Fabre susnommé le tuteur entreprise.

L'Ecole Polytechnique de l'Université François Rabelais de Tours est représentée par Yannick Kergosien susnommé le tuteur académique.

Par l'utilisation de ce modèle de document, l'ensemble des intervenants du projet acceptent les conditions définies ci-après.

L'auteur reconnaît assumer l'entière responsabilité du contenu du document ainsi que toutes suites judiciaires qui pourraient en découler du fait du non respect des lois ou des droits d'auteur.

L'auteur atteste que les propos du document sont sincères et assument l'entière responsabilité de la véracité des propos.

L'auteur atteste ne pas s'approprier le travail d'autrui et que le document ne contient aucun plagiat.

L'auteur atteste que le document ne contient aucun propos diffamatoire ou condamnable devant la loi.

L'auteur reconnaît qu'il ne peut diffuser ce document en partie ou en intégralité sous quelque forme que ce soit sans l'accord préalable du tuteur académique et de l'entreprise.

L'auteur autorise l'école polytechnique de l'université François Rabelais de Tours à diffuser tout ou partie de ce document, sous quelque forme que ce soit, y compris après transformation en citant la source. Cette diffusion devra se faire gracieusement et être accompagnée du présent avertissement.



## Pour citer ce document

Victor Coleau, *Automate pour logiciel d'affectation de véhicules en temps réel*, Projet Recherche & Développement, Ecole Polytechnique de l'Université François Rabelais de Tours, Tours, France, 2018-2019.

```
@mastersthesis{
  author={Coleau, Victor},
  title={Automate pour logiciel d'affectation de véhicules en temps réel},
  type={Projet Recherche \& Développement},
  school={Ecole Polytechnique de l'Université François Rabelais de Tours},
  address={Tours, France},
  year={2018-2019}
}
```

# Table des matières

Liste des intervenants	a
Avertissement	b
Pour citer ce document	c
Table des matières	i
Table des figures	v
<b>1 Introduction</b>	<b>1</b>
<b>2 Contexte de réalisation</b>	<b>2</b>
1 Contexte .....	2
2 Objectifs .....	2
3 Base méthodologique .....	2
<b>3 Description générale</b>	<b>4</b>
1 Environnement du projet .....	4
2 Caractéristiques des utilisateurs .....	5
3 Fonctionnalités du système .....	6
4 Condition de fonctionnement .....	6
5 Structure générale du système .....	6
<b>4 État de l'art</b>	<b>8</b>
1 Introduction.....	8
2 Algorithmes dans le milieu du transport.....	8
2.1 Vehicule Routine Problem .....	8

2.2	Fixed Interval Scheduling.....	9
2.3	Dispatching Buses.....	11
3	Bilan .....	12
<b>5</b>	<b>Analyse et conception</b>	<b>13</b>
1	Algorithme choisi .....	13
2	Conception et modélisation .....	15
2.1	Parser.....	16
2.2	Model .....	16
2.3	Événements .....	17
2.4	Core.....	18
2.5	Tests.....	19
2.6	Résultats.....	20
2.7	Diagramme de classes.....	20
<b>6</b>	<b>Mise en œuvre</b>	<b>21</b>
1	Environnement .....	21
2	Structure de données.....	21
2.1	Algorithme événementiel à liste.....	21
2.2	Gestion des données .....	22
3	Algorithme de selection du véhicule.....	22
4	Algorithme de selection de la place de parking.....	23
<b>7</b>	<b>Bilan et conclusion</b>	<b>25</b>
1	Bilan du S9.....	25
2	Conclusion .....	25
	<b>Annexes</b>	<b>26</b>
<b>A</b>	<b>Cahier des charges</b>	<b>27</b>
1	Description du besoin.....	27
1.1	Représentation des données .....	27
1.2	Contraintes.....	29
<b>B</b>	<b>Cahier de spécifications</b>	<b>30</b>
1	Entrée et sortie de l'automate.....	30
1.1	Mission.....	30
1.2	Vehicule.....	31
1.3	Immobilisation.....	31
1.4	Cible_Kilométrique .....	31

1.5	Emplacement .....	32
1.6	Trajet .....	32
2	Contraintes liées aux données .....	32
2.1	Contraintes fortes .....	33
2.2	Contraintes faibles .....	34
2.3	Contraintes relaxables .....	34
3	Fonction objectif .....	34
<b>C</b>	<b>Cahier de l'utilisateur</b> .....	<b>36</b>
1	Installation .....	36
2	Utilisation .....	36
<b>D</b>	<b>Cahier du développeur</b> .....	<b>37</b>
1	Environnement de développement .....	37
2	Structure du programme .....	37
3	Description des classes .....	38
3.1	Diagramme de classes .....	38
3.2	Détails des classes .....	40
3.2.1	Classe EventListAlgorithm .....	40
3.2.2	Classe Data .....	41
3.2.3	Classe Parking .....	41
3.2.4	Les événements .....	42
3.2.5	Le module de validation .....	43
4	Description des fichiers d'entrée .....	43
5	Description du fichier de sortie .....	45
<b>E</b>	<b>Cahier de tests</b> .....	<b>47</b>
1	Tests unitaires .....	47
1.1	DateTest .....	47
1.2	PlaceTest .....	48
1.3	ParkingTest .....	48
2	Tests de validation .....	50
2.1	Petites instances .....	50
2.2	Grandes instances .....	50
2.3	Instances personnalisées .....	51
2.4	Instances réelles .....	52
<b>F</b>	<b>Gestion de projet</b> .....	<b>53</b>
1	Diagramme de Gantt .....	53
2	Initialisation .....	53

3	Définition du besoin.....	53
4	Analyse .....	54
5	Conception.....	54
6	Développement.....	54
7	Tests .....	54
8	Mise en place .....	55
9	Diagramme de Gantt revisité.....	55
<b>Comptes rendus hebdomadaires</b>		<b>56</b>



# Table des figures

## 2 Contexte de réalisation

1	Diagramme du modèle en « cascade » .....	3
---	--	---

## 3 Description générale

1	Diagramme de séquence avant l'intégration de l'automate .....	4
2	Diagramme de séquence après l'intégration de l'automate .....	5
3	Diagramme de composants .....	7

## 4 État de l'art

1	Graphe d'un problème Fixed Interval Scheduling .....	11
---	--	----

## 5 Analyse et conception

1	Diagramme de classe partiel du package Parser.....	16
2	Diagramme de classe partiel du package Model .....	17
3	Diagramme de classe partiel du package Evenement .....	18
4	Diagramme de classe partiel du package Core .....	19
5	Diagramme de classe partiel du package Test .....	20
6	Diagramme de classe partiel du package Results.....	20

## D Cahier du développeur

1	Diagramme de package simplifié de l'automate.....	38
2	Diagramme de classe de l'automate.....	39
3	Exemple de fichier CSV des véhicules .....	44
4	Exemple de fichier CSV des missions .....	44

5	Exemple de fichier CSV des emplacements.....	44
6	Exemple de fichier CSV des cibles kilométriques .....	44
7	Exemple de fichier CSV des immobilisations .....	44
8	Exemple de fichier CSV des trajets .....	44
9	Exemple de fichier actuel de résultats .....	45
10	Exemple de fichier de résultats tel qu'attendu .....	46
 <b>F Gestion de projet</b>		
1	Diagramme de Gantt initialement prévu.....	53
2	Diagramme de Gantt avec niveaux de complétion .....	55

# 1

## Introduction

Ce document présente la conception et le développement d'un automate d'affectation de véhicules à des missions, réalisé dans le cadre d'un PRD (Projet de Recherche & Développement) intitulé "Automate pour logiciel d'affectation de véhicules en temps réel".

Ce projet a été proposé par l'entreprise Keolis et est encadré par Mr Yannick Kergosien, maître de conférence à Polytech Tours, et Mr Sylvain Fabre, tuteur entreprise.

Ce projet de fin d'étude fait suite à un précédent PRD, réalisé par Charly Moreau, ancien étudiant de Polytech Tours. Ce document reprend donc certaines parties du cahier de spécifications système qui avait été rédigé lors de ce projet ainsi que certaines idées et analyses émises par Charly Moreau.

# 2

## Contexte de réalisation

### 1 Contexte

L'entreprise Keolis est actuellement en charge du réseau de bus et de tram de la ville de Tours. A ce titre elle planifie quotidiennement la répartition des véhicules disponibles sur les différentes lignes du réseau ainsi que le remisage des dits véhicules.

Au sein de Keolis Tours, l'attribution des missions quotidiennes et le remisage des véhicules est aujourd'hui conduit à travers le logiciel LGV. Il offre la possibilité d'affecter les véhicules et les services aux emplacements du parc.

Cette tâche est cependant manuelle, et par conséquent fastidieuse, et ne permet pas une prise en compte optimale de toutes les contraintes associées à chacune des missions.

Ainsi, Keolis Tours souhaite développer un outil automatique permettant de réaliser ce travail.

### 2 Objectifs

L'objectif de ce projet est donc de concevoir et de développer un automate qui réalisera l'affectation des véhicules à leur mission quotidienne ainsi qu'à un emplacement de remisage.

Cet automate devra donc prendre en compte toutes les contraintes du parc que nous allons détailler dans ce document et proposer une solution respectant au mieux les exigences du réseau.

Néanmoins, il ne doit en aucun cas être indispensable. Il doit pouvoir être coupé à tout moment en cas d'imprévu. Pour ce faire, l'automate sera force de proposition pour le service d'exploitation de Keolis Tours en proposant une solution pour chacun des jours de la planification.

### 3 Base méthodologique

Le développement de cet automate faisant suite à un autre projet, il fut d'abord envisagé de reprendre une partie du code précédemment produit. Cela eut permis une économie de temps car certaines parties du code source peuvent être copiées sans modification. Le langage de programmation choisi aurait donc été le même : C++.

Cependant, au vu du nombre de modifications à réaliser dans l'ensemble du programme et de mes compétences personnelles dans les différents langages, le Java fut finalement choisi.

En ce qui concerne la modélisation des fonctionnalités du système, le langage UML sera privilégié.

Enfin, concernant la gestion de projet, un modèle en « cascade » permettra de visualiser l'avancée du projet (Figure 1). Ce type de modèle est composé d'étapes successives dont l'échéance est prévue et datée à l'avance.

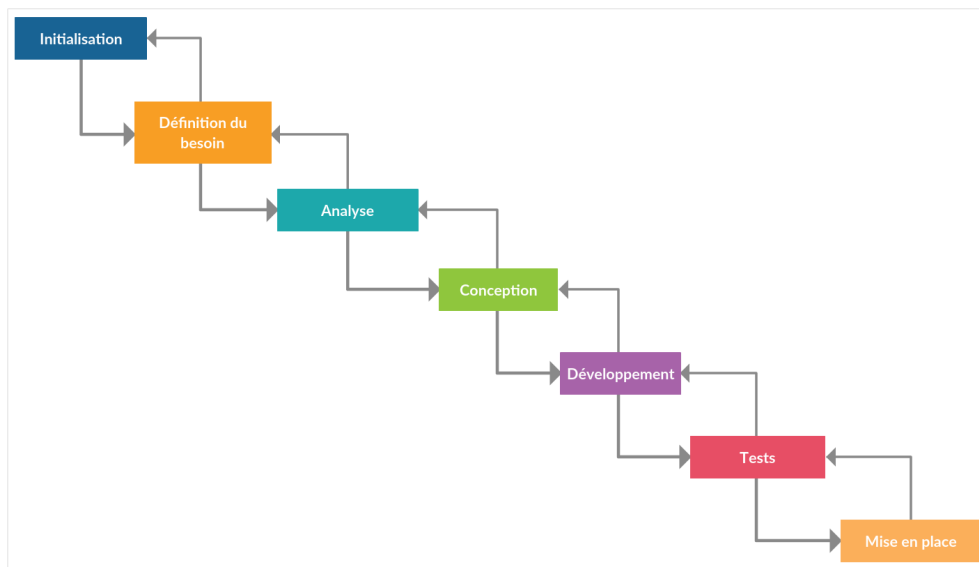


Figure 1 – Diagramme du modèle en « cascade »

A la fin de chaque étape, une série de vérifications propres à la tâche en cours est réalisée. Ces vérifications couvrent non seulement les performances de la tâche en elle-même mais aussi sa cohérence avec la précédente et effectuera une rétroaction si nécessaire. Si les résultats sont jugés satisfaisants, l'étape est validée et on peut entamer la suivante.

Ce type de gestion de projet est particulièrement adapté au PRD puisqu'il permet une validation progressive d'étapes identifiées en amont à des dates précises. De plus, le fait qu'il s'agisse d'un projet réalisé individuellement limite les effets négatifs et retards en cas de rétroaction à une étape précédente.

# 3

## Description générale

### 1 Environnement du projet

L'automate développé s'intégrera au sein du logiciel LGV existant. Il se situera en amont de l'affichage utilisateur.

Actuellement, toutes les données concernant les missions, véhicules, places de stationnement, etc. sont stockées dans une base de données. Au lancement du logiciel, elles en sont extraites afin d'être présentées de manière graphique à l'utilisateur.

C'est à partir de cette interface que l'utilisateur peut planifier une journée.

L'automate viendra se positionner entre ces deux composantes. Il intégrera les informations que la base de données retourne, puis, après traitement, renverra son résultat, qui sera alors présenté à l'utilisateur en lieu et place des informations brutes.

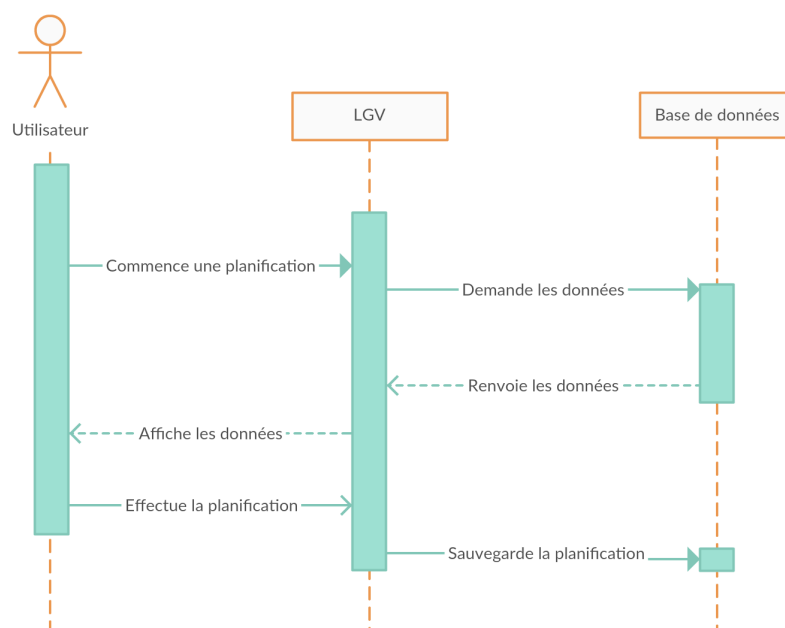


Figure 1 – Diagramme de séquence avant l'intégration de l'automate

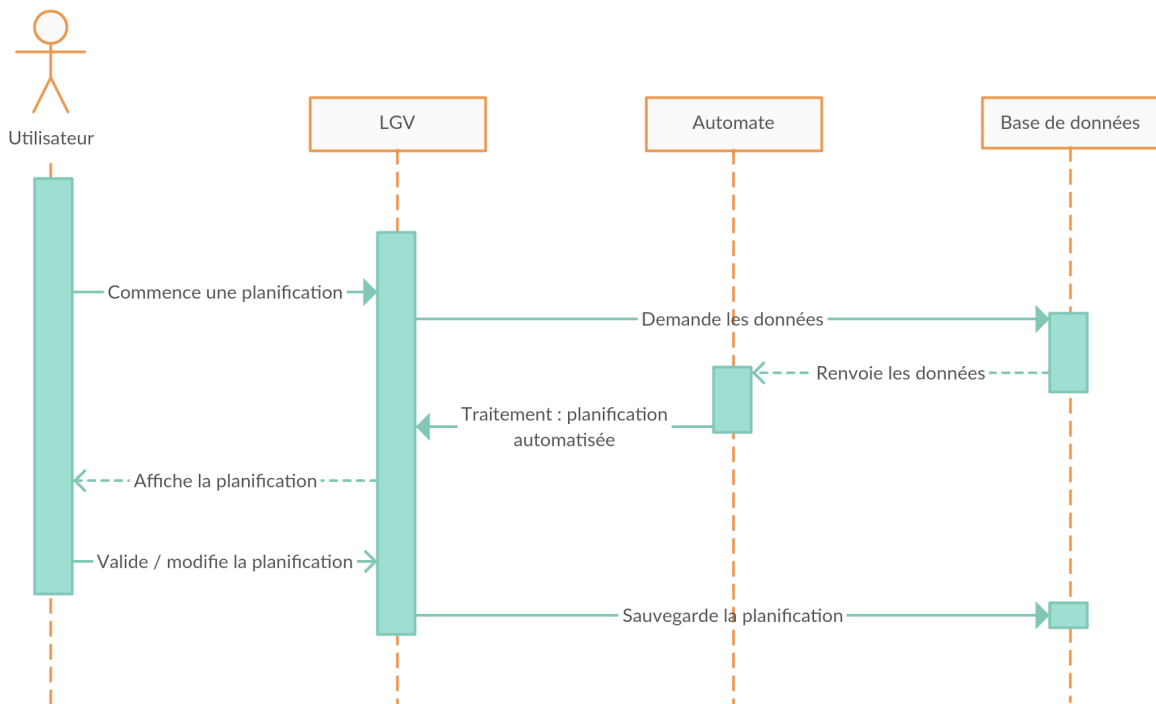


Figure 2 – Diagramme de séquence après l'intégration de l'automate

Comme visible sur les figures précédentes, l'automate devra effectuer le travail de planification à la place de l'utilisateur. Ce dernier n'aura alors plus qu'à valider ce travail. Néanmoins le logiciel LGV de Keolis dans sa version actuelle permet la modification de la planification proposée par l'automate. Celui-ci n'est donc pas une force de décision mais uniquement de conseil.

Dans le cas où l'automate ne serait pas appelé (peu importe la raison), le système reprendrait son ancien état sans automate, dans lequel l'attribution des véhicules est à la charge de l'utilisateur humain.

## 2 Caractéristiques des utilisateurs

L'objectif de ce projet étant la réalisation d'un automate, par définition autonome, la notion d'utilisateur est ici très réduite.

On peut toutefois considérer les utilisateurs actuels du logiciel LGV comme des utilisateurs de l'automate puisque ceux-ci manipuleront les résultats retournés par l'automate.

Actuellement, ces utilisateurs affectent manuellement chaque véhicule à une mission afin de remplir un planning quotidien. Lorsque l'automate sera en place, ses résultats serviront de base au planning.

Les utilisateurs n'auront alors plus qu'à valider le planning proposé si celui-ci remplit toutes les conditions requises. Dans le cas où la solution proposée par l'automate n'est pas valide, les utilisateurs devront le modifier afin de le rendre acceptable.

L'interface graphique utilisateur n'étant pas affectée par la présence ou non de l'automate, modifier un planning pré-construit ne demandera aucune formation supplémentaire par rapport à la manipulation du logiciel LGV actuel.

### 3 Fonctionnalités du système

#### Communication avec LGV

Les informations brutes fournies à l'automate seront sous la forme des fichiers CSV. L'automate devra donc pouvoir lire et interpréter ces fichiers.

De la même façon, afin de pouvoir transmettre sa solution au reste du système, l'automate devra produire lui-même un fichier CSV formaté de telle sorte que le logiciel LGV puisse le comprendre.

#### Résolution

Par la suite, l'automate devra répondre à deux problématiques plus ou moins liées :

Premièrement, à chaque mission doit être associée un véhicule dont les caractéristiques correspondent aux attentes de la mission. (Les différents types de missions - service et immobilisation - seront expliquées par la suite)

Deuxièmement, entre chaque mission, les véhicules doivent être placés dans le parc de stationnement. Cependant, les emplacements ne sont pas indépendants : entrer et sortir d'un emplacement de stationnement nécessite de passer par une ou plusieurs autres places de stationnement, devant donc être vides.

Il s'agit donc d'un problème d'affectation de ressources à des tâches, chacune des tâches demandant des caractéristiques précises que les ressources devront avoir afin d'y être associées.

Théoriquement, l'automate devra pouvoir planifier jusqu'à trois jours. Dans les faits, il sera exécuté plus souvent que tous les trois jours et, suivant les aléas du réseau, réadaptera sa solution qui pourra donc ne plus correspondre à l'ancienne planification.

### 4 Condition de fonctionnement

Comme décrit dans la [Section 2](#) (Chapitre 2), l'automate développé ne devra être qu'une aide aux utilisateurs humains. L'intégration de ce nouveau système au sein de l'existant respectant ce critère de non-nécessité sera à la charge du logiciel LGV. En effet, l'automate lui-même n'a pas le contrôle sur son environnement, il revient à ce dernier de faire appel ou non à l'automate et à se servir des résultats proposés.

Cependant, afin de rendre son utilisation invisible pour les utilisateurs, l'automate devra respecter les formats d'entrée des données (lecture) et surtout le format de sortie des données. De ce fait, le logiciel LGV pourra intégrer directement les propositions de l'automate sans action (de formatage ou autre) supplémentaire de la part d'un humain.

Enfin, le temps d'exécution de l'automate ne devra pas excéder quelques secondes afin de permettre une utilisation régulière et répondre rapidement aux aléas du réseau.

### 5 Structure générale du système

Afin de présenter la structure interne du logiciel, on conçoit un diagramme de composants ([Figure 3](#)).



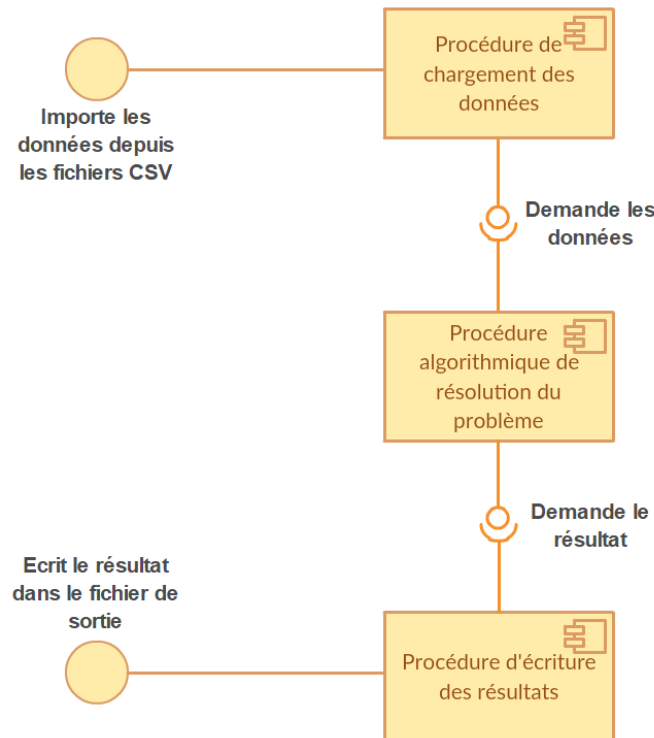


Figure 3 – Diagramme de composants

Ce diagramme se compose de trois composants principaux : deux procédures d'entrée/sortie et une de résolution.

Bien que le système global contienne une base de données, l'automate n'y aura pas accès directement. Afin de lui fournir les données nécessaires, un autre programme a été développé par Keolis. Le-dit programme exécutera un certain nombre de requêtes afin d'extraire les données demandées par l'automate et les fournira à celui-ci par le biais de fichiers CSV. Le premier module « Procédure de chargement des données » sera donc une procédure de lecture de fichier.

La deuxième procédure « Procédure algorithmique de résolution du problème » est la plus importante du système car c'est elle qui résoudra le problème d'attribution des véhicules aux missions.

La troisième et dernière procédure « Procédure d'écriture des résultats » recueillera le résultat fourni par la précédente et le formatera afin de fournir un fichier de sortie compréhensible par la suite du système (logiciel LGV).

# 4

## État de l'art

### 1 Introduction

Ce chapitre présente un ensemble de travaux réalisés dans le domaine des algorithmes de résolution de problèmes dans le milieu du transport.

Afin de bien cerner leur pertinence dans le cas du problème de Keolis, il est important de définir leurs hypothèses restrictives de base, leur complexité ainsi que leur champ d'action.

Il est important de noter que le sujet de Keolis est particulier car il regroupe deux sous-problématiques : le premier est un problème d'affectation de ressources limitées et le second est un problème de rangement dans un entrepôt limité et contraint. Il est hautement improbable de trouver un article parlant d'un double-sujet similaire (encore plus si l'on tient compte de toutes les particularités propres à Keolis pour chacun des sous-problèmes). Les recherches effectuées se sont donc concentrées sur l'un ou l'autre des problèmes.

### 2 Algorithmes dans le milieu du transport

Les articles traitant de problèmes liés au milieu du transport sont nombreux car faisant partie des sujets les plus traités dans la littérature de Recherche Opérationnelle. Ci-dessous sont présentés trois articles dont l'étude peut se rapprocher des problématiques de Keolis.

#### 2.1 Vehicule Routine Problem

Lors de l'étude d'une problématique en rapport avec le transport, le premier exemple-type de problème classique de la littérature venant à l'esprit est celui du voyageur de commerce.

Le Vehicule Routine Problem (ou Problème de Tournées de Véhicules en français) est un dérivé du voyageur de commerce. Son principe de base consiste à déterminer les tournées d'une flotte de véhicules afin de desservir un ensemble de clients tout en minimisant les coûts (la notion de coûts est à définir au cas par cas, suivant les demandes de l'utilisateur final de l'algorithme). Il s'agit d'un problème NP-complet comme d'habitude dans l'article [0].

A cette définition de base du problème on peut ajouter une ou plusieurs contraintes :

- **Contraintes de capacité :** Les véhicules ont une capacité d'emport maximum (quantité, taille, poids, etc.), ce qui limite le nombre de clients qu'ils peuvent servir en un seul trajet.
- **Contraintes liées aux ressources et aux clients :** Disponibilité, localisation, compétences requises, etc. doivent être respectées lors de la définition de tournées.
- **Tournées de véhicules avec fenêtre de temps :** Pour chaque client on impose une fenêtre de temps dans laquelle la livraison doit être effectuée.
- **Tournées de véhicules avec collecte et livraison :** Avant d'être desservies aux clients, les marchandises doivent être collectées d'un autre site.

La résolution de tels problèmes peut rapidement devenir compliquée lorsque les contraintes se cumulent. Comme tous les problèmes NP-complet, trouver une solution optimale unique sur une grande instance est, bien que théoriquement possible, quasiment impossible dans les faits. Il est donc recommandé de commencer par chercher une solution valide grâce à une heuristique puis d'affiner celle-ci grâce à une (ou des) méta-heuristique(s). Cette méthode permet généralement d'obtenir une « bonne solution » dans un temps de calcul raisonnable.

En fonction des paramètres de base et des contraintes ajoutées, de nombreux algorithmes furent développés pour répondre à une multitude de variantes de ce problèmes. Ceux-ci ne seront cependant pas développés dans ce document.

Malgré sa popularité, ce type de problème ne peut être étudié dans le cas de ce projet car la problématique soumise est trop éloignée de celle proposée par Keolis.

En effet, un problème de type Tournées de Véhicules correspondrait à la définition des lignes de bus dans la ville, cherchant à desservir de manière « optimale » tous les arrêts de bus. Cette problématique a cependant déjà été résolue dans notre cas.

## 2.2 Fixed Interval Scheduling

Le Fixed Interval Scheduling (ou Ordonnancement à Intervalle Fixé en français) est un problème qui peut être défini comme suit.

On dispose de  $n$  tâches non préemptives chacune définie par un intervalle de temps pendant lequel elles doivent être exécutées. On dispose aussi de  $m$  machines indépendantes.

Une machine ne peut exécuter qu'une tâche à la fois et une tâche ne peut (et ne doit) être exécutée que par une seule machine.

Chaque machine  $l$  est associée un ensemble de tâches  $N_l$  tel qu'aucune tâche  $j \notin N_l$  ne puisse être exécutée sur la machine  $l$ , avec  $l = 1, \dots, m$ .

De plus, chaque machine possède des intervalles d'inactivité notés  $U_{lv} = [s_{lv}, e_{lv}]$ ,  $s_{lv} < e_{lv}$ ,  $v = 1, \dots, u_l$  de telle sorte que la machine  $l$  ne puisse exécuter aucune tâche dans ces intervalles. On note  $U_l = U_{l1} \cup \dots \cup U_{lu_l}$ .

Sur la machine  $l$ , la tâche  $j$  peut être exécutée dans l'un des intervalles fixes  $I_{ljk} = [s_{ljk}, e_{ljk}]$ ,  $s_{ljk} < e_{ljk}$ ,  $k = 1, \dots, n_{lj}$ .

Un poids  $w_{ljk}$  est associé à chaque intervalle  $I_{ljk}$ , représentant la valeur de la tâche  $j$  si elle est exécutée sur cet intervalle. Ces valeurs seront utilisées dans la fonction objectif, servant à quantifier le potentiel d'une solution et ainsi à comparer deux solutions entre-elles.

Chaque tâche  $j$  ne peut être exécutée que sur l'un de ses intervalles, ou sur aucun (rejetée). (Voir l'article [0])

En résumé, dans la définition du problème tel que présenté dans l'article [Scheduling jobs with fixed start and end times], on retrouve :

- $m$  le nombre de machines disponibles
- $n$  le nombre de tâches à exécuter
- $N_l$  l'ensemble des tâches pouvant être exécutée sur la machine  $l$
- $u_l$  le nombre d'intervalles d'inactivité de la machine  $l$
- $U_{lv}$  les intervalles d'inactivité de la machine  $l$ ,  $v = 1, \dots, u_l$
- $U_l = U_{l1} \cup \dots \cup U_{lu_l}$  l'ensemble des intervalles d'inactivité de la machine  $l$
- $I_{ljk} = [s_{ljk}, e_{ljk}]$  les intervalles de la tâche  $j$  exécutée par la machine  $l$ ,  $k = 1, \dots, n_{lj}$
- $w_{ljk}$  le poids associé à l'intervalle  $I_{ljk}$ .

Au vue des éléments énoncés précédemment, le sous-problème de Keolis d'affectation des véhicules aux missions peut être vu comme un Fixed Interval Scheduling. En effet, on peut associer les missions aux tâches et les véhicules aux machines.

Cependant, notre sous-problème n'en est qu'un cas particulier.

Dans notre cas, les tâches (missions) ne sont définies que par un seul intervalle. De plus, les machines (véhicules) sont toujours disponibles. Enfin, le temps d'exécution d'une tâche est fixe et ne dépend pas de la machine (véhicule) qui lui est associée.

Nous pouvons ainsi reformuler le problème :

- $m$  le nombre de véhicules
- $n$  le nombre de missions
- $N_l$  l'ensemble des tâches pouvant être exécutée par le véhicule  $l$
- Il n'existe plus d'intervalle d'inactivité. Donc  $U_l = \emptyset$
- $I_{ljk} = I_j = [s_j, e_j]$ . On omet  $k$  car les missions n'ont qu'un seul intervalle disponible. De plus, on omet  $l$  car la mission aura toujours la même durée, quelque soit le véhicule choisi
- $w_{ljk} = w_j$  le poids associé à l'intervalle  $I_j$ . Pour les mêmes raisons,  $l$  et  $k$  ne sont plus pertinents

Ainsi présenté, le problème correspond au chapitre 4 de l'article [0]. Les auteurs proposent donc la solution de construire un graphe à partir des tâches et des machines.

Pour construire un tel graphe, on part de principe qu'un nœud représentera l'état d'association des jobs aux machines (dans le cas de Keolis, ça sera l'état d'association des missions aux véhicules). Autrement dit, chaque nœud est un  $n$ -vecteur représentant l'état courant des  $n$  machines. De ce fait, chaque nœud représentera soit le début d'un travail (nouvelle attribution d'une tâche à une machine), soit la fin d'un travail (libération d'une machine). On y ajoute un état de départ, dont partent tous les chemins, et un état de fin, où convergent tous les chemins. Ce seront des états où toutes les machines sont disponibles.

Suite à cela, les arcs sont construits entre les nœuds de sorte que chaque nœud correspondant à la fin d'un travail (où plus simplement si la machine considérée est déjà libre) soit relié à un nœud de début de tâche pour la machine.

Tous ces arcs forment alors des ramifications représentant les différentes permutations tâches-machines et convergeront tous vers l'état final, où plus aucune tâche n'est attribuée à une machine.

Enfin, chaque arc d'attribution d'une tâche est pondéré par le poids  $w_j$  associé à l'intervalle  $I_j$ .

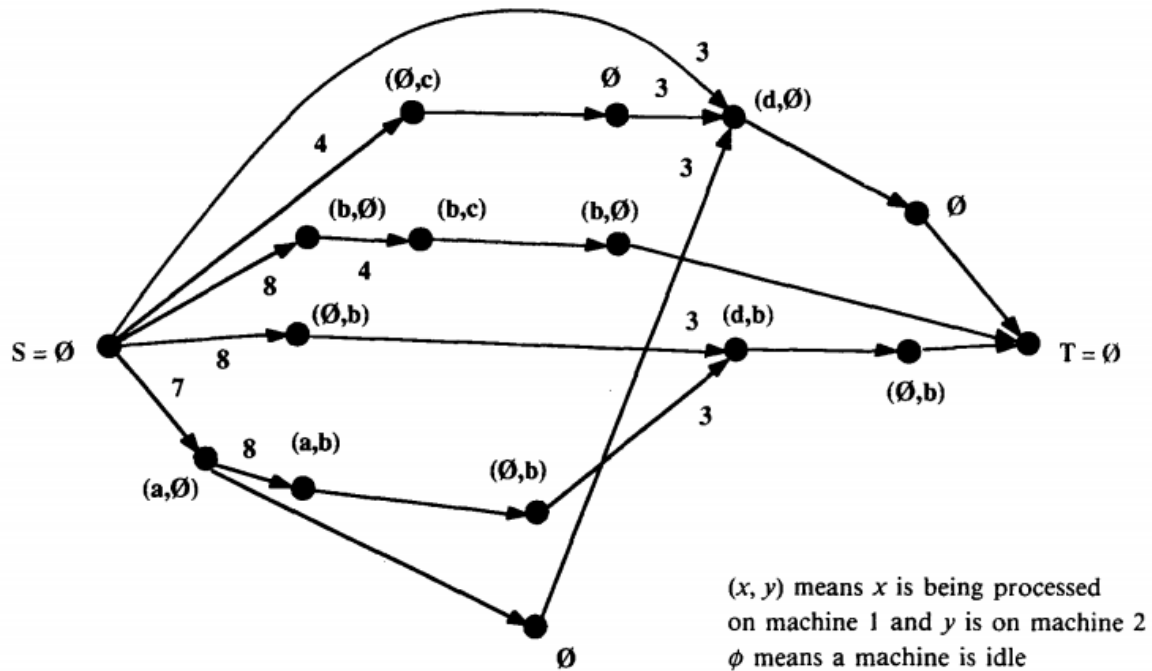


Figure 1 – Graphe d'un problème Fixed Interval Scheduling

Une fois le graphe construit, la résolution du problème revient à trouver le chemin le plus long reliant l'état de départ et l'état final. En effet, plus un chemin contiendra d'arc fortement pondéré, plus il accumulera de poids. Concrètement, cela signifie que plus une suite d'attributions tâche-machine sera longue, plus elle remplira les conditions demandées. On voit aussi que si un chemin contient tous les arcs d'attribution, alors il sera une solution optimale. Dans le cas où ce n'est pas possible, le chemin le plus long correspondra à la solution remplissant au mieux (mais pas parfaitement) les contraintes imposées.

Résoudre un tel graphe n'est pas chose aisée mais une astuce permet de la simplifier. Si on inverse chaque poids, trouver le chemin le plus long revient alors à trouver le plus court.

Il suffit donc, en théorie, d'appliquer un algorithme de plus court chemin tel que celui de Dijkstra.

Cependant, la complexité de cette résolution n'est pas acceptable pour la problématique de Keolis. En effet, dans le cas de l'affectation des missions aux véhicules, le nombre de ces missions et la quantité de bus provoqueraient un graphe de plusieurs milliards de nœuds. De plus, cette solution ne tient pas compte des potentiels kilométriques, pourtant primordiaux dans l'estimation qualitative d'une solution.

### 2.3 Dispatching Buses

L'article [0] présente un problème ressemblant fortement à celui du stockage des bus de Keolis. Il consiste à trouver un arrangement optimal d'un certain nombre de véhicules dans un entrepôt de taille finie et de structure précise.

L'article pose les hypothèses suivantes :

- L'entrepôt est formé de colonnes de taille finie dans lesquelles les véhicules viendront se garer
- Les véhicules entrent d'un côté et sortent de l'autre (structure d'une file FIFO)
- La taille de colonnes est connue et fixe
- La taille de chaque véhicule est connue et fixe
- Tous les véhicules quittent l'entrepôt le matin et y reviennent le soir (ce qui implique qu'il y a autant de missions à remplir hors de l'entrepôt que de véhicules)
- Les heures d'arrivée et de départ de chaque mission sont connues

Mathématiquement, nous avons :

- $C = 1, 2, \dots, k$  l'ensemble des colonnes
- $L_k, \forall k = 1, 2, \dots, k$  la taille de chaque colonne  $k$
- $V = 1, 2, \dots, n_v$  l'ensemble des arrivées
- $D = 1, 2, \dots, n_d$  l'ensemble des départs
- $n_v = n_d$ , comme dit plus haut, il y a autant de départs que d'arrivées
- Soit  $t'_i, t''_j, l'_i$  et  $l''_j$  respectivement le type du véhicule arrivant  $i$ , le type de véhicule demandé par le départ  $j$ , la taille du véhicule arrivant  $i$  et la taille du véhicule demandé  $j$

L'objectif est donc d'ordonner les véhicules entrants et les missions en  $m$  sous-ensemble de manière à avoir :

- $V_k = i_1, \dots, i_{n(k)}$
- $D_k = j_1, \dots, j_{n(k)}$
- $i_h > i_{h+1}$  et  $j_h > j_{h+1}$  avec  $h = 1, \dots, n(k) - 1$
- $\sum_{h=1}^{n(k)} l'_h = \sum_{h=1}^{n(k)} l''_h \leq L_k$

Bien que ce problème corresponde bien, à première vue, à la problématique de Keolis, la contrainte imposant que tous les véhicules quittent l'entrepôt chaque jour est une limite trop importante. De plus, cette formulation ne prend pas en compte les immobilisations obligatoires et planifiées.

### 3 Bilan

Les problématiques liées au transport font partie des sujets les plus traitées dans la littérature de Recherche Opérationnelle. Cependant, les problèmes qui leur sont associés sont aussi variés que nombreux, chacun possédant ses caractéristiques propres découlant des contraintes métiers.

Parmi les trois exemples détaillés ci-dessus, deux sont particulièrement intéressants et se rapprochent des attentes de Keolis.

Le Fixed Interval Scheduling permettrait de répondre partiellement aux demandes du sous-problème d'attribution des bus aux missions et le Dispatching Buses à celui de stockage au sein du parc.

Il est à noter tout de même que, au vu de l'ampleur des données à traiter, celui-ci offre une résolution bien trop complexe. Cela s'explique par le fait qu'il recherche à tout prix une solution optimale (si celle-ci existe). Le temps de calcul que cela demanderait est donc inenvisageable pour notre automate devant être appelé plusieurs fois par jours.

Le second en revanche semble bien adapté à la demande, à l'exception de la contrainte obligeant tous les véhicules à sortir de l'entrepôt, bloquant donc une contrainte clé de Keolis : les immobilisations.

# 5

## Analyse et conception

### 1 Algorithme choisi

L'algorithme de résolution présenté ci-dessous fut proposé par M. Kergosien.

Cet algorithme fut construit à partir de deux hypothèses majeures :

- Premièrement, les techniciens de la maintenance affectuant les immobilisations peuvent déplacer un ou plusieurs bus lorsqu'ils souhaitent en emmener un dans l'atelier, et ce quelle que soit la position sur le parking du bus concerné par l'immobilisation. Concrètement cela implique que l'automate n'aura pas à se soucier des contraintes de sortie d'un véhicule lorsque celui-ci commence une immobilisation.
- Deuxièmement, lorsqu'une immobilisation est effectuée par les techniciens, le bus n'est pas rangé sur le parking normal mais reste sur un parking secondaire, sans contrainte. Concrètement, cela signifie qu'un bus ayant terminé une immobilisation n'est pas de nouveau considéré sur le parking (donc sans les contraintes de place associées) mais dans une liste secondaire où tous les bus s'y trouvant sont directement accessibles.

```
1:  $L_e \leftarrow$  liste d'événements (début de missions et d'immobilisations)
2:  $L_r \leftarrow$  liste d'événement résolu (vide)
3: while  $L_e \neq \emptyset$  do
4:    $E \leftarrow L_e.\text{first}$ 
5:    $L_e \leftarrow L_e \setminus E$ 
6:   if  $E.\text{type} = \text{'startMission'}$  then
7:      $L_v \leftarrow$  liste véhicules dispo (bus en bout de parking et bus sortis de maintenance)
8:      $L_v \leftarrow L_v \setminus$  bus ayant une immo avant la fin de mission
9:      $L_v \leftarrow L_v \setminus$  bus ne respectant pas les critères
10:     $V \leftarrow$  bus maximisant  $L_e$  potentiel kilométrique  $\in L_v$ 
11:     $L_e \leftarrow L_e \cup$  nouvelle fin de mission ( $V$ )
12:     $L_r \leftarrow L_r \cup$  nouvelle attribution de mission ( $V$ )
13:  else if  $E.\text{type} = \text{'endMission'}$  then
14:     $L_p \leftarrow$  liste place dispo (places en bout de ligne d'entrée)
15:     $L_p \leftarrow L_p \setminus$  bus ne respectant pas les critères
16:     $P \leftarrow$  place maximisant l'inter-temps  $\in L_p$ 
```

```

17:      $L_e \leftarrow L_r \cup L_r$  nouvelle attribution de place (P)
18:   else if E.type = 'startImmo' then
19:     if E.vehicule  $\in$  parking then
20:       parking  $\leftarrow$  parking  $\setminus$  E.vehicule
21:        $L_e \leftarrow L_e \cup$  nouvelle fin d'immo (E.vehicule)
22:        $L_r \leftarrow L_r \cup$  nouvelle attribution immo (E.vehicule)
23:     end if
24:   else if E.type = 'endImmo' then
25:     parkingMaintenance  $\leftarrow$  parkingMaintenance  $\cup$  E.vehicule
26:   end if
27: end while

```

**Etape 0 :** L'automate doit tout d'abord lire les fichiers d'entrées sous format CSV afin d'obtenir le jeu de données composant le problème à résoudre. Cette étape ne fait pas réellement partie du processus de résolution mais est nécessaire.

**Etape 1 :** On construit une liste d'événements. Ceux-ci correspondent aux moments où l'automate devra prendre une décision d'affectation. Ces événements peuvent être le début d'une mission (affectation d'un bus disponible en fonction des critères de la mission), le début d'une immobilisation, le retour d'un mission (rangement du bus sur le parking) ou la fin d'une immobilisation. Ces événements sont triés par ordre croissant d'heure de résolution.

Si, pendant l'exécution de l'automate, d'autres événements sont ajoutés à la liste, celle-ci est automatiquement retriée afin de conserver sa cohérence. Cette liste est initialisée avec le début des missions et des immobilisations lues dans les fichiers d'entrée.

De plus, on construit une liste d'événements résolus initialement vide. A chaque fois qu'un événement est considéré par l'automate et qu'une décision est prise, on l'ajoute à cette liste.

**Etape 2 :** On lit (et supprime) le premier élément de la liste d'événements et, dépendamment de son type, on résout l'étape 3, 4, 5 ou 6. Si la liste est vide, tous les événements devant être planifiés le sont, l'automate peut donc s'arrêter (passer à l'étape 7).

**Etape 3 :** Si l'événement lu à l'étape 2 est le début d'une mission. On fait la liste des bus disponibles, comprenant ceux en bout de parking et ceux sur le parking secondaire de la maintenance. Puis on filtre la liste obtenue selon les critères suivants :

- On élimine les bus ayant une immobilisation commençant avant la fin de la mission.
- On élimine les bus ne remplissant pas les critères Energie, Type, Taille et Habillage.
- Parmi les restants, on choisit celui avec le plus grand écart entre la distance déjà parcourue et le potentiel kilométrique à atteindre.

Si après l'application d'un des critères, il ne reste plus qu'un véhicule, celui-ci est attribué à la mission, même s'il ne remplit pas les suivants. De la même façon, si un critère élimine tous les véhicules, il est abandonné et l'algorithme passe au critère suivant. De cette façon, même si un critère n'est pas respecté et que le véhicule n'est pas parfaitement adapté à la mission, ce dernier sera choisi le mieux possible.

Une fois le véhicule à affecter trouvé, on ajoute le retour de la mission à la liste des événements, on ajoute ce début de mission à la liste d'événements résolus et nous retournons à l'étape 2.

**Etape 4 :** Si l'événement lu à l'étape 2 est un retour de mission. On commence par mettre à jour le total de kilomètres parcourus par le véhicule. Puis, on cherche une place de parking où le véhicule peut se garer. Pour cela, on fait la liste des places disponibles (en réalité, seules les places « en bout de ligne d'entrée » seront considérées car il n'y a aucun intérêt



à laisser des places vides et inaccessibles). Parmi ces places, on filtre celles ne respectant pas le critère de taille du véhicule entrant. Enfin, on choisit celle maximisant l'inter-temps (c'est-à-dire le temps séparant l'arrivée du véhicule courant de l'arrivée du véhicule occupant la place devant).

Une fois la place trouvée et attribuée, on ajoute ce retour de mission à la liste d'événements résolus et nous retournons à l'étape 2.

**Etape 5 :** Si l'événement lu à l'étape 2 est le début d'une immobilisation. On vérifie que le véhicule concerné n'est pas parti en mission. Si ce n'est le cas, on supprime le dit véhicule du parking (peu importe sa place comme expliqué par l'hypothèse n°1), on ajoute l'événement de fin d'immobilisation à la liste des événements, on ajoute ce début d'immobilisation à la liste d'événements résolus et on retourne à l'étape 2.

**Etape 6 :** Si l'événement lu à l'étape 2 est une fin d'immobilisation. On ajoute le bus au parking secondaire (comme expliqué par l'hypothèse n°2) et on retourne à l'étape 2.

**Etape 7 :** La dernière étape consiste à écrire la solution trouvée par l'automate. Pour cela, on se base sur la liste d'événements résolus.

Cet algorithme permet la résolution des deux sous-problèmes d'un seul coup, ce qui est un gain de temps conséquent. De plus, le fait que les prises de décision de l'automate soient effectuées par ordre chronologique permet, pour chacune d'entre-elles, d'être résolue selon l'état courant du système.

## 2 Conception et modélisation

Au vu de l'algorithme précédemment détaillé et des fonctionnalités attendues du système, le programme sera divisé en six parties principales :

**Parser :** Le parser lira les données brutes fournies en entrée dans les fichiers CSV. Son rôle sera uniquement de transformer les fichiers texte en ensembles de « chaînes de caractères » et non de vérifier la cohérence des données.

**Model :** L'ensemble des classes de la partie Model représenteront les données métier (Véhicule, Mission, Place de stationnement, etc.). Les objets seront instanciés depuis les informations fournies par le parser et de ce fait devront vérifier leur cohérence (bon formatage, bon type, etc.).

**Evénements :** L'ensemble des événements demandant une prise de décision de la part de l'automate, modifiant l'état du système (attribution d'un véhicule à une mission, rangement d'un véhicule de retour de mission au sein du parc, etc.). C'est au sein des méthodes de ces classes que seront réellement implémentés les mécanismes de résolution décrits dans l'algorithme (recherche des places disponibles, des véhicules respectant les critères d'une mission, etc.).

**Coeur :** Coeur du programme, cette partie est principalement composée de la queue d'événements à résoudre. C'est dans cette partie que sera stocké l'état du système à chaque instant et que se fera l'appel aux méthodes des événements modifiant ce système. Ce package aura aussi pour rôle de sauvegarder chaque changement pour la partie suivante.

**Test :** Une fois l'algorithme exécuter, il est nécessaire de vérifier que la solution proposée soit effectivement réalisable. Pour cela, cette partie contiendra des classes et méthodes différentes de celles utilisées dans les autres parties afin de contrôler que chaque action est correcte.

**Résultats :** Cette partie récupérera les résultats du coeur et les formatera comme attendu (plus écriture dans le fichier de sortie).

## 2.1 Parser

Le package Parser n'est composé que d'une seule classe, elle aussi nommée Parser. Cette classe ne contient qu'une unique méthode statique ayant pour rôle de lire un fichier CSV à partir de son chemin. Si le fichier donné est bien un CSV, la méthode renverra un tableau de tableaux de chaîne de caractères (*String[][]* en java). Le tableau principal sera les lignes du fichier tandis que les sous-tableaux contiendront les éléments de chaque ligne séparés par des point-virgules.

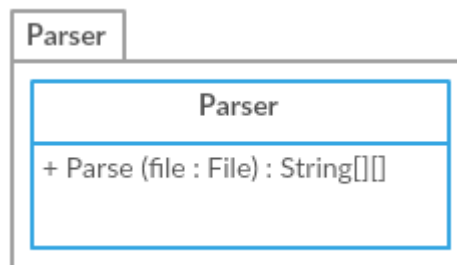


Figure 1 – Diagramme de classe partiel du package Parser

## 2.2 Model

Ce package est composé de 8 classes :

- Date : Implémentation personnelle d'une date. Contient aussi les méthodes personnalisées afin de coller au mieux aux cas particuliers pouvant être rencontrés.
- Immobilization : Contient les informations relatives à une immobilisation (véhicule concerné, date, etc.).
- KilometerTarget : Contient les informations relatives à une cible kilométrique (nombre de kilomètres, date, etc.). Sera contenu dans un objet Vehicle.
- Mission : Contient les informations relatives à une mission (date, critère, etc.).
- Parking : Contient l'ensemble des places de parking ainsi que les méthodes permettant de trouver les libres, les accessibles, etc.
- Path : Contient les informations relatives à un chemin d'accès ou de sortie d'une place de parking. Sera contenu dans un objet Place.
- Place : Contient les informations relatives à une place de parking (numéro, chemin d'accès et de sortie, présence d'un véhicule, etc.).
- Vehicle : Contient les informations relatives à un véhicule (numéro, cible kilométrique, etc.).

Ces classes sont donc la représentation des « objets métiers ». Il est à noter que la classe Date sert d'abord à représenter une date / heure (de mission par exemple) mais sera aussi utile afin de marquer la temporalité du déroulement de l'algorithme.

De plus, chacune de ces classes (à l'exception de Date et de Parking) contient une méthode *extractXXX* prenant en paramètre un *String[][]* renvoyé par le Parser et instanciant tous les objets nécessaires (une ligne du fichier d'entrée lu correspondant à un objet).

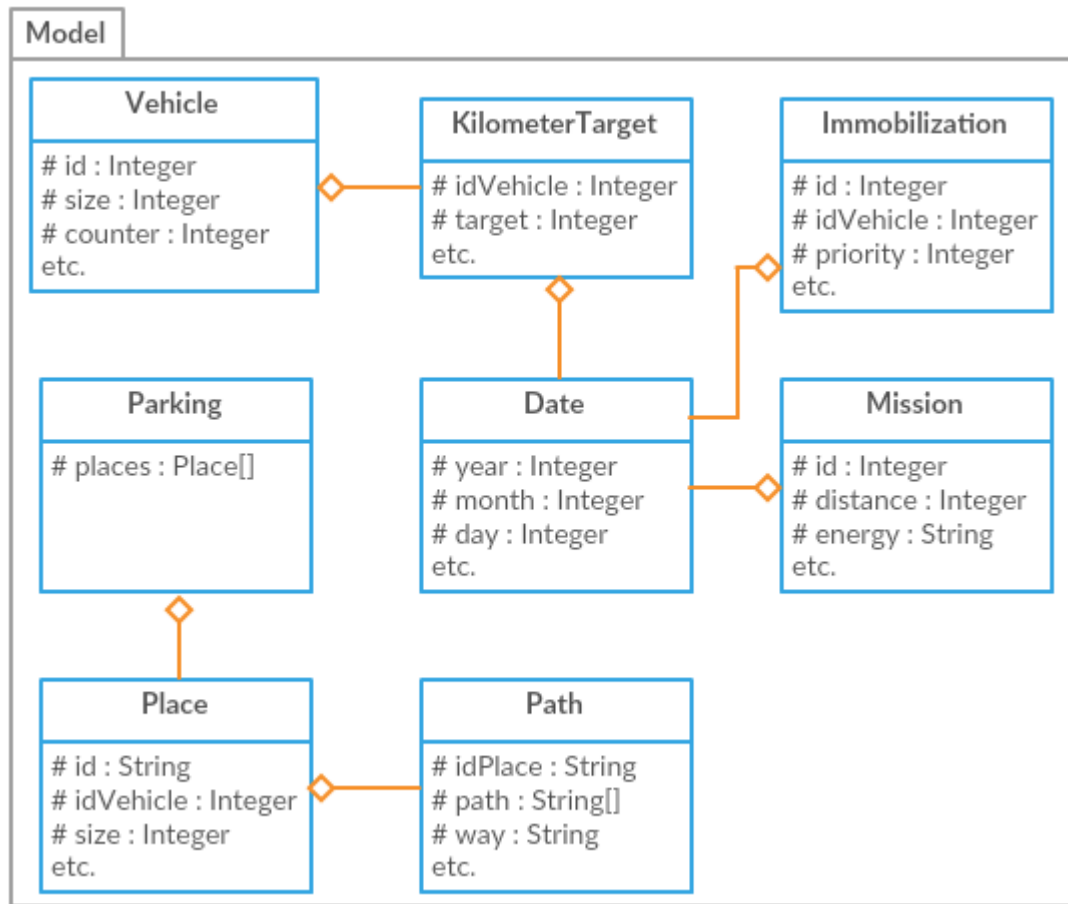


Figure 2 – Diagramme de classe partiel du package Model

## 2.3 Événements

Le package Event contient en premier lieu une classe Event. Il s'agit d'une classe abstraite servant de base à tous les autres événements. Elle ne contient qu'un attribut *time* fixant sa position chronologique par rapport aux autres événements et une méthode abstraite *exec* implémentant réellement les actions à effectuer à chaque événement.

De cette classe mère sont dérivées quatre filles :

- MissionStartEvent : Représente le début d'une mission (Voir [Section 1](#))
- MissionEndEvent : Représente la fin d'une mission (Voir [Section 1](#))
- ImmoStartEvent : Représente le début d'une immobilisation (Voir [Section 1](#))
- ImmoEndEvent : Représente la fin d'une immobilisation (Voir [Section 1](#))

Chacune implémente dans sa méthode *exec* les actions à réaliser lorsque l'événement correspondant se produit.

On trouve aussi dans ce package un comparateur d'événements (de la classe Event ou dérivé) basé sur leur *time* permettant de l'ordonner par ordre chronologique.

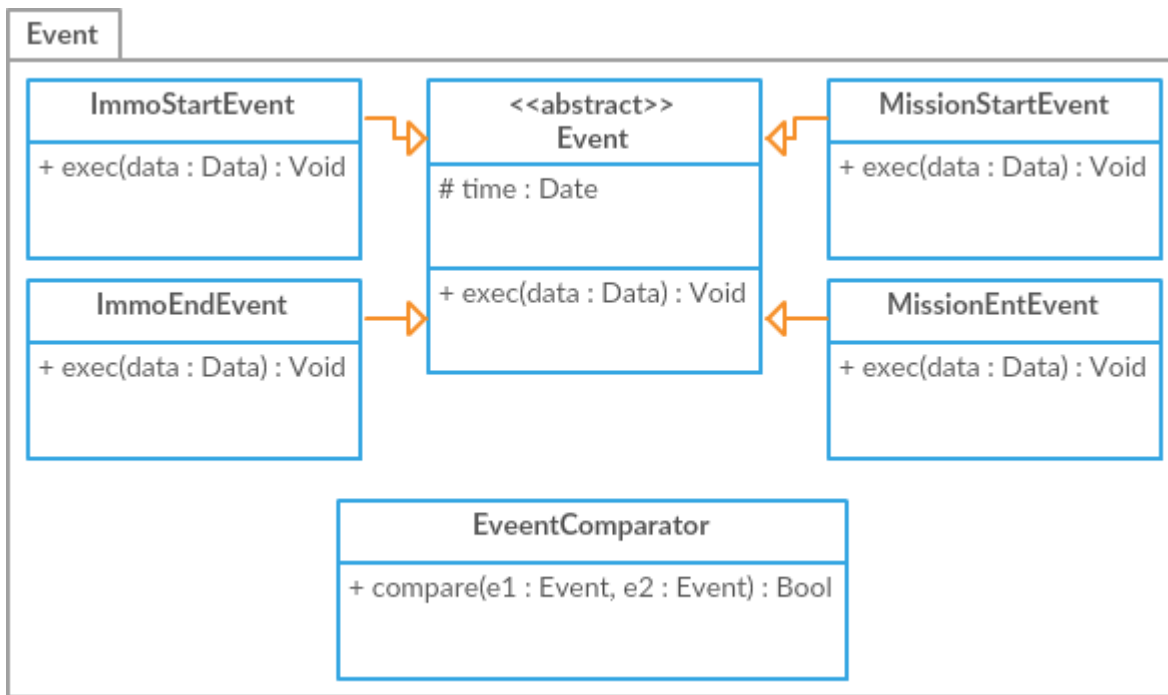


Figure 3 – Diagramme de classe partiel du package Evenement

De manière similaire, on trouve dans ce package une classe abstraite *EventFinished* représentant un événement une fois achevé et résolu. De la même façon, quatre classes filles en héritent :

- *MissionStartEventFinished* : Représente le début d’une mission résolu
- *MissionEndEventFinished* : Représente la fin d’une mission résolue
- *ImmoStartEventFinished* : Représente le début d’une immobilisation résolu
- *ImmoEndEventFinished* : Représente la fin d’une immobilisation résolue

Ces classes représentent donc une *solution* pour chaque événement qu’elles décrivent. Ainsi, dans un objet *MissionStartEventFinished* sera sauvegarder la mission en question, le véhicule qui lui a été attribué et l’heure de cette attribution.

Ces objet sont instanciés à la fin de la méthode *exec* de l’*Event* correspondant et stockés dans une liste ordonnée par date. Une fois l’algorithme fini et la planification exécutée, cette liste d’*EventFinished* permet de connaître l’état du parking, des missions et des immobilisations à chaque instant. C’est grâce à cette liste que la partie Tests pourra vérifier qu’à chaque instant la solution est réalisable et cohérente.

## 2.4 Core

Coeur du programme, ce package contient la classe centrale *EventListAlgorithme*. Cette classe est le point d’entrée de l’automate. A son initialisation, elle demande la lecture de tous les fichiers d’entrée, la construction de tous les objets associés et les stocke dans un attribut *Data*.

De plus, elle contient un objet *PriorityQueue* contenant tous les événements à réaliser (classés par ordre chronologique). Tant que cette Queue n’est pas vide, l’automate ne s’arrête pas.

Ce package contient aussi la classe *Data* qui n’est autre que l’ensemble des informations nécessaires au fonctionnement de l’automate.

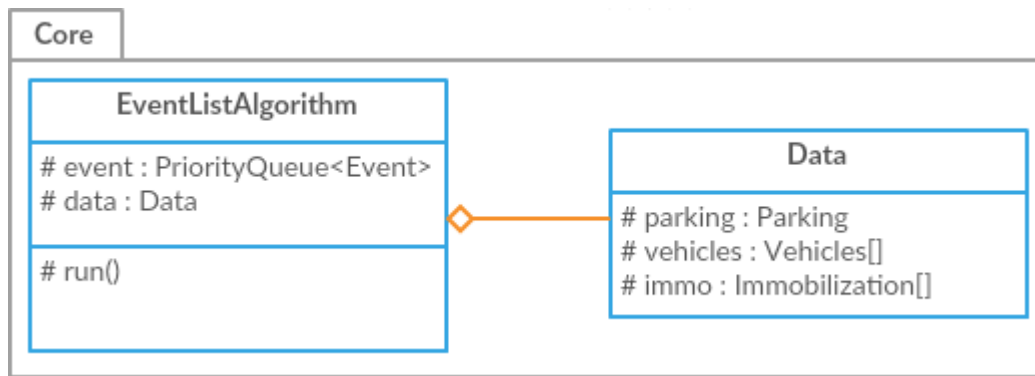


Figure 4 – Diagramme de classe partiel du package Core

## 2.5 Tests

La classe principale de ce package s'appelle CheckClass. Il s'agit d'une classe générale qui vérifiera après l'exécution de l'algorithme que la solution proposée est valide. Une unique objet de cette classe est instancier en début de programme, mais la vérification ne se fera qu'à la fin.

Deuxièmement, une classe CheckParking permet de représenter à tout instant l'état du parking. La encore, un objet unique de cette classe, contenu dans l'objet CheckClass sera instancié en début de programme. Il sera d'ailleurs une copie de l'état initial du parking.

Les méthodes de cette classe permettent d'exécuter les "actions basiques" du parking telles qu'ajouter un véhicule sur un place, en retirer, etc. mais aussi de vérifier qu'un véhicule peut sortir ou encore qu'une place est libre avant qu'un nouveau véhicule ne l'occupe. Aucune de ces méthodes ne fait appelle à du code des classes et méthodes des autres packages afin de ne pas propager les éventuelles erreurs.

Enfin, une classe CheckPlace, représentant donc une place de parking permet de réaliser les vérification propre à une place. Plusieurs instances de cette classe sont présentes dans un objet CheckParking.

A la fin du programme, une fois la liste d'*EventFinished* remplie, la classe CheckClass va parcourir ces *EventFinished* et vérifier que chaque action qu'ils proposent (attribution de tel véhicule à telle mission, etc.) est possible. De plus, elle va mettre à jour la classe CheckParking afin de s'assurer que l'état du parking est toujours cohérent et que les véhicules sortants sont bien disponibles, etc.

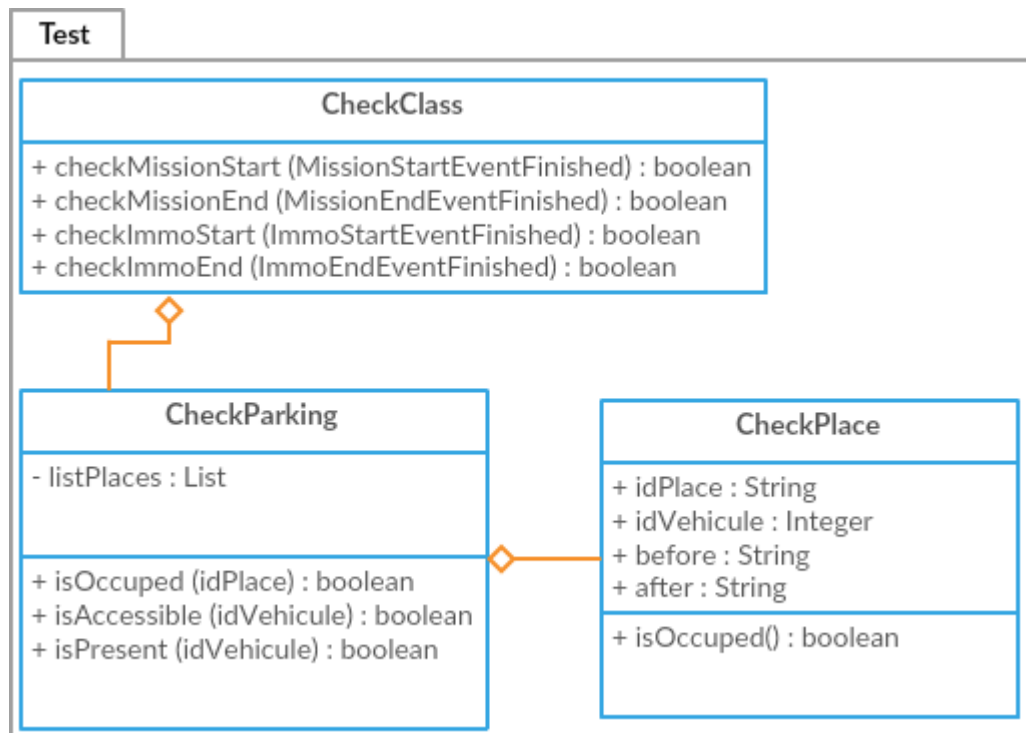


Figure 5 – Diagramme de classe partiel du package Test

## 2.6 Résultats

Le package Résultats ne contient qu'une seule et unique classe Results ayant pour but d'écrire le fichier CSV de sortie. Une seule méthode est disponible dans cette classe : write. Celle-ci prend en entrée une collection d'*EventFinished* et écrit pour chacun d'eux une ligne contenant leurs informations utiles (date de l'événement, mission concernées, véhicules, etc.) dans le fichier de sortie.

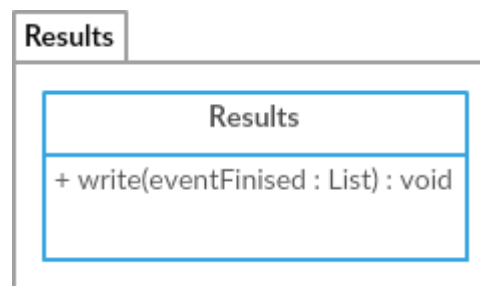


Figure 6 – Diagramme de classe partiel du package Results

## 2.7 Diagramme de classes

Le diagramme de classes final est visible ici : [Figure 2](#) (Annexe D).

# 6

## Mise en œuvre

### 1 Environnement

Le programme fut développé en Java pour plusieurs raisons. Premièrement, il s'agit d'un langage facilement portable d'une architecture à l'autre, dépendamment du système sur lequel tournera le programme. Deuxièmement, Java possède un certain nombre de structures de données intégré dans la version native sans avoir à ajouter de packages externes.

L'automate peut donc s'exécuter avec un simple JRE 8.

### 2 Structure de données

#### 2.1 Algorithme événementiel à liste

Comme détaillé dans la [Section 1](#) (Chapitre 5), l'algorithme prend la forme d'une liste d'événements ordonnés par date qui s'exécutent les uns après les autres, modifiant l'état des données et du système. Pour se faire, un objet `EventListAlgorithme` est créé en début de programme. Cet objet a à la fois la tâche d'initialiser les événements et de les exécuter.

Cette forme à l'avantage d'être déterministe : si on exécute deux fois le programme avec les mêmes données d'entrée, on obtiendra deux fois le même résultat.

En contre-partie ce fonctionnement ne permet pas une approche heuristique. L'objectif de cet algorithme est donc, dans un premier temps, de tenir compte de toutes les contraintes et d'essayer de trouver une solution les respectant toutes. Pour cela, chaque événement qui s'exécutera se pliera au maximum aux exigences du système. Dans le cas où il n'y a pas de solution réalisable (aucun véhicule disponible, etc.), l'automate recommencera l'ensemble des événements en baissant son niveau d'exigence.

Ce niveau d'exigence se matérialise principalement par un niveau de priorité pour les immobilisations. En effet, les immobilisations possèdent toutes un niveau entre 1 et 10 représentant leur importance. La première fois, l'automate tentera de toutes les appliquer. S'il n'y arrive pas, il recommencera sans prendre en compte celles avec le niveau de priorité le moins important. De la même façon, s'il n'y a toujours pas de solution possible, il abaissera encore son niveau d'exigence.

## 2.2 Gestion des données

Comme expliqué dans la partie ci-dessus, les événements exécutaient vont modifier l'état du système. L'ensemble de ces données se trouvent dans un objet de la classe Data. Cet objet est instancié à la création de l'objet EventListAlgorithme et initialisé par lecture des fichiers d'entrée grâce au Parser.

### Classe Data

L'ensemble des informations lues par le Parser est stocké dans cette classe :

- Liste des événements
- Liste des événements finis
- Liste des véhicules
- Liste des immobilisations
- Le parking (Voir [Section 2.2](#))
- Le niveau de priorité actuel de l'algorithme

A l'exception du parking et des listes d'événements se vidant et se remplissant au cours de la résolution, ces données ne sont pas destinées à être modifiées. L'ensemble des informations modifiables sont dans l'objet Parking.

### Classe Parking

Cette classe contient :

- L'ensemble des places du parking (avec leurs chemins d'entrée et de sortie, et le véhicule actuellement stationné)
- Les véhicules sur le parking temporaire de la zone de maintenance

L'ensemble des modifications d'état du système dues aux arrivées et départs de véhicules seront répercutées dans cet objet.

Lorsqu'un véhicule quitte le parking (pour une mission ou une immobilisation) son identifiant est simplement supprimé de la place qu'il occupait. Le véhicule est donc considéré en cours d'utilisation et la place libre.

Lorsqu'un véhicule à fini une immobilisation, son identifiant est alors ajouté à la liste des véhicules sur le parking de la maintenance.

Lorsqu'un véhicule revient de mission et se gare sur le parking principale, après lui avoir trouvé une place, son identifiant est donc écrit dans cette même place, signifiant que le véhicule est revenu et la place de nouveau occupée.

## 3 Algorithme de selection du véhicule

Lorsqu'une mission se lance l'automate doit trouver le véhicule le plus approprié pour celle-ci. Pour cela, une succession de choix et de tris est effectuée sur la liste de véhicules.

### Véhicules disponibles

La première chose à faire est de ne considérer que les véhicules qui peuvent sortir du parking. En effet, cette condition est non négociable car l'un des objectifs de l'automate est de ne pas créer de mouvements de véhicules inutiles. Pour cela, l'algorithme regarde la position actuelle de tous les véhicules sur le parking et vérifie qu'aucun autre véhicule ne se trouve sur l'une des places de son chemin de sortie. Si c'est le cas, le véhicule est considéré comme disponible. A cela on rajoute les véhicules situés sur le parking de la maintenance car ce parking est supposé sans contrainte.



### Véhicules n'ayant pas d'immobilisation prévue

La prochaine étape est d'enlever les véhicules qui ont une immobilisation prévue entre le début et la fin de la mission. Pour cela il suffit de chercher, par véhicule, si une immobilisation dont la date de début est entre les dates de début et de fin de la mission leur est attribuée. Si c'est le cas, on les supprime de la liste des véhicules disponibles pour la mission.

A noter que les immobilisations ont un niveau de priorité croissant entre 1 et 10. Ne sont considérées pour cette étape que les missions dont la priorité est inférieure ou égale au niveau de priorité générale accepté par l'algorithme.

### Tri selon les critères de mission

A partir de cette étape, tous les véhicules retenus aux précédentes sont également disponibles pour la mission. Cependant ils ne sont pas tous équivalents dans leur respect des contraintes.

Le but de cette étape est donc de trier les véhicules selon le nombre de critères qu'ils respectent. Pour cela, une méthode va comparer les caractéristiques de la mission avec celles du véhicule et renvoyer le nombre de critères respectés. Une fois cela fait pour tous, on peut trier les véhicules par ordre décroissant (du plus respectueux au moins).

### Tri selon le potentiel kilométrique

Enfin, parmi les véhicules ayant le plus de caractéristiques communes avec la mission, il faut choisir ayant le meilleur critère de potentiel kilométrique. Plusieurs choses sont à noter ici :

- Chaque véhicule peut avoir plusieurs potentiels kilométriques associés. Il ne faut considérer que le dernier en date. C'est-à-dire celui dont la date limite n'est pas encore passée mais la plus proche de la date actuelle.
- A chaque véhicule et son potentiel kilométrique, on va leur associer un ratio égal à  $\frac{\text{Kilomtrage\_actuel} + \text{Kilomtres\_de\_la\_mission}}{\text{Potentiel\_a\_atteindre}}$ . Ce ratio représente donc le pourcentage qu'aura effectué un véhicule après la mission par rapport à son objectif. Si le ratio est supérieur à 1 alors le véhicule aura dépassé sa limite.
- Ce ratio est calculé par une méthode qui renverra celui-ci négatif s'il dépasse 1.

Exemples

- Renvoie 0 si le ratio égal 0
- Renvoie 0.5 si le ratio égal 0.5
- Renvoie -1 si le ratio égal 1
- Renvoie -1.5 si le ratio égal 1.5

A partir de cela, les véhicules sont triés par ordre décroissant de ratio. De cette façon, un véhicule dépassant sa limite sera poussé à la fin de la liste. Le premier véhicule sera donc celui pour lequel la mission le rapprochera le plus de son objectif.

A la suite de ces deux tris, le véhicule en tête de liste est donc celui respectant le plus les critères de la mission et dont la limite de kilomètres à effectuer est la plus proche.

## 4 Algorithme de selection de la place de parking

Lorsqu'une mission se termine, le véhicule revient au parking et l'automate doit alors lui trouver la place la plus adaptée. De la même façon que pour trouver un véhicule adapté à une mission, l'automate va effectuer une série de tri mais cette fois sur les places de parking.

### Places disponibles et accessibles

La première chose à faire est de trouver toutes les places de parking disponibles et accessibles. Disponibles signifie qu'aucun véhicule ne s'y trouve déjà et accessible qu'aucun véhicule ne se trouve sur une des places du chemin d'entrée. Pour vérifier cela, l'automate va simplement itérer, pour chaque place libre, sur chaque place de son chemin d'entrée et vérifier qu'aucun véhicule ne s'y trouve. Si c'est le cas, le chemin est traversable et la place accessible, donc retenue.

### Places respectant le critère de taille

Ensuite, parmi les places accessibles, on élimine toutes celles dont la taille est trop petite pour le véhicule entrant. Cette condition est non négociable puisqu'un véhicule ne peut pas se garer sur une place trop petite pour lui.

### Dernières places accessibles

Jusque-là toutes les places accessibles étaient considérées, cependant, dans les faits, seule la place la plus avancée de chaque ligne est importante. En effet, garer un véhicule sur une place alors qu'une autre, sur la même ligne et plus avancée, est aussi disponible n'a aucun intérêt et serait même handicapant puisqu'une place deviendrait inaccessible pour les autres véhicules.

L'objectif de cette étape est donc de trier les places pour qu'il ne reste que les plus avancées dans le parking. Pour cela, l'algorithme parcourt toutes les places précédemment retenues et ne garde cette fois que celles dont la place juste devant est occupée (aucune place de perdue) ou n'existe pas (place en bout de file, ne peut pas avancer plus).

### Tri selon l'inter-temps

Enfin, afin de choisir la place la plus adéquate et de rendre la solution robuste aux aléas, l'automate va trier les places selon leur inter-temps.

L'inter-temps est le temps séparant l'arrivée de deux véhicules sur des places successives. Le but est donc de maximiser cet inter-temps afin d'éviter, en cas de retard du premier véhicule, que celui-ci ne puisse plus atteindre sa place car le suivant serait déjà arrivé et bloquerait donc l'accès à sa place.

Pour cela, chaque place garde en mémoire l'heure d'arrivée du véhicule actuellement stationné dessus. Lorsqu'un nouveau véhicule arrive et cherche une place, le programme va calculer l'écart entre l'heure actuelle et l'heure enregistrée par chacune des places étudiées, donnant ainsi l'inter-temps. Les places sont alors triées selon leur inter-temps décroissant.

A la suite de cela, la place en tête de liste est donc celle respectant le mieux les critères d'emplacement, de taille et d'inter-temps.

# 7

## Bilan et conclusion

### 1 Bilan du S9

La première partie de ce PRD m'a permis de mettre en pratique plusieurs compétences acquises en cours, notamment en ce qui concerne la gestion de projet, la spécification et la conception. Le fait qu'il s'agisse d'un projet réalisé en individuel permet de mieux se rendre compte des impacts et conséquences sur les phases en aval des choix pris durant les phases en amont.

Au 28/11/2018, les phases d'Initialisation, de Définition du besoin, d'Analyse ont été réalisées dans les temps, en plus de la rédaction de ce document.

La phase de Conception est bien entamée comme prévu dans le diagramme de Gantt, tandis que celle de Développement a démarré plus tôt que prévu. Ces dernières se chevauchent plus qu'initialement prévu, certains éléments découverts durant le développement influençant la Conception du système. Cela ne ralentit cependant pas le projet dont le rendu final est prévu pour fin Mars 2019.

### 2 Conclusion

Pour Keolis comme pour beaucoup d'entreprises, l'automatisation de certaines travaux permet la diminution du nombre de tâches répétitives, peu intéressantes et peu valorisantes.

Ce Projet de Recherche et Développement m'a permis de mettre concrètement en application bon nombre de compétences acquises lors de ma formation à Polytech.

Les phases d'analyse et de conception furent l'occasion de se plonger pleinement dans un problème industriel avec toutes les contraintes, exigences et subtilités que cela implique afin de proposer une solution viable, réaliste et réalisable à un problème concret. La phase de développement, s'entendant sur une plus longue période que les différents projets jusqu'alors réalisés dans un cadre scolaire, permet de comprendre les enjeux des bonnes pratiques de codage et de leur importance.

Enfin, ce projet mit particulièrement l'accent sur la gestion de projet et sa nécessité au déroulement d'un projet.

Au final, bien que les dernières phases ne purent être réalisées, le projet mena à un livrable nécessitant un peu de travail supplémentaire afin d'aboutir à un résultat parfaitement utilisable en conditions réelles.

## Annexes

# A

# Cahier des charges

## 1 Description du besoin

### 1.1 Représentation des données

Plusieurs éléments sont à prendre en compte dans la modélisation du problème global :

- Les véhicules
- Les missions
- Les services
- Les immobilisations
- Les emplacements de stationnement

#### Les véhicules

Les véhicules sont différenciés par un ensemble de caractéristiques :

- Un identifiant
- Une distances déjà parcourue
- Des critères
  - Un type (standard, articulé, etc.) : chaîne de caractères
  - Une énergie (gasoil, électrique, etc.) : chaîne de caractères
  - Un habillage BHNS : valeur Vrai/Faux devant correspondre au critère similaire propre à la mission attribuée
  - Une taille : chiffre définissant la place qu'occupe un véhicule. A comparer avec la caractéristique taille d'un emplacement pour savoir si le véhicule peut être stationné sur ledit emplacement.
- Un potentiel kilométrique : une certaine distance à parcourir avant une date donnée. Dans le logiciel LGV existant, le potentiel kilométrique est représenté par 3 valeurs (Faible, Moyen, Fort) représentant la « quantité » à parcourir par un véhicule pour une journée. Une valeur similaire caractérise chaque mission et doit, le plus possible, correspondre à celle du véhicule associé.  
Cependant, une telle réduction des données n'est pas nécessaire dans le cas d'un automate. Le potentiel kilométrique sera donc représenté par nombre de kilomètres à effectuer dans la journée.

- Un ensemble d’immobilisations d’opportunité : vérification de la maintenance non obligatoire mais préférable, à effectuer dans la journée.
- Une date : date et heure de passage, non renseignée si aucune immobilisation n’est prévue.
- Une durée : durée estimée de l’intervention, non renseignée si aucune immobilisation n’est prévue.
- Un flag de déplacement : le véhicule peut-il être déplacé ?

### Les missions

Les missions sont les activités devant être impérativement associées à un véhicule. A l’origine, elles ne représentaient que les services à effectuer dans une journée. Cependant, les immobilisations obligatoires comportant un certain nombre de caractéristiques communes avec les services, il fut décidé de les regrouper dans une seule catégorie. Les missions sont divisées en deux sous-catégories : les services et les immobilisations.

Les services correspondent à un ensemble de lignes effectuées par le véhicule entre les horaires de départ et de retour. Plus particulièrement, ils sont définis par les caractéristiques suivantes :

- Des critères
- Un type de véhicule (standard, articulé, etc.) : chaîne de caractères devant correspondre exactement à celle du véhicule associé.
- Une énergie de véhicule (gasoil, électrique, etc.) : chaîne de caractères devant correspondre exactement à celle du véhicule associé.
- Un habillage BHNS : valeur Vrai / Faux devant correspondre exactement à celle du véhicule associé.
- Ligne principale : identifiant de la ligne principale.
- Un potentiel kilométrique : Nombre de kilomètres que représente la mission, qui seront donc effectués par le véhicule associé
- Une date : date à laquelle la mission doit être effectuée (couvre implicitement jusqu’au lendemain 03 :00).
- Heure de départ
- Heure de retour

Les critères du véhicule doivent correspondre à ceux demandés par le service pour que celui-ci puisse être attribué.

Les immobilisations sont quant à elles de deux types : les impératives et les planifiées.

Les immobilisations impératives doivent obligatoirement être effectuées. Les immobilisations planifiées ne sont pas obligatoires mais, en fonction de leur priorité, doivent être plus ou moins prises en compte. Une immobilisation planifiée non réalisée sera reportée et sa priorité augmentée.

Après discussion et analyse de la problématique, une nouvelle difficulté apparut : dans le cas où une immobilisation non-obligatoire est planifiée à un jour et une heure donnés par l’automate, le personnel de la maintenance fait ses préparatifs afin de réaliser la maintenance prévue. Cependant, si, entre temps, l’automate est relancé, il se peut que l’immobilisation non-obligatoire ne soit plus à la même heure, désorganisant l’équipe de maintenance.

Pour remédier à cela, une troisième catégorie d’immobilisations fut créée : les immobilisations relaxables. Celles-ci sont d’anciennes immobilisations non-obligatoires, à un moment planifiées par l’automate et approuvées par la maintenance. Elles doivent donc être à présent considérées comme obligatoires dans les nouvelles itérations de l’automate. Cependant, dans le cas où l’automate ne trouverait aucune solution possible, les immobilisations relaxables pourraient, en dernier recours, être abandonnées.

Quel que soit son type, toute immobilisation est définie par une date et un heure de début ainsi qu'une date et une heure de fin. Entre ces deux dates, le véhicule concerné n'est pas disponible pour d'autres missions.

### Le stationnement

Entre chaque mission, les véhicules sont stockés dans un parc de stationnement dont les places ne sont pas indépendantes les unes des autres.

En effet, chaque place nécessite d'en traverser certaines autres pour s'y garer ainsi que certaines autres pour en sortir.

De ce fait, il est nécessaire que tous les emplacements d'entrée soient vides pour qu'un véhicule se gare, et que tous ceux de sortie le soient pour que le véhicule puisse commencer un nouveau service.

De plus, chaque emplacement comporte une contrainte de taille pouvant être différente d'un emplacement à l'autre. Cette caractéristique sera à comparer avec celle des véhicules. Un véhicule plus grand qu'un emplacement ne pourra pas s'y garer. Un véhicule plus petit pourra s'y garer bien que, si l'emplacement est bien plus grand que nécessaire, cela puisse mener à une perte, et donc à une solution non-optimale.

### Caractéristique supplémentaire

Lors de la phase de stationnement, une nouvelle caractéristique a émergé : l'inter-temps.

En théorie peu importe le temps séparant l'arrivée de deux véhicules. Le premier terminant sa mission sera rangé sur la première place et le second sur la deuxième place.

Dans les faits, il est possible que le premier véhicule ait du retard. Dans le cas où le second véhicule arriverait avant le premier, ce dernier ne pourrait plus accéder à sa place de stationnement, car bloquée par le second véhicule.

Cette situation peut être évitée (ou du moins minimiser ses chances de se produire) en s'assurant un certain temps entre les arrivées des deux véhicules. Ainsi, même si le premier est en retard de quelques minutes, le deuxième n'arrivant pas immédiatement après, il aura le temps de se garer malgré son retard.

Le temps planifié entre les arrivées des deux véhicules est appelé inter-temps. Plus celui-ci est élevé (pour chaque paire de véhicule), plus la solution proposée est robuste aux aléas.

## 1.2 Contraintes

Les contraintes sont l'ensemble des règles devant être prises en compte et respectées pour que la solution proposée soit valide.

Les contraintes décrites ci-dessous ne sont que celles imposées explicitement par Keolis et ne prennent pas en compte les restrictions imposées par les données et leur représentation elles-mêmes.

On en compte six :

- Tous les services doivent être affectés.
- Toutes les immobilisations impératives doivent être affectées.
- Les véhicules doivent pouvoir sortir de leur emplacement de stationnement.
- Les véhicules doivent pouvoir accéder à leur emplacement de stationnement.
- Les caractéristiques de véhicule correspondent aux critères de ligne.
- Les caractéristiques de véhicule correspondent aux critères d'emplacement.

# B

# Cahier de spécifications

## 1 Entrée et sortie de l'automate

Bien que l'automate ne doit pas être nécessaire au bon fonctionnement de la planification des missions d'une journée, celui-ci n'est pas indépendant du reste du système.

De ce fait, son exécution se base sur des données d'entrée et il doit produire des données de sortie formatées de façon à ce que le reste du système puisse les interpréter.

En entrée, six fichiers CSV sont fournis :

- Mission.csv
- Vehicule.csv
- Immobilisation.csv
- Cible\_Kilometrique.csv
- Emplacement.csv
- Trajet.csv

Ces fichiers représentent l'état de parc de véhicules à l'instant où les données sont extraites de la base de données et fournies à l'automate.

Dans le cas où l'automate serait lancé en cours de journée (ou plus simplement alors qu'une mission est en cours), il devra être capable de s'adapter aux données « en cours ». Cela signifie que, si un véhicule est en cours de mission à l'exécution de l'automate, ce dernier devra l'ignorer jusqu'à son heure de fin de mission. Ce n'est qu'après la fin de la mission en cours (et donc le retour du bus au parc) que l'automate pourra de nouveau considérer ce véhicule et le réaffecter à une nouvelle mission.

### 1.1 Mission

Ce fichier contient toutes les informations relatives aux services devant être effectués.

- SM
- Jour : 0 = journée courante, 1 à x pour chaque journée projetée
- Heure début : heure de début du service matériel
- Heure fin : heure de fin du service matériel
- Kilomètres : nombre de kilomètres que représente le service matériel



- Ligne Principale : identifiant (mnémoLigne) de la ligne principale associée au SM
- Modèle : STD | ART | ELEC (chaîne qui se rapporte aux valeurs de Modele dans le fichier Vehicules)
- BHNS : Oui | Non (booléen qui se rapporte à la valeur BHNS dans le fichier Vehicules)
- Energie : GO | ELEC (chaîne qui se rapporte aux valeurs de Energie dans le fichier Vehicules)

## 1.2 Vehicule

Ce fichier contient une partie des informations relatives aux véhicules. Il sera à croiser avec le fichier Cibles\_Kilométriques.

- Numéro véhicule : l'identifiant unique du véhicule
- Taille : 1 à x (nombre permettant de fixer la taille du véhicule)
- Compteur Kms : dernier relevé du compteur kilométrique du véhicule
- Opportunité Date : date / heure de démarrage pour une immobilisation d'opportunité ou Null si aucune demande
- Opportunité Durée : durée minimum en minutes d'immobilisation sur parc nécessaire à l'intervention d'opportunité (par défaut 45 minutes)
- BHNS : Booléen Oui | Non
- Energie : GO | ELEC
- Modèle : STD | ART | ELEC

## 1.3 Immobilisation

Contient les données d'immobilisation pour les véhicules concernés. Ses informations devront être prises en compte lors de l'affectation des véhicules.

- Numéro immobilisation : identifiant unique d'une immobilisation
- Date / heure début : début prévisionnel de l'immobilisation
- Date / heure fin : fin prévisionnelle de l'immobilisation arrondie à J+x (où x est la valeur du paramètre de jours projeté pour le Parc)
- Numéro véhicule : l'identifiant du véhicule concerné
- Priorité : priorité de l'immobilisation de -1 à 10, où -1 correspond à une immobilisation impérative. De 0 à 10, priorité croissante.

## 1.4 Cible\_Kilométrique

Ce fichier contient les potentiels kilométriques que chaque véhicule doit effectuer avant une date fixée.

- Numéro véhicule : l'identifiant du véhicule concerné
- Cible Kms : le nombre de kilomètres que le véhicule devra avoir parcouru à la date échéance (sous-entendu à 03 :00 le lendemain)
- Date échéance : la date à laquelle la cible kilométrique doit être atteinte

### Mise à jour

Après analyse de la situation, plusieurs difficultés sont apparues concernant le potentiel kilométrique tel que décrit ci-dessus.

Premièrement, les dates échéances des potentiels étant particulièrement lointaines en comparaison avec les trois jours de planification de l'automate, les cibles kilométriques n'auraient que peu d'impact dans l'algorithme.

Deuxièmement, la cible kilométrique elle-même étant bien plus grande que le nombre de kilomètres parcourus lors d'une mission, ce critère n'influencerait quasiment pas la planification des missions, alors qu'il s'agit tout de même d'une donnée essentielle à prendre en compte.

Afin de remédier à cela, une nouvelle présentation des données fut proposée. Les potentiels kilométriques ne serait plus à long termes mais au jour le jour. Pour se faire, un processus hors de l'automate devra estimer le nombre de kilomètres que chaque véhicule devra effectuer chaque jour.

Ces prévisions journalières devront être réitérées à chaque lancement de l'automate. De ce fait, les aléas du réseau et les inexactitudes des anciennes planifications de l'automate seront prises en compte dans le calcul du nouveau potentiel kilométrique quotidien.

## 1.5 Emplacement

Contient les données concernant un emplacement à un instant donné.

- Numéro emplacement : identifiant unique d'un emplacement
- Taille : taille de l'emplacement, à faire correspondre avec la taille du véhicule
- Véhicule : véhicule déjà présent sur l'emplacement

## 1.6 Trajet

Ce fichier complète le précédent en indiquant, pour chaque emplacement, les places qu'un véhicule devra traverser pour y accéder et pour en sortir.

- Numéro emplacement : identifiant de l'emplacement concerné
- Chemin : places que le véhicule devra traverser
- Entrée / Sortie : indique s'il s'agit d'un chemin pour accéder à l'emplacement ou pour en sortir

Ces fichiers contiennent l'ensemble des données utilisables pour chacune des variables. Ce seront les seules sources d'information que l'automate devra considérer.

Pour ce faire, l'automate devra donc intégrer un module de lecture de fichiers afin de récupérer les données nécessaires à son exécution.

En sortie, l'automate produira lui aussi un fichier CSV.

## 2 Contraintes liées aux données

Toute la problématique de ce projet consiste à trouver une solution optimale (ou s'en rapprochant) de répartition de ressources à des tâches. Dans le cas présent, les ressources sont les véhicules alors que les tâches sont les missions.

Cependant, un ensemble de contraintes restreint les possibilités de solutions.

Ces contraintes peuvent être divisées en deux grandes catégories : les contraintes générales, imposées par Keolis, et les contraintes implicites, liées aux données.

## 2.1 Contraintes fortes

Les contraintes générales sont :

- Tous les services doivent être affectés.
- Toutes les immobilisations impératives doivent être affectées.
- Les véhicules doivent pouvoir sortir de leur emplacement de stationnement.
- Les véhicules doivent pouvoir accéder à leur emplacement de stationnement.
- Les caractéristiques de véhicule correspondent aux critères de ligne.
- Les caractéristiques de véhicule correspondent aux critères d'emplacement.

Ce sont des contraintes fortes : toute solution ne respectant pas l'ensemble de ces contraintes sera invalide et ne pourra pas être proposée.

Par la suite, d'autres contraintes ont pu être identifiées.

Tout d'abord, certains services contiennent deux plages horaires distinctes. Ces deux sous-services peuvent ne pas être effectués par le même véhicule. De ce fait, les services de ce type seront divisés en deux missions. Ces nouvelles missions seront traitées comme n'importe qu'elle autre.

Deuxièmement, il fut envisagé de commencer par associer les missions aux emplacements de stationnement. En effet, les contraintes sur les emplacements sous surtout imposées par les horaires de départ et de retour de mission. Hors, ces horaires dépendent des missions en elles-mêmes et non des véhicules qui leur sont associées. Cette redéfinition permet de réduire le problème puisqu'un véhicule n'aura plus qu'à être associé à une mission, son stationnement étant maintenant implicitement défini par sa mission.

La notion d'inter-temps, définissant le temps séparant le retour de deux véhicules, devra être prise en compte ici, afin de rendre la solution la plus robuste qui soit. Il fut décidé qu'un inter-temps minimal de 10 minutes devra être respecté.

Cependant, les missions et emplacements étant liés, leurs autres contraintes le sont aussi. Ainsi, la liste des contraintes d'une mission (type de véhicule, énergie, etc.) et celle d'un emplacement (taille, etc.) fusionnent et le véhicule qui leur sera associé devra respecter l'ensemble de ces contraintes.

Parmi celles-ci, on retrouve :

- Le type de véhicule, devant correspondre à celui demandé par la mission.
- L'énergie du véhicule, devant correspondre à celle demandée par la mission.
- L'habillage BHNS, devant correspondre à celui demandé par la mission.
- La taille du véhicule, devant être inférieure à celle de l'emplacement prévu.

Toutes ces contraintes sont fortes et doivent donc être impérativement respectées.

La contrainte de la taille du véhicule est cependant moins stricte.

Il est à noter que, si pour un ou plusieurs critères, aucune condition n'est spécifiée par la mission, cela signifie que ce critère doit être ignoré et que n'importe quel véhicule conviendra.

Une immobilisation impérative est elle-aussi une contrainte forte. A noter tout de même que, le véhicule concerné par l'immobilisation étant unique et défini, celui-ci ne sera pas sujet à répartition parmi les autres missions.

De la même façon, les immobilisations relaxables doivent être considérées comme des contraintes fortes. La possibilité de les annuler devant être prise en compte uniquement dans le cas où aucune solution valide n'a pu être trouvée par l'automate.

## 2.2 Contraintes faibles

Parmi les contraintes plus faibles, on trouve notamment le potentiel kilométrique. L'objectif est qu'un véhicule effectue un nombre de kilomètres proche du potentiel kilométrique demandé.

Cependant, il s'agit d'une contrainte faible car le but est de s'en rapprocher le plus possible, sans pour autant attendre qu'il réalise strictement la distance souhaitée.

Les immobilisations planifiées font aussi partie des contraintes faibles.

Ces contraintes n'étant pas prioritaires, elles ne devront être considérées que dans un second temps. Ainsi, pour une mission, si plusieurs véhicules remplissent les conditions, ceux sans immobilisation planifiée devront être priorisés (si tous en ont, celui avec la plus petite priorité devra être choisi).

## 2.3 Contraintes relaxables

Les contraintes relaxables sont toutes les contraintes (fortes ou faibles) pouvant être abandonnées pour une raison ou une autre.

Dans l'absolu, toutes les contraintes peuvent être relaxables car, comme expliqué plus haut, l'automate n'a pour but que de fournir une proposition modifiable par des humains et non une solution stricte et définitive.

Cependant, l'objectif étant de proposer une solution viable, toutes les contraintes ne peuvent être abandonnées sans réflexion. Ainsi, si l'automate ne trouve pas de solution réalisable respectant toutes les contraintes imposées, il devra renoncer à certaines, dans un ordre pertinent.

Parmi les contraintes abandonnables en premier en compte :

- **Les immobilisations relaxables** : Étant d'anciennes immobilisations non-obligatoires, ces immobilisations doivent être les premières à être remises en question.
- **Les critères de ligne** : Bien que faisant partie des contraintes fortes, certains critères bus-lignes pourront être délaissés si besoin.
- **Les immobilisations obligatoires** : A ne considérer qu'en dernier recours.

## 3 Fonction objectif

La fonction objectif est un moyen de quantifier la qualité d'une solution et donc d'en comparer plusieurs entre elles.

Plusieurs fonctions objectif ont déjà été proposées par Charly Moreau.

La première consiste à maximiser le nombre d'immobilisations planifiées effectuées. Celles-ci étant optionnelles, on peut considérer qu'une solution en intégrant plus qu'une autre sera meilleure que cette dernière. Cependant, cette fonction ne mesure la « qualité » que sur des critères optionnels, et donc moins importants que les critères obligatoires.

Une deuxième idée de fonction objectif serait de respecter au mieux le potentiel kilométrique de chaque véhicule. Ainsi, un simple pourcentage (se rapprochant de la perfection à 100%) permettrait de comparer deux solutions.

Il serait aussi possible de baser notre comparaison sur l'inter-temps minimal d'une solution.

L'objectif serait donc de maximiser l'ensemble des inter-temps. Cependant, ne prenant en considération qu'un seul temps, il se peut qu'une solution particulièrement performante se voit déconsidérée pour un unique inter-temps jugé insuffisant.

Pour améliorer cette fonction objectif, il serait possible de baser notre comparaison sur la moyenne des inter-temps.

La difficulté de trouver une fonction objectif pertinente pour cet algorithme vient du fait que la plupart des contraintes sont obligatoires. De ce fait, les solutions ne remplissant pas les conditions demandées ne sont pas seulement « moins bonnes » mais sont tout simplement rejetées.

Les seuls critères d'évaluations restants sont donc des contraintes optionnelles, donc par définition moins importantes, qui en réalité n'influent que peu sur la qualité globale des solutions proposées.

Afin d'assouplir le problème et de faciliter la définition d'une fonction objectif, il fut décidé, dans un premier temps, de considérer tous les contraintes comme faibles.

De ce fait, toutes les solutions pourront être valides, y compris celles ne respectant pas les contraintes dites fortes. Cela est envisageable car l'automate ne sera qu'une aide à la décision et que les solutions proposées pourront être modifiées par un humain par la suite.

En prenant en compte ces nouvelles données, définir une nouvelle fonction objectif devient aisé.

Une première idée serait de compter le nombre de missions dont l'ensemble de critères est rempli. Un pourcentage de « missions valides » servirait donc de point de comparaison entre deux solutions.

De manière plus détaillée, on peut imaginer compter non plus les missions valides mais le nombre de critères remplis, toute mission confondue. Il serait de plus possible de pondérer chaque critère afin de donner plus de poids aux critères normalement forts, et moins aux faibles.



# Cahier de l'utilisateur

## 1 Installation

Le programme fut conçu de sorte à ne dépendre d'aucune librairie externe. Il suffit donc que l'utilisation possède Java sur sa machine (JRE 8 minimum téléchargeable ici : [Java SE Runtime Environment 8 Downloads](#)) et que celui-ci soit ajouté au PATH (variable d'environnement) de la machine.

Pour l'instant, aucune archive .jar ne fut générée, le projet n'étant pas finalisé à 100%. Lorsque ce sera le cas, il suffira de télécharger celui-ci sur la machine en question.

## 2 Utilisation

Si l'utilisateur possède Java 8 et le .jar du programme, il n'a plus qu'à exécuter ce dernier avec la commande suivante :

```
java -jar LGV-Automate.jar input output
```

"LGV-Automate.jar" devra être remplacé par le nom final du .jar à exécuter. "input" et "output" sont les chemins des dossiers d'entrée et de sortie. "input" devra contenir les fichiers CSV regroupant les données utilisées par l'automate alors que "output" sera le dossier où sera écrit le fichier de sortie.

Attention, les deux chemins sont obligatoires et doivent mener à des dossier, dans le cas contraire, le programme ne pourra pas s'exécuter. Il est aussi recommandé de mettre ces chemins entre guillemets.

# D

# Cahier du développeur

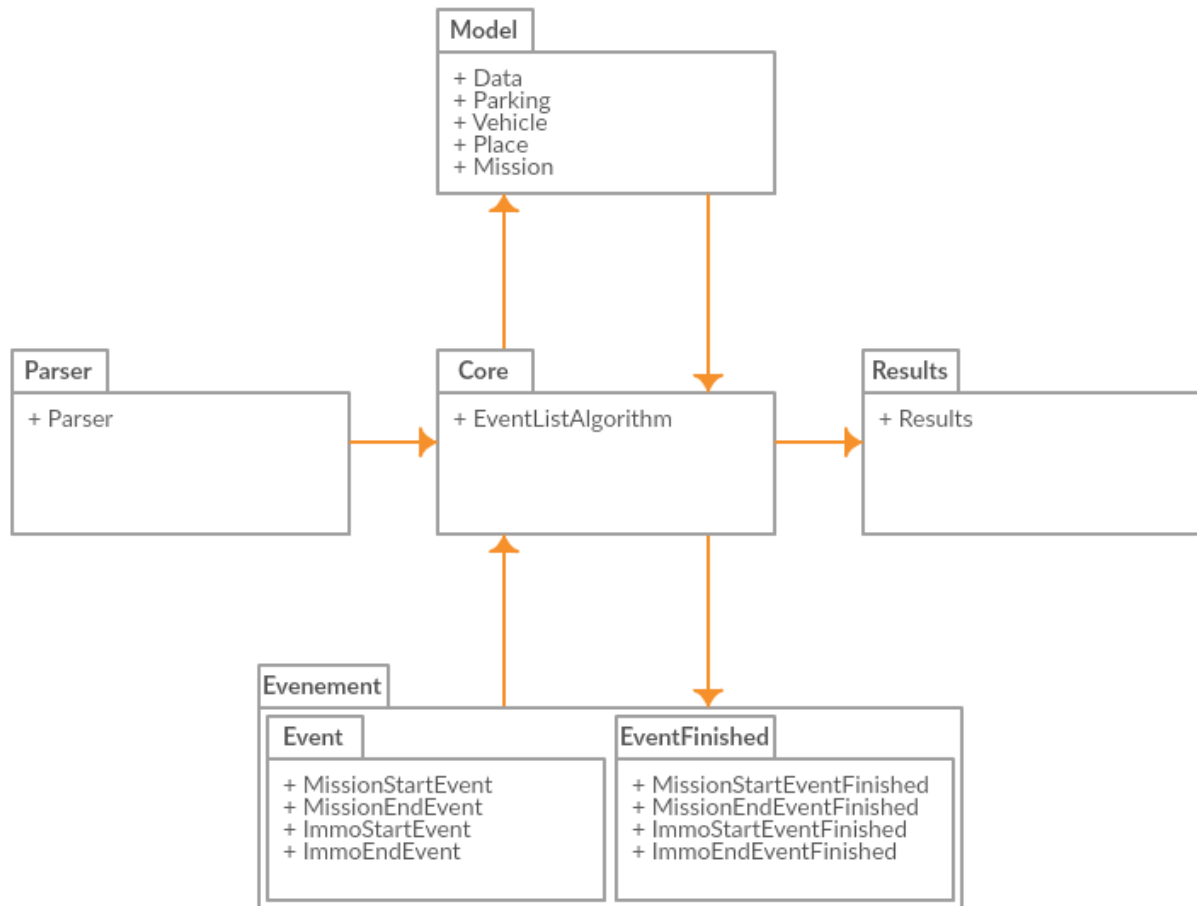
## 1 Environnement de développement

Le projet fut développé sous IntelliJ IDEA. Aucun outil de gestion de production ne fut utilisé afin de limiter au plus les dépendances. Dans la même optique, aucune librairie extérieure ne fut utilisée. Le programme peut donc se lancer avec un simple JRE 8 ( [Java SE Runtime Environment 8 Downloads](#)).

Cependant, la résolution des tests unitaires ne peut se faire qu'avec le framework JUnit4. Le fichier .jar nécessaire se trouve dans les sources du code fourni.

Le programme prend deux paramètres en entrée : le chemin d'accès au dossier contenant les fichiers CSV d'entrée et le chemin du dossier de sortie dans lequel sera écrit le fichier de résultats. Ce dernier fichier s'appellera "Resultats.csv" (ce nom est modifiable dans la méthode write() de la classe Results).

## 2 Structure du programme



**Figure 1** – Diagramme de package simplifié de l'automate

Le programme est séparé en cinq packages principaux, représentés sur le diagramme ci-dessus. Le package Core contient les classes centrales qui exécute l'algorithme principal. Avant de pouvoir se résoudre, l'automate a besoin de lire les données d'entrée grâce au package Parser. Le package Results servira en dernier lieu pour écrire dans un fichier de sortie les résultats produits.

Quant aux packages Evenement et Model, ceux-ci servent à la bonne résolution de l'algorithme en fournissant les classes et méthodes nécessaires aux stockages de données.

### 3 Description des classes

#### 3.1 Diagramme de classes





**Figure 2 – Diagramme de classe de l'automate**

## 3.2 Détails des classes

### 3.2.1 Classe EventListAlgorithm

Il s'agit de la classe principale et centrale du programme. Un unique objet de cette classe instancié dans la fonction Main est suffisant à l'exécution de l'automate.

#### Initialisation

Pour cela l'objet doit tout d'abord être initialisé grâce à sa méthode `init()`. Celle-ci prend deux paramètres : le chemin du dossier contenant les fichiers CSV d'entrée (input) et celui de sortie où sera écrit le fichier de résultats (output). Ces deux chemins sont normalement ceux passés en paramètres globaux du programme (variable "args" de la méthode Main).

Cette étape d'initialisation a pour but de lire les fichiers d'entrée et de traduire leur contenu en objets métier que l'automate utilisera par la suite afin de résoudre l'ordonnancement. De plus, l'ensemble des événements à exécuter par l'automate sera instancié à ce moment ainsi que le niveau de priorité globale des immobilisations.

Un troisième argument optionnel peut être donné à la méthode `init()`. Il s'agit d'un objet `Date` (Attention : il s'agit de la classe `Date` présente dans le package `model` et non de la classe par défaut du package `java.util`). Cette date représente le jour de début de l'ordonnancement. En effet, dans le fichier d'entrée `Mission.csv`, les missions devant être remplies ne sont pas définies par une date absolue (par exemple le 12/05/2018) mais par une date relative (jour 0, jour 1, jour 2, etc.). Ces dates relatives seront donc traduites en absolues à partir de celles passées en paramètre (le jour 0 deviendra donc le 12/05/2018, le jour 1 le 13/05/2018, etc.). Dans le cas où cet argument n'est pas fourni, la date courante servira de référentiel.

Grâce à cela, l'automate peut en théorie être utilisé pour planifier des missions lointaines, dans le cas où toutes les autres données (présence des véhicules, dates des immobilisations, etc.) concordent.

Enfin, il est important de prendre en compte qu'après la lecture des données l'automate supprimera tout événement précédant la date référentielle. Par exemple, si une mission devait être réalisée le 12/05/2018 à 08 :00 et que l'automate est lancé ce même jour à 09 :00 sans modifier la date référentielle, la mission de 08 :00 ne sera pas prise en compte dans la résolution du problème.

#### Résolution

Par la suite la résolution peut être lancée par l'appel à la méthode `run()`. Par défaut cette méthode peut ne prendre aucun paramètre, mais une surcharge à deux paramètres booléens fut créée.

Le premier paramètre définit si la méthode doit écrire dans le résultat dans le fichier de sortie (par défaut à `True`). Le second définit si la solution trouvée doit être testée par le module secondaire de validation des solutions (détaillé dans la [Section 3.2.5](#) - par défaut à `False`).

Dans tous les cas, la méthode `run()` commencera par lire les événements créés lors de l'initialisation et les résoudra dans l'ordre (Voir la [Section 3.2.4](#) pour le détail des événements).

Dans le cas où l'un des événements ne pourrait pas être résolu, le programme diminuera son niveau d'exigence en abaissant la priorité globale des immobilisations avant de retenter. Pour cela, il réinitialisera toutes les données en faisant de nouveau appel à la méthode `init()` puis parcourra tous les événements. Il est donc important de ne pas supprimer les fichiers d'entrée

de leur dossier avant la résolution complète de l'algorithme.

Dans le cas où l'automate ne peut trouver une solution après avoir baissé son niveau d'exigence au minimum, une exception de type `ImpossibleSituationException` sera lancée. Cela signifie qu'il n'existe aucune solution pour la situation définie par les données d'entrée.

### 3.2.2 Classe Data

La classe `Data` est la classe au sein de laquelle sont stockées toutes les données lues par le Parser.

#### Structure

Les deux premières données importantes stockées dans cette classe sont la liste des événements à exécuter (Voir [Section 3.2.4](#)) et celle des événements déjà exécutés (Voir ??).

La première est remplie à l'initialisation et la seconde se remplira durant la résolution de l'algorithme.

Deuxièmement, on retrouve l'ensemble des véhicules et des immobilisations lus dans les fichiers d'entrée. De la même façon, une liste unique des priorités des immobilisations et la priorité courante de l'exécution sont stockées.

Enfin, un objet de type `Parking` est présent. Celui-ci regroupe les places et leurs chemins respectifs lus dans les fichiers d'entrée.

#### Utilisation

Comme dit ci-dessus, la liste des événements à exécuter se videra au fur et à mesure que ceux-ci seront résolus. Il se peut que certains événements en créent d'autres. Ceux-ci seront alors ajoutés par date croissante à cette même liste. Dans tous les cas, l'algorithme se s'arrête que lorsque tous les événements sont résolus.

La liste des événements finis, elle, commence vide et se remplit à chaque événements exécutés. Celle-ci servira à la fin, par le module de validation (Voir [Section 3.2.5](#)) et celui d'écriture.

Concernant les listes de véhicules et d'immobilisations, celles-ci n'ont pas pour but d'entrée modifiée après initialisation. Elles ne servent qu'à stocker toutes les informations et non à représenter un quelconque état du système. Par exemple, lorsqu'un véhicule part en mission, il n'est pas retiré de cette liste.

L'objet `Parking` est quand à lui celui qui évoluera au cours de la résolution. Les places de parking et leur chemin d'entrée et de sortie ne changeront pas mais la présence ou non de véhicules garés sur ces places représentera l'état courant du système. (Voir [Section 3.2.3](#)).

### 3.2.3 Classe Parking

#### Structure

La classe `Parking` est composée de la liste des places lues par le Parser ainsi que la liste des véhicules ayant finis une immobilisation et étant donc sur un parking à part.

#### Utilisation

Lors de la création d'un objet `Parking`, celui-ci est vide. Il faut donc commencer par ajouter les places à l'aide de la méthode `addPlaces()` puis les chemins avec la méthode `addPaths()`. Ces

méthodes doivent être appelées dans cet ordre car les chemins donnés ne sont ajoutés que si la place correspondante est déjà présente.

Par soucis d'efficacité, les places sont en réalité stockées dans une structure de type Map dont les clefs sont les identifiants des places et les valeurs les objets Place eux-mêmes.

Pour les même raisons, le parking secondaire n'est qu'une liste des identifiants des véhicules qui y sont actuellement garés. Si l'on souhaite obtenir l'objet Vehicle complet, il faudra le chercher dans la liste des véhicules de l'objet Data.

Les méthodes présentes dans cette classe permettent principalement de savoir si une place est libre, accessible, occupée, etc.

### 3.2.4 Les événements

Il existe quatre types d'événements : MissionStartEvent, MissionEndEvent, ImmoStartEvent et ImmoEndEvent. Ces quatre classes héritent d'une ème classe abstraite Event.

#### La classe abstraite Event

Cette classe ne contient qu'un attribut Date et une méthode abstrite exec().

La date sert à représenter le moment auque l'événement se produira et servira donc à trier ceux-ci dans la liste des événements à résoudre.

La méthode exec() quant à elle sera le corps de chaque événement : les actions et modifications qu'il appliquera au parking.

#### MissionStartEvent

Représente le début d'une mission.

L'objectif de cet événement est d'attribuer un véhicule à une mission. Après avoir sélectionné le meilleur, il le supprimera de sa place de parking actuelle et créera un événement de type MissionEndEvent à la date de fin de mission prévue. De plus, un événement fini de type MissionStartEventFinished regroupant les informations de ce début de mission (mission, véhicule choisi, etc.) sera créé.

#### MissionEndEvent

Représente la fin d'une mission.

L'objectif de cet événement est d'attribuer une place au véhicule revenant de mission. Après avoir sélectionné la meilleure place, il ajoutera le véhicule à celle-ci. De plus, un événement fini de type MissionEndEventFinished regroupant les informations de cette fin de mission (mission, véhicule, place etc.) sera créé.

#### ImmoStartEvent

Représente le début d'une immobilisation.

L'objectif de cet événement est de vérifier qu'un véhicule est bien disponible pour l'immobilisation prévue. Après avoir vérifié que le véhicule concerné est bien présent sur l'un des parkings, il le supprimera de sa place de parking actuelle et créera un événement de type ImmoEndEvent à la date de fin d'immobilisation prévue. De plus, un événement fini de type ImmoStartEventFinished regroupant les informations de ce début d'immobilisation sera créé.

### ImmoEndEvent

Représente la fin d'une immobilisation.

Cet événement n'a que peu de choses à faire puisse qu'il lui suffit d'ajouter le véhicule finissant son immobilisation au parking secondaire. De plus, un événement fini de type ImmoEndEvent-Finished regroupant les informations de cette fin d'immobilisation sera créé.

### 3.2.5 Le module de validation

Le module de validation permet de vérifier qu'une solution proposée par l'automate est correct. Dans les faits, le module se basera sur la liste des événements finis et vérifiera pour chacun d'eux que la situation qu'il décrit est correct.

#### La classe CheckClass

Cette classe possède les méthodes vérifiant tous les types d'événements. Un attribut de type CheckParking permet de stocker l'état du parking à chaque instant.

Une instance de cette classe est créée à l'initialisation de l'objet EventListAlgorithm et l'objet CheckParking est calqué sur l'objet Parking de base passé en paramètre.

Lors de la vérification de chaque événement, les méthodes appelées modifieront l'objet CheckParking pour mettre à jour son état.

#### La classe CheckParking

Cette classe représente de manière simplifiée les deux parkings. Elle possède une liste d'objets de type CheckPlace. Ses méthodes permettent globalement de savoir si une place est occupée, libre, accessible, etc. ainsi que d'ajouter ou de supprimer un véhicule d'une place.

Le contenu de ses places sera modifié au cours de la validation de la solution.

#### La classe CheckPlace

Sert à représenter de manière simplifiée une place de parking.

Très simple, cette classe ne possède qu'un identifiant, l'identifiant de la place présente, l'identifiant de la place suivante et l'identifiant du véhicule garé.

Ses méthodes servent à ajouter ou supprimer un véhicule et à savoir si la place est libre ou non.

## 4 Description des fichiers d'entrée

Les fichiers d'entrée sont nécessaires au lancement de l'automate, même si ceux-ci sont vides. Ils doivent tous se trouver dans le même dossier dont le chemin d'accès sera le premier paramètre passé au programme.

Les fichiers sont actuellement lus de la manière suivante :

— Vehicule :

Id - Energie - Taille - Model - BHNS - Compteur Kilométrique - Date d'immo - Durée d'immo - Priorité d'immo

```

1 324;GO;3;ART;0;595677;;;1
2 325;GO;3;ART;0;576377;;;1
3 326;GO;3;ART;0;576451;;;1

```

Figure 3 – Exemple de fichier CSV des véhicules

## — Mission :

Id - Jour - Heure début - Heure fin - Distance - Model - BHNS - Energie - IdMainLine

```

1 1001;0;04:38;21:02;274.059;ART;1;;346
2 1001;1;04:38;21:02;274.059;ART;1;;346
3 1001;2;04:38;21:02;274.059;ART;1;;346

```

Figure 4 – Exemple de fichier CSV des missions

## — Emplacement :

Id - Taille - Id du véhicule garé

```

1 B1;0;
2 B2;0;292
3 B3;0;266

```

Figure 5 – Exemple de fichier CSV des emplacements

## — Cible kilométrique

Id du véhicule - Cible - Date cible

```

1 51;290928;14/05/2019;
2 51;300000;14/06/2019;
3 51;311776;28/07/2019;

```

Figure 6 – Exemple de fichier CSV des cibles kilométriques

## — Immobilisations :

Id - Date de début - Date de fin - Véhicule concerné - Priorité

```

1 179433;09/06/2017 19:49;07/07/2017 19:49;324;1
2 180575;03/07/2017 06:45;05/07/2017 06:45;325;1
3 181202;19/07/2017 08:00;19/07/2017 12:00;335;1

```

Figure 7 – Exemple de fichier CSV des immobilisations

## — Trajet :

Id de la place concernée - Chemin (liste des id des places à traverser) - "E" ou "S" (entrée / sortie)

```

1 R51;R55,R54,R53,R52;E;
2 R52;R55,R54,R53;E;
3 R53;R55,R54;E;

```

Figure 8 – Exemple de fichier CSV des trajets

Plusieurs cas particuliers sont à noter :

- Tous les véhicules présents dans le fichier "Vehicule.csv" seront lus et ajoutés aux données du programme. Cependant, seuls ceux étant garés au parking (leur id est sur une place dans le fichier "Emplacement.csv") sont considérés.

- Dans le fichier "Vehicule.csv", les trois dernière colonne sont optionnelles : elles peuvent ne pas être présentes, le programme ne crashera pas. Cependant, si elles sont présent mais que la date de l'immobilisation OU la durée de l'immobilisation n'est pas renseignée, l'immobilisation ne sera pas considérée (mais le véhicule en lui-même si).
- Il est possible que le fichier "Trajet.csv" soit totalement vide. Ce cas signifie que toutes les places du parking sont indépendantes et que le parking n'a plus aucune contrainte d'entrée et de sortie.

## 5 Description du fichier de sortie

Le fichier de sortie est d'une importance capitale puisque celui-ci sera interprété par un logiciel extérieur afin d'intégrer à son processus la solution fournie par l'automate. L'objectif étant qu'aucun humain n'est à intervenir entre les deux programmes, le fichier de sortie doit suivre un format bien particulier afin d'être lu sans erreur possible par la suite du système.

Pour des questions de manque d'information et de temps, le format de sortie attendu n'a pas pu être respecté, bien qu'il s'agisse tout de même d'un fichier CSV. A la place, une autre format, supposé temporaire, est actuellement implémenté. Celui-ci a pour but d'écrire tous les événements finis, par ordre chronologiques, et leurs informations importantes.

Ce format est le suivant : Identifiant de l'événement - Date - Identifiant principal - Identifiant secondaire

```
1 3;25/04/2017 00:00;154698;156
2 1;13/03/2019 04:38;1001;611
3 2;13/03/2019 07:58;123;11
```

Figure 9 – Exemple de fichier actuel de résultats

- **Identifiant de l'événement** 1 / 2 / 3 : Décrit le type d'événement écrit. Cela influence les identifiants principal et secondaire
  - 1 Début d'une mission
  - 2 Fin d'une mission
  - 3 Début d'une immobilisation
- **Date** Date à laquelle l'événement s'est produit
- **Identifiant principal** Dépend de l'événement :
  - Début de mission** Identifiant de la mission
  - Fin d'une mission** Identifiant du véhicule revenant
  - Début d'une immobilisation** Identifiant de l'immobilisation
- **Identifiant secondaire** Dépend de l'événement :
  - Début de mission** Identifiant du véhicule associé à la mission
  - Fin d'une mission** Identifiant de la place de parking choisie
  - Début d'une immobilisation** Identifiant du véhicule concerné

Ce format regroupe toutes les informations importantes, qui sont de toute façon contenues dans les objets EventFinished du programme, mais n'est pas celui attendu.

```
1 Immobilisation;181257_0;R11;02/08/2017 09:00;0;  
2 Vehicule;54;R11;02/08/2017 11:10;6;  
3 Vehicule;66;R11;02/08/2017 11:10;9;  
4 Service;5426_3;R11;02/08/2017 11:10;3;  
5 Vehicule;64;R11;02/08/2017 11:10;8;  
6 Service;5426_6;R11;02/08/2017 11:10;8;  
7 Vehicule;65;R11;02/08/2017 11:10;1;  
8 Service;5425_8;R11;02/08/2017 11:10;0;
```

**Figure 10** – Exemple de fichier de résultats tel qu'attendu

Il s'agit d'un point très important. Le projet ne pourra pas être considéré comme fini tant que le fichier de sortie n'aura pas le bon format. Il est donc capital que cela soit repris et finalisé.



# E

## Cahier de tests

L'objectif des tests est de vérifier que le programme fonctionne comme attendu. Cela passe par plusieurs étapes à plusieurs niveaux.

### 1 Tests unitaires

Le but des tests unitaires est de vérifier que chaque méthode s'exécute bien selon ce qui est prévu. Cela passe non seulement par la vérification des valeurs de retour mais aussi du comportement lorsque les paramètres sont exceptionnels (tels que des valeurs Null ou vide).

#### 1.1 DateTest

##### **testDate**

Vérifie que le constructeur à partir d'un String lit et comprend bien la bonne date. Se fait par comparaison avec une date créée à partir du constructeur général demandant une année, un mois, un jour, une heure et des minutes.

##### **testAddDays**

Vérifie que la méthode addDays ajoute bien le nombre de jours passé en paramètres à la date concernées sans modifier les unités plus petites (heures et minutes) et en ajustant les unités plus grandes (année, mois, jour) en cas de dépassement. Vérifie aussi que si le nombre passé est négatif, les jours sont retirés.

##### **testAddMinutes**

Vérifie que la méthode addMinutes ajoute bien le nombre de minutes passé en paramètres à la date concernées en ajustant les unités plus grandes en cas de dépassement. Vérifie aussi que si le nombre passé est négatif, les minutes sont retirées.

**testSetHourAndMin**

Vérifie que la méthode setHourAndMin prenant un String en paramètre lit et modifie correctement la date concernées.

Vérifie aussi le cas particulier où la nouvelle heure est entre 00 :00 inclu et 03 :59 inclu. Dans ce cas, l'heure doit être modifiée mais le jour doit aussi être augmenté de 1.

**testBefore**

Vérifie que la méthode before renvoie bien True dans le cas où la date passée en paramètres est après la date concernée et False dans le cas inverse.

**testAfter**

Vérifie que la méthode before renvoie bien True dans le cas où la date passée en paramètres est avant la date concernée et False dans le cas inverse.

**1.2 PlaceTest****testIsAvailable**

Vérifie que la méthode isAvailable renvoie bien True dans le cas d'un place libre (son attribut IdVehicule est à -1) et False dans tout autre cas. Une place libre peut être contruite avec le constructeur ne demandant pas l'id du véhicule déjà garé ou avec le constructeur le demandant si l'id passé est -1.

**testPutVehicle**

Vérifie que la méthode putVehicle ajout bien le véhicule demandé dans la place concerné. Dans le cas où la place est libre, vérifie que l'id du véhicule garé correspond bien à celui passé en paramètre et que la date d'arrivée est égale à celle donnée. Dans le cas où la place est déjà occupée, une exception IllegalStateException est lancée.

**testPollVehicle**

Vérifie que la méthode pollVehicle supprime bien à véhicule de la place concernée et que la méthode renvoie bien l'id du véhicule qui était garé.

Dans le cas où la place est libre, vérifie que la méthode lance bien une exception IllegalStateException.

**testGetNextPlace**

Vérifie que la méthode getNextPlace renvoie bien l'id de la place suivant celle concernée d'après son chemin de sortie. Dans le cas où la place n'a pas de chemin de sortie (place en bout de ligne), vérifie que la méthode renvoie bien Null.

**1.3 ParkingTest****testAddPlaces**

Vérifie que la méthode addPlaces ajout bien toutes les places passées en paramètres au parking. Vérifie le nombre de place ajoutées et les vérifie une par une qu'il s'agit des bonnes.

Vérifie aussi qu'aucune place n'est ajoutée lorsque la liste passée en paramètre est vide ou égale à Null.

**testAddPaths**

Vérifie que la méthode addPaths ajoute bien l'ensemble des chemins passé en paramètre aux places associées. Vérifie aussi que dans le cas où la liste est vide ou Null, aucun chemin n'est ajouté.

Vérifie que si un chemin à ajouter ne correspond à aucune place dans le parking, celui-ci est ignoré.

**testGetPlace**

Vérifie que la méthode getPlace renvoie bien la place correspondant à l'id demandé. Dans le cas où l'id passé en paramètre ne correspond à aucune place, renvoie Null.

**testIsReachable**

Après ajout d'une structure de parking simple, contenant déjà quelque véhicule, vérifie que pour chaque place, la méthode isReachable renvoie True si toutes les places dans son chemin d'entrée sont vide, sinon False.

**testGetReachablePlaces**

Après ajout d'une structure de parking simple, contenant déjà quelque véhicule, vérifie que la méthode getReachablePlaces renvoie bien la liste des id des places vérifiant la condition isReachable décrite plus haut.

**testGetReachableAndLastPlaces**

Après ajout d'une structure de parking simple, contenant déjà quelque véhicule, vérifie que la méthode getReachableAndLastPlaces renvoie bien les places passant la conditions isReachable et étant les dernières libre dans leur chemin.

**testFindVehicle**

Vérifie que la méthode findVehicle retourne bien la place sur laquelle le véhicule demandé se trouve. Vérifie aussi que si l'id du véhicule cherché est -1 (signifiant normalement que la place est libre, Null est retourné. De même dans le cas où aucun véhicule ne correspond à l'id demandé.

**testGetAvailableVehiclesId**

Vérifie que la méthode getAvailableVehiclesId renvoie bien la liste de tous les véhicules qui peuvent sortir du parking à l'instant où la méthode est appelée. Vérifie la taille attendue de la liste ainsi que chacun des véhicules attendus.

**testCanGetOut**

Après ajout d'une structure de parking simple, contenant déjà quelque véhicule, vérifie pour chaque véhicule que la méthode canGetOut renvoie bien True si celui-ci est le dernier de son chemin de sortie, et renvoie False dans le cas contraire.

**testGetInterTime**

Après l'arrivée d'un nouveau véhicule sur le parking, vérifie que la méthode getInterTime renvoie bien la différence de temps entre la date courante et la date d'arrivée du dit véhicule.

## 2 Tests de validation

L'objectif des tests de validation est de vérifier que le programme vérifie les exigences définies dans le cahier des charges et le cahier de spécifications. De plus, ils ont pour but de vérifier la fiabilité et les performance du système.

Dans notre cas, les tests de validation ont été conduit de la façon suivante : plusieurs instances ont été passées en entrée (différentes tailles de fichiers et de données) afin de voir comment l'automate réagissait à différents volumes d'information à traiter mais aussi à des cas plus pathologiques.

### 2.1 Petites instances

Nombre de véhicules	≈ 20
Nombre de missions	≈ 40
Nombre d'immobilisations	≈ 5
Nombre de places de parking	≈ 40
Réussite	Oui
Temps d'exécution	≈ 0.8 seconde

Ces premières instances des test ont été fournies par Keolis. Cependant, leur taille et structure correspondrait plus à une situation liée au tramway de la ville et non au parc de bus. De ce fait, le cohérence du résultat n'a, dans ce cas, que peu d'importance. L'intérêt de cette instance est donc surtout de vérifier que l'ensemble des algorithmes (sélection d'un véhicule, d'une place, etc.) fonctionnent, que la fonction d'écriture génère bien un fichier de sortie et d'estimer la durée d'exécution du programme.

En ce qui concener le temps d'exécution, une petite instance met environ 0.8 seconde à être résolue, ce qui, au vu des résultats des précédents programmes et de l'utilisation potentielle de l'automate, est satisfaisant.

On voit aussi pour chaque instance, l'automate réussit à trouver une solution. Cela s'explique par le fait qu'il y a peu de véhicule mais surtout peu de missions à réaliser et peu d'immobilisations pouvant gêner la planification.

### 2.2 Grandes instances

Nombre de véhicules	≈ 80
Nombre de missions	≈ 160
Nombre d'immobilisations	≈ 20
Nombre de places de parking	≈ 160
Réussite	Oui
Temps d'exécution	≈ 1.2 seconde

Ces instances ont été générées par multiplication des petites. Dans notre cas, leur volume fut multiplié par environ 4 afin de ressembler à des instances de taille réelle. Bien sur, la cohérence de données fut vérifiée au sein de ces instances (pas deux véhicules de même identifiant, pas deux véhicules sur la même place de parking, etc.).

Tout d'abord, on remarque que dans tous les cas, l'automate trouve une solution. Cela est du au fait que si les petites instances ne provoquaient pas de situation bloquante, les grandes n'auront elles aussi pas de situation problématique provoquant l'arrêt du programme.

Ces tests ont tout de même permis de repérer certaines erreurs mineures au sein de l'algorithme de sélection de véhicule pour une mission et de les corriger.

Enfin, le temps d'exécution a augmenté mais pas de façon linéaire : la multiplication par 4 de la taille des instances n'a pas multiplié par 4 le temps d'exécution.

### 2.3 Instances personnalisées

Nombre de véhicules	≈ 120
Nombre de missions	≈ 350
Nombre d'immobilisations	≈ 90
Nombre de places de parking	≈ 125
Réussite	Parfois
Temps d'exécution	≈ 1.5 seconde

Ces instances furent générées par un second programme. Des trois types d'instances testées, ce sont les plus importantes car, du fait de leur génération totalement aléatoire, elles peuvent représenter des situations dites pathologiques (cas particuliers). L'objectif est donc de voir comment l'automate réagit en présence de cas spécifiques, rares ou impossibles.

Ces différents cas ont permis de mettre en évidence plusieurs points d'intérêt :

- Dans le cas d'instances générées complètement aléatoirement, le taux moyen de respect des critères d'une mission par le véhicule associé chute. Cela vient du fait qu'il y a peu de chance de trouver un véhicule parfait pour une mission quand il y a 5 critères à respecter et que chacun de ces critères peut prendre 2 ou 3 valeurs. La forme du parking a aussi un impact puisqu'elle influe sur le nombre de véhicules disponibles pour une mission.
- Une cause principale d'échec de résolution est la forme et la taille du parking. En effet, si celui-ci est composé de peu de lignes, chacune contenant un grand nombre de places, l'automate arrivera assez vite à une situation où aucune place n'est disponible pour garer un véhicule rentrant de mission. Cela s'explique par le fait que, si en début de journée, le parking est presque plein et que tous les véhicules ne partent pas en mission, à leur retour, les véhicules restant au fond du parking bloqueront le retour de ceux qui sont partis. Cette situation peut être résolue en intégrant une mécanique d'avancement des véhicules en cours de journée. Cela reste tout de même à vérifier auprès de la partie métier.

Enfin, on remarque toujours que le temps d'exécution reste tout à fait correct malgré la taille conséquente des instances testées.

## 2.4 Instances réelles

Nombre de véhicules	$\approx 160$
Nombre de missions	$\approx 720$
Nombre d'immobilisations	$\approx 50$
Nombre de places de parking	$\approx 140$
Réussite	Parfois
Temps d'exécution	$\approx 1.2$ seconde

Ces instances ont été fournies par Keolis et représentent donc des situations réelles courantes. Il est bien évidemment pertinent de tester le programme avec de telles instances car ce sera le type de situations que l'automate rencontrera le plus souvent. Cependant, ne pouvant tester une infinité de cas, ces seules instances ne sont pas suffisantes car elles ne peuvent pas représenter des cas particuliers que l'autoate serait amené à rencontrer. L'utilisation des intances "virtuelles" testées précédemment est donc primordiale.

# F

# Gestion de projet

## 1 Diagramme de Gantt

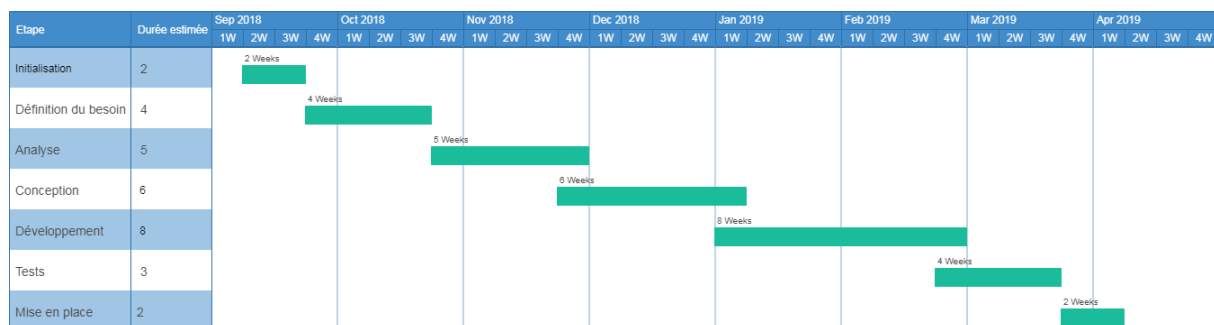


Figure 1 – Diagramme de Gantt initialement prévu

## 2 Initialisation

La première étape du projet consista à prendre connaissance du sujet et de ses enjeux.

Faisant suite à un projet collectif, un stage et un PRD réalisés par Charly Moreau sur un sujet similaire, la période d'initialisation fut aussi l'occasion de lire les différents documents produits.

## 3 Définition du besoin

La phase de définition du besoin consista en la prise de connaissance de l'ensemble des attentes de Keolis vis-à-vis de ce PRD. Cela concerna la définition de la problématique, les contraintes propres au problème que l'environnement préexistant auquel le résultat produit devra s'intégrer et les fonctionnalités attendues.

Pour ce faire, plusieurs réunions et échanges furent organisés avec M. Kergosien et M. Fabre afin de cerner au mieux toutes les implications et subtilités du sujet.

## 4 Analyse

Cette partie consista principalement en la formalisation d'un algorithme de résolution du problème. Pour cela, les précédentes analyses de Charly Moreau furent utiles afin d'éviter un certain nombre de pièges qu'il avait déjà identifiés.

La mise à plat réelle du problème permit de soulever quelques points importants de définition du besoin qui aurait pu ralentir le développement par la suite.

## 5 Conception

La partie conception consista à concevoir un ensemble de classes Java afin de coller au mieux aux besoins de l'algorithme précédemment imaginé. Il fut aussi question de la récupération des résultats finaux et des modifications que cela apporte à l'algorithme.

Plusieurs autres problématiques se sont ajoutées au cours de la conception telles que la possibilité de tester une solution une fois qu'elle sera générée, la gestion des données d'entrée ainsi que la structure interne du programme stockant et utilisant les données.

Cette partie fut légèrement plus rapide que prévue, le programme n'étant qu'une mise en application d'un algorithme précédemment défini et ne faisait pas appel à des éléments extérieurs (système de base de données, connexion internetn ou autre).

## 6 Développement

La phase de développement consiste à écrire réellement le code conçu lors de la phase précédente. Dans les faits, plusieurs éléments perturbateurs sont venus ralentir le développement.

De plus, la volonté d'écrire un programme libre de toute librairie extérieure à Java 8 contraint à réécrire certaines classes utilitaires que l'on aurait simplement pu importer de packages extérieurs.

## 7 Tests

La phase de tests consiste à vérifier que l'ensemble des composants fonctionnent comme prévu. Cela intervint à plusieurs niveaux : les tests unitaires vérifiant que chacune des méthodes s'exécute correctement et les tests de validations vérifiant la qualité des solutions trouvées par l'automate.

Concrètement, cette phase fut la plus longue à réaliser et celle qui provoqua le plus de retard. Plusieurs éléments peuvent expliquer cela. Premièrement, un grand nombre de fichiers d'entrée furent testés afin de simuler à maximum de situations pouvant être problématiques. Deuxièmement, lorsqu'on remarque qu'une solution n'est pas valide, identifier l'erreur dans un premier temps puis trouver la cause du problème dans un second peut être extrêmement long en fonction de la taille de l'instance testée.

Dans les faits, la phase de développement était prévu sur une durée de 8 semaines mais fut réalisée en 6. Cependant, cette avance fut perdue car la phase de tests estimée à 3 semaines dura aux alentours de 6 semaines.



## 8 Mise en place

Par manque de temps du au retard accumulé sur la phase de tests, cette partie ne fut pas réalisée. Le programme fut donc transmit en l'état aux équipes de développement de Keolis afin que celles-ci finalisent et surtout déploient l'automate.

## 9 Diagramme de Gantt revisité

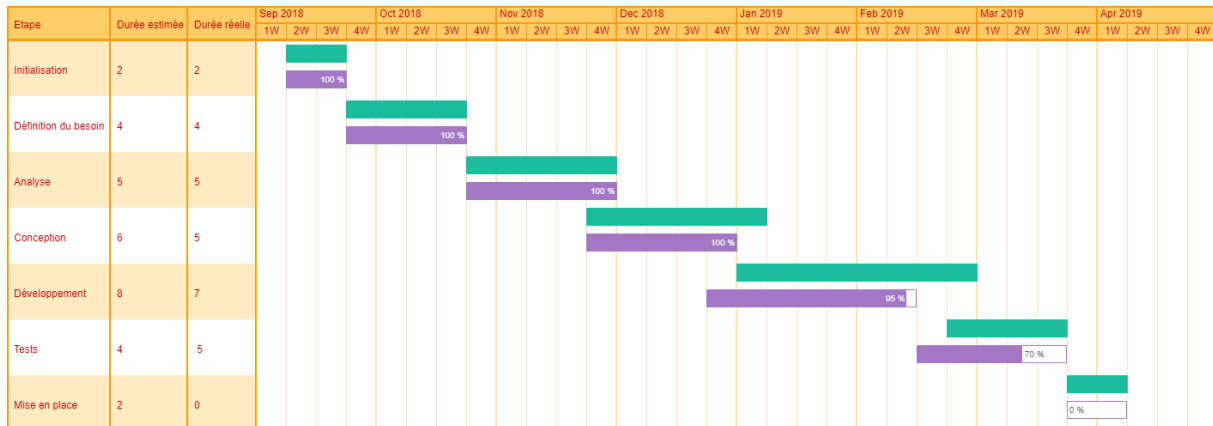


Figure 2 – Diagramme de Gantt avec niveaux de complétion

Comme on peut le voir sur le diagramme ci-dessus, la première moitié fut réalisée dans les temps et complètement. Cependant, dans la seconde moitié, la partie tests prit beaucoup de retard car plusieurs problèmes ont été identifiés. Cela a eut pour conséquences de faire revenir le projet sur la partie développement (Voir le modèle en cascade [Figure 1](#) (Chapitre 2)), qui ne peut donc plus être considéré comme achevé à 100%.

Par soucis de temps, la phase de tests n'a pas pu être terminée. De ce fait, la mise en place au sein du système de Keolis ne fut pas réalisée.



# Comptes rendus hebdomadaires

## **Compte rendu n°1 du 20/09/2018**

19/09/2018

- Prise de connaissance du sujet
- Prise de connaissance des documents partiels réalisés par l'étudiant précédemment en charge du sujet

20/09/2018

- Rédaction d'un document regroupant l'ensemble des informations comprises tirées des documents partiels

## **Compte rendu n°2 du 27/09/2018**

26/09/2018

- Rencontre avec Sylvain Fabre, encadrant entreprise du projet, dans le locaux de Keolis

27/09/2018

- Début de rédaction du cahier des charges suite à la rencontre avec l'encadrant entreprise
- Début de rédaction du cahier des spécifications suite à la rencontre avec l'encadrant entreprise

## **Compte rendu n°3 du 04/10/2018**

03/10/2018

- Finalisation du cahier des charges
- Finalisation du cahier des spécifications

04/10/2018

- Début de rédaction du rapport S9
  - Début Introduction
  - Début Contexte de réalisation (avec diagramme de séquence)
  - Début Description générale
  - Début Analyse et conception

**Compte rendu n°4 du 11/10/2018**

10/10/2018

- Début de rédaction État de l'art
- Début d'étude du code de Charly Moreau

11/10/2018

- Mise au point avec M. Kergosien et M. Fabre.
- Mise à jour du rapport suite à la rencontre du matin

**Compte rendu n°5 du 18/10/2018**

17/10/2018

- Mise à jour du rapport (cahier de spécifications)
- Finition de l'état de l'art

18/10/2018

- Bibliographie
- Condition de fonctionnement
- Diagramme modèle en « cascade »
- Structure générale du système

**Compte rendu n°6 du 24/10/2018**

24/10/2018

- Réunion avec M. Ragot (questions sur le rapport)
- Mise à jour du rapport suite à la réunion

25/10/2018

- Début d'analyse et de reprise du code de CHarly Moreau

**Compte rendu n°7 du 31/10/2018**

31/10/2018

- Pause pédagogique

01/11/2018

- Pause pédagogique

**Compte rendu n°8 du 07/11/2018**

07/11/2018

- Mise à jour et complétion du rapport
- Développement des fonctionnalités de lecture de l'automate

08/11/2018

- Réunion avec M. Kergosien au sujet de l'algorithme de l'automate
- Rédaction de la partie 'Analyse' du rapport suite à la découverte de l'algorithme

**Compte rendu n°9 du 14/11/2018**

14/11/2018

- Mise à jour et complétion du rapport (analyse)

15/11/2018

- Mise à jour et complétion du rapport (conception)
- Développement des fonctionnalités de lecture de l'automate

**Compte rendu n°10 du 21/11/2018**

21/11/2018

- Développement de la partie model de l'automate

22/11/2018

- Mise à jour et complétion du rapport (conception)
- Relecture, correction et complétion du rapport

**Compte rendu n°11 du 28/11/2018**

28/11/2018

- Développement de la partie événement de l'automate
- Finalisation du rapport du S9

29/11/2018

- Préparation de la présentation du S9

**Compte rendu n°12 du 05/12/2018**

05/12/2018

- Préparation de la présentation du S9

06/12/2018

- Préparation de la présentation du S9

**Compte rendu n°13 du 09/01/2019**

09/01/2019

- Développement de la partie événement de l'automate

10/01/2019

- Développement de la partie algorithme de l'automate

**Compte rendu n°14 du 16/01/2019**

16/01/2019

- Développement de la partie algorithme de l'automate

17/01/2019

- Développement de la partie algorithme de l'automate

**Compte rendu n°15 du 23/01/2019**

23/01/2019

- Développement de la partie test de l'automate

24/01/2019

- Développement de la partie test de l'automate

**Compte rendu n°16 du 30/01/2019**

30/01/2019

- Première utilisation de la partie test et debuggage

31/01/2019

- Création de données plus importantes pour des tests plus conséquents
- Debuggage sur les nouvelles données d'entrée

#### **Compte rendu n°17 du 06/02/2019**

06/02/2019

- Avancé du rapport du S10

07/02/2019

- Debuggage sur les nouvelles données d'entrée
- Avancé du rapport du S10

#### **Compte rendu n°18 du 13/02/2019**

13/02/2019

- Création d'un second programme de génération automatique de fichier test

14/02/2019

- Création d'un second programme de génération automatique de fichier test

#### **Compte rendu n°19 du 20/02/2019**

20/02/2019

- Création d'un second programme de génération automatique de fichier test

21/02/2019

- Test et debuggage avec les nouveaux tests

#### **Compte rendu n°20 du 27/02/2019**

27/02/2019

- Test et debuggage avec les nouveaux tests

28/02/2019

- Avancé du rapport du S10

#### **Compte rendu n°21 du 06/03/2019**

06/03/2019

- Avancé du rapport du S10

07/03/2019

- Avancé du rapport du S10

#### **Compte rendu n°22 du 13/03/2019**

13/03/2019

- Ecriture des tests unitaires
- Avancé du rapport du S10

14/03/2019

- Avancé du rapport du S10



# Automate pour logiciel d'affectation de véhicules en temps réel

**Victor Coleau**

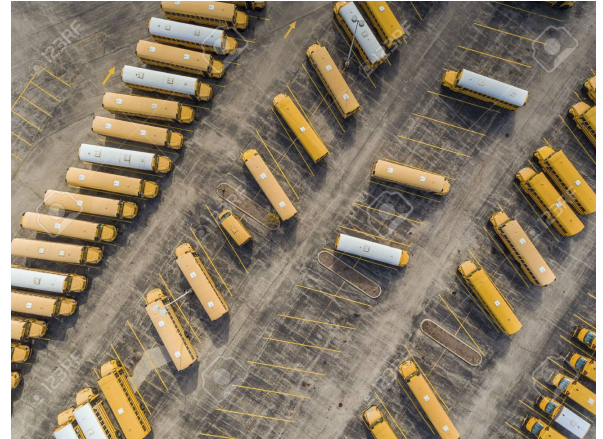
**Encadrement : Yannick Kergosien**

En collaboration avec Keolis

## Objectif

L'objectif est de concevoir un automate pouvant attribuer des bus de ville à des missions en tenant compte d'un certain nombre de critères tels que l'énergie utilisée, le modèle du véhicule, ces période de maintenance, etc.

Afin de rendre son utilisation la plus transparente possible, celui-ci devra pouvoir s'intégrer parfaitement dans le système préexistant en utilisant les mêmes structures de données.

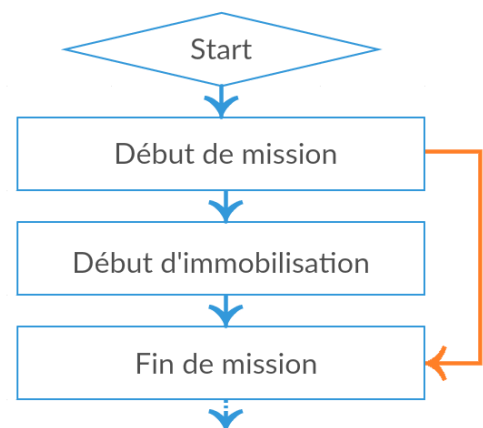


*Vue aérienne d'un parking de bus*

## Algorithme

L'algorithme choisi est de type événementiel.

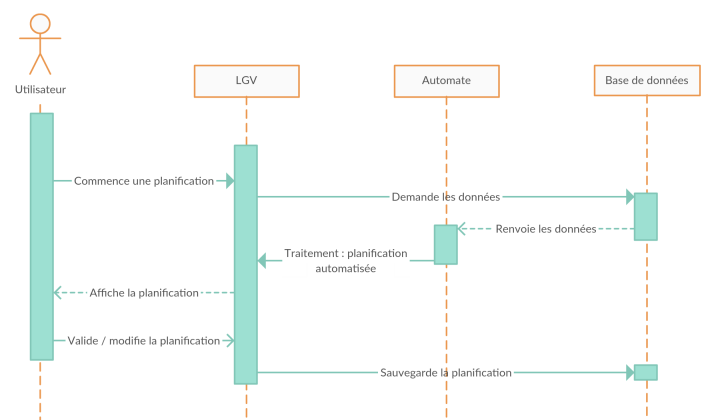
Chaque événement important d'une période de planification (début de fin de mission, de maintenance, etc.) est représenté par un événement. Une fois la liste finalisée et ordonnée chronologiquement, l'automate peut les résoudre de manière séquentielle et ainsi résoudre le problème global.



*Algorithme événementiel à liste*

## Performance

A terme, le but est de pouvoir exécuter plusieurs fois par jour l'automate afin de répondre rapidement aux aléas de la réalité de réseau par un nouvel ordonnancement. Pour cela, celui-ci doit donc fournir des solutions viables, quelque soit la situation, et dans un temps acceptable.



*Schéma d'utilisation de l'automate*

# Automate pour logiciel d'affectation de véhicules en temps réel

Victor Coleau

Encadrement : Yannick Kergosien

En collaboration avec Keolis

## Objectif

L'objectif est de concevoir un automate pouvant attribuer des bus de ville à des missions en tenant compte d'un certain nombre de critères tels que l'énergie utilisée, le modèle du véhicule, ces période de maintenance, etc.

Afin de rendre son utilisation la plus transparente possible, celui-ci devra pouvoir s'intégrer parfaitement dans le système préexistant en utilisant les mêmes structures de données.

## Algorithme

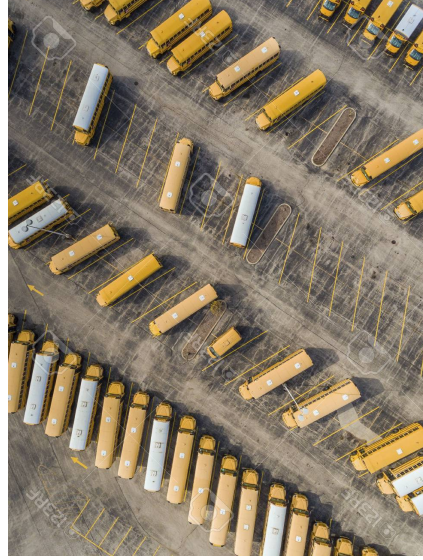
L'algorithme choisi est de type événementiel.

Chaque événement important d'une période de planification (début de fin de mission, de maintenance, etc.) est représenté par un événement. Une fois la liste finalisée et ordonnée chronologiquement, l'automate peut les résoudre de manière séquentielle et ainsi résoudre le problème global.

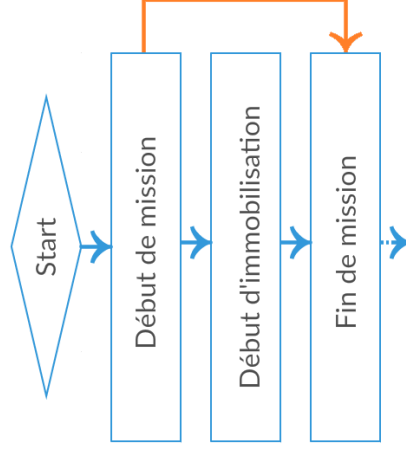
## Performance

A terme, le but est de pouvoir exécuter plusieurs fois par jour l'automate afin de répondre rapidement aux aléas de la réalité de réseau par un nouvel ordonnancement.

Pour cela, celui-ci doit donc fournir des solutions viables, quelque soit la situation, et dans un temps acceptable.



Vue aérienne d'un parking de bus



Algorithme événementiel à liste

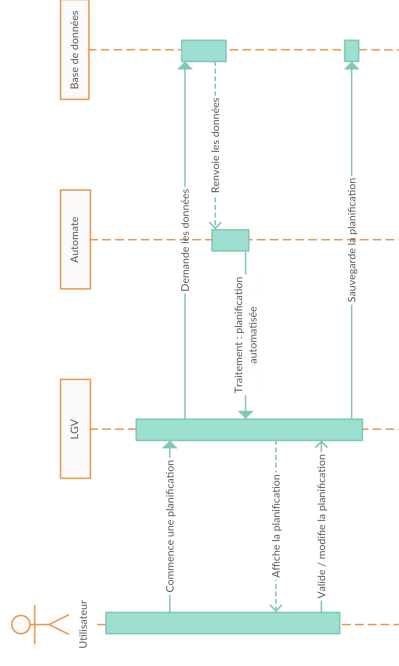


Schéma d'utilisation de l'automate



# Automate pour logiciel d'affectation de véhicules en temps réel

## Résumé

Création d'un automate permettant l'attribution de missions à un parc de véhicules en fonction de critères et l'attribution de places de parking aux véhicules.

## Mots-clés

Automate, Java, Transport, Algorithme

## Abstract

Creation of a robot to assign missions to a fleet based on criteria and to allocate parking spaces to vehicles.

## Keywords

Automate, Java, Transport, Algorithm

**Entreprise**

Keolis

**Tuteur entreprise**

Sylvain FABRE

**Étudiant**

Victor COLEAU (DI5)

**Tuteur académique**

Yannick KERGOSIEN