

LO43 Flotte de véhicule autonomes

Buri Theo Florian Lacour

Table des matières

Introduction	3
I Présentation du sujet	4
1.2 Objectif	5
1.3 Reformulation du sujet	5
1.3.1 Plateau	5
1.3.2 Passagers et contrôleur	5
1.3.3 Contraintes et libertés prises sur le sujet	6
II Conception UML	7
1 Les Diagramme UML	8
1.1 Diagramme des cas d'utilisation (annexe A)	8
1.2 Diagramme de séquence(annexe B)	8
1.3 Diagramme de classes(annexe C)	8
2 Description des classes	9
2.1 Classe BoiteAuxLettre	9
2.2 La classes Contrôleur	9
2.3 Les classes Requêtes	10
2.4 La classe Cerveau	10
2.5 La Classe Flotte de Véhicules	10
2.6 La Classe Véhicule	10
III Implémentation Java	11
1 Choix d'architecture	12
1.1 Le pattern MVC	12
1.2 Des Threads pour régir les déplacements	12
1.3 Un mot sur la génération automatique des requêtes	12
2 Fonctions importantes	13
2.1 Classe Controleur	13
2.2 Classe Vehicule	14
2.3 Classe Flotte de Véhicules	15
2.4 Classe Boite au lettres	16
2.5 Classe Cerveau	16
3 Difficultés rencontrées	18

Introduction

Dans le cadre de l'UV LO43 " Bases fondamentales de la programmation orientée objet", il nous a été demandé de réaliser un projet de groupe, afin de mettre en pratique les connaissances acquises lors des cours et TDs du semestre.

Trois sujet nous ont été présentés. Nous avons fait le choix de traiter le sujet de la "Flotte de véhicules autonomes", et ceci pour plusieurs raisons :

- Premièrement, ce sujet nous semblait être le plus à même d'être modifié, amélioré, contrairement aux deux jeux de plateau, aux règles déjà préétablies...
- Ensuite, n'étant que deux, sur un maximum de quatre étudiants par groupe autorisés, les autres sujets ne nous ont pas paru réalisables en temps et en heures et sans bugs majeurs...

Nous présenterons tout d'abord le sujet, ses contraintes, et les libertés prises par rapport à celles-ci. Par la suite, nous parlerons des différents diagrammes UML, et les expliquerons. Enfin, nous terminerons par l'implémentation en Java et l'interface graphique.

Première partie

Présentation du sujet

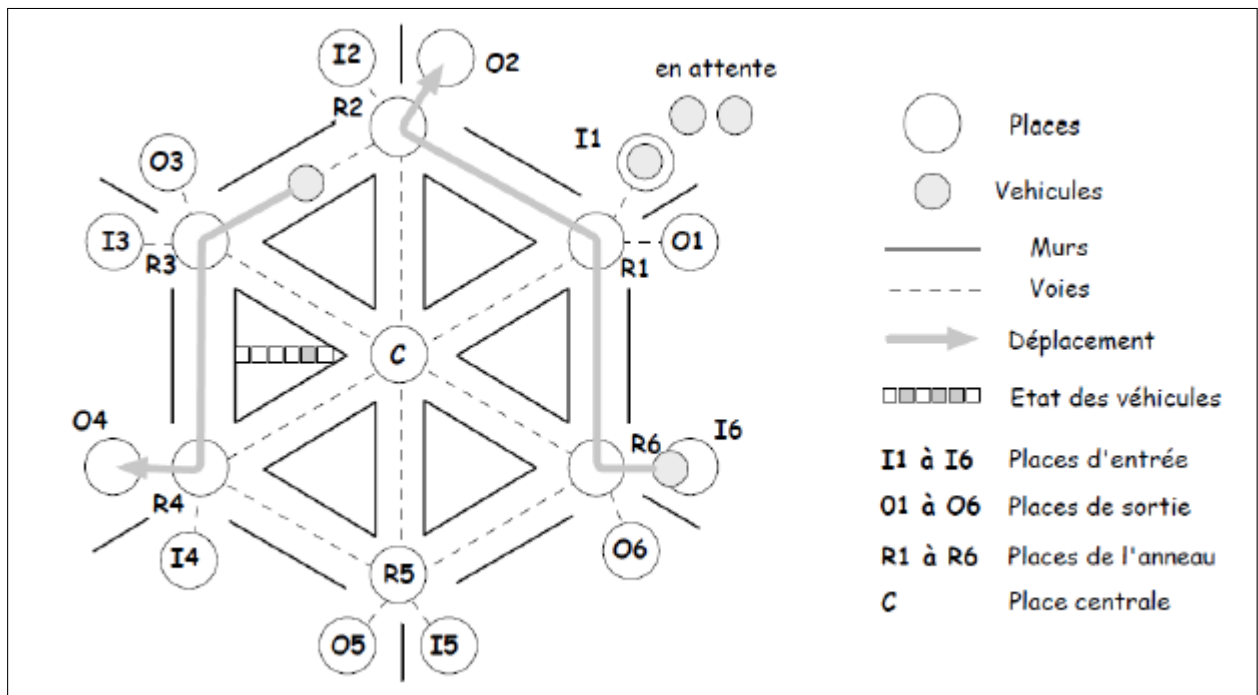
1.2 Objectif

Le programme à réaliser consistait en la modélisation d'une flotte de véhicules évoluant dans une infrastructure de circulation partagée. Pour cela, il a tout d'abord fallu modéliser la partie calculatoire à l'aide du langage UML, puis ensuite l'implémenter en Java et lui donner une interface graphique

1.3 Reformulation du sujet

1.3.1 Plateau

Le plateau donné par le sujet est le suivant :



Chaque place, exceptée celle du centre, est rattachée à une place de départ et une place de sortie. Les voitures disposant d'un passager partent depuis une place d'entrée (que celui-ci leur donne) et, à la fin de celle-ci, rejoignent une place de sortie (même chose).

La place centrale n'est utilisée que lors d'un trajet reliant deux places opposées...

1.3.2 Passagers et contrôleur

Chaque passager doit être transporté d'une place à une autre. Pour ce faire, il envoie une requête au module qui se charge du contrôle du plateau : le contrôleur. Par la suite, celui-ci assignera cette "mission" à un des véhicules de sa flotte (sous la forme d'un passager, ayant un départ et une arrivée). Celui-ci calculera ensuite le trajet qu'il suivra pour la mener à bien.

C'est alors le véhicule qui décide de partir : pour cela il doit envoyer une requête au contrôleur pour savoir si son chemin est déjà réservé (c'est à dire, si une des places qu'il souhaite utiliser a déjà été réservée par un autre véhicule). Si ce n'est pas le cas, le contrôleur l'y autorise, et le véhicule part. La requête se fait sous la forme d'une Request Map, constituée de booléens indiquant l'intention du véhicule de réserver une place ou non.

Pour exemple, lorsqu'une voiture désire aller de la place I1 à O2, elle aura besoin de réserver I1, R1, R2 et O2. La Request Map sera alors la suivante :

I1	I2	I3	I4	I5	I6	R1	R2	R3	R4	R5	R6	O1	O2	O3	O4	O5	O6	C
T	F	F	F	F	F	T	T	F	F	F	F	F	T	F	F	F	F	F

Afin d'éviter toute erreur lors de la réservation, nous avons défini la priorité des places qui suit :

$$I1 < I2 < I3 < I4 < I5 < I6 < B1 < B2 < B3 < B4 < B5 < B6 < O1 < O2 < O3 < O4 < O5 < O6 < C$$

Ceci permettra d'éviter que deux véhicules tentent de réserver la même place à deux moments différents, et que le contrôleur les valide quand même...

1.3.3 Contraintes et libertés prises sur le sujet

S'ajoutent alors quelques contraintes, qui régissent le programme :

- Les véhicules ne peuvent utiliser la place centrale qu'à la seule condition qu'ils doivent se rendre à la place en face de la leur.
- Une place ou une route (reliant deux places) ne peut être utilisée que par un seul et unique véhicule.
- Un véhicule qui souhaite emprunter un chemin occupé doit attendre que celui-ci se libère.

Ainsi, par rapport à ces différentes consignes, nous avons choisi de prendre certaines libertés :

- Un véhicule ne peut pas contourner une place occupée pour mener sa mission à bien.
- Un passager, lors de sa demande, est placé en liste d'attente. Il se voit alors immédiatement embarqué dans un véhicule, et attends le départ.
- S'il n'y a aucun véhicule disponible, il attends qu'un véhicule se libère et le prenne en charge.
- Lors de la fin de sa mission (débarquement du passager), le véhicule ne reste pas sur sa place. Il est "libéré" (il rejoint les véhicules libres), et sera donc capable de partir de n'importe quelle place de départ lors de sa prochaine mission (et non pas uniquement de celle correspondant à la place d'arrivée de sa mission précédente). Ceci permet au véhicule d'éviter d'effectuer un trajet supplémentaire en début de mission pour aller chercher le passager sur sa place de départ...

Deuxième partie

Conception UML

Chapitre 1

Les Diagramme UML

1.1 Diagramme des cas d'utilisation (annexe A)

Le diagramme des cas d'utilisation permet de représenter les actions que l'utilisateur peut faire. Dans notre projet, en lançant le programme, l'utilisateur a plusieurs solutions :

- Voir les crédits (c'est à dire les auteurs du programme, nous)
- Choisir de jouer en mode manuel, dans ce cas il devras par la suite choisir lui-même les missions attribuées aux véhicules.
- Jouer en simulation (mode automatique), dans ce mode toutes les requête sont générées par le programme : l'utilisateur se contente de regarde la simulation s'opérer.
- Jouer en simulation paramétrée (mode fichier), dans ce mode toutes les requête sont générées par le programme, mais elles ont été définies à l'avance dans un fichier texte, et selon le format qui suit :

[Départ] [Arrivée] [Moment d'arrivée] (chaque nombre est séparé par une tabulation)

Le moment d'arrivée correspond au moment t (en secondes, depuis le début de la simulation) où cette requête sera envoyée au contrôleur...

- Avoir accès aux options : une fois dans ce menu, il peut choisir de changer la rapidité des voitures.

Nous avons choisi de diviser l'utilisateur principal en deux autres : un utilisateur qui utiliserait le mode manuel, et un autre qui utiliserait le mode automatique ou le mode fichier (les actons possibles pour ces deux modes sont similaires). Ceci étant dû au fait que l'utilisateur en mode manuel pourra accéder à des fonctions interdites au mode automatique et au mode fichier (la création d'un passager).

En outre, une fois sur l'écran de "jeu" (où se déplacent les véhicules), l'utilisateur (et quel que soit son mode) pourra mettre la simulation en pause ou, si nécessaire, réinitialiser celle-ci.

1.2 Diagramme de séquence(annexe B)

Le diagramme de séquence permet de représenter les interactions entre les différentes classes selon un ordre chronologique. Ici, il représente l'envoi d'une requête de départ par l'utilisateur, et les actions qui s'en suivent, jusqu'à la fin du trajet du véhicule.

1.3 Diagramme de classes(annexe C)

Nous verrons en détail, une description de chaque classe. Le diagramme présenté en annexe constitue la partie calculatoire du programme : nous avons choisi d'ignorer la partie graphique...

Chapitre 2

Description des classes

2.1 Classe BoiteAuxLettre

Il s'agit de la classe recevant toute les requêtes, qu'elles soient envoyées depuis la partie graphique (c'est à dire des requêtes pour la création d'une nouvelle mission, et donc d'un nouveau passager), ou du modèle. Il y a 4 requêtes différentes, stockées chacune dans une liste différente :

Boîte aux lettres
<code>rFin : LinkedList<RFinTrajet></code> <code>rMap : LinkedList<RMap></code> <code>rDepart : LinkedList<RDepart></code> <code>rLib : LinkedList<RLib></code>
<code>getBoite() : void</code> <code>getFirst() : Requete</code> <code>addRequete(RDepart) : void</code> <code>addRequete(RMap) : void</code> <code>addRequete(RFin) : void</code> <code>getRFinTrajet() : RFinTrajet</code> <code>getDepart() : RDepart</code> <code>getRMap() : RMap</code> <code>getRLib() : RLib</code> <code>resetAll() : void</code>

- Une liste de requêtes de départ : cette liste contient toutes les demandes de trajet. Ces trajet peuvent être demandés soit par l'utilisateur, soit générés par le modèle.
- Une liste de requêtes de trajet : ce sont les requêtes qu'envoient chacun des véhicules ayant un passager pour demander au contrôleur la permission d'utiliser un chemin (la réponse n'est positive que si le chemin concerné est libre)...
- Une liste de requêtes de fin de trajet : ce sont les requête qu'envoient les véhicules lorsqu'ils ont mené leur mission à bien (et donc qu'ils sont arrivés à leur place d'arrivée). Elles permettent alors de libérer le véhicule.
- Enfin, une liste de requêtes de libération de ressources : ces requêtes sont envoyées par les véhicules lorsqu'il n'ont plus besoin d'une place (lorsqu'ils sont déjà passés de dessus). Le contrôleur peut alors considérer ces places comme disponibles.

2.2 La classes Contrôleur

Contrôleur
<code>general : HashMap</code> <code>boite : BoiteAuxLettres</code> <code>maFlotte : FlotteVehicules</code> <code>running : boolean</code>
<code>accepterMission() : boolean</code> <code>donnerMission() : void</code> <code>traiteRequete(RDepart) : void</code> <code>traiteRequete(RFinTrajet) : void</code> <code>traiteRequete(RMap) : void</code> <code>traiteRequete() : void</code> <code>debutSim(int) : void</code> <code>resetAll() : void</code> <code>pauseAll() : void</code> <code>run() : void</code>

Il s'agit de la classe principale : c'est elle qui s'occupe de gérer les déplacements des différents véhicules sur la carte. Le contrôleur a directement accès à la boîte aux lettres, et donc aux requêtes qu'elle contient.

C'est également lui qui possède la "carte" (ici, une HashMap) des différentes places du réseau routier : il est le seul à agir dessus (les rendre disponibles ou non). La méthode `traiteRequete()` se charge de traiter les requêtes selon leur priorité :

- Les requêtes de fin de trajet sont prioritaires sur toutes les autres : rendre un véhicule à nouveau disponible signifie le débarquement d'un passager, et donc la capacité d'en accepter de nouveaux.
- Les requêtes de libération de ressources arrivent en second. En effet, rendre une place disponible le plus vite possible est prioritaire pour pouvoir valider les trajets d'autres véhicules.
- Viennent ensuite les requêtes de départ, permettant aux passagers d'être pris en charge plus vite que les requêtes de trajet, bien plus nombreuses.

- Les requêtes de trajet sont les moins prioritaires, notamment en raison de le nombre : en effet, elles sont générées à intervalle régulier par les véhicules, jusqu'à ce qu'ils aient pu emprunter le chemin qu'ils désirent.

Par la suite, et en fonction des requêtes qu'il reçoit, le contrôleur a plusieurs tâches. Il peut libérer un véhicule en fin de trajet, en réserver un autre et lui attribuer un passager. Enfin, il valide (ou non) les chemins demandés par les véhicules, et se charge ensuite de réserver les ressources correspondantes. Il dispose également d'une fonction pour réinitialiser la totalité du système (à la demande de l'utilisateur), ainsi qu'une autre pour le mettre en pause, ou reprendre (même chose)...

2.3 Les classes Requêtes

Etant donné que le but de chaque requête a été explicité ci-dessus, et que la structure de celles-ci ne présente pas d'intérêt particulier, nous nous intéresserons à leurs caractéristiques générales.

Chaque type de requêtes a sa propre liste, contenue dans la boîte aux lettres. Une requête, une fois traitée, est immédiatement détruite. Enfin, à chaque liste de requêtes est attribuée une priorité, afin que le contrôleur traite les requêtes les plus importantes en premier.

2.4 La classe Cerveau

Cerveau
graphtop : boolean
start : boolean
maj : boolean
IDVehiculeGraphique : int
requestmap : RMap
corps : Vehicule
running : boolean
fin : boolean
run() : void

Il s'agit du Thread chargé de négocier le départ puis l'arrivée d'un véhicule. Il est ajouté au véhicule lorsque celui-ci reçoit un passager. C'est alors lui qui se charge, après avoir calculé le chemin à emprunter, de faire la demande de réservation auprès du contrôleur et, si celui-ci l'autorise, à faire partir le véhicule. Cette autorisation se fait par le biais d'un booléen, modifié par la flotte de véhicules, à la demande du contrôleur.

Par la suite, il se charge d'acheminer le véhicule jusqu'à sa destination, et signale ensuite au contrôleur la fin du trajet. Il est alors détruit, et le véhicule devient à nouveau disponible. Celui-ci recevra un nouveau "Cerveau" lors de l'embarquement du passager suivant. En parallèle, le cerveau se charge également de

synchroniser l'avancée du trajet avec la partie graphique : en effet, afin d'éviter des bugs d'affichage, le modèle doit attendre que le véhicule affiché atteigne la

place suivante avant d'envoyer les requêtes de libération et, à la fin du trajet, la requête de fin de trajet.

2.5 La Classe Flotte de Véhicules

Cette classe possède la totalité des véhicules existant dans le système. Elle est elle-même instanciée dans le contrôleur. Elle se charge, à la demande du contrôleur, de gérer les passagers : à leur apparition (requête de début de trajet), ils sont placés dans une file d'attente. Si la flotte possède des véhicules libres, elles les attribuera aux passagers en attente. Le contrôleur demandera recevra alors régulièrement des requêtes pour réserver des trajets de leur part.

La flotte vérifiera régulièrement (là-encore, à la demande du contrôleur) si un véhicule s'est libéré, et est ainsi disposé à accepter un nouveau passager. Enfin, c'est elle qui libère les véhicules à la fin de leur mission.

2.6 La Classe Véhicule

Chaque véhicule possède un identifiant unique donné par la classe "Flotte de Véhicules" et stocké dans un Integer "identifiant". Les véhicules sont instanciés sans trajet ni cerveau. Ceux-ci lui seront attribués plus tard, la de l'embarquement d'un passager. Le Cerveau se charge alors d'appeler les fonctions du véhicule pour calculer le trajet, envoyer des requêtes... A la fin de son trajet, seul le cerveau est détruit. Le véhicule redevient alors libre, dans l'attente d'un nouveau passager.

Troisième partie

Implémentation Java

Chapitre 1

Choix d'architecture

1.1 Le pattern MVC

Afin de permettre une meilleure représentation des différentes parties du programme, le choix a été fait d'utiliser le pattern "Modèle - Vue - Contrôleur". Il est constitué comme suit :

- La vue est la partie visible pour l'utilisateur : elle rassemble les différentes fenêtres du programme, les boutons et autres moyens pour l'utilisateur d'interagir avec celui-ci.
- Le contrôleur (afin d'éviter toute confusion avec le nom de la classe principale du programme, nous l'appellerons Vérificateur) se charge de vérifier que les données venant de la vue (et donc souvent de l'utilisateur), afin d'éviter l'envoi d'arguments non attendus aux fonctions internes. Il se charge par la suite de transmettre ces données au modèle.
- Le modèle est la partie calculatoire du programme. C'est lui qui, dans notre cas, se charge de la totalité de la simulation (voitures, contrôleur, trajets, etc...). Il peut alors envoyer des données à la vue pour qu'elle les affiche (faire bouger un véhicule, par exemple).

Ces trois parties sont mises en relation par le biais d'interfaces "Observer" et "Observable", qui permettent au modèle d'envoyer des informations à la vue sans même avoir conscience de son existence. Cette indépendance permet de modifier la vue à volonté sans avoir à modifier le modèle...

1.2 Des Threads pour régir les déplacements

Nous avons fait le choix d'utiliser des Threads pour permettre à chaque véhicule de bouger indépendamment. En effet, chaque véhicule actif (c'est à dire embarquant un passager) est dirigé par un Thread (la classe Cerveau). De cette manière, ils peuvent tous agir indépendamment. De plus, le contrôleur possède lui aussi un Thread, afin de pouvoir traiter de lui-même les requêtes qui arrivent dans la boîte aux lettres. Enfin, une classe située au niveau du vérificateur en comporte un, afin de pouvoir générer des requêtes à des moments donnés (mode fichier et mode automatique)...

Cette utilisation de nombreux Threads pourrait être problématique si un grand nombre de véhicules étaient présents simultanément sur la carte. Nous avons malgré tout persisté en se basant sur le fait que, telle que la carte et les contraintes sont faites, le nombre maximum de voitures sur la carte à un instant donné est de 3. Pas de risque de trop grande consommation des ressources, donc.

1.3 Un mot sur la génération automatique des requêtes

La génération automatique des requêtes se fait à l'aide d'un Thread, situé (selon le modèle MVC) entre la Vue et le Modèle, en parallèle du Vérificateur (qui exploite d'ailleurs une référence dessus). Ne pas le placer dans le modèle permet d'avoir une partie calculatoire souple, qui traitera toutes les requêtes, quelle que soit leur provenance... Cette classe se divise en deux fonctionnalités.

- Le mode automatique correspond à une génération aléatoire des requêtes. Elles sont envoyées toutes les deux secondes dans la boîte aux lettres et ce, tant que la simulation continue.
- Le mode fichier, qui exploite un fichier texte contenant des requêtes écrites selon un format particulier (voir II.1.1). Ce mode se sert d'un compteur que le Thread incrémente chaque seconde pour savoir quand envoyer la requête.

Chapitre 2

Fonctions importantes

Ce chapitre est destiné à mettre plus en lumière certaines fonctions que nous avons jugé assez importantes pour nous y attarder. Nous en donnerons le code, puis expliciterons sa fonction et son fonctionnement.

2.1 Classe Controleur

La fonction principale de cette classe et la méthode `traiteRequete` : en effet il s'agit de la méthode qui doit se charger de vérifier si il y a des requêtes dans la boîte aux lettres et, si c'est le cas, effectuer un traitement différent selon la nature de la requête. De plus, les requêtes n'étant pas toutes envoyées en même temps, il est possible que le contrôleur aie à attendre l'arrivée d'une requête...

Nous pouvons alors nous attarder sur la méthode `traiteRequete(RMap)`. Cette méthode doit, dans un premier temps vérifier que toutes les places demandées sont libres, puis elle doit les réserver dans la hashmap générale. Elle se charge également de signaler à la voiture qu'elle peut (ou non) partir...

```
private void traiteRequete(RMap r) {
    ArrayList<Boolean> m = r.getRequest_map();
    // compteur pour savoir ou on en est dans les boucles
    int i = 0;
    // il y aura au maximum 5 routes de réserver+ l'ietreateur sur ce
    // tableau
    int[] tab = new int[5];
    int iterateurTab = 0;
    // entier pour savoir combien de true il y a dans la requestMapr e
    // general
    int trueInRequete = 0;
    // entier pour connaitre le nombre de true dans la liste general;
    int trueInGeneral = 0;
    for (Boolean b : m) {
        if (b == true) {
            trueInRequete++;
            // si la place est libre
            if (general.get(i)) {
                trueInGeneral++;
                tab[iterateurTab] = i;
                iterateurTab++;
            }
        }
        i++;
    }
    // si le nombre true dans la request map et dans la hasmap general sont
    // egaux alors les ressource sont libre et on peut les réserver+passer
    // le boolean de la voiture en true
    if(trueInRequete==trueInGeneral){
        //fonction permettant de de metre les ressource en
        indisponibilite
        for(int j=0;j<i;j++){
```

```

        general.put(tab[i], false);
    }
    maFlotte.lancerVehicule(r.getIdentifiant(), true);
}
}

```

2.2 Classe Vehicule

Une des méthodes les plus importantes de cette classe est la méthode `findPath` qui, à partir d'un passager et des informations qu'il contient, crée une liste de places qui constitueront le trajet à suivre par le véhicule. Pour cela, il faut tout d'abord faire la différence entre deux cas possibles :

- La place d'arrivée est située en face de la place de départ, auquel cas le trajet passe obligatoirement par le centre.
- La place d'arrivée est située autre part. En ce cas, il faut alors calculer le chemin le plus rapide pour s'y rendre.

```

public void findPath(Passager p) {
    // On vide la liste
    trajet.clear();
    // On stocke les places de depart et de fin (on parle ici des places R,
    // pas des places d'entree ou de sortie)
    int dep = p.getDebut() + 10;
    int fin = p.getFin() + 10;
    int a, b;
    int count = -1;
    // On cree deux tableaux pour parcourir la carte dans les deux sens
    int[] l1 = new int[3];
    int[] l2 = new int[3];
    // On commence par la place de depart (son identifiant est egal a la
    // place R - 10)
    trajet.add(dep - 10);
    // Si la place d'arrivee est en face, on passe par le centre
    if (Math.abs(dep - fin) == 3) {
        trajet.add(dep);
        trajet.add(30);
    } else {
        // On commence a la place R de depart
        a = dep;
        b = dep;
        // Ensuite, on parcourt la carte dans les deux sens pour
        // determiner
        // le chemin le plus rapide
        while (a != fin && b != fin) {
            count++;
            // On stocke le trajet
            l1[count] = a;
            l2[count] = b;
            // On se charge de faire un cycle (le 6 est suivi du 1)
            // S'agissant des places R, leur identifiant est decale
            // de 10
            if (a == 16) {
                a = 10;
            }
            if (b == 11) {
                b = 17;
            }
            // On se decale dans les deux sens
            a++;
            b--;
        }
    }
}

```

```

    } // Si le trajet b est le plus court
    if (b == fin) {
        // On le stocke dans le premier tableau
        l1 = l2;
    } // Enfin, on stocke le trajet dans la liste
    for (int i = 0; i < l1.length; i++) {
        if (l1[i] != 0) {
            trajet.add(l1[i]);
        }
    }
}
// On termine par la place R finale, puis la place d'arrivee (R + 10)
trajet.add(fin);
trajet.add(fin + 10);
}

```

Grâce à cette liste de points, la classe va pouvoir générer une RMap qui sera par la suite envoyée à la boîte aux lettres, formatée comme indiqué dans la partie Requête.

2.3 Classe Flotte de Véhicules

```

public synchronized void checkAttente() {
    boolean allUse = true;
    for (Vehicule v : vehicules) {
        if (v.isDispo() == true) {
            allUse = false;
        }
    }
    if (listeAttente.size() == 0 || allUse == true) {
        try {
            wait(500);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    } else {
        int j = 0;
        while (j < listeAttente.size()) {
            Passager p = listeAttente.get(j);
            boolean dispo = false;
            // verification qu'il reste des vehicule libre
            for (Vehicule c : vehicules) {
                if (c.isDispo()) {
                    dispo = true;
                }
            }
            if (dispo == true) {
                int i = 0;
                //on va chercher le vehicule dispo
                while (i < vehicules.size()-1 &&
                    !vehicules.get(i).isDispo()) {
                    i++;
                }
                p.setEmbarque(true);
                vehicules.get(i).setDispo(false);
                vehicules.get(i).setPassager(p);
                vehicules.get(i).setCerveau(new
                    Cerveau(vehicules.get(i)));
                vehicules.get(i).getCerveau().start();
                listeAttente.remove(p);
                j--;
            }
        }
    }
}

```

```

        }
        j++;
    }
}

```

C'est cette fonction qui permet à la Flotte de Véhicules de vérifier s'il n'est pas possible d'assigner un véhicule (qui se serait libéré entre temps) à un passager en attente. Pour cela, on commence par vérifier si au moins un véhicule est libre. Si c'est le cas, on cherche à assigner un véhicule au plus de passagers possible. Chaque véhicule nouvellement réservé se voit attribuer un cerveau (instancié pour l'occasion), cerveau dont le Thread est alors lancé.

2.4 Classe Boite au lettres

Cette classe possède 4 listes, chacune de ses liste a la même logique : dans un premier temps le Contrôleur doit pouvoir accéder à la taille de cette liste afin de savoir si elle est vide ou non.

Ensuite, lorsque le Contrôleur veut une requête il faut lui envoyer puis supprimer cette requête de la liste. Voici un exemple d'implémentation pour la méthode `getDepart` :

```

public RDepart getDepart() {
    RDepart r = new RDepart();
    // on recupere le dernier element
    r.clone(rDepart.get(rDepart.size() - 1));
    rDepart.remove(getSizeRDepart() - 1);
    return r;
}

```

2.5 Classe Cerveau

La classe Cerveau ne contient qu'une seule fonction à proprement parler, et celle-ci est de grande importance : c'est elle qui fait bouger le véhicule, lui fait envoyer des requêtes, et le synchronise avec la partie graphique. Pour cela, elle se divise en deux phases : la première, où le Cerveau, après avoir calculé le trajet, demande (par le biais du véhicule) au Contrôleur l'autorisation de partir. La seconde, où l'autorisation a été donnée.

```

public void run() {
    int i=1;
    //On calcule le trajet
    corps.findPath(corps.passager);
    requestmap = corps.trajetToMap(corps.trajet);
    //Tant qu'on a pas eu d'autorisation
    while (start != true) {
        //Si on a eu une reponse (negative)
        if (maj) {
            corps.sendRMap(requestmap);
            maj = false;
        } else {
            try {
                sleep(300);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
    //On se positionne au depart et on recupere l'ID du vehicule graphique
    attribue
    idVehiculeGraphique = corps.notifyDebutMission(corps.trajet.get(0));
    //Et on va de points en points
    while (i < corps.trajet.size()) {
        //Si la voiture graphique a termine de bouger
        if(graphtop){
            graphtop=false;

```



```

        corps.sendRLib(corps.trajet.get(i-1));
        //Nouvelles coordonnees
        corps.notifyCoords(idVehiculeGraphique,
            corps.trajet.get(i));
        i++;
    } else {
        try {
            sleep(100);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
} //Le trajet est termine
while (fin == false) {
    //Si la voiture graphique a termine de bouger
    if(graphtop){
        graphtop=false;
        corps.notifyCoords(idVehiculeGraphique, 99);
        //L'identifiant 99 correspond a la fin du trajet
        corps.sendRFinTrajet(corps.trajet.get(i-1));
        fin = true;
    } else {
        try {
            sleep(100);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
}
}

```

Son fonctionnement nécessite quelques explications.

Premièrement, les booléens :

- **maj** est mis à *True* par le cotrôleur, lorsqu'il a traité la requête du véhicule. Cela indique au véhicule qu'il a reçu une réponse (il peut alors renvoyer une requête), et lui évite donc de surcharger la boîte aux lettres de requêtes inutiles.
- **start** est mis à *True* lorsque le contrôleur autorise le véhicules à partir.
- **graphtop** est mis à *True* par le véhicule "graphique" (celui qui est affiché) lorsqu'il a atteint la place suivante. Cela permet alors au véhicule de se rendre à la place d'après, puis d'attendre que la partie graphique aie fini de l'afficher, etc...
- **fin** est mis à *True* par le cerveau lui-même, lorsqu'il arrive à la place d'arrivée. il attends donc une dernière fois le véhicule graphique, puis entame la procédure de libération.

Ensuite, la fonction *notifyCoords(int, int)* permet au cerveau (en passant par le véhicule) d'envoyer l'identifiant de la place suivante au véhicule graphique. Celui-ci est fait de telle sorte qu'il cherchera toujours à atteindre les cordonnées qu'on lui donne. Il rejoindra donc la place donnée par le cerveau (et mettra graphtop à *True* lorsqu'il y sera), après avoir tranformé l'identifiant en coordonnées (x, y).

Chapitre 3

Difficultés rencontrées

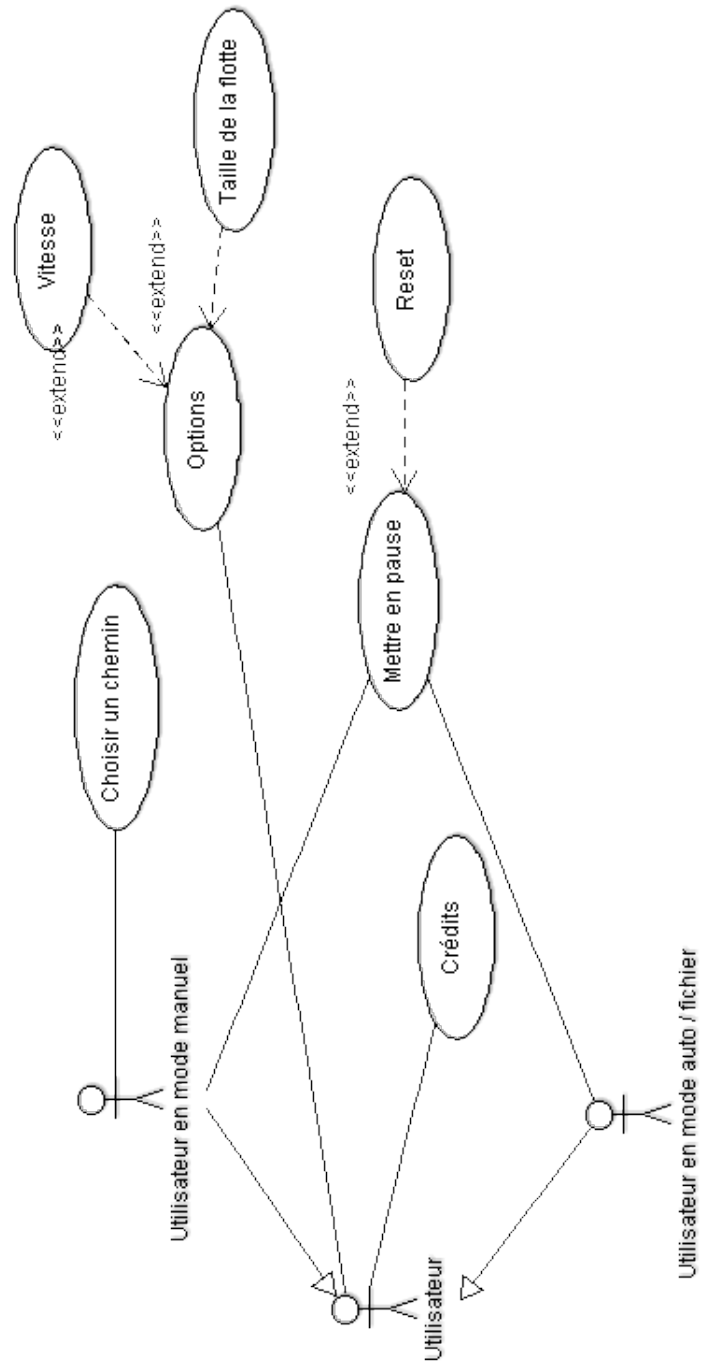
Au cours du développement de ce projet, nous avons dû faire face à plusieurs difficultés :

- Ce projet était pour nous une occasion d'utiliser l'outil Git pour la première fois. Cela a donc engendré certains problèmes de fichiers perdus, ou encore des *merge* ayant corrompu des fichiers UML. Fort heureusement, le versionnage du code nous a permis de retrouver les fichiers fonctionnels assez facilement...
- L'implémentation des Threads, que nous avions commencé avant d'avoir suivi les TDs les concernant, a été difficile. Ce n'est qu'après avoir pu nous familiariser avec la synchronisation (mot clé *synchronized*) que de nombreux bugs ont pu être corrigés.
- D'autres bugs sont apparus, dûs au fait que nous n'utilisons pas le même système d'exploitation : l'un est sous Linux, l'autre sous Windows. Et malgré la portabilité du langage Java, certaines fonctions ne se comportent pas exactement de la même façon sur les deux plateformes, occasionnant quelques bugs graphiques...

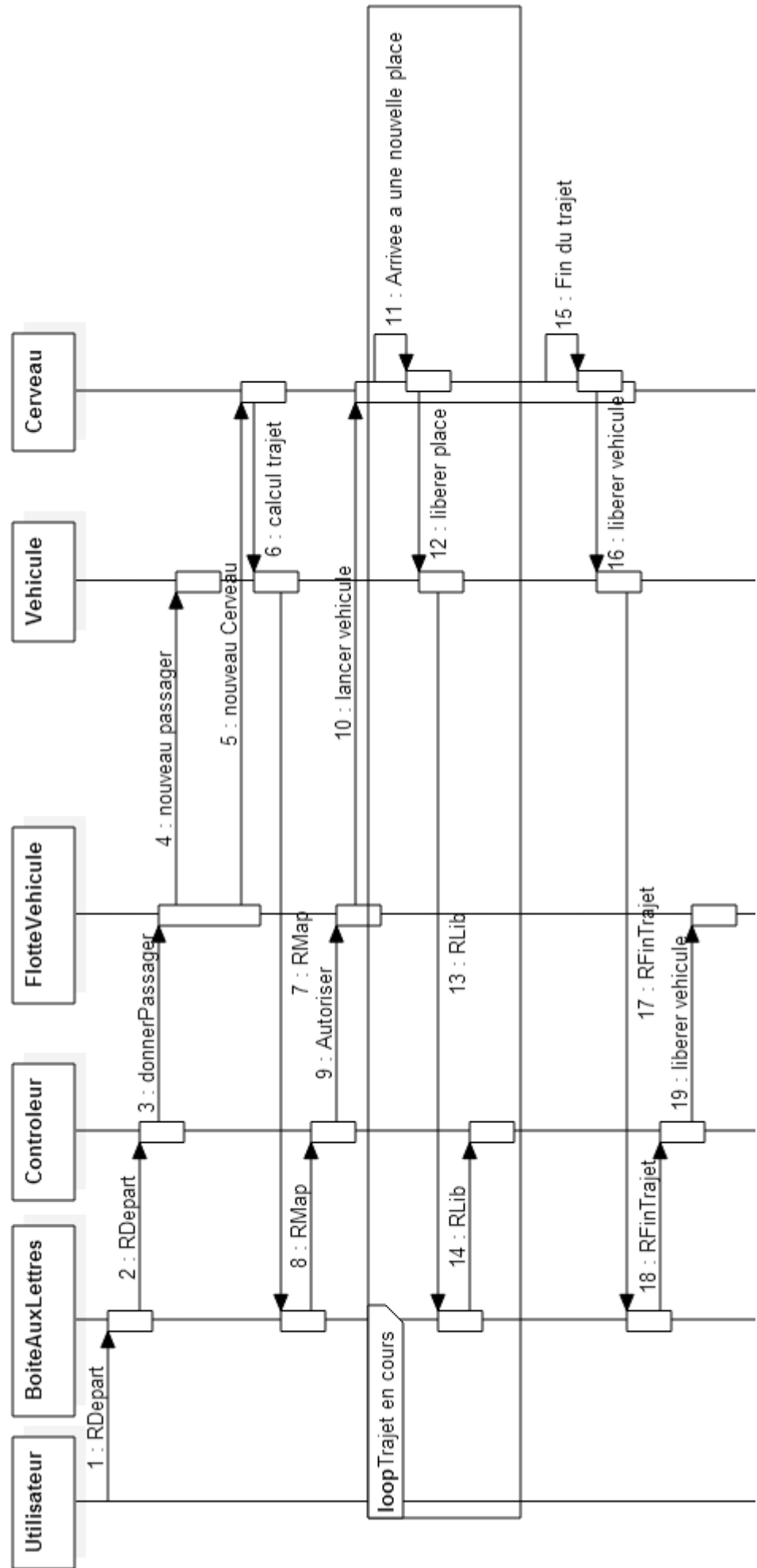
Conclusion

Grâce à ce projet, vous avons pu faire une modélisation de notre projet à l'aide du langage UML. Ceci nous a permis d'en découvrir l'utilité et la praticité, ainsi que d'engager un travail de réflexion sur notre projet en amont... Nous avons alors pu économiser du temps sur un éventuel défaut d'architecture que l'on aurait pu commettre sans l'aide de ce langage. De plus, ce projet nous a fait découvrir de nouvelles fonctionnalités de Java, à savoir les Threads et les exception. Enfin, nous avons implémenté dans ce projet une Javadoc, afin que la communication entre les différents développeurs (notamment les futurs) soit plus simple. L'outil Git nous a également été d'une grande aide, pour stocker notre projet, gérer les différents bugs et simplifier la gestion d'un code à deux (versionnage, travail simultané sur un fichier, etc...).

Annexe A



Annexe B



Annexe C

