COMPTE RENDU DU TP LONG

SERTR 2009

LOUVET Anthony - Chef de Projet
D'INCAU Benoît : Responsable modèle
HERRMANN Thomas
LALOUETTE Jerome
MOCQUAX Clémence - Responsable Statistiques
COSTE Pauline
SECK Omar - Responsable Interface Graphique
ROUZAUD Cédric
MADRANGE Nicolas
Professeur-client : M. BOUTHIER et M. DERNIAME

REMERCIEMENTS

Jean Claude DERNIAME, et Christophe BOUTHIER, sont les premières personnes à remercier. Les connaissances qu'ils nous ont enseignées ont été capitales pour la réussite du projet.

Nous n'oublierons pas de si tôt les conseils avisés et les anecdotes croustillantes de M. DERNIAME, ni les astuces et la bonne humeur inconditionnelle de M. BOUTHIER qui s'est énormément investi pour nous.

Cet engouement s'est transmis à tous les élèves, et chacun d'entre nous a su se rendre utile pour l'avancée du projet. Par notre travail personnel nous avons tous contribué à la réussite finale du projet.

TABLE DES MATIERES

I. INTRODUCTION DU SUJET

Les premières séances ont été l'occasion pour nous de découvrir le sujet, et d'en discuter avec les clients (M. DERNIAME et M. BOUTHIER) dans le but de le préciser au maximum. La répartition des tâches et les méthodes de travail ont aussi été abordées pendant ces premières heures.

Nous rappelons dans ce qui suit les points importants qui sont ressortis du cahier des charges.

1. Présentation du cahier des charges

Voici les critères les plus pertinents du cahier des charges :

Le projet possède un objectif général qui est la mise en œuvre d'un outil d'aide à la décision pour le responsable du transport dans une mairie de grande ville.

Objectifs pédagogiques du projet :

- Appropriation de technologies de développement.
- Approche du travail en équipe de taille conséquente et conduite de projet logiciel.

Critères d'évaluation:

- Respect du cahier des charges
- Bon fonctionnement de la démonstration finale à savoir validation des scenaris approuvés par le client.
- Agrément et sophistication de l'interface graphique réalisée.
- Communication entre les sous-groupes de travail
- "Qualité de la solution retenue".

Souhaits du client :

La circulation au centre ville est souvent difficile et aux heures de pointe, des blocages arrivent souvent le long d'un axe AB constitué souvent de deux fois deux voies.
Le client envisage de mettre en place un tramway en site propre dans les parties où c'est possible sur les trajets entre A et B.

Pour cela :

- Il souhaite connaître une évaluation de l'impact prévisible de ce projet sur un certain nombre de données concernant la circulation en centre ville. Pour cela, il veut avoir la possibilité d'afficher plusieurs valeurs :

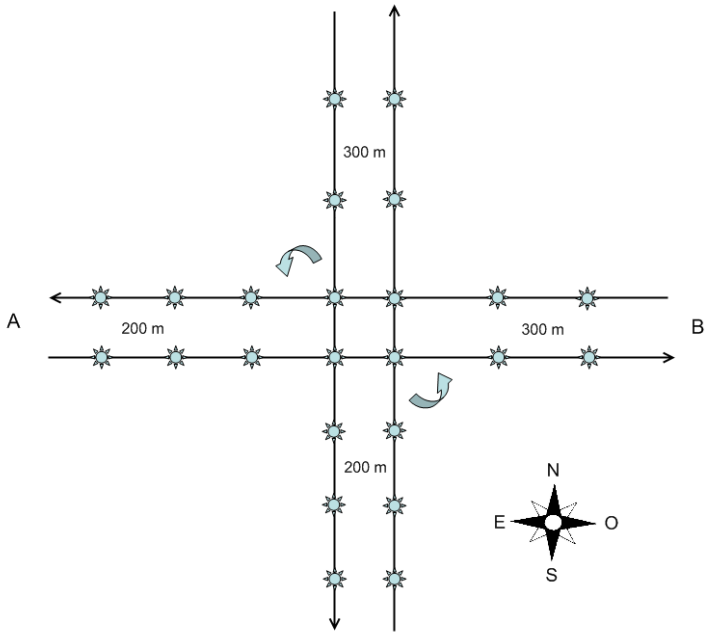
  - le temps moyen de trajet de A à B pour chaque type de véhicules (bus, tram, voiture) sur la durée du temps simulé.
  - le nombre de voiture arrêtées à chaque feu (en moyenne sur la durée du temps simulé) ainsi que le temps moyen d'attente sur la durée du temps simulé sur le trajet (de A à B) pour chaque type de véhicules (bus, tram, voiture).
  - Il souhaite également afficher une courbe présentant le temps de blocage de chaque type de véhicules en fonction de la charge, sur la durée du temps simulé.

➢ L'interface graphique devra respecter ces conditions :

- afficher en temps réel l'état des feux à savoir vert ou rouge.
- afficher en temps réel le déplacement des véhicules (voiture, tram et bus)
- Respecter à l'échelle les vitesses et la position des voitures
- Des options « pause » et « et pas à pas » seront disponibles.
- Possibilité de zoomer sur la fenêtre graphique initiale pour obtenir une meilleure définition graphique autrement les véhicules ne seront pas visibles.
- Pour différencier les tram, des bus et des voitures, on choisira pour longueur d'un tram une longueur de quelques voitures et un bus une longueur situé entre les deux..
- afficher un menu permettant à l'utilisateur de choisir les options qu'il veut pour une simulation (temps de simulation, temps réel, quantité de trafic, choix du scenario…)
- renvoyer à l'écran en temps réel les résultats de la simulation (temps d'attente moyen des voitures, temps de parcours moyen et les autres informations utiles).
- Un bouton permettant n'importe quand de lancer sur les routes un véhicule prioritaire.

➢ Il souhaite une interface lui permettant d'entrer les paramètres nécessaires pour une simulation et d'afficher les résultats. Les données modifiables par l'utilisateur sont :

- Les charges globales d'utilisateurs sur les axes est-ouest et nord-sud qui seront entrées en paramètres au début de la simulation.
- La charge d'utilisateurs sur les différents moyens de transport en pourcentage.
- Le nombre de bus, de rames et de wagons par rame pendant la simulation.
- (Possibilité d'introduire un véhicule prioritaire à tout moment).
- Le mode d'exécution de la simulation : il sera possible au client de simuler le modèle de manière interactive, soit par des boutons Démarrer/Pause/Arrêter, soit en mode pas à pas. L'utilisateur pourra aussi simuler de manière automatique en entrant directement la durée du temps simulé, avec la possibilité d'y insérer des points d'arrêt. La simulation pourra se faire avec ou sans modèle graphique et en choisissant la vitesse d'observation.
- L'utilisateur aura la possibilité de sauvegarder les résultats ou charger ceux d'une simulation antérieure, afin de pouvoir fusionner plusieurs résultats.



2. L'organisation du travail

La première étape aura été de répartir les tâches, et de s'assurer que chaque groupe ait bien cerné le travail qu'il devra accomplir. Quatre groupes on donc été définis : la partie modélisation, le calcul statistique, l'interface graphique et le réseau.

Au commencement, nous avons beaucoup réfléchi ensemble à la façon dont le modèle de la ville allait être réalisé, et comment il allait se comporter.Nous avons pu ensuite nous attacher plus particulièrement à notre groupe de travail.

Par la suite, chaque groupe a été suffisamment autonome pour poursuivre ses objectifs internes. Mais aucun groupe n'était indépendant des trois autres, et chaque séance a été l'occasion de se concerter à propos de problèmes communs ou de choix qui se devaient d'être fait en accord avec plusieurs personnes. On peut citer par exemple les API du réseau, ou bien la numérotation des tronçons de la ville.

Comme il était prévu dans le cahier des charges, nous avons utilisé un répertoire de travail commun (sous google code), ce qui nous a énormément facilité la tâche lorsque nous avons travaillé séparément.

II. LA PARTIE RESEAU

1. Introduction

Un des souhaits du client est d'avoir une application répartie sur trois ordinateurs. Il est donc nécessaire d'avoir un moyen de communication entre ces trois machines. Nous utiliserons pour cela le RMI (Remote Method Invoke), qui permet de définir une API de communication.

2. Le RMI

Le RMI s'utilise en définissant une interface sur un objet. Toute méthode définie dans cette interface doit être implémentée par l'objet, et est susceptible d'être appelée par un autre objet distant. Les méthodes de cette interface définissent en fait le comportement de l'objet, vu par le réseau.

Avant de pouvoir être visible en réseau, un objet doit s'enregistrer auprès du « rmiregistry », qui est un annuaire répertoriant les objets disponibles sur le réseau. De manière symétrique, lorsque l'on veut utiliser un objet disponible sur le réseau, on va le chercher par le « rmiregistry ».

3. Les interfaces de chaque objet

Pour la partie modélisation, c'est un objet de type modèle qui est transmis par le RMI. Tous les évènements qu'il génère doivent pouvoir être récupérés par les statistiques et la vue. De plus, la simulation est tributaire des ordres donnés par l'utilisateur par le biais de la vue graphique. Le modèle doit donc implémenter des méthodes permettant d'agir sur la simulation, comme le démarrage, la mise en pause, et l'arrêt de celle-ci.

La méthode « initParametres(...) » par exemple permet d'envoyer au modèle tous les paramètres dont il a besoin pour s'initialiser (sans pour autant démarrer la simulation). Ces valeurs sont spécifiées par l'utilisateur, et sont donc envoyées au modèle en paramètres de la méthode.

Pour la partie statistique, c'est un objet de type stats qui est transmis par le RMI. Les statistiques doivent pouvoir observer les informations générées par le modèle, et envoyer le résultat de leurs calculs à l'interface graphique. Cela se fait par un observateur – observable que nous détaillerons plus tard. Une méthode à été utilisée pour envoyer les statiques finales, lorsque la simulation est terminée.

Pour l'interface graphique, aucun objet ne s'enregistre, puisqu'on se place dans ce cas, uniquement en consommateur des services proposés par le modèle et les statistiques.

4. L'observateur – observable

En plus des APIs classiques, on utilise l'observateur pour envoyer ou recevoir des informations. Un objet déclaré comme observable peut notifier ses observateurs (dont il a connaissance) d'une nouvelle information. Un objet déclaré observateur implémente une méthode update, qui récupère l'information en paramètre, et se charge de la traiter.

Schématiquement, on obtient le fonctionnement suivant :

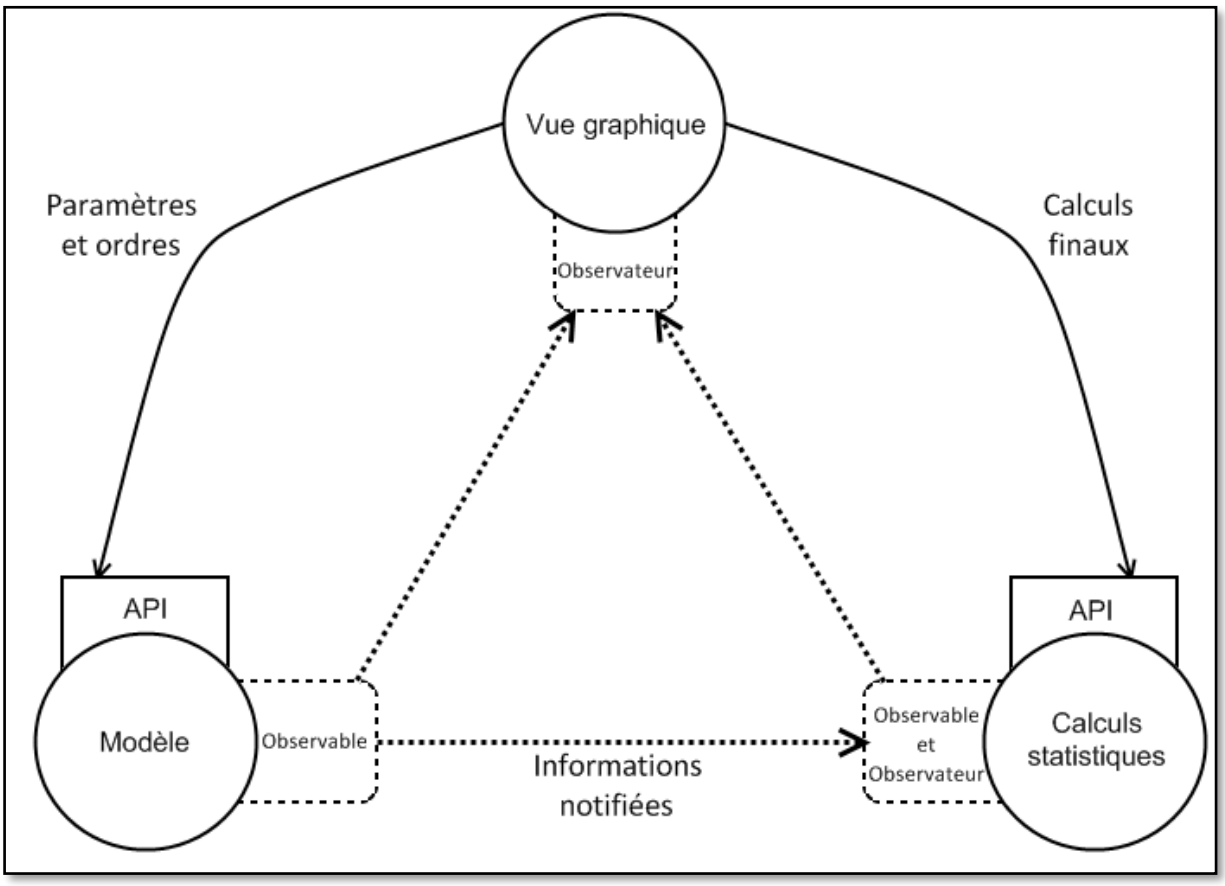


Figure 1: Schéma de fonctionnement

III. EXPLICATION DU FONCTIONNEMENT ET CHOIX DU MODELE

Comme nous l'avons vu auparavant, le modèle se situe au centre de l'architecture. Il est chargé d'instancier la ville, de la simuler selon les choix de l'utilisateur et d'envoyer, à l'interface graphique et au module de statistique, les différents événements ayant lieu lors de la simulation. Ceci se fait grâce au schéma de communication vu précédemment (cf. figure 1 partie I.4.).

Pour résumer, le modèle ne fait que proposer ces informations aux différents modules et récupère les ordres passés par l'IHM. Le fonctionnement du modèle doit être totalement transparent pour les autres modules. Pour cela nous avons créé une classe *Modèle* qui s'occupera de l'interface entre l'IHM et l'ensemble des autres classes. Ainsi la vue aura accès à des méthodes de cette classe *Modèle* afin d'agir sur l'ensemble des paramètres de la simulation (vitesse de simulation, charge…).

Selon le schéma de communication, le module de statistique et le module graphique doivent pouvoir récupérer les informations envoyées par le modèle. Pour cela on utilise le *design patternObservable/Observers*. Le module statistique et le module interface graphique sont des vues enregistrées dans un tableau de vues de la classe *Modèle*. Ces vues sont réveillées lorsque le modèle possède des informations à envoyer. Pour instancier ceci, on fait dériver la classe *Modèle* de la classe *JavaUtilObservable*.

Pour simuler le modèle nous avons choisi d'utiliser des événements. Chaque événement représentera une action qui modifiera l'état du modèle, par exemple faire avancer une voiture ou faire passer un feu du rouge au vert. Nous avons pris la décision d'utiliser une programmation asynchrone plutôt que synchrone : nous ne pouvons pas prévoir à l'avance quels événements vont arriver et à quel moment, c'est pourquoi nous utilisons un ordonnanceur seulement programmable en asynchrone.

Lors du fonctionnement du modèle, on doit être capable de gérer en parallèle la communication avec les autres modules (statistique et interface) tout en simulant notre modèle. Pour cela on utilise deux threads qui fonctionneront "simultanément". Un s'occupera des commandes venant de l'extérieur (*Main*), le second créera les événements et les consommera (*ThreadOrdonnanceur*). Ce dernier point apporte un avantage majeur, il garantit l'impossibilité de créer un événement à une date antérieure de celle courante dans l'Ordonnanceur. En effet, vous ne pouvez pas dans ce cas créer une voiture à 8h01 alors qu'il est 8h02. Notons que cette notion est liée au fait que la création et l'exécution sont gérées dans le même thread.

Vous trouverez en annexes le diagramme des classes réalisées en UML de l'ensemble du modèle.

1. La classe de Modèle

Comme nous l'avons expliqué, la classe *Modèle* réalise l'interface entre l'IHM et le modèle : elle permet ainsi de créer le modèle en tant que tel (*Ordonnanceur, Ville, Parametres…*) quand l'utilisateur le souhaite. Elle contient les actions appelées par l'IHM par l'intermédiaire du RMI telles que *demarrer()*, *arret()*…Ces actions agissent directement sur la simulation donc sur le thread de simulation *ThreadOrdonnanceur*. La classe *Parametres* est en fait l'objet créé lors du passage des paramètres de simulation (temps de simulation, nombre d'utilisateur…) de l'IHM vers le modèle par l'intermédiaire de la méthode *initParametre()*.

On implémente la classe *Modèle* sous forme de singleton, ainsi il est en accès global dans tout le modèle. Ceci est nécessaire afin de pouvoir réveiller les vues lorsqu'un événement a lieu (*notify()*). Ces réveils peuvent être utilisés n'importe où dans le modèle et c'est la classe *Modèle* qui est observable.

2. Modélisation de la ville

On avait le choix de modéliser la ville par des tronçons de cases ou des files d'attente : nous avons choisi le premier cas, car une file d'attente ne montre pas la position d'une voiture sur un tronçon, elle montre juste le nombre de voitures attendant à un feu. Nous avons donc créé une classe *Troncon* qui est composée d'un ensemble de cases.

Une ville est en fait un tableau de tronçons. Chaque tronçon pointe sur les deux suivants sur lesquels les voitures peuvent avancer et sur les deux tronçons d'où peuvent être issus les véhicules. Donc un tronçon pointe vers 4 tronçons. Afin d'identifier un tronçon ou une case, chacun possède son propre numéro.

Afin de gérer les modifications d'état des feux, nous avons créé une classe *CaseFeu* qui hérite de la classe *Case*. Ce type de case se situe à la fin d'un tronçon. La gestion des feux doit être synchronisée pour chaque case feu d'un même carrefour. C'est pourquoi nous avons créé une classe *Carrefour* qui connaît l'ensemble des feux d'un même carrefour et qui poste un événement *evCarrefour*. Ce dernier créera au moment souhaité des *evFeuRouge* ou *evFeuVert* afin de modifier l'état des feux.

Nous avons instauré des règles de circulation dans notre ville afin de simplifier le modèle. Voici la circulation des voitures à un carrefour :

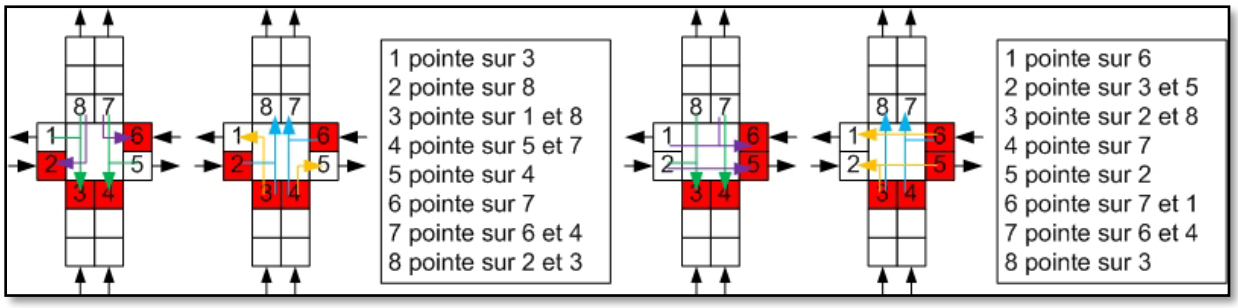


Figure 2: Règles de circulation aux carrefours.

On peut voir la façon dont les tronçons et les cases s'organisent pour constituer la ville sur la carte.

Vous trouverez ci-dessous le diagramme des classes réalisées en UML modélisant la ville :

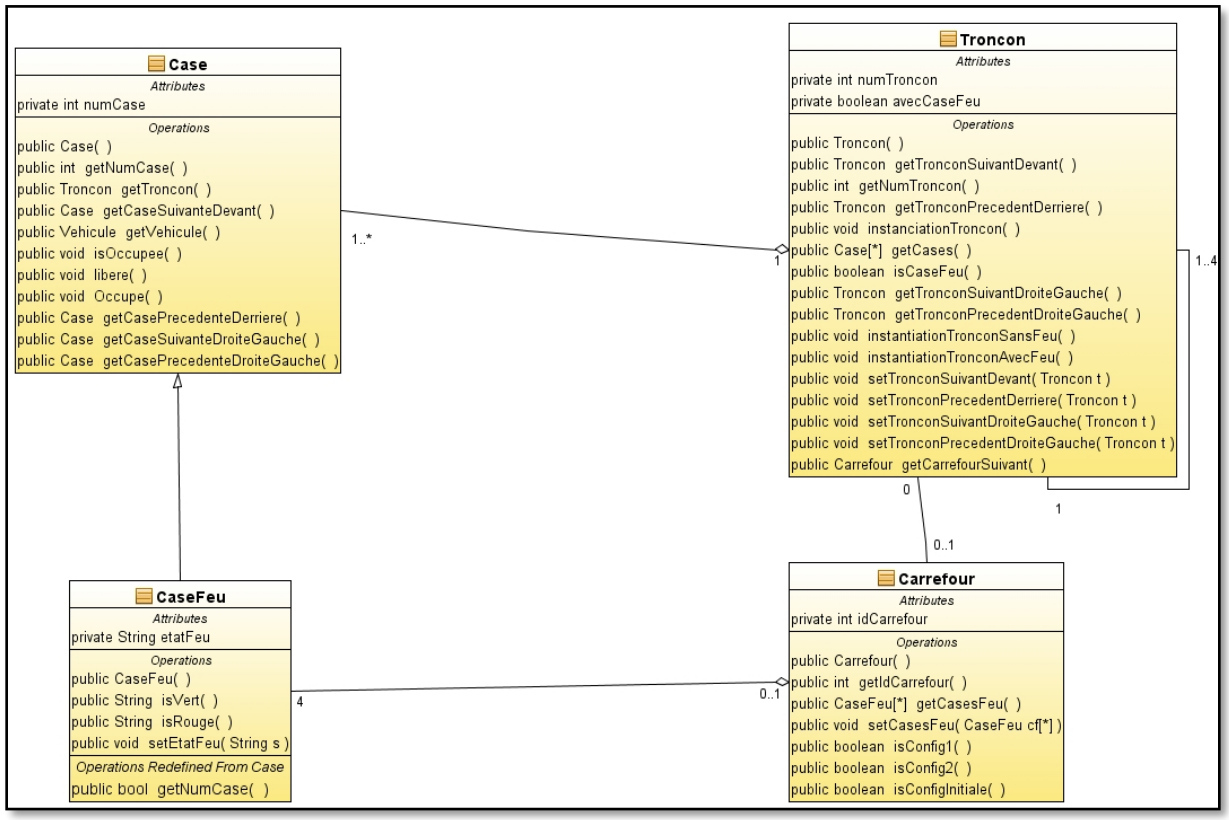


Figure 3 : Modélisation de la ville

3. Thread Ordonnanceur

Ce thread est créé lorsque l'on démarre le modèle par l'intermédiaire de l'interface graphique. Son rôle est d'appeler l'Ordonnanceur à des moments précis pour consommer tour à tour les évènements.

Il s'agit donc d'une boucle infinie active tant que l'utilisateur n'a pas décidé de mettre fin à la simulation ou que celle-ci n'est pas arrivée à son terme (plus aucun évènement à consommer, nombre d'utilisateurs devant emprunter le trajet nul, durée de simulation atteinte).

Ce thread a plusieurs manières de fonctionner. A l'instar d'un debugger, il peut soi tourner en continu jusqu'à ce que l'utilisateur décide de le mettre en pause, auquel cas le thread se met en attente d'un ordre de redémarrage, soit s'exécuter en mode pas à pas.

Dans ce dernier cas, il est chargé de consommer tous les premiers évènements de l'Ordonnanceur possédant une date d'exécution identique.

Le principe du fonctionnement continu est similaire à celui du pas à pas. La consommation de chaque groupe d'évènements possédant la même date d'exécution est séparée d'une durée définie par l'utilisateur (*vitesse de simulation*).A tout moment, l'utilisateur peut passer du mode pas à pas au fonctionnement normal. Le thread reçoit l'ensemble de ses ordres de la part de la classe *Modèle*.

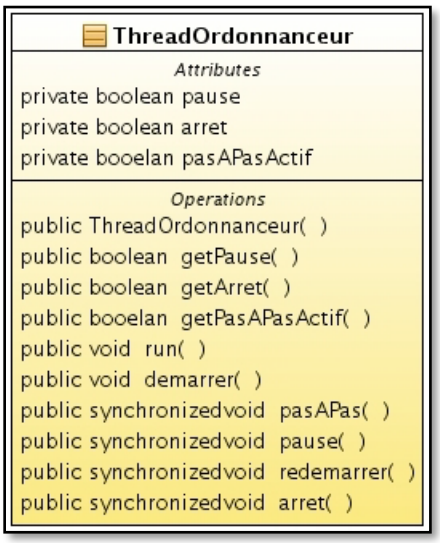


Figure 4 : Classe ThreadOrdonnanceur

4. Gestion des événements et Ordonnanceur

Les événements sont générés par les différents objets sur lesquels ils doivent faire effet *(voiture, bus, tram, feu, carrefour).* Ce sont seulement ces objets qui savent à quel moment ils peuvent réaliser une action ou une autre. Ce sont donc eux qui créent les événements et qui les postent dans l'Ordonnanceur.

Fondamentalement, un événement contient la date à laquelle il doit être exécuté. Néanmoins, selon les types d'évènement, l'exécution diffère. Deux solutions ont été envisagées : soit ajouter un attribut de différentiation dans la classe *Evenement*, soit utiliser l'héritage. La première solution nécessite de tester l'attribut à chaque fois que l'on traite un événement, ce qui peut être une grande source d'erreur.

Nous avons donc décidé de décliner chaque type d'évènement selon un objet, possédant une *date d'exécution* et un autre paramètre (*la case de destination* pour un événement faisant avancer une voiture, par exemple), ainsi qu'une méthode d'exécution propre. Il y a dans notre modèle trois familles d'évènements : créer un véhicule et faire avancer un véhicule, spécialisées ensuite pour les trois types de véhicules, et changer l'état d'un feu.

Tous les évènements sont stockés chronologiquement dans l'Ordonnanceur. Celui-ci a pour but de gérer la liste des évènements (sous forme d'un *Vector*), c'est-à-dire d'ajouter des événements à la liste, de les récupérer et les traiter (en appelant la méthode d'exécution adéquate). C'est également lui qui met à jour *l'horloge* du modèle ; elle est actualisée à la date d'un événement lors de sa consommation.

L'Ordonnanceur doit être accessible par tous les objets du modèle afin qu'ils puissent ajouter un événement quand c'est nécessaire. Pour cela trois méthodes sont possibles :

- Utilisation du design pattern **Singleton**
- Déclaration de l'*Ordonnanceur* en tant que classe *statique*
- Passage de l'*Ordonnanceur* comme paramètre de chaque objet qui l'utilise

On utilisera ici, la première solution qui a comme principal avantage de rendre l'objet global et donc accessible par tous. La seconde solution ne peut être utilisée puisqu'on modifie l'état du modèle dans l'objet, ce qui est incompatible avec cette méthode. La troisième quant à elle n'est pas propre, car cela reviendrait à avoir l'*Ordonnanceur* comme attribut de tous les objets du modèle. De plus, en définissant l'*Ordonnanceur* comme un *Singleton*, nous sommes assurés qu'il ne sera créé qu'une et une seule fois.

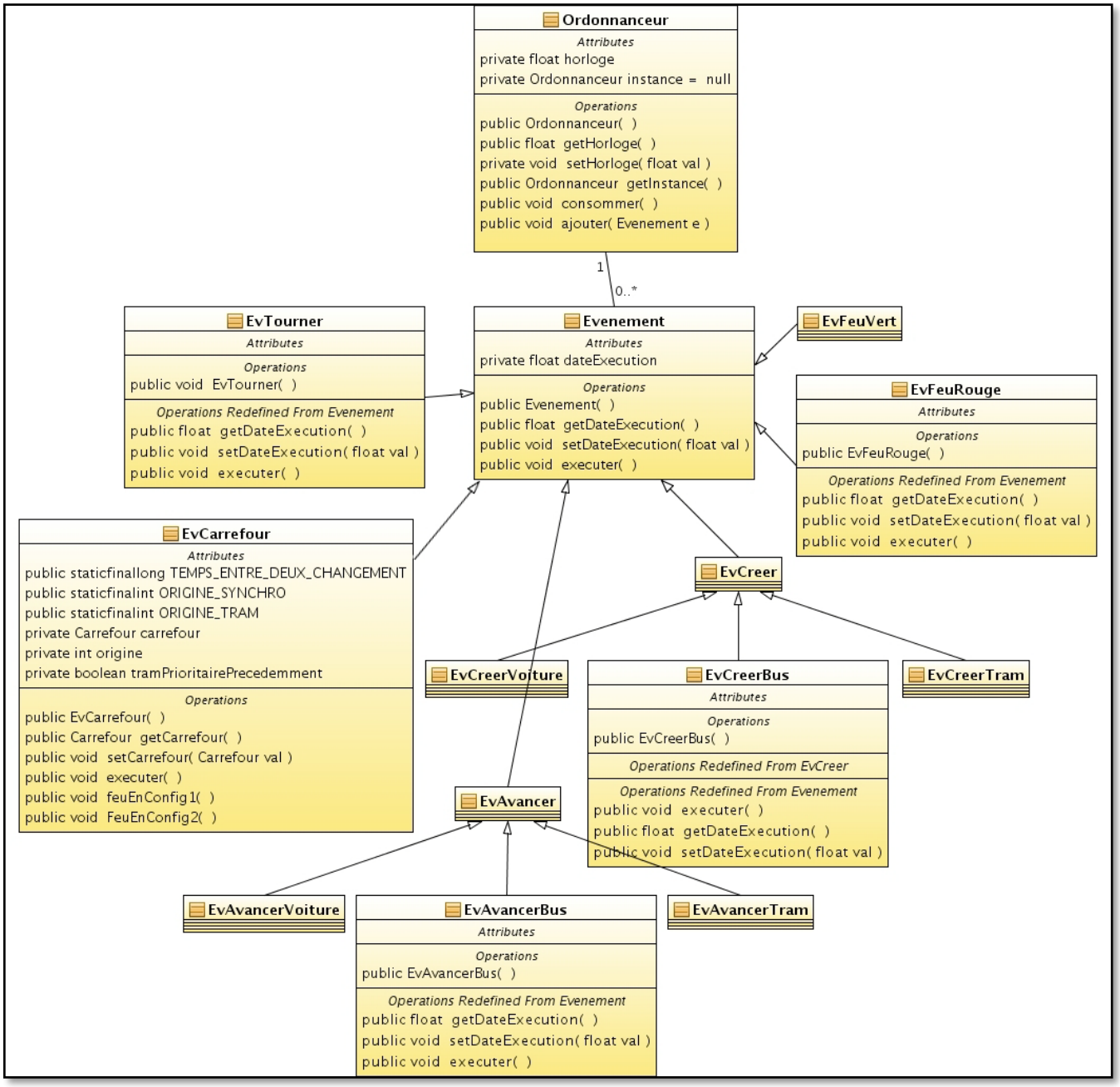


Figure 5 : Les évènements et l'Ordonnanceur

5. Mises à jour

Les mises à jour correspondent aux différentes informations qu'envoie le modèle à la vue et aux statistiques. Elles peuvent être considérées comme des évènements, mais n'ont aucune utilité dans le modèle. Nous les avons séparées des évènements du modèle car elles n'ont pas la même fonction ni les mêmes attributs.

Pour que la vue puisse afficher les véhicules, elle a besoin de leurs positions et de leurs identifiants. De même, le module de statistique requiert également les dates d'exécution des évènements pour pouvoir effectuer les divers calculs. De plus, pour différentier chaque type d'évènement (arrêt ou avancée d'un véhicule, passage d'un feu au rouge ou au vert...), nous avons utilisé la même méthode que pour les évènements, c'est-à-dire qu'ils sont tous définis par des classes distinctes héritant d'une super classe *MiseAJour*.

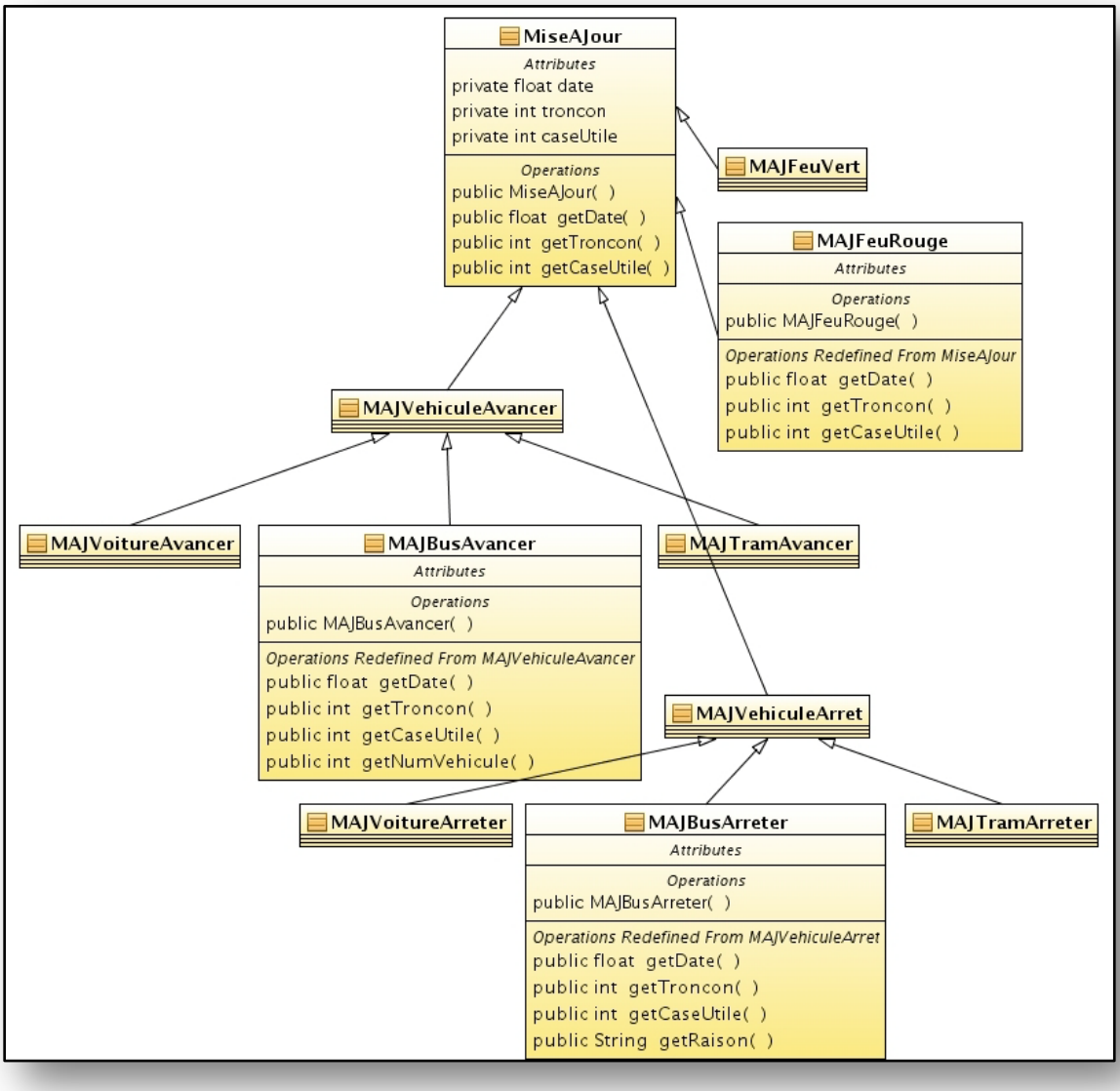


Figure 6 : Les mises à jour

6. Gestion des véhicules

Notre modèle prend en compte trois types de véhicules différents : les voitures, les bus et les trams. Leurs comportements seront très semblables, c'est pourquoi nous avons décidé de créer une super classe *Véhicule* dont les trois types cités hériteront. Ainsi nous définirons les méthodes communes de façon abstraite dans la super classe et les implanterons de manière différente dans chaque classe spécialisée. Cette façon de procéder permet d'utiliser un paradigme de programmation très présent en java : le polymorphisme.

Nous détaillerons d'avantage le comportement des voitures et nous indiquerons uniquement les modifications à apporter au comportement des deux autres classes de véhicule. Voyons alors tout de suite comment nous avons choisi de modéliser une voiture.

1. Gestion des Voitures

Le modèle a été pensé de telle sorte que la longueur d'une case soit la même que la longueur d'une voiture. Ainsi une voiture sera modélisée par l'occupation d'une case. Ceci a pour but de simplifier la modélisation des voitures. Une voiture est associée à une case et une case est occupée par un véhicule.

Pour naviguer dans la ville, une voiture doit être capable de faire les choses suivantes :

- Avancer
- Tourner
- Sortir du modèle
- Créer une voiture

Tout d'abord la création d'une voiture. Celles-ci se créent à des endroits bien précis, sur le début des axes principaux et à chaque entrée localisée au niveau des carrefours.

La création d'une voiture est une des méthodes les plus simples. Elle initialise tout d'abord tous ses paramètres, comme le *nombre de passagers*, le *comportement du conducteur* ou encore son *identifiant*. Ensuite on teste si la case de sa création est libre. Si c'est le cas, alors on crée effectivement la voiture grâce à son constructeur, qui va en même temps poster le prochain événement de création d'une voiture au même endroit. Mais si la case est occupée, alors on oublie cet événement et on en poste un prochain. Ainsi chaque objet à la responsabilité de poster les événements concernant sa fonctionnalité.

Maintenant qu'une voiture est créée, il faut bien évidemment la faire avancer. C'est une méthode maîtresse dans notre modèle que nous avons tout particulièrement soignée car beaucoup de choses sont à prendre en compte. Pour en déterminer la majeure partie de ces aspects, nous avons fait appel aux diagrammes de séquences.

En voici un exemple partiel :

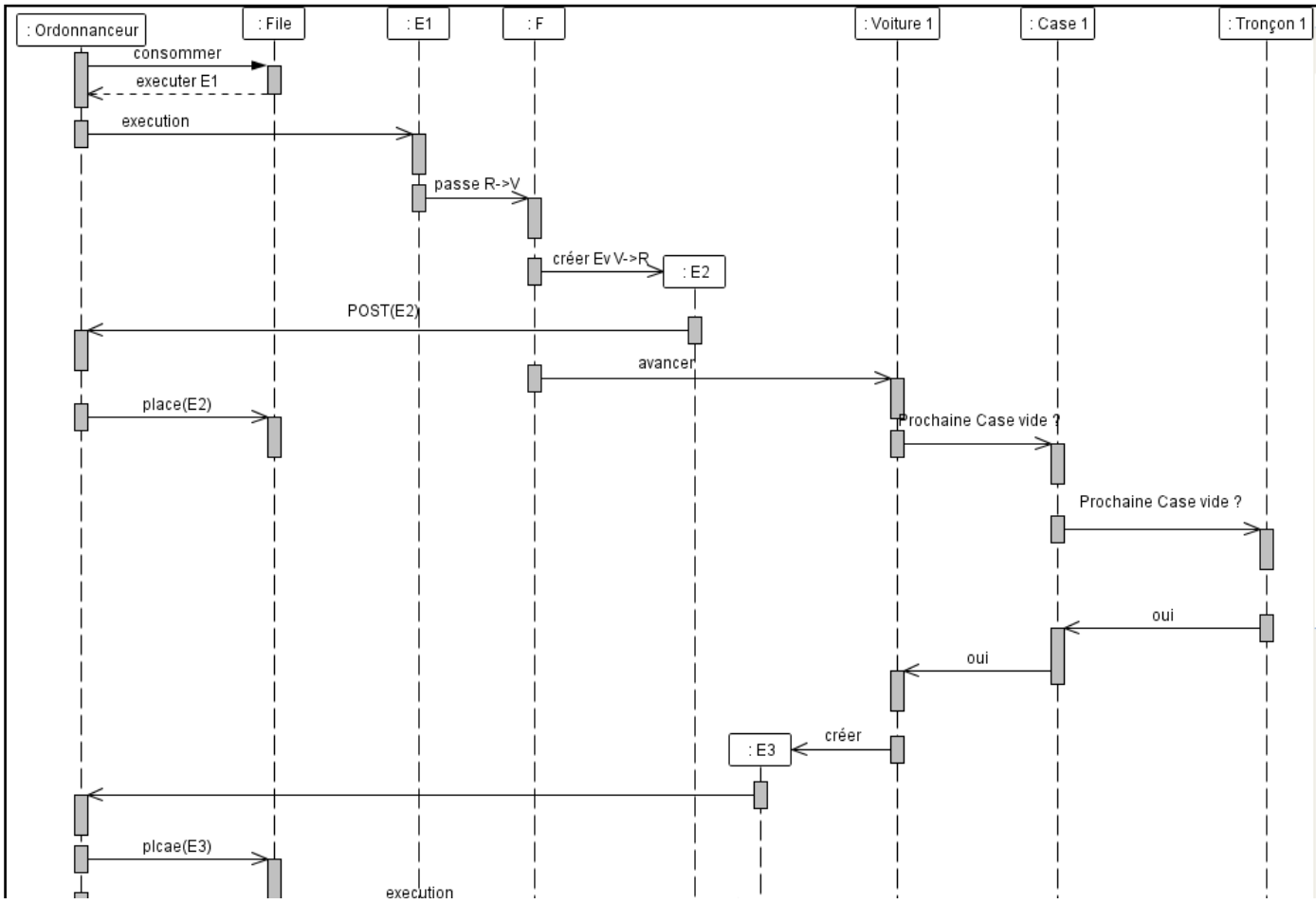


Figure 7 : Diagramme des séquences pour la méthode *avancer()*

A l'aide de cette outil nous avons pu dégager et résoudre un grand nombre de problèmes liés à la méthode *avancer()*. En voici maintenant les grands principes :

- En premier lieu on vérifie si la case où se trouve la voiture est une case avec un feu. Si le feu est rouge, on arrête la voiture et on attend que le feu passe au vert, sinon on continue le traitement.
- On teste alors si la case de destination est libre ou non. Si elle ne l'est pas, on arrête la voiture et on attend qu'elle se libère.
- Et en dernier lieu, si tout est réalisé, on poste un événement avancer qui avancera réellement la voiture à la bonne case et à la bonne date.

Au cours de ces étapes que l'on vient de décrire, nous avons créé des méthodes à part pour simplifier la lecture et la compréhension de la méthode avancer. Nous en avons créé nous permettant de déterminer la case suivante en prenant en compte le fait que la voiture puisse tourner, une autre permettant de nous dire si cette case est occupée ou non ou encore une méthode faisant sortir la voiture du modèle si la case suivante n'existe pas (cela veut dire que nous sortons du modèle).

Pour qu'une voiture puisse être réellement créée dans le modèle ou encore avancer, il faut que l'Ordonnanceur exécute les événements correspondants à ces deux

choses. Ainsi une fois que l'on est sûr que la voiture peut bien avancer, on poste l'événement qui fera avancer la voiture, puis, pour la création, c'est au moment de l'exécution de l'événement que l'on regarde si c'est possible, et si c'est le cas on le fait instantanément. Cela s'explique par le fait que la création d'une voiture est auto-suffisante, tandis qu'une voiture peut être réveillée pour avancer par l'intermédiaire d'autres méthodes.

- En effet, une voiture fait appel à la méthode *avancer()* pour les cas suivants :
- La voiture vient d'avancer et essaie de nouveau si cela est possible
- La case devant elle vient de se libérer
- Le feu sur lequel se trouve la voiture vient de passer du rouge au vert
- Dès qu'une voiture est créée, elle essaie d'avancer

Pour finir sur la description des méthodes relatives aux voitures, nous évoquerons la méthode gérant la vitesse des voitures. Car pour rendre la simulation plus réaliste nous avons voulu que les voitures accélèrent progressivement jusqu'à leur vitesse de croisière et non pas avoir une vitesse constante. Pour réaliser cela il nous faut conserver la vitesse courante d'une voiture. Voilà comment nous procédons pour réaliser cela :

- Une voiture a le droit d'avancer, il nous faut alors déterminer dans combien de temps elle aura traversé la case.
- On prend une accélération constante pour les voitures, on intègre deux fois pour avoir l'équation de la distance en fonction du temps.
- On résout cette équation en prenant comme inconnu le temps, et ainsi on détermine le temps mis pour traverser une case, et on actualise la vitesse courante.
- Si des fois la vitesse en sortie de case est supérieure à la vitesse maximale autorisée pour la voiture, on calcule le temps mis pour atteindre cette vitesse puis le temps qu'elle met pour terminer de traverser la case.

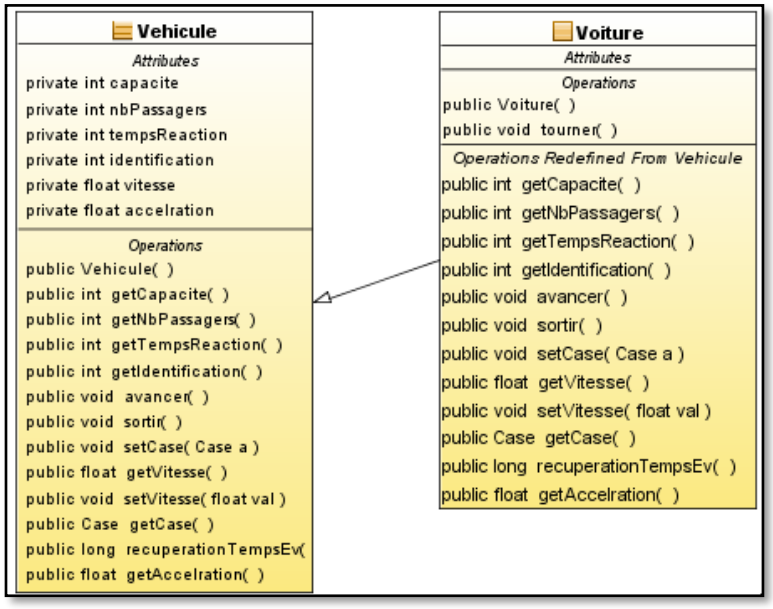


Figure 8 : Diagramme de classes de Vehicule et voiture

2. Gestion des Bus et Trams

En ce qui concerne maintenant les bus et les trams, peu de changement sont à noter. Nous avons affecté aux bus une taille de trois cases et de six cases pour les trams. Ces deux véhicules ne peuvent se déplacer que sur l'axe Est-Ouest. Ainsi ils n'ont bien évidemment pas la faculté de tourner lorsqu'ils arrivent à un carrefour.

En ce qui concerne leur création, cela se passe de la même manière que les voitures, à la différence près qu'ils ne peuvent être créés uniquement au début des axes Est-Ouest et Ouest-Est.

Et pour finir, la stratégie pour les faire avancer est exactement la même que pour les voitures. Nous ajouterons juste une capacité supplémentaire aux trams. En effet, ceux-ci seront prioritaires au niveau des feux, ainsi si un tram s'arrête à un feu rouge, il ordonnera immédiatement son passage au vert.

Pour définir toutes ces méthodes nous avons créé une classe, voiture, bus et tram héritant d'une super classe véhicule. Puis trois classes *EvCreerVoiture*, *EvCreerBus* et *EvCreerTram* servant à la création des véhicules. Celles-ci héritent de la super classe *EvCreer*. Et pour terminer nous avons encore eu besoin de définir trois classes servant à faire avancer les véhicules : *EvAvancerVoiture*, *EvAvancerBus* et *EvAvancerTram*, qui héritent d'*EvAvancer*. Nous avons aussi *EvCreer* et *EvAvancer* qui elles même héritent de la super classe *Evenement*.

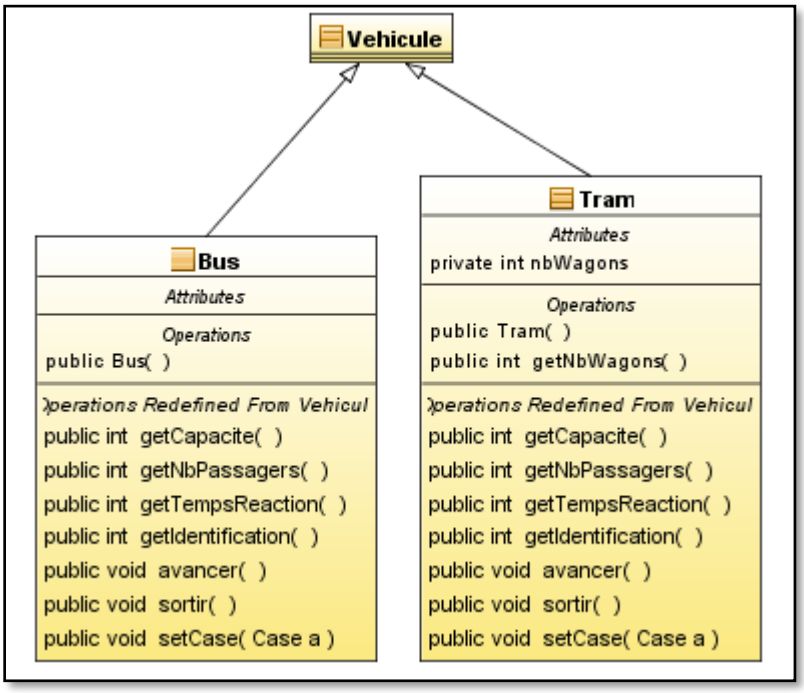


Figure 9 : Diagramme de classes de Bus et Tram

IV. GESTION ET CALCULS DES STATISTIQUES

Dans cette partie, nous allons voir comment on traite les informations utiles en provenance du modèle pour faire les calculs qui nous intéressent puis en transmettre les résultats à l'interface.

Pour la communication avec les autres entités, les statistiques ont un statut particulier puisqu'ils sont à la fois observable (par l'interface qui récupère les résultats statistiques en temps réel) et observeur (du modèle pour traiter les mises à jour).

L'utilisation des mises à jour, et la réalisation des calculs se font dans la classe principale de ce package statistiques : la classes Stats. Mais pour ces calculs, et de façon plus générale, pour modéliser les différents éléments que nous manipulons, il nous faut créer des objets très précis par le biais de plusieurs autres classes.

1. Les objets de la ville

1. Les véhicules

Pour un calcul statistique, il va de soi que nous travaillons essentiellement sur les informations portées par les véhicules eux-mêmes. L'existence d'une classe Véhicule est donc évidente. Elle a pour attributs les informations utiles concernant les véhicules : dateEnA (date à laquelle le véhicule passe au point A défini, soit la date d'arrivée à la première case du tronçon 1 ou du tronçon 2), dateEnB (date de passage en B, soit date d'arrivée sur les tronçons 75 et 76, tronçons de disparition). Ces deux dates, de type long, sont initialisées à -1. De cette façon, on peut tester si le véhicule est déjà passé à l'un des deux points en fonction de la valeur de cette date. Un objet véhicule a également comme attribut son identifiant de type entier, un booléen « isAvance » qui est vrai si le véhicule avance et faux s'il est arrêté, et un entier « tronçon » correspondant au numéro du tronçon sur lequel se trouve le véhicule.

Les méthodes de cette classe sont essentiellement celles qui permettent de récupérer ces attributs (puisqu'ils sont privés), c'est-à-dire les méthode get(), et celles permettant d'affecter des valeurs aux attributs, les set().

Mais pour effectuer les calculs, nous avons besoin de différencier les voitures, les bus et les trams. C'est pourquoi il existe une classe Donnees, dans laquelle on définit un vecteur de voitures, un vecteur de bus et un vecteur de trams (qui sont les trois attributs privés de la classe). Ces vecteurs représentent les véhicules circulant effectivement dans la ville.

Les méthodes de la classe sont les méthode get() et set() des attributs, les méthodes d'ajout et de suppression de véhicule au vecteur, et les méthodes qui permettent de récupérer un véhicule dans le vecteur ou de vérifier si le véhicule existe déjà. Tout cela sera utile par la suite pour déterminer les calculs à effectuer en fonction des mises à jour reçues.

2. Les feux

Pour calculer le temps d'attente d'un véhicule, ou bien le nombre de voitures arrêtées à un feu, il faut stocker quelque part les informations relatives aux arrêts aux feux, et aux départs. Chaque véhicule sera donc intimement lié à plusieurs objets Feu (un objet feu pour chaque Feu de la ville), qui auront chacun pour attribut le numéro d'un feu (entier privé).

Les informations que l'on doit pouvoir retirer d'un objet feu sont la date d'arrivée d'une voiture à ce feu (arrêt) et sa date de départ (redémarrage). Si le trafic est congestionné, une voiture peut être amenée à s'arrêter au feu rouge puis redémarrer au feu vert mais devoir s'arrêter encore si le feu passe au rouge avant qu'elle n'ait eu le temps de le dépasser. On aura alors plusieurs dates de départ et d'arrivée au feu. On a donc créé un objet DateFeu qui a pour attribut dateDepart et dateArrivee qui permettent, par l'appel des méthodes setDepartFeu(long dateDepart) et setArriveeFeu(long dateArrivee), d'affecter toutes les dates nécessaires à un véhicule.

Une voiture ne peut pas redémarrer d'un feu si elle ne s'y est pas arrêtée auparavant. Il faut donc, pour toute date de départ d'un feu, avoir une date d'arrivée. Dans la classe DateFeu, on a donc la méthode dateFeu(long dateArrivee), qui renvoie une erreur si on essaie de mettre une date de départ sans date d'arrivée.

De plus, si un véhicule passe un feu sans s'y arrêter, on lui affecte une date d'arrivée au feu et une date de départ du feu égales à la date de passage du feu pour que les calculs ne soient pas faussés et que pour chaque feu soit associé au moins un couple de valeurs.

2. Traiter les mises à jour et effectuer les calculs

Comme nous l'avons écrit plus haut, la classe Stats est la classe principale du package, parce que d'une part elle récupère les informations envoyées par le modèle (à savoir les différentes mises à jour qui nous intéressent) et d'autre part parce qu'elle effectue les calculs. Les lignes qui suivent expliquent comment tout cela s'enchaîne.

1. Recevoir les mises à jour

Rappelons que la classe Stats est Observeur du modèle. Ainsi, quand l'Ordonnanceur du modèle consomme une des mises à jour qui concernent les statistiques (MAJVoitureAvancer, MAJBusAvancer, MAJTramAvancer, pour le calcul du temps de trajet moyen et du temps d'attente moyen aux feux ; MAJVoitureArret, MAJBusArret, MAJTramArret, pour le calcul du temps d'attente moyen aux feux ; MAJFeuVert pour le calcul du nombre de voitures moyen arrêtées aux différents feux), la méthode Update() de Stats est appelée, avec la mise à jour en question en paramètre. On peut alors associer à chaque appel de Update() une méthode correspondant à la mise à jour (traiterEvenementNomDeLaMAJ()).

2. Traiter les mises à jour

A la réception d'une mise à jour, la première chose à faire est de rafraîchir toutes les données des véhicules et feux concernés par la mise à jour. Puis, dans un deuxième temps, on effectue éventuellement un calcul. En effet, il n'y a pas forcément lieu de faire un calcul à chaque fois : par exemple, le temps de trajet moyen et le temps moyen d'attente ne se calculant que sur le trajet de A à B, on ne fait le calcul que lorsqu'un véhicule arrive en B ! La majeure partie des méthodes du type traiterEvenementNomDeLaMAJ() est donc la mise à jour de toutes les données, dans tous les cas envisageables : on a donc eu besoin de créer un certain nombre de méthodes nous renvoyant un booléen pour tester si on était dans un cas ou dans l'autre.

3. Effectuer les calculs

Il y a trois types de méthodes de calculs à proprement parler, que nous allons commenter l'un après l'autre.

Le premier concerne le temps de trajet moyen, qui est décliné pour chaque type de véhicules. L'idée est de rechercher les voitures (resp. les bus ou les trams) qui sont passées en A à une date antérieure (puisque des voitures peuvent tourner et arriver sur l'axe n'importe où, il faut vérifier qu'elles étaient déjà sur l'axe en A pour que les calculs soient cohérents) et qui sont passées en B depuis le dernier calcul du temps de trajet moyen, et de recalculer la moyenne des temps de trajets en prenant en compte ces véhicules. Notons ici qu'il est indispensable de supprimer au fur et à mesure les véhicules qui ont été pris en compte dans les calculs. En effet, à chaque fois que l'on effectue un calcul, on sauvegarde le résultat dans la classe Résultat, puis on récupère ce résultat pour mettre à jour sa valeur. Donc les données correspondant aux véhicules déjà passés en B sont contenues dans les résultats enregistrés. C'est pourquoi il existe une méthode voiturePasseEnB() (resp. BusPasseEnB() et tramPasseEnB()), qui liste les calculs à effectuer et qui se termine par une appel de la fonction supprimerVoiture() (resp. supprimerBus() ou supprimerTram()).

On trouve ensuite le temps moyen d'attente aux feux qui, comme pour le calcul précédent, est décliné pour chaque type de véhicules. Le principe de calcul est le même que pour le calcul précédent, et le schéma d'appel aussi : quand la voiture (par exemple) a terminé son trajet, on appelle voiturePasseEnB(), qui à son tour appelle tempsAttenteVoiture(), avant de supprimer la voiture en question de la « base de données ».

Pour les deux calculs précédents on calcule une durée de parcours. On effectue donc, pour le temps de trajet moyen, le calcul suivant : date d'arrivée en B – date d'arrivée en A. Pour le calcul du temps d'attente, il faut faire la somme des temps d'attente sur le parcours. C'est pour cela que nous avons choisi d'enregistrer pour chaque véhicule, les dates d'arrêt à un feu (dateArrivée) et date de redémarrage (dateDepart). De cette façon, si un véhicule s'arrête plusieurs fois à un même feu, on lui affectera plusieurs couples de valeurs et l'on pourra effectuer la soustraction date de départ du feu – date d'arrivée au feu.

Enfin, on calcule aussi le nombre de voitures à chaque feu. Il a été décidé d'effectuer ce calcul lorsqu'un feu passait au vert : en effet, c'est à ce moment là que le nombre de voitures arrêtées devant est maximal, et c'est celui qui nous intéresse pour savoir si la ville est congestionnée ou non. Pour cela, on parcourt également la « base de données », à la recherche des voitures arrêtées à un feu en particulier (son numéro étant passé en argument de la méthode), quelque soit leur parcours final. En les comptant, on a directement le résultat voulu.

3. Envoyer les résultats

1. Enregistrement des résultats

Suite à certaines discussions, nous avons décidé d'enregistrer nos résultats statistiques pour alléger les calculs. En effet, à chaque nouvel événement, on met à jour les statistiques en moyennant avec les valeurs précédentes. Pour ce faire, il y a deux possibilités : enregistrer tous les évènements, les classer par type et reprendre tous ceux qui nous intéressent à chaque fois que l'on doit mettre un résultat à jour. On peut aussi enregistrer le résultat à chaque calcul puis, lorsque l'on doit mettre à jour la valeur statistique, on récupère le résultat et on le moyenne avec le nouvel événement. De cette façon, on réduit considérablement le nombre de données à garder en mémoire.

C'est cette dernière méthode qui a été retenue pour alléger les calculs. Nous avons donc créé une classe Resultat dont les attributs sont les informations qu'il faut enregistrer pour mettre à jour les calculs à chaque événement.

On enregistre les temps d'attente moyen pour chaque type de véhicules (tempsAttenteVoiture, tempsAttenteBus, tempsAttenteTram) ainsi que le nombre total de véhicules de chaque type passés en B (nbVoitureTotal, nbTramTotal, nbBusTotal). De cette façon, à chaque mise à jour qui influe sur le temps d'attente d'un véhicule, on multiplie le temps d'attente moyen de l'ensemble des véhicules par le nombre de véhicules pour obtenir un temps d'attente total de ce type de véhicule. On peut alors ajouter le temps d'attente total du véhicule considéré et moyenner ce temps en divisant par le nombre total de véhicules rafraichi.

On enregistre également le temps de trajet moyen pour chaque type de véhicules (tempsTrajetVoitire, tempsTrajetBus, tempsTrajetTram), le nombre de voitures moyen arrêtées à un feu (un vecteur d'entiers nbVoiture ayant une composante pour chaque feu) ainsi que le nombre de fois où chaque feu est passé au vert (vecteur d'entiers nbFeuVert) pour moyenner le nombre de voitures arrêtées à chaque feu. Cette dernière variable est actualisée lorsque l'on calcule le nombre de voitures à un feu. En effet, cette méthode n'est appelée qu'en cas de passage d'un feu de rouge à vert, donc quand on reçoit une mise à jour de type feuVert.

2. Envoi des statistiques à la vue

La vue doit recevoir les résultats statistiques dans deux cas différents. Tout d'abord, au cours de la simulation, la vue reçoit les résultats en temps réel. C'est-à-dire

que dès que le module statistique reçoit une mise à jour intéressante, il effectue les calculs correspondants et envoie les résultats par la méthode notify(). En effet, les statistiques sont observables par la vue.

La vue doit également mettre à jour les résultats à la fin de la simulation, ce sont les statistiques post-mortem. Pour accéder à ces résultats, nous avons définit une interface dans le package Reseau :

StatCalculees statsPostMortem() throws RemoteException;

Pour l'envoi des résultats, nous avons implémenté la classe StatCalculees. Cette classe est donc serializable pour être envoyée à la vue. Ses attributs sont les résultats effectifs des calculs. Ces données sont mises à jour à partir des attributs de la classe Resultat. La différence entre ces deux classes est la façon de mettre à jour les attributs et les attributs eux-mêmes. Dans la classe StatCalculees, on a seulement définit comme attributs les sept informations à afficher : le temps moyen de parcours de A à B pour les trois type de véhicules, le temps moyen d'attente sur le parcours pour les trois types de véhicules et le vecteur contenant le nombre de voitures arrêtées en moyenne à chaque feu.

A chaque notify(), on met en paramètre la méthode statsTempsReel() de la classe Stats. Cette méthode met à jour les attributs d'un objet de la classe StatCalculees à partir de ceux de la classe Resultat, par des méthodes de type set. Elle retourne cet objet StatCalculees qui est ensuite transmis à la vue par le notify().

Pour envoyer les statistiques post-mortem, la vue appelle la méthode statsPostMortem() qui a le même fonctionnement que statTempsReel(). Elle met à jour un objet StatCalculees puis le retourne.

La vue peut ainsi obtenir toutes les informations à afficher qui ont été définies dans le cahier des charges.

V. DEVELOPPEMENT DE L'INTERFACE GRAPHIQUE

Notons dés à présent que chaque phase de développement est suivie d'une phase de test et d'une validation des modifications.

1. Son rôle dans le projet

L'interface Graphique représente l'IHM de notre projet, ou Interface Homme Machine. C'est donc elle qui devra contenir tout ce qui va lier l'utilisateur aux packages *Modèle* et *Statistiques*. Ainsi, certaines modifications de l'interface, à travers ses boutons ou autres, affecteront l'interface elle même, d'autres ne l'affecteront peut être pas directement.

La réalisation de ce package passe donc par un fort travail de concertation avec les deux autres parties du groupe afin de collecter tous les paramètres dont chacun va avoir besoin. Le moindre oubli d'une fonction dans l'interface aura pour conséquence l'inutilisation totale d'une certaine fonction de la classe Modèle ou de la classe Statistiques. Mais "l'inventaire" de fonctions nécessaires qui précédera le développement devra bien évidemment être constamment mis a jour pour satisfaire les nouvelles exigences du client, de la partie *Modèle* ou des *Statistiques*.

Mais l'interface Graphique, c'est aussi la partie Graphique justement. Elle devra donc afficher une carte représentant les routes et des véhicules se mouvant en synchronisation avec l'apparition d'événements dans la partie *Modèle* ou la partie *Statistiques*. Cet aspect qui concerne davantage l'interface Graphique elle même ne devra pas être négligée non plus. Son développement pourra se faire de façon isolée par rapport aux autres groupes. On s'attachera ici à créer une vue la plus esthétique possible avec les connaissances et les moyens qui sont nôtres. Un compromis apparaîtra forcement lors du développement : assurer une complexité visuelle suffisamment importante sans pour autant ralentir de trop la vitesse de simulation.

Ainsi on peut affirmer que si la partie Interface est davantage centrée sur la rigueur (il ne faut négliger aucune fonction), la partie Graphique sera, elle, davantage centrée sur l'esthétique et l'efficacité. La majorité de notre temps de travail dans notre groupe sera accordée à cette partie Graphique, de simples composants JAVA Swing de type JButton ou JSlider étant suffisant pour la première.

L'acquisition de nouvelles connaissances sera nécessaire tout au long de ce projet si l'on veut réaliser une interface ne serait ce que suffisante vis a vis de notre cahier des Charges. On apprendra donc à utiliser le modèle Vue Contrôleur au sein même de notre interface mais aussi de nouveaux composants JAVA Swing tels que les JfileChooser ou JScrollpane. Toutes ces nouvelles connaissances seront abordées dans le résumé des différentes grandes phases de développement qui suit.

2. Notre Cahier des Charges

La partie Interface Graphique à son cahier des Charges propre défini au début du projet. Il est composé des exigences du client et conséquemment des exigences des parties *Modèle* et *Statistiques*.

Aux niveaux des exigences des deux autres parties, elle devra :

- envoyer à la partie *Modèle* les paramètres initiaux suivants :
  - Temps de Simulation,
  - Nombre d'utilisateurs (la Charge),
  - Nombre de bus par heure,
  - Nombre de trams par heure,
  - Pourcentage d'utilisation du bus,
  - Pourcentage d'utilisation du tram,
- envoyer à la partie *Modèle* les informations suivantes durant la simulation:
  - Démarrer la simulation,
  - Mettre la simulation en pause,
  - Arrêter la simulation,
  - Passer en mode Pas-à-Pas,
  - Indiquer la vitesse de simulation voulue.

Enfin, le client lui-même a certaines exigences à savoir :

Pouvoir sauvegarder les paramètres initiaux pour pouvoir les réutiliser,
Afficher durant la simulation des statistiques effectuées en temps réel,
Afficher à l'écran une carte montrant le déplacement des véhicules et le changement des feux en temps réel
Pouvoir zoomer sur la carte,
Pouvoir cacher la carte afin d'accélérer la simulation,
Afficher à la fin de la simulation :

- Les statistiques post-mortem de la simulation,
- Afficher sous forme de courbes les temps d'attente pour chaque type de véhicules, en fonction de la charge

3. Les préliminaires au développement

A la base du développement de l'Interface Graphique, il y eu concertation du groupe de projet afin d'élaborer l'idée commune que l'on avait du futur logiciel. Certaines "directions" de développement furent ainsi apportées à notre partie : les portions de routes entre deux feux seront modélisées par des tronçons. Ces tronçons de route seront composés de cases. Une voiture occupe une case entière, comme une case à une longueur "absolue" d'environ 5-6 mètres dans la réalité, on en déduit le nombre de case que contient chaque tronçon. Toute la carte est finalement modélisée concrètement sur papier sous forme de cases et de tronçons, chaque partie du projet en a une copie(Voir annexes).Le graphique final obtenu après simulation représentera la

variation des temps d'attente énoncée plus haut en fonction de la charge, les trois courbes étant superposées.

L'interface Graphique sera, quand à elle, réalisée sous JAVA en utilisant la classe Swing et composants du même nom.

4. Répartitions des taches

Comme on l'a dit, l'interface Graphique est réalisée à l'aide de composants affichables à l'écran : les composants Swing. Apres réflexion sur les fonctionnalités demandées de l'interface, il est rapidement décidé que le logiciel sera composé de trois fenêtres principales. Il a ainsi été décidé que :

Cédric s'occupera du développement de la première fenêtre qui doit gérer la saisie par l'utilisateur des paramètres initiaux, la sauvegarde de ces paramètres et leur chargement après.
Omar s'occupera de la réalisation de la fenêtre principale avec affichage de la carte, gestion des saisies de contrôle de simulation (Stop, Play, etc.) et affichage des statistiques temps réel.
Nicolas s'occupera du développement de la fenêtre finale qui affiche les résultats post-mortem et le graphique.

Au niveau de ces trois fenêtres, Anthony s'occupera du code permettant de se connecter au réseau et d'envoyer ou récupérer des données des 2 autres packages.

5. Développement de la première fenêtre.

1. Création de la partie visuelle.

On crée une première classe*FenetreDemarrage* qui hérite du code de la classe *JFrame*, composant Swing qui gère l'affichage de Fenêtre. On y place des *Jslider*, composant Swing qui au déplacement d'un curseur par rapport à une barre renvoie la valeur entière choisie. Pour récupérer cette valeur et simplement pour indiquer au logiciel de lire les actions de la souris sur les images de ces composants à l'écran, on fait en sorte que notre classe implémente les *listeners* de souris nécessaires. On ajoute des composants Swing *JButton* à notre classe afin de simuler des boutons à l'écran. On y place d'autres *listeners* de souris afin de détecter l'action sur ses boutons. Pour le texte, des Composants *Jlabel* permettent d'afficher du texte où l'on veut dans la fenêtre et des *JTextfield* permettent d'afficher des cadres où l'on pourra à la fois afficher et récupérer du texte.

Enfin les composants Swing *JMenuBar, JMenu et JMenuItem* permettent de réaliser un menu on l'on met les options d'enregistrement et de chargement. A l'appui sur ses boutons on affiche alors à l'écran une nouvelle fenêtre qui affiche l'explorateur de documents de la machine sur laquelle on démarre le package *visuelle.* Ce composant Swing est un *JFileChooser*.

Figure 10 : JfileChooser

Un composant *JCombox* permettra de choisir la résolution. Ce choix non demandé par le client permettra simplement une meilleure gestion de l'affichage sur écran des autres fenêtres. Finalement, on obtient visuellement la fenêtre ci dessous :

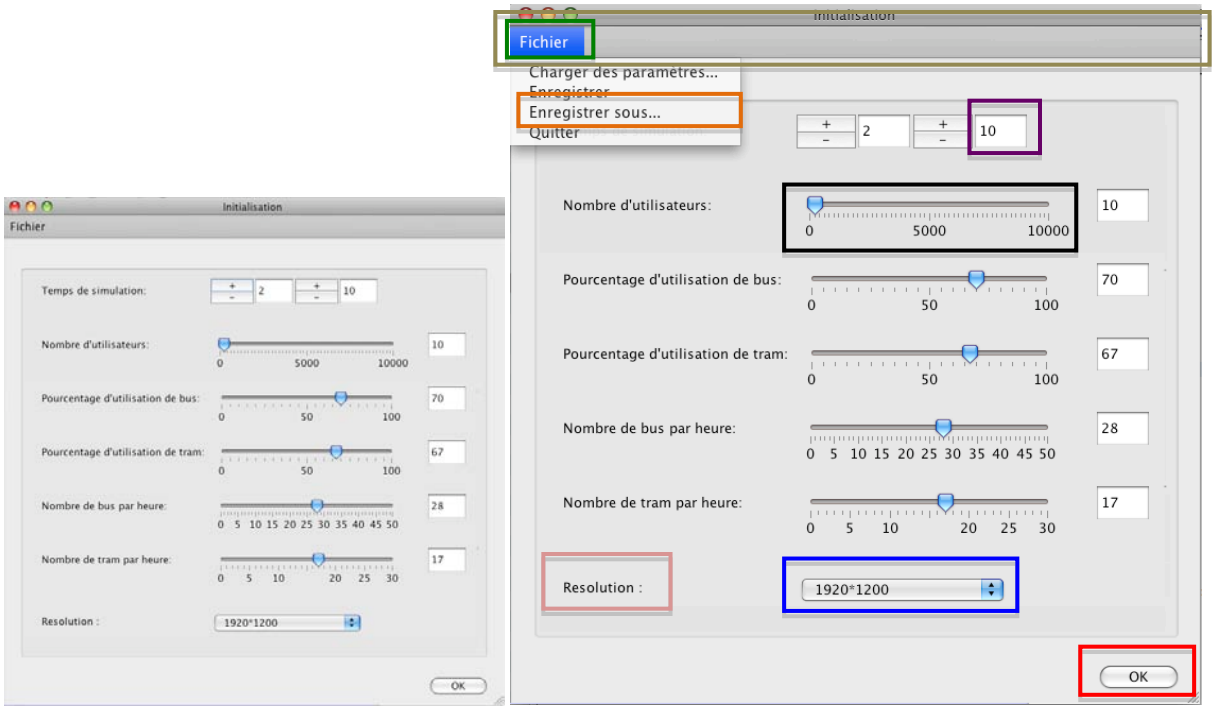


Figure 11 : Première fenêtre

2. Mise en place de la Sauvegarde et du Chargement

Après quelques recherches sur la façon d'utiliser les *JfileChooser*, on code le chargement puis la sauvegarde. Pour la sauvegarde, il faut gérer les tâches suivantes :

Récupération du texte écrit par l'utilisateur et du chemin qu'il a choisi
création d'un fichier texte *.txt* du même nom
écriture dans ce fichier des paramètres initiaux

Pour le chargement, il faut gérer les tâches suivantes :

Récupération du chemin du fichier choisi par l'utilisateur,
Ouverture de ce dernier et récupération de la chaine caractère incluse dedans,
Découpage au niveau du caractère "/ " et récupération des entiers
On "réinjecte" ces valeurs dans la fenêtre en modifiant les valeurs des *JSlider* et autres *JTextField.*

On ajoute ensuite une sauvegarde automatique grâce à la méthode *enregistrer().* A l'appui sur l'item "enregistrer" du menu, un fichier se crée dans le dossier Sauvegarde situé à l'adresse contenue dans l'attribut CheminSave de la classe Main. Son nom est composé de la valeur des paramètres précédés de "*simul_*". Les deux premiers nombres indiquent le temps de simulation choisi affiché en heures. Par exemple : pour le fichier Simul_0-10-70-67-28-17.txt, le texte écrit à l'intérieur est le suivant :

```
Temps de simulation: 600
Pourcentage d'utilisation de bus: 70
Pourcentage d'utilisation de tram: 67
Nombre de bus par heure: 28
Nombre de tram par heure: 17
NuméroRésolution: 2
FIN DE LA SAUVEGARDE DES PARAMETRES
Nombre d'utilisateurs: 5600/13000/2300/8900/6000/
Temps moyen d'attente des voitures : 123000/56000/86000/220000/
Temps moyen d'attente des bus : 99000/78666/21623/456789/
Temps moyen d'attente des Tram : 36987/123879/56987/12304/
FIN DU CHARGEMENT DES RESULTATS
```

Le temps de simulation est de 600 minutes soit 10 heures et 0 minutes. D'où le nom du Fichier.

Mais la volonté d'afficher une courbe utilisant une certaine forme d'historique du logiciel, nous oblige à compliquer un peu la méthode d'enregistrement. En effet, comme on le voit plus haut, on sauvegarde aussi dans ce fichier les résultats successifs des

simulations effectuées avec le même groupe de paramètres. A une charge donnée, on enregistre les trois temps moyens trouvés à la fin.

Or la charge doit justement être enregistrée par la fenêtre de démarrage. C'est la raison pour laquelle on voit qu'alors que chaque paramètre a 4 sauvegardes, 5 charges sont enregistrées. La 5ème sauvegarde des résultats sera faite par la dernière fenêtre du logiciel (voir la partie Développement de la fenêtre finale).

Finalement le déroulement total d'une sauvegarde est le suivant :

Lorsque l'utilisateur effectue un chargement de paramètres, on charge les paramètres mais on sauvegarde en même temps dans la classe les valeurs de ces paramètres chargés. La charge n'est pas chargée dans la fenêtre puisqu'elle va être modifiée. On sauvegarde aussi tout le texte compris entre " FIN DE LA SAUVEGARDE DES PARAMETRES" et "FIN DU CHARGEMENT DES RESULTATS".

Après modification de la charge, lorsque l'utilisateur appuie sur enregistrer, on écrit dans le fichier portant le nom donné par ses paramètres. On écrit donc dans le fichier que l'on avait chargé en effaçant par la même le précédent texte. On réécrit les valeurs des paramètres que l'on a sauvegardés puis tout le texte qui était compris entre " FIN DE LA SAUVEGARDE DES PARAMETRES" et "FIN DU CHARGEMENT DES RESULTATS". A la fin de la ligne concernant le nombre d'utilisateurs, on ajoute la nouvelle charge choisie puis "/".

Le fichier contient l'historique des différentes simulations.

3. Améliorations apportées

Par rapport au simple cahier des charges, des améliorations ont été apportées à cette fenêtre :

Les fonctions chargement et enregistrement étant créées, on en profite pour faire une sauvegarde automatique dans un fichier "LastSession" des paramètres dés que l'utilisateur appuie sur "OK". Ils sont restaurés ainsi automatiquement dés le démarrage du logiciel sans avoir forcément fait d'enregistrement.

La taille de la fenêtre est bloquée afin d'éviter que l'utilisateur ne perturbe l'affichage de la fenêtre sans le vouloir.

Un appuie sur le bouton "OK" ferme la première fenêtre et crée la deuxième que nous présentons dés à présent , en lui envoyant comme paramètres de construction le nom du Fichier, son chemin dans l'ordinateur, le temps final, la résolution choisie et les 5 paramètres des *JSlider.*

6. Réalisation de la carte visible dans la deuxième Fenêtre

1. Développement de la 1ère version de la fenêtre

La carte-Vue, ses tronçons et ses cases sous formes de JPanel

La carte a évoluée tout au long du projet à l'image de nos connaissances. On crée la d'abord le deuxième objet fenêtre *InterfaceGraph*.

Dans la première version de la carte, les routes sont représentées par des tronçons modélisés dans un premier temps par des composants Swing *JPanel* qui sont de simples conteneurs colorés ou non dans lesquels ont peu mettre d'autres composants *JSwing*. On plaçait ensuite dans ces tronçons d'autres *Jpanel* modélisant les cases. Toutes les cases étaient simplement grisées et les cases pleines (véhicule et/ou feu) étaient emplies de *JLabel*contenant l'image du feu et/ou de la voiture. Tous les tronçons étaient ensuite placés dans un *JPanel* plus grand pouvant contenir toute la carte. On créait ainsi un premier objet CarteVue qui utilise les objets créés *Troncons* et *CaseVue.*

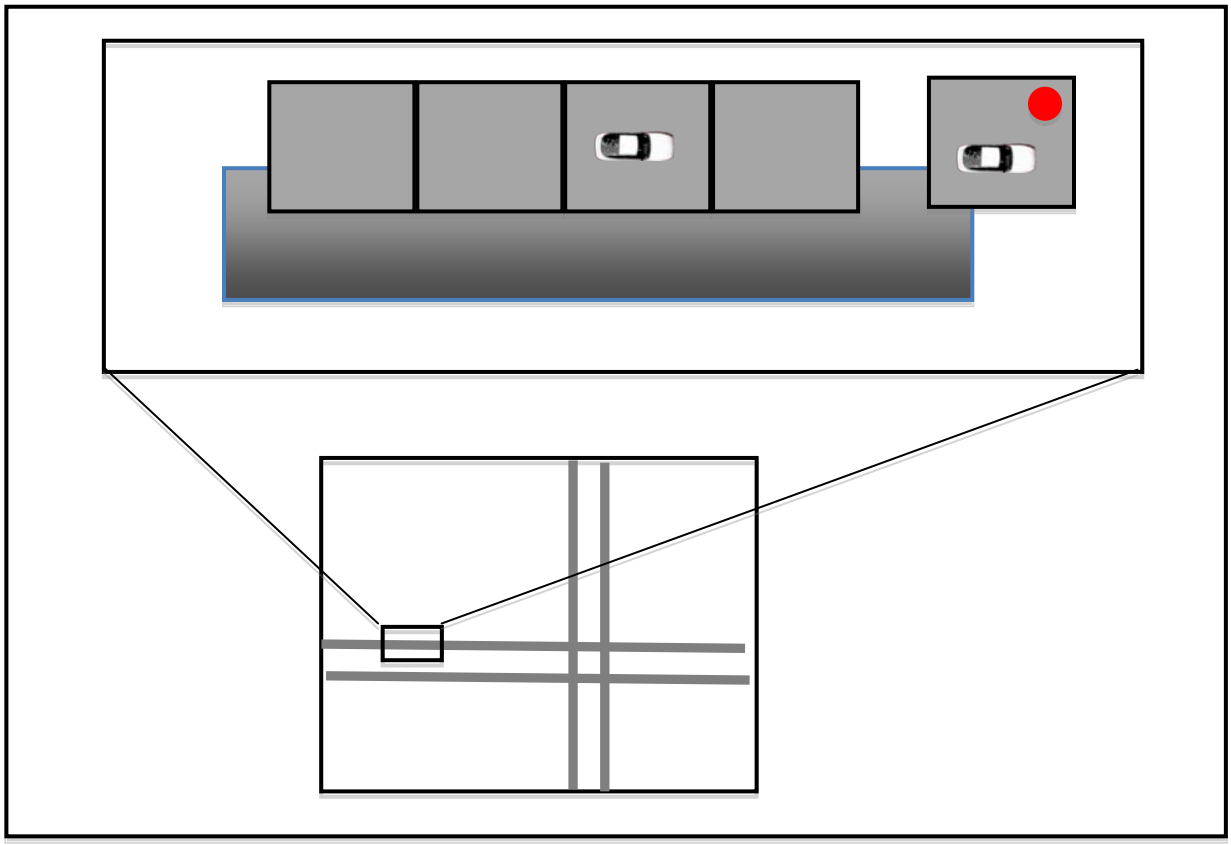


Figure 12 : 1ère Architecture de la carte

Les images

Concernant les images présentes dans le logiciel, elles sont toutes récupérées sur internet puis rognées et redimensionnées. La méthode *setIcon()* des *JLabel* permet ensuite rapidement de les placer à l'intérieur. En modifiant, le *Layout* des *JPanel* on peut si l'on veut placer plusieurs images par *JPanel*.

Le JScrollPane

La carte assez grande contient tout de même 325 cases en hauteur et en largeur. Soit même avec des cases carrées ayant simplement 5 pixels de coté, on aboutit a une carte de 1625*1625 pixels. Ce format trop grand n'est pas pris en compte par tous les écrans. On a donc tout de suite eu besoin d'utiliser un autre composant *Swing* : le *JScrollPane*

JScrollPane est un conteneur permettant de munir un composant de barres de défilement. Ceci permet de visualiser des composants plus grands que l'espace dans lequel ils sont visualisés. Le composant *scrollé* doit implémenter l'interface *Scrollable*.

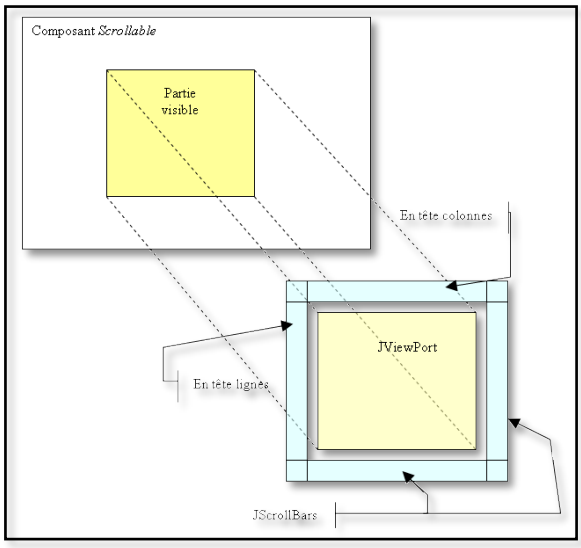


Figure 13 : Le JScrollPane

Le Zoom

Une première fonction de Zoom est ensuite codée, le *JScrollPane* assurant la bonne insertion de celui ci dans la fenêtre. Pour cela, on fait en sorte que la carte implémente un listener de souris. Ainsi lorsque l'utilisateur scrolle au niveau de la molette de la souris, la taille des cases en pixel dans le code change et ainsi la carte apparait plus grande ou plus petite. On crée une dizaine de version de chaque image des véhicules et feux correspondant aux différents Zoom, dés que le zoom passe un certain pas l'image est changée par une image plus grande. Enfin, à chaque modification du zoom la carte est recrée ce qui est assez pénalisant pour la rapidité du logiciel.

Création du panneau de test

Le fait que la partie modèle n'ait à ce moment pas encore un code utilisable mais surtout notre volonté de pouvoir améliorer notre partie indépendamment des autres parties, nous a poussé à créer un autre objet JAVA de type *JPanel* : le *PanneauEssai* de la classe du même nom. Ce panneau contient quelques boutons permettant de:

- Ajouter un véhicule sur une case donnée,
- Avancer un véhicule choisi d'une case,
- Déplacer un véhicule sur une autre case pas forcement juxtaposée,
- Supprimer un véhicule,

On ajoute ensuite un *JSlider* à 3 valeurs permettant de choisir le type de véhicule entre voiture, tram et bus.
La classe CaseVue, qui représente les cases visibles, implémente enfin un listener de souris qui va permettre, lorsqu'on clique sur une case, de renvoyer directement son numéro et le numéro du tronçon sur lequel elle est dans des *JTextField*présents dans le panneau. A ce stade, à chaque modification de la carte (apparition, disparition, mouvement d'un véhicule) une nouvelle carte est recrée ce qui est aussi assez pénalisant pour la rapidité du logiciel.

Finalement, on peut, par ce panneau, simuler l'action qu'aura la partie modèle sur notre partie à travers le réseau. Il fonctionne sans problème.

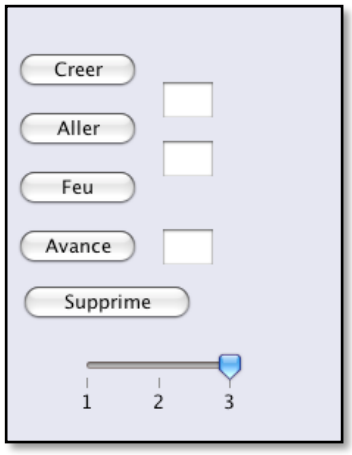


Figure 14 : Panneau Essai

2. Amélioration du Zoom

La première version du zoom, puisqu'elle recrée une nouvelle carte, replaçait systématiquement le *JViewport* en haut à gauche de la *CarteVue* ce qui est assez gênant à l'utilisation. On aimerait que le zoom se fasse par rapport au point qui est au centre du *JViewport*. Sur les conseils de M. Bouthier, on améliore donc cette fonction, et l'on parvient au résultat voulu. Une simple sauvegarde de position antérieure au changement de zoom permet de réussir cela. La même personne nous apprend ensuite comment modifier une image pour en obtenir une résultante de taille différente grâce à une méthode interne à JAVA.. En voici un bout de code :

```
Image image = new ImageIcon(CheminImage + "image.png").getImage();

ImageIcon imageIcon = new ImageIcon(image.getScaledInstance( int echelleX,
int echelleY, Image.SCALE_DEFAULT));

JLabel jlabel = new JLabel() ;

Jlabel.setIcon(imageIcon) ;
```

Par cette méthode, nous n'avons plus à avoir plusieurs exemplaires d'une image pour qu'elle s'adapte aux zooms. Avec une unique image, on s'adapte aux zooms

différents en réappliquant la méthode avec une échelle différente. (on passe de 900 images théoriques à 90 !). L'ajout d'une image à la carte à partir de ce moment fut chose aisée (il fallait avant retailler l'image à la taille d'affichage réel au pixel juste dans un logiciel!).

3. Amélioration du déplacement dans la fenêtre

Jusqu'à présent la navigation dans la fenêtre se faisait grâce aux 2 barres situées autour du *JViewport*. Grâce à l'aide de M. Bouthier, on modifie le code pour que la navigation se fasse de plus façon plus instinctive en faisant simplement glisser la *CarteVue* dans le *JViewport* à l'aide de la souris (drag). Pour cela, on ajoute des listeners de souris au *JViewport*. On décale verticalement et horizontalement la *CarteVue* d'une distance égale aux déplacements de la souris entre l'appui sur l'un de ses boutons et le relâchement de celui-ci. On sauvegarde en fait la position de la souris dans le *JViewport* à chaque appui (le point *startDragPoint)*, et au relâchement on fait la soustraction de la nouvelle position de la souris à ce moment avec la position sauvegardée.

4. Modification de l'architecture de la Carte

Jusqu'à présent, la *CarteVue* était un objet *JPanel* contenant d'autres *JPanel* tronçons puis cases. Toutes les informations de la carte étaient enregistrées dans les classes *CarteVue*et *CaseVue* et *Tronçons.* Cela ne correspondait pas aux modèles vue-contrôleur que l'on voulait utiliser.

On crée donc deux nouveaux objets *CarteModel* et *CaseModel* qui vont jouer le rôle de contrôleur.

Les informations de la carte telles que sa taille, son nombre de tronçons, le nombre de véhicules présents dessus et leurs positions sont à présent enregistrées sous forme d'attributs dans la *CarteModel*. La classe *CarteVue* représente à présent simplement la vue du modèle Vue-Contrôleur. Elle va donc chercher les informations dans la *CarteModel* pour afficher les tronçons et cases comme il faut : elle ne s'occupe plus que de l'affichage. L'attribut principal de la *CarteModel* est ainsi un tableau de *Tronçon* qui contiendra toutes les informations "dures ". L'action sur l'un des paramètres du *PanneauEssai* va modifier un attribut de la classe *CarteModel* pour sauvegarder le changement, cette même classe va ensuite commander une nouvelle création de carte à *CarteVue*, celle ci s'affichant selon les informations contenues dans *CarteModel.*

De même pour les cases, *CaseModel* conserve les informations telles que la position de la case dans la carte et le véhicule présent dessus. La classe *CaseVue* ne gère à présent plus que l'affichage à l'écran des Cases.

5. Modification de l'affichage de la carte

On réfléchit à une amélioration de l'affichage de la carte tant au niveau de son code qu'au niveau simplement esthétique. La carte ne contient en effet à ce moment que des routes de couleurs grises et les véhicules, il n'y a encore aucun élément de décor. La taille totale de la carte lors d'un zoom important empêche de placer simplement une énorme image de fond d'un seul tenant représentant tout le décor de la ville : au plus

fort niveau de zoom cette image aurait une taille de 38.000*38.000 pixels. Les ressources de nos ordinateurs géreraient très mal cela.

On note de plus à ce point que les tronçons représentés sous forme de *JPanel* sont recouverts par d'autres *JPanel* que sont les cases. On ne les voit donc pas à l'écran et il ne sert à rien de les afficher.

Les cases vides enfin sont aussi modélisées sous forme de *JPanel* coloriés en gris. Il serait beaucoup plus simple d'avoir des routes dessinées en permanence, et qu'à l'apparition d'un véhicule sur une case un *JLabel* soit mis à l'endroit du véhicule sur la carte. Ainsi une case vide ne générerait aucun affichage supplémentaire et le code serait plus "léger". De même les feux, seraient de simples *JLabel* mis sur l'image des routes.

Une carte plus efficace

On modifie donc dans un premier temps les classes *Troncon* et *CaseVue*.
L'objet *Troncon* n'est à présent plus un objet Swing *JPanel* mais un simple tableau de case, de *CaseModel* pour être précis, qui va pointer vers les cases qui le composent.
L'objet *CaseVue* n'est à présent plus un *JPanel* mais un *JLabel.* Lorsque *CarteVue*, va vouloir construire une carte, il va créer la carte de fond puis au dessus des *JLabelCaseVue*non vide(si la case est vide : pas de création, on ne fait rien) représentant les véhicule et/ou feu. Notons qu'un véhicule seul est une image, un feu seul est une image et enfin qu'un véhicule sur un feu est une troisième image : on ne superpose pas les *JLabel.* Au final, une *CaseVue*EST UN JLabel qui contient UNE image. Une*CaseVue* VIDE N'EXISTE PAS dans le logiciel.

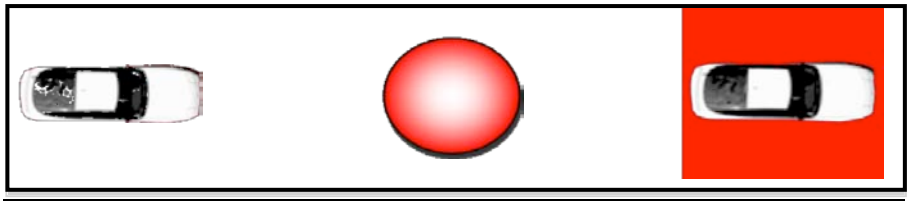


Figure 15 : Les Jlabels Vehicules et feux

Enfin, afin d'améliorer l'affichage, on fait en sorte que les méthodes de création, de mouvement et de suppression de véhicules ainsi que celles de changement de feu modifient directement l'endroit de la carte où se situe le changement sans recréation complète de la carte. Par exemple lorsqu'on fait avancer une voiture, on efface simplement le *JLabel* la représentant où elle était avant, et on le replace à sa nouvelle position. Le reste de la carte ne change pas. Ainsi, *CarteModel,* plutôt que de commander la création d'une nouvelle *CarteVue* à chaque événement reçu par le modèle, va simplement commander ces apparitions, disparitions et changement de *CaseVue(s)*.
L'affichage devrait s'en trouver accéléré.

Une carte plus belle

Parlons à présent de la carte de fond. A ce stade, la carte est totalement grise et vide mis à part aux endroits où il y a des véhicules et/ou feux. Plutôt que d'y placer une

seule grande image de fond, on place plusieurs fois la même image que l'on redimensionne grâce à la méthode vue plus haut. On peut aussi choisir des parties de la carte où cette image de fond est modifiée par une autre de même taille. On place ainsi dans la carte un décor un minimum diversifié. Ces éléments de décor seront ajoutés par la *CarteVue*.
On ajoute sur cette carte, en tant qu'élément du décor, les routes, carrefours, ponts, etc.
Après un test, deux constations apparaissent: l'affichage est certes beaucoup plus riche et esthétique mais cette méthode utilise tout de même trop de mémoire. L'explication est rapidement trouvée : aux plus hauts niveaux de zoom, on demande à l'ordinateur de modéliser en une fraction de seconde plus de mille images carrées de 100 *100 pixels soit au total une carte de 100.000*100.000 pixels !

Une nouvelle méthode d'affichage pour le décor

On décide alors de travailler sur le *JViewport* pour parvenir à améliorer cet affichage. Le *JViewport* est la partie du *JScrollPane* correspondant à la zone visible de la carte (voir le schéma du *JScrollPane* plus haut). Sa taille dépend donc de la taille de la fenêtre principale et plus précisément de la taille de l'élément qui le porte : le *JScrollPane.* Sa position renvoie la position de la partie visible de la carte dans la carte réelle. On utilise ces propriétés pour n'afficher à la création d'une carte QUE les éléments de décor visible dans le*JViewport.* Afin de conserver un affichage continu des véhicules et des feux, ces derniers sont en revanche affichés en permanence même dans la partie non visible du *JViewport.*

On teste et le résultat est selon nos attentes : l'affichage, grâce aux modifications effectuées plus haut (*une carte plus efficace*) est rapide et le décor ne s'affiche que dans le *JViewport*comme voulu.

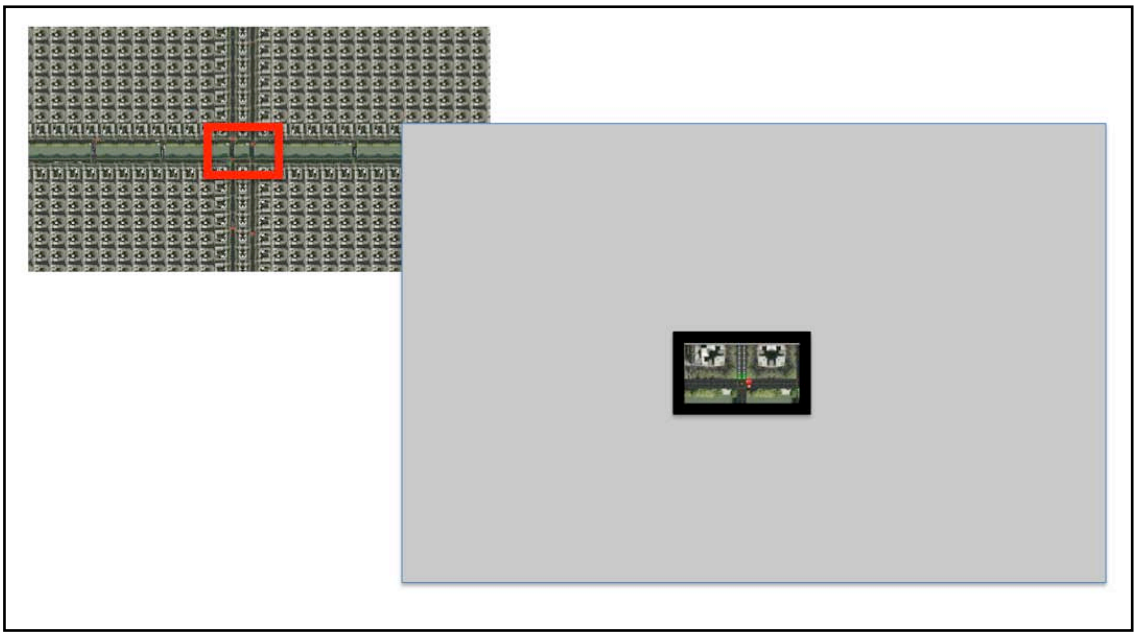


Figure 16 : Principe de la nouvelle méthode d'affichage

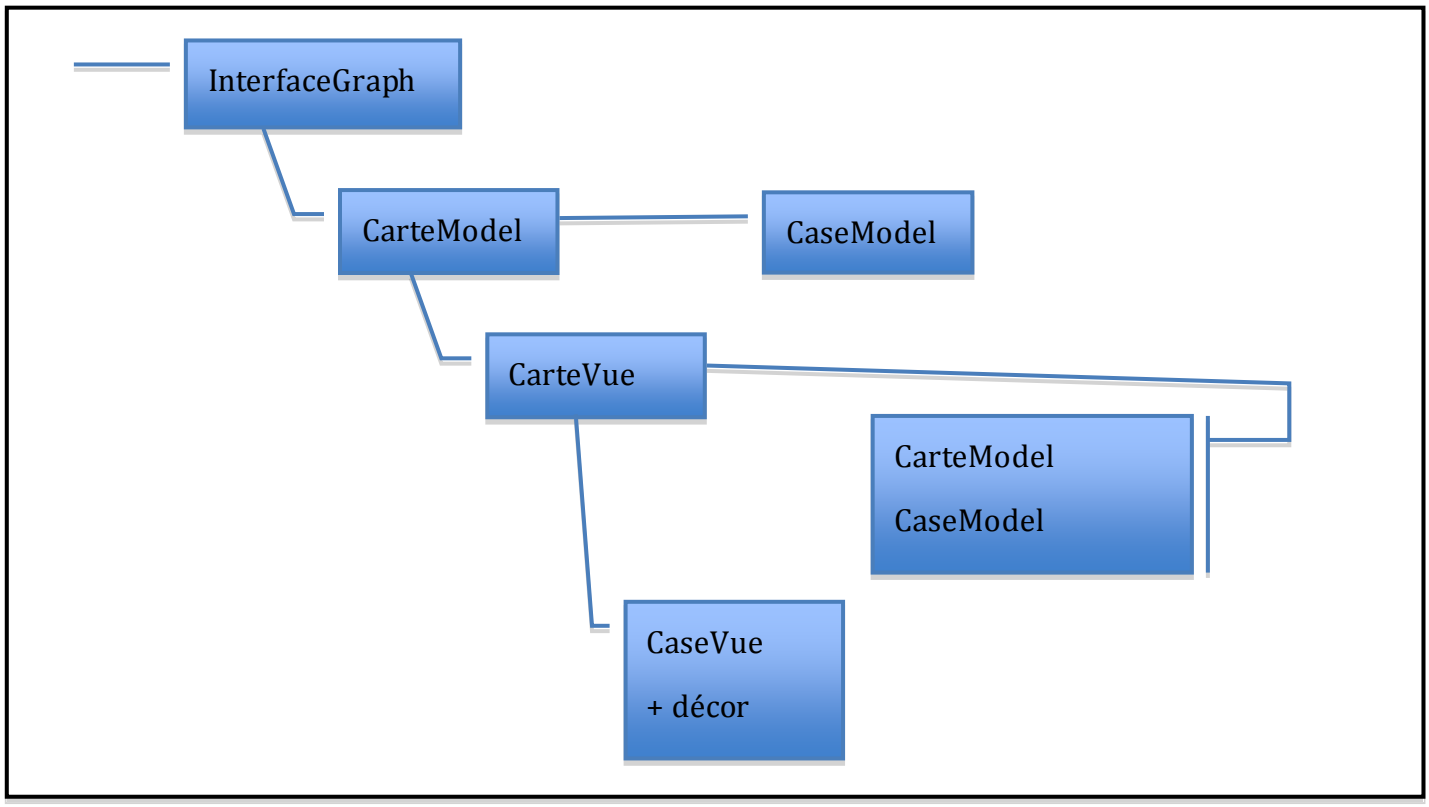


Figure 17 : Etapes successives de méthode d'affichage de la carte

7. Développement des autres objets contenus dans la deuxième fenêtre

1. Zone de bouton

La liaison avec la partie modèle n'est pas encore totale. Des boutons de l'interface Graphique doivent commander le déroulement de la simulation en amont grâce à l'API.

On ajoute donc un *JSlider* en bas de la fenêtre sous la carte qui enverra sa valeur au modèle : ce sera la vitesse de simulation. On crée aussi un objet *ZoneBoutton* de type *JPanel* qui va contenir tous les boutons de contrôle ainsi qu'un *JSlider* qui va commander le zoom (en plus donc de la molette de la souris).

Les boutons insérés, au nombre de 7, ont pour fonctions :

Le démarrage/la mise en pause de la Simulation,
L'arrêt de la simulation,
Le passage en mode pas-à-pas et un avancement du temps d'un pas
L'affichage/le masquage des statistiques temps réel dans le panneau correspondant (voir plus bas).
L'affichage/le masquage du panneau de rappel des paramètres initiaux (voir plus bas).
La réduction de la fenêtre en "mode réduit ", mode où la fenêtre n'affiche plus que les boutons de contrôle. Dans ce mode, la CarteModel reçoit bien les événements du modèle ou du *PanneauEssai*mais n'envoie aucun ordre

d'affichage. La vitesse de simulation est passée automatiquement au maximum (ordre envoyé au modèle). L'absence d'affichage libère un nombre important de ressources mémoires. Un autre appui replace la fenêtre en mode par défaut avec affichage de la carte (on recrée une carte grâce aux informations sauvegardées par *CarteModel*).

L'activation/désactivation simple de la carte. Ici, la fenêtre conserve sa taille mais la carte est remplacée par une photo dans le *JScrollPane*. De même, que précédemment, *CarteModel* n'envoie plus d'ordre d'affichage dans ce cas.

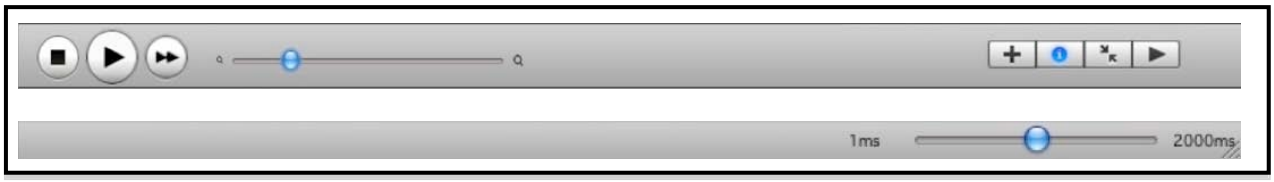


Figure 18 : Zone de bouttons

Dans la volonté de créer un logiciel davantage esthétique, nous n'avons, comme on le voit, pas utiliser les boutons JAVA Swing *JButton*. Les boutons sont créés de toutes pièces à l'aide de *JLabel*. On leur a ajouté un listener afin qu'ils reconnaissent l'appui de souris. Un bouton tel que le bouton *Stop* n'a que deux états : appuyé/relâché, donc il nécessite deux images correspondant à ses deux états (l'utilisateur doit voir visuellement qu'il appuie sur un bouton). Ainsi pour le bouton *Stop,* par exemple, les deux images sont :

Les boutons de ce type sont *Stop, PasAPas* et*FermeJusteCarte.*

Les autres boutons ont 4 états (en face de chaque fonction, l'image associée pour le bouton *Play/Pause*):

relâché en fonction1 : 

appuyé en fonction1 : 

relâché en fonction 2 : 

appuyé en fonction2 : 

Les boutons agissent sur le modèle comme on peut le voir. Or le modèle à ce moment du développement n'était pas encore connecté notre package. On a donc créé une classe *StubModel* qui représente le modèle dans notre package pour nos phases de test.

A l'appui sur un bouton, on appel une méthode de *StubModel* qui affichera dans la console l'ordre en question sous forme de texte ("System.out.println("Play")" par exemple). La connexion avec le modèle plus tard n'en sera que plus aisée car il suffira d'indiquer à ces méthodes regroupées dans *StubModel* la fonction à appeler dans le vrai modèle.

2. Ecran LCD

Dans l'optique de concevoir un logiciel assez complet, on ajoute quelques fonctions supplémentaires non demandées dans le cahier des charges. Ces fonctionnalités sont regroupées dans un nouvel objet *EcranLCD* (parce qu'il simule un écran LCD à l'écran).

Cet objet de type *JPanel* sera ajouté au conteneur *JPanelZoneBoutton*. Il ne contient que des *JLabel*(s) contenant ou du texte ou une image.

On retrouve ainsi :

Le nom du fichier de la simulation en cours,
Une icône en forme de loupe avec à sa droite le facteur de Zoom courant (la taille d'un coté d'une case en fait),
La vitesse de simulation affichée à droite de "Vitesse : ",
Le nombre de véhicules présents sur la carte à droite de l'icône en forme de voiture.
Une barre de défilement du temps créée par nos soins à l'aide de 3 *JLabel(s)* (le losange noir au fond, le rectangle noir autour et la "trainée" du losange) avec à gauche le temps en cours de simulation et à droite le temps restant.

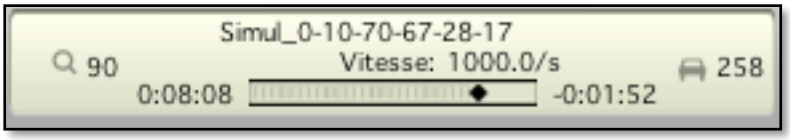


Figure 19 : Ecran LCD

L'idée nous vint ensuite d'utiliser les deux icones en forme de loupe et voiture comme des boutons (à l'image du logiciel *iTunes* ® de Macintosh). On a donc codé ces *JLabel*(s) comme des boutons à l'image de ceux de *ZoneBoutton*(boutons à 2 états ici seulement).

Un appui sur l'icône voiture va ainsi basculer l'écran LCD sur un nouvel affichage contenant toujours le facteur de zoom et le nombre de véhicules présents, mais le reste de la place est occupé par trois icones représentant une voiture, un bus et un tram. Le numéro affiché à côté représente le nombre de véhicule du type de l'icône présent sur la carte. La somme des trois nombres doit évidemment être égale au nombre total de véhicules sur la carte comme on le voit ci dessous.

Figure 19 : Fonction avancé de l'écran

Un appui sur l'icône en forme de loupe ouvre une nouvelle fenêtre que nous présentons dés maintenant.

3. Fenêtre de Zoom

A l'appui sur l'icône en forme de loupe de l'écran LCD, une petite fenêtre apparaît au dessus de la fenêtre principale. On l'appelle Fenêtre de Zoom. Elle représente en fait une image de la carte réduite (utilisation d'un *JLabel* encore) avec au dessus un rectangle bleu. Ce rectangle représente en fait à l'échelle de l'image de la carte la position ET la taille du *JViewport* par rapport à la carte totale. Ce rectangle change donc selon que l'on :

Modifie la taille du *JViewport,* il s'adapte à la forme et à la taille de celui-ci
Déplace le *JViewport* sur la carte, il se déplace sur l'image de celle ci à l'échelle,

Mais inversement, et là est tout l'intérêt de cette fenêtre, on peut directement déplacer le petit rectangle bleu dans la fenêtre et modifier la position du *JViewport* sur la carte : le système est "réflexif".
Pour parvenir à cela, on a utilisé la méthode vue au niveau du *JScrollPane* pour déplacer la carte en la faisant glisser avec la souris. On l'a transposée au rectangle bleu pour qu'il puisse de déplacer dans sa fenêtre.
Au niveau des mises a jour, dans la classe *InterfaceGraph* on envoie des mises à jour à cet objet *FenetreZoom* pour refléter les modifications possibles de l'affichage de la vraie carte. Dans la classe *FenetreZoom,* on envoie une mise à jour à *InterfaceGraph* pour qu'elle affiche le même endroit de la carte que *FenetreZoom.* On rappelle que tout cela est possible grâce aux listeners de souris (*MouseListeners).*

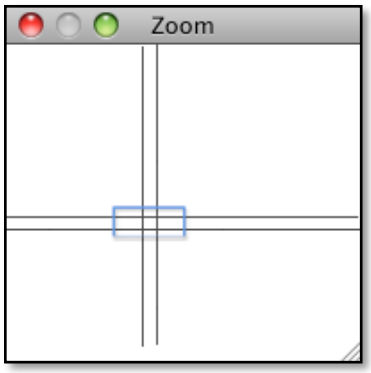


Figure 20 : Fenetre de Zoom

4. Rappel de paramètre

Dans la volonté de disposer rapidement à l'écran de toutes les informations utiles durant la simulation, on crée un nouvel objet *PanneauRappel* de type *JPanel*. Ce panneau va simplement afficher les paramètres initiaux choisis par l'utilisateur dans la première fenêtre. L'affichage/masquage de se panneau est commandée par ce bouton :  .

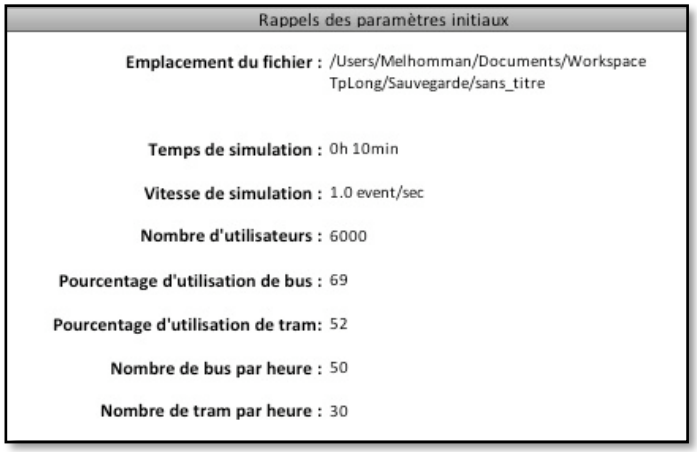


Figure 21 : Panneau Rappel

5. Affichage des statistiques temps réel

Dans le prolongement du dernier composant, on crée l'objet *AffichageStat* qui affichera plus tard les statistiques temps réel reçues de la partie *Statistiques* via le réseau. L'affichage/masquage de se panneau est commandé par ce bouton  .

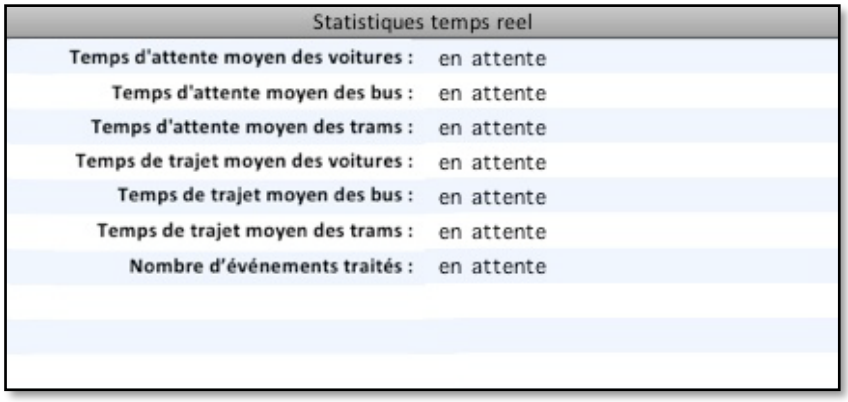


Figure 22: Affichage des statistiques temps réel

Enfin, on décide de placer ces deux panneaux dans un composant Swing *JSplitPane,* lui même placé dans un autre *JSplitPane* avec le *JScrollPane* contenant la carte.

Pour information, un *JSplitPane* permet d'afficher deux composants séparés verticalement ou horizontalement. La barre de division qui apparait entre les deux composants peut être déplacée. L'orientation du *JSplitPane* peut être :

JSplitPane.HORIZONTAL_SPLIT : les deux composants sont alignés horizontalement
jSplitPane.VERTICAL_SPLIT : les deux composants sont alignés verticalement.

Dans notre cas, les deux panneaux sont alignés verticalement dans le 1er *JSplitPane*, puis ce dernier et le *JScrollPane* sont ensuite alignés horizontalement dans le 2ème *JSplitPane.*

Les boutons d'affichage des deux panneaux vus plus haut commandent simplement le placement de l'une des barres de division des *JSplitPane*.

8. Développement de la dernière Fenêtre

Cette fenêtre est créée lorsque la simulation se finit toute seule ou si l'utilisateur appuie sur le bouton *Stop.* La fenêtre principale *InterfaceGraph* se ferme alors. A la création de cette fenêtre *FenetreResultat,* on envoie comme paramètre au constructeur le chemin du fichier de simulation, l'objet *StatCalculees* du package statistiques qui contient les résultats post Mortem, le nombre de véhicules, de voiture, de bus et de tram simulés finalement.

1. Affichage du texte utile

Afin de pouvoir avoir plus de choix d'affichage, on affiche tous les paramètres du constructeur (hormis le chemin de fichier) sous forme de *JLabel* plutôt qu'utiliser un *JTextPane*. En fonction du temps de trajet moyen des trams et des voitures, le logiciel détermine la réponse qu'il va renvoyer à l'utilisateur sous forme d'un texte coloré : "PROJET INTERESSANT!!!!!" ou "PROJET DANGEREUX!!!!".

2. Chargement des résultats des simulations précédentes

De la même façon que dans la fenêtre de démarrage, on charge le fichier de simulation dans lequel on récupère cette fois tous les paramètres chiffrés situés entre "FIN DE LA SAUVEGARDE DES PARAMETRES" et "FIN DU CHARGEMENT DES RESULTATS". On stocke dans 4 vecteurs les valeurs successives des paramètres. Les temps moyens de trajet des voitures, des bus et des trams ainsi que les charges successives des précédentes simulations contenues dans le fichier de sauvegarde sont ainsi récupérés. Pour finir, on ajoute à la fin de chacun des trois vecteurs représentant le temps de trajet d'un véhicule la nouvelle valeur obtenue par la dernière simulation. Le nombre d'éléments contenus dans chaque vecteur est ainsi à présent identique (le vecteur charge contenait jusqu'alors un élément en plus).

On prend soin aussi dans le téléchargement de sauvegarder toutes les chaines de caractères situées avant "FIN DE LA SAUVEGARDE DES PARAMETRES".

3. Enregistrement des résultats

On réécrit ensuite dans ce même fichier en écrasant l'ancien texte. On ajoute d'abord toute la chaine de caractère sauvegardée pour que la fenêtre de démarrage retrouve bien tous ces paramètres initiaux. Ensuite, on ajoute les paramètres sauvegardés dans les vecteurs en respectant la mise en page des fichiers de sauvegarde(séparer les valeurs par un caractère "/").

4. Affichage du Graphique

L'enregistrement effectué, il ne reste qu'à dessiner le graphique. On a dit que le graphique devait montrer l'évolution, au cours des simulations, des temps de trajet des 3 types de véhicule en fonction de la charge. On choisit donc une couleur par type et on va chercher à afficher les trois courbes superposées la charge en abscisse, le temps en ordonnée. Pour cela on crée deux nouveaux objets : l'objet *Trait* qui n'est rien qu'un tableau de 4 points représentant donc un segment ou trait, et enfin l'objet *Graphik*.

Cet objet de type *JPanel*appelle la méthode *paint()*surchargée par nos soins. Cette méthode dessine au sein du composant des segments d'une couleur choisie. L'ensemble des traits à dessiner est contenu dans un vecteur de *Trait*nommé *form. Paint()*lit les éléments du vecteur *form*et les dessine un à un. Une autre méthode de la classe *Graphik* permet d'ajouter des points au vecteur*form* en donnant en paramètres 4 entiers et une couleur, celle de la future courbe. La méthode *repaint()* permet enfin de redessiner le composant et donc de mettre à jour de nouveaux points.

Il nous suffit ensuite d'ajouter trois instances de ce composant de type *JPanel* à la fenêtre de résultats. En les superposant et en ajoutant une image de fond (avec des axes x et y), on obtient le résultat voulu. Pour dessiner les courbes, il suffit ensuite de remplir le vecteur *form* de chaque Graphik avec les points (Charges, temps de trajet moyen du véhicule du type affiché). Ces points sont la concaténation des valeurs du vecteur contenant les charges et de l'un des autres vecteurs. Auparavant, il faut tout de même remettre ses valeurs par charges croissantes si l'on veut que les courbes soient compréhensibles. On met donc en place ce tri et on obtient finalement une fenêtre de résultat tel que ci-dessous :

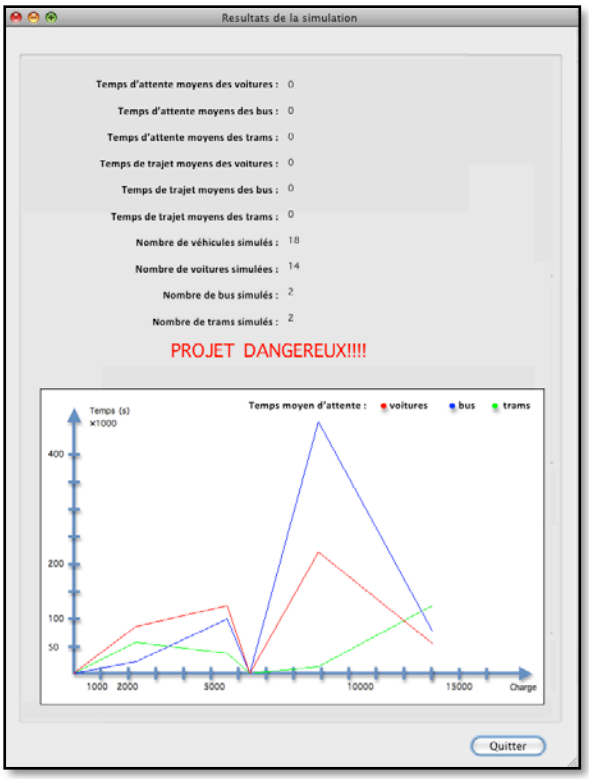


Figure 23 : La dernière fenêtre

9. Mise en Place du réseau

A ce stade, l'interface Graphique est finie est testable à l'aide du *PanneauEssai.* Lorsqu'on veut la connecter aux autres packages via le réseau, on enlève le *PanneauEssai* de la fenêtre principale et on implante la méthode *update*(). Selon le type de mise à jour reçu, ou on modifie l'affichage de la carte avec les informations reçues du *modèle* et on incrémente le temps, ou on met à jour la fenêtre *AffichageStat* avec les valeurs reçues du package *statistiques*.

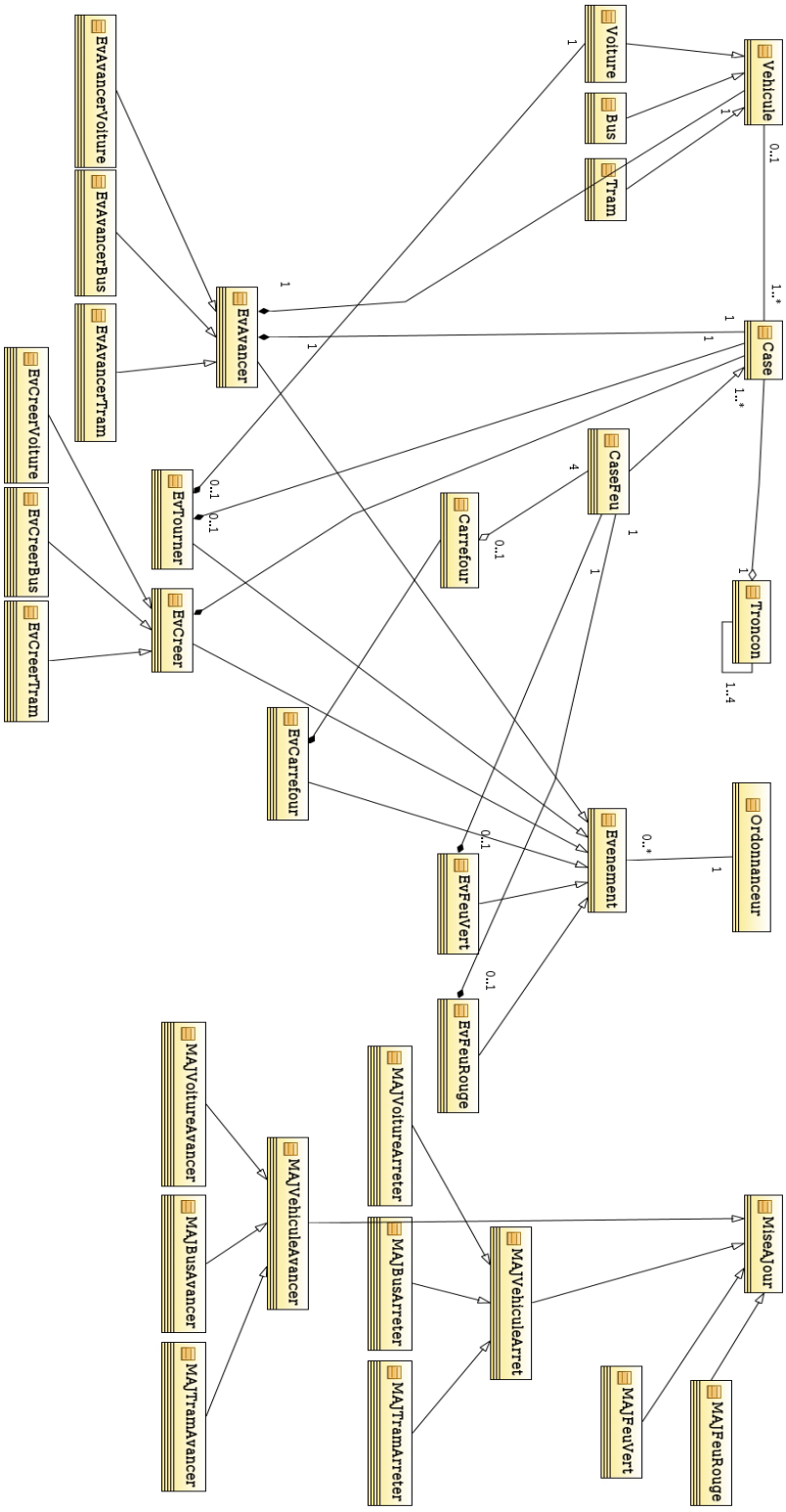
Dans la classe *Stubmodel*, on implémente les méthodes de l'API qu'il faut pour pouvoir envoyer les paramètres initiaux au modèle et le contrôler durant la simulation.

VI. CONCLUSION

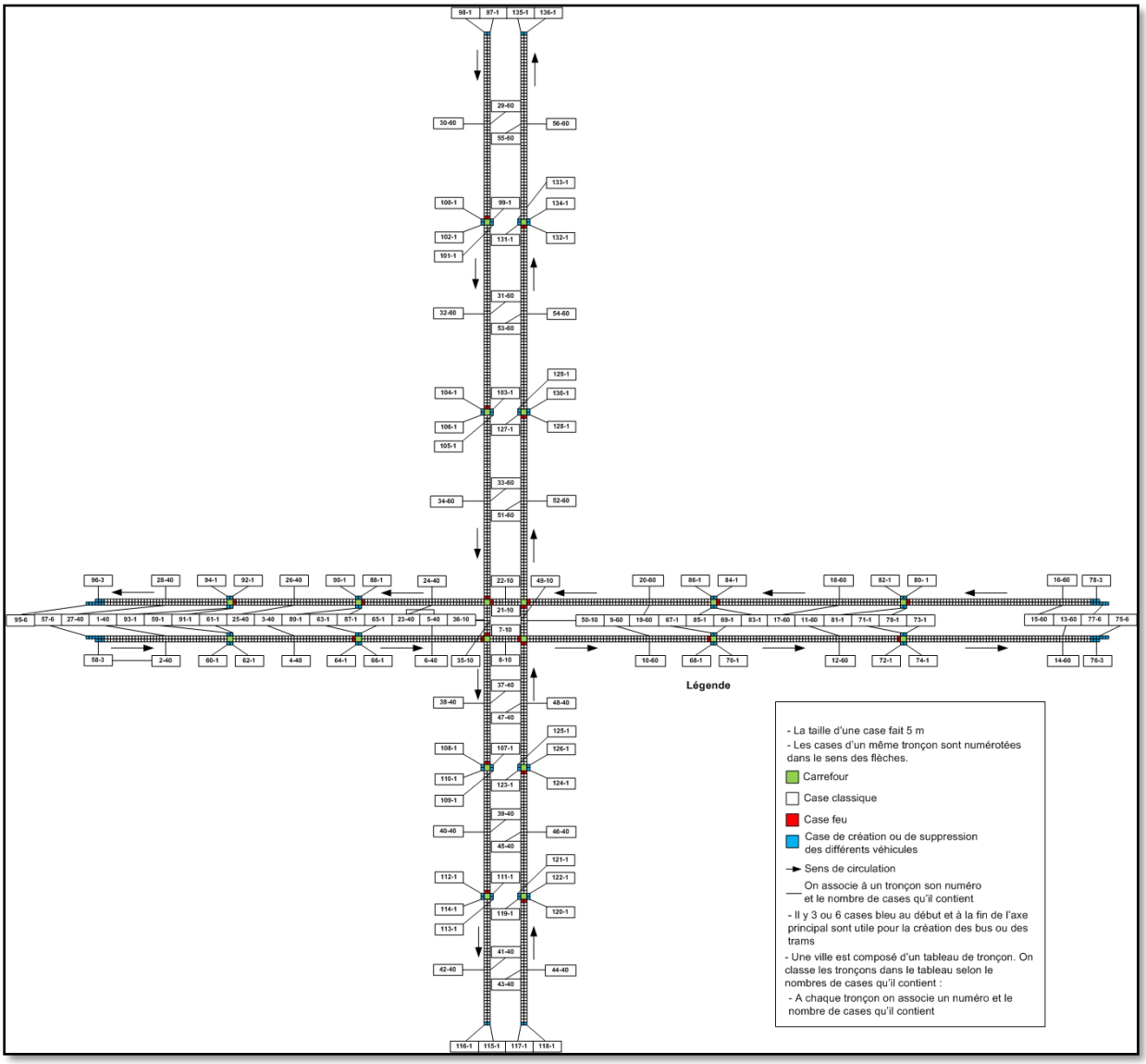
Nous avons eu la chance de former un groupe de neuf personnes intéressées par le projet, et capables de travailler ensemble en passant par dessus les différents que nous avions pu rencontrer. Sans cet esprit d'équipe, il aurait certainement été impossible d'aboutir au même résultat. Chacun peut être fier de sa contribution à ce TP peu ordinaire.

Ce projet aura été l'occasion de nous confronter à une situation quasiment professionnelle, et est indéniablement une expérience positive pour le travail d'équipe que nous allons réaliser dans quelques semaines. Ce TP a aussi été riche en enseignements personnels, autant sur le plan technique, que sur les attentes de notre vie professionnelle à venir.

VII. ANNEXES



le diagramme des classes réalisées en UML de l'ensemble du modèle



Carte initiale utilisé lors du développement