LOG2810

Structures discrètes

Mohamed Salah Eddine 1805984

Marc El Khoury 1798468

Alexandre Clark 1803508

Polytechnique Montréal

2016-12-02

**1 INTRODUCTION**

Dans le but de répondre à un appel d’offre d’une entreprise, nous avons été chargé de développer une application permettant de mettre en place un service d’autopartage de véhicules autonomes dans la ville de Montréal. L’autopartage est un service donnant la possibilité aux citoyens de partager des véhicules pour se déplacer en ville, plutôt que d’avoir à acheter une voiture qui serait inutilisée pendant la majorité du temps. Les voitures utilisées dans le service seraient entièrement autonomes et seraient réparties dans plusieurs zones découpées par le conseil d’administration. En tant que programmeur, notre rôle était d’implémenter une interface permettant de simuler l’utilisation de ces véhicules dans un territoire donné. Pour réussir à modéliser le problème, il est très utile d’utiliser les concepts appris dans le cours de *Structures discrètes* tel que les automates.

**2 EXPLICATION DE LA SOLUTION**

La solution que nous proposons se base sous les principes de la programmation orientée objet. Nous avons donc créé des classes permettant d’implémenter les différentes fonctions servant à obtenir une simulation adéquate.

Premièrement, il y a la classe **Node** qui s’interprète comme un état d’une machine à état. Chaque **Node** aura donc un parent, un enfant et possèdera un nom propre à lui. Chaque état (Node) est un caractère *char* dans notre cas.

Deuxièmement, il y a la classe **Quartier,** cette classe possède l’implémentation de l’automate en utilisant la classe Node cité ci-dessus. Ainsi, chaque quartier était un automate en soit. Selon notre compréhension du problème, nous avons déterminé que chaque quartier était représenté par les 3 premières lettres du code postal. Ce qui nous mène au fait que chaque quartier était un automate en soit. Chaque quartier contenait un vecteur d’adresses appartenant à celui-ci.

Troisièmement, la classe **Adresse** est représentée par les 3 derniers caractères d’un code postal. Chaque adresse appartient donc à un quartier d’une zone donnée. Éventuellement, les utilisateurs partiront d’une adresse de départ et arriveront à une autre adresse. C’est pourquoi, on construit chaque adresse avec un nom, un quartier et une zone afin de bien se retrouver dans le code et mieux implémenter nos fonctions.

Quatrièmement, la classe **Véhicule** implémentera les véhicules de notre simulation. En effet, chaque véhicule appartient à une zone. Donc on construit toujours un véhicule en lui attitrant une zone donnée. Cette classe détermine aussi, par un booléen, si un véhicule est occupé ou non. De plus, on lui attitre un utilisateur et une position d’adresse. Ainsi, chaque véhicule est situé à une adresse donnée dans un la zone donné. Chaque véhicule a un utilisateur, qui lorsque celui-ci utilise le véhicule, le véhicule se met en mode occupé.

Cinquièmement, la classe **Utilisateur** définit chaque utilisateur en lui donnant une adresse d’origine, qui correspond à l’adresse de départ ainsi que l’adresse d’arrivée. Les utilisateurs sont ajoutés à des groupes par la suite. De plus, comme cité plus haut, chaque utilisateur, **dans la mesure du possible**, possède une voiture. Autrement dit, il se pourrait que le nombre de voiture disponible dans la zone ne soit pas assez pour servir tous les utilisateurs de cette zone. Un équilibrage devra donc être fait (nous expliquerons notre façon de faire un peu plus tard.)

Sixièmement, la classe **Zone** représente les différents emplacements où les véhicules seront dispersés. Chaque zone aura ainsi le tiers du nombre de véhicule total et possèdera un lexique contenant toutes les adresses qu’elle englobe. Il faudra après chaque groupe d’utilisateurs équilibrer la flotte pour ainsi diminuer les chances qu’une zone n’ait plus assez de véhicule pour servir un client.

Finalement, la classe **Carte** représente le territoire entier couvert par l’application ; il est composé de tous les zones et connait les différents groupes de clients, connait l’emplacement des véhicules et possède les fichiers pour créer les lexiques. C’est l’objet qui va gérer la flotte, c’est-à-dire l’initialiser et l’équilibrer si nécessaire.

**2.1 CreerLexiques**

La fonction créer lexique a pour but de lire les fichiers textes d’un répertoire donné correspondant aux zones données. Avec cette fonction, nous avons créé un automate qui reconnaissant un quartier.

L’automate fonctionnait de la façon suivante : la première lettre correspondait à l’état initial de l’automate, elle possède un nom et a envoyait vers un enfant qui représentait l’état suivant (le deuxième caractère dans ce cas-ci). Le deuxième état avait aussi un enfant qui était l’état final de notre automate et qui correspondait à son tour au dernier caractère d’un quartier.

Une fois le premier automate construit, nous l’ajoutions dans un vecteur de quartier qui allait nous permettre de détecter si les adresses suivantes étaient appartenues au même quartier. Le cas échéant, un autre automate est créé de la même façon que le précédent.

Chaque fichier texte était une zone en soit. Une fois la lecture des fichiers terminée, nous obtenions un lexique ayant autant de zones qu’il y a de fichiers textes. Ce lexique était un vecteur de quartier. Et chaque vecteur de quartier contenait un vecteur de pointeur d’adresses.

**2.2 EquilibrerFlotte**

La fonction équilibrer flotte est appelée après chaque simulation. Autrement dit, lorsqu’un service est donné à un groupe, on s’attend forcément à ce que certaines zones aient moins de véhicule que d’autres. Il se pourrait aussi que des zones puissent être complètement désertées.

EquilibrerFlotte remet donc toutes les voitures dans une zone tampon dans la carte. Pour ce faire, on (pop\_back()) les véhicules qui sont attitrés à une zone. Par la suite, on prend le nombre total de véhicules disponible et on le divise par le nombre de zone. Le reste des véhicules est aussi distribué un à un sur chaque zone jusqu`à épuisement du nombre. Chaque déplacement fait par le véhicule est donc compté pour donner le nombre total de déplacement du véhicule sans aucun usager à bord.

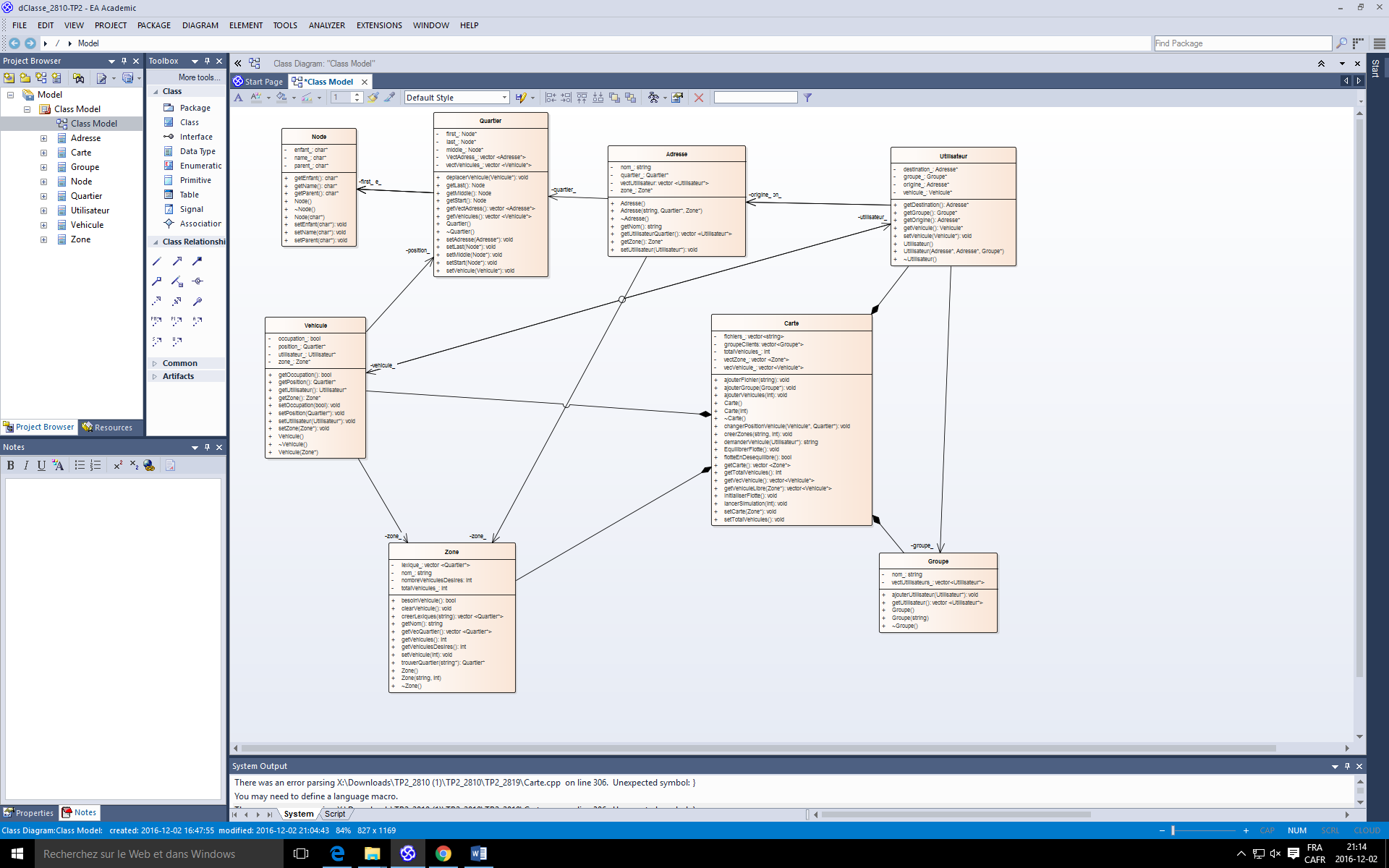
**2.3 LancerSimulation**

En ce qui concerne la fonction LancerSimulation, elle permettait de mettre en œuvre tout le travail effectué jusque-là. Pour chaque groupe d’utilisateur entré, la fonction donnait le résultat obtenu quant à la réussite ou l’échec du service pour chaque client du groupe. Ainsi, lorsqu’un client ne reçoit pas son service, un message est affiché à l’écran indiquant un manque de véhicule pour la zone donnée.

**2.4 Interface**

Pour notre interface nous avons décidé de faire une boucle for infini qui prendra le paramètre de l’utilisateur, si l`entrée est différente de `c` on exécute des fonctions, sinon on arrête le programme.

L’option `a` permet de créer le lexique avec le répertoire des fichiers textes entré par l’utilisateur. Par la suite l’option b permet de créer le groupe d’utilisateur et les utilisateur en entrant chaque paramètre utilisé par la simulation soit : l’adresse de départ ainsi que l’adresse d’arrivé. L’option c permet de lancer la simulation telle que décrite dans le point 2.3.

**2.5 Diagramme de classe**

**3 DIFFICULTÉS**

Une des difficultés rencontrés dans ce TP est une bonne organisation de l’ensemble de nos classes. Par exemple, notre façon de structurer le tout et l’organisation de nos pointeurs de manières à ce que le tout soit bien cohésif sans être trop compliqué. On aurait pu aussi programmer le TP en java par exemple.

De plus, une bonne compréhension de l’objectif du sujet était une grande partie des difficultés rencontrés car quelques points portent à confusion dans l’énoncé du Tp. Par exemple : lorsque les véhicules se déplacent pendant l’équilibrage, il fallait compter le nombre de déplacements. Bref, ce n’était pas très clair.

Somme toute, comme la programmation en C++ nécessite une bonne gestion des pointeurs, une bonne partie de notre temps a été passée sur le débogage de ces derniers. Tout changement nécessaire, devenait un problème, car il fallait réarranger toute la structure du système. Exemple : devrions nous ajouter un vecteur de véhicules dans la classe zone ET dans la classe quartier ?

Notez bien, il est nécessaire d’entrer les bons caractères pendant la simulation, sinon cela risque de faire boguer le système.

**4 CONCLUSION**

Pour conclure, ce TP est un travail qui nous a montré nos points faibles afin de pouvoir les améliorer au futur. Consacrer un temps plus important pour organiser et structurer nos idées est trivial. Nos attentes pour les prochains est une charge de travail plus petite surtout quand c’est une fin de session, et une clarification du sujet plus importante.