

LOG2810

STRUCTURES DISCRÈTES

**TP2 : AUTOMATES – MACHINES À ÉTATS**

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

VINCENT FORTIN (1791999)

DOMINIQUE PICHÉ (1766981)

IBRAHIMA SÉGA SANGARÉ (1788085)

REMIS LE VENDREDI 2 DÉCEMBRE 2016

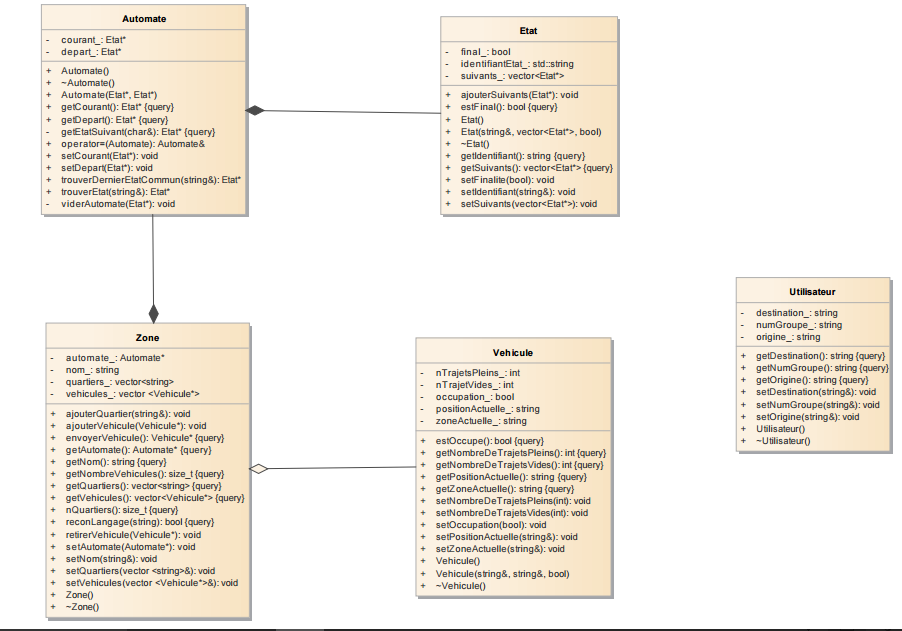
**Introduction :**

Alors qu’il est assez coûteux d’avoir une voiture à soi-même, que ce soit à cause du prix d’achat, des frais de réparation, des frais d’entretien et du prix de l’essence, un bon nombre d’utilisateurs cherchent une solution alternative plus abordable. Une de ces solutions étant prévue d’arriver dans un futur proche est l’autopartage à l’aide de la voiture autonome, tel qu’il est déjà possible de l’observer chez Google. Toutefois, afin d’étudier la viabilité du système il nous est demander de concevoir un système permettant de simuler le tout. L’autopartage consistant en le fait que les usagers n’ont pas de véhicule qui leur soit propre, mais à l’aide d’une application ils sont capable de demander l’accès à un véhicule qui les portera vers une destination choisie.

Toutefois, le nombre de véhicules étant limité, il est de notre devoir de concevoir un système qui attribue à chaque client possible un véhicule le plus près de celui-ci afin de l’apporter à destination. Par la suite, les zones surpeuplées en voitures doivent envoyer les voitures en excès vers les zones moins peuplés pour être sûr qu’il y a toujours un véhicule disponible près d’un client. Le tout sera donc implémenté à l’aide d’automate représentant les quartiers où, par exemple, les options H1A et H1B seraient représentées par un automate où H est lié à 1 et 1 permet d’aller soit à A ou à B.

**2. Présentation des travaux :**

Lors de l’élaboration de l’application, nous avons procédé au codage en utilisant une approche orientée objet. Cela a rendu la tâche plus simple pour mettre en évidence les fonctionnalités à implémenter. La solution contient donc les quatre classes suivantes : Automate, Etat, Utilisateur, Vehicule et Zone. Le diagramme de classes complet se présente comme suit.



**Fig. 1 :** *Diagramme de classes complet de la solution*

*Avec Enterprise Architect*

**2.1 Automate :**

La classe automate permet de représenter chaque quartier à l’aide d’un automate, soit un type d’arbre donc les nœuds représentent les différentes lettres du code ZIP et par le fait même, la position d’un quarter. La classe automate contient les méthodes habituelles telles que les constructeurs, le destructeur, les setters et les getters qui sont reliées à l’état principal et prochain de l’automate. Les attributs privés sont l’état présent, l’état suivante et le vecteur d’ensemble d’états.

**2.2 Etat :**

Directement reliée à la classe automate, la classe état représente la position d’une voiture dans un automate qui lui représente un quartier. Elle contient les méthodes de getters et setters habituelles. Les attributs privés représentent l’état présent, le vecteur d’état suivants et l’identifiant de l’état.

**2.3 Utilisateur :**

La classe utilisateur représente un usager qui désire utiliser le service d’autopartage. Elle contient un destructeur, un constructeur par défaut et un par paramètre ainsi que les getters et les setters. Ses attributs privés sont le quartier d’origine, le quartier de destination et le numéro de groupe. La valeur de tous ces attributs étant directement prise des entrées de l’utilisateur de l’application.

**2.4 Vehicule :**

La classe vehicule représente évidemment un véhicule et contient les méthodes habituelles en plus d’une méthode qui retourne un booléen représentant si le véhicule est occupé ou non. Ses attributs privés sont sa zone actuelle, sa position actuelle, son occupation, le nombre de trajets plein et le nombre de trajets vides.

**2.5 Zone :**

À part les méthodes habituelles, la classe zone contient la méthode *envoyerVehicule()* qui permet d’assigner les véhicules qui ne sont pas présentement occupés. Elle contient aussi la surcharge de l’opérateur ‘<’ qui permet de déterminer facilement quelle zone contient le plus de véhicules et donc quelle zone devrait se débarrasser de véhicules afin d’équilibrer la quantité de véhicules à travers les zones. Ses attributs privés sont un pointeur de l’automate, le nom de la zone, un vecteur de pointeurs de véhicules et un vecteur des quartiers.

**2.6 Fonction principale (main) :**

C’est là que l’important de l’application se déroule. Le main contient à la fois le menu, ainsi que les trois fonctions majeurs : *creerLexiques()*, *equilibrerZones()* et *lancerSimulation()*. Par défaut l’usager voit le menu et celui-ci lui propose quatre options différentes. L’option (a) permet à l’usager d’inscrire le chemin absolu du dossier où se trouvent les fichiers texte qui représentent les zones. Le programme prend alors tous les fichiers dans ledit dossier et garde en mémoire chaque quartier ainsi que la zone qui y est associée. L’option (b) n’est disponible que si l’usager a effectué une entrée valide lors de l’option (a) et elle permet à l’usager d’entrer le nombre de clients désirés ainsi que le quartier de départ, le quartier d’arriver et le numéro de groupe pour chaque client. Par la suite, le programme demande à l’usager le nombre de véhicule désiré ainsi que la zone pour chaque véhicule. L’option (c) dépend de bonnes entrées de l’option (b) et permet de lancer la simulation, affichant alors deux tableaux. Un premier tableau illustrant le nombre de trajets avec clients pour chaque véhicule et le nombre de trajets vides pour chaque véhicule, puis le deuxième tableau illustrant le nombre de véhicules par zone, au début et à la fin de la simulation. L’option (d) peut être effectuée avant ou après n’importe quelle des trois autres options et permet à l’usager de quitter le programme.

**2.6.1 Fonctions**

**- creerLexiques** : reçoit le chemin du dossier contenant les fichiers texte représentant les zones en paramètre et créée les automates à partir de cette information. Le chemin doit être écrit de façon absolue, il ne doit pas y avoir d’espace (sinon le cin arrête la lecture de l’entrée trop tôt) et doit se terminer par une barre oblique pour que le programme n’ait qu’à ajouter le nom de chaque fichier au nom du chemin pour les lire.

**- equilibrerZones** : permet de comparer des zones entre elles grâce à l’opérateur ‘<’ surchargé et envoie alors des véhicules des zones où il y en a trop à des zones où il y en a moins, afin d’équilibrer les zones.

**- lancerSimulation** : débute une simulation dans laquelle aucune notion temporelle n’est prise en considération. Tous les membres du premier groupe réalisent leur requête, on effectue un équilibrage des zones, les membres du deuxième groupe font leur requête puis on équilibre et ainsi de suite jusqu’à ce les requêtes de chacun des groupes soient effectuées.

**3. Difficultés rencontrées :**

Réussir à lire tous les fichiers dans un dossier fut un gros problème pour notre équipe. Nous n’avions jamais réalisé cette opération avant en C++ et avons donc rechercher longtemps afin de trouver comment la réaliser, en plus en essayant de faire que ça fonctionne sur Linux en plus de Windows. Nous avons finalement réussi à l’aide du chemin absolu, mais là encore on demande d’écrire le chemin de façon spécifique, car sinon nous rencontrons d’autres problèmes que nous n’avons pas pu réglé. Par exemple, afin de libre un espace dans le nom du chemin absolu nous avons essayé d’utiliser *noskipws*, mais malheureusement cela fait que le programme affiche successivement le menu à l’infini ce qui est très indésirable et nous est arrivé souvent alors que nous tentions d’implémenter autre chose dans le menu. Ces problèmes étant surtout dû au fait que l’on travaille constamment à l’intérieur d’un switch case.

**Conclusion :**

En conclusion, ce TP nous a permis d’utiliser nos connaissances apprises en structures discrètes à propos des langages pour créer une application qui simule un système d’autopartage. Nous avons donc une fois de plus, pu améliorer nos connaissances en orienté objet à l’aide de l’IDE Visual Studio en utilisant le langage C++.