IS1220 – Final Project

MyVelib – a bike sharing system

**RAPPORT**

Félix Hubert & Tom Dupuis

Sommaire

1. Introduction
2. Contexte et compréhension de l’énoncé
3. Architecture et algorithmes
   1. Principaux design patterns
      1. Factory pattern
      2. Observer pattern
      3. Strategy pattern
   2. Architecture principale « core »
      1. Bicycle
      2. Card
      3. User
      4. Station
      5. Ride
      6. MyVelib
      7. PathFinder
      8. MyVelib
   3. CLUI
      1. Système Switch/Case
      2. Exceptions
4. Scénarios et tests
5. Répartition des tâches

**1.Introduction**

Ce projet porte sur la simulation exhaustive d’un système de vélo en libre-service, comme le système Vélib’ à Paris. Les objets qui devait apparaitre au cœur de notre implémentation étaient les **stations**, les **usagers** et les **vélos**. La partie la plus difficile du projet a été pour nous d’implémenter les relations entre les usagers User, les stations Station et la planification de trajet qui proposait aux usagers des itinéraires Ride. Il a été complexe de trouver le design pattern observer/observable qui nous a permis de résoudre le problème et de le coder. Nous avons beaucoup appris sur l’utilité des design patterns et l’utilisation de JUnit pour avancer sereinement dans la programmation.

**2.Contexte et compréhension de l’énoncé**

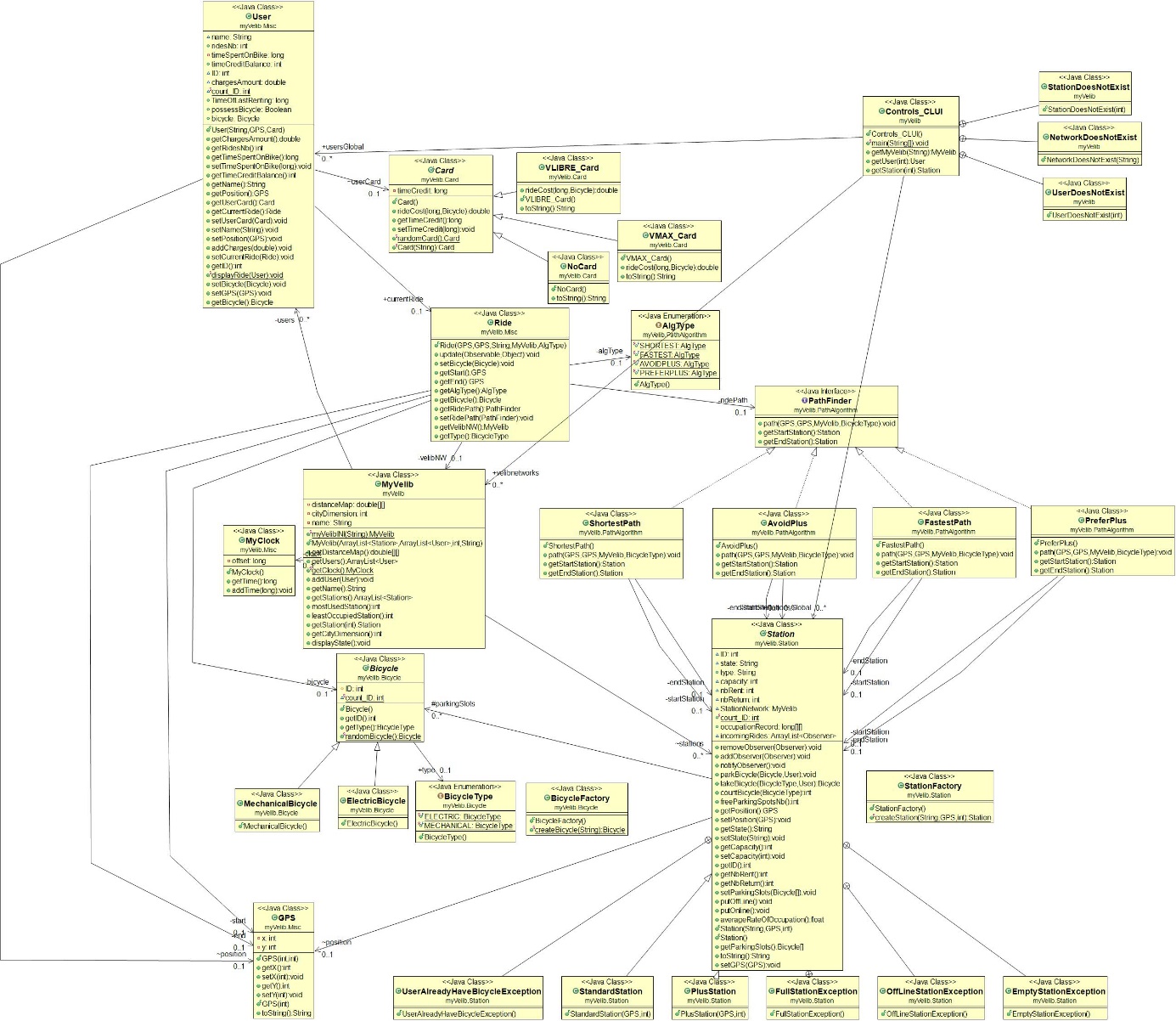
Dans le cas d’un système type Vélib’, les stations sont constituées de plusieurs places. Celles-ci peuvent être vides, contenir un vélo ou être hors-service. Les stations elles-mêmes peuvent donc se retrouver sans aucune place libre ou sans aucun vélo à fournir. L’usager ne peut donc pas prendre son vélo et le reposer à n’importe quelle station, il faut y faire attention.

Les préférences de l’usager en termes de vélo, la carte qu’il possède, sa position dans la ville, tout cela influe sur son choix optimal de station de départ et d’arrivé lors d’un voyage. Ce choix difficile peut être fait par des programmes de planification de trajet qui simplifie ainsi la vie des citadins au quotidien. Nous en avons implémenté.

Pour améliorer le futur ces installations, des études statistiques sont indispensables. Ils faut pouvoir identifier les station les plus et les moins sollicitées pour fluidifier le débit de vélo et augmenter le rendement du système.

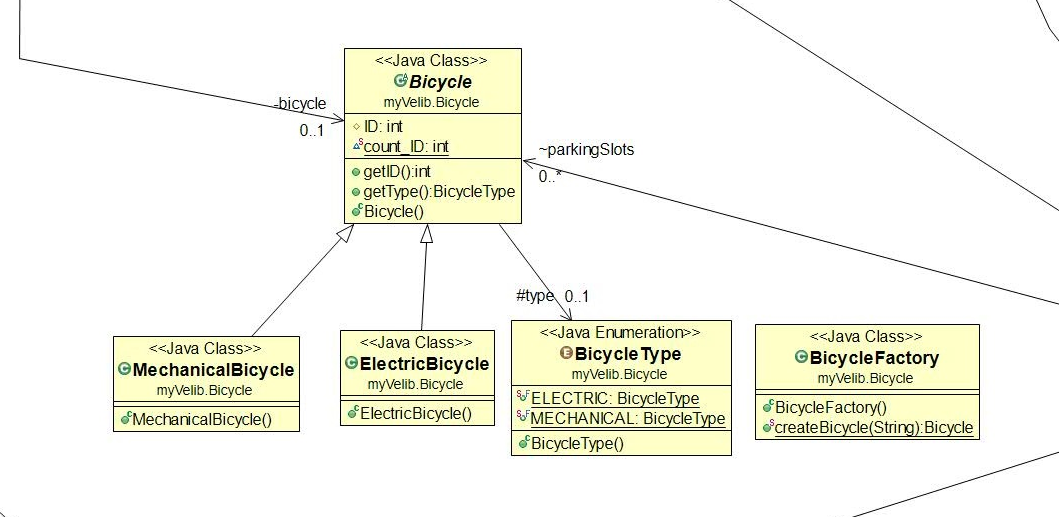
**3. Architecture et algorithmes**

Le diagramme de classe UML de notre projet est le suivant :



Comme nous pouvons le voir, la complexité de ce diagramme nécessite une description et analysé détaillée de chacune des parties le composant.

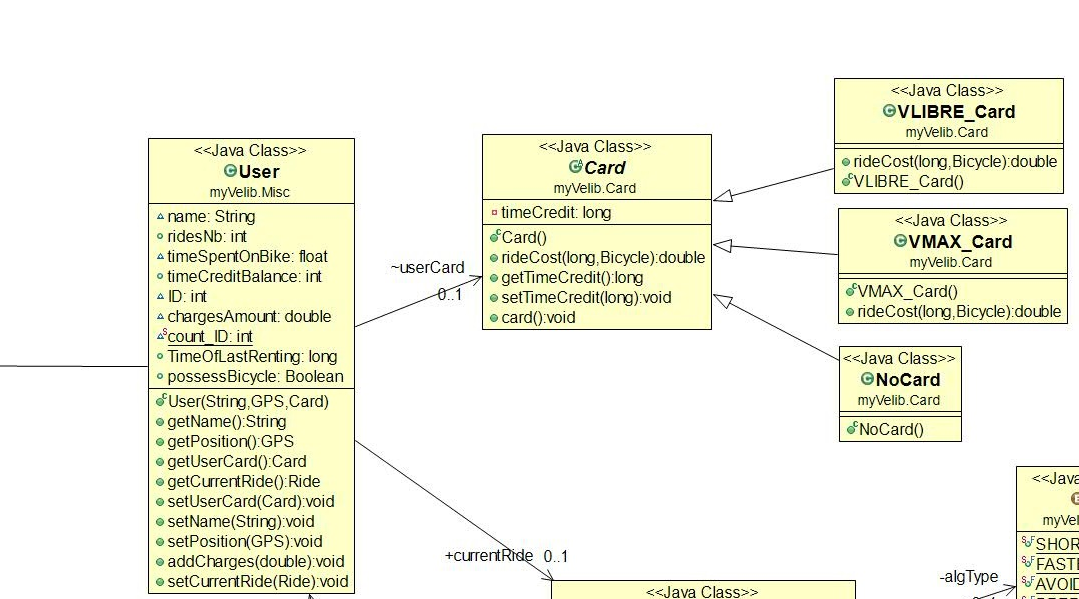
**a. Principaux design patterns**

****

Tout d’abord, nous avons géré la gestion d’un ensemble de vélos par un design de factory. Bicycle est ici une abstract class détenant l’ensemble des méthode inhérentes à un vélo ainsi que ces attributs. MechanicalBicycle et ElectricalBicycle sont les implémentations réelles de vélo particuliers.

Afin de générer un vélo, on utilise donc la BicycleFactory.

Nous avons choisi ce modèle en vu de pouvoir ajouter facilement d’autres type de vélo, et car il se pretait bien à un cas où l’on considère plusieurs sous items qui doivent être créés (sorte de marchandise).



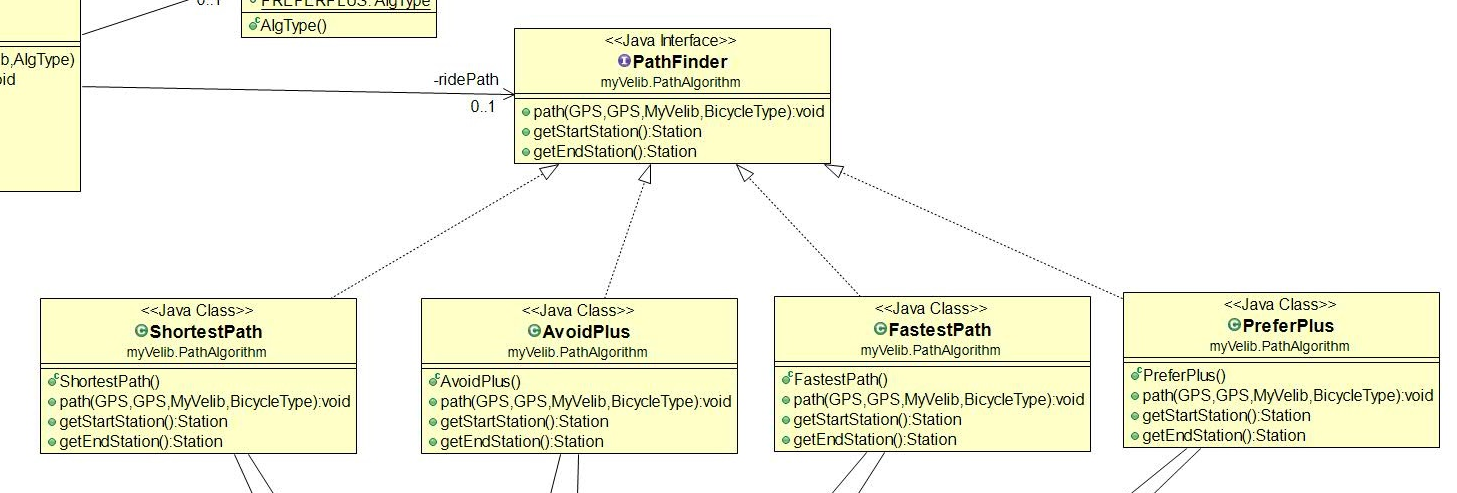
Vient ensuite les cartes utilisateur. Nous avons ici aussi opté pour définir une abstract class Card et ensuite définir des sous-classes réelles selon le type de Card, afin de faciliter leur implémentation.

Il s’est avéré également judicieux de définir une classe NoCard afin de traiter les utilisateurs qui n’en auraient pas, plutôt que de mettre null dans l’attribut concerné, afin de pouvoir réaliser les même opérations sur l’User.

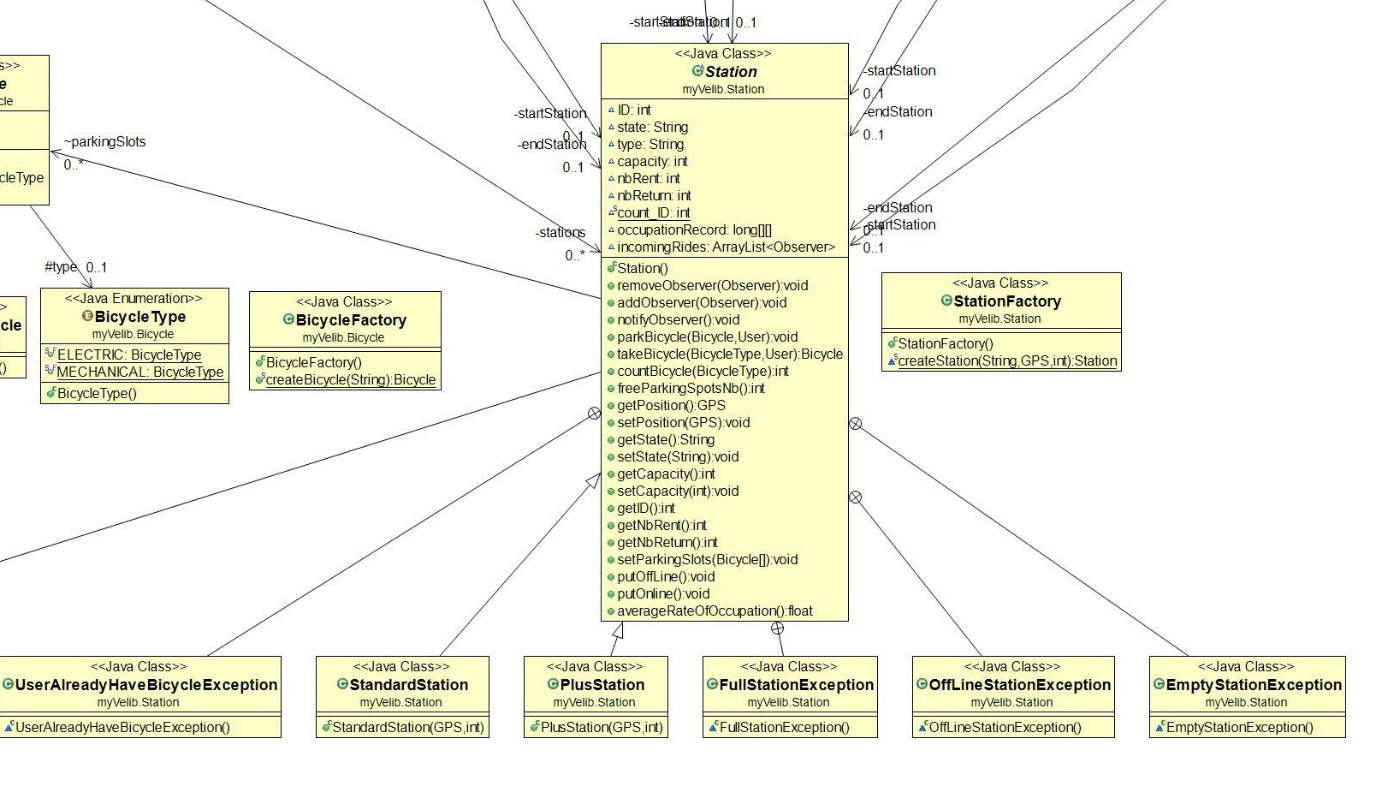
Autre pattern remarquable, nous avons utilisé le stragey pattern pour pouvoir gérer les différents algorithmes demandés pour trouver la station de départ et d’arrivée en fonction des coordonnées gps du départ et de destination.

Nous avons ainsi créé l’interface PathFinder avec l’abstract method void path. Cette méthode calcul les deux stations et les place dans les attributs de l’instance path.

Puis les 4 différents algorithmes sont simplement des classes implémentant cette interface , et en rajoutant l’algorithme correspondant dans chaque méthode path.



Finalement, les stations sont implémentées aussi grâce au factory pattern : on a une StationFactory , ainsi qu’une abstract Station, et 2 sous classes réelles PlusStation et StandardStation qui l’étendent.

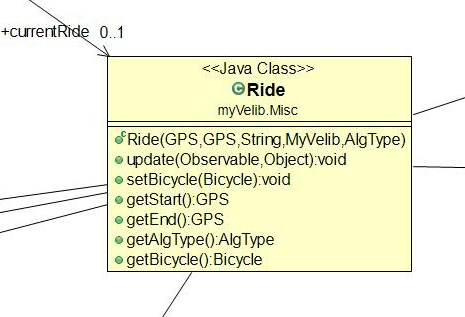


Afin de prendre en compte le fait qu’une station peut se remplir ou devenir offline au cours du trajet et donc avertir l’utilisateur en changeant la station d’arrivée, nous avons utlisé l’observer pattern.

L’observable était donc la station d’arrivée et l’observer la classe Ride qui contient les informations importantes du trajet, et qui exécute les méthodes path des classes associées.

Ainsi, si une station devient inaccessible, elle notifie Ride qui va alors update la station d’arrivée en appelant de nouveau la méthode path de son attribut PathFinder, puis en récupérant alors la station d’arrivée contenue dedans.

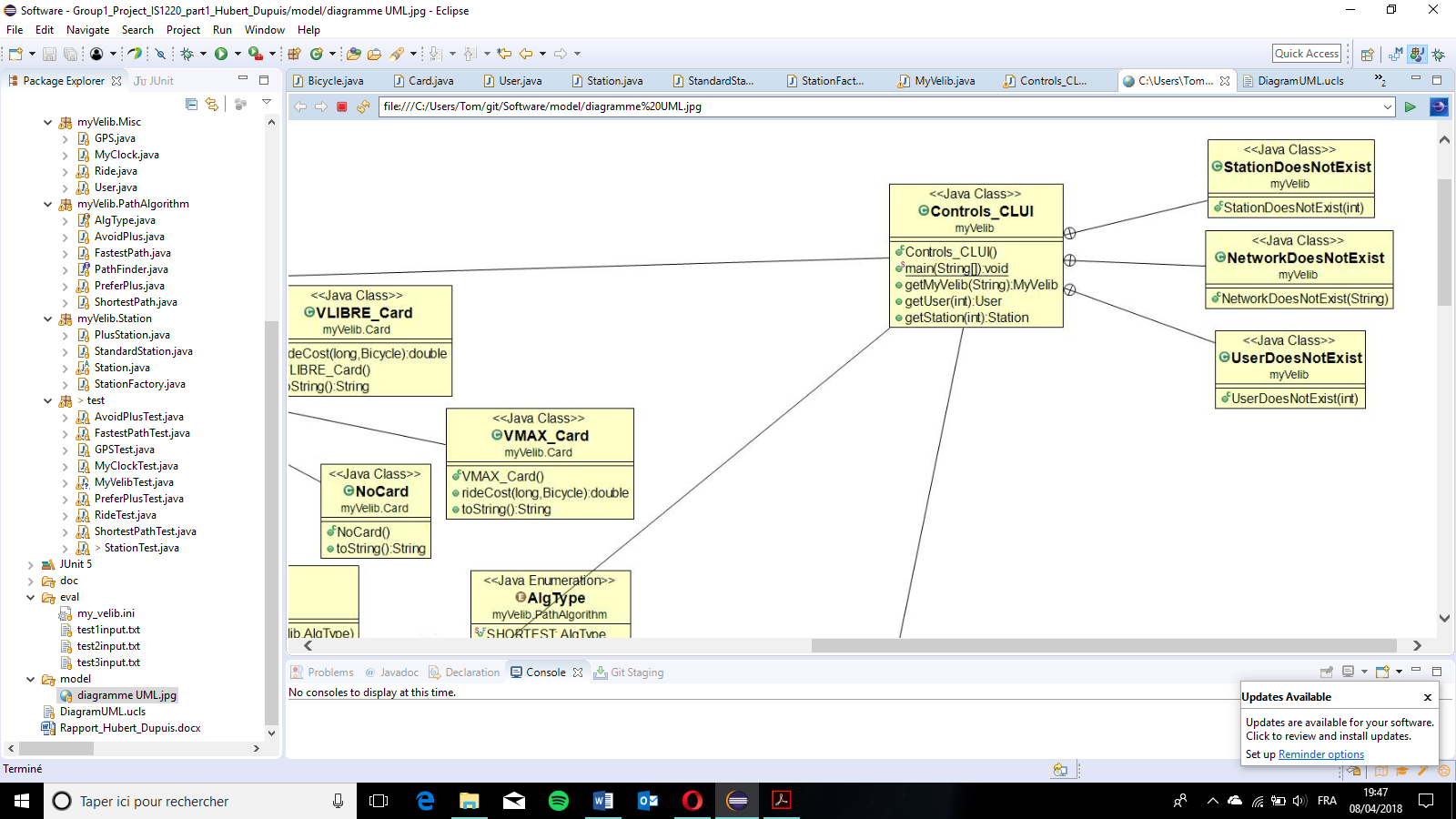
**b. Architecture principale « core »**



La classe Ride gère toutes les informations concernant le trajet de l’usager et appelle les algorithmes de pathfinder pour trouver les stations de départ et d’arrivée, et les stocker, ainsi que les temps de trajet.

Nous avons également créé une classe MyVelib qui peut être instanciée et contient toutes les stations et utlisateurs, ainsi qu’une classe User qui contient entre autre les coordonnées GPS, une carte et un vélo lorsque celui-ci utilise un velib. Elle fait ainsi le lien entre toutes les autres et fait tourner tout le système MyVelib.

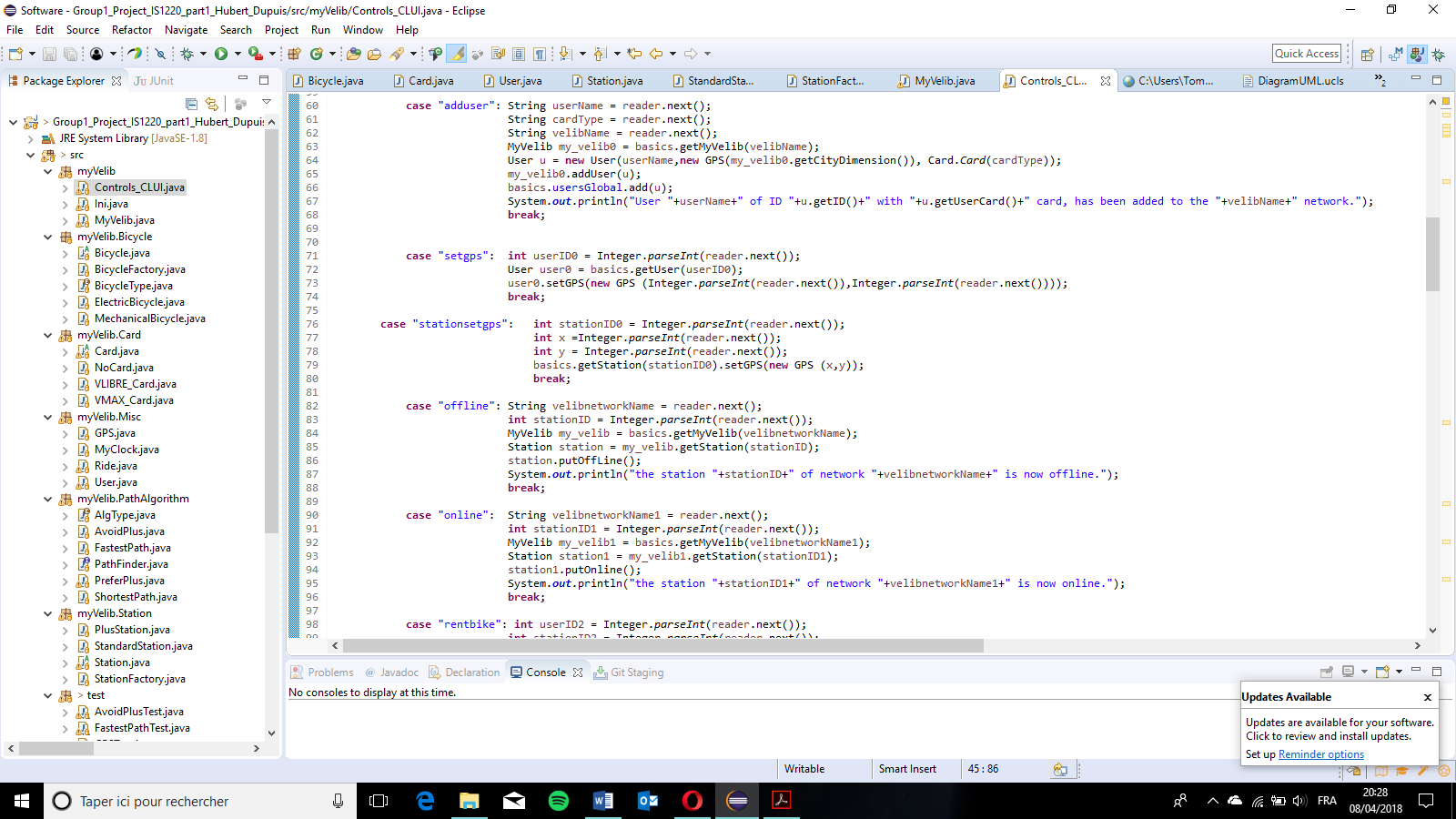
Le GPS est géré par la classe GPS et implémenté par 2 coordonnées x et y.

 **c. CLUI**

Nous avons opté pour un contrôle par commande utilisateur. L’avantage étant d’être facilement et rapidement modifiable pour ajouter des commandes au système. Dans le meilleur des mondes, il est vrai qu’il faudrait une GUI pour être utilisable de manière intuitive pour un utilisateur lambda.

Nous avons utilisé une méthode main avec un switch{ case… } comprenant toutes les commandes possibles. Chaque commande déclenche alors un case et exécute une certaine opération.

Exemple :



Afin d’éviter toute tentative de rentrer des commandes erronées, nous avons mis en place un système d’exceptions dans le cas où la commande serait fausse ou bien si celle-ci ne peut s’appliquer (Station-User-NetworkDoesNotExist).

**4. Tests et scénarios**

Scénario 1

Nous nous attachons dans ce scénario à tester deux types de comportement possible pour les usagers. Dans un cas, l’usager demande une planification de trajet au système, dans l’autre il effectue un trajet sans demander d’aide. On va créer un réseau de station nommé « paris », et le garnir de deux usagers : tom et felix. Tom demande une planification de son trajet jusqu’à une position GPS (3,9) avec l’algorithme shortest, dont on peut vérifier la validité grâce à la carte affichée. Il prend son vélo à la station proposé avec la commande startride et le reposer avec la commande endride. On en profite pour tester que quand la station d’arrivé d’un trajet est mise hors ligne, l’usager concerné est notifié et son trajet modifié. Felix lui va juste emprunter un vélo à une station quelconque, et le ranger à une autre station quelconque. On vérifie que les statistiques des usagers sont bien à jour, et notamment que le coût de leur trajet est exact.

|  |
| --- |
| setup paris  adduser tom VLIBRE paris  adduser felix VMAX paris  display paris  displayuser 0  displayuser 1  askforrideplan 0 3 9 ELECTRIC shortest paris  passingtime 90  startride 0  putofflineendstation 0  displayride 0  passingtime 90  endride 0  displaystation paris 2  rentbike 1 2 MECHANICAL  displaystation paris 2  passingtime 130  displaystation paris 8  returnbike 1 8  displaystation paris 8  displayuser 0  displayuser 1  exit |

Pour tester ce scénario, copier/coller dans le CLUI la commande :

***runtest eval/test1input.txt***

Scénario 2

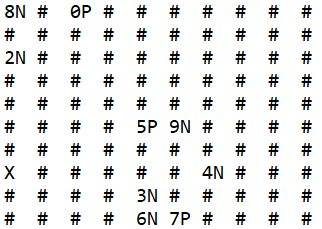
|  |
| --- |
| setup p  adduser tom VLIBRE p  display p  displayuser 0  rentbike 0 5 MECHANICAL  passingtime 75  returnbike 0 3  displayuser 0  exit |

Pour tester ce scénario, copier/coller dans le CLUI la commande : ***runtest eval/test2input.txt***

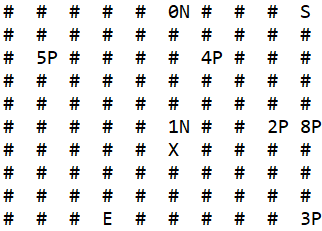
? – CLUI

La CLUI est fonctionnelle. Elle a été codé avec une condition switch mis à l’intérieur d’une boucle while. Pour pouvoir lire à la fois les commandes rentrées par l’utilisateur et celle situé dans un fichier scénario, il a fallu faire alterner le port d’entrée du scanner, entre system.in et le fichier. Un point décisif et qui fait peut-être la différence entre notre système et celui des autres étudiants, c’est que le temps s’écoule véritablement dans le système. Nous avons fait une mise à l’échelle telle que 1 seconde dans le monde réel correspond à 1 heure dans le programme. Cela permet de simuler efficacement l’écoulement du temps quand on rentre les commandes une à une, mais pas lors de la lecture d’un fichier scénario entier (les instructions se succèdent trop rapidement…). C’est pour cela qu’une commande passingtime <time> permet de faire avancer artificiellement le temps à l’intérieur du système.

Un manque de temps nous a empêché de parfaire la CLUI. Elle gère très mal les mauvaises commandes en entrée, et plutôt que de prévenir l’utilisateur que sa commande est erronée, La CLUI se ferme à la moindre erreur en envoyant le message d’erreur correspondant. Bien que n’ayant pas fait de GUI, une carte rudimentaire du réseau s’affiche à l’appel display <nomduréseau>. Elle permet de vérifier la justesse des algorithmes.



Une ville de taille 10\*10. 8N correspond à la position de la station d’ID 8, de type Normal. P pour Plus. Les X correspondent à des stations hors ligne.



La commande displayride <userID> affiche une autre carte de la ville, avec S comme start correspondant à la coordonnée GPS de l’utilisateur qui demande la planification de trajet. E comme end correspondant à la coordonnée GPS de l’arrivée.

Voici la liste des commandes implémentées dans la CLUI :

setup <MyVelibNetworkName> : to create a myVelib network with given name and as described as in the my\_velib.ini file.

runtest <Path> : run a test scénario, the path is typically eval/testNinput.txt

adduser <UserName> <CardType> <VelibName> : to add a user with name userName and card cardType (i.e. ‘‘none’’ if the user has no card) to a myVelib network velibnetworkName.

setgps <UserID> <x> <y> set the gps position of user using x and y int coordonnate.

stationsetgps <StationID> <x> <y> set the gps position of station using x and y int coordonnate.

offline <VelibNetworkName> <StationID> : to put oﬄine the station stationID of the myVelib network velibnetworkName

online <VelibNetworkName> <StationID> : to put online the station stationID of the myVelib network velibnetworkName

rentbike <userID> <StationID> : to let the user userID renting a bike from station stationID (if no bikes are available should behave accordingly)

returnbike <userID> <StationID> : to let the user userID returning a bike to station stationID at a given instant of time time (if no parking bay is available should behave accordingly).

displaystation <VelibNetworkName> <StationID> : to display the statistics (as of Section ??) of station stationID of a myVelib network velibnetwork.

displayuser <userID> : to display the statistics of user userID.

display <VelibNetworkName> : to display the entire status (stations, parking bays, users) of an a myVelib network velibnetworkName.

askforrideplan <userID> <x> <y> <BikeType> <AlgType> <VelibNetworkName> : comput a ride path for user userID who wants to get to position (x,y) on the map, with a bicycle of type BikeType ( ELECTRIC or MECHANICAL ), with a path of type AlgType (fastest, shortest, avoidplus or preferplus) of an a myVelib network velibnetworkName.

displayride <userID> : to display the current ride of an user userID if he has planned it of course.

passingtime <time(int)> : make a jump forward in time of time minutes.

putofflineendstation <userID> : put offline the station at the end of the ride of userID if he has any planned ride at the moment.

startride <userID> : make automatically user of ID userID take a bike at the start station of the ride he has planned.

endride <userID> : make automatically user of ID userID return his bike at the end station of the ride he has planned.

exit : exit the CLUI.

**5. Répartition du travail**

|  |  |
| --- | --- |
| Bicycle, Card et User (Core) | Felix |
| Ride et GPS (Core) | Tom |
| MyVelib (Core) | Felix et Tom |
| Système de stations (Core) | Felix |
| Algorithmes de recherche de station PathFinfer (Core) | Tom |
| Controls\_CLUI | Felix |
| Scénarios | Felix et Tom |
| JUnits | Tom et Felix |
| Javadoc | Tom |
| Rapport | Felix et Tom |

**6. Conclusion**

Pour finir, les avantages de notre projet sont le respect de l’open close principle. En effet nous avons utilisé le factory pattern qui permettrait de rajouter des offres de carte Velib ou bien d’autres types de vélo facilement sans trop changer les autres classes (de même pour les algorithmes de recherche du plus court chemin). L’avantage du CLUI était que nous avons pu le coder de manière plus propre qu’un GUI. Ainsi il est extrêmement facile de rajouter des commandes.

Cependant, il aurait été plus judicieux dans la pratique d’implémenter un GUI si l’application était vouée à être utilisée par des utilisateurs lambdas. De plus, le calcul du plus court chemin s’effectue en supposant que l’utilisateur se déplace en ligne droite, ce qui est extrêmement exagéré. La nécessité de s’aider d’une carte réelle aurait néanmoins augmenté grandement la difficulté pour trouver les stations les plus proches.

Ce projet nous a permis tous deux d’augmenter grandement nos compétences en Java, au vu de l’étendu des connaissances nécessaires à son impélementation.

Merci pour votre lecture.