IS1220 – Final Project

MyVelib – a bike sharing system

**RAPPORT**

Félix Hubert & Tom Dupuis

**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**

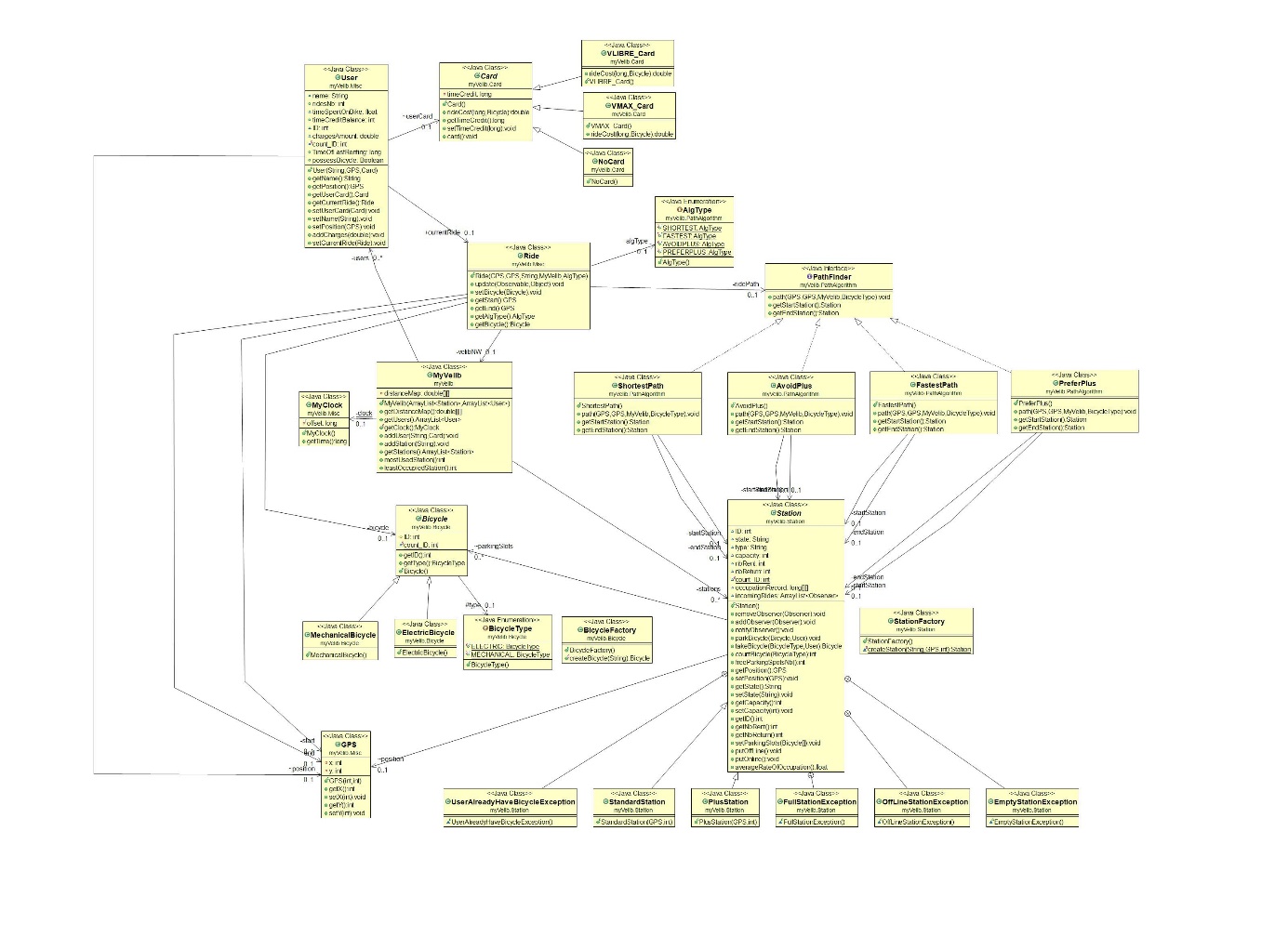
Dans le cas d’un système type Vélib’, les stations sont constituées de plusieurs places. Celles-ci peuvent être vides, contenir un vélo ou être hors-service. Les stations elles-mêmes peuvent donc se retrouver sans aucune place libre ou sans aucun vélo à fournir. L’usager ne peut donc pas prendre son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

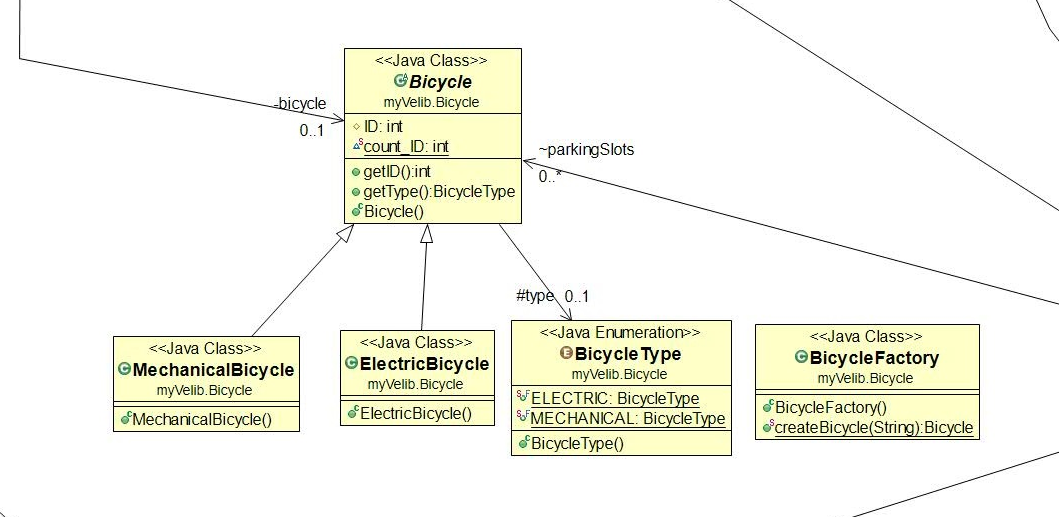
**3. Architecture du programme**

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.

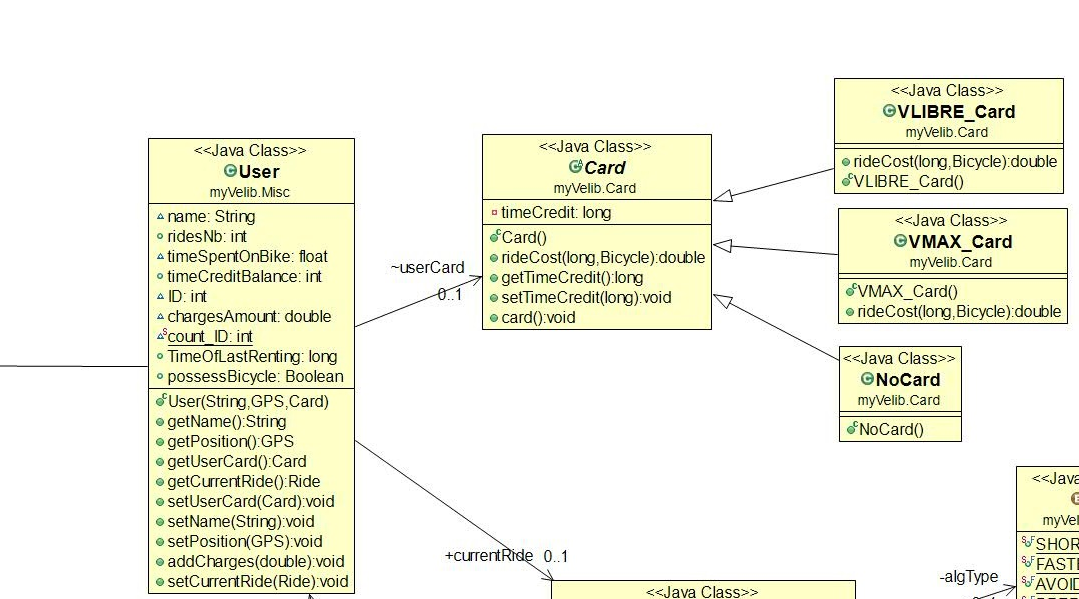
Tout d’abord, nous avons géré la gestion d’un ensemble de vélos par un design de factory. Bicycle est ici une abstract class détenant l’ensemble des méthode inhérentes à un vélo ainsi que ces attributs. MechanicalBicycle et ElectricalBicycle sont les implémentations réelles de vélo particuliers.

****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.

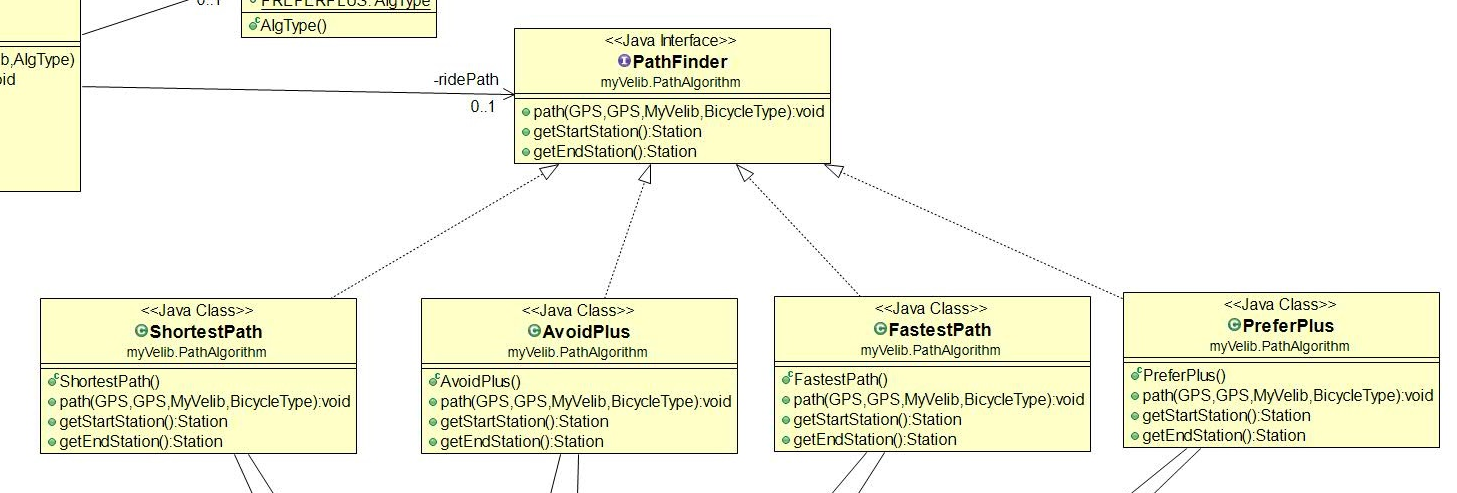
Il s’est avéré également judicieux de définir une classe NoCard afin de traiter les utilisateurs qui n’en auraient pas, plutôt que de mettre null dans l’attribut concerné, afin de pouvoir réaliser les même opérations sur l’User.



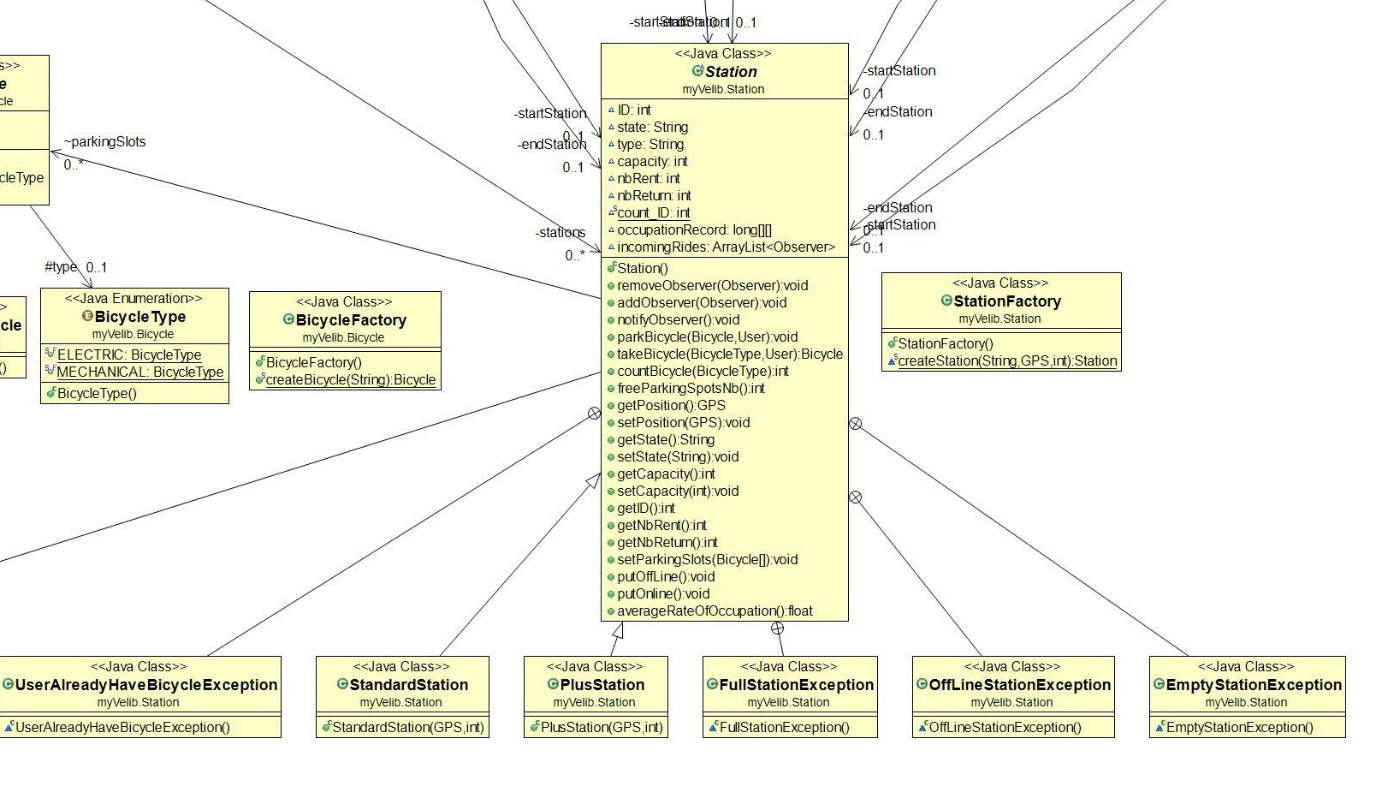
Autre pattern remarquable, nous avons utilisé le stragey pattern pour pouvoir gérer les différents algorithmes demandés pour trouver la station de départ et d’arrivée en fonction des coordonnées gps du départ et de destination.

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 proche dans chaque méthode path.



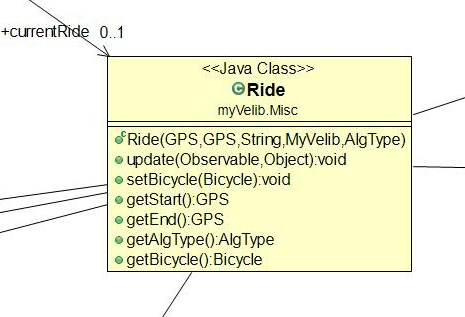
Finalement, les stations sont implémentées aussi grâce au factory pattern : on a une StationFactory , ainsi qu’une abstract Station, et 2 sous classes réelles PlusStation et StandardStation qui l’étendent.



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.



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.

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