CENTRALESUPELEC

IS1220 - Object Oriented Software Design

MY VELIB – BICYCLE SHARING SYSTEM

RAPPORT FINAL

Authors :
Aurélien Pasteau
Othmane Jebbari
Group 7

Professor :
Paolo Ballarini

CentraleSupélec

Table des matières

Table des figures

1. Introduction

Notre projet, intitulé « MyVelib » représente un système de location de vélos en libre-service entièrement codé en Java. Il mêle donc à la fois une réflexion sur la structure de la solution nécessaire pour répondre aux exigences d'un tel réseau de stations, de bicyclettes et d'utilisateurs, mais également une application technique directe par un code source en Java.
Ce projet a été extrêmement enrichissant tant au niveau de la réflexion qu'il entraîne (quelles sont les structures et les design pattern en Java les plus adaptées ?) qu'au niveau des compétences techniques qu'il exige (implémentation des classes, stratégies).
Bien évidemment, la structure même exigée de la solution a nécessité un travail incrémental. Nous avons tout d'abord réfléchi ensemble sur papier à l'architecture du système conditionnée à la fois par les exigences des spécifications multiples du projet, mais aussi par l'exigence de respect du principe « Open-Close ». Il a donc fallu définir les liens entre les différentes entités, et le fonctionnement du système à plusieurs échelles : dans sa globalité puis pour chaque fonctionnalité en détail, en vérifiant toujours que l'ajout de nouveaux types de vélos, cartes, critères de tri des stations, ou encore critère de planification de trajets ne pousse à modifier des classes déjà codées.
La principale difficulté a résidé dans le choix parmi la multitude de solutions possibles, mais le fait de réfléchir à deux a fortement accéléré le processus, dans la mesure où nos idées étaient souvent complémentaires et que nous pouvions souvent choisir entre 2 solutions.
Bien que nous soyons guidés dans l'approche du système, il pouvait parfois exister de nombreuses solutions répondant à certaines exigences du système (le choix devait prendre en compte la flexibilité permise par la solution choisie).
Presque chaque design pattern étudié en classe pouvait être utilisé à de multiples reprises : le plus dur a été de choisir les plus adaptés et d'évaluer leur nécessité. Il a également souvent fallu adapter un design pattern vu en cours, ou encore trouver une autre manière de respecter le principe open-close dans des cas où aucun des design pattern vus en cours ne convenait. Ensuite, une fois le squelette du système tracé, la répartition des tâches nous a permis de conserver une certaine liberté dans l'implémentation des méthodes. Ainsi, nous avons pu petit-à-petit rajouter des méthodes ou même des classes de manière incrémentale lorsque nous nous rendions compte que nous avions oublié de penser à une certaine fonctionnalité.
Nous avons ainsi appris l'importance de fixer un cadre commun (le squelette du système), que nous avons ensuite pu développer chacun selon les besoins rencontrés pendant les phases de code ou de test, en se tenant toujours au courant des modifications apportées.
Le fait de partager le travail a mis en exergue la nécessité de commenter le code créé pour permettre au partenaire de suivre son évolution.
Tester notre programme nous a paru être une étape difficile, car elle demande une rigueur irréprochable et surtout de la motivation, mais essentielle, car elle a permis de soulever quelques coquilles ou disfonctionnements dans le code et les relations entre les classes.
Notre répartition du travail en Code-Test pour la première partie du projet puis Test-Code pour la deuxième partie a permis d'éviter ce manque de motivation pour les tests unitaires et de porter un regard critique sur le code que l'autre a produit.

Aussi, l'objectif de ce rapport est de présenter à la fois notre solution, comment elle répond aux attentes et exigences du système, quels ont été les choix effectués en termes de design

patterns, et enfin comment nous avons fait face aux difficultés qui sont apparues dans notre cheminement.
Nous commencerons par introduire le contexte du projet, et la manière dont nous l'avons analysé, pour ensuite développer les choix que nous avons effectués (design/structures), l'implémentation de notre solution, les tests effectués et les résultats finalement obtenus.

2. Contexte

L'image d'une ville est devenue un facteur déterminant dans la société et pour l'économie. En effet, le tourisme d'une ville dépend beaucoup de son image, et il est pour la plupart des gens préférable d'habiter dans une ville où il fait bon vivre. Un critère décisif aujourd'hui est la propreté de la ville, ainsi que le mode de vie des personnes qui y vivent. C'est pourquoi de plus en plus de villes proposent des systèmes de location de vélos en libre-service.

Ces systèmes, comme Vélib à Paris ou encore Vélhop à Strasbourg qui connaissent un grand succès auprès des citadins, demandent cependant le développement de tout un système d'information qui permet son bon fonctionnement. Ce système d'information doit respecter beaucoup de spécifications tout en faisant face à de nombreuses contraintes, et c'est exactement la réalisation de ce système d'information qu'il nous est demandé de réaliser dans ce projet avec le système MyVelib.

En quelques mots, il s'agit de modéliser un tel système par le biais d'un réseau de stations, de vélos, d'utilisateurs munis ou non de cartes d'abonnement qui confèrent des avantages. Il s'agit également de planifier éventuellement les trajets des utilisateurs, et de modéliser ces trajets afin de pouvoir éventuellement les facturer puis mettre à jour l'état du réseau. Le système développé doit également permettre d'avoir accès facilement à diverses statistiques sur le réseau. Toute cette architecture est développée dans la première partie du projet et nous allons en donner le détail dans la partie suivante. Le projet continue alors avec le développement d'une interface client en lignes de commandes qui permet aux utilisateurs d'interagir avec le système.

Répondre aux exigences du système a nécessité d'en dessiner un squelette grâce au modèle du diagramme de classes UML, afin de définir toutes les composantes et toutes les interactions entre ces composantes.

Nous avons donc exposé dans la partie suivante le design de notre solution, qui correspond à l'analyse du sujet que nous avons faite.

3. Analyse, conception et implémentation

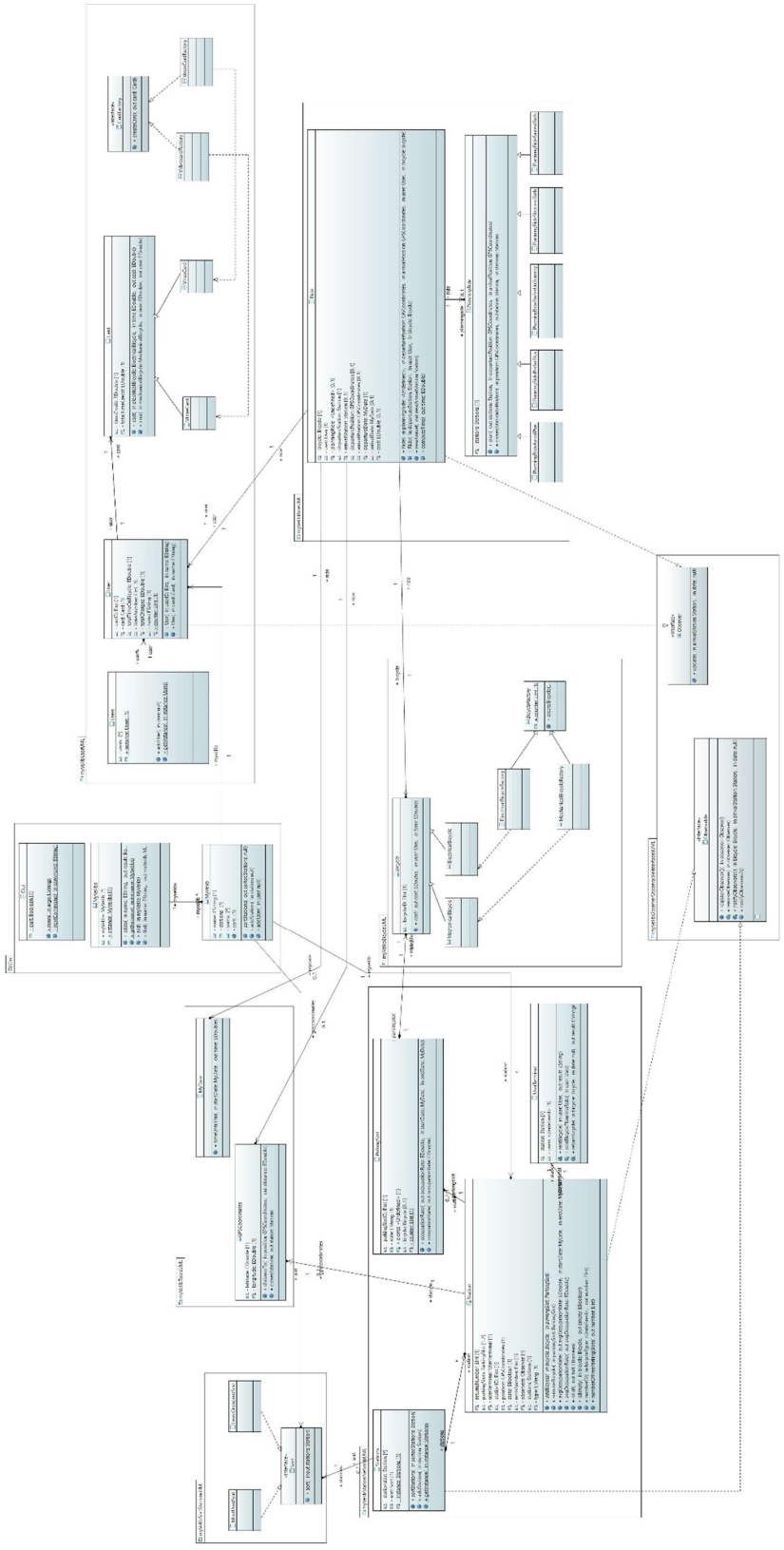

Figure 1 : MyVelib UML Diagram

3.1. Le réseau de stations et vélos de MyVélib

Le système MyVelib est constitué de stations, elles-mêmes constituées d'un ensemble de places de parkings, qui peuvent ou non contenir un vélo. Dans une station, un client peut louer un vélo à l'aide du « User Terminal », de plusieurs manières différentes : une location spontanée, ou bien une location non spontanée due à une prévision préalable du trajet. Un client peut également rendre un vélo dans cette station via le « User Terminal ». La classe Station comporte de plus des méthodes permettant de calculer des taux d'occupation moyenne de la station, soit depuis la création de la station, soit entre 2 dates à indiquer. Comme on le verra plus bas, il nous a paru nécessaire de créer également une classe Stations, qui comporte notamment comme attribut une ArrayList<Station> contenant toutes les stations créées. Cette classe est dotée du **singleton design pattern** pour s'assurer que l'instance de cette classe reste unique une fois qu'elle est créée. Lorsqu'on crée une nouvelle station, il faut entrer cet objet Stations en argument du constructeur, et la nouvelle station est alors automatiquement ajoutée à la liste de stations de l'objet Stations. Nous avons de plus fait le choix de ne pas créer 2 sous-classes PlusStation et StandardStation, car cela ne présentait pas d'intérêt. On se contente donc de distinguer les types de station par un attribut de type String qui représente le type de la station.

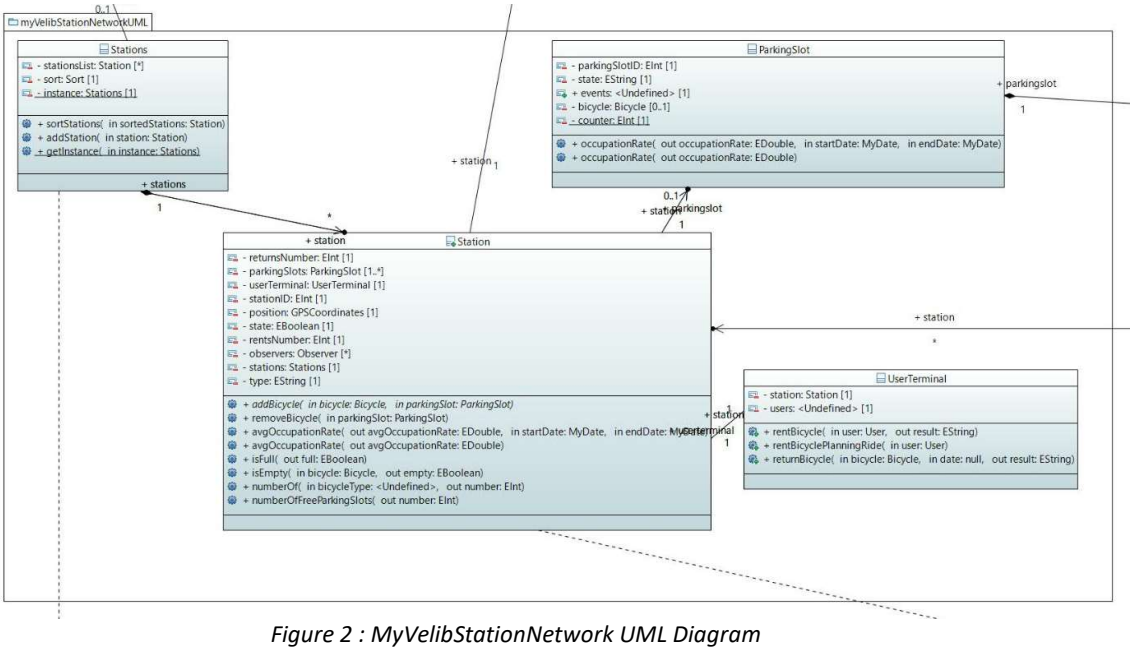

Figure 2 : MyVelibStationNetwork UML Diagram

Chaque Parkingslot a un attribut state de type String qui indique si ce parkingSlot est occupé, vide, ou hors-service. Un vélo est identifié par un identifiant entier unique. Une abstract classe Bicycle est alors étendue en deux sous-classes MechanicalBicycle et ElectricalBicycle. Pour créer des vélos, nous avons décidé de mettre en place un **Factory Method Pattern**, qui permet de respecter le principe open-close dans le cas où on rajouterait un jour un nouveau type de vélo. La classe Bicycle est dotée d'une méthode abstraite cost qui est overridé dans les deux sous-classes, afin de calculer le

cout d'un trajet qui aura utilisé ce vélo. Nous verrons dans une autre partie du rapport pourquoi ce choix à priori étrange a été fait : c'est un choix qui nous a permis de respecter le principe open-close alors que le cout d'un trajet dépend à la fois de l'utilisateur, du vélo utilisé et de l'éventuel type de carte de l'utilisateur.

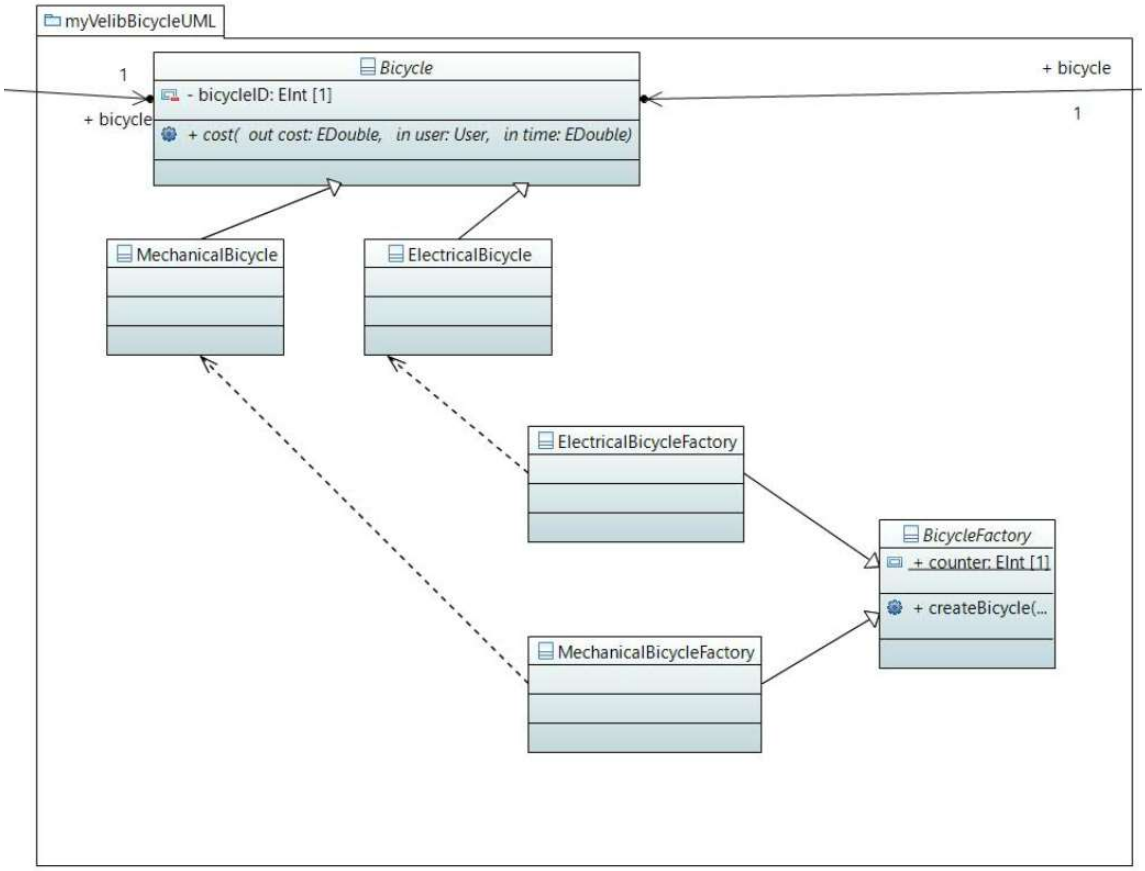

Figure 3 : MyVelibBicycle UML Diagram

3.2. Les utilisateurs et trajets de MyVelib

Une classe User permet de créer des objets utilisateurs. Elle contient comme attribut en plus d'un nom et d'un identifiant, toutes les statistiques de l'utilisateur, comme le nombre de trajets effectués, le temps total à vélo ou encore les charges totales. L'attribut card de type Card permet d'affecter éventuellement une carte à un utilisateur. En effet, une classe abstraite Card, qui contient notamment comme attribut les time credit et total time credit du propriétaire de la carte, est étendue en deux sous-classes : VlibreCard et VmaxCard, qui contient notamment comme attribut les time credit et total time credit du propriétaire de la carte. Elles overrident les deux méthodes de calcul de coût créées dans la superclasse abstraite. Afin de créer des objets cartes et dans l'anticipation éventuelle de l'apparition de nouveaux types de cartes, nous avons à nouveau mis en place un **Factory Method Pattern.**

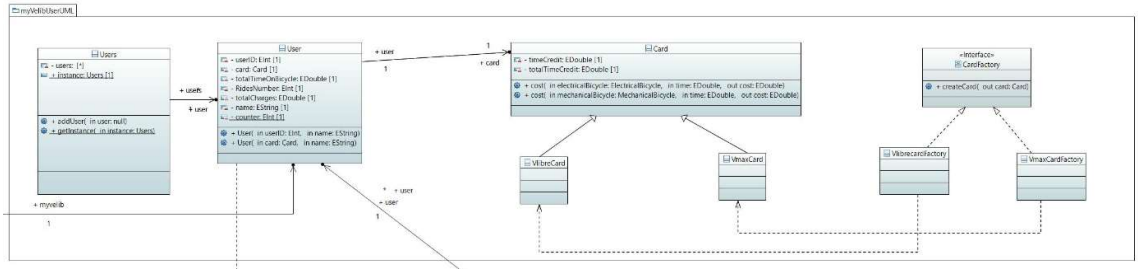

Figure 4 : MyVelibUser UML Diagram

Un trajet est représenté par une classe Ride. Dans le système Myvelib global, un trajet peut être initialisé de deux manières différentes :

- Soit l'utilisateur arrive à une station et indique qu'il souhaite louer un vélo, sans planification préalable du trajet, grâce à la méthode rentBicycle du UserTerminal de cette station, qui utilise alors un premier constructeur de la classe ride.
- Soit l'utilisateur se trouve à un point quelconque de la ville et lance une planification de trajet en entrant ses positions de départ et d'arrivée et le type de trajet qu'il veut (type de vélo, plus court, plus rapide, préservation uniformité de la répartition…)

La classe Ride contient plusieurs attributs : un attribut de type Bicycle, qui est le vélo utilisé pendant le trajet, un attribut de type User (utilisateur faisant le trajet), un autre de type PlanningRide qui vaut éventuellement null dans le premier cas ci-dessus, d'autres qui décrivent stations de départ et d'arrivée et positions de départ et d'arrivée, date de départ, date d'arrivée, et le coût du trajet pour l'utilisateur.

On se doute bien que lors de l'initialisation d'un trajet, le trajet n'a pas encore eu lieu, et nombre de ces attributs ne peuvent pas encore être renseignés. C'est pourquoi nous avons recours à un **observer pattern** : Chaque station du système MyVelib a un attribut ArrayList<Observer> qui contient en fait tous les trajets en cours, c'est-à-dire ceux dont le vélo n'a pas encore été ramené à une station via son UserTerminal. Les objets Ride sont ainsi les observers et les objets Station les observables. Lorsqu'un trajet est initialisé, celui-ci est automatiquement ajouté à la liste d'observers de chaque station via l'instance de Stations, et toujours via cette instance, le trajet est retiré de ces listes lorsque celui-ci est terminé.

Pour résumer la partie trajet dans les deux cas possibles :

- Un utilisateur arrive à une station et souhaite louer un vélo. La méthode rentBicycle du UserTerminal de la station est déclenchée : si la station contient un vélo, un objet de type Ride est créé, est automatiquement ajouté à la liste des observers de toutes les stations (ce qui est fait dans le constructeur de Ride), le vélo est retiré de la station et certaines statistiques de l'utilisateur et de la station sont incrémentées. L'objet Ride qui vient d'être créé a alors pour uniques attributs renseignés user, departureStation, departureDate et bicycle, les autres gardant leur valeur par défaut. Lorsque l'utilisateur arrive à sa station d'arrivée, la méthode returnBicycle du UserTerminal est exécutée. S'il y a de la place, l'utilisateur peut déposer le vélo à la place indiquée, et l'objet Ride est mis à jour (les attributs non encore renseignés sont renseignés). Pour cela, la méthode notifyObservers de la station d'arrivée est appelée, ce qui a pour effet de trouver parmi la liste des trajets non terminés (les observers) celui qui utilise le vélo venant d'être rendu, puis d'appeler la méthode update de ce Ride-là, et enfin d'ôter ce Ride de la liste d'observers de chaque station. Cette méthode update renseigne les attributs arrivalStation, arrivalDate, calcule le temps du trajet, met à jour d'autres statistiques, ajoute éventuellement du time credit à l'utilisateur selon la nature de la station

d'arrivée, et calcule le coût du trajet en appelant la méthode cost dans l'objet Bicycle utilisé pour le trajet.
La méthode cost de Bicycle est overridée dans les deux sous-classes. Ainsi, si on rajoute un jour un nouveau type de vélo, il n'y a pas à modifier les classes existantes. Dans chacune de ces implémentations de la méthode cost, si l'utilisateur du trajet n'a pas de carte, il est facturé selon les spécifications. Sinon, une des deux méthodes cost de l'objet Card est appelée, selon si le vélo utilisé est électrique ou mécanique. De nouveau, la méthode cost de Card est overridée dans les deux sous-classes pour calculer le coût du trajet suivant les spécifications propres à chaque type de carte. Encore une fois, le principe open-close est ici respecté, car si on ajoute un nouveau type de carte, il ne faudra pas modifier de classes déjà existantes.

- Un utilisateur se trouve à une position quelconque et souhaite se rendre à une autre position quelconque. Un objet Ride est alors créé, avec notamment comme attribut un objet de type PlanningRide, qui permet de calculer les stations de départ et d'arrivée selon la sous-classe de PlanningRide dont l'objet est une instance et selon le type de vélo souhaité. Lorsque cet utilisateur arrive à la station de départ qui lui a été attribuée, la méthode rentBicyclePlanningRide du UserTerminal de cette station est exécutée. Elle renseigne la date de départ du trajet, et les autres attributs manquants, et met à jour certaines statistiques. La fin du trajet s'effectue comme dans le cas précédent. Néanmoins, la station d'arrivée étant connue d'avance, si celle-ci devient hors-ligne ou pleine alors que le trajet n'est pas terminé, l'utilisateur reçoit une notification. S'il le veut, une autre station d'arrivée peut lui être attribuée. Ce système de notification d'utilisateur fonctionne à nouveau grâce à un **observer pattern** où les observers sont les utilisateurs et les observables sont les stations. Ainsi, lorsqu'une station devient pleine, un utilisateur qui serait dans un trajet planifié dont la station d'arrivée est celle-ci serait notifié, et il pourrait choisir de se voir recalculer une autre station d'arrivée.

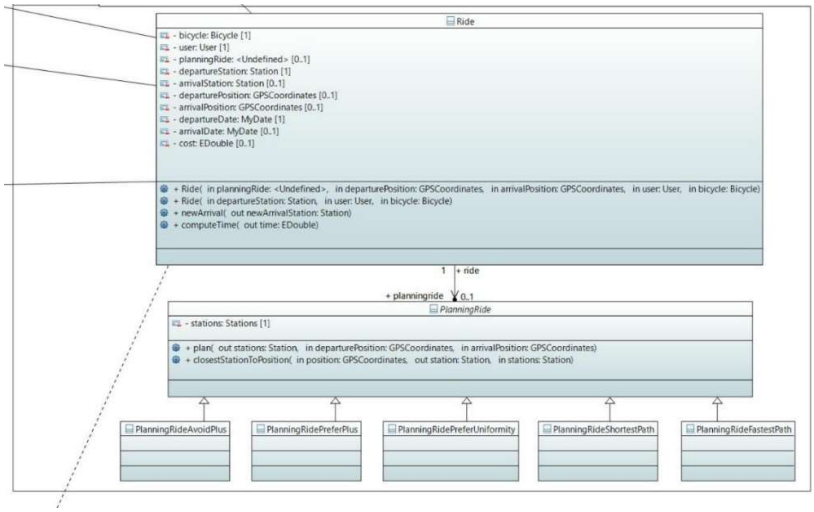

Figure 5 : MyVelibRide UML Diagram

Afin de répondre au besoin d'avoir plusieurs modes de planification différents, nous avons opté pour un **strategy pattern**. Ainsi, 5 sous-classes héritent de la classe abstraite PlanningRide, et la méthode plan est overridée dans chacune d'entre elles pour répondre au besoin spécifique.

Dans les deux cas ci-dessus, lorsqu'un vélo est rendu dans une station (à la fin d'un trajet donc) le coût du trajet est calculé. Dans les cas où l'utilisateur doit payer (par exemple un utilisateur avec carte qui roule plus d'une heure), nous avons pris quelques initiatives : tout d'abord, le time credit de l'utilisateur est ôté tant qu'il n'est pas nul ou qu'il n'a pas compensé entièrement la durée de la partie payante du trajet. Si le time credit de l'utilisateur est nul et qu'il n'a pas compensé toute la durée de la partie payante du trajet, celui-ci paye le reste ainsi : prix de l'heure * temps restant à payer. Ainsi, si un utilisateur possède une carte Vlibre avec 5 minutes de time credit et emprunte un vélo électrique pendant 75 minutes, il sera facturé (75 – 60 – 5)/60*2 = 0,33 euros.

3.3. Les statistiques, le tri et autres outils de MyVelib

Les statistiques sur les utilisateurs ou les stations sont stockées dans des attributs des classes correspondantes et mis à jour aux moments opportuns.
Pour le tri des stations, nous utilisons à nouveau un **strategy pattern**. L'unique instance de la classe Stations possède un attribut de type Sort. Sort est une interface implémentée par deux classes, MostUsedSort et LeastOccupiedSort, dans lesquels la méthode sort de Sort est overridée pour trier convenablement la liste des stations. Pour cela, ces deux sous-classes implémentent également la classe Java Comparator<> et en overridents la méthode compare.

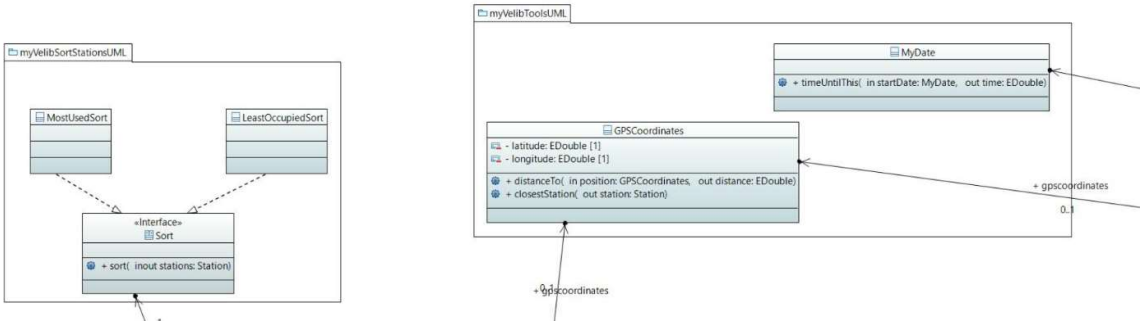

Figure 6 : MyVelibSort et MyVelibTools UML Diagrams

Enfin, nous avons dû créer une classe GPSCoordinates pour les positions avec notamment une méthode de calcul de distances, et une classe MyDate qui hérite de la classe Date de java pour les dates et le calcul de durées.

3.4. Le Command Line User Interface

La classe Clui est équipée d'une méthode main, qui charge tout d'abord le fichier my_velib.ini, afin d'initialiser un système MyVelib nommé my_velib et composé de 10 stations comprenant chacune 10 parkingslots et réparties dans un carré de côté 10 km avec 75% d'occupation par des vélos. Ensuite, tant que l'attribut statique de type boolean quit vaut faux, la methode main demande une nouvelle commande à l'utilisateur, puis appelle la méthode statique nextcommand, qui interprète et met en œuvre la commande. Naturellement, si l'utilisateur entre une commande non autorisée ou des paramètres non

conventionnels pour une commande, un message d'erreur est affiché et l'utilisateur peut réessayer de rentrer sa commande. Les commandes doivent commencer par un des mots clés ci-dessous et être suivi des paramètres adéquats, simplement séparés par un espace. Nous avons créé une classe MyVelib, qui permet d'instancier des objets représentant des systèmes MyVelib dans lesquels on stocke notamment tous les utilisateurs et toutes les stations du système. Cette classe MyVelib a également un attribut sort de type Sort qui permet de trier grâce à une méthode les stations d'un système MyVelib en particulier selon la polotique de tri correspondant au type de tri (Avec la classe Stations, on trie toutes les stations créées indépendamment du système MyVelib auquel elles appartiennent, ce qui est différend). Une classe MyVelibs dotée d'un **singleton design pattern** est également créée afin de stocker les différents systèmes MyVelib créés, et de pouvoir parcourir cette liste afin de retrouver un système MyVelib précis.

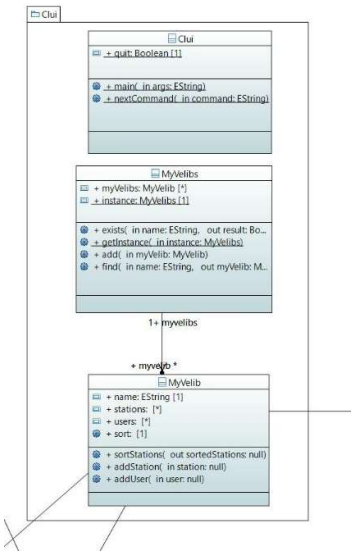

Figure 7 : Clui UML Diagram

Liste des commandes acceptées et définitions :

- **setup <name>** qui initialise un système MyVelib de nom name selon la même disposition que via le fichier my_velib.ini

- **setup1 <name> <nbstations> <nbparkingslots> <size> <nbbikes>** qui initialise un système MyVelib de nom name avec nbstations stations, composées chacune de nbparkingslots parkingslots, réparties aléatoirement dans un carré de coté size km avec en tout nbbikes vélos répartis aléatoirement.

- **adduser <name> <cardtype> <myvelib_system's_name>** qui créé un utilisateur qui s'appelle name, qui a une carte du type correspondant si cardtype vaut vlibrecard ou vmaxcard, et aucune si cardtype vaut none, et qui appartient au système MyVelib myvelib_system's_name.

- **offline < myvelib_system's_name > <stationID>** qui met en mode offline la station stationID du système MyVelib indiqué si ces deux entités existent

- **online < myvelib_system's_name > <stationID**> qui met en mode online la station stationID du système MyVelib indiqué si ces deux entités existent

- **rentbike <userID> <stationID>** qui permet à l'utilisateur d'identifiant userID de louer un vélo dans la station stationID, si ces deux entités existent, et qu'un vélo est disponible, et de faire tout le nécessaire (enlever un vélo de la station, mettre à jour des statistiques…)

- **returnbike <userID> <stationID> <year> <month> <day> <hour> <minute>** qui permet à l'utilisateur d'identifiant userID de rendre un vélo dans la station stationID à la date indiquée, si cet utilisateur existe et a en effet déjà emprunté un vélo, et que la station existe et est non pleine ou hors-ligne. Dans le cas contraire, un message d'erreur est renvoyé. Notez qu'à chaque commande, c'est l'identifiant de l'utilisateur qu'il faut rentrer et non son nom. L'identifiant du premier identifiant créé est 0, celui d'après 1, indépendamment du système MyVelib auquel ils sont ajoutés, et de même pour les stations.

- **displaystation <myvelib_system's_name> <stationID>** qui affiche le nombre de départs et de retours de vélos dans cette station, ainsi que son taux moyen d'occupation depuis sa création.

- **displayuser <myvelib_system's_name> <userID>** qui affiche le nombre de trajets de l'utilisateur, ses charges totales, le temps qu'il a passé sur un vélo, et son timecredit actuel ainsi que le timecredit total cumulé s'il possède une carte.

- **sortstation <myvelib_system's_name> <sortingpolicy>** qui renvoie la liste triée des stations du réseau en question via la politique de tri indiquée : mostusedsort ou leastoccupiedsort

- **display <myvelib_system's_name>** qui affiche les informations relatives à un système MyVelib : la liste des stations avec pour chacune la liste des parkingslots et leur état actuel, ainsi que la liste des utilisateurs.

- **runtest <file_name>** qui exécute les commandes indiquées dans le fichier texte file_name qui doit obligatoirement être situé dans le dossier eval. Le fichier doit contenir une commande par ligne, et laisser 2 lignes vides à la fin. Un fichier avec la réponse de la console à chacune des commandes est alors créé et placé dans le dossier eval.

- **exit** qui donne la valeur true à la variable statique quit, ce qui a pour effet d'arrêter l'exécution du programme. Cette commande est la seule manière d'arrêter l'exécution du programme.

4. Gestion des exceptions

Dans ce projet, nous avons beaucoup exploité les exceptions java. Cependant, cela ne suffit souvent pas. De nombreuses méthodes ne s'appliquent souvent que sous des conditions d'existence de certains objets ou de bonnes valeurs de certains attributs d'objets. Nous avons ainsi recours à quelques exceptions maison. Cependant, nous en avons créé en nombre limité, dans la mesure où le plus souvent, les méthodes ne sont appelées qu'à l'intérieur d'un test conditionnel qui vérifie les conditions d'application de la méthode, ce qui a pour conséquence qu'une exception pour cette méthode ne serait jamais lancée. Les exceptions que nous avons créées sont listées ci-dessous :

WrongParkingSlotException lancée dans les méthodes **addbicycle** et **removeBicycle** de Station
WrongCommandException lancée dans le CLUI
WrongRideException lancée dans la méthode **newArrival** de Ride
WrongTimePeriodException lancée par la méthode **avgOccupationRate** de Station

5. Tests

Test du MyVelib Core :

On a choisi de tester la plupart des méthodes dans un seul fichier Junit nommé RideTest pour éviter d'initialiser à chaque fois les stations et les vélos. Cela permet tout de même de tester toutes les méthodes, ie. toutes les fonctionnalités.

On a créé 4 stations chacune comporte 3 places libres et 7 occupées : comportant 5 vélos mécaniques et 2 vélos électriques.

Dans la vraie vie, la station station0 a les coordonnées GPS de la station de bus « les algorithmes », la station station1 a les coordonnées GPS de la station de bus « Joliot Curie », la station station2 a les coordonnées GPS de la station de bus « Moulon» et la station station3 a les coordonnées GPS de la station de bus « université Paris-Saclay ».

Après on passe à la création des users : user0 a une carte Vlibre, user1 a une carte Vmax, et user2 n'a pas de carte.

Au début chaque user prend un vélo de la station station0. User0 remet son vélo dans station0, User1 remet son vélo dans station1 et User2 remet son vélo dans station2.

Ensuite, User1 veut aller d'un lieu près de station0 à un lieu près de station1 et choisit un vélo mécanique et une stratégie « avoidplusstations ». Sa station de départ va etre station0 et sa station d'arrivée va etre station1. Le trajet dure 3 secondes.

Après, User1 veut aller d'un lieu près de station0 à un lieu près de station1 et choisit un vélo mécanique et une stratégie « preferplusstations ». Sa station de départ va etre station0 et sa station d'arrivée va etre station1. Le trajet dure 3 secondes.

Après, User1 veut aller d'un lieu près de station0 à un lieu près de station3 et choisit un vélo mécanique et une stratégie « shortestpath ». Sa station de départ va etre station0 et sa station d'arrivée va etre station3. Le trajet dure 3 secondes.

User2 veut aller d'un lieu près de station0 à un lieu près de station3 et choisit un vélo mécanique et une stratégie « fastestpath ». Sa station de départ va etre station0 et sa station d'arrivée va etre station3. Le trajet dure 3 secondes.

Enfin, User1 veut aller d'une position se trouvant presque au milieu des deux stations Station0 et Station1 mais un peu plus proche de station0 que de station1. Il choisit un vélo mécanique et une stratégie « uniformity preservation ». Sa station de départ va etre Station1 (meme si il est plus proche de Station0) et sa Station d'arrivée va etre Station3 . Le trajet dure 3 secondes. Après ces 3 secondes, user1 aurait pris la dernière place de la Station3. Par conséquent, User2 recoit une notification qui dit que il n'y a plus de places dans la Station3 et on lui propose une station d'arrivée différente.

On teste les méthodes de calcul du « total time credit » pour les User0 et User1 et du « total charges » pour User2.

On teste les méthodes de calcul du nombre total de locations et de remises de vélos pour toutes les stations.

On teste la méthode de calcul du temps pris par un trajet pour le ride3.

On teste la méthode de calcul du temps total passé sur un vélo pour user1.

On teste les deux méthodes de tri « MostUsedSort » et « LeastOccupiedSort ».

On teste la méthode de calcul du « average occupation rate » pour la station3.

Dans les Junit BicycleTest, ElectricalBicycleFactoryTest et MechanicalBicycleFactoryTest on teste les méthodes de création des deux types de vélos.

Enfin, On teste la méthode distanceto() de la classe GPSCoordinates dans le Junit test GPSCoordinatesTest

L'exécution du test Junit s'effectue sans encombre.

Test du MyVelib CLUI :

Afin de tester le command line user interface, nous avons procédé en deux étapes : tout d'abord, nous avons testé les mots clés autorisés un par un, en donnant des commandes justes mais aussi des commandes fausses pour vérifier la bonne réaction de la console. Nous avons ensuite écrit 2 fichiers texte de test, contenant une série de commandes que nous pouvons exécuter à l'aide de la commande runtest <filename> : ils sont disponibles dans le dossier eval aux côtés du fichier my_velib.ini.

Dans le premier fichier, **testScenario1.txt**, toutes les commandes sont écrites de manière juste. En plus de l'initialisation initiale grâce au fichier my_veli.ini, on initialise un réseau velib1 avec 10 stations, 10 parkingslots dans chaque station, le tout réparti dans un carré de 5 km de côté avec 50 vélos en tout.
On ajoute à ce réseau 2 utilisateurs : bob, avec une carte vmax, et vanessa, avec une carte vlibre. On met la station 13 et la 16 offline, puis bob loue un vélo dans la station 14. La

station 13 redevient online, et bob rend son vélo à la station 13 le 15 mai 2018 à 8h36. On affiche les statistiques de la station 13 et de bob. On trie ensuite les stations de velib1 selon la politique mostusedsort. Enfin, on affiche la contenance du réseau velib1 et on quitte le CLUI.

Dans le deuxième fichier, **testScenario2.txt**, quelques commandes sont écrites de manière fausse ou bien de sorte que le CLUI renvoie un message d'erreur. On initialise en plus de l'initialisation initiale un réseau velib2, avec 10 stations contenant chacune 5 parkingslots, réparties dans un carré de côté 5 km, avec 30 vélos au total.
On essaye d'ajouter un utilisateur u0, qui possède une « randomcard ». Ce type de carte étant inconnu, le clui doit renvoyer un message d'erreur, tout en continuant avec les commandes suivantes. On ajoute ensuite 6 autres utilisateurs u1 à u6, de manière correcte. Ces six utilisateurs, qui ont donc comme identifiant 0, 1, 2, 3, 4 et 5 essayent tous de louer un vélo à la même station d'identifiant 15. Comme il n'y a que 5 parkingslots, même si la station est pleine, au moins un des utilisateurs se verra être dans l'impossibilité de louer un vélo. De nouveau, un message d'erreur doit s'afficher. On essaye ensuite de faire louer un vélo à l'utilisateur d'identifiant 6, qui n'existe pas, pour encore une fois observer la réaction du CLUI. Enfin, l'utilisateur d'identifiant 5 essaye de rendre un vélo à la station 16, bien que, comme nous l'avons vu, il n'ait pas de vélo. On observe normalement un nouveau message d'erreur. On quitte ensuite le CLUI.

Pour lancer ces tests vous-même, exécutez la méthode main de la classe Clui, puis écrivez **runtest testScenario1.txt** ou encore **runtest testScenario.txt**. Les 2 fichiers output vont alors apparaître dans le dossier eval.

Ces tests ont permis de repérer de nombreuses petites erreurs ou de petits oublis dans le code, ce qui les rend indispensable à une mise en production ultérieure.

6. Conclusion et pistes d'amélioration

A travers ce projet, nous avons fourni l'analyse et l'implémentation d'une structure de type réseau de vélos de location en libre-service, capable de relier des stations, des vélos, des utilisateurs, et de garder en mémoire toute une série d'informations, tout en étant capable d'interagir avec l'utilisateur. Dans ce rapport, nous avons présenté le contexte de ce projet, notre analyse des spécifications, notre implémentation et la manière dont elle met en pratique les concepts du cours ainsi que le principe open-close. Nous l'avons illustré par une représentation sous forme de diagrammes de classes UML. Nous avons enfin présenté les tests qui nous ont permis de corriger ou d'ajuster de nombreuses parties de notre code. Notre système répond donc à la majeure partie des exigences imposées à une telle plateforme.
Cependant, nous sommes conscients des limites de notre production, qui pourrait toujours être améliorée et complétée.
En effet, nous pourrions par exemple fonctionner moins avec des tests if, et plus avec de la gestion d'exceptions.
En termes de fonctionnalités, notre système bien qu'assez complet ne prend pas en compte toute la chaine d'utilisation d'un utilisateur : il manque par exemple le système d'interaction

réel avec le client. En effet, le Clui sert pour un gérant du système à ajouter des stations ou encore des clients sur le système informatique, mais nous n'avons pas implémenté d'interface graphique ou d'application mobile avec lesquelles un client interagit vraiment. Sur une telle interface, un client pourrait par exemple procéder au paiement d'un trajet via une plateforme de paiement. Nous aurions également pu prendre en compte tous les autres partis qu'impose un tel système : un tel système informatique doit en réalité prendre en compte les équipes de maintenance, plus de matériel que de simples stations….
En bref, il resterait encore beaucoup de travail pour pouvoir produire un système réaliste. Tout de même, ce projet fut une expérience très enrichissante tant du côté technique que du côté gestion de gros projet à durée limitée. Il nous aura mis à assez rude épreuve par moments, et nous aura ainsi procuré un gain de maturité ainsi que de précieuses compétences techniques utiles pour notre vie future. Le travail en équipe nous a permis d'appréhender une autre dimension indispensable au métier d'ingénieur, ce qui nous a permis d'être plus efficace et d'apprendre avec et grâce à l'autre.

Répartition du travail

Aurélien Pasteau	**Othmane Jebbari**
Nous avons effectué l'analyse des spécifications ensemble et avons défini tout le fonctionnement du système.	Nous avons effectué l'analyse des spécifications ensemble et avons défini tout le fonctionnement du système.
Construction du diagramme de classe UML	Test du core system et correction des erreurs
Codage de toutes les classes du Core system	Codage de tout le CLUI
Aide pour résoudre les problèmes identifiés pendant les tests du Core system	
Tests du CLUI et modifications du code	
Ecriture du rapport sauf la partie test	