



Rapport de Projet : My Vélib'

Encadrement

Paolo Ballarini

Arnault Lapitre

Anis Ben Said

Léonard Boussioux

25 Mars 2018

Table des matières

1	Introduction	4
2	Les classes utilisées	5
2.1	Les ressources autour du système de partage de vélos	5
2.1.1	Le client MyVelib	5
2.1.2	Le réseau Network	6
2.1.3	Station	6
2.1.4	ParkingSlot	7
2.1.5	Bicycle	8
2.2	Les ressources autour de l'utilisateur	8
2.2.1	L'utilisateur User	8
2.2.2	Le système des cartes	11
2.2.3	La carte de crédit CreditCard	11
2.2.4	Les cartes Velib	12
2.3	Les classes liées à l'utilisation du réseau Velib	13
2.3.1	La classe GPS	13
2.3.2	Les classes de planification de chemins	13
3	L'architecture utilisée	15
3.1	Le diagramme UML	15
3.2	La gestion du temps :	15
3.3	Visitor pattern	16
3.4	Observer pattern	17
3.5	Strategy pattern	18
3.6	CLUI et Factory pattern	18

4	Core	20
4.1	Création de l'architecture Velib en elle-même	20
4.1.1	Ajouter stations, slots et vélos dans le réseau	20
4.2	Planifier une course	21
4.2.1	Les méthodes de TripPreference pour trouver les stations les plus proches	21
4.2.2	Les méthodes de TripPreference permettant de déterminer le chemin	22
4.2.3	Replanifier une course	24
4.3	Effectuer la course	24
4.3.1	Commencer la location d'un vélo	24
4.3.2	Déposer un vélo	25
4.4	Les fonctions statistiques et de tri	26
4.4.1	Autour des utilisateurs	26
4.4.2	Autour des stations	26
5	Comment tester notre code	28
5.1	Tester avec le CLUI	28
5.1.1	testCore.txt	28
5.1.2	testRentBikeReturnBike.txt	29
5.1.3	testCost.txt	29
5.1.4	testCreditTime.txt	29
5.1.5	testPlanRide.txt	29
5.1.6	testObserverPattern.txt	30
5.1.7	testTimeHandling.txt	30
5.1.8	testSortStation.txt	30
5.1.9	A vous de jouer!	30
5.2	Nos JUnit Test	32
5.2.1	Création d'un réseau MyVelib	32
5.2.2	Location d'un vélo	32
5.2.3	Simulation d'une course à vélo	32
5.2.4	Calcul de statistiques	32
5.3	Intérêt des tests	33

6	Critique de notre modèle	34
6.1	Avantages	34
6.1.1	Réponse aux spécifications du sujet	34
6.1.2	Respect du principe <i>Open / Close</i> et usage de patterns	34
6.1.3	Prise en compte de nombreuses exceptions	35
6.1.4	Prise en compte de la complexité de calculs	35
6.1.5	Optimisation de l’affichage des résultats	35
6.2	Défauts	35
6.2.1	Si on veut ajouter un nouveau type de vélo	35
6.2.2	Simulations avec des <i>threads</i>	35
6.2.3	Création de la GUI	35
7	Répartition et méthode de travail	36
8	Conclusion	38

1 Introduction

L'objectif du projet est de modéliser un système de partage de vélos et son fonctionnement en programmation orientée objet, afin de mesurer sa performance en fonction du nombre et de l'allocation de ses ressources. La finalité est de fournir une interface en ligne de commande afin de pouvoir utiliser et simuler un tel système facilement. Nous utiliserons ici la technologie JAVA avec l'environnement Eclipse. Nous réserverons également la police en gras pour les classes JAVA et italique pour les méthodes.

Nous présentons ici l'ensemble du projet, qui comprend les points suivants :

- le choix des différentes ressources et de leur structure
- les problématiques importantes qui sont survenues lors de la modélisation
- les avantages et défauts de notre modélisation
- la répartition du travail.

Nous avons fait le choix d'un rapport le plus exhaustif possible qui permette de comprendre nos choix et notre code sans avoir à le regarder. L'inconvénient est que nos bonnes idées peuvent se retrouver noyées. Pour s'éviter une lecture peut-être rébarbative nous invitons donc le lecteur à regarder le chapitre 6 *Critique de notre modèle* où nous avons condensé nos bonnes idées en proposant des renvois à certaines sections si besoin.

2 Les classes utilisées

Pour les classes, nous avons majoritairement suivi les indications données par la description du projet. Nous allons détailler chaque classe de façon détaillée, car elles dépendent fortement de notre choix de modélisation.

2.1 Les ressources autour du système de partage de vélos

Nous décrivons ici l'architecture donnée au système de partage de vélos en lui-même. La classe **Network** représente le système Velib qui va contenir les différentes stations **Station** où l'on peut prendre ou déposer un vélo **Bicycle**. Chaque **Station** contient un certain nombre d'emplacements **ParkingSlot** pouvant accueillir des vélos.

Remarque : pour chacune des classes précédentes, on garantit pour chaque objet un unique identifiant de type **int** en utilisant un compteur static qu'on incrémente à chaque nouvelle création d'objet.

2.1.1 Le client MyVelib

Cette classe regroupe l'ensemble des réseaux **Networks**. Elle permet aussi la gestion du temps. En effet, le temps est stocké dans un attribut de MyVelib et chaque action (louer et rendre un vélos) va faire avancer le temps dans notre simulation. MyVelib a pour attributs :

- `private ArrayList<Network> networks` : qui contient la liste des networks.
- `private Timestamp iniTime` : le temps de création de la classe. Lors de l'initialisation d'un objet MyVelib, iniTime reçoit le temps machine.
- `private Timestamp currentTime` : le temps actuel. Au fur et à mesure qu'on effectue des opérations, ce temps va évoluer vers le futur.

La classe **MyVelib** contient aussi des getters pour trouver un réseau, une station, ou un utilisateur à partir de leurs identifiants en plus de méthodes permettant de faire évoluer le temps et de l'afficher.

2.1.2 Le réseau Network

La classe **Network**, représente un réseau contenant les stations, les emplacements de vélos, les vélos, les utilisateurs, etc.

Il a pour attributs :

- private **String** name : le nom du réseau.
- private ArrayList<**Station**> stations ; une liste de **Station** contenant l'ensemble des stations du réseau.
- private ArrayList<**User**> users : une liste d'utilisateurs contenant l'ensemble des utilisateurs du réseau
- private ArrayList<**Bicycle**> bicycles : une liste contenant l'ensemble des vélos du réseau.

La classe contient par ailleurs des méthodes permettant de :

- accéder et modifier certains attributs de la classe.
- retrouver des stations remplissant certaines conditions comme : être de type plus, avoir l'état OnService ou Offline, se situer dans un certain rayons, être la station la proche d'une certaine station etc... Ces méthodes sont utilisés par les permettant de calculer les chemins selon les préférences de l'utilisateur.

2.1.3 Station

Classe représentant une station. Chaque station est caractérisée par :

- private Network network : le réseau auquel appartient la station.
- private **String** name : un nom de type **String** ;
- private **int** id : un unique identifiant de type **int**. Pour garantir l'unicité de chaque id, nous utilisons un attribut : static int IDcounter=0 qu'on incrémente de +1 à chaque fois qu'on crée une nouvelle station. Nous utilisons cette méthode pour toutes les classes donnant naissance à des objets avec un identifiant unique.
- private StationType type : normal ou plus. Nous utilisons une énumération **public enum** StationType **Normal, Plus** pour créer ces deux sortes de station.
- private **GPS** position : une position GPS, classe que l'on décrit en 2.3.1.
- private **Status** status : représentant l'état de la station. Nous utilisons une énumération **public enum** Status **OnService, Offline** ; pour créer ces deux sortes d'état.
- private ArrayList<ParkingSlot> parkingSlots : une liste d'emplacements de stationnement.

- private **int** numberOfRentals et private **int** numberOfReturns : deux entiers représentant le nombre de départs et de retours de vélos dans la station. Nous avons créé ces deux entiers pour des raisons de praticité pour classer les différentes stations selon le nombre de retours et départs.
- private ConcurrentSkipListMap <Timestamp, int[]> stationHistory : un dictionnaire ordonné permettant de connaître à chaque instant combien il y a d'emplacements libres, cassés ou occupés dans la station. On utilise la structure d'une **ConcurrentSkipListMap** <**Timestamp**, **int**[]>, structure que l'on décrira davantage en 3.2. A chaque fois qu'un des emplacements change de statut, on ajoute une entrée dans le dictionnaire avec pour clé la date du changement et valeur un quadruplet donnant le nombre d'emplacements libres, cassés, occupés par des vélos mécaniques et occupés par des vélos électriques.

La classe contient aussi un attribut static final **Duration** plusTimeCredit qui permet de stocker le Time Credit qu'on attribue à un utilisateur possédant une carte Velib lorsqu'il rend son vélo dans une station plus. Ce genre de reflex permet de respecter une architecture OPEN/CLOSE pour ce code.

La classe contient par ailleurs des méthodes permettant de :

- accéder et modifier certains attributs.
- ajouter des **ParkingSlot**, de récupérer le nombre d'emplacements occupées et les types de vélos qui y sont garés, le nombre d'emplacement hors services, de trouver un vélo d'un certain type ou un emplacement vide.
- louer et rendre des vélos.
- calculer les statistiques de chaque station.
- faire de beaux affichages avec les méthodes *String.format*.

2.1.4 ParkingSlot

Chaque *parking slot* ou emplacement pour parquer vélo est caractérisé par :

- un nom de type **String**
- un unique identifiant de type **int**
- la station à laquelle l'emplacement appartient
- une variable **Bicycle** qui contient l'objet **Bicycle** si le vélo est stationné à l'emplacement, null sinon.
- un statut : libre, occupé par un vélo électrique, occupé par un vélo mécanique ou hors service. Nous utilisons une énumération **enum Status Free, OccupiedByElectrical, OccupiedByMechanical, Broken** ;

- un dictionnaire ordonné **ConcurrentSkipListMap** \langle **Timestamp**, **ParkingSlot.Status** \rangle représentant les états du *ParkingSlot* au cours du temps. A chaque fois qu'un des emplacements change de statut, on ajoute une entrée dans son dictionnaire avec pour clé la date du changement et valeur son nouveau statut. Nous reviendrons sur la structure en 3.2.

2.1.5 Bicycle

Cette classe décrit les vélos qui vont être utilisés par les utilisateurs. Chaque vélo est caractérisé par :

- un unique identifiant de type **int**
- un type : électrique ou mécanique. Nous utilisons une énumération **enum BicycleTypeElectrical, Mechanical** pour créer ces deux sortes de vélo. L'utilisation de **enum** permet une gestion simple et plus rapide des différents types de vélo.
- une vitesse de type **int**, celle qui sera utilisée lors des calculs de chemins. On définit une vitesse de vélos électriques et une vitesse de vélos mécaniques.

2.2 Les ressources autour de l'utilisateur

2.2.1 L'utilisateur User

L'utilisateur est celui qui va utiliser le système Velib pour louer des vélos et se déplacer. Nous avons implémenté le pattern Observer pour répondre à la problématique de prévenir l'utilisateur si la station qui l'intéresse pour déposer un vélo est désormais hors ligne ou pleine. Nous reviendrons sur le pattern Observer en 3.4. Chaque utilisateur possède :

- un nom de type **String**
- un unique identifiant de type
- une vitesse de déplacement à pied, de type **int**
- une position GPS de type GPS (voir en 2.3.1)
- une carte de crédit de type **CreditCard** (voir en 2.2.3)
- une carte Velib de fidélité de type **VelibCard** (voir en 2.2.4)
- une **UserBalance** qui est une classe que nous avons créée pour contenir toutes les informations relatives aux statistiques d'un utilisateur (plus de détails dans le prochain paragraphe).
- un **Ride** qui est une classe que nous avons créée pour contenir toutes les informations relatives à la course à vélo que l'utilisateur est en train d'effectuer (plus de détails dans le prochain paragraphe).

- une boîte de messagerie **ArrayList<Message>** qui contiendra les notifications de l'utilisateur s'il souscrit à une station. Nous détaillons un peu plus loin la classe **Message** que nous avons créée pour contenir les notifications.
- une **ArrayList<Observable>** qui est la liste des stations observées par l'utilisateur.
- un dictionnaire ordonné **ConcurrentSkipListMap <Timestamp, Ride>** gardant la trace de chaque course effectuée par l'utilisateur avec pour clé le temps de départ et comme valeur la course **Ride**.

UserBalance

Cette classe donne une vue sur les statistiques de l'utilisateur liée à l'utilisation d'un système Velib. Elle est caractérisée par quatre attributs :

- private **int** numberOfRides : le nombre total de courses effectuées par l'utilisateur
- private **Duration** totalTime : le temps total passé sur un vélo, stocké sous forme de **Duration**
- private **Double** totalCharges : le coût total des courses effectuées par l'utilisateur
- private **Duration** totalTimeCredit ; le total de temps gagné par un utilisateur après avoir déposé ses vélos sur des stations Plus.

Initialement nous stockions tous ces attributs dans la classe User, mais pour des raisons de clarté et du principe Open Close, nous avons trouvé beaucoup plus pratique de créer une classe dédiée au stockage des données statistiques de l'utilisateur. Chaque attribut est accessible par un Getter. Nous avons bien pris soin de mettre à jour comme il se doit chaque attribut à chaque fois qu'un utilisateur effectue une opération dans le système Velib.

Ride

Cette classe sert à stocker toutes les informations relatives à une course afin de les garder de manière pratique dans les historiques des utilisateurs. Cette classe a pour attributs :

- la date de départ de la course sous forme de **Timestamp**
- la date de fin de la course sous forme de **Timestamp**
- la durée de la course sous forme de **Duration**. Bien sûr on peut la connaître en soustrayant les dates de départ et d'arrivée mais pour des raisons de praticité on en fait un attribut.
- la station où l'utilisateur a pris son vélo
- la station où l'utilisateur a déposé son vélo

- le vélo que l'utilisateur a loué
- le type du vélo que l'utilisateur a loué
- le coût de la course, de type **Double**
- le temps total de crédit gagné par l'utilisateur, de type **Duration**

Nous sommes contents de la création de cette classe car elle est très claire et permet de stocker efficacement les actions de l'utilisateur. Au début nous avons du mal à imaginer comment stocker l'ensemble des informations pertinentes pour les statistiques demandées par le sujet et gardions seulement les dates où l'utilisateur louait ou déposait un vélo. Finalement, la possibilité de stocker en valeur dans le dictionnaire précisément chaque course s'est imposée à nous au travers de la création de **Ride**.

PlannedRide

Sous classe de **Ride**, **PlannedRide** est une classe représentant un trajet planifié. Nous avons utilisé le **Strategy Design Pattern** pour calculer le chemin à emprunter selon les préférences de l'utilisateur. Cette classe fait appel aux classes **TripPreference**, **RecalculatePath**, **ShortestPath** et **FastestPath** développées en 2.3.2.

La classe possède comme attributs :

- private **TripPreference** preference : contient les préférences de l'utilisateur dans la planification de son voyage. En fonction des entrées lors de l'initialisation de **PlannedRide**, on affecte de préférence un objet des sous-classes **FastestPath**, **ShortestPath** ou **RecalculatePath**.
- private **GPS** departure : représente la position de départ (généralement la position de l'utilisateur).
- private **GPS** arrival : représente la position d'arrivée de l'utilisateur.
- private **Network** network : le réseau Vélib dans lequel est planifié le voyage.
- private **boolean** plus : true si l'utilisateur préfère déposer son vélo dans une station Plus, false sinon.
- private **boolean** uniformity : true si l'utilisateur préfère préserver l'uniformité des stations du réseau, false sinon.
- private **boolean** fastest : true si l'utilisateur veut avoir le chemin le plus rapide, false sinon.
- private **boolean** alreadyHaveBicycle : permet de savoir si l'utilisateur a déjà un vélo. Si ce booléen vaut true, preference recevra un objet de la classe **RecalculatePath**.
- private **Station[]** path : contient le chemin planifié (station de départ et station d'arrivée).

MessageBox

La boîte de messagerie de l'utilisateur est en fait codée sous la forme d'une **ArrayList<Message>**. Nous avons ainsi créé la classe **Message** qui a pour attributs :

- un booléen indiquant si le message a été lu (*true*) ou non (*false*)
- un **String** représentant le contenu du message.

Ces messages permettent de notifier à l'utilisateur un changement d'état d'une station à laquelle un utilisateur a souscrit.

2.2.2 Le système des cartes

Chaque utilisateur doit disposer d'une carte de crédit lui permettant de payer ses locations de vélo. En plus d'une carte de crédit, un utilisateur peut avoir une carte de fidélité Vélib qui peut être soit une carte Vlibre soit une carte Vmax. La possession d'une carte de fidélité entraîne des modification dans le calcul du coût d'une location. C'est pour cela que nous avons opté pour un Visitor Pattern pour effectuer ce calcul.

L'interface Card

Les classes implémentant cette interface sont les éléments visités lors du calcul du coût d'une location. La seule méthode incluse dans cette interface est la célèbre méthode *accept*.

L'interface CardVisitor

Les classes implémentant cette interface sont les visiteurs qui permettent de visiter les classes implémentant l'interface Card et ainsi de calculer le coût d'une location en fonction du type de carte que possède l'utilisateur. Dedans on retrouve trois méthodes visit qui correspondent aux trois classes implémentant l'interface Card et qui représentent les différents types de cartes (Vlibre, Vmax ou Carte de crédit i.e. pas de carte de fidélité.)

2.2.3 La carte de crédit CreditCard

Cette classe définit les cartes de crédit que les utilisateurs utilisent pour payer les courses. Elle implémente l'interface **Card** afin de pouvoir utiliser le pattern **Visitor** en implémentant *accept*.

Cette classe a pour attributs :

- un identifiant unique de type **int**
- l'utilisateur **User** à qui elle appartient

2.2.4 Les cartes Velib

Chaque utilisateur peut posséder une carte Velib pour utiliser le réseau. Cette carte peut permettre une diminution des coûts et favoriser la fidélité avec la politique d'offrir du temps de crédit si l'on ramène un vélo dans une station Plus. L'utilisateur peut donc souscrire à une carte Vlibre ou Vmax. Nous avons donc fait le choix d'implémenter une super-classe dont ces deux dernières cartes vont hériter.

VelibCard

La classe **VelibCard** a pour attributs :

- l'utilisateur **User** à qui la carte appartient
- le temps de crédit obtenu en ramenant des vélos dans des stations Plus, de type **Duration**

VlibreCard

La classe **VlibreCard** est une sous-classe de **VelibCard**, implémente l'interface **Card** et a pour attributs :

- un unique identifiant de type **int**
- le coût de type **double** de la première heure de vélo pour un vélo mécanique, fixé à 0.0
- le coût de type **double** de la première heure de vélo pour un vélo électrique, fixé à 1.0
- le coût de type **double** des heures suivantes à vélo pour un vélo mécanique, fixé à 1.0
- le coût de type **double** des heures suivantes à vélo pour un vélo électrique, fixé à 2.0

VmaxCard

La classe **VmaxCard** est une sous-classe de **VelibCard**, implémente l'interface **Card** et a pour attributs :

- un unique identifiant de type **int**
- le coût de type **double** de la première heure de vélo, fixé à 0.0 indépendamment du type de vélo
- le coût de type **double** des heures suivantes à vélo, fixé à 1.0 indépendamment du type de vélo

2.3 Les classes liées à l'utilisation du réseau Velib

2.3.1 La classe GPS

Cette classe sert à représenter des positions GPS. Pour des raisons de simplicité tout au long de ce projet, nous avons considéré qu'une position GPS était déterminée par deux réels i et j de type **Double** ce qui simplifie les calculs de distance. Nous avons néanmoins fait en sorte que si un jour l'on décide de coder les positions GPS comme des vraies positions GPS, il y ait seulement des modifications à effectuer dans la classe GPS, au niveau des attributs et de la méthode permettant de calculer la distance euclidienne entre deux points.

2.3.2 Les classes de planification de chemins

L'utilisateur doit pouvoir planifier un circuit entre une position de départ et une position d'arrivée. Pour cela nous devons lui indiquer à quelle station aller chercher son vélo et à quelle station déposer son vélo tout en respectant certaines politiques de choix du parcours. Pour ce faire, nous avons utilisé une Strategy Pattern.

TripPreference

Une abstract class qui contient la méthode *abstract setPath* calculant le circuit que doit emprunter l'utilisateur en fonction de ses préférences. **TripPreference** possèdent trois sous-classes implémentant *setPath* : **FastestPath**, **ShortestPath** et **RecalculatePath**.

TripPreference contient aussi les méthodes *getDepartures* et *getArrivals* permettant d'appliquer les contraintes d'uniformisation et la préférence des stations plus afin de déterminer les ensembles de stations de départ et d'arrivées utilisés par **FastestPath**, **ShortestPath** et **RecalculatePath**

ShortestPath

La classe **ShortestPath** est une sous-classe de **TripPreference** et permet d'implémenter *setPath* et utilise *getDepartures* et *getArrivals* pour calculer le chemin le plus court.

FastestPath

La classe **ShortestPath** est une sous-classe de **TripPreference** et permet d'implémenter *setPath* et utilise *getDepartures* et *getArrivals* pour calculer le chemin le plus rapide.

RecalculatePath

La classe **RecalculatePath** hérite de la classe **TripPreference**. On utilisera **RecalculatePath** lorsqu'un utilisateur souhaite déterminer une nouvelle station d'arrivée alors qu'il avait déjà un vélo. En effet, on fera appel à cette classe lorsque par exemple la station d'arrivée est désormais pleine ou hors ligne ce qui signifie que l'utilisateur ne pourra donc plus y déposer son vélo. On lui propose donc de déterminer une nouvelle station d'arrivée selon les diverses politiques de choix proposées dans **TripPreference**. La notification en pleine course est rendue possible à travers l'implémentation d'un Observer Pattern dont on parlera plus en détail en 3.4.

3 L'architecture utilisée

3.1 Le diagramme UML

Nous avons créé deux diagrammes UML. Le premier avec ObjectAID qui est un plug-in d'Eclipse. Le fichier UML est disponible dans le package myVelib. Il est exhaustif mais difficile à lire. Nous avons créé une deuxième version avec l'environnement Papyrus, plus claire.

3.2 La gestion du temps :

Le sujet introduit des problématiques liées à la gestion du temps. La première question qui s'est posée est : Comment modéliser le temps et comment le faire avancer ? Nous avons répondu à cette question en implémentant deux attributs dans la classe **MyVelib** :

- private **Timestamp** iniTim : qui représente l'origine du temps correspondant au temps d'initialisation. Lors de la construction d'un objet **MyVelib**, iniTim reçoit le temps système.
- private **Timestamp** currentTim : qui représente le temps de la simulation.

Pour faire avancer le temps, nous avons codé une méthode *advanceTime* qui prend en argument une durée de type **Duration** afin de faire avancer currentTim de la durée saisie.

Concrètement, les seules dates qui sont importantes dans cette modélisation du système Vélib, c'est les dates des opérations de location et de retour des vélos. Par conséquent, ce sont les fonctions/commandes rentbike et returnbike qui ont une influence sur le temps. En particulier, à chaque fois qu'un utilisateur rend son vélo, il va faire avancer le temps d'une durée égale à la durée de son dernier voyage.

Une fois ce choix fait, il fallait de garder une trace temporelle des diverses opérations effectuées dans le réseau, par exemple la location et le retour de vélos. Nous avons cherché dans la doc de JAVA, les structures les plus adaptées pour gérer proprement les événements et nous avons découvert la structure du dictionnaire ordonné **ConcurrentSkipListMap**.

Ce dictionnaire présente l'avantage de permettre le respect du stockage des valeurs selon des clés respectant un ordre chronologique. Les éléments de cette collection sont

triés selon leur ordre naturel en implémentant l'interface **Comparable** ou en utilisant une instance de type **Comparator** fournie en paramètre du constructeur de la collection. Les opérations de la classe **ConcurrentSkipListMap** sont *thread-safe*.

La classe **ConcurrentSkipListMap** présente plusieurs caractéristiques :

- elle permet la modification concurrente de la collection sans la bloquer dans son intégralité
- elle est optimisée pour les opérations de lectures qui sont non bloquantes
- par défaut les éléments sont triés selon leur ordre naturel ou selon l'ordre défini par l'instance de **Comparator** associée à la collection
- l'utilisation de **null** n'est pas possible ni pour la clé ni pour la valeur d'un élément

Nous avons décidé que tout instant t serait du type **Timestamp**, ce qui permet d'utiliser toutes les méthodes liés à ce type. La classe **java.sql.Timestamp** encapsule un instant exprimé en millisecondes et des informations permettant une expression de cet instant avec une précision à la nanoseconde. L'usage de **Timestamp** permet aussi l'usage de **Duration** pour représenter les durées, notamment utiles pour des questions de calcul de coût et de statistiques.

Ainsi pour tout ce qui création d'historiques d'utilisation des divers objets du système, nous utilisons une **ConcurrentSkipListMap** avec pour clés des **Timestamp** et pour valeurs associées les informations que l'on souhaite garder dans l'historique.

3.3 Visitor pattern

Le Visitor Pattern a été utilisé pour le calcul du coût d'une location de vélo.

Le calcul du coût dépend du type de carte que possède l'utilisateur. Dans cette version du projet, il fallait donc implémenter 3 méthodes différentes. Le Visitor Pattern permet d'éviter de polluer une des classes **User** ou **Station** avec ces méthodes là, en plus de garantir l'approche Open/Closed.

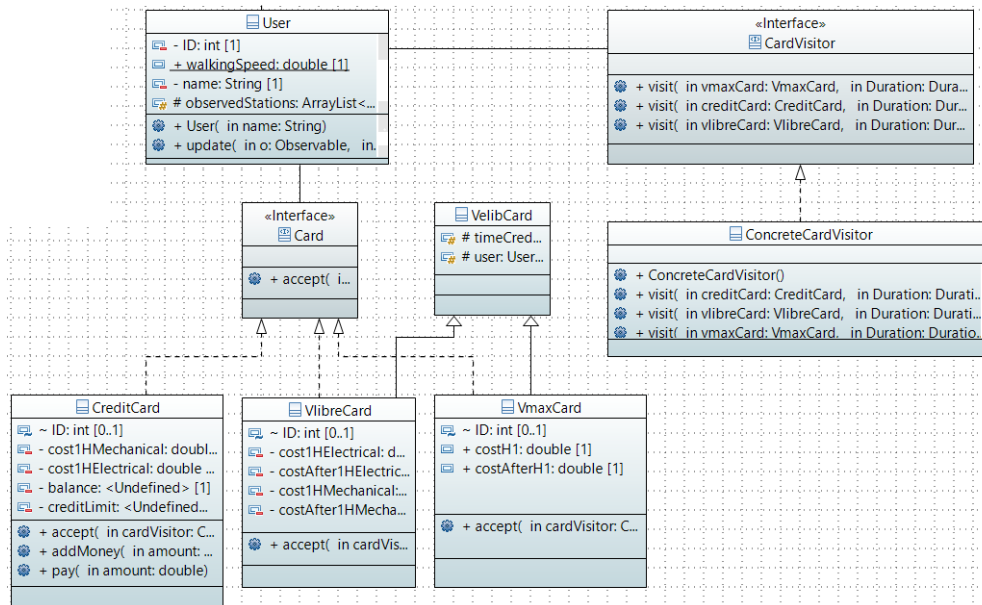


FIGURE 3.1 – Diagramme UML du **Visitor Pattern**

Supposons qu'on veuille ajouter un nouveau type de carte. Il suffirait d'ajouter une sous-classe à **VelibCard** et d'implémenter dans **ConcreteCardVisitor** la méthode *visit* adaptée à la nouvelle carte. Supposons qu'on veuille faire un tarif préférentiel pour certains clients, on peut aisément rajouter une classe implémentant **CardVisitor** et qui aura à son tour plusieurs implémentations différentes de *visit*.

3.4 Observer pattern

L'usage d'un système de locations de vélo conduit naturellement à offrir la possibilité aux utilisateurs d'être au courant en temps réel de l'état des stations qui l'intéressent pour louer ou déposer un vélo. Nous avons donc implémenté le pattern **Observer** pré-implémenté en JAVA. Chaque utilisateur peut devenir un **Observer** d'une station **Observable**. Ainsi, on bénéficie des fonctions pré-implémentées en JAVA pour notifier les utilisateurs qui ont souscrit à une station et souhaite être au courant de ses changements de statut.

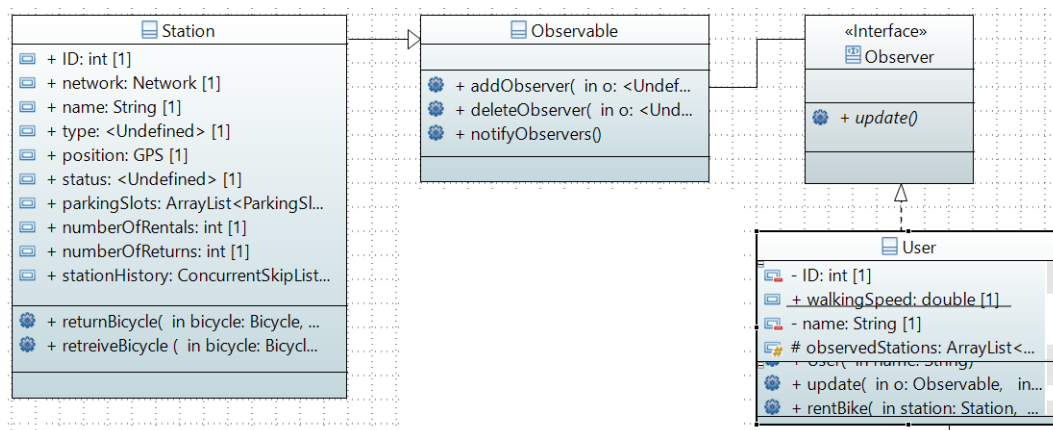


FIGURE 3.2 – Diagramme UML du **Observer Pattern**

3.5 Strategy pattern

L'usage de ce design pattern permet de rendre le code plus lisible pour tout ce qui est planification de chemin. En effet, il y a plusieurs politiques de choix possibles pour la détermination des stations de départ et d'arrivée et le Strategy Pattern permet de répondre à cette problématique et présente l'avantage de permettre facilement à l'avenir la création de nouvelles politiques de choix.

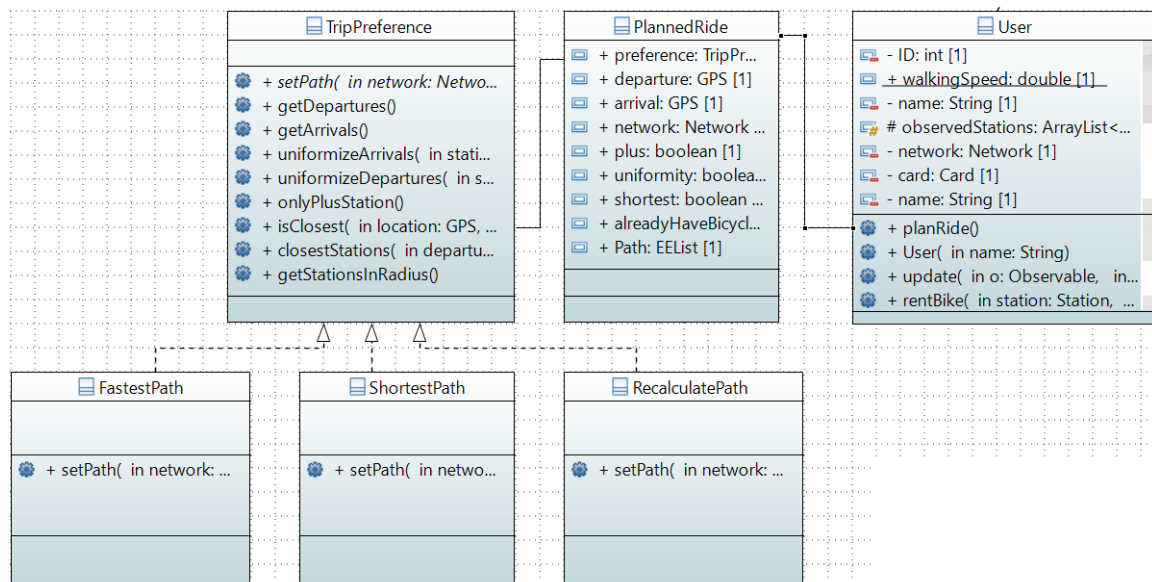


FIGURE 3.3 – Diagramme UML du **Strategy Pattern**

3.6 CLUI et Factory pattern

Nous avons utilisé une méthode assez simple et classique pour créer notre CLUI. La classe CLUI prend en argument un objet **CommandFactory**, un parser de la classe **CmdParser** et un réseau myVelib de la classe **MyVelib**.

Lorsqu'on lance la méthode *execute()* du CLUI, on crée un scanner qui va prendre les entrées de l'utilisateur. Chaque entrée est une chaîne de caractère contenant la commande et les arguments que l'utilisateur vient de saisir. Le parser va prendre cette chaîne de caractère et lui appliquer la méthode *parse(String line)* qui permet de séparer la chaîne de caractère saisie par l'utilisateur en deux parties :

- La première partie est le premier que contient line et qui représente le nom de la commande cmd. Ce mot est stocké dans l'attribut cmd de **CmdParser**.
- La deuxième partie est une `ArrayList<String>` contenant la liste des arguments. Cette liste est stockée dans l'attribut args de **CmdParser**.

Une fois que nous avons la commande et la liste des arguments. La commande *interpret()* de **CLUI** va faire appel à **CommandFactory** pour créer la commande souhaitée (une des sous-classe de la classe **Command**). L'exécution de la méthode *execute()*, héritée par toutes les sous-classes de **Command** va exécuter la commande souhaitée.

Tout ce processus est répété dans une boucle **while** qu'on peut arrêter en tapant *exit*.

Afin de créer de nouvelles commandes avec un minimum de code à modifier nous avons implémenté le Factory Pattern. Nous avons donc créé une super-classe **Command** possédant deux méthodes :

- *execute()* : permet l'exécution des fonctionnalités spécifiques de la commande
- *check()* : permet de vérifier que l'utilisateur a bien donné le bon nombre d'arguments.

Nous pouvons ensuite créer aisément de nombreuses sous-classes représentant les différentes fonctionnalités de la CLUI. Il suffit d'indiquer alors dans la classe **CommandFactory** que nous avons implémenté une nouvelle commande.

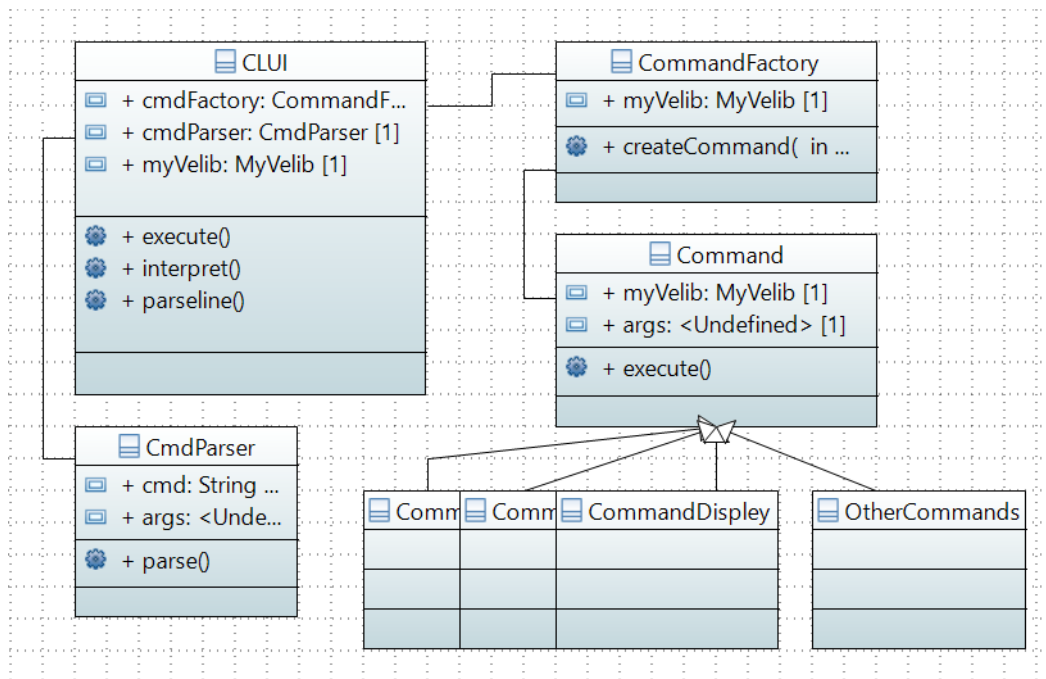


FIGURE 3.4 – Diagramme UML du package CLUI. Nous n'avons représenté que quelques sous-classe de **Command** pour éviter de surcharger.

4 Core

Dans ce chapitre, nous traitons de manière explicite l'ensemble de nos méthodes implémentées pour permettre le fonctionnement d'un réseau Velib. Nous ne parlerons pas des Constructeurs ni des Getters / Setters sauf cas particuliers.

4.1 Création de l'architecture Velib en elle-même

Nous allons décrire dans cette section l'ensemble des méthodes que nous avons mises en place pour permettre la création d'un système Velib.

4.1.1 Ajouter stations, slots et vélos dans le réseau

Les méthodes de Network

Nous avons défini des méthodes de **Network** permettant de faciliter la création de tests.

- *Network.stationList(int n)* : cette méthode retourne une liste de n stations réparties aléatoirement dans le carré de taille $n*n$ dont le coin inférieur droit a pour coordonnées $[0,0]$.
- *Network.stationWithSlotList(ArrayList<Station> stations, int m)* : cette méthode permet d'ajouter m parking slots de manière uniforme (à 1 slot près) entre les stations et de retourner la liste de ces stations.
- *Network.stationWithBicycles(ArrayList<Station> stations, int numberOfSlots, int percentageOfOccupation, int percentageOfMechanical)* : cette méthode permet de retourner la liste des stations donnée en arguments après avoir rajouté un certain nombre de vélos dans les emplacements des stations selon des contraintes spécifiées de pourcentage de slots à remplir et de pourcentage de vélos mécaniques.

Ces méthodes permettent donc de créer facilement des listes de stations remplies afin de tester des scénarios d'usage.

D'autres méthodes d'ajout d'objets

Nous avons également créé des méthodes permettant d'ajouter une station à un network, un parking slot à une station, d'assigner un vélo à un parking slot, etc.

4.2 Planifier une course

4.2.1 Les méthodes de `TripPreference` pour trouver les stations les plus proches

`TripPreference.isClosest`

La méthode `ArrayList<Station> isClosest(GPS position, ArrayList<Station> stations)` permet de retourner la liste des stations les plus proches d'une position GPS donnée. Nous avons pris soin, étant donné que les nombres flottants peuvent être arrondis lors des calculs de distance, de retourner une liste des stations à la distance la plus proche plus ou moins 10^{-5} .

`TripPreference.getStationsInRadiusPercent`

La méthode `ArrayList<Station> getStationsInRadiusPercent(ArrayList<Station> stations, GPS position, Double percent, boolean onlyPlus)` retourne une liste des stations situées dans un rayon de pourcentage donné de la distance entre une position GPS et la station de laquelle elle est la plus proche.

On donne donc en argument :

- la liste des stations dont on veut déterminer les stations autour dans un rayon donné lié à un pourcentage de distance entre ces stations et la position GPS donnée en argument
- une position GPS, de type **GPS**
- un pourcentage de distance, de type **Double**
- un **boolean** donnant la possibilité de ne prendre en compte que les stations de type Plus si on le fixe à *true*.

`TripPreference.closestStations`

La méthode `Station[] closestStations(ArrayList<Station> departures, ArrayList<Station> arrivals)` retourne le couple de stations de départ et d'arrivée dont la distance est la plus petite.

On donne donc en argument :

- la liste des stations de départ possibles
- la liste des stations d'arrivée possibles

4.2.2 Les méthodes de `TripPreference` permettant de déterminer le chemin

Il y a en tout 5 choses à prendre en compte lors du calcul de chemin :

- la préférence d'uniformité des stations.
- la préférence des stations d'arrivée plus.
- la préférence du type de chemin : le plus court ou le plus rapide.
- le fait que l'utilisateur possède déjà un vélo ou pas.

La contrainte d'uniformité va avoir un rôle dans la détermination des stations de départ et les stations d'arrivée. Une fois que nous avons la position de départ (celle de l'utilisateur en général), il suffit de trouver la distance qui sépare cette position de station qui lui est la plus proche, puis de prendre toutes les stations dans un rayon valant 105% cette distance et sélectionner les stations qui permettent d'uniformiser l'ensemble des stations. On fait de même pour appliquer cette contrainte aux stations d'arrivées.

La contrainte de préférence des stations Plus n'a d'effet que sur la sélection de la station d'arrivée. A partir de la position d'arrivée (la destination de l'utilisateur), on trouve la distance qui sépare cette position de la station qui lui est la plus proche à puis on cherche les stations plus dans un rayon de 110%.

Les contraintes de chemin le plus court ou le plus rapide n'interviennent qu'une fois que nous avons déterminé l'ensemble des stations de départ et celui des stations d'arrivée après avoir appliqué les contraintes d'uniformisation et la préférence des stations plus.

Nous avons donc commencé par coder les méthodes qui permettent de déterminer ces ensembles qui seront ensuite utilisés pour calculer le chemin le plus court ou le plus rapide.

`TripPreference.getDepartures`

La méthode *`ArrayList<Station> getDepartures(Network network, GPS departure, boolean uniformity)`* retourne la liste des stations de départ possibles remplissant le critère d'uniformité si l'utilisateur le souhaite. Autrement, ce seront les stations les plus proches qui seront proposées.

On donne donc en argument :

- le réseau **Network** que l'utilisateur veut utiliser
- la position **GPS** d'où l'on souhaite partir
- le **boolean** valant *true* si l'on souhaite respecter la politique d'uniformité

Afin de proposer la politique de choix d'uniformité, on utilise la méthode *`TripPreference.uniformiseDepartures`* décrite en 4.2.2.

TripPreference.getArrivals

La méthode *ArrayList<Station> getArrivals(Network network, GPS arrival, boolean uniformity, boolean plus)* retourne la liste des stations d'arrivée possibles remplissant le critère d'uniformité et de proposer d'aller dans une station Plus si l'utilisateur le souhaite.

On donne donc en argument :

- le réseau **Network** que l'utilisateur veut utiliser
- la position **GPS** où l'on souhaite arriver
- le **boolean** *uniformity* valant *true* si l'on souhaite respecter la politique d'uniformité
- le **boolean** *plus* valant *true* si l'on souhaite favoriser le retour dans une station plus

Afin de proposer la politique de choix d'uniformité, on utilise la méthode *TripPreference.uniformiseArrivals* décrite en 4.2.2.

TripPreference.uniformiseDepartures

La méthode *ArrayList<Station> uniformiseDepartures(ArrayList<Station> stations, GPS departure, Bicycle.BicycleType bType)* retourne la liste des stations qui ont le plus de vélos du type désiré par l'utilisateur parmi la liste de stations donnée en arguments. Une telle politique de choix de stations permet de diminuer les moments où une station se retrouvera à cours de vélos car l'on favorise les stations où il y a le plus de vélos à récupérer.

On donne donc en argument :

- une liste des stations, a priori celles situées dans un rayon compris dans une distance de 105% de celle qui sépare l'utilisateur et la station la plus proche de lui. On cherche parmi ces stations celles qui ont le plus grand nombre de vélos du type désiré.
- la position GPS de l'utilisateur
- le type du vélo que l'utilisateur souhaite louer

TripPreference.uniformiseArrivals

Cette méthode est le pendant de la méthode précédente pour les stations d'arrivées. La méthode *ArrayList<Station> uniformiseArrivals(ArrayList<Station> stations, GPS arrival)* retourne la liste des stations qui ont le plus d'emplacements libre parmi les stations données en entrée. Une telle politique de choix de stations permet de diminuer les moments où une station se retrouvera à cours d'emplacements libres car l'on favorise les stations où il y a le plus de disponibilité pour déposer un vélo.

On donne donc en argument :

- une liste des stations, a priori celles situées dans un rayon compris dans une distance de 105% de celle qui sépare la destination finale de l'utilisateur et la station la plus proche de cette destination. On cherche parmi ces stations celle ou celles (en cas d'égalité) qui ont le plus grand nombre d'emplacements libres.
- la position GPS de la destination de l'utilisateur

ShortestPath

La classe **ShortestPath** décrite précédemment en 2.3.2 est une sous-classe de **TripPreference** permettant de déterminer une course respectant la politique du chemin le plus court.

FastestPath

La classe **FastestPath** décrite précédemment en 2.3.2 est une sous-classe de **TripPreference** permettant de déterminer une course respectant la politique du chemin le plus rapide.

4.2.3 Replanifier une course

Lorsque l'utilisateur est déjà en train d'effectuer une course et qu'il souhaite recalculer une station d'arrivée, il a cette possibilité grâce à la sous-classe **RecalculatePath** de **TripPreference**, décrite en 2.3.2. Cette classe intervient par exemple quand l'utilisateur a planifié une course et que la station d'arrivée devient hors ligne.

4.3 Effectuer la course

4.3.1 Commencer la location d'un vélo

Cette fonctionnalité est l'une des plus importantes de l'architecture du code. Nous avons codé la méthode void *User.rentBike(Station s, Bicycle.BicycleType bType, Timestamp t)* qui permet à un utilisateur de louer un vélo d'un type choisi dans une station de son choix à un instant donné.

Gestion des exceptions

On commence par gérer les exceptions que peut déclencher une telle méthode. Nous en avons redéfini trois : **AlreadyHasABikeException**, **OfflineStationException** et **NoBikesAvailableException**, déclenchées respectivement lorsque l'utilisateur essaie de louer un vélo alors qu'au même instant il en a déjà loué un, lorsqu'il essaie de louer un vélo dans une station hors ligne, et lorsqu'il n'y a plus de vélos disponibles dans la station.

Sélection d'un vélo

S'il n'y a pas d'exception, l'utilisateur peut donc louer un vélo. Nous avons implémenté la méthode **Bicycle** *Station.retrieveBicycle(BicycleType bType, Timestamp t)* qui sélectionne un vélo pour l'utilisateur et s'occupe de libérer l'emplacement où il était.

Mise à jour des divers attributs

Après récupération du vélo, on met à jour l'historique de l'utilisateur à l'aide de la méthode **User.updateUserHistory(Timestamp t, Ride ride)** en prenant soin de bien spécifier les attributs de ride. On met aussi à jour l'historique de la station concernée grâce à la méthode **Station.addEntryToStationHistory(Timestamp t)**. On met également à jour les statistiques de l'utilisateur et de la station.

4.3.2 Déposer un vélo

Cette fonctionnalité va de pair avec la précédente. Nous avons codé la méthode **void User.returnBike(Station s, Timestamp t)** qui permet à un utilisateur de déposer son vélo dans une station choisie à un instant donné.

Gestion des exceptions

On commence par gérer les exceptions que peut déclencher une telle méthode. Nous en avons redéfini deux : **OfflineStationException** et **NoAvailableFreeSlotsException**, déclenchées respectivement lorsque l'utilisateur essaie de déposer un vélo dans une station hors ligne, et lorsqu'il n'y a plus d'emplacements disponibles dans la station.

Dépôt du vélo

S'il n'y a pas d'exception, l'utilisateur peut donc déposer son vélo. Nous avons implémenté la méthode **void Station.returnBicycle(Bicycle bicycle, Timestamp t)** qui s'occupe de trouver l'emplacement pour déposer le vélo puis de l'y déposer.

Mise à jour des divers attributs

Après dépôt du vélo, on met à jour l'historique de la station concernée grâce à la méthode **Station.addEntryToStationHistory(Timestamp t)**. On met également à jour les statistiques de l'utilisateur et de la station.

Calcul de la durée de la course

En utilisant, la classe **Duration** préimplémentée en JAVA, on calcule la durée de la course, ce qui va nous servir pour calculer son coût pour l'utilisateur et pour mettre à jour **UserBalance**.

Calcul du coût de la course

On doit prendre en compte le fait que la station soit de type Normal ou Plus, auquel cas on doit commencer par créditer du temps sur la **VelibCard** de l'utilisateur. Ensuite, on doit calculer le temps total à facturer en prenant en compte le crédit de temps disponible pour l'utilisateur.

On peut maintenant faire payer la course. On utilise le pattern Visitor, détaillé en 3.3.

Enfin, on met à jour les attributs de la carte de l'utilisateur, de sa course ride, ainsi que ceux de **UserBalance**.

Pour finir, l'utilisateur quitte la liste d'observateurs de la station dans laquelle il a rendu son vélo.

4.4 Les fonctions statistiques et de tri

4.4.1 Autour des utilisateurs

Nous avons décrit en 2.2.1 la classe **UserBalance** qui donne des statistiques intéressantes autour d'un utilisateur donnée.

4.4.2 Autour des stations

Station.getNumberOfReturns()

Cette méthode permet de retourner le nombre de retours dans une station. Pour respecter le principe KISS, nous avons tout simplement affilié à chaque station un attribut `numberOfReturns` de type **int** donnant le nombre total de retours de vélos dans la station depuis sa création. On incrémente ce nombre lorsqu'on utilise la méthode *User.returnBike(Station s, Timestamp t)*.

Station.getNumberOfRentals()

Cette méthode permet de retourner le nombre de locations dans une station. Pour respecter le principe KISS, nous avons tout simplement affilié à chaque station un attribut `numberOfRentals` de type **int** donnant le nombre total de locations de vélos depuis la station depuis sa création. On incrémente ce nombre lorsqu'on utilise la méthode *User.rentBike(Station s, Bicycle.BicycleType bType, Timestamp t)*.

Network.mostUsedStations()

Cette méthode permet de retourner la liste des stations du Network sous forme d'**ArrayList<Station>** triée par ordre croissant selon le nombre total d'opérations effectuées auprès d'elle (somme des locations et des retours de vélos).

Nous avons utilisé la commande *Collections.sort(stationsToSort, Station.mostUsedComparator)* pour trier les stations. Nous avons donc dû implémenter le **Comparator mostUsedComparator** dans la classe **Station**. Ce **Comparator** permet d'introduire une relation d'ordre reposant sur le nombre total d'opérations effectuées dans les stations.

Network.leastOccupiedStations(Timestamp t1, Timestamp t2)

Cette méthode permet de retourner la liste des stations du Network sous forme d'**ArrayList<Station>** triée par ordre décroissant selon le taux d'occupation dans l'intervalle $[t_1, t_2]$ défini comme $\frac{\sum_{i=0}^N t_i^\Delta}{\Delta \cdot N}$ où $\Delta = t_2 - t_1$.

Implémentation d'un Comparator Nous avons utilisé la commande *Collections.sort(stationsToSort, Station.leastOccupiedComparator)* pour trier les stations. Nous avons donc dû implémenter le **Comparator leastOccupiedComparator** dans la classe **Station**. Ce **Comparator** permet d'introduire une relation d'ordre reposant sur le taux d'occupation de la station.

Calcul du taux d'occupation Nous avons également dû implémenter une méthode *Station.occupationRate(Timestamp t1, Timestamp t2)* capable de déterminer le taux d'occupation d'une station entre deux dates.

Cette méthode fait elle-même appel à une méthode de **ParkingSlot** : *ParkingSlot.occupationTime(Timestamp t1, Timestamp t2)* qui détermine le taux d'occupation d'un unique slot. Cette fonction a été très fastidieuse à écrire car faisant intervenir de nombreuses méthodes des types **Timestamp** et **Duration**, et pouvant soulever de nombreuses erreurs liées au choix de $t1$ et $t2$. Les tests ont été particulièrement utiles pour découvrir de nombreux points à changer. En effet, pour déterminer un taux d'occupation nous utilisons l'attribut **ParkingSlot.slotHistory** qui contient l'historique des états du slot, libre, occupé, cassé. Nous avons par exemple pris conscience du fait que lorsque l'on crée un slot pour des raisons de debuggage, nous devons aussi mettre de suite un état dans son historique associée à une clé t représentant la date à laquelle on crée ce slot.

5 Comment tester notre code

Dans la phase de développement du code, nous avons testé nos diverses méthodes avec des JUnit Test mais dès lors que nous avons codé notre CLUI il s'est avéré extrêmement plus pratique, plus rapide et plus riche de pouvoir tester notre code directement avec. C'est pourquoi nous conseillons de tester notre code avec les scénarios que nous proposons à partir de la CLUI.

5.1 Tester avec le CLUI

Pour utiliser notre CLUI, il faut exécuter la classe : `CLUI.CLUITest.java`.

Il est ensuite possible d'exécuter la commande *help* pour voir les commandes disponibles avec les arguments et les explications.

Nous proposons également des scénarios pré-implémentés qui démontrent l'utilisation de toutes les fonctionnalités que nous proposons. Ces scénarios se situent dans le dossier *simulations*. Pour les exécuter, lancer `CLUITest.java` et écrire la commande *runtest* `<path>` où `<path>` est le chemin du scénario à exécuter. Petite astuce, pour accéder au path de la simulation depuis Eclipse : faites un clic droit sur la simulation `.txt` et aller dans propriétés ou appuyer sur Alt+Entrée. Une fenêtre apparaît avec le chemin. Il suffit de faire un copier-coller !

Attention : nous préconisons de relancer complètement **CLUITest** avant chaque utilisation d'un scénario pour des questions d'ID que nous avons prévues qui ne seraient pas les mêmes si vous aviez déjà utilisé certaines commandes.

Nous exposons dans ce qui suit les simulations créées. Chaque simulation est commentée : les lignes commençant par des % sont les commentaires qu'on fait passer du `.txt` à la CLUI grâce à la commande `print`.

On vous encourage vivement à faire tourner ces simulations avant de commencer à utiliser le CLUI par vous-même.

5.1.1 testCore.txt

Dans cette simulation nous exposons les fonctions les plus basiques du code : louer un vélo et le rendre. Nous ne rentrons pas vraiment dans les détails dans cette simulation.

```

===== Station Infos =====
Station Name      : Station10
Station ID       : 10
Type             : Normal
Localisation      : x=0.042 , y=3.999
Status           : On Service
===== Station Occupation =====
Free Slots       : 8
Broken Slots     : 0
Occupied Slots   : 2
Nb Electrical Bikes : 2
Nb Mechanical Bikes : 0
===== Station Balance =====
Nb of Rentals    : 5
Nb of Returns    : 0
Nb of OPS        : 5
Occupation Rate  : 0.678

===== Parking Slots =====
Number | Status |
=====
0      | Occupied by electrical |
1      | Occupied by electrical |
2      | Free |
3      | Free |
4      | Free |
5      | Free |
6      | Free |
7      | Free |
8      | Free |
9      | Free |
=====

```

FIGURE 5.1 – Exemple d’un affichage de station.

5.1.2 testRentBikeReturnBike.txt

Dans cette simulation nous effectuons la location et le retour d’un vélo. Nous nous intéressons plus particulièrement à des actions susceptibles de faire bugger le code comme lorsqu’on essaye de louer un type de vélo dans une station qui ne contient pas de vélo, ou encore, lorsqu’on essaye de rendre un vélo dans une station pleine.

5.1.3 testCost.txt

Dans cette simulation nous testons le calcul des coûts des voyages. Nous simulons donc des voyages avec différents types de cartes, différents types de vélos et différentes durées. Cependant, dans cette simulation tous les vélos sont rendus à des stations de type **normal** pour ne pas faire intervenir les bonus credit time des cartes Vlibre et Vmax. Ce point est testé dans la simulation suivante.

5.1.4 testCreditTime.txt

Dans cette simulation nous testons le calcul des coûts en prenant en compte le time credit des utilisateurs possédants des cartes Vlibre et Vmax.

5.1.5 testPlanRide.txt

Dans cette simulation nous testons la commande planRide. Nous ne pouvons pas anticiper les résultats de cette commande parce que les stations du réseaux ont des positions aléatoires différentes d’une simulation à une autre. Néanmoins, nous simulons des

planifications de chemin avec toutes les combinaisons de préférences possible. Pour vérifier la cohérence des fonction, le testeur peut s’amuser à faire les calculs en s’aidant des display du network contenant le statut des stations et des utilisateurs.

5.1.6 testObserverPattern.txt

Dans cette simulation nous testons l’Observer Pattern impliquant la planifications de chemin. Étant donné que nous ne pouvons savoir quelles est la position de la station d’arrivée indiquée par la commande planRide (les positions des stations étant différentes d’une simulation à une autre) le testeur devra finir la simulation en utilisant la commande Offline Paris <StationID> pour changer le statu de la station d’arrivée en Offline. Vous aurez alors la possibilité de recalculer un nouveau chemin.

5.1.7 testTimeHandling.txt

Dans cette simulation nous exposons comment le temps a été gérer, et en particulier le rôle des commandes rentbike et returnbike dans l’évolution du temps. On vous encourage vivement à exécuter cette simulation ainsi que toutes les autres.

5.1.8 testSortStation.txt

Dans ce test nous initions un network de 10 stations dans lequel un unique utilisateur loue et retourne de nombreuses fois des vélos. Nous faisons donc la démonstration de nos fonctions de location qui permettent également de calculer le coût de la course. Nous démontrons également nos fonctions de classements de stations selon leurs statistiques.

Station Name	Station ID	Type	Location	Status	Nb of rentals	Nb of returns	Nb of ops	Occupation rate
Station8	8	Normal	x=3.565 , y=0.828	On Service	0	0	0	0.684
Station9	9	Plus	x=0.264 , y=0.322	On Service	0	0	0	0.684
Station10	10	Normal	x=0.437 , y=0.399	On Service	0	0	0	0.684
Station2	2	Normal	x=3.515 , y=1.243	On Service	0	1	1	0.861
Station3	3	Plus	x=3.273 , y=1.67	On Service	0	2	2	0.67
Station5	5	Normal	x=0.374 , y=1.268	On Service	0	2	2	0.67
Station6	6	Plus	x=1.691 , y=2.707	On Service	1	1	2	0.678
Station7	7	Normal	x=2.313 , y=3.857	On Service	3	5	8	0.684
Station4	4	Normal	x=1.031 , y=0.094	On Service	7	5	12	0.67
Station1	1	Normal	x=3.919 , y=0.639	On Service	18	13	31	0.859

FIGURE 5.2 – Exemple de réseau où l’on a classé les stations par ordre croissant selon leur utilisation.

5.1.9 A vous de jouer !

Nos simulations ont couvert une bonne partie des fonctionnalités de notre code. Vous pouvez explorer le code par vous même à travers la CLUI. Un descriptif des commandes et afficher en tapant help.

Welcome to the MyVelib Network Management Center. Enter help to get help !

help|

Commands available:

- setup <velibnetworkName>:

to create a myVelib network with given name and consisting of 10 stations each of which has 10 parking slots and such that stations are arranged on a square grid whose of side 4km and initially populated with a 75% bikes randomly distributed over the 10 stations
<velibnetworkName> is a String.

- setup <name> <nstations> <nslots> <sidearea> <nbikes>:

to create a myVelib network with given name and consisting of nstations stations each of which has nslots parking slots and such that stations are arranged on a square grid whose of side sidearea and initially populated with a nbikes bikes randomly distributed over the nstations stations
<name> is a String, <nstations> <nslots> <sidearea> <nbikes> are integers.

- addUser <userName> <cardType> <velibnetworkName>:

to add a user with name userName and card cardType (i.e. CreditCard if the user has no card) to a myVelib network velibnetworkName. Card types available :

Vlibre - Vmax - CreditCard

- offline <velibnetworkName> <stationID>:

to put offline the station stationID of the myVelib network velibnetworkName.

<stationID> is an integer. Use the command display <velibnetworkName> to access those IDs.

- online <velibnetworkName> <stationID>:

to put online the station stationID of the myVelib network velibnetworkName.

<stationID> is an integer. Use the command display <velibnetworkName> to access those IDs.

- rentBike <userID> <stationID> <bicycletype>:

to let the user userID renting a bike from station stationID (if no bikes are available it behaves accordingly). You have to chose a type for your bike :
electrical - mechanical

- returnBike <userID> <stationID> <time>:

to let the user userID returning a bike to station stationID at a given instant of time time (if no parking bay is available it behaves accordingly). This command displays also the cost of the rent.
<userID> is an integer. Use the command display <velibnetworkName> to access those IDs.<stationID> is an integer. Use the command display <velibnetworkName> to access those IDs.

- displayStation <velibnetworkName> <stationID>:

to display the statistics of station stationID of a myVelib network velibnetwork.

<stationID> is an integer. Use the command display <velibnetworkName> to access those IDs.

- displayUser <velibnetworkName> <userID>:

to display the statistics of user userID of a myVelib network velibnetwork.

<userID> is an integer. Use the command display <velibnetworkName> to access those IDs.

- sortStation<velibnetworkName> <sortpolicy>:

to display the stations in increasing order according to the sorting policy of user sortpolicy. Sort Policies available :

mostUsedStations - leastOccupiedStations

- display <velibnetworkName>:

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

- moveUser <userID> <networkName> <x> <y>:

to move the user to the location (x,y).

- planRide <userID> <x of destination> <y of destination> <preference> <uniformity> <plus> :

to make the user with <userID> plan a ride according to his preferences .

<stationID> is an integer. Use the command display <velibnetworkName> to access those IDs.<x of destination> and <y of destination> are doubles<preference> is a String. Preferences available are :Fastestpath - Shortestpath <uniformity> is a boolean (true or false). Type true if you want to uniformize the stations, else type false.<plus> is a boolean (true or false). Type true if you prefere plus stations, else type false.

- runtest <filepath>:

to execute a test scenario path. You must provide the path of the scenario.

- help:

get help with commands.

FIGURE 5.3 – Affichage de la commande help résumant toutes les commandes disponibles

5.2 Nos JUnit Test

5.2.1 Création d'un réseau MyVelib

Nous proposons dans le package tests le JUnit Test : **ClientTest**. Nous testons le scénario suivant :

1. Le client crée un nouveau réseau avec 50 stations, avec un total de 2000 emplacements pour vélos, dont 70% vont être remplis par des vélos et dont 70% de ces vélos sont mécaniques.
2. Le client ajoute deux utilisateurs du système Velib, un qui possède une carte Vmax (Anis) et l'autre non (Léonard).

5.2.2 Location d'un vélo

Dans ce même test **ClientTest** nous continuons avec le scénario suivant :

1. Léonard loue un vélo mécanique dans la station 5 à l'instant $t = 20$.
2. Léonard dépose le vélo dans la station 6 à l'instant $t = 10000000$.
3. Léonard doit payer 3 euros.

5.2.3 Simulation d'une course à vélo

Toujours dans ce même test **ClientTest** nous continuons avec le scénario suivant :

1. Léonard qui est situé au point (10,10) souhaite aller au point (45,45).
2. Il demande un plan de route.
3. Il récupère un vélo mécanique dans la station de départ à un instant donné et commence sa course.
4. La station d'arrivée prévue devient hors-ligne.
5. Léonard reçoit une notification et se voit proposer la possibilité de déterminer une nouvelle station d'arrivée.
6. Léonard demande une nouvelle station d'arrivée.

5.2.4 Calcul de statistiques

1. Nous n'avons pas effectué ce scénario d'usage mais allons le faire prochainement :
2. On crée un réseau comme celui décrit dans le scénario 1.
3. On fait effectuer de nombreuses courses à de nombreux utilisateurs.
4. On effectue les calculs de statistiques et on trie les stations selon leur taux d'occupation ou selon leur nombre d'utilisations.

5.3 Intérêt des tests

Nous avons pu constater au travers de tests simples nombre de petites erreurs de code : un signe \leq à la place d'un signe \leq , des variables mal initialisées, beaucoup de **NullPointerException** qui révélaient des défauts, etc.

6 Critique de notre modèle

6.1 Avantages

6.1.1 Réponse aux spécifications du sujet

Nous avons implémenté toutes les fonctionnalités demandées par le sujet. Nous avons très largement réfléchi à créer une architecture qui reflète les véritables systèmes de location de vélos et propose les fonctionnalités dont on a besoin dans la réalité. Nous nous sommes donnés beaucoup de mal pour une gestion du temps optimale qui puisse satisfaire les besoins d'un opérateur Velib dans la réalité. La gestion du temps a été une des plus grandes difficultés. L'implémentation du calcul de statistiques et de classements a demandé de nombreux tests et refontes de codes (voir en 3.2).

Nous avons été parfois plus loin que demandé en codant des commandes supplémentaires dans la CLUI comme par exemple *planRide* qui permet de mettre en valeur toutes nos méthodes de calculs de chemins.

6.1.2 Respect du principe *Open / Close* et usage de patterns

Nous nous sommes efforcés de respecter le principe ouvert/fermé du code. Nous avons découplé au maximum les diverses méthodes afin de rendre le code le plus clair et réutilisable possible. Nous avons séparé au maximum les classes. Nous détaillons dans les chapitres 2 et 4 nos diverses méthodes et classes.

Par exemple si l'acheteur de notre code souhaite un jour rajouter une nouvelle sorte de carte de fidélité, il suffira d'ajouter une sous-classe de **VelibCard** et d'implémenter le design pattern **Visitor** et tout le reste du code reste fonctionnel, même la méthode *User.returnBike* où il y a un calcul de coût de la course.

Toutes les constantes du problèmes sont codées sous forme d'attribut final. Pour changer le coûts des locations par exemple, il suffit juste de modifier quelques constantes et nous n'avons pas besoin de parcourir tout le code.

Par ailleurs si l'on souhaite proposer de nouvelles politiques pour choisir sa station de départ ou d'arrivée, l'usage du **Strategy pattern** permet la création facile de cette nouvelle politique et de l'incorporer dans le code avec un nombre de changements minime.

Notre CLUI est également conçue pour un ajout facile de nouvelles fonctionnalités grâce à l'usage du pattern **Factory**.

Nous nous sommes efforcés d'implémenter un maximum de *design patterns* afin de mieux les comprendre (pour plus de détails voir en 3.

6.1.3 Prise en compte de nombreuses exceptions

Nous avons pris le temps de penser aux diverses exceptions possibles et de les rattraper. Cette tâche nous a paru très fastidieuse mais se révèle gratifiante en ce qui concerne l'utilisabilité du code.

6.1.4 Prise en compte de la complexité de calculs

Le sujet ne demandait pas spécifiquement de réflexion sur les complexités des méthodes. Cependant, nous avons réfléchi l'architecture et les fonctionnalités de façon à ce que le code soit réellement utilisable par une entreprise de location de vélos. L'usage de **ConcurrentSkipListMap** permet de chercher des clés avec un coût en $O(\log(n))$ ce qui permet des calculs de statistiques plus rapides. Nous tenons également à jour au cours du temps les diverses données intéressantes pour les utilisateurs (crédit disponible, temps de crédit disponible) ou le réseau (nombre de locations dans telle station,...) ce qui permet d'accéder en temps constant aux diverses données.

6.1.5 Optimisation de l'affichage des résultats

Nous nous sommes efforcés de réaliser des `toString()` propres afin de rendre lisibles les affichages.

6.2 Défauts

6.2.1 Si on veut ajouter un nouveau type de vélo

En ce qui concerne les différentes sortes de vélo, nous n'avons pas toujours bien respecté Open/Close. Si l'on souhaitait rajouter une nouvelle sorte de vélo il y aurait des multiplications nombreuses à effectuer même si nous avons essayé de les réduire au minimum. En effet, nous effectuons parfois des disjonctions de cas *Electrical* / *Mechanical* dans nos méthodes ce qui en cas de prise en compte d'un nouveau vélo demanderait des réécritures.

6.2.2 Simulations avec des *threads*

Nous concevons nos propres scénarios d'usages même si nous pouvons lancer des simulations d'usages du système de location de manière aléatoire. Avec plus de temps nous aurions tenté l'utilisation de **thread** par exemple.

6.2.3 Création de la GUI

Nous aurions voulu créer une GUI et nous sommes documentés plus avant sur sa création mais nous avons fait le choix de passer davantage de temps pour travailler au mieux la CLUI et l'enrichir de fonctionnalités non demandées.

7 Répartition et méthode de travail

Nous nous sommes réparti le travail de manière très équitable et collective. Nous avons commencé par réfléchir ensemble sur l'architecture du réseau, le diagramme UML et les patterns à mettre en place. Cela nous a pris une semaine pour fixer correctement. Nous avons représenté sur une grande feuille de papier les classes, leurs méthodes et leurs attributs. Nous avons beaucoup appris de cette expérience, elle est encore plus importante qu'on ne le pensait. Nous nous sommes rendus compte que plus l'on est précis et plus l'on pense les détails dès le début, plus on peut coder vite et efficacement ensuite. Nous avons effectué au cours du temps de nombreux changements de stratégie d'implémentation parce que nous n'avions pas suffisamment poussé loin la réflexion de l'ensemble du code. Néanmoins, il y a de nombreuses problématiques dont nous n'avions pas encore idée au moment de designer l'architecture et c'est pour cela que nous avons fait des itérations sur l'architecture prévue, pendant que l'on codait.

Nous sommes finalement satisfaits de notre architecture, car nous la trouvons assez claire, logique, et respectant le principe Open / Close. Nous avons également fait l'effort d'implémenter un maximum de *pattern* dans une visée pédagogique. Nous les avons implémentés ensemble car nous souhaitions tous deux bien les comprendre.

Afin de coder ensemble, nous avons utilisé GitHub Desktop qui permet un partage facilité du code. Nous avons codé régulièrement, presque chaque jour et avons discuté également presque chaque jour par messagerie à propos de nos choix de code et de nos questions. Ce véritable travail d'équipe nous a permis une très bonne harmonie et collaboration ainsi que la possibilité d'avancer ensemble au même rythme. En ce qui concerne la répartition du code, nous avons quasiment tout codé de concert en collaborant étroitement et souvent en travaillant à côté mais :

- Anis s'est occupé seul de tout ce qui est planification des courses et des *toString()*.
- Léonard s'est occupé seul des fonctions de dépôt et retour des vélos ainsi que des statistiques.

Nous avons testé ensemble les classes et méthodes. Pour avancer dans le code nous avons commencé par coder le squelette architectural en préparant notamment l'utilisation de *patterns*. Nous avons ensuite codé les méthodes les plus simples pour terminer avec les plus rébarbatives. Nous avons gardé en tête tout du long l'utilisation du principe *KISS*. Nous avons également tenté d'adopter la démarche du *Test Driven Development* mais nous ne l'avons pas fait depuis le début, seulement à partir d'environ la moitié du projet. Nous avons appris de ça, que si cela peut sembler fastidieux alors qu'on a l'enthousiasme d'avancer dans le code en lui-même, nous sommes en fait bien contents à la fin d'avoir testé tout au fur et à mesure. Nous gardons cela à l'esprit pour un prochain projet.

Enfin, nous avons rédigé ensemble ce rapport en collaborant en temps réel grâce à Overleaf.

8 Conclusion

Nous comprenons tout l'enjeu de compléter un cours de Programmation Orientée Objet par la réalisation d'un projet, aussi chronophage soit-il. Nous avons particulièrement apprécié de coder un système que nous avons utilisé dans la réalité et que nous visualisons particulièrement bien. Nous avons découvert de nombreuses subtilités auparavant invisibles lorsque nous étions alors que de simples locataires de vélos. Nous nous sommes efforcés tout du long du projet, de proposer un code qui puisse être utilisé par un client souhaitant un logiciel pour gérer son système Velib et simuler son utilisation.

Pour notre expérience de codeur, nous retenons tout particulièrement que la conception d'une architecture claire et détaillée en amont permet un véritable gain de temps sur le long terme. L'usage du principe KISS guide également la réflexion. Nous retenons également qu'utiliser le *Test Driven Development* dès le début est un véritable atout sur la fin de projet.

En ce qui concerne notre conclusion personnelle, nous sommes très heureux de notre travail d'équipe qui a permis d'aboutir à une solution bien meilleure que celle qu'on aurait effectuée seuls. Nous avons bien vu tout l'intérêt qu'il y a de confronter sa vision stratégique de l'architecture du code avec celle de quelqu'un d'autre afin d'envisager des situations non prises en compte auparavant. Écrire un code est également une véritable aventure humaine, nous avons passé de très nombreuses journées à travailler, à des heures souvent tardives, mais nous nous sommes toujours soutenus.

Nous débutions tous les deux en programmation orientée objet et si nous avons de nombreuses difficultés au démarrage, nous avons clairement pu constater que notre aisance grandissait au fur et à mesure. Nous avons notamment développé un réel goût pour la réflexion autour des meilleures stratégies d'implémentation. Ce goût a même surpassé notre affection pour les bières si bien que nos amis se sont étonnés de ne pas nous voir pendant un mois. Nous avons été forcé d'avouer notre nouvelle addiction pour cette nouvelle drogue qu'est la POO.

Merci pour votre attention,

Anis et Léonard