IS1220 - Programmation Orientée Objet
CentraleSupélec

Rapport de Projet : My Vélib'

Encadrement

Paolo Ballarini

Arnault Lapitre

Anis Ben Said

Léonard Boussioux

25 Mars 2018

Table des matières

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. Notamment, la finalité est de fournir une interface en ligne de commande, voire une interface graphique, afin de pouvoir 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.

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.

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 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.

2.1.2 Station

Chaque station est caractérisée par :

- un nom de type **String**

- un unique identifiant de type **int**
- un type : normal ou plus. Nous utilisons une énumération **enum StationType-Normal, Plus** pour créer ces deux sortes de station.
- une position GPS, classe que l'on décrit en 2.3.1.
- un statut : remplie, disponible, hors ligne. Nous utilisons une énumération **enum Status Full, Available, Offline** pour créer ces trois sortes de statut.
- une liste d'emplacements de stationnement sous forme de **ArrayList<ParkingSlot>**. L'utilisation de **ArrayList** permet d'ajouter ou d'enlever facilement des **Parking-Slot**.
- une liste d'utilisateurs sous forme de **ArrayList<User>**. L'utilisation de **Array-List** permet d'ajouter ou d'enlever facilement des **Users**. Nous reviendrons en 3.4. Cette liste joue un rôle dans l'implémentation de l'Observer Design pattern qui permet de notifier un utilisateur en cas de changement du Status de la station.
- deux entiers représentant le nombre de retours et de départs 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.
- 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.

2.1.3 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 <Timestamp, ParkingSlot.Status>** 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.

2.2 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 avions 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 2.3.3.

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 object 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 recevera 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 visitors 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 différent 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**
- une somme d'argent sur le compte, de type **double**
- une limite de crédit autorisé, de type **double** fixée à un solde nul
- 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 Le réseau en lui-même Network

Le réseau, **Network**, représente l'ensemble du système contenant les stations, les emplacements de vélos, les vélos, les utilisateurs, etc.

Il a pour attributs :

- un nom de type **String**
- une liste de stations codée sous la forme d'une **ArrayList<Station>** : c'est l'ensemble des stations du réseau
- une liste d'utilisateurs codée sous la forme d'une **ArrayList<User>** : c'est l'ensemble des utilisateurs du réseau
- une liste de vélos codée sous la forme d'une **ArrayList<Bicycle>** : c'est l'ensemble des vélos du réseau.

2.3.3 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 *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 en parlera plus en détail en 3.4.

2.3.4 Le client MyVelib

Cette classe représente le client qui peut créer des **Networks**. Nous utilisons cette classe notamment pour essayer l'ensemble de nos fonctions et créer des scénarios d'usage.

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 avec ConcurrentSkipListMap

Le sujet introduit des problématiques liées à la gestion du temps. Il s'agit de garder une trace temporelle des diverses opérations effectuées dans le réseau, par exemple la location et le retours 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.

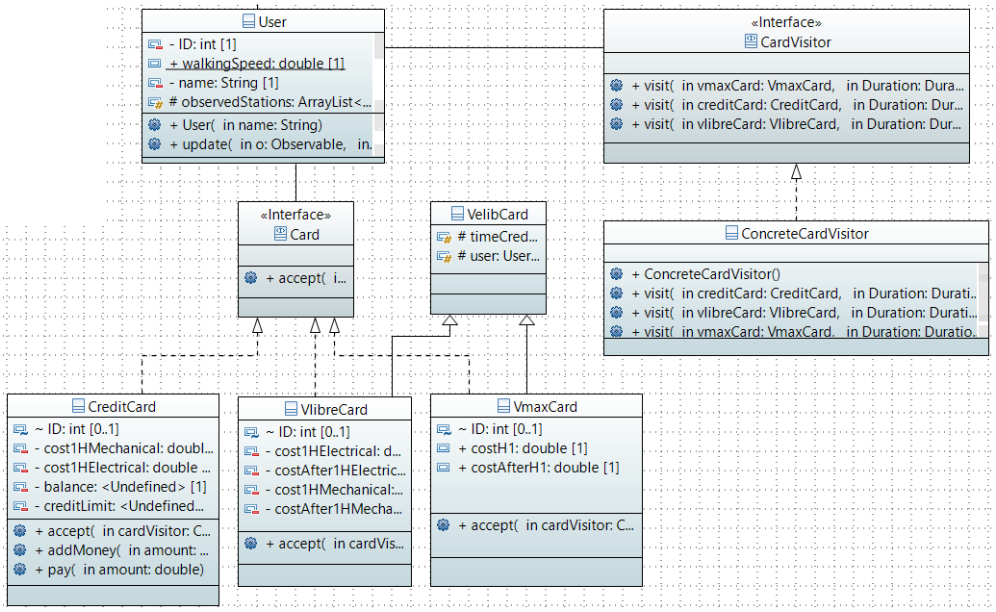

FIGURE 3.1 – Diagramme UML du **Visitor Pattern**

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.

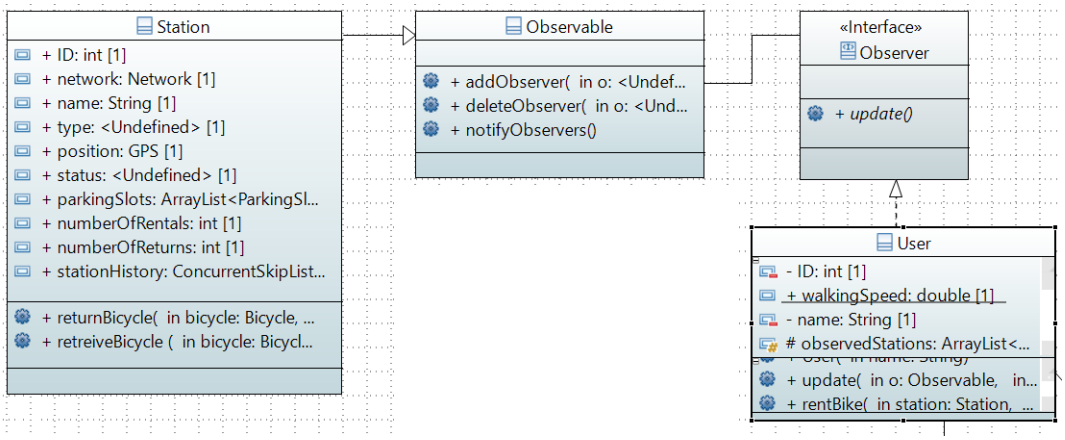

FIGURE 3.2 – Diagramme UML du **Observer Pattern**

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.

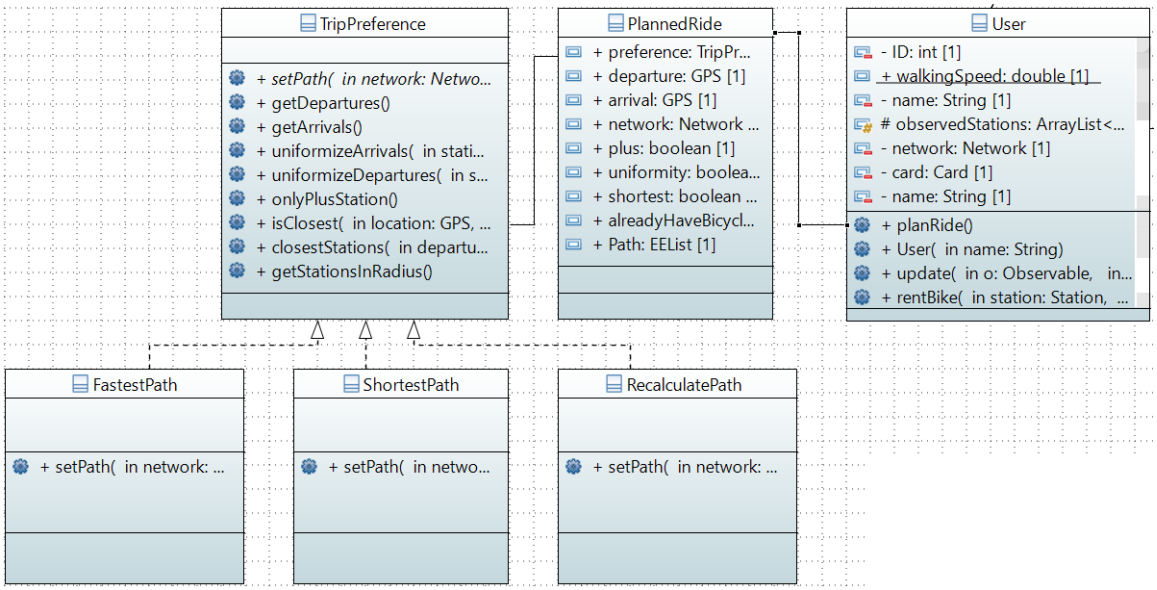

FIGURE 3.3 – Diagramme UML du **Strategy Pattern**

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 ses stations les plus proches.

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 et 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 2.3.3 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 2.3.3 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 2.3.3.

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. Nous devons prendre garde au fait de diminuer de la bonne quantité le temps de crédit pour que l'utilisateur ait le moins à payer possible, en utilisant le moins possible de son temps de crédit.

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 croissant 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(stationsToSor 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 Simulations et tests

5.1 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 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.

5.3 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.

5.4 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.

5.5 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 **NullPointException** 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.

6.1.2 Respect du principe *Open / Close*

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 avons codé les constantes en tant que static. 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.

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.

6.1.3 Un modèle pédagogique

Nous nous sommes également efforcés d'implémenter un maximum de *design patterns* afin de mieux les comprendre.

6.1.4 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.

6.1.5 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.

6.2 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 encore insuffisantes

Il nous reste encore du temps d'ici la date finale de remise des projets, mais nous aurions souhaité créer des simulations autonomes du système Velib et pour le moment, nous entrons les commandes toujours manuellement et nous concevons nos propres scénarios d'usages. Nous souhaitons donc utiliser des **thread** par exemple.

6.2.3 Un manque de toString()

En l'état actuel du code, nous avons écrit peu de *toString()*. C'est un des points que nous allons améliorer d'ici la fin du projet.

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 implémenté 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 globalement tout codé en collaborant étroitement mais :

- Anis s'est occupé seul de tout ce qui est planification des courses.
- Léonard s'est occupé seul des fonctions de dépôt et retour des vélos ainsi que de tout ce qui est gestion du temps et 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 avions 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 avions 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'avoué notre nouvelle addiction pour cette nouvelle drogue qu'est la POO.

Merci pour votre attention,

Anis et Léonard