CentraleSupélec

Rapport du projet MyVelib

Axel Chalayer | Elian Mangin

Génie Logiciel Orienté Objet

2023

Résumé

Ce document est un rapport d'un projet nommé MyVelib de l'électif Génie Logiciel Orienté Objet (2A CentraleSupélec). Il décrit le travail effectué par un binôme dans le cadre du projet, notamment la planification des tâches et la manière dont le travail a été organisé malgré les défis posés par la communication à distance. Le document décrit également l'architecture et le design de la solution, ainsi que la conception de l'interface utilisateur en ligne de commande. Des tests ont été effectués pour garantir la fiabilité de la solution, et les obstacles rencontrés ont été surmontés grâce à des stratégies spécifiques.

Table des matières

1 Introduction

Ce document met en lumière le travail accompli par notre binôme dans le cadre du projet MyVelib. En dépit des défis posés par la coordination à distance et la communication, nous avons réussi à travailler efficacement en tandem, à travers une répartition judicieuse des tâches respectant les talents et les disponibilités de chacun.

Après une analyse approfondie du sujet, nous avons établi un plan de travail, qui traçait notre trajectoire pour chaque étape du développement. Dans notre approche de conception pour MyVelib, nous avons respecté le principe OPEN-CLOSED, un concept qui préconise des modules ouverts à l'extension, tout en étant fermés à la modification. Nous avons également pris certaines libertés en relation avec le sujet, en intégrant ces modifications de manière harmonieuse dans notre solution.

Le projet a été mené en utilisant Java, en se conformant aux principes de la programmation orientée objet. Nous avons conçu une architecture modulaire, facilitant ainsi l'ajout ou la suppression de fonctionnalités. Nous avons aussi porté une attention particulière à l'interface utilisateur en ligne de commande (CLUI), résultant en une interface simplifiée et intuitive qui facilite l'interaction des utilisateurs avec le système.

Au fil de ce rapport, nous détaillerons l'organisation du travail, l'architecture et le design de notre solution, ainsi que la conception de l'interface utilisateur en ligne de commande (CLUI). Nous discuterons des tests que nous avons menés pour garantir la fiabilité de la solution, et nous partagerons les obstacles rencontrés, tout en expliquant les stratégies que nous avons adoptées pour les surmonter.

Nous sommes fiers de notre travail sur MyVelib et espérons que ce rapport permettra de comprendre notre solution et notre processus de développement.

2 Organisation du travail

Pour réaliser ce projet, il a fallu une certaine organisation. En effet, entre le fait que l'un des membres du groupe n'habite pas sur le campus, le fait que chacun ait des impératifs personnels extra-scolaires, et le travail demandé par les autres électifs - différents pour chacun des deux membres - il était compliqué de se retrouver pour travailler ensemble et nous avons dû énormément travailler en différé. Pour pallier ce problème il a fallu répartir les tâches dès le début du projet et essayer d'être efficace dans nos communications pour qu'il n'y ait pas de quiproquo et que le travail de l'un n'empiète pas sur les tâches de l'autre.

2.1 Calendrier du projet

— Semaine 1 (01/05 → 07/05) :

1. 1er appel visio pour réaliser ensemble un premier diagramme UML et partir sur les mêmes bases
2. Création du Git pour partager le code
3. Travail chacun de notre côté pour améliorer l'UML
4. Ecriture des classes de base du projet validées ensemble

— Semaine 2 :

1. 2e appel visio pour mettre en commun nos travaux sur le diagramme et valider le modèle
2. Creation d'un fichier LaTeX partagé sur Overleaf pour le rapport
3. Développement du code (core)

— Semaine 3 :

1. 3e appel pour discuter de l'avancée et répondre aux questions respectives. Redistribution des tâches
2. Développement des tests JUnits CORE.
3. Finalisation du code CORE.
4. Réflexion au modèle CLUI puis Développement
5. Implémentation du rapport

— Semaine 4 :

1. Correction des erreurs soulevées par les tests
2. Rédaction des tests scénario
3. Finalisation du code CLUI
4. Développement des tests JUnit CLUI
5. Ajustements, corrections de bugs, nettoyage du code, génération JavaDoc
6. Finalisation du rapport

2.2 Gestion du Git

Nous nous sommes mis d'accord pour réaliser des commits et de push très régulièrement lors du développement, à chaque plus ou moins grande modification qui n'entrainait pas d'erreur, afin d'éviter d'avoir deux versions différentes qui seraient difficiles a merge. Nous nous sommes également mis d'accord pour s'efforcer de mettre des titres clairs pour chacun des commits. L'idée était de pouvoir suivre mutuellement l'avancée de l'autre et de comprendre clairement les modifications apportées.

https://github.com/elianmangin/Group04_MyVelib_Mangin_Chalayer.git

2.3 Répartition des tâches

Mise au point du modèle (UML)	Commune
code CORE V1	Elian
code CORE V2	Axel
code CLUI	Axel
test JUint CORE	Elian
test JUnit CLUI	Axel
test Scenarios	Elian

On remarque qu'il y a eu deux versions du code CORE. La raison de cela sera détaillé plus loin mais pour résumer, nous avons mis un peu de temps à appréhender le sujet avant d'arriver à une version satisfaisante. Dans le souci de respecter un peu plus le principe OPEN-CLOSED et d'intégrer plus de pattern, il a fallu modifier le code et en faire une nouvelle version.

2.4 Bilan d'organisation

Nous avons essayé de faire notre maximum pour organiser notre manière de travailler, mais nous avons réalisé que c'était une tâche ardue. Ce projet nous a permis de prendre réellement conscience de l'importance de la modélisation de la solution avant de procéder au développement et de ne pas céder à la précipitation ainsi que l'importance de la communication au sein d'un groupe de développeur.

3 Architecture et design

3.1 Première version CORE

3.1.1 Analyse du sujet

Cette partie vise à synthétiser la description du système pour retenir les éléments à implémenter dans notre solution.

Pour commencer, nous avons analysé l'énoncé pour bien comprendre le système que nous devions conceptualiser. Il est très important de savoir où l'on va avant de coder. Dans cette optique, nous avons commencé par lister les fonctionnalités que doivent avoir le produit final.

3.1.2 Description simple des fonctionnalités

Un utilisateur (User) doit pouvoir louer et déposer des vélos (Bicycle) dans des stations (DockingStation) ou dans la rue. Le coût des courses dépend du temps de location, mais aussi d'un abonnement et autres éléments relatifs à une carte d'abonnement (Card). L'utilisateur doit pouvoir être en mesure de demander un itinéraire suivant un planning définis. Il existe deux types de vélos influant sur le coût d'une course. Il doit être possible d'avoir un compte rendu des statistiques des utilisateurs et des stations.

Ceci est un très bref résumé, mais cette analyse nous a permis d'analyser les principales classe qu'il nous faudrait coder :

— DockingStation
— DockingStationBalance
— ParkingSlot
— User
— UserBalance
— Card
— Bicycle

Mais identifier les acteurs du système n'est pas suffisant pour définir la manière dont ils interagissent entre eux.

3.1.3 Choix de design

Après avoir effectué cette première analyse, nous avons réfléchis plus en détail au fonctionnement du système. Cette partie va détailler nos premiers choix de conception.

Outre les classes décrites dans la partie analyse du sujet, il nous est vite apparu que d'autres classes était nécessaires.

Pour commencer, plusieurs éléments du système ont besoin d'être localisé. Il convient donc de créer une classe Coordinates que l'on dotera des fonctions distance() et equals().

Pour les emprunts et les retours de vélos, nous avons créé une classe Renter qui gère l'emprunt de vélo et notifie toutes les autres instances de classes qui sont influencés par l'emprunt du vélo. Cela nous a semblé plus simple qu'intégrer cette fonctionnalité à une classe déjà existante.

Un autre choix que nous avons fait est de créer une classe Ride pour sauvegarder les trajets faits par les utilisateurs. Nous avons intégré le calcul du cout du trajet à cette classe. Dès lors qu'un utilisateur emprunte ou rend un vélo, on sauvegarde toutes les informations dans Ride. Lorsque le vélo est rendu le cout du trajet est automatiquement calculé et facturé. De plus, le trajet est ajouté à la balance de l'utilisateur et à la liste des trajets que le système conserve.

Enfin, il faut bien que tous les éléments du système soient stockés quelque part. Pour cela, nous avons opté pour la création d'une classe MyVelib. Cette classe permettra aussi de gérer les différents tri (sort station) demandés dans le sujet.

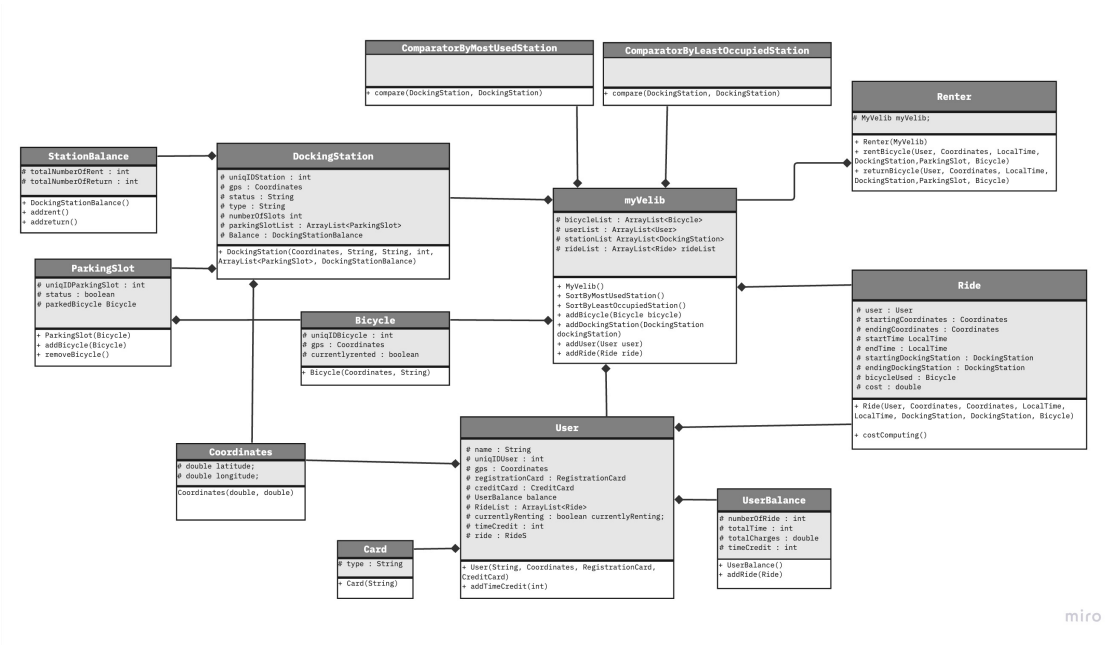

FIGURE 1 – Diagramme UML V1

Ce modèle était une bonne base, cependant lorsque nous avons commencé à coder, nous nous sommes vite rendus compte qu'il était non seulement incomplet, mais qu'il n'utilisait pas de pattern vu en cours et ne respectait pas le principe OPEN-CLOSED.

Voici pourquoi nous avons dû retravailler sur le diagramme UML et nous assurer de sa cohérence avant de se remettre au code.

3.2 Evolution de l'architecture CORE

Lors de la révision de notre modèle, nous avons pris le temps de relire le sujet, le cours et de se poser les bonnes questions. Nous avons essayé de nous rattacher au réel pour lever les zones d'ombres et au cours pour y trouver de l'inspiration dans les patterns étudiés pour établir un modèle optimal.

3.2.1 Libertés prises vis-à-vis du sujet

Dans cette sous-partie, nous allons énumérer les zone d'ombre du sujet et les choix que nous avons fait en conséquence.

Premièrement, le sujet mentionne qu'un utilisateur est détenteur d'une carte de crédit. Nous avons fait le choix de modéliser ceci en un simple attribut de User "creditBalance" de type double qui spécifie le montant d'argent détenu par un utilisateur, laissant la classe "Card" pour la carte d'abonnement qui, elle, est propre au système de velib.

Ensuite, on évoque un terminal pour les DockingStation. Ici, nous avons décidé de garder notre idée de Renter plutôt que de modéliser un terminal pour une station pour les mêmes raisons qu'évoquées en partie 3.1.3. On peut imaginer dans la vraie vie, un utilisateur serait muni d'un téléphone avec une application pour effectuer ses opérations de location et de retour. C'est ce rôle qu'occupe la classe Renter.

Dernière liberté concernant le CORE, pour le comparateur leastOccupied, nous avons décidé d'intégrer à notre class DockingStation un attribut nbSlotsOccupied qui permettra d'effectuer cette comparaison. Les tests que l'on va effectuer comporte un nombre limité d'opérations de location (environ une dizaine maximum pour un test), la formule proposée dans le cours nous a parue alors peu précise. Aussi, concernant le type des stations, nous avons décidé qu'il ne serait pas figé et évoluerait en fonction du nombre de vélos garé. Une station plus deviendra normale si elle est occupée par un nombre suffisant de vélos (nbSlotOccupied $>$ 0.4*nbSlots) et inversement si une station n'en a pas assez (nbSlotOccupied $<$ 0.4*nbSlots) de sorte à encourager les utilisateurs à entretenir une répartition uniforme des vélos sur les stations.

3.2.2 Pattern et respect du principe OPEN-CLOSED

Comme expliqué plus tôt, nous nous sommes efforcés en réalisant cette version de respecter le principe OPEN-CLOSED. Mais tout d'abord, nous nous sommes demandés quelles étaient les parties du CORE susceptibles de pouvoir être étendue un jour par les patrons d'un système velib comme celui-ci. Nous en avons identifié 3.

Cost Policy

En effet, les gérants du système de velib pourraient vouloir sortir de nouveaux abonnements vélib (autre que Vlibre et Vmax) dont les avantages et le calcul du coût seraient différents.

Pour répondre à cela, nous avons envisagé un mélange du Factory Pattern et du Strategy pattern. L'idée est que la méthode de calcul de coût soit enregistrée dans la carte de l'utilisateur. Lorsque la carte est créée (à la création de l'utilisateur), on enregistre le type sous forme de String et le constructeur de Card associe la politique de coût au type de la carte.

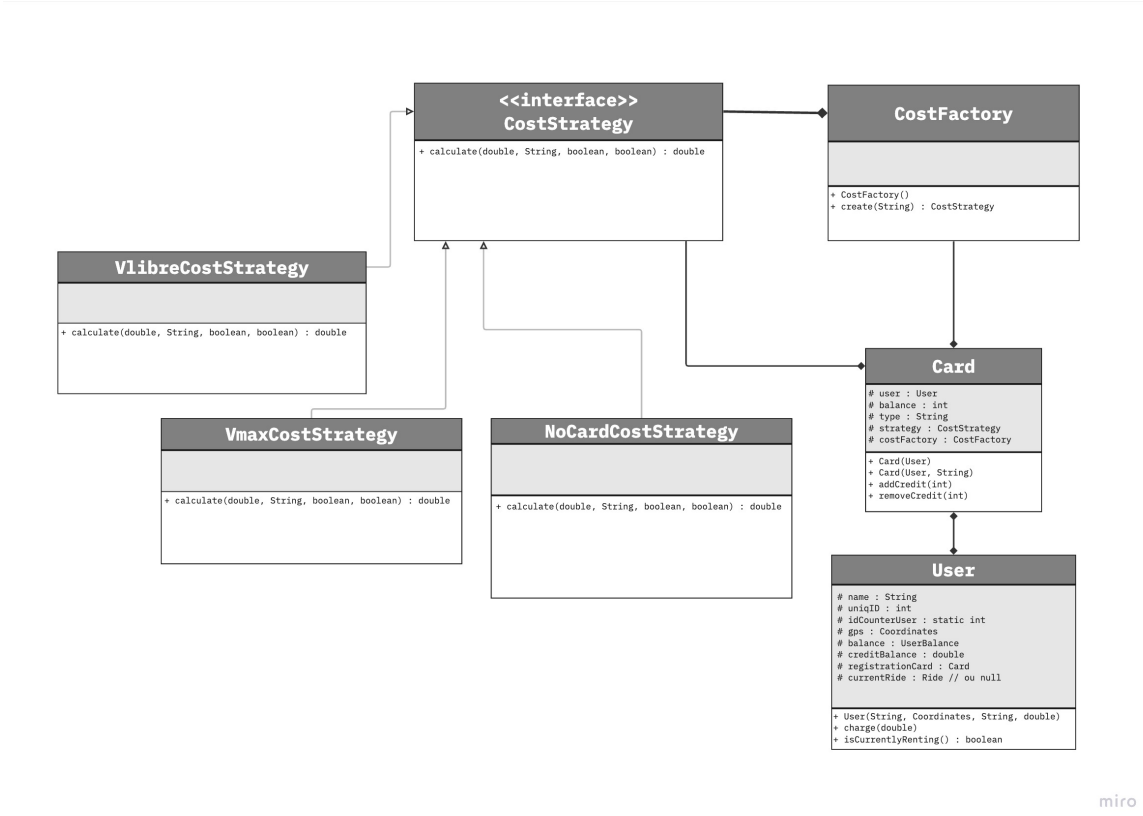

FIGURE 2 – UML CostFactory

Le constructeur de User créer une Card à partir du String qui représente le type de carte. Lui-même lance la CostFactory qui renvoie la CostPolicy correspondante et l'enregistre en attribut de la carte.

Avec ce pattern, on peut facilement étendre le modèle en ajoutant de nouveaux abonnements sans modifier le code.

Ride Planning

Ici, nous n'avons pas eu besoin de nous poser trop de question puisque la possibilité d'ajout de nouveaux RidePlanning est explicitement demandée dans le sujet. Nous allons donc détailler la solution retenue pour rendre cela possible. L'idée était de réaliser le même pattern que pour le CostPolicy, seulement pour le RidePlanning, nous avons une autre problématique. En effet, lorsqu'on lit les descriptions des planning facultatifs proposés par le sujet, on se rend compte que le départ et l'arrivée renvoyés ne sont pas toujours des stations, mais peuvent aussi être des coordonnées ou des vélos. La figure 3 montre le diagramme UML de la solution qui a été retenue.

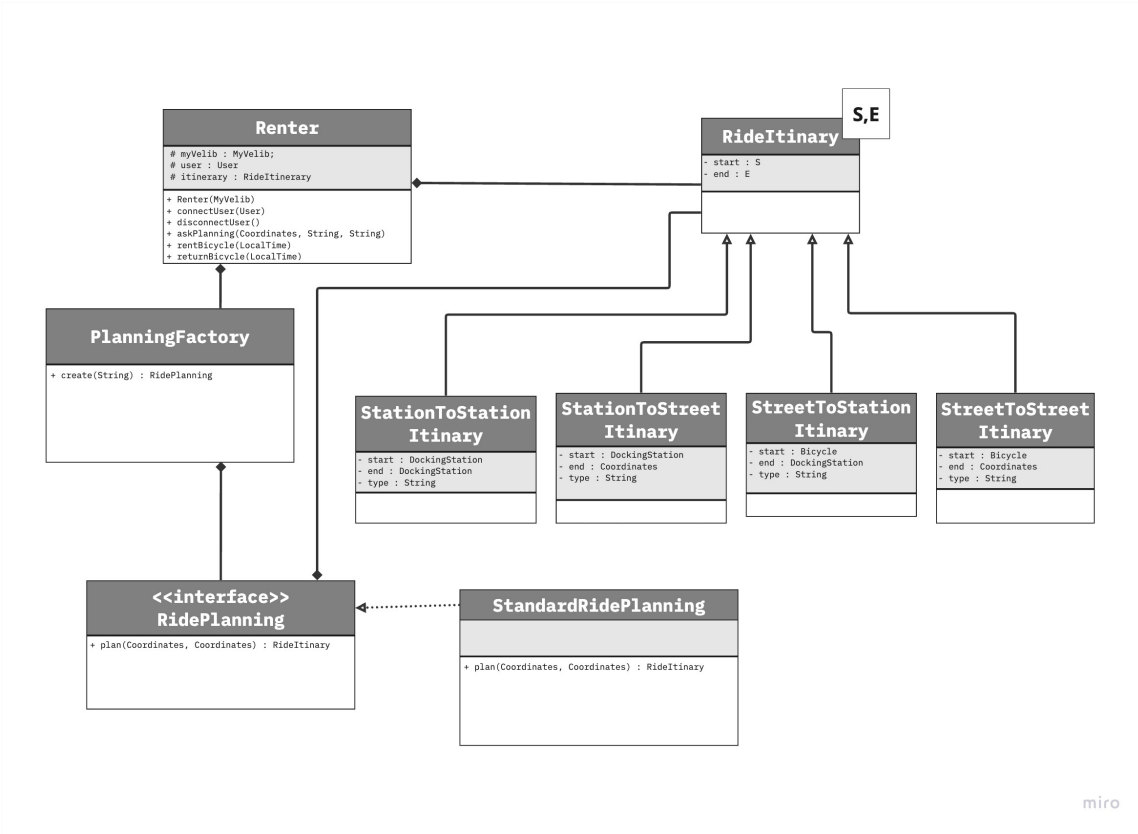

FIGURE 3 – UML PlanningFactory

L'interface RidePlanning dispose d'une méthode plan(Coordinates, Coordinates) qui renvoie un itinéraire pour se rendre d'un point A à un point B. Cependant, le départ et l'arrivée de l'itinéraire n'ont pas toujours le même type. On utilise donc une classe abstraite avec des types générique. Quatre classes viennent préciser ces types en fonction du type d'itinéraire. Par exemple, "StandardRidePlanning" renvoie un "StationToStationItinerary". Lorsque l'on utilise la fonction askPlanning de Renter après y avoir connecté un User, l'attribut Itinerary du renter prend le RideItinerary renvoyé. On peut ensuite utiliser la fonction rentBicycle() qui va tester le type de

start (if (start instanceof ...)) et exécuter les commandes adéquates (respectivement pour return). Cela nous fait une bonne transition pour parler du Renter.

Renter

Comme évoqué précédemment, le Renter a pour but d'effectuer les opérations de location et de retour. Nous utilisons ici le pattern Observeur Observable, avec en Observer le Renter, et en Observables : User, DockingStation et Bicycle.

En effet, lorsque l'on effectue une location avec rent(), il faut qu'un vélo soit récupéré dans une station et des changements de status vont opérer chez les observables. En voici quelques-uns :

— station.takeBicycle(bicycle) ;
— bicycle.setCurrentlyRentedBicycle(true) ;
— user.setCurrentRide(new Ride(...)) ;

Nous sommes conscients que le pattern n'est pas suivi à la lettre, mais il nous a inspiré dans la méthode "une classe déclenche les actions d'une autre".

La figure 4 montre le diagramme UML de la solution. Les flèches rouges représentent les attributs affectés par rent ou return.

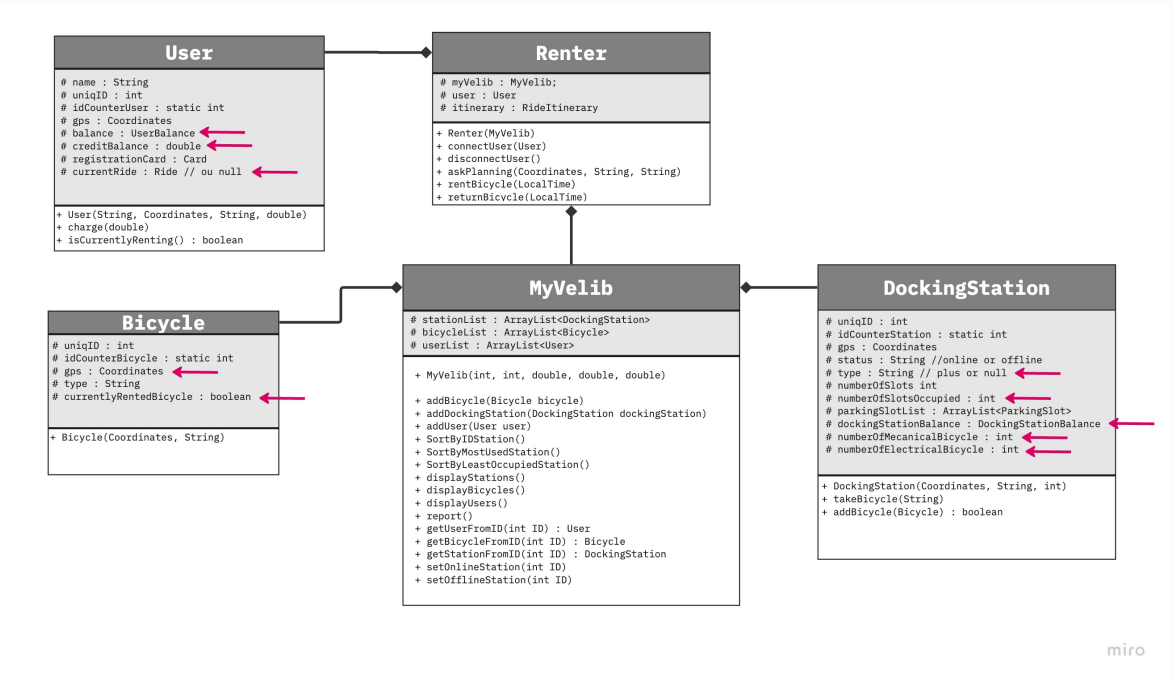

FIGURE 4 – UML PlanningFactory

Autres

Les autres éléments de ce modèle sont les comparateurs qui viennent implémenter l'interface Comparator<DockinStation> qui permettent le bon fonctionnement des différentes méthodes de tri de la classe MyVelib.

Le Diagramme de classe UML complet de la solution CORE est disponible à la racine du projet sous le nom de ClassUmlCore.jpg et en figure 5.

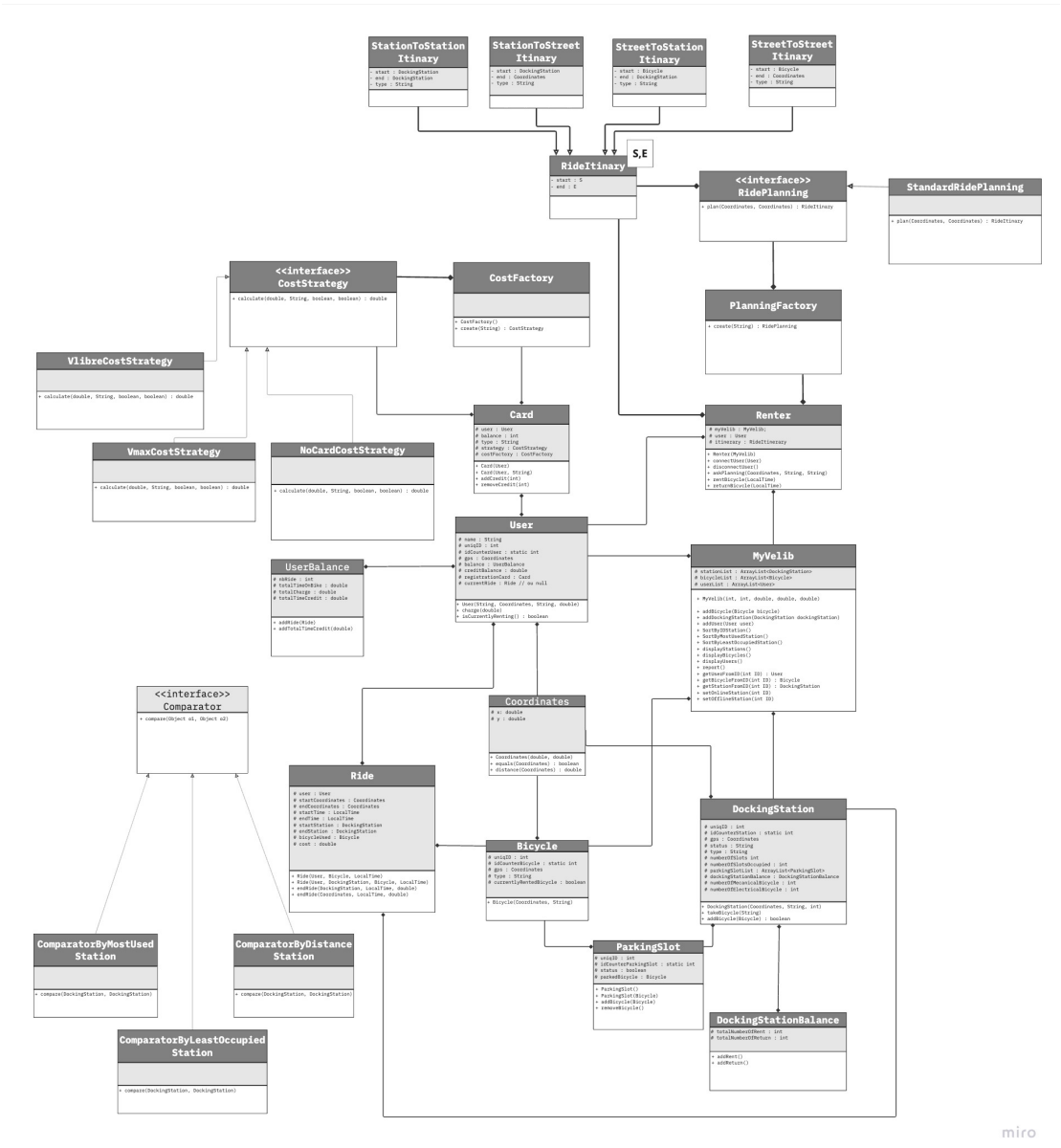

FIGURE 5 – UML PlanningFactory

3.3 Le CLUI

3.3.1 Design

Pour le CLUI le design a été bien plus rapide, de même que le développement puisqu'il reprend les méthodes du CORE. Nous avons besoins d'une classe avec un main qui initialise le système avec le fichier .ini et qui effectue une boucle avec un scanner pour récupérer les inputs de la console. Il nous faut aussi une classe qui sert à reconnaitre les commandes et à exécuter les méthodes correspondantes La figure 6 nous montre le diagramme de classe UML du CLUI.

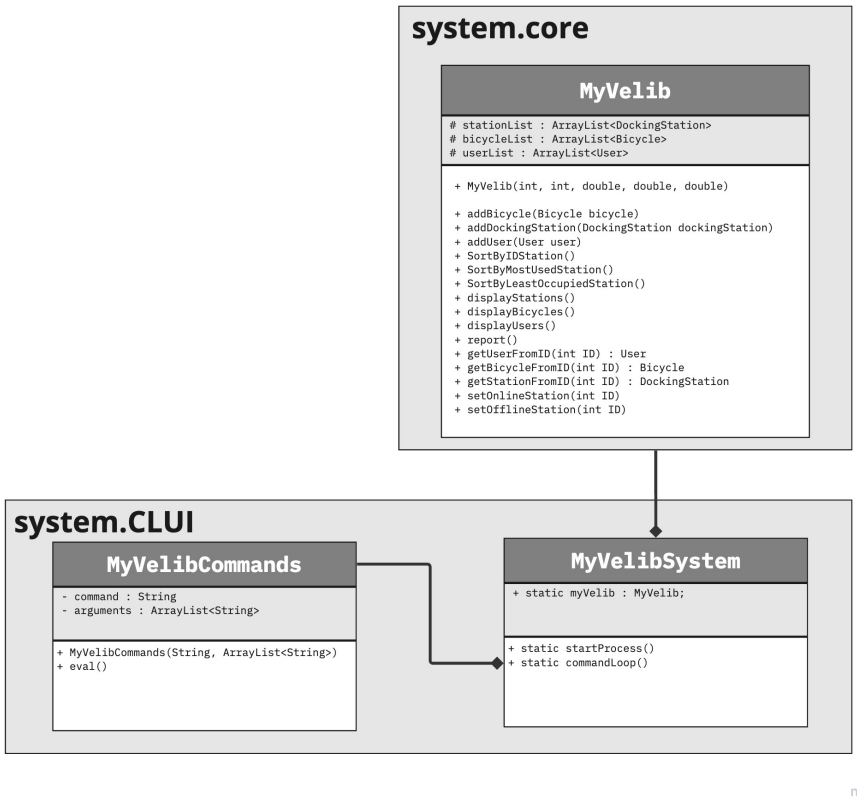

FIGURE 6 – UML PlanningFactory

La méthode commandLoop() lit les commandes du scanner et exécute des MyVelibCommands. Ces commandes font évoluer le système myVelib qui a été initialisé avec startProcess().

3.3.2 Les commandes disponibles

Nous avons développé les commandes obligatoires du sujet et ajouté quelques autres. Dans certains cas, nous avons pris des libertés vis-à-vis des arguments des

commandes de l'énoncé. Pour chaque commande, si c'est le cas, la raison sera expliquée.

help <>

Cette fonction s'utilise sans argument et permet d'afficher toutes les commandes dans la console et comment les utiliser.

runtest <testinput.txt>

Cette fonction exécute une à une toutes les commandes contenues dans le fichier txt.

setup <>

Créé un réseau myVelib avec 10 stations, 10 places par station dans un carré de 4km de côté dont les places sont occupées par des vélos à 75%.

setup <nstations> <nslots> <s> <nbikes>

Créer un réseau myVelib avec nstations stations, nslots slots par station, réparties dans un carré de s km de côté, avec nbikes vélos répartis uniformément sur les stations.

Nous avons pris la liberté d'enlever le <name> des arguments. Cela ne fait pas trop sens d'avoir plusieurs system de vélib pour une société dans une ville donc pas besoin de name. Et notre system est unique. Chaque commande setup réinitialise le system.

addUser <name> <cardType> <initBalance> <x> <y>
addUser <firstName> <lastName> <cardType> <initBalance> <x> <y>

Ajoute un utilisateur avec un nom donné, un type d'abonnement, un montant initial d'argent et une position.

Ici, nous avons ajouté des arguments qui nous semblent essentiels pour bien définir nos utilisateurs.

moveUserToBicycle <userID> <bicycleID>

Nous avons ajouté cette commande, car il semble contre-nature qu'un utilisateur puisse louer un vélo s'il n'est pas à côté. Il faut donc d'abord qu'il se rende à sa position.

moveUserToBicycle <userID> <stationID>

Même chose. Vaut aussi pour rendre un vélo.

moveUserToCoord <userID> <x> <y>

Même chose pour déposer un vélo dans la rue.

offline <stationID>

Désactive la station correspondant à l'ID.

online <stationID>

Active la station correspondant à l'ID.

askPlanning <userID> <xDestination> <yDestination> <wantedTypeOfBicycle> <planning>

Indique dans la console le départ et l'arrivé optimisé par le planning.

rentBike <userID> <stationID> <type>

Un utilisateur loue un vélo d'un type donné d'une station donnée.

rentBike <userID> <bicycleID>

Un utilisateur loue un vélo dans la rue.

returnBike <userID> <stationID> <duration>

Un utilisateur dépose un vélo à une station après une durée donnée.

returnBike <userID> <x> <y> <duration>

Un utilisateur dépose un vélo dans la rue après une durée donnée.

displayStation <stationID>

Affiche le toString d'une station donnée

displayUser <userID>

Affiche le toString d'un utilisateur donnée

sortStation <comparator>

Tri les station selon un critère donné ('ID' / 'leastOccupied' / 'mostUsed').

display <>

Affiche la liste des station activé, désactivé et des utilisateur. Pour chaque station, on peut voir la liste des ID des vélos qui l'occupe.

exit <>

Quitte le programme.

4 Tests

4.1 JUnits CORE

4.1.1 Classes testées

Nous avons testé chaque classe individuellement pour vérifier qu'elles avaient le comportement voulu. Les tests de plusieurs classes associées à une même fonction comme le calcul du cout ont étés testés dans un même test

Voici les tests pour les classes :
TestBicycle
TestCard
TestCoordinates
TestCost
TestDockingStation
TestRenter
TestRide
TestRidePlanning
TestUser
TestUserBalance

Ces tests ont permis de détecter de nombreux problèmes présents dans le code que nous avons corrigés. Les test on d'abord mis en évidence des bugs qui ont été corrigés par exemple des variables mal déclarés ou des erreurs dans les formules pour calculer le cout d'un voyage. Mais surtout les test nous ont fait remettre en question certains choix préalables. Par exemple en codant les test nous nous sommes renud compte que RideStart etait une classe inutile et que rajouter un constructeur avec moins d'argument à Ride était une solution plus simple.

4.1.2 Problème lié au compteurs pour ID statiques

Nous avons rencontré un problème lié au fait que les CounterUniqId sont des variables statiques. Pour la plupart des tests, on utilise un setUp() précédé d'un @BeforeEach, ce qui permet de créer tous les objets nécessaires avant de tester la fonctionnalité. Malheureusement, lorsqu'on fait cela, si l'on ajoute un utilisateur dans le setUp Avant chaque test, le CounterUniqId pour user va être incrémenté de 1. Donc l'utilisateur ajouté va voir son ID être incrémenté à chaque test, ce qui peut poser un problème. C'est un problème pour tous les éléments possédants un ID comme les utilisateurs, les stations et les vélos.
Pour pallier ce problème, nous avons décidé de toujours fixer les compteurs statiques à zéro à la création de l'objet MyVelib.

4.2 JUnits CLUI

Ces tests ont testé individuellement le bon fonctionnement de chaque commande du CLUI et ont révélé beaucoup d'erreurs. Que ce soit dans le code du CLUI mais aussi dans le code du CORE. Dans l'ensemble, ce n'était pas que le code était mal fait, mais plutôt qu'il n'était pas complet. Les problèmes étant que des situations qui ne sont pas censées se produire était possible. La majorité des problèmes a pu être réglé en jetant des exceptions. Je profite d'ailleurs de cette section pour parler de la classe GeneralException.

Cette classe a été développée dans le package CORE. Son but est de nous permettre de différencier lors de nos tests les exceptions renvoyées volontairement (pour bloquer un mauvais usage du système) des erreurs de code.

Nous allons énumérer des problèmes remontés par ces tests

4.2.1 Location depuis une station offline

Un test a montré qu'il était possible de louer un vélo depuis une station offline, ce qui ne devrait pas être possible. Nous avons donc fait en sorte que dans cette situation, une exception soit lancée avec un message précisant que cette action n'est pas possible

4.2.2 Arret du CLUI si arguments du mauvait type

Lors du MyVelibCommands.eval(), on commence par convertir les Strings dans les bons types. Par exemple, s'il y a un ID en argument, on le convertit en entier. Un test a donc montré un problème lorsque à l'emplacement d'un argument censé être un double ou un int, on écrit des lettres. Cela stoppait l'exécution du programme. Pour régler ce problème, lors du cast des types, on ajoute un try / catch qui récupère une Exception, et on lance une GeneralException à la place avec le message d'erreur correspondant.

4.2.3 Comarator inversé

Un test a montré que la fonction sortStation avec l'argument leastOccupied triait en fait la station de la plus occupée à la moins occupée (inverse de ce qu'elle est supposée faire). Ici, nous avons simplement corrigé l'erreur dans la fonction compare du comparateur correspondant.

4.2.4 Commande inconnue a la fin des test scenario

Lorsque l'on utilisait la fonction runtest, a la fin de l'exécution, il arrivait d'avoir des commandes inconnues. Cela venait du fait qu'il y avait des lignes vide à la fin du test qui était lues comme des commandes. Ce problème a été réglé avec l'aide

de M. Lapitre (notre professeur de TP) avec la méthode trim() pour supprimer les espaces.

4.2.5 "parkingSlot.parkedBicycle" is null

Lors de l'exécution de rentBike, il pouvait arriver que l'erreur "Cannot invoke "myVelibProject.Bicycle.getType()" because "parkingSlot.parkedBicycle" is null" survienne. La raison était que la methode takeBicycle de DockingStation vérifiait sur le type de vélo correspondait à celui demandé pour chaque place de parking avant de vérifier si un vélo était bien garé dessus. Nous avons réglé le problème avec l'ajout de "if (parkingSlot.status)".

4.3 Tests Scénarios

Ensuite, nous avons implémenté des tests d'utilisation réelle. Cela permet de vérifier que les classes interagissent bien entre elles et permettent au système global de fonctionner. Pour cela, nous avons simplement implémenté les cas d'utilisations suggérés par le sujet en ajoutant un scénario supplémentaire relatif au changement de type de station dont nous avons parlé en section 3.2.1. Ce ne sont pas des tests à proprement parler puisqu'on ne vérifie rien. Ce sont juste des scénarios que l'on peut lancer, le succès du test reste subjectif. Les scénarios ci-dessous sont pour nous réussit, nous vous invitons à les essayer.

Voici les tests d'utilisation réelle :

4.3.1 testScenarioRentalOfABike

> Initialisation du système avec setup
> Affichage de la station 1
> Ajout de l'utilisateur John Snow détenteur d'une carte Vmax
> Tentative de location de John Snow d'un vélo mécanique à la station 1 (Erreur car pas au mêmes coordonées)
> Déplacement de John snow jusqu'à la station 1
> Tentative de location de John Snow d'un vélo mécanique à la station 1 (réussie Velo 1)
> Déplacement de John Snow aux coordonnées (2.34 2.34)
> Dépot du vélo 1 dans la rue aux coordonnées (2.34 2.34) après 90 minutes
> Affichage de John Snow
> Ajout de l'utilisateur John Travolta détenteur d'une carte Vlibre
> Déplacement de John Travolta aux coordonées (2.34 2.34)
> Location du vélo 1 aux coordonnées (2.34 2.34) (réussi)
> Déplacement de John Travolta à la station 2

> Dépot du vélo 1 à la station 2
> Affichage de John Travolta

4.3.2 testScenarioRidePlanning

> Initialisation du système avec setup
> Ajout de l'utilisateur John Snow détenteur d'une carte Vmax
> John Snow demande un itinéraire pour se rendre aux coordonnée (1.537677 3.445788) avec un velo mecanique en suivant le planning standard

4.3.3 testScenarioSetUp

> Initialisation du systeme avec setup 10 15 105 10
> Ajout de John Snow Vmax 100€ (2.456784 6.344574)
> Ajout de John Carpenter Vlibre 100€ (5.124323 9.123561)
> Ajout de John Travolta null 100€ (5.124323 9.123561)
> Ajout de John Lennon none 100€ (8.137458 1.624970)

4.3.4 testScenarioStatitics

> Initialisation du systeme avec setup
> Ajout de John Snow Vmax 100€ (2.456784 6.344574)
> Il loue un vélo mécanique de la station 1 à la station 2
> Il loue un vélo electrique de la station 2 à la station 3
> Il loue un vélo electrique de la station 3 à la station 2
> Affichage de JohnSnow et des stations 1, 2 et 3
> Tri des stations par la plus utilisée puis affichage du rapport systeme
> Tri des stations par la moins occupée puis affichage du rapport systeme

4.3.5 testScenarioVisualisation

> Initialisation du systeme avec setup
> Ajout de John Snow Vmax 100€ (2.456784 6.344574)
> Affichage de John Snow
> Affichage de la station 1
> Affichage du rapport du systeme

4.3.6 testScenarioSwitchStationType

> Affichage de la station 9 (type=null)
> Affichage de la station 1 (type = plus)
> L'utilisateur 1 va chercher 2 fois de suite un vélo à la station 9 pour le ramener à la station 1

> Affichage de la station 9 (elle n'a plus assez de vélo donc type = plus)
> Affichage de la station 1 (elle a suffisament de vélo donc type = null)

Lancement des Scénario

Pour faciliter le lancement des scénarios et pour faire gagner du temps au correcteur, nous avons créé un Test JUnit qui lance tous les scénarios un par un. Comme c'est un test JUnit il est possible de lancer tous les scénarios à la suite ou alors individuellement (1 scénario = 1 test). Lancer system.CLUI.RunTestScenario.java.

5 Conclusion

Pour conclure, ce projet a été une opportunité d'appliquer concrètement les notions abordées durant nos cours et travaux pratiques. Ainsi, nous avons élaboré un système de location de vélos comportant plusieurs fonctionnalités, notamment la planification d'itinéraires et le classement des stations selon leur taux d'occupation.

Par ailleurs, nous avons créé une interface utilisateur en ligne de commande (CLUI) pour faciliter l'interaction avec le système. Cela nous a permis d'appréhender comment les futurs utilisateurs pourraient interagir avec notre système et de créer une interface plus intuitive.

Nous regrettons cependant de ne pas avoir eu le temps d'intégrer davantage de fonctionnalités optionnelles. Même si notre système est opérationnel et répond aux critères fixés, nous estimons qu'il y avait encore de nombreuses pistes à explorer pour améliorer l'expérience utilisateur.

Enfin, nous avons mis en œuvre des tests approfondis, comprenant des tests unitaires et des tests de scénarios. Cette démarche nous a permis d'identifier et de corriger divers problèmes pour assurer le bon fonctionnement du système dans différentes situations.

Au final, malgré quelques regrets, nous avons le sentiment d'avoir donné le meilleur de nous-mêmes dans ce projet. Nous espérons que ce rapport donnera un aperçu fidèle de nos efforts et du travail accompli.