Génie logiciel orienté objet

(Cours de P. Ballarini et A. Lapitre)

Rapport de projet

MyVelib

Groupe P (Quentin Delacroix & Pierre Ollivier)

Juin 2022

Table des matières

Introduction

Le but de ce projet est d'implémenter un système de partage de vélos à l'échelle d'une ville, tel que Vélib' Métropole (anciennement Vélib') à Paris et en Île-de-France. Un tel système de partage de vélos est doté d'un certain nombre de stations (1438 pour Vélib' Métropole au 10 mai 2022) et de vélos. Les vélos se répartissent en deux catégories : les vélos électriques et les vélos mécaniques. La plupart des utilisateurs possèdent une carte, qui leur permet de louer des vélos, mais il est aussi possible de faire sans. Dans le cas de MyVelib, on considère trois possibilités : soit un client ne possède aucune carte, soit il possède une carte de type Vlibre, soit il possède une carte de type Vmax. Le coût du trajet sera amené à changer en fonction du type de la carte éventuellement possédée par l'utilisateur.

Figure : Logo de Vélib' Métropole

Dans une grande ville dotée d'un système de vélos partagés, il y a souvent des stations plus utilisées que d'autres, ou encore des trajets davantage faits dans un sens que dans l'autre. Prenons l'exemple d'une rue en forte pente : les usagers pourront prendre un vélo pour descendre la rue en allant vite et en éprouvant des sensations agréables sans trop se fatiguer, mais privilégieront un autre moyen de transport pour remonter à cause de la pente. Dans ce cas, la station située en haut de la rue verra son nombre de vélos progressivement diminuer (les vélos seront empruntés mais jamais rendus), à l'inverse la station située en bas de la rue verra son nombre de vélos augmenter (beaucoup de vélos y seront restitués mais peu y seront employés). Les opérateurs tels que Smovengo (qui gère le réseau Vélib' Métropole) devront alors eux-mêmes faire remonter les vélos jusqu'à la station située en haut de la côte, par exemple en affrétant des camions. Cela induit un coût pour l'entreprise. L'opérateur a ainsi intérêt à favoriser les utilisateurs qui vont prendre un vélo dans une station en ayant beaucoup (ce qui libère une place) et/ou rendre un vélo dans une station en ayant peu. C'est pourquoi, dans MyVelib, on considère deux types de stations : certaines stations, notées « plus », octroient un bonus quand on y dépose un vélo. De même, on pourra chercher à faire faire aux utilisateurs un petit détour pour qu'ils restituent leur vélo dans une station moins remplie.

Dans la vraie vie, les stations subissent des dégradations et ne sont pas forcément opérationnelles. Dans MyVelib, ce comportement existe : les stations peuvent être en ligne ou hors ligne, et il est impossible d'emprunter ou de rendre un vélo dans une station hors ligne. En revanche, on aurait pu imaginer un même système pour les vélos, mais cela n'était pas proposé dans le sujet et nous ne l'avons pas implémenté.

1. Notre projet

a. Caractéristiques principales et « design decisions »

1. Cœur du projet (Core)

Dans cette sous-partie, nous allons décrire chacune des classes implémentées dans le core du système, ainsi que leurs principales caractéristiques. Afin de rendre la vision globale plus claire, nous partirons des classes les plus profondes dans le diagramme et remonterons jusqu'à atteindre MyVelib.java, la

classe principale du projet, qui implémente la plupart des fonctionnalités phares du système et repose sur toutes les autres classes pour ce faire.

La grande majorité des classes qui sont ici présentées possèdent un identifiant (int), généré de manière unique à la création en incrémentant une variable statique *uniqueId.* Les classes pertinentes (User, Station, Location, Rent et MyVelib) *override* tous la méthode Objet *toString()*, afin d'afficher des statistiques et états utiles au cours d'un scénario.

De très nombreuses méthodes renvoient des exceptions, accompagnées d'un message (par exemple : « Unknown bicycle type »). Ces exceptions remontent jusqu'aux méthodes principales de MyVelib, qui les attrapent.

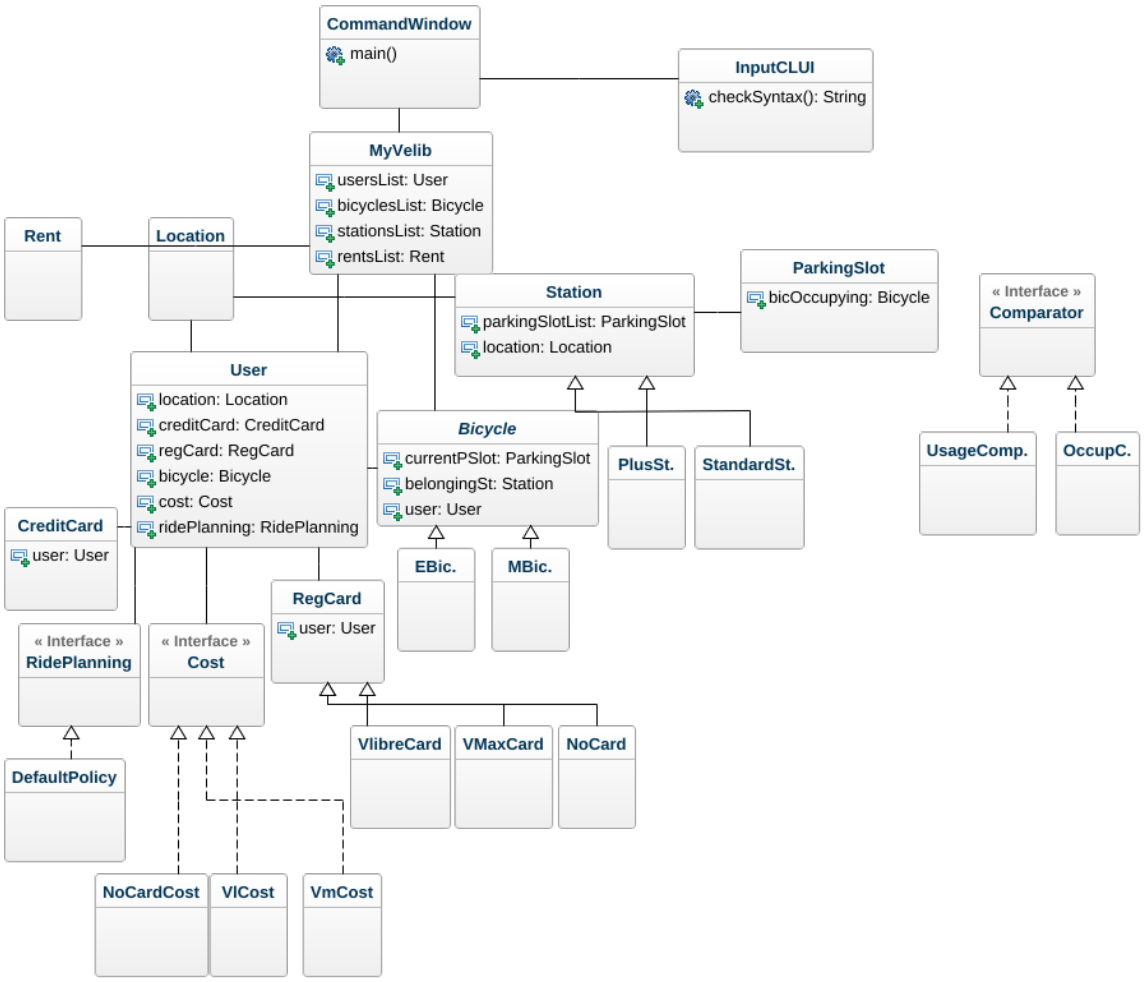

Figure 1 : Diagramme de classes simplifié de notre projet. Toutes les variables en …List désignent des listes et sont donc du type ArrayList<type>.

a. Registration cards

RegistrationCard est une classe abstraite qui permet d'implémenter le système de cartes d'abonnement au sein du système. Trois cartes implémentent cette classe : VlibreCard, VmaxCard et NoCard. Chaque utilisateur possède l'une de ces cartes, et chaque carte possède (a priori) un utilisateur : cette implémentation peut ainsi permettre à un utilisateur de changer de carte (il pourrait en prendre une autre, ou prêter la sienne, par exemple). Même lorsqu'un utilisateur n'a pas de carte, son attribut registrationCard est de classe « NoCard » par défaut.

Chacune de ces cartes correspondent au type de carte qu'un utilisateur peut posséder. Leur principal intérêt est d'enregistrer le time balance de l'utilisateur qui la possède. La différence entre les trois cartes est le type, qui est un attribut String permettant d'identifier le type de la carte.

NoCard est un peu spécial : il s'agit simplement d'une « fausse » carte donnée aux utilisateurs n'en possédant pas, et qui est présent dans le code afin d'éviter des exceptions nullPointer dans les méthodes de coûts.

b. *Cost*

Cost est une interface, implémentée selon un design pattern « Strategy ». C'est cette interface qui permet au système de différencier le coût d'une location selon le type de carte que porte l'utilisateur lorsqu'il rend le vélo.

Cost possède une unique méthode *cost()*, qui est donc implémentée par 3 « concrete strategies » : NoCardCost, VlibreCost et VmaxCost. Cette méthode, qui prend en argument le type de vélo, la date de location et de retour, ainsi que la carte d'abonnement (pour pouvoir utiliser/mettre à jour le *time balance*), calcule puis renvoie le coût d'une location, selon la stratégie explicitée dans le sujet du projet. Cette fonction s'occupe également de mettre à jour le time balance enregistré sur la carte d'abonnement, le montant total qu'a payé l'utilisateur ainsi que le temps total de location de ce dernier. Par ailleurs, cette méthode vérifie bien que la date de retour n'est pas antérieure à celle de location. Elle renvoie également une exception lorsque le type de vélo n'est pas reconnu.

c. *Observer pattern*

Avant de poursuivre sur les classes principales du programme, il est nécessaire d'aborder le design pattern que nous avons décidé d'utiliser afin de les implémenter. En effet, notre code implémente un Observer pattern : notre projet possède donc deux interfaces Observer et Observable, qui implémentent respectivement les méthodes *update()*, puis *registerObserver()* (et *removeObserver()*).

Les utilisateurs ainsi que les stations sont les *concrete observers*, alors que les vélos sont les *concrete observables*. La raison pour laquelle nous avons décidé d'implémenter tout cela de cette manière est que nous pensions pouvoir utiliser la méthode *update()* afin de transmettre des informations sur l'état du vélo à la station où il est garé ; ou à l'utilisateur qui l'utilise. En pratique, dans l'état actuel du code, cette méthode n'est pas utilisée : il était plus pratique de procéder autrement.

d. *User*

User.java permet de définir et gérer chaque utilisateur du système, ainsi qu'enregistrer ses statistiques (montant total prélevé, temps total de location, etc.). En particulier, un utilisateur possède une carte d'abonnement, mais aussi une carte de crédit et une position.

Les cartes de crédit sont gérées par la classe CreditCard. Chaque carte de crédit a un possesseur (utilisateur), et son seul but en pratique est d'enregistrer son solde ; qui peut d'ailleurs être négatif. Nous avons choisi d'implémenter le solde d'un utilisateur de cette manière afin qu'il puisse changer de carte (par exemple, un utilisateur peut posséder plusieurs cartes de crédit, et choisir avec laquelle payer).

La position d'un utilisateur (comme celle d'un vélo et d'une station, en fait) est enregistrée dans un attribut Location. Location est une classe qui possède deux attributs longitude et latitude, et

implémente la méthode *distanceTo(Location l)*, qui calcule et renvoie la distance entre cette instance de Location et la position entrée en paramètre. Au départ, chaque utilisateur/station avait simplement deux attributs longitude et latitude, mais lorsque nous avons implémenté le ride planning et que nous avions besoin de calculer des distances entre positions, il s'est avéré plus pratique de créer la classe Location.

User possède de nombreux constructeurs. En effet, un utilisateur peut ou peut ne pas posséder une carte d'abonnement ; sa carte de crédit peut avoir un solde de départ non nul ou l'inverse ; sa position peut être précisée ou bien commencer à (0,0) par défaut ; etc.

User possède aussi des attributs Cost et RidePlanning, qui sont implémentés selon le design pattern « Strategy ». Un utilisateur peut donc suivre différentes politiques de coût (selon la carte qu'il porte) et de planning.

Enfin, un utilisateur, en tant qu'observer, peut observer un vélo, qui lui est « attaché » le temps d'une location. Utilisateur possède également deux attributs *rentTime* et *rentStation*, valant null par défaut, et qui permettent d'enregistrer le temps et la station de départ lors de la location d'un vélo.

e. Stations & parking slots

Comme on s'y attendait, la classe Station est l'une des plus denses du core. Il s'agit d'une classe abstraite, étendue par les sous-classes StandardStation et PlusStation, dont la seule différence est un attribut String « type », afin de les différencier.

Un objet Station possède toujours une ArrayList qui enregistre les emplacements de parking, définis par la classe ParkingSlot. Un ParkingSlot est un objet simple qui possède un attribut de classe Bicycle afin d'identifier l'éventuel vélo qui y est garé (*null* si aucun vélo n'est garé). Le statut d'un parking slot est décrit par un attribut String *status*, pouvant prendre trois valeurs : « free », « occupied » et « OoS » (Out of Service). Enfin, ParkingSlot implémente deux méthodes empty() et fill(Bicycle), afin de respectivement supprimer le vélo de cet emplacement et rajouter un vélo à cet emplacement.

Station enregistre des statistiques à propos de son utilisation : *nbRents*, *nbReturns*. Une station peut être « online » ou « offline », selon un attribut booléen *online*. Enfin, une station enregistre les vélos qui s'y trouvent par une *ArrayList<Bicycle>* et un entier *bicycleCount*.

Station implémente quelques méthodes utilisées pour vérifier l'état de ses parking slots. Ainsi, *countParkingSlotsOccupied(type)* et *countParkingSlotsFree()* renvoient respectivement le nombre d'emplacements pris (pour un type de vélo donné) et le nombre d'emplacements libres. *findParkingSlotOccupied(type)* et *findParkingSlotFree()* renvoient respectivement le premier parking slot occupé (pour un type de vélo donné) et le premier parking slot libre de la liste. Toutes ces méthodes sont importantes afin de pouvoir louer et rendre un vélo à une Station ; ou, au contraire, renvoyer une exception lorsque la station n'est pas disponible.

Les exceptions renvoyées dans ces méthodes s'attachent à préciser la nature de l'erreur : « No bicycle found », « No bicycle found of this type », « No free parking slot », etc.

f. Ride planning

Ride planning est une classe abstraite, qui sert de base à un design pattern « Strategy ». À l'origine, il s'agissait d'une interface ; mais nous nous sommes rendu compte que la classe avait besoin d'attributs pour fonctionner comme nous le souhaitions, alors nous l'avons changée en classe abstraite.

RidePlanning est donc une classe qui est étendue par les différentes politiques de planning. Dans l'état actuel du projet, il n'y en a qu'une : DefaultPolicy, qui est la politique par défaut. RidePlanning possède 4 attributs : d'abord *startLocation* et *endLocation* qui sont les positions de départ et la destination souhaitée par l'utilisateur, et dont les valeurs sont attribuées par le constructeur de RidePlanning. Les deux autres attributs, *startStation* et *endStation*, sont, comme leur nom l'indique, les stations de départ et d'arrivée que la méthode de planning a identifiées comme étant les meilleures, quand à la position de départ et la destination. Ces deux attributs sont initialisés comme *null,* et sont mis à jour par la méthode principale de RidePlanning, sobrement nommée *ridePlanning().*

Cette méthode prend en arguments une ArrayList<Station> et un String *type*, pour le type de vélo souhaité. Implémentée dans DefaultPolicy, la méthode cherche simplement, l'une après l'autre, la station qui minimise la distance à l'emplacement de départ, puis pour la destination. Pour calculer les distances et les comparer, il fait appel à la méthode *distanceTo()* implémentée dans Location. Bien que simple, l'algorithme n'est pas stupide et peut renvoyer 4 exceptions différentes :

- Aucune station de départ disponible avec le type de vélo demandé

- Aucune station de départ plus proche que de simplement marcher à la destination

- Aucune station d'arrivée disponible avec le type de vélo demandé

- Aucune station d'arrivée plus proche que la station de départ (c'est-à-dire que la meilleure station d'arrivée est aussi la station de départ). Dans ce cas, il vaut mieux marcher.

Enfin, RidePlanning *override* également *toString()*, afin d'afficher les stations trouvées et leur position. C'est très utile pour vérifier si l'algorithme trouve bel et bien le meilleur chemin.

g. Bicycle

Bicycle, avec MyVelib, est la deuxième classe du core la plus dense. En fait, en suivant l'Observer pattern et désignant Bicycle comme l'observable (et qui implémente d'ailleurs cette interface), nous avons choisi d'y implémenter toutes les méthodes permettant à un vélo de se garer, de quitter une station, d'être emprunté par un utilisateur, et d'être restitué par cet utilisateur.

Bicycle est une classe abstraite, implémentée par les deux types de vélo : ElectricBicycle et MechanicBicycle. La seule différence entre ceux-ci est l'attribut String *type* qui permet d'identifier leur type. Bicycle enregistre également la station, le parking slot et/ou l'utilisateur auquel il est actuellement attaché.

Bicycle, en tant qu'observable concret, *override* les méthodes d'Observable :

- *registerObserver(Station st)* est une méthode qui en cache une autre : *addToStation(Station st)*, qui permet, comme son nom l'indique, d'ajouter le vélo à la station en argument (ou renvoie une exception si c'est impossible). Pour cela, la méthode vérifie d'abord que la station ait un parking slot disponible en appelant st.hasParkingSlotAvailable(). Si aucune exception n'est renvoyée, la méthode appelle st.findParkingSlotFree(), et renvoie le premier parking slot disponible, avant de mettre à jour à la fois les attributs de Bicycle et st pertinents.

- *registerObserver(User user)* appelle directement *addUser(User user)*, qui « attache » un utilisateur à ce vélo. Cette méthode est donc appelée lorsqu'un utilisateur réserve un vélo. Pour cela, la méthode vérifie d'abord que l'utilisateur n'a pas déjà loué un vélo, et que ce vélo n'est pas déjà loué par un

autre utilisateur. Ensuite, les attributs pertinents de ce vélo et de user sont mis à jour pour prendre en compte leur lien.

- *removeObserver(Station st)* appelle directement *leaveStation()*. Cette méthode « détache » le vélo de la station et du parking slot dans lesquels il se trouve, en vérifiant préalablement que le vélo est bel et bien dans une station. Cette fonction est apelée lorsque le vélo est emprunté par un utilisateur.

- *removeUser(User user)* appelle directement *leaveUser()*. Cette méthode « détache » le vélo de son utilisateur, en vérifiant préalablement que le vélo est bel est bien emprunté. Elle est appelée lorsque le vélo est retourné à une station.

Enfin, les méthodes *park(ParkingSlot ps)* et *unpark()* sont des méthodes intermédiaires appelées pour attacher ou détacher un vélo d'un ParkingSlot en particulier.

h. Rent

Cette classe sert à enregistrer les données d'une réservation, c'est-à-dire : l'utilisateur, la date d'emprunt, la date de retour, la station d'emprunt, la station de retour, le vélo utilisé et le montant payé. Elle inclut également une méthode *toString()* pour afficher ces données.

i. MyVelib

Et enfin, la classe MyVelib. C'est la classe principale du core : la plupart des méthodes qu'un scénario appelle afin d'effectuer des actions sur le système sont définies ici. Bien sûr, toutes ces méthodes reposent sur les classes et méthodes déjà abordées précédemment.

MyVelib enregistre dans des ArrayList toutes les stations, utilisateurs, vélos, cartes d'abonnement, cartes de crédit et l'historique des réservations. Elle garde également en mémoire le nombre de stations de chaque type, le nombre total de parking slots, et le nombre de vélos de chaque type lors de la création d'un système.

Le constructeur de MyVelib est de loin la méthode la plus longue du core. Cette méthode prend en argument le nom du système, le nombre de stations souhaité et le nombre total de parking slots. Le système est ainsi créé de cette manière :

- 30% des stations sont de type plus, les autres standards

- 70% des parking slots sont occupés par des vélos

- 30% des vélos sont électriques, les autres mécaniques.

La raison pour laquelle ce constructeur est si complexe est qu'il construit toujours un système en équilibrant la répartition de ses différents éléments ; c'est-à-dire que, par exemple, les 30% de vélos électriques ne se trouvent pas sur une seule station, mais sont répartis équitablement (dans la mesure du possible) sur toutes les stations. Même chose pour les parking slots, pour les emplacements occupés, etc.

La première méthode importante de cette classe est *rent(user, station, type, time)*. C'est la méthode qui permet à un utilisateur d'emprunter un vélo du type choisi, à la station entrée en argument et à la Date time (en effet, la plupart des classes du core importent la bibliothèque java.util.Date afin de représenter le temps). Cette méthode vérifie que la station est bien en ligne, puis lance diverses méthodes déjà décrites plus haut : trouver un emplacement occupé par le bon type de vélo, *bicycle.registerObserver(user), bicycle.removeObserver(station)*, mettre à jour *rentTime* et *rentStation* dans user, et enfin mettre à jour les statistiques de la station. Cette méthode attrape toutes les

exceptions qui pourraient être renvoyées par les méthodes appelées. Dans ce cas, il renvoie une exception « User (name) couldn't rent : (exception) ». De cette manière, on peut retrouver d'où provient l'exception lorsqu'un utilisateur essaye de réserver un vélo.

La seconde méthode, naturellement, est *returnRent(user, station, time)*. Elle est appelée lorsqu'un utilisateur souhaite rendre son vélo à la date *time*. Après avoir vérifié que l'utilisateur a bel et bien réservé un vélo, et que la date de retour n'est pas antérieure à celle de location, cette méthode appelle les méthodes présentées dans Bicycle : *bicycle.registerObserver(station)*, *bicycle.removeObserver(user)*. Au passage, si la station est une station plus et que l'utilisateur possède une carte d'abonnement, il gagne 5 minutes de *time credit*. Une fois cela fait, la méthode *charge()* (présentée ci-dessous) est appelée afin de faire payer l'utilisateur. Enfin, les différentes statistiques sont mises à jour, et en particulier un objet *Rent* est créé, puis ajouté à l'historique des réservations. Cette méthode attrape toutes les exceptions qui pourraient être renvoyées par les méthodes appelées : dans ce cas, il renvoie une exception « User (name) couldn't return : (exception) ».

La dernière méthode importante est la méthode *charge(user, type, rentTime, returnTime).* Elle est appelée dans *returnRent()* pour facturer l'utilisateur, c'est-à-dire appeler *user.cost()* pour calculer le coût de la réservation, puis mettre à jour le solde sur sa *credit card*. Cette méthode attrape toutes les exceptions qui pourraient être renvoyées par les méthodes appelées : dans ce cas, il renvoie une exception « Couldn't charge User (name) : (exception) ».

MyVelib inclut également d'autres méthodes comme findStation(id), findUser(id) et findBicycle(id), qui prennent comme un argument un identifiant (int) et renvoient la station, l'utilisateur ou le vélo correspondant. MyVelib *override* lui aussi *toString()*, afin d'afficher des informations comme la liste des stations, leur statut et leur type ; la liste des utilisateurs et leur état (s'ils sont en réservation ou non), idem pour les vélos.

j. Comparators

Deux comparateurs ont également été implémentés afin de répondre au cahier des charges : UsageComparator et OccupyingComparator. Ils implémentent Comparator<Station>, et servent à trier la liste des stations de MyVelib selon l'usage ou le taux d'occupation, comme décrit dans le sujet.

2. Interface graphique et CLUI

L'énoncé suggérait de coder une CLUI (Command-Line User Interface). Nous avions deux possibilités : utiliser le terminal du logiciel Papyrus ou coder notre propre interface graphique pour simuler un terminal et permettre à l'utilisateur d'entrer ses commandes.

Nous avons opté pour la deuxième solution. En effet, utiliser notre propre interface graphique nous permet de nous affranchir de Papyrus. C'est une bonne chose puisque, si l'on s'éloigne du cadre fermé du cours de CentraleSupélec, l'application que nous avons créée aurait vocation à être distribuée à des clients. Or on voit mal demander aux clients d'installer Java et Papyrus pour pouvoir exécuter notre code. Avec l'interface graphique, il suffit d'exécuter la fonction main pour que la fenêtre s'affiche, indépendamment de l'IDE utilisé.

L'interface graphique a été créée en utilisant la classe JFrame, qui représente une fenêtre. La fenêtre se décompose en plusieurs widgets (éléments) : une zone de texte (JTextEdit) doit permettre à l'utilisateur de rentrer ses commandes, et une zone de texte en affichage doit permettre d'afficher la sortie : soit que la syntaxe employée par l'utilisateur est incorrecte (par exemple si une commande est mal écrite ou s'il manque des arguments à une commande), soit que l'opération demandée a bien été effectuée. La plupart des opérations n'ont pas de sortie (quand un nouvel utilisateur est ajouté au réseau, il n'y a pas besoin d'afficher quoi que ce soit, on affiche donc une confirmation que l'utilisateur a bien été ajouté), mais certaines en ont une : toutes les opérations relatives à l'affichage du réseau, d'une station ou encore d'un utilisateur. Nous avons aussi ajouté un widget de texte (JLabel) incitant l'utilisateur à entrer ses commandes dans la zone de texte adéquate.

Pour le widget matérialisant la sortie, nous aurions pu opter pour un JLabel mais cette classe ne semblait pas gérer les textes s'étendant sur plusieurs lignes. Nous avons donc opté pour un JTextEdit, qui permet d'afficher un texte sur plusieurs lignes. Nous avons désactivé le caractère modifiable de ce JTextEdit, si bien que l'utilisateur ne peut pas modifier le texte affiché en sortie. Le seul texte modifiable par l'utilisateur est donc le texte qui matérialise ses commandes, notre interface graphique se veut donc simple et intuitive. Pour séparer les affichages normaux des affichages d'erreur, les messages normaux s'affichent en noir tandis que les messages d'erreur s'affichent en rouge.

Figure 2 : La fenêtre de notre application. On retrouve en 1 le label de texte guidant l'utilisateur, en 2 la zone de texte en entrée et en 3 la zone de texte en sortie.

Il reste à trouver un moyen pour l'utilisateur de « valider » la commande qu'il a entrée. La solution la plus simple aurait été d'ajouter un bouton, que l'utilisateur aurait pu presser pour valider sa commande. Mais cette solution n'est pas très élégante, et rend l'utilisation de l'interface graphique plus lourde (entrer du texte peut se faire facilement, alors que cliquer sur un bouton nécessite le recours à une souris ou un pavé tactile, donc nécessite de bouger les mains et le curseur, et prend plusieurs secondes). Heureusement, nous avons implémenté une autre solution : chaque fois que l'utilisateur appuie sur la touche Entrer, la commande est exécutée. Cette solution présente deux avantages : elle est très intuitive, et permet aussi un retour à la ligne automatique entre deux commandes. L'utilisateur peut donc accéder facilement à la liste des commandes déjà passées puisque celle-ci est affichée là où les commandes ont été entrées.

L'interaction entre l'appui sur la touche Entrer et la mise à jour de la commande (i.e. l'envoi de la commande au cœur de l'application) se fait grâce à la classe java.ewt.event.KeyEvent (que nous nommerons KeyEvent dans la suite) et l'interface java.ewt.event.KeyListener (KeyListener). Le fonctionnement de ces entités est simple : on crée un objet de type KeyListener, ce qui nous permet de réimplémenter trois fonctions : keyTyped(KeyEvent arg0), keyReleased(KeyEvent arg0) et

keyPressed(KeyEvent arg0). Chaque fois que la touche X sera tapée, relâchée ou pressée (respectivement) un objet de type KeyEvent(keyCode = KeyEvent.X) sera généré. Il suffit ensuite de vérifier que la touche X est bien la touche Entrer (KeyEvent.VK_ENTER) pour appeler les fonctions que nous désirons appeler lorsque la touche Entrer est utilisée.

Une fois que l'utilisateur a fini d'entrer sa commande et a utilisé la touche Entrer, le texte entré par l'utilisateur est analysé. Plus précisément, il est d'abord découpé en lignes. En effet, appuyer sur la touche Entrer, en plus de générer le KeyEvent susmentionné, ajoute un retour à la ligne au texte en entrée ; toutes les commandes sont donc sur une ligne différente. On ne considère que la dernière commande (les autres ont déjà été exécutées), donc la dernière ligne de la zone de texte en entrée. Mais il faut éviter que cette commande soit vide, au cas où l'utilisateur souhaiterait sauter plusieurs lignes entre les commandes. On considère donc la dernière commande non vide.

Cette commande est envoyée à la classe InputCLUI (en fait à sa fonction `checkSyntax`), qui sert essentiellement à vérifier que la syntaxe employée est correcte. Nous nous sommes strictement conformés à la syntaxe donnée par l'énoncé, même si nous avons jugé qu'elle n'était pas très pratique (notamment en ce qui concerne le nom du réseau, étant donné qu'on ne gère qu'un réseau à la fois et que la demande du nom du réseau n'apparaît que dans certaines commandes mais pas toutes).

La vérification de la syntaxe se fait en découpant la commande en mots, les espaces surnuméraires étant supprimées. Ainsi, la commande `setup network` (avec 1 espace) sera équivalente à la commande `setup  network` (avec plusieurs espaces). Le premier mot correspond au type de commande (setup, addUser, display, sortStation…), les suivants aux arguments. La fonction renvoie une erreur lorsque le nombre d'arguments n'est pas correct, en indiquant la bonne syntaxe pour que l'utilisateur puisse facilement réécrire sa commande d'une manière qui pourra être comprise par la CLUI.

C'est ensuite la fonction analyzeText de CommandWindow.java qui va effectuer l'analyse proprement dite, en récupérant les arguments de la commande et en appelant les fonctions adéquates du cœur de l'application.

b. Tests

1. Cœur du projet

Les différentes méthodes du core sont testées au travers de 10 fichiers JUnit. Comme décrit plus tôt dans ce rapport, un soin particulier a été accordé aux messages portés par les exceptions du programme, afin qu'il soit facile d'identifier d'où vient l'erreur facilement. Ainsi, les tests qui testaient si une exception était bien renvoyée dans les cas problématiques devaient vérifier (par exemple) : `assertTrue(e.getMessage().contains("No parking slot available in station"))`.

Il est important de noter que, au lieu de multiplier les méthodes de test, nous avons préféré tester plusieurs assertions en une seule fois, au sein d'une même méthode de test. En ce qui concerne les tests concernant des positions (comme dans le ride planning), nous générons aléatoirement la position des stations au début de chaque test ; on peut donc relancer plusieurs fois le test afin de vérifier les assertions.

Toutes les méthodes de test portent des noms clairs, afin de comprendre ce qu'ils testent exactement.

2. CLUI

Les tests conduits sur la CLUI sont de deux types.

Tout d'abord, la vérification de la syntaxe a été abondamment testée. De nombreux cas de figure ont été abordés, incluant un utilisateur se trompant dans le nombre ou dans le type des arguments, ou encore fournissant des arguments fantaisistes (par exemple du mauvais signe). La fonction de vérification de la syntaxe renvoyant une exception lorsque la syntaxe employée n'est pas correcte, il a fallu, dans les tests devant générer une exception, récupérer l'exception et vérifier que le message qu'elle portait était correct, le test échouant si aucune exception n'était levée.

Pour la fenêtre en elle-même et l'appel des bonnes commandes, il n'a pas été possible d'écrire des tests JUnit traditionnels parce que la structure des fonctions employées ne s'y prêtait pas. Toute cette partie repose en effet sur un affichage, et la chaîne de caractères renvoyée par la fonction analyzeText est directement utilisée pour être affichée. Il est donc impossible de récupérer les variables intermédiaires comme l'état de telle ou telle station. Nous aurions pu modifier le code pour y avoir accès, mais nous trouvions incohérent de modifier la structure du code pour le rendre testable, pour devoir ensuite remettre la structure initiale. Nous avons donc choisi, en guise de tests, d'exécuter un certain nombre de scénarios et de voir si les sorties étaient conformes à ce qui était attendu. Les tests ont donc été effectués manuellement directement via l'interface graphique, ainsi il n'y en a aucune trace dans un fichier JUnit. Toutes les commandes ont cependant été testées. C'est le seul endroit dans le projet où une politique de tests traditionnelle n'a pas pu être mise en place.

c. Limites de nos solutions

1. Partie 1

Pour cette partie, la principale limite de notre implémentation se trouve dans le ride planning. En effet, dans l'état actuel du projet, le ride planning est une fonction qui est appelée à un instant t, et qui donne les meilleures stations de départ et d'arrivée pour cet instant t. Cela veut dire que l'utilisateur ne sera pas averti si, entre le moment où il appelle le ride planning et le moment où il arrive à la station de départ (ou d'arrivée), l'une des stations n'est plus disponible (trop de vélos, plus de vélos, ou simplement hors ligne). Afin de régler ce problème, il faudrait que le ride planning observe l'état des différentes stations du système, qu'il soit averti lorsqu'il y a une mise à jour importante, puis qu'il avertisse à son tour l'utilisateur qui utilise le ride planning. Pour implémenter ceci, on pourrait utiliser un Observer design pattern, avec le Ride Planning qui observerait le système MyVelib.

Outre les politiques de ride planning optionnelles que nous n'avons pas eu le temps d'implémenter, nous regrettons également de ne pas avoir eu le temps de paralléliser notre code, afin de pouvoir faire tourner plusieurs threads simultanément et ainsi reproduire un vrai système de location de vélos en temps réel.

2. Partie 2

Nous avons choisi une interface graphique, et que toutes les commandes soient exécutées via un langage spécifique (décrit plus loin dans la partie concernée aux différentes commandes prises en charge par l'interface graphique).

Malheureusement, nous n'avons pas eu le temps de recréer une commande d'interface graphique pour chaque fonctionnalité effectivement implémentée, ce qui signifie que certaines fonctionnalités, bien que codées, ne sont pas accessibles via la CLUI mais seulement via le code .java lui-même. Il est par exemple impossible, via la CLUI, de construire un réseau sur un carré de côté 4 km, alors qu'un

constructeur adéquat existe bien dans la classe MyVelib. Toute la partie sur la simulation d'itinéraire le plus rapide est aussi absente de la CLUI.

En plus de cela, la CLUI a un défaut majeur : elle ne fait que de l'affichage et ne gère pas de variables internes. Même dans le cas où la recherche d'itinéraire était implémentée sur la CLUI, il serait par exemple impossible, dans un scénario, de dire « Alice va prendre le chemin le plus court pour aller d'un point A à un point B ». En effet, quand bien même on pourrait afficher la station de départ de l'itinéraire le plus rapide, on ne pourrait pas récupérer cette station et dire à Alice d'y aller. En revanche, cela serait possible à condition de ne pas passer par un fichier scénario : on pourrait afficher la station à emprunter (et donc son identifiant), puis dire à Alice d'aller prendre un vélo à la station dont l'id a été affiché juste avant.

d. Répartition du travail

Tous les choix d'implémentation ont été faits à deux pour que nous ayons chacun en tête la structure du projet. Nous avons pris le temps de réfléchir ensemble à comment nous allions nous organiser, nous avons analysé le sujet et dessiné le diagramme UML ensemble. Ce n'est qu'ensuite que nous nous sommes réparti le travail.

Il est à noter que les auteurs (Kellysto correspondant à Quentin Delacroix (Q.D.) et pioll à Pierre Ollivier (P.O.)) indiqués en haut des différents fichiers ne sont pas forcément représentatifs. Certaines classes ont en effet bénéficié d'allers-retours, la partie la plus basique ayant été codée par l'un d'entre nous et le reste des fonctionnalités par l'autre. Cela l'est encore plus pour les tests, où certains noms d'auteurs sont différents.

Partie	Réflexion préalable	Code	Tests	Rapport
Package 1 (cœur du projet)	Tous	Essentiellement Q.D.	Tous	Q.D.
Package 2 (CLUI)	Tous	P.O.	Essentiellement Q.D.	P.O.

Figure 3 : Répartition des tâches au sein du projet

Comme mentionné ci-dessus, la vérité est un peu plus contrastée que ce qui est présentée ci-dessus. « Tous » signifie que la tâche a été répartie de manière à peu près équitable. En tout, la répartition des tâches était équitable, il n'y a pas eu de déséquilibre entre nous sur notre investissement dans le projet.

2. Prise en main du projet et tests

a. Fichier de configuration .ini

Un fichier `my_velib.ini` est inclus dans le projet, directement dans le dossier src (l'application y accède à partir de son adresse, donnée par la commande `Thread.currentThread().getcontextClassLoader().getResource(''my_velib.ini'').getFile()`).

Ce fichier permet d'ajouter quelques utilisateurs au réseau choisi pour ne pas commencer avec un réseau totalement vide. Ainsi, il ajoute quatre utilisateurs, Alice, Bob, Carol et Daphne, avec les spécifications suivantes :

Nom	Longitude	Latitude	Solde bancaire	Type de carte	Temps restant sur la carte
Alice	6,8 km	4,1 km	0 €	Vlibre	20 min
Bob	6,8 km	4,1 km	50 €	Vmax	30 min
Carol	1,0 km	2,7 km	0 €	Vlibre	0 min
Daphne	8,4 km	9,5 km	50 €	(pas de carte)	(0 min)

Figure 4 : Les différentes spécifications incluses dans le fichier my_velib.ini

La syntaxe, pour le modifier, est la suivante : pour chaque utilisateur à ajouter, ajouter une ligne de la forme

`<nom> <longitude> <latitude> <solde bancaire> <type de carte> <tempsRestant>`
en posant tempsRestant = 0 si l'utilisateur n'a pas de carte de fidélité.

Ainsi, quatre utilisateurs sont ajoutés. Deux le sont au même endroit, deux sur quatre ont un solde bancaire strictement positif, un n'a pas de carte, un a une carte de type Vmax et deux de type Vlibre, enfin deux parmi les trois qui ont une carte ont un temps restant non nul sur celle-ci.

b. Description des scénarios

Nous avons élaboré plusieurs scénarios pour tester notre application et pour nous assurer qu'elle donne bien les bons résultats (comme indiqué ci-dessus).

Un premier scénario, inclus dans le projet, s'intitule « TestScenario1.txt ». Il correspond aux actions suivantes :

- Créer un réseau system1, constitué de 10 stations contenant chacune 10 places de parking. Les 10 stations sont en ligne, 7 d'entre elles sont de type standard et 3 de type plus. Les 10 stations sont situées aléatoirement dans un carré de côté 10 km (représenté par des paramètres latitude et longitude allant de 0 à 10 – ces grandeurs ne désignent donc pas vraiment des degrés et il faudrait passer par une conversion pour obtenir une valeur de type longitude = 2.133370).
- Charger le fichier my_velib.ini dans le réseau system1. Comme vu dans la section précédente, cela permet d'ajouter quatre utilisateurs, Alice, Bob, Carol et Daphne, au réseau.
- Afficher le réseau system1.
- Afficher Alice.
- Alice loue le vélo mécanique le plus proche d'elle.
- Afficher Alice.
- Alice se déplace aux coordonnées (1,2 ; 2,7).
- Le déplacement d'Alice a duré 20 minutes. Elle dépose son vélo le plus près possible de sa nouvelle localisation.
- Afficher Alice.
- Alice décide ensuite de faire une escapade à vélo. Elle loue le vélo électrique le plus proche pour douze heures, et le restitue ensuite au plus près de la position (5,5 ; 5,5).
- Afficher Alice.
- Afficher le réseau system1.

Ce scénario est réaliste dans le sens où il teste les actions les plus simples : louer un vélo, le restituer, payer – avec la carte Vlibre dont dispose Alice sur laquelle il y a un crédit de temps de 20 minutes. Nous l'avons exécuté plusieurs fois, ce qui a permis de vérifier le bon fonctionnement du système de prime dans le cas où des stations plus intervenaient.

Nous avons élaboré un deuxième scénario pour nos tests, qui ne s'appuie pas sur l'interface graphique. Ce scénario se trouve dans la classe *UseCaseScenario.java*, il est exécuté au sein de la méthode main. Concrètement, ce scénario reprendre le « Use Case Scenario » décrit dans le sujet, à la fin de la partie I.

Le scénario commence avec la création de quatre utilisateurs, Alice, Bob, Carol et Daphne, de la même manière que ce qui a été fait dans le fichier my_velib.ini.

Alice loue un vélo mécanique à la station dont l'identifiant vaut 2 (qui est une station standard) à la date rentTime (qui correspond au jour et à l'heure où le scénario est lancé). Après 145 minutes, elle restitue son vélo à la station dont l'identifiant vaut 8 (qui est une station plus). Dans le même temps, à la date rentTime, Bob loue un vélo électrique à la station plus d'identifiant 8 (en même temps donc qu'Alice loue son vélo mécanique). Daphne, quant à elle, est initialement située aux coordonnées (8,4 ; 9,5). Elle cherche à se rendre aux coordonnées (1,0 ; 1,0). Pour ce faire, elle cherche l'itinéraire le plus rapide utilisant un vélo électrique. Puis elle loue son vélo électrique pour suivre son itinéraire.

Une fois toutes ces actions prises en compte, un affichage est réalisé pour afficher l'ensemble des opérations de location ayant eu lieu, les utilisateurs et les stations. Enfin, les stations sont triées selon leur taux d'usage et leur taux d'occupation.

c. Récapitulatif des commandes de la CLUI

Il s'agit ici de récapituler l'ensemble des commandes qui peuvent être soumises à la CLUI.

- setup <velibNetworkName> : permet de créer un nouveau réseau, de nom velibnetworkName. Le réseau créé possède 10 stations, chacune d'entre elles étant dotée de 10 places de parking. Les stations sont réparties sur un carré de 10 km de côté, et 70 % des places de parking sont initialement occupées par un vélo. Parmi les vélos, 70 % sont mécaniques et 30 % sont électriques.

- setup <velibNetworkName> <nstations> <nslots> <s> <nbikes> : même chose mais avec un nombre de stations égal à nstations, un nombre total de places de parking égal à nslots, un carré de côté s km et un nombre total de vélos égal à nbikes. La proportion de 70 % de vélos mécaniques est conservée.

- addUser <userName> <cardType< <velibNetworkName> : ajoute un utilisateur au réseau velibNetworkName, dont le nom est donné par userName, le type de carte par cardType (son crédit est initialement fixé à 0 min).

- offline <velibNetworkName> <stationId> : met hors-ligne la station du réseau velibNetworkName dont l'id est stationId, si elle existe.

- online <velibNetworkName> <stationId> : met en ligne la station du réseau velibNetworkName dont l'id est stationId, si elle existe.

- rentMechanicalBike <userId> <stationId> : symbolise la location par l'utilisateur dont l'id est userId d'un vélo mécanique à la station stationId.

- rentElectricalBike <userId> <stationId> : symbolise la location par l'utilisateur dont l'id est userId d'un vélo électrique à la station stationId.

- rentBike <userId> <stationId> : symbolise la location par l'utilisateur dont l'id est userId d'un vélo mécanique à la station stationId (résultat strictement identique à rentMechanicalBike).

- rentClosestMechanicalBike <userId> : symbolise la location par l'utilisateur dont l'id est userId d'un vélo mécanique à la station la plus proche contenant un vélo mécanique disponible.

- rentClosestElectricalBike <userId> : symbolise la location par l'utilisateur dont l'id est userId d'un vélo électrique à la station la plus proche contenant un vélo électrique disponible.

- rentClosestBike <userId> : symbolise la location par l'utilisateur dont l'id est userId d'un vélo mécanique à la station la plus proche contenant un vélo mécanique disponible (résultat strictement identique à rentClosestMechanicalBike).

- Les six mêmes commandes mais avec return à la place de rent et un paramètre <duration> en plus à la fin : il s'agit de la restitution et non de la location. La restitution se fait au bout d'un certain temps (la durée de location), cette durée, exprimée en minutes, est contenue dans la variable duration.

- displayStation <velibNetworkName> <stationId> : affiche le contenu de la station du réseau velibNetworkName dont l'id est stationId, si elle existe. L'affichage comprend le numéro de la station, son statut en ligne ou hors-ligne, son type (standard ou plus), sa latitude, sa longitude, son nombre total de places de parking, le nombre total de vélos qui y sont garés, et, pour chaque place de parking, son identifiant et si elle est occupée ou non.

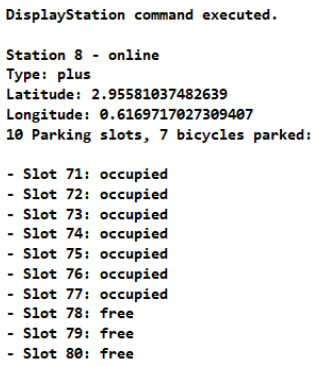

Figure 5 : Affichage du contenu d'une station via la CLUI

- displayUser <velibNetworkName> <userId> : affiche des informations sur l'utilisateur du réseau velibNetworkName dont l'id est userId, s'il existe. L'affichage comprend le nom de l'utilisateur, son identifiant, sa latitude, sa longitude, s'il loue actuellement un vélo ou non, l'identifiant de sa carte de crédit ainsi que son solde s'il en possède une, sa dette totale envers le système, l'identifiant, le type et le solde de sa carte de fidélité s'il en possède une, le temps de crédit total gagné via les stations plus, et enfin son nombre total de voyages et son temps de location total.

```
DisplayUser command executed.

User name:Carol
User id:3
Latitude:1.0
Longitude:2.7
Not riding a bicycle
Credit card id:3
Credit card balance:0.0
Total charges:0.0
Registration card id:3
Registration card type:Vlibre
registration card time balance:0
Total time credit earned:0
Number of rides:0
Total rent time:0
```

Figure 6 : Affichage du contenu d'un utilisateur via la CLUI

- sortStation <velibNetworkName> <sortPolicy> : trie les stations du réseau velibNetworkName par ordre décroissant en fonction de sortPolicy. Deux valeurs sont possibles pour sortPolicy : usage (tri décroissant selon l'usage des stations) et occupation (tri décroissant selon l'occupation des stations).

- display <velibNetworkName> : affiche des informations sur le réseau velibNetworkName. Ces informations comprennent l'identifiant unique du système velibNetworkName, son nombre de stations, son nombre de stations standard, son nombre de stations plus, un récapitulatif de toutes les stations contenant leur numéro, leur type et si elles sont en ligne ou non, le nombre total de places de parking, le nombre total de vélos, le nombre total de vélos électriques, le nombre total de vélos mécaniques, le nombre total d'utilisateurs, et enfin un récapitulatif de tous les utilisateurs, comprenant leur nom, leur identifiant et s'ils conduisent actuellement un vélo ou non.

```
Display command executed.

myVelib system id:1
Stations:10
Standard stations:7
Plus stations:3
- standard Station 1: online
- standard Station 2: online
- standard Station 3: online
- standard Station 4: online
- standard Station 5: online
- standard Station 6: online
- standard Station 7: online
- plus Station 8: online
- plus Station 9: online
- plus Station 10: online
Parking slots:100
Bicycles:70
Mechanic bicycles:49
Electric bicycles:21
4 users:
- Alice id 1: Not riding a bicycle
- Bob id 2: Not riding a bicycle
- Carol id 3: Not riding a bicycle
- Daphne id 4: Not riding a bicycle
```

Figure 7 : Affichage du contenu d'un réseau via la CLUI

- runtest <filename> : joue le scénario dont l'adresse est spécifiée dans la variable filename.

- setCoordinates <userId> <x> <y> : simule un déplacement de l'utilisateur dont l'identifiant unique est userId. Ses nouvelles coordonnées sont données par x (pour la longitude) et y (pour la latitude).

- setBalance <userId> <balance> : modifie le solde bancaire de l'utilisateur dont l'identifiant unique est userId, la nouvelle valeur étant spécifiée via la variable balance.

- setTimeBalance <userId> <timeBalance> : modifie le temps restant sur la carte de fidélité de l'utilisateur dont l'identifiant unique est userId, la nouvelle valeur étant spécifiée via la variable timeBalance.

- loadIni <velibNetworkName> : charge le contenu du fichier my_velib.ini dans le réseau velibNetworkName.

La syntaxe de certaines commandes était imposée par le sujet, nous ne sommes donc pas revenus dessus, quand bien même nous avons jugé peu pratique – et peu réaliste – l'obligation de devoir systématiquement ou presque (pas dans rentBike ou returnBike) le nom du réseau utilisé. Dans notre monde, il est rare que deux systèmes de vélos en libre-service aient le même opérateur, et, même dans ce cas, il suffirait de créer plusieurs copies de l'application, une par réseau, pour que les réseaux soient bien différentiés. L'interface graphique serait plus agréable à utiliser s'il ne fallait pas spécifier à chaque fois ou presque le nom du réseau alors que c'est au fond toujours le même (system1 ou network par exemple).

Conclusion

Malgré le peu de temps qui nous était alloué par l'école pour le projet, nous pensons avoir créé une solution qui répond au problème posé dans le sujet. Les commandes essentielles (location et restitution d'un vélo, prise en compte du coût en fonction de la carte de fidélité) sont fonctionnelles et notre interface graphique permet à l'utilisateur d'entrer ses commandes de manière relativement instinctive, un peu comme dans un terminal.

Il reste évidemment beaucoup à faire. En priorité, il faudrait compléter l'interface graphique avec les commandes manquantes, et en ajouter d'autres pour pouvoir gagner en flexibilité (et par exemple pouvoir se passer de l'argument velibNetworkName). On pourrait ensuite s'intéresser à des formes de ville non carrées ou avec une répartition non-uniforme des stations, à de nouvelles politiques de tri. Avec un plan de ville réel, nous pourrions envisager de nommer les stations, et de proposer une recherche d'itinéraire incluant la présence éventuelle de relief dans la ville. Nous pourrions aussi implémenter les options d'itinéraire proposées dans le sujet, pour optimiser la répartition des vélos, tant électriques que mécaniques, au sein des stations. Un point, mineur et en même temps essentiel, est que la commande returnBike de la CLUI est rendue équivalente à returnElectricalBike, alors qu'il suffirait de considérer le type du vélo détenu par l'utilisateur concerné. De même, on pourrait imaginer assigner à chaque utilisateur un type de vélo (électrique ou mécanique) par défaut, et cette préférence pourrait être appliquée par les commandes rentBike et rentClosestBike (en optant par exemple pour le deuxième type dans le cas où le type préféré ne serait pas disponible).

L’application telle qu’elle est actuellement n’est pas optimisée pour le partage à un client. Il faudrait pouvoir compiler le code et générer un fichier exécutable, qui puisse être ouvert à partir de n’importe quel ordinateur sans que le langage Java soit forcément installé (ce qui est possible si le compilateur convertit le code Java en code binaire). On pourrait ensuite étendre à d’autres plateformes : les smartphones, qui occupent une place grandissante dans notre quotidien, les tablettes…).

Une autre possibilité d’évolution serait de mettre en place une architecture client-serveur. En effet, on imagine bien, en pratique, que les commandes de type addUser sont générées par le site de Vélib lorsqu’on y crée un compte, ou encore que la commande setBalance est appliquée soit via l’application Vélib, soit via les bornes. Ajouter une architecture client-serveur impliquerait cependant de répondre à des problèmes d’accès concurrentiel, par exemple si deux utilisateurs décident simultanément de réserver le dernier vélo disponible dans une station.