CentraleSupélec

IS1220 - Object Oriented Software Design

Myvelib Bike Sharing System

# Rapport Final de Projet

*Auteurs:*
Ahmed Djermani
Roméo Sandoz

*Professeur:*
Paolo Ballarini

April 8, 2018

CentraleSupélec

# Sommaire

# 1 Introduction

L'objectif ce de projet est de développer en langage Java une plateforme de location de vélos, à la manière du système Velib'. A partir d'un cahier des charges détaillant toutes les fonctionnalités requises, et de quelques cas d'utilisation, nous avons donc d'abord déterminé la stratégie à adopter afin de modéliser ce système et ses interactions internes.
Nous avons donc commencé par construire ensemble un diagramme UML représentant les différentes classes et méthodes qui constitueraient notre programme. Pour cela, nous nous sommes inspirés de la plateforme Vélib', en imaginant comment notre système serait utilisé par de véritables abonnés (commande de certaines actions à distance, retour du vélo directement à une borne, etc). Ensuite, nous nous sommes appuyés notamment sur les « modèles » (pattern) vus en cours, qui permettent d'obtenir un code « Open-Close », c'est-à-dire adaptable. Ainsi, dans le cas où la plateforme est amenée à évoluer, en ajoutant de nouvelles fonctionnalités, de nouveaux types d'objets (cartes, vélos, utilisateurs ...), le code sera facile à adapter, et ne nécessitera principalement que des ajouts, et très peu de modifications. Dans un second temps, nous avons programmé le système en Java, à partir des classes obtenues sur le diagramme UML. Pour cela, nous nous sommes répartis les différentes parties du code, en essayant de confier à une même personne les classes les plus interdépendantes. La précision du diagramme UML fut primordiale dans cette tâche, afin que chaque programmeur puisse utiliser les classes codées par son collègue, en sachant exactement comment elles devaient fonctionner. Nous avons aussi décidé de coder en aval les JUnit Test correspondant aux classes de l'autre, afin de s'assurer du fonctionnement des classes tel qu'il était prévu à l'origine, dans l'optique du « Test Driven Development ». Ce projet nous a permis de mettre en pratique les divers aspects du cours tant dans la partie code (polymorphisme, classes abstraites, class nesting, exceptions...) que dans la partie planification, notamment à travers les patterns (observer, visitor, strategy, singleton, factory).

# 2 Conception du système

## 2.1 Contexte

L'intensification du traffic routier urbain, la densification des villes, la prise de conscience sociale et sanitaire sont autant de facteurs ayant entraîné la hausse du nombre de vélos dans nos villes. Les plus petits comme les plus grands remontent les artères principales des grands agglomération pour se rendre à l'école, au travail ou simplement s'aérer. Mais où déposer son vélo dans une ville où les vols sont fréquents ? Ce macrophénomène ne se propage pas sans contraintes sur les politiques publiques et exige des municipalités la mise en place d'un réseau centralisé permettant à chacun de disposer d'un véhicule collectif et offrant des points de collecte judicieusement situés. Les grandes villes ont rapidement offert à leurs citadins des systèmes de grande ampleur telles que Vélov à Lyon ou Vélib' à Paris.Seulement, une telle infrastructure représente un vrai défi organisationnel : il s'agit de créer un système centralisé, accessible par tous depuis chaque station et chaque smartphone, possédant un nombre limité de stations et de vélos, situés à des endroits bien précis, plus ou moins idéalement pour les usagers. Toutes ces spécificités sont autant de contraintes pour le chef de projet, dont l'objectif est de déterminer la modélisation la plus optimale possible. C'est là que notre équipe intervient en tant que concepteur et programmeur du système.
En quelques mots, le système MyVelib doit :

- Rajouter les *utilisateurs* dans la base de données selon le type d'abonnement qu'ils possèdent.
- Rajouter des *stations* selon le type (Plus ou Standard), leur attribuer un nombre d'emplacements et lui donner les moyens d'interagir avec les utilisateurs à distance (planification de voyages) et via un terminal de commande (emprunt et retour de vélo).
- Être capable de collecter les différents *parcours*, de les créer selon la stratégie de parcours employée par l'utilisateur.
- Attribuer à chaque élément physique une *position* dans l'espace et de créer un système de *temps* efficace.

### 2.2 Classes primaires

Lors de la conception de notre programme, nous avons déterminé les classes primaires, les briques du programme en quelque sorte. **Network**

**:** Classe principale qui contient toutes les informations du système (listes des vélos, utilisateurs, stations, courses), dans laquelle les autres parties du programme peuvent venir chercher les informations nécessaires à leur fonctionnement. (Voir figure 1) Il s'agit de la base de données du système. **Network** possède des constructeurs et une liste statique de *networks* : ainsi, on peut enregistrer les différents réseaux du programme et implémenter les méthodes et attributs de chaque réseau dans la même classe.

**Station :** Cette classe représente les stations et est donc centrale dans la location et les retours de vélos par les utilisateurs. Elle doit donc gérer ces différentes situations ainsi que le paiement, en essayant d'en cacher la plus grande partie à l'utilisateur. Chaque station possède une liste de slots de type **Slot** qui est défini comme une classe à l'intérieur de la classe **Station** elle-même : c'est donc une *nested-class* ayant accès à tous les attributs de la station. Une action sur un slot (receiveBike()) est d'abord gérée localement puis après traitement propre, renvoyée au niveau supérieur, c'est-à-dire la station qui s'occupe des tâches plus globales (appel de la méthode returnBike() de l'utilisateur puis calcul du coût). Le terminal est aussi une *nested-class* dont la fonction principale est de clarifier le code en concentrant les fonctions de calcul : c'est lui qui gère les calculs de statistiques. (Voir figure 1)

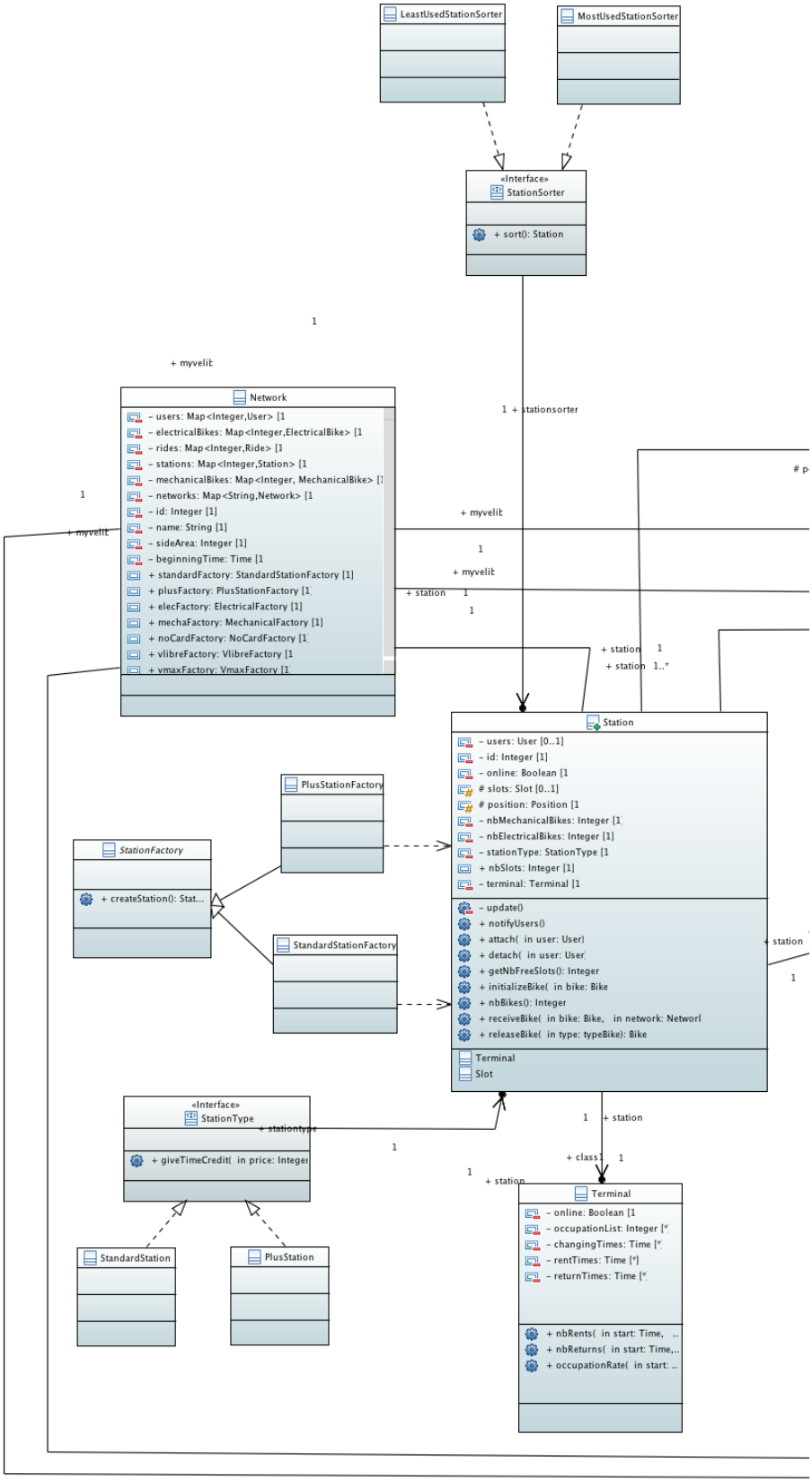


Figure 1: Network and Station UML class diagram.

**User :** Classe chargée de représenter les utilisateurs, et donc liée à toutes leurs potentielles actions (location de vélo, planification de trajet, paiement, etc). Elle est centrale car interagit avec beaucoup d'autres parties du programme. La classe User stocke les statistiques de l'utilisateur (montant total chargé, temps total passé sur un vélo, crédit de temps gagné) qui sont mises à jour à chaque action de celui-ci. Il lui est donné la possibilité de planifier un parcours selon une stratégie prédéfinie (plus court/long chemin, éviter/préférer les stations Plus, préserver l'uniformité du nombre de vélos) à partir de sa position actuelle. Selon le type de carte qu'il possède et le type de vélo qu'il souhaite utiliser, le prix du trajet varie. (Voir figure 2)

**Bike :** Classe représentant les vélos. Bien que ceux-ci soient évidemment présents partout, la classe elle-même contient assez peu de méthodes et attributs, elle est principalement utilisée par les autres parties. Deux classes instanciables héritent de Bike : ElectricalBike et MechanicalBike.(Voir figure 2)

**Card :** Cette classe correspond aux différentes cartes d'abonnés que l'utilisateur peut choisir de posséder. De par leur rôle important dans le calcul du coût d'un trajet, elles implémentent la méthode qui est chargée de ce calcul et sont donc un attribut décisif de la classe User.(Voir figure 2)

**Ride :** Cette classe n'était pas explicitement visible dans le cahier des charges mais est absolument indispensable. Elle représente les trajets effectués par les utilisateurs, et est donc omniprésente, que ce soit après la planification d'un trajet depuis l'application ou dès l'emprunt d'un vélo auprès d'une station. Elle simplifie beaucoup les calculs de coûts, le paiement et la planification de trajets. A chaque trajet est associé un temps de départ (retrait du vélo) qui servira au calcul du temps de parcours une fois terminé.(Voir figure 2)

**Time :** Le système de temps repose sur les modifications apportées à chaque action. Les commandes proposées par notre système doivent une contenir un paramètre time dont la valeur est ajouté à la classe. Le temps augmente donc par à-coups. Nous avions initialement choisi une classe héritant de **Date**, dont le temps est donné par la machine mais les durées de trajet n'étaient pas réalistes à l'échelle d'un test scénario que le processeur exécuterait en moins d'une seconde.

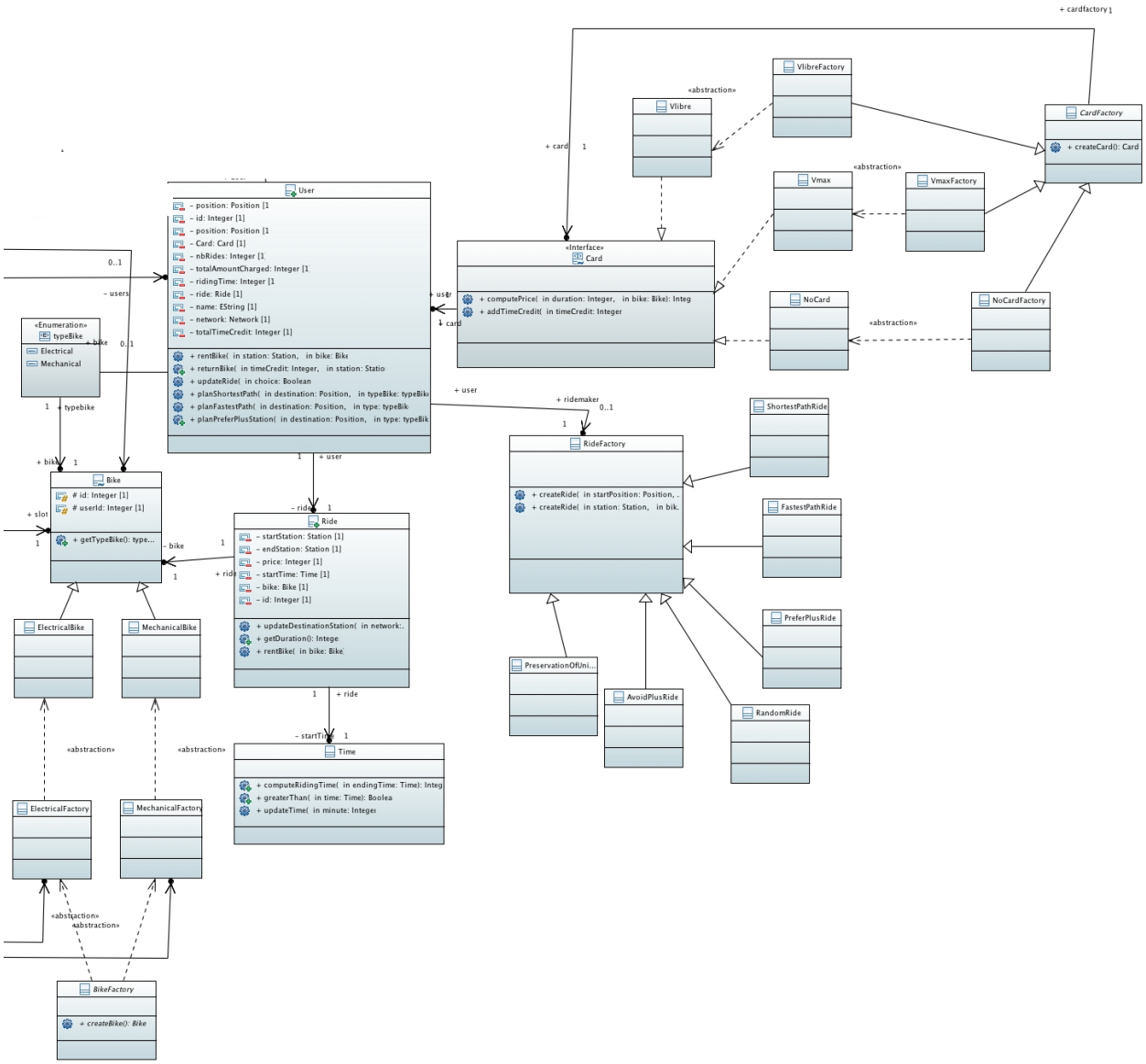


Figure 2: User, Bike, Card and Ride class diagram.

### 2.3 Patterns implémentés

Afin de rendre notre programme aussi peu sensible que possible aux modifications de fonctionnalités, nous avons eu recours à l'utilisation de différents patterns.

**Factory Pattern :** Nous avons utilisé ce pattern à maintes reprises, afin de séparer la création des différents types d'objets du programme client, et de s'adapter ainsi facilement à l'ajout de potentielles variantes de ces objets. Il est attaché aux classes Bike, Ride, Station et Card. Chaque sous-classe héritant de celles-ci correspond à une *factory* dont il suffit d'appeler la méthode *create...()* pour initialiser un nouvel objet qui s'ajoute automatiquement au réseau (par exemple, elecFactory de type *ElectricalBikeFactory* via *elecFactory.createElectricalBike()*).

**Strategy Pattern :** Ce pattern est très utile lorsque certains objets proches sont amenés à exécuter des actions différentes selon certaines de leurs caractéristiques. Il permet ainsi d'ajouter de nouveaux moyens de se comporter pour ces objets. Ainsi, nous avons pu l'exploiter sur les objets Card pour calculer le prix des courses, et sur les objets Station pour ajouter des crédits aux utilisateurs en fonction du type de station de retour. (Voir figure 3)

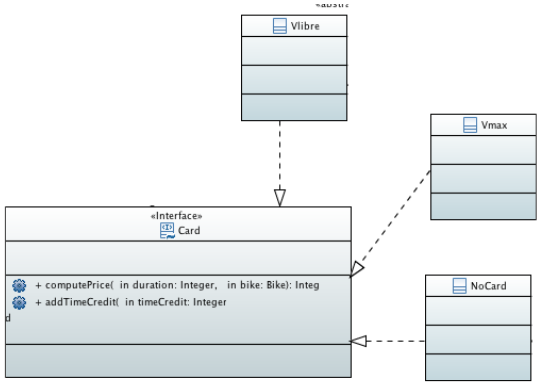


Figure 3: Card Strategy Pattern.

**Observer Pattern :** L'observer pattern permet à un objet (observer) d'être tenu au courant de toute modification d'un autre objet (observable). Dans notre cas, cela s'est avéré particulièrement utile lorsqu'il a fallu mettre à jour la destination des utilisateurs se dirigeant vers une station donnée lorsque celle-ci passe hors-ligne. La station est alors observable par les utilisateurs observers dont la liste est tenue dans la classe

Station.

**Singleton Pattern :** Le singleton pattern est assez simple, il permet de s'assurer qu'un objet n'est instancié qu'une fois. Il a donc été très utile pour les générateurs d'identifiants de chaque classe. Chaque générateur est associé à une classe dont il emprunte le nom : par exemple, StationIdGenerator permet de générer l'Id de chaque station à l'appel de son constructeur.

Le diagramme UML complet est disponible en Complément en fin de rapport.

### 2.4 Implémentation

Une fois le diagramme UML dessiné et les classes bien établies, l'implémentation concrète du code commence. Il s'agit tout d'abord de se répartir intelligemment les tâches de façon à ce que les classes les plus interdépendantes soient codées par la même personne.
Nous avons distingué deux blocs relativement indépendants reliés par un pont : le réseau, ie la classe Network. L'un construisait le bloc **Station - StationSorter** tandis que l'autre programmait le complexe **User - Card - Ride - Station**. Puis les deux blocs compatibles ont fusionné par l'intermédiaire de la classe Network pour former un tout cohérent et harmonieux.

| Tâches | Roméo Sandoz | Ahmed Djermani |
|---|---|---|
| Design | Liste des classes et méthodes | Liste des classes et méthodes |
| | Design UML diagramme | Design UML diagramme |
| Code : myVelib Core | Observer Pattern | Factory Patterns |
| | Strategy Patterns | Package UserandCard |
| | Package Station, Slot, Sorter | Package Ride et RidePlanning |
| | | Package System et Client Application |
| | | Package Utilities |
| Code : myVelib CLUI | | Design et programmation |
| Code : myVelib GUI | | Design et programmation |
| Tests | JUnit tests de Ahmed | JUnits test de Roméo Vérification du bon fonctionnement de la CLUI et de la GUI |
| Scénarios test | | Rédaction des scénarios et vérification du résultat |
| Rapport | Introduction, Classes principales, Patterns employés | Contexte, Tests, CLUI, GUI, Exceptions, Mise en page du rapport en Latex |

Figure 4: Répartition des tâches.

### 2.5 Gestion des exceptions

Cette pléthore de fonctionnalités ne vas pas sans problème : les classes étant fortement connectées entre elles et les méthodes très nombreuses, il est évident que bon nombre de "non conformités" apparaissent. Par exemple, que faire si l'utilisateur s'adresse à une station qui ne possède pas de vélo du type souhaité ? Que faire s'il essaye de retourner un vélo sur une station pleine ? Beaucoup d'autres questions similaires, intrinsèques à la nature même du problème doivent être anticipées et appréhendées par celui-ci. Toutes les éventualités doivent être envisagées pour éviter l'interruption voire l'arrêt complet du système. C'est pourquoi nous créons des Exceptions dont le but est d'indiquer la présence d'un dysfonctionnement et d'apporter la réponse adéquate, c'est-à-dire le comportement que doivent adopter les classes pour permettre au programme de suivre son cours. Ci-dessous une liste des exceptions crées pour pal-

lier au éventuelles interruptions du processus ainsi que leur description :

| Exception | Description |
|---|---|
| FreeException | Une classe tente de changer un état pour rien |
| FullStationException | Un usager tente de retourner un vélo sur une station pleine |
| MaxBikeException | Un usager essaie de louer un deuxième vélo |
| MaxRideException | Maximum de 1 ride par usager |
| MissingDataException | Manque de données entre t1 et t2 |
| NoDestinationAssignedException | Aucune destination d'arrivée n'est prévue |
| NotAnAnswerException | La donnée rentrée ne correspond à aucune réponse prédéfinie |
| NotAvailableException | Aucun vélo du type voulu n'est disponible à cette station |
| OccupiedException | Retour d' un vélo sur un emplacement occupé |
| OfflineException | Un usager tente une action sur une station hors-ligne |

Table 1: Liste des exceptions.

### 2.6 Command Line User Interface

Pour ouvrir la CLUI, rendez-vous dans la classe Interface (myVelibCLUI package) et lancez la méthode *main*. Vous pouvez entrer "runtest testScenario1.txt" pour lancer les scénarios. `my_velib`.ini, le fichier de configuration garantissant la présence d'au moins un réseau, est automatiquement chargé.

Remarque : il est normal que l'affichage dans la console semble lent; nous avons choisi de ponctuer le code de *Thread.sleep()* de sorte à ce que vous puissiez voir l'évolution du scénario sans devoir remonter entièrement la console.

La CLUI (Command Line User Interface) est un formidable outil permettant d'automatiser l'utilisation du service proposé. L'implémentation jusque-là proposée est certes fonctionnelle mais peu pratique dans la mesure où nous n'avons pas un moyen simple, efficace et pertinent de l'utiliser.
La CLUI que nous avons programmée permet de lire un fichier texte préalablement rempli de commandes et de les exécuter ligne par ligne. Il s'agit donc de fournir un moyen d'interpréter des lignes de texte en appels de méthodes de classes en y incluant des paramètres.
Pour cela, nous avons défini deux classes :

- **Interface :** elle contient la méthode *main()* et l'ensemble des paramètres du problème sous forme de String. La méthode *main()* appelle une seconde méthode *process()* dont le rôle est de lire les scénarios texte ligne par ligne en utilisant un **BufferedReader**. Chaque ligne est analysée : un appel de la méthode *split()* permet

de la tronquer en un tableau de mots correspondant pour le premier à un nom de commande et pour les autres à des arguments. *Process()* vérifie la conformité du nombre de paramètres avec la nature de la commande puis envoie le tableau à la classe **Processing** pour traitement.

- **Processing :** elle contient une liste de méthodes statiques de traitement (une pour chaque commande). Le tableau passé en paramètre d'une méthode est analysé : chaque élément correspond à un type d'entrée dont la conformité est vérifiée (méthode *checkUserId()* par exemple). Si tous les éléments sont valides, le traitement approprié est opéré (par exemple, *setup5params()* pour initialiser un nouveau réseau).

## 2.7 Graphical User Interface

Pour ouvrir la fénêtre de l'application, rendez-vous dans la classe ClientApplication (myVelibGUI package) et lancez la méthode *main*; ou lancez simplement MyVelibGUI.jar situé à la racine du projet.

Dans l'idée de rendre notre application encore plus facile d'accès, nous avons développé une GUI (Graphical User Interface) via laquelle nous pouvons exécuter des commandes sans aucune connaissance sur le fonctionnement du système. À chaque commande correspond un onglet dans la fenêtre dont la partie inférieure est un écran destiné à afficher tous les événements du système.

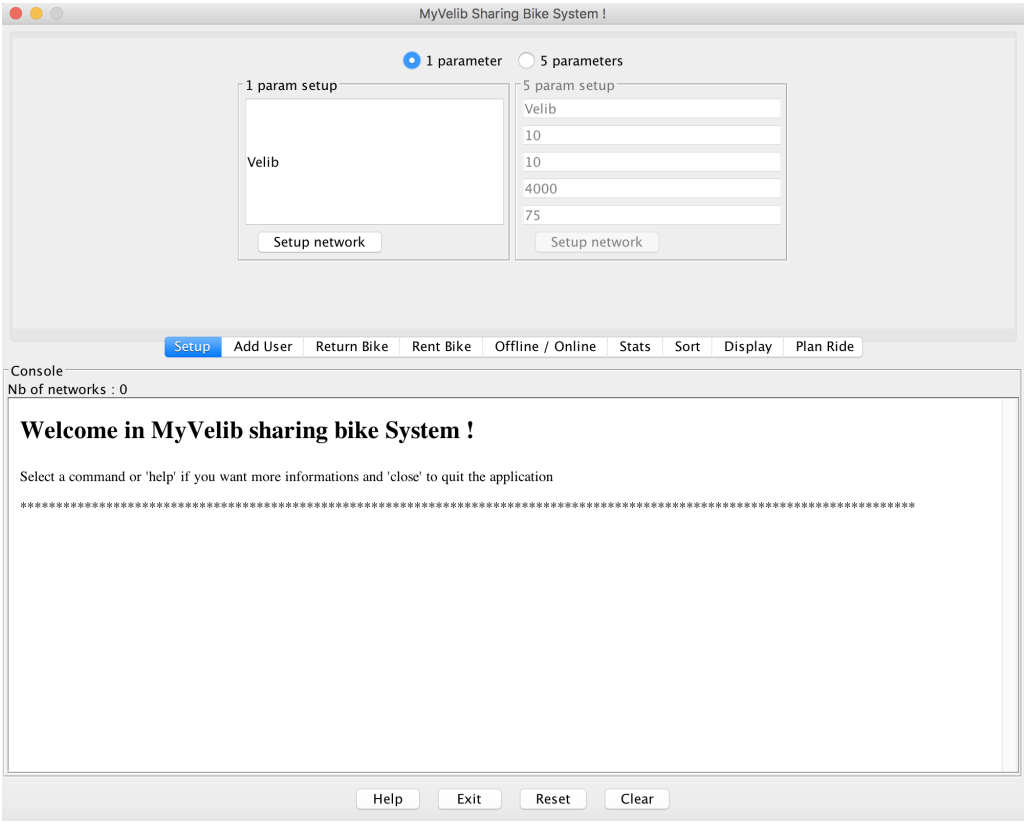


Figure 5: Accueil GUI.

La partie haute est donc une **JTabbedPane** contenant plusieurs objets. Ces objets sont des sous-classes de **JPanel**, créées pour l'occasion dont le rôle est d'afficher les champs relatifs à une commande en particulier et d'implémenter les méthodes permettant son interprétation. Nous avons donc plusieurs classes héritant de **JPanel** (par exemple, **SetupTab**) qui initialisent les composants d'un onglet et sont en lien permanent avec la console, partie inférieure de la fenêtre.

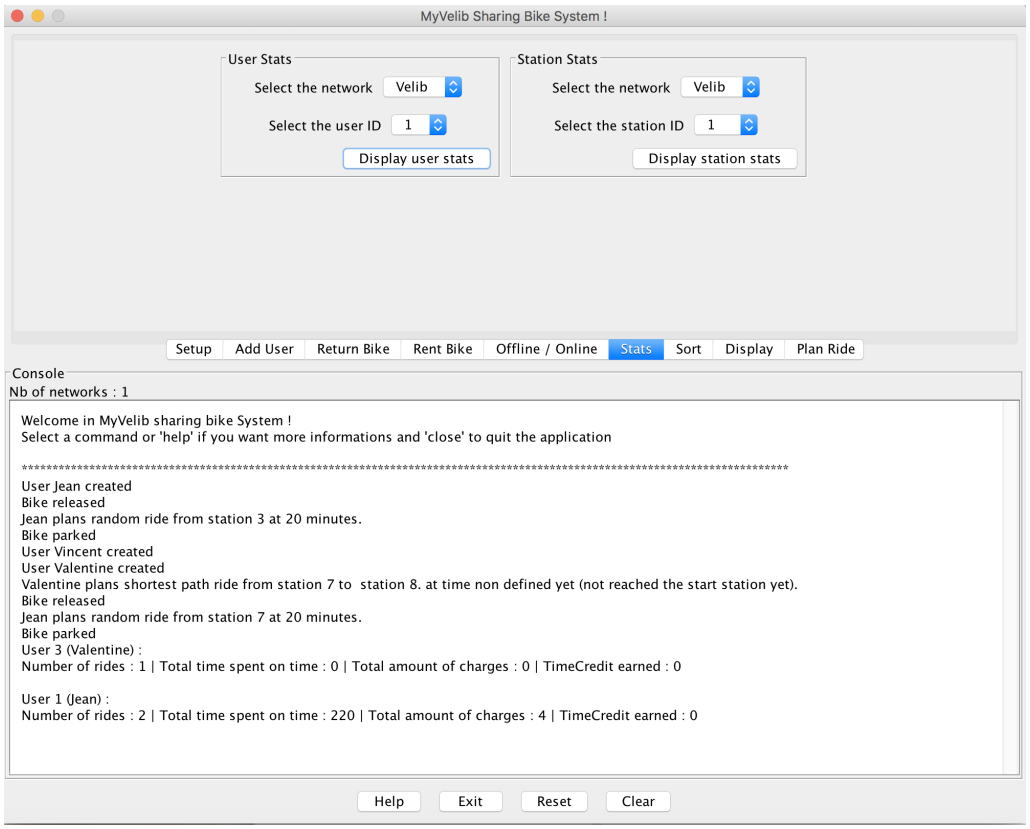


Figure 6: Scénario exemple sur GUI.

Le fonctionnement de cette interface repose principalement sur la CLUI précédemment codée. Chaque bouton de validation présent dans un onglet admet un **ActionListener** qui, appelé, se charge de collecter les données rentrées, les transformer en une ligne de commande String prête à être envoyée à la classe Interface pour traitements comme précédemment. La suite ne change donc pas de ce qui précède à l'exception faite que nous avons créé dans le *package myVelibUtilities* une classe **PrintConsole** dont le rôle est de collecter un message dès qu'une action est réalisée. Les principales méthodes de chaque classe sont en effet équipées d'une ligne **PrintConsole**.*updateScreenMessage(String message)* qui met automatiquement à jour l'attribut de la classe tout en appelant la classe **ClientApplication** (la fenêtre héritant de **JFrame**) à mettre à jour son écran en affichant le nouvel événement. Ainsi, une validation via l'appui sur un bouton dans l'interface déclenche la mise à jour automatique du contenu de l'écran de l'application.
La partie plus ardue de l'implémentation de cette interface réside dans les nécessaires mises à jour des onglets suite aux actions des autres. En effet, si vous créez un nouveau réseau dans l'onglet **Setup**, le sys-

tème doit automatiquement mettre à jour tous les onglets de sorte que ce nouveau réseau soit sélectionnable. De la même manière, lors de la planification d'un trajet, c'est le choix du réseau qui va déterminer les possibilités de choix dans les **JComboBox** de *userID*. Cela implique donc que les **JComboBox**, à l'intérieur même des onglets s'écoutent. Pour ce faire, des implémentations de **ItemListener** ont été créées, qui affichent les listes des stations et utilisateurs du réseau sélectionné dans l'onglet. Par ailleurs, chaque appel à la méthode *update()* de **ClientApplication** déclenche automatiquement l'appel des fonctions de mise à jour de chaque onglet, de sorte que pour chaque action, quelle qu'elle soit, entraîne un rafraîchissement global de l'interface. Remarquons que l'implémentation de la classe **HashMap** et l'utilisation dans les boucles de mises à jour de **Entry** ne garantissent pas le bon ordre des différents Ids affichés dans les **JComboBox**.

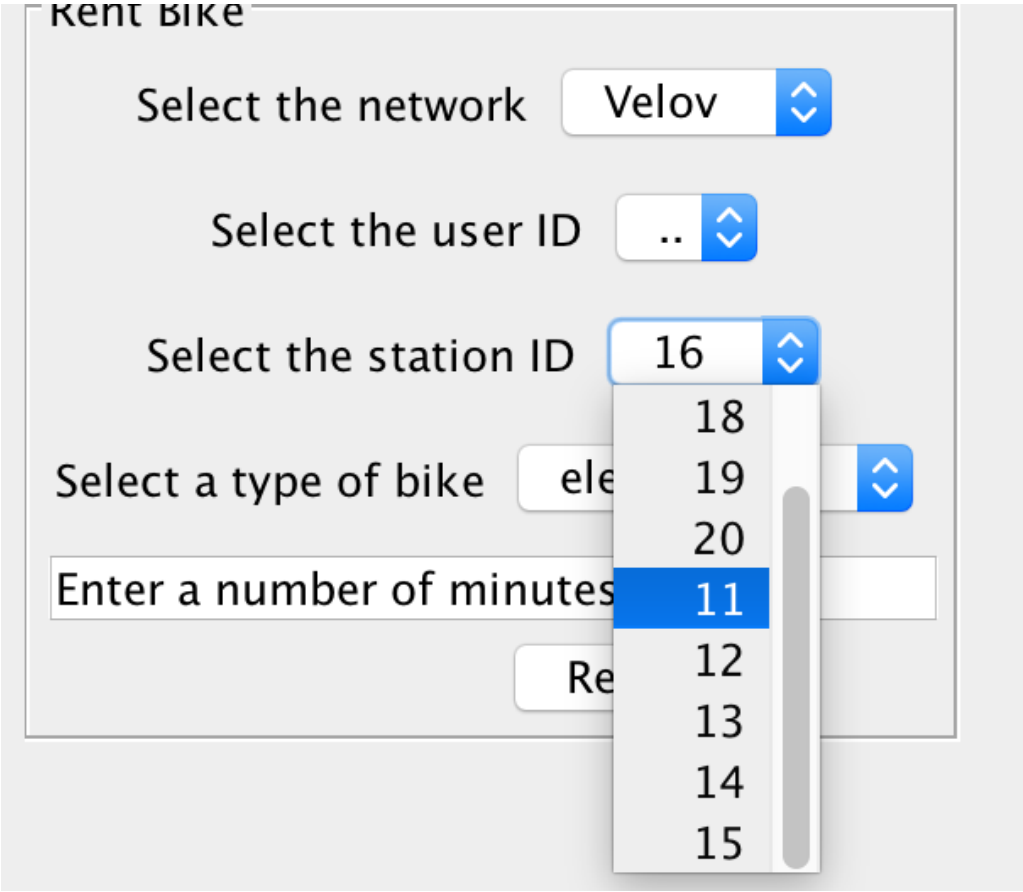


Figure 7: Ordre non garanti.

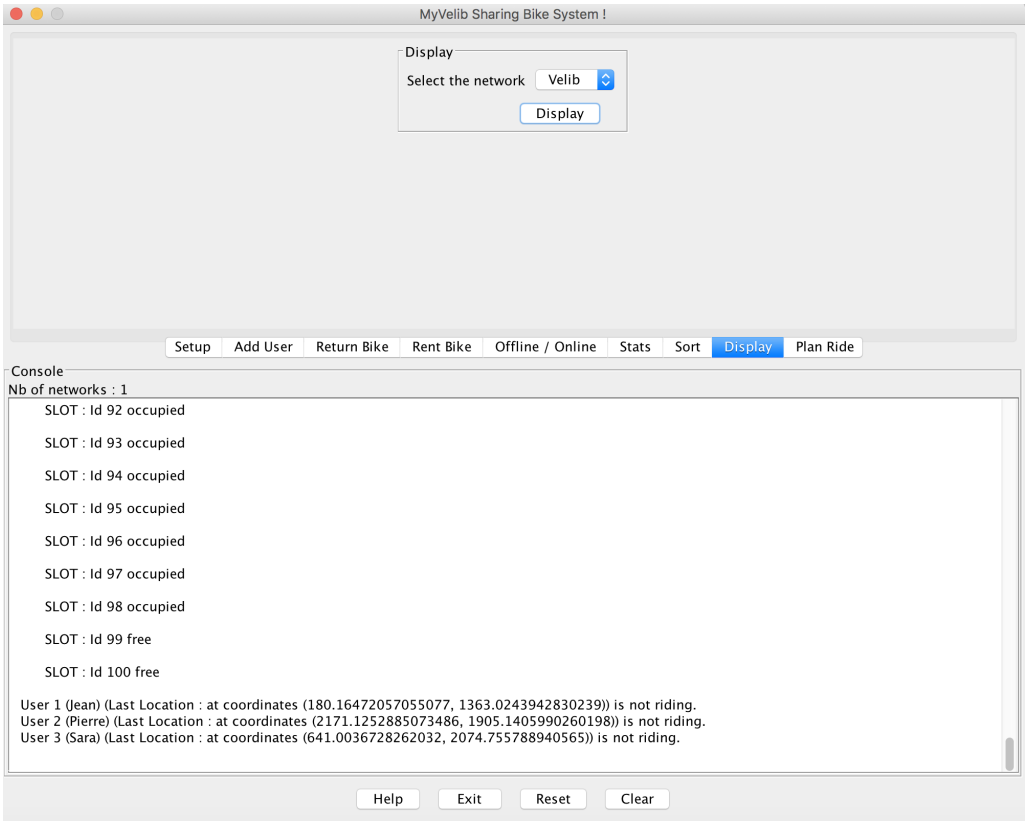


Figure 8: Ecran avec scrollBar.

# 3 Tests et Vérifications

Les tests du programme sont primordiaux. En effet, ils sont la seule garantie de l'efficacité du code. Nos tests étaient séparés en deux parties : les « Java Unit Tests » et les « cas d'utilisation ».

## 3.1 JUnit Test Cases

Les Java Unit Tests permettent de tester des bouts de code séparément. Ainsi, nous pouvions facilement observer si chacune des méthodes se comportait comme on l'attendait dans différentes situations. Ils assurent ainsi le fonctionnement de chaque partie du programme aussi indépendamment que possible des autres, et constituent ainsi une première étape importante (plus simple à mettre en place et débugger) avant les cas d'utilisation. Pour rendre cette opération plus efficace, et en accord avec le plan initial contenu dans le diagramme UML, nous avons codé ces tests avant d'implémenter nos classes, selon la méthode du *Test Driven Development*. De plus, chacun a écrit les tests des classes créées par l'autre membre du binôme, afin d'éviter le contournement de certains tests.
Ces tests ont permis la résolution de nombreux bugs au cours du développement et d'améliorer en permanence la conception et le codage des différentes méthodes. Des problèmes de construction du système sont alors apparus pour la première fois, ce qui témoigne de l'absolue nécessité de ces vérifications.

| JUnit Test | Description |
|---|---|
| AvoidPlusRideTest | Vérifie la conformité du Ride calculé |
| PreferPlusRideTest | |
| PreservationOfUniformityTest | |
| ShortestPathRideTest | |
| FastestPathRideTest | |
| NoCardTest | Vérifie le bon fonctionnement des cartes |
| VlibreTest | |
| VmaxTest | |
| PositionTest | Vérifie l'exactitude des méthodes de calcul de distance |
| StationTest | Vérifie le bon fonctionnement des principales méthodes |
| UserTest | |
| TimeTest | Atteste l'exactitude des calculs de durée d'un trajet |

Table 2: Liste des tests.

Aucun **Junit** test n'a été jugé utile pour la **CLUI** et la **GUI** dans la mesure où ces deux blocs reposent entièrement sur le cœur du système dont le bon fonctionnement a été attesté. Nous avons néanmoins procédé à de très nombreux tests via la console en réitérant divers scénarios et en testant toutes les commandes disponibles en variant les paramètres et les situations (station pleine ou indisponible, type de vélo souhaité non présent à la station...).

### 3.2 Scénarios test

Quoi de mieux que des cas pratiques pour mettre à l'épreuve la robustesse du système ? Nous avons créé des scénarios de fonctionnement du système mettant en jeu différents utilisateurs, réalisant des actions à différents moments sur différentes stations, physiquement ou à distance. Ces scénarios sont écrit ligne par ligne dans un fichier texte respectant les codes exigées par le cahier des charges. Ainsi, chaque commande est écrite sous la forme :
*nom-commande <paramètre1> <paramètre2> etc.*
Pour lancer l'exécution de ces cas d'utilisation, il vous suffit d'écrire dans la console une commande respectant la norme:
*runtest testScenarioX.txt*

### 3.3 Scénario 1

Le premier scénario regroupe l'ensemble des cas d'utilisation suggérés par l'énoncé. On commence par créer un réseau nommé **VelibParis** dont les paramètres sont renseignés. Puis l'on ajoute 6 utilisateurs au réseau, possédant des cartes différentes. Deux d'entre eux retirent un vélo d'un type voulu au bout d'une certaine durée (respectivement 20 et 40 minutes) puis le premier retourne son vélo 75 minutes plus tard à une station quelconque (ici 8) du réseau. L'usager 2 planifie alors une course en choisissant la plus courte distance et en renseignant sa destination et le type de vélo souhaité. La station 1 tombe en panne. L'initialisation étant aléatoire, cela peut provoquer une notification chez cet usager si la station est sa destination. Puis celui-ci retourne son vélo. On affiche enfin l'ensemble du réseau (statistiques de chaque station et de chaque individu).L'ensemble des événements a été enregistré dans le fichier texte **testScenario1Output.txt**.

### 3.4 Scénario 2

Le deuxième scénario est un peu plus complexe : deux réseaux sont initialisés, l'un par défaut, l'autre en choisissant ses paramètres. Puis le premier reçoit 4 usagers, le second en reçoit 1. De manière similaire au premier scénario, une série de location/retour est effectuée mais une nouveauté a été apportée : la commande **returnPlanRide** prenant en arguments *userId*, *time* et *networkName* a été ajoutée pour pouvoir retourner le vélo d'un usager dont le trajet a été planifié et donc dont la station d'arrivée a été calculée à l'avance. Nous avons donc deux commandes pour rendre un véhicule : une par défaut et l'autre pour "automatiser" le retour du vélo dans le cadre d'une planification. Enfin, nous affichons les statistiques relatives aux deux réseaux.L'ensemble des événements a été enregistré dans le fichier texte **testScenario2Output.txt**.

## 4 Conclusion et suggestions d'amélioration

Le cours nous a permis d'apprendre, comprendre et de mettre en pratique à travers des cas simples les techniques de base de la programmation objet; ce projet a été l'occasion de les appliquer sur un problème beaucoup plus complexe et stimulant. De la réflexion initiale sur les différents patterns à employer aux implémentations finales de la CLUI et de la GUI, nous avons découvert les innombrables possibilités que nous offre la POO mais également ses contraintes en se confrontant pour la première fois à la conception d'un système complexe. La répartition du travail a nécessité une vision d'ensemble claire et partagée des tâches à réaliser et a mis à l'épreuve notre capacité à organiser le code, le rationaliser puis le simplifier. Nous avons développé peu à peu des réflexes de programmeur qui nous ont permis de résoudre de plus en plus vite les dysfonctionnements, de les anticiper et d'en comprendre les causes.

Cependant, notre système a ses limites et n'a pas la prétention d'être optimal. Nous avons dû faire des compromis entre complexité et caractère "open-close" du code. La conception d'origine a été repensée au fil des implémentations et des tests mais un programme d'une telle ampleur implique inéluctablement une certaine rigidité : nous avons divisé et organisé le code de la manière la plus optimale possible pour décentraliser au mieux les tâches de telle sorte à rendre l'ajout de nouvelles fonctions, cartes, types de vélo et station beaucoup moins coûteux.

Une première amélioration consisterait à déplacer les méthodes *planXRide()* de la classe User vers une classe à part qui serait appelée par celle-ci afin de planifier un ride en passant le type de trajet souhaité en paramètre. Le code serait ainsi plus propre mais nécessiterait tout de même une mise à jour de la classe lors de l'ajout une nouvelle stratégie. Le principe "open-close" est un idéal à atteindre mais trouve ses limites dans la complexité des projets, pour lesquels il convient de déterminer les compromis les moins coûteux en terme de modularité.

Une seconde proposition serait de créer une classe **Map** dans le package *myVelibGUI*, héritant de **JPanel** et permettant l'affichage d'une carte dont les points représentent la position des usagers, des vélos et des stations. Un code couleur permet de distinguer les stations d'un même réseau et un second les stations hors ligne et en ligne. Un premier essai modeste d'une telle classe est visible mais sa programmation simple et efficace requiert l'usage de librairies de représentation 2D dont

nous n'avons pas eu le temps d'étudier le fonctionnement.

# 5 UML class diagram

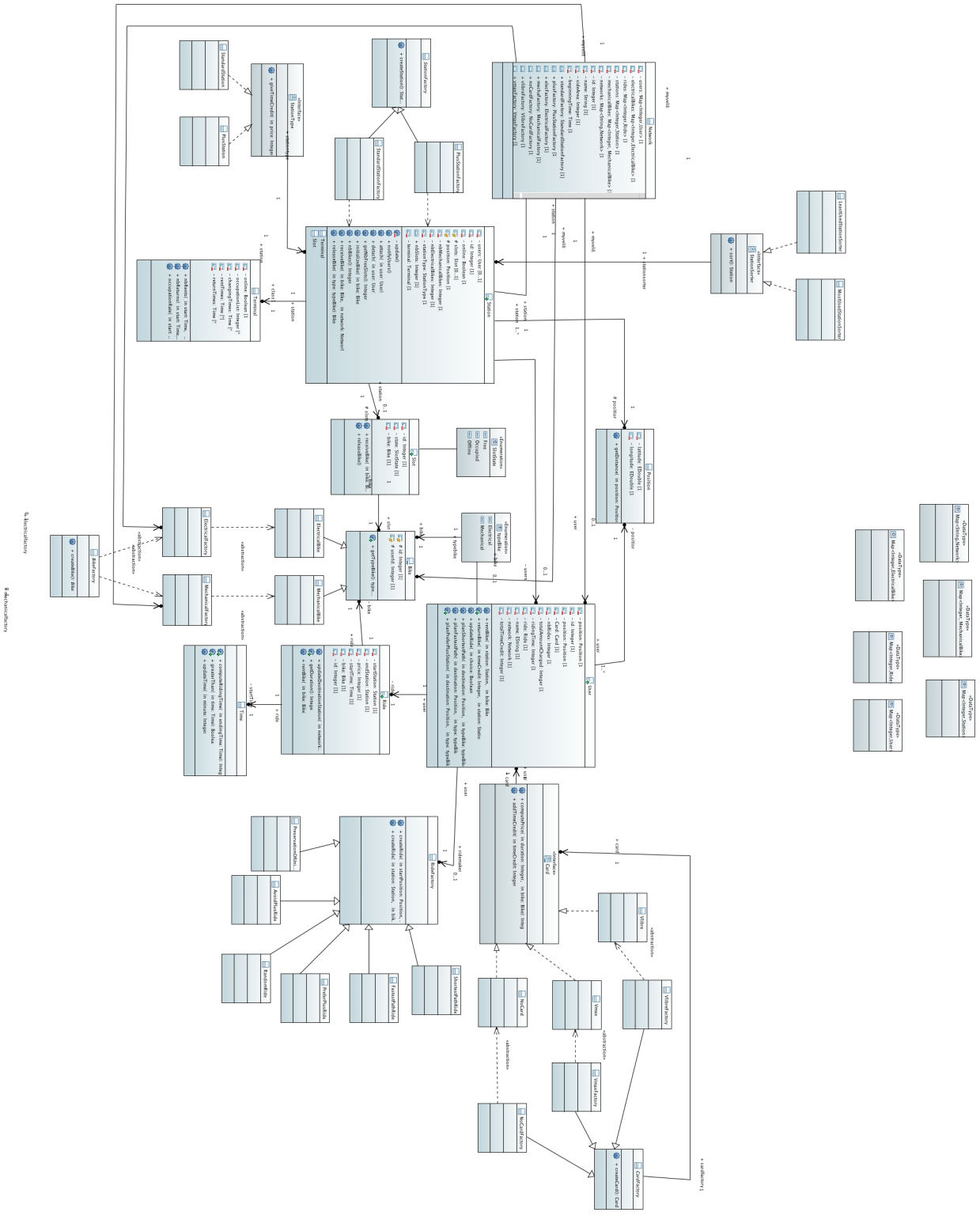

Figure 9: UML class diagram.