Projet myVelib

Timothé Chaumont
CentraleSupélec
Gif-sur-Yvette, France
timothee.chaumont@student-cs.fr

Jean Bourgeois
CentraleSupélec
Gif-sur-Yvette, France
jean.bourgeois@student-cs.fr

ABSTRACT

A bike sharing system (like, for example, Velib in Paris) allows inhabitants to rent bicycles and cycle around a metropolitan area. Such a system consists of several interacting parts including: the renting stations (displaced in key points of a metropolitan area), different kind of bicycles (mechanical and electrically assisted), the users (which may posses a registration card), the maintenance crew (responsible for collecting/replacing broken down bicycles), etc. Based on the systems requirements we were required to develop a Java framework, hereafter called myVelib for managing bike sharing.

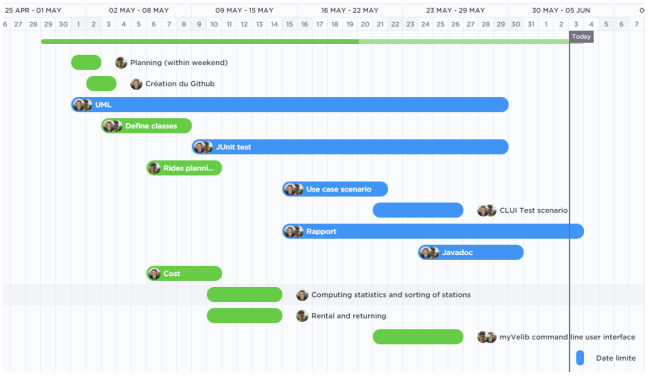

Figure 1: Diagramme de Gantt du projet avec l'attribution des tâches

1 INTRODUCTION

Ce projet final de l'électif *Object oriented software engineering* nous a permis de se confronter au développement d'un programme complexe. Cela s'est fait par la création de tests, la rédaction de documentation et l'utilisation de design patterns.

Les objectifs de ce projets étaient de créer une application de gestion d'un parc de vélo. Pour cela, il nous a fallu représenter les stations et leur vélos qui y sont garés, les utilisateurs, et créer un certain nombre de fonctions utilitaires correspondant aux actions possibles aux sein des stations. Ce projet pourrait servir de backend à une plateforme de gestion locative de vélo, et pourrait interagir avec un site web ou une application mobile.

Nous avons commencé par développer les classes principales *core classes* puis ajouté progressivement les fonctionnalités décrtites dans le *system requirements*.

2 PLANNIFICATION DES TÂCHES ET ORGANISATION

Pour la réalisation de ce projet, nous avons décidé de partager l'intégralité des actions à mener en deux parties qui nous semblaient équitables et cohérentes. Nous avons réalisé ce découpage des tâches dès le premier jours où le sujet du projet était disponible et nous en avons profité pour établir un calendrier de type digramme de Gantt à l'aide de l'outil collaboratif clickup.com. Le planning initialement établi est visible Figure 1. On peut en plus y retrouver l'attribution des tâches : afin que vous puissiez nous identifier via nos photos de profile, Jean Bourgeois s'est par exemple occupé de réaliser le planning tandis que Timothé Chaumont a crée le Github.

En procédant ainsi nous avons pu avoir une forte autonomie vis-à-vis de l'autre : nous n'avons jamais été bloqué parce que l'autre était en retard. De plus cela nous permet de nous focaliser sur notre partie sans avoir à constamment essayer de comprendre ce qu'avait fait l'autre. Bien sur par moment il nous fallait utiliser le code produit pas l'autre, dans ces moments là nous avons organisé des réunions afin de résoudre nos problèmes plus rapidement. De même lorsque quelqu'un trouvait un bug sur le code produit par l'autre.

Enfin nous avons utilisé le gestionnaire de versions Git via Github tout au long du projet, notamment pour sa simplicité d'utilisation mais aussi car il est largement compatible avec la plupart des IDE et notamment InteliJ que nous avons tous les deux utilisé. Nos travaux sont accessibles ici https://github.com/tim99oth99e/myVelib.

3 COEUR DE L'APPLICATION

Dans cette section nous aborderons les choix de design ainsi que les éventuelles difficultés que nous avons rencontré lors du développement des classes au coeur de l'application. Ces classes permettent de modéliser les vélos **Bicycle**, les stations **Station**, les utilisateurs **Users** ainsi qu'une dernière classe **Record** qui permet de stocker et notifier les évènements générés sur le réseau myVelib, ceci afin de calculer les diverses statistiques.

3.1 Bicycle

La classe `Bicycle` permet de modéliser un vélo qui peut soit être électrique soit être mécanique. Pour gérer le type de vélo nous avons décidé d'utiliser des Enums. Nous nous assurons de l'unicité de l'identifiant d'un vélo avec l'intermédiaire d'un variable statique privée

```
private static Integer uniqueId = 0;
```

que l'on va incrémentée à chaque appel au constructeur.

Nota : nous utiliserons la même technique dans les autres classes à chaque fois qu'il faudra garantir l'unicité d'un identifiant.

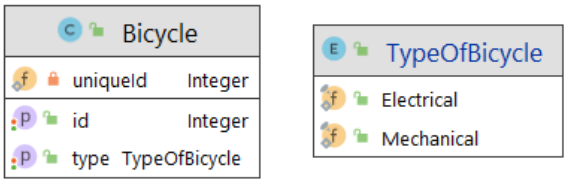

Figure 2: Diagramme UML relatif à la gestion des vélos

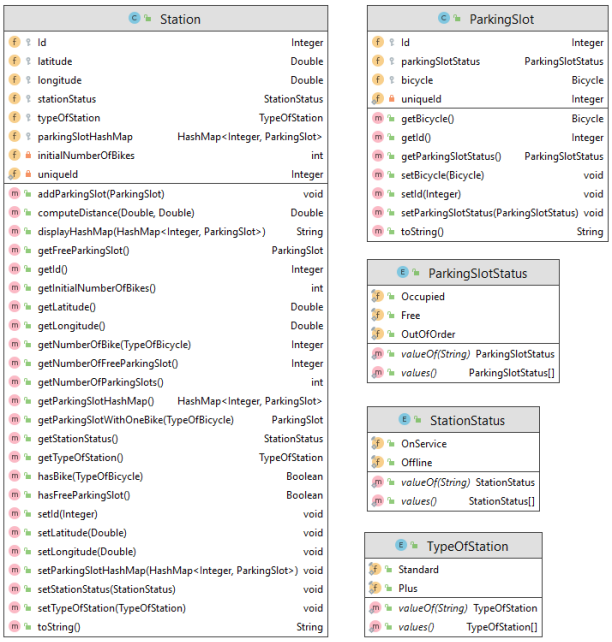

Figure 3: Diagramme UML relatif à la gestion des stations

3.2 Station

La classe `Station` permet de modéliser une station avec les attributs suivants :

(1) Son identifiant unique.
(2) Sa latitude et longitude.
(3) Son status qui peut être On Service ou Offline.
(4) Son type qui peut être Standard ou Plus.
(5) Les places de parking dont elle dispose.

Pour gérer le status ainsi que le type de la station nous avons décidé d'utiliser des Enums. De plus pour gérer les places de parking au sein de la station nous utilisons une HashMap qui contient les différents `ParkingSlot`.

Nous avons donc décider de séparer complètement la définition des places de parking de celle d'une station. Pour ce faire nous avons défini une classe `ParkingSlot` qui modélise une place de parking avec les attributs suivants :

(1) Son identifiant unique.
(2) Son status qui peut être Occupied, free ou bien Out of order.
(3) Le vélo qui est éventuellement garé sur cette place de parking.
(4) Les places de parking dont elle dispose.

Pour gérer le status de la place de parking nous avons encore décidé d'utiliser des Enums. Vous pouvez retrouver le diagramme UML relatif à la gestion des stations Figure 1.

La classe`Station` dispose d'une multitude de méthode dont la documentation est disponible sous forme de Javadoc.

3.3 User

La classe `User` contient tous les attributs d'un utilisateur du système, ainsi que des statistiques concernant les locations de vélo qu'il a effectué.

Pour assurer l'unicité des identifiants attribués au instances, nous stockons les identifiants actuellement utilisés dans une ArrayList statique. Lorsqu'un nouvel utilisateur est créé, le système lui attribue comme identifiant le plus petit entier positif non utilisé. Cette liste est mise à jour lors de la suppression d'un utilisateur.

Nous avons créé des Exceptions assurant la validité des paramètres utilisés lors de l'instanciation d'un utilisateur. Les coordonnées de latitude, par exemple, doivent être comprises entre -90° et +90°, et le numéro d'une carte de crédit doit être composée de 16 chiffres.

Enfin, avons ajouté deux champs `rentedBicycle` et `rentdatetime` afin d'empêcher un utilisateur de pouvoir louer deux vélos simultanément. Ces attributs sont actualisés par les méthodes `park` et `rent`, qui mettent également à jour les statistiques de l'utilisateur.

3.4 Registration cards

Pour représenter le type de carte d'un utilisateur, nous avons tout d'abord choisi d'utiliser des Enums. Mais, étant donné que chaque carte avait sa manière de calculer le coût d'une location, nous avons décidé de faire une classe pour chaque type de carte.

Pour ne pas avoir à répéter de code et pour permettre d'ajouter, dans le futur, d'autres types de cartes, nous avons utilisé le *strategy*

pattern. Pour cela, nous avons créé une classe abstraite `RegistrationCard` (car certains attributs et méthodes étaient communs à toutes les cartes), puis créé trois sous-classes `NoRegistrationCard`,`VlibreRegistrationCard` et `VmaxRegistrationCard`.

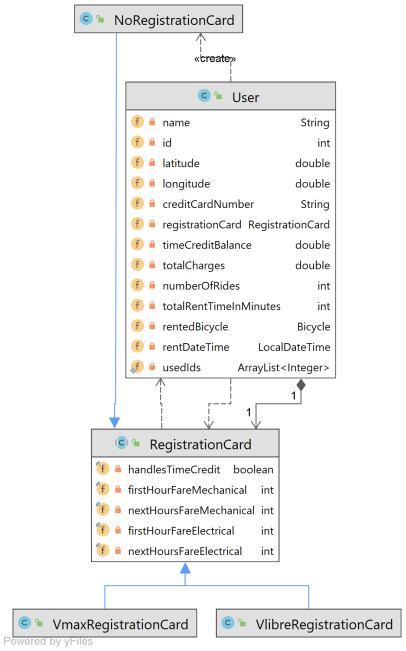

Figure 4: Diagramme UML des classes User et RegistrationCard

4 RIDES COST COMPUTATION

Pour le calcul de coût d'une location, nous avons envisagé deux options différentes :

(1) facturer l'utilisateur pour chaque heure qu'il a commencé. Seulement, le timeCredit, gagné en rapportant un vélo à une station *plus* ne semblait pas viable :
   - si l'on retirait les minutes en plus d'une heure, en général, l'utilisateur n'économiserait pas d'argent
   - si l'on retire une heure à chaque fois qu'une heure de timeCredit est gagnée:
     → Il faudrait attendre longtemps avant de pouvoir obtenir une réduction : au maximum 5 minutes sont gagnées par location.
     → Pour que le prix d'un trajet soit réduit, il faut qu'il dure au moins 2h.

(2) facturer une location au nombre de minutes passées. Cela aurait réglé le problème du time credit. Cependant, dans ce cas, les consignes n'étaient pas respectées le prix d'une location d'une heure n'aurait pas été exactement d'un euro.

Nous avons donc décidé de donner en crédit les minutes non utilisées de la dernière heure facturée.

Par exemple, si une location dure 1h25, alors l'utilisateur sera facturé 2h et 35 minutes seront ajoutées à son *timeCredit balance*.

Le calcul du coût d'une course est réalisé par la classe `RegistrationCard`. Le calcul est toujours réalisé de la même manière.

(1) calcul du crédit de temps gagné pendant la course,
(2) calcul des heures facturées : nombre d'heures commencées - réduction du time credit,
(3) mise à jour du *time-credit* de l'utilisateur,
(4) calcul du coût des heures facturées.

La méthode `computeRideCost` calculant le coût d'une course, est la même pour toutes les cartes. Seuls les tarifs des différentes heures varie selon le type de carte et la gamme de vélo. Les tarifs sont résumés dans la Table 1.

Table 1: Ride fare table in euros

Card	time credit	Electrical 1st h	Electrical 2+ h	Mechanical 1st h	Mechanical 2+ h
None	No	2	2	1	1
Vlibre	Yes	1	2	0	1
Vmax	Yes	0	1	0	1

5 RIDES PLANNING

Nous devions équiper le framework myVelib d'une fonctionnalité d'aide à la planification de trajet pour l'utilisateur. À partir de coordonnées GPS de l'utilisateur ainsi que des coordonnées GPS du point de destination l'outil de planification doit être capable d'identifier le trajet optimal et d'indiquer dans quelles stations le vélo doit être loué et rendu.

Afin d'être compatible avec le principe ouvert/fermé (open-close principle), nous avons décider de dédier une classe mère à la planification de trajet (`RidePlanningNormal`) en la dotant de deux méthodes :

(1) findStartStation() qui permet d'identifier le station de départ optimale.
(2) findDestinationStation() qui permet d'identifier le station d'arriver optimale.

L'idée était d'étendre ensuite la classe `RidePlanningNormal` avec trois autres classes *RidePlanningAvoidPlusStation*, *RidePlanningPreerveUniformityOfBicycle* et *RidePlanningPreferPlusStation* qui décrivent d'autre définition de station 'optimale' en faisant bon usage du polymorphisme.

Ainsi on va venir *override* les méthodes findStartStation() et findDestinationStation() lorsqu'on en a besoin. Cette approche à deux avantages majeurs : elle est rapide, il suffit d'un appel à la méthode findStartStation() quelque soit le type de `RidePlanning` que l'on souhaite utiliser pour trouver notre station de départ ; et elle est flexible, il est aisé de rajouter un nouveau type de `RidePlanning` si on le souhaite car il ne faudra en aucun cas modifier tout ce qui a déjà été fait auparavant.

Vous pouvez retrouver le diagramme UML relatif à la planification de trajet Figure 5.

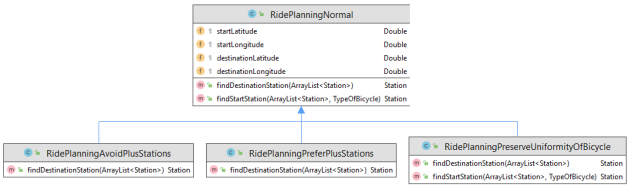

Figure 5: Diagramme UML relatif à la planification de trajet

Nota : La documentation de toutes ces méthodes, notamment les définitions d'une station 'optimale', est disponible sous forme de Javadoc.

6 COMPUTATION OF STATISTICS STATIONS SORTING

Nous avons créé une classe `Record` permettant le stockage des données du système : les stations et leurs places, les utilisateurs, et les évènements (location et retour de vélos, changement de statut d'une place).

Nous avons stocké les utilisateurs et les stations dans deux `hashmap` liant chaque identifiant à son instance, et les évènements dans une liste triée par date. Cependant, pour rendre plus efficace le traitement des données (utilisateurs, stations, locations) nous aurions pu les stocker dans une base de données externe, par exemple sous un format SQL.

Pour éviter de stocker des doublons, nous avons ajouté des vérifications avant le stockage de chaque type d'objet (User, Station et Event).

6.1 User statistics

Afin de simplifier cette partie du programme, nous avons décidé de stocker et mettre à jour les statistiques concernant les utilisateurs directement dans les instances de |User|. L'affichage des statistiques d'un utilisateur correspond donc simplement à la lecture de 4 de ses attributs.

6.2 Station balance

Le calcul de l'évolution du nombre de places libres consiste à parcourir tous les évènements et à compter le nombre de vélos loués et le nombre de vélos garés dans cette station.

6.3 Station average occupation rate

Cette méthode a été la plus longue à implémenter, et nous avons dû changer plusieurs fois de manière de stocker les données pour la faire fonctionner correctement.

La version finale de cette méthode parcourt tous les évènements stockés, vérifie s'ils sont arrivés dans la station étudiée, et entre les bornes de la fenêtre de temps donnée en paramètres.

Si c'est le cas, le programme calcule l'occupation de la station entre cet évènement et le dernier rencontré, et met à jour l'occupation totale.

Finalement, l'occupation moyenne est renvoyée.

6.4 Stations sorting

Le tri des stations a été réalisé en utilisant un strategy pattern (permettant plus tard l'ajout d'autres manières de trier les stations).

Le tri fonctionnait comme suit :

(1) calcul du score de chaque station vis à vis du critère choisi - cela est implémenté dans les sous-classes - et stockage de ces score dans une `LinkedHashMap` (pour avoir un ordre entre les éléments)
(2) la Map est convertie en liste
(3) la liste est triée dans le sens croissant ou décroissant (cela dépenda du tri)
(4) la liste est convertie en Map

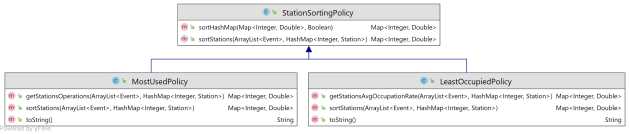

Figure 6: Diagramme UML des différents tris

7 JAVADOC

Pour documenter l'intégralité de ce projet nous avons utilisé Javadoc qui est un outil développé par Oracle, permettant de créer une documentation d'API en format HTML depuis les commentaires présents dans un code source en Java. Notre Javadoc est disponible dans l'archive de notre projet *.\myVelib\doc*, en ouvrant *index.html* dans un navigateur web.

8 UI

8.1 CLUI

Pour réaliser notre interface utilisateur en ligne de commandes, nous avons utilisé une *read–eval–print loop*.

Le fonctionnement est le suivant :

(1) **read** : lecture de la commande entrée par l'utilisateur du système;
(2) **eval** : *parsing* de la commande, de ses paramètres et, si elle est valide, exécution de cette commande;
(3) **print** : affichage du résultat correspondant.

Notre code pour cette interface utilisateur est séparée en deux classes. `MyVelibSystem` décrit la structure logique du fonctionnement de cette CLUI, et `VelibCommand` *parse* et exécute la commande entrée par l'utilisateur.

Cette dernière classe utilise un `switch` dans sa méthode `eval` pour effectuer des actions différentes en fonction des entrées de l'utilisateur.

8.2 Initialisation d'un réseau myVelib

Afin de tester notre projet nous avons implémenter un fichier de configuration *my_velib.ini* qui permet de charger une configuration initiale d'un réseau myVelib.

Nous avons décider d'inclure des informations sur les stations ainsi que sur les utilisateurs qui devront être instanciés à l'exécution

du programme. Ci-dessous le contenu typique de notre fichier de configuration :

```
; Last modified 02 June 2020 by Jean Bourgeois
; Example of INI File for myVelib
; my_velib.ini

[stations]
nstations=10
nslots=5
s=15.5
nbikes=30

[users]
names=Jean Bourgeois,Timothé Chaumont
latitudes= 82.2,34.3
longitudes=67.2,123.3
creditCardNumbers= 1235384958374038,1235384939027403
```

On peut y voir dans l'en-tête [stations] qu'il faudra instancier 10 stations possédant 5 places de parking chacune et placer aléatoirement 30 vélos parmi ces places de parking.

Comme il n'existe pas solution native pour gérer les fichier de configuration .ini en Java nous avons décider d'utiliser la bibliothèque largement utilisée **ini4j**. En conséquence il vous est nécessaire d'installer cette bibliothèque sur votre machine afin de pouvoir utiliser notre CLUI.

Pour l'installation de cette librairie vous pouvez suivre http://ini4j.sourceforge.net/download.html.

L'utilisation d'un fichier de configuration s'en trouve grandement simplifiée puisque il suffit par exemple de ces deux lignes de code pour récupérer le nombre de station à instancier :

```
Wini ini = new Wini(new File("UI/src/classes/my_velib.ini"));
String nstations = ini.get("stations", "nstations", String.class);
```

8.3 Scénario de test

Pour lancer un scénario de test, il faut :

(1) lancer la *CLUI* : run le fichier `myVelibSystem.java`,
(2) puis taper la commande `runtest nom_du_test.txt`. *Par exemple : > > > runTest realScenario.txt*

Toutes les commandes du scénario test seront effectuées et leurs résultats seront affichées dans la fenêtre de sortie de l'éditeur Java.

Nous avons écrit 4 scénarios de test, dont deux vérifiant le fonctionnement de commandes valides et l'échec de mauvaises commandes, ainsi qu'un test d'une situation réelle : `realTestScenario.txt`.

Rappel : la classe `myVelibSystem` nécessite d'avoir avoir ajouté le package de gestion des fichiers d'initialisation `ini4j` (instructions dans la partie ci-dessus) pour pouvoir se lancer.