MyVelib Documentation - A bike sharing plateform

Clément DUFOUR

Michel LEROY

30 May 2023

Group No. 21
Professor: Mr. Ballarini

CentraleSupélec

Contents

Introduction

In modern societies, traffic congestion and air pollution have become major challenges. To mitigate these problems and promote sustainable mobility, bike-sharing systems have gained popularity in many cities worldwide. Among these systems, Vélib' in Paris is one of the most emblematic and successful, offering urban residents a practical and eco-friendly alternative to traditional modes of transportation.

However, for such systems to reach their full potential and meet the growing needs of users, it is imperative to develop a robust and scalable IT platform. This platform plays an essential role in managing and optimizing the day-to-day operations of a bike-sharing system, enabling rental management, bike tracking, data collection, route planning, and much more.

1 Context and Understanding of the Specification

1.1 MyVelib

Before coding a support application for Vélib', we first identify the various features we want to integrate into our model.

- Bike rental:

The user goes to a Vélib' station.
The user identifies themself using a Vélib' card.
The user chooses one of the available bikes. They can only rent one bike at a time.
The bike is unlocked and the user can start their journey.

- Bike return:

The user must park the bike in a free docking spot at a Vélib' station.
If they leave their bike outside a station, they will incur a penalty.
The trip cost is calculated automatically and the user is automatically charged if fees apply.

- Vélib' station:

A Vélib' station is a place where bikes can be rented and returned.
It consists of bike docking spots and a terminal for user interaction.
Stations can be in service or offline, with different types (standard or "plus").
Each station has a unique numeric identifier.

- Bikes:

There are two types of bikes: mechanical and electric.
Each bike has a unique identification number.
Bikes can be rented and returned at a station or at their current GPS position (if parked outside a station).

- User:

Each user has a name, a unique numeric identifier, and a geographic location (GPS coordinates).
The user may have a credit card and a registration card.
Some users have a Vlibre or Vmax card, which offers cost benefits and time-credit.

- Bike usage costs:

The cost of a trip is calculated based on duration, bike type, user card type, and return location.
Different pricing applies based on these factors, with discounts or penalties for returns at a station or outside docking areas.

- Time-credit management:

Some users accumulate time-credit by returning bikes at "plus" stations.
Time-credit can be used to reduce the actual cost of a trip.

- Vélib' cards:

There are two types of registration cards: Vlibre and Vmax.
Cards can be virtual on the user's mobile phone.
By following these steps, users can enjoy the Vélib' system to travel easily by bike in the city while managing their associated costs and time-credits.

- Trip planning:

The user provides their GPS location, the destination location, and trip preferences (for example, if they want to avoid certain station types), and the system then returns the optimal stations according to the requested criteria.

1.2 Project Challenges

In this project, we aim to develop a program capable of executing the instructions provided above.

To this end, we decided to code an object-oriented program in Java using the Eclipse compiler. This project will put into practice the main aspects covered in the software engineering course. Thus, we will pay particular attention to the use of classes, design patterns, and rigorous programming with the addition of tests.

This report therefore outlines our work and explains how to interpret the associated code.

2 Methodology

2.1 Architecture

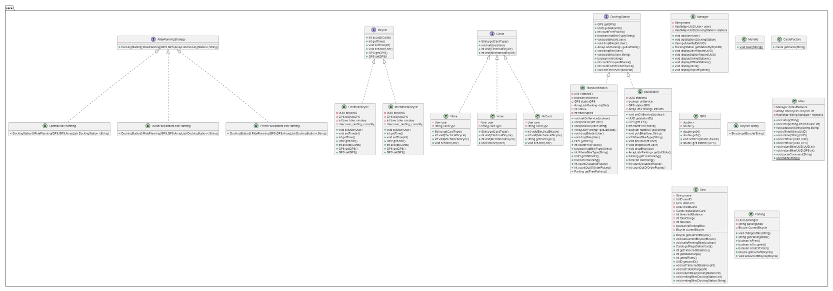

Figure 1: Class diagram of our MyVelib project created on IntelliJ

Here is the overall image of the project: there are several major classes which are the main objects of the system and will interact with each other. There are also various classes that build objects required throughout the code (e.g.: GPS).

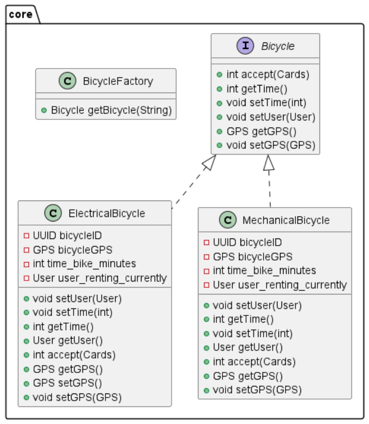

Figure 2: UML diagram of bicycle interface and subclasses

We therefore decided to implement an interface and a factory to respect the OPEN-CLOSE principle. The bike is associated with a user when rented, and with its GPS position. It is also able to accept a visitor pattern described just below.

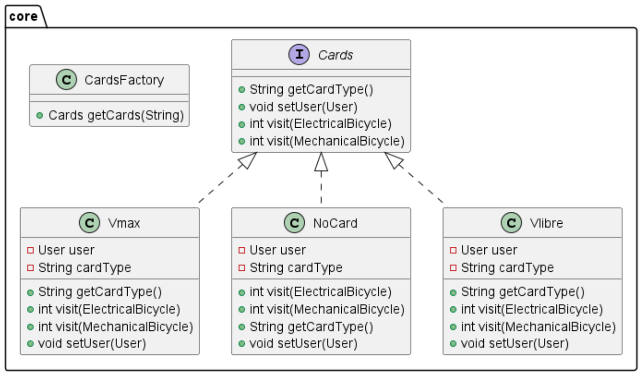

Figure 3: UML diagram of Cards interface and subclasses

Subscription cards allow regular users to get discounts on their usage. The discounts take the form of cost functions that depend on the card and the rented bike. It is therefore necessary to find a method to preserve the OPEN-CLOSE principle, which is especially important here since these subscriptions are likely to change. The visitor pattern addresses this issue. The different cards associated with a user thus “visit” the bikes and calculate the cost by taking all parameters into account.

We now need to build the station objects that will host bikes and interact with users. There are two types: Standard and Plus. The main methods are `pickBike` and `dropBike`, which allow the user to interact with the station by renting or dropping a bike. Other helper methods (e.g. `hasBikeType`) simplify operations performed in other classes (planning strategies, etc.).

Three different strategies will be implemented later (Figure 5): an optimal one, one avoiding plus stations, and one favoring plus stations. We will then use a strategy pattern, which seems the most suitable for this problem. All strategies implement comparators to determine the optimal stations with minimal time.

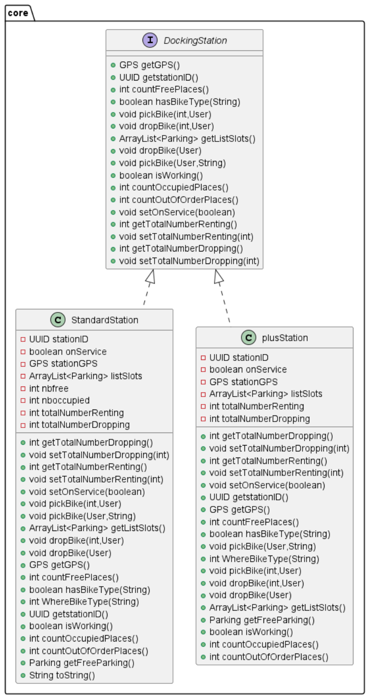
core
DockingStation
GPS getGPS()
UUID getstationID()
int countFreePlaces()
boolean hasBikeType(String)
void pickBike(int,User)
void dropBike(int,User)
ArrayList<Parking> getListSlots()
void dropBike(User)
void pickBike(User,String)
boolean isWorking()
int countOccupiedPlaces()
int countOutOfOrderPlaces()
void setOnService(boolean)
int getTotalNumberRenting()
void setTotalNumberRenting(int)
int getTotalNumberDropping()
void setTotalNumberDropping(int)
StandardStation
UUID stationID
boolean onService
GPS stationGPS
ArrayList<Parking> listSlots
int nbfree
int nboccupied
int totalNumberRenting
int totalNumberDropping
int getTotalNumberDropping()
void setTotalNumberDropping(int)
int getTotalNumberRenting()
void setTotalNumberRenting(int)
void setOnService(boolean)
void pickBike(int,User)
void pickBike(User,String)
ArrayList<Parking> getListSlots()
void dropBike(int,User)
void dropBike(User)
GPS getGPS()
int countFreePlaces()
boolean hasBikeType(String)
int WhereBikeType(String)
UUID getstationID()
boolean isWorking()
int countOccupiedPlaces()
int countOutOfOrderPlaces()
Parking getFreeParking()
String toString()
plusStation
UUID stationID
boolean onService
GPS stationGPS
ArrayList<Parking> listSlots
int totalNumberRenting
int totalNumberDropping
int getTotalNumberDropping()
void setTotalNumberDropping(int)
int getTotalNumberRenting()
void setTotalNumberRenting(int)
void setOnService(boolean)
UUID getstationID()
GPS getGPS()
int countFreePlaces()
boolean hasBikeType(String)
void pickBike(User,String)
int WhereBikeType(String)
void pickBike(int,User)
void dropBike(int,User)
void dropBike(User)
ArrayList<Parking> getListSlots()
Parking getFreeParking()
boolean isWorking()
int countOccupiedPlaces()
int countOutOfOrderPlaces()

Figure 4: UML diagram of station interface and subclasses

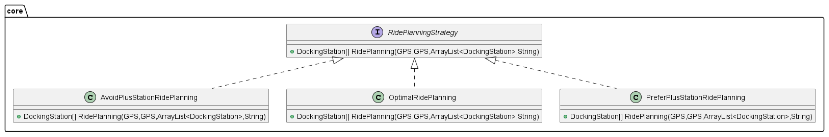
core
RidePlanningStrategy
DockingStation[] RidePlanning(GPS,GPS,ArrayList<DockingStation>,String)
AvoidPlusStationRidePlanning
DockingStation[] RidePlanning(GPS,GPS,ArrayList<DockingStation>,String)
OptimalRidePlanning
DockingStation[] RidePlanning(GPS,GPS,ArrayList<DockingStation>,String)
PreferPlusStationRidePlanning
DockingStation[] RidePlanning(GPS,GPS,ArrayList<DockingStation>,String)

Figure 5: UML diagram of RidePlanning interface and subclasses

GPS is a class that allows converting a pair of doubles into GPS coordinates (lat, long). It provides various useful methods such as `getDistance`, which differs significantly from Euclidean distance, and simpler ones like `toString` and `fromString`.

The `User` class describes the behavior of a user and interacts with stations using `returnBike` and `rentingBike`. And `Parking`, which is invisible to the user but interacts with stations (in summary, a `DockingStation` is just a list of parkings).

The `Manager` class interacts with `User` and `Station` to create two `HashMap`s: `stations` and `users`. These `HashMap`s associate objects with their UUIDs and serve as the system's database. It can then sort its data or call objects in its `HashMap`s via their UUIDs to report on their states.

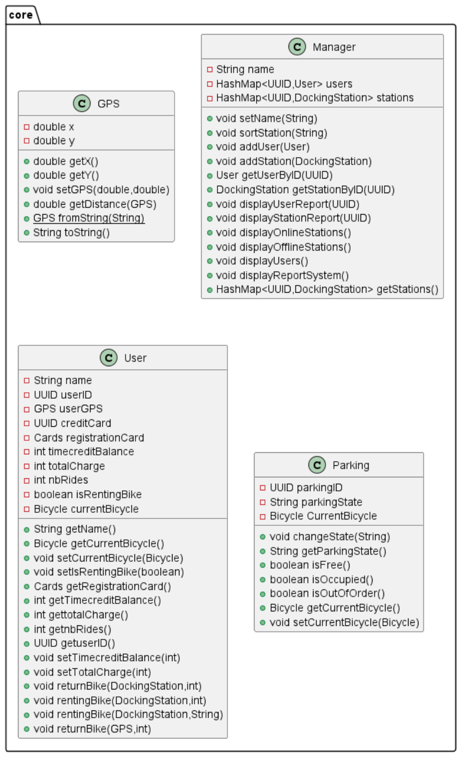

Figure 6: UML diagram of utility classes

Comparators will be used in the CLUI for the `sortStation` function. They extend Java's native `Comparator` class.

2.2 Design Patterns

• Factory Pattern:

When designing the card and bicycle objects, we encountered the following difficulty:

There were several variants of the same object, each having different applications (notably in pricing). This risked overloading the constructor or making the addition of new object variants more complex.

To remedy this, we chose to use the simple factory pattern. Indeed, it allows calling only the factory, and the factory will then create the object specified by the parameters. We preferred to use a simple factory pattern rather than an abstract factory pattern because, given the nature of the MyVelib project, the client would not need to interact with the code.

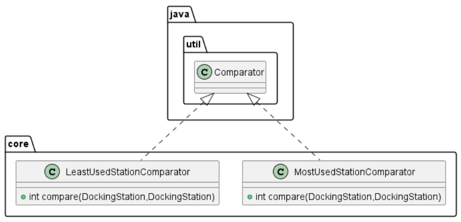
java
util
Comparator
core
LeastUsedStationComparator
int compare(DockingStation,DockingStation)
MostUsedStationComparator
int compare(DockingStation,DockingStation)

Figure 7: UML diagram of comparators

- Strategy Pattern:

When we needed to guide the user along the best path, we encountered the following problem:

A trip could be made in several ways, depending on whether we wanted the optimal path or to take plus stations into account. How to design our classes to easily switch between the different ride planning algorithms?

We used the common interface `RidePlanningStrategy`, which is then implemented by the concrete strategies: optimal trip, avoiding plus stations, and preferring plus stations.

Thus, we implemented 3 ride planning strategies: the optimal strategy that goes from point A to B in the fastest way; the prefer-plus-stations strategy that prefers a plus station if it adds less than 10% to the optimal trip; and the avoid-plus-stations strategy that avoids plus stations if they add less than 10% to the optimal trip.

- Visitor Pattern:

When we had to calculate the cost of each trip, we encountered the following problem:

How to associate the bike usage time with the user's remaining time-credit to calculate the price of their ride without overloading the existing classes?

The visitable interface is `Bicycle`, and the visitor is `Cards`.

2.3 Discussion on Architecture

We are satisfied with our implementation of the Factory, Strategy, and Visitor patterns because their use simplifies our code and allows us to respect the open-closed principle.

However, some other parts of our code do not adhere to this principle and could therefore be improved by using patterns. For example, in our current code it would be more coherent for the manager to be a singleton since we need all system information to be contained in the `HashMap`s of the same instance.

Moreover, it would be more pertinent to split our manager according to the Model-View-Controller pattern. In this case, the manager would only act as the controller and serve as an intermediary between the View and the Model, which would receive the commands. Therefore, the `HashMaps` would need to be moved to a "Log" class that would act as the Model and perform operations on the system elements (station-bike observer).

3 Implementation

3.1 CLUI Commands

The CLUI integrates several commands. You must first start the system by running `Main.java`, which is located in the CLUI package.

The `help` command gives you information about the possible commands to run. The `quit` command allows you to exit the CLUI. And the `runtest TestNumber` command allows you to run one of the pre-implemented scenarios.

The `planningRide startGPS endGPS bikeType` command is the only one I will detail here. GPS coordinates must be entered in the form (2.1,0.3) and the `bikeType` must be `Mechanical` or `Electrical`. Other planning strategies have been implemented but are not accessible via the CLUI. Indeed, the default setup does not include plus stations, so preferring plus stations would not change anything compared to the optimal strategy.

3.2 Main Classes and Methods

In this section we will detail the main classes of our code and how they work. For more information, we invite you to refer to the documentation.

The interfaces `Bicycle, Cards, RidePlanningStrategy & DockingStation` are abstract classes used to implement the various patterns explained above.

The classes `BicycleFactory & CardsFactory` are classes that allow easy instantiation of a bike or a card according to the factory pattern.

The classes `ElectricalBike & MechanicalBike` are concrete classes composed of getters and setters. Note that each bike can be associated with a user.

The classes `NoCard, Vlibre and Vmax` are concrete classes distinguished by their visitor to bikes. Thus, these classes are also capable of calculating the price of each trip according to the type of bike and the pricing associated with the card. Note that to function, a card must be associated with a user.

The `GPS` class indicates the x,y position of an object. In addition to getters and setters, this class mainly has the method `getDistance(GPS gps)` which allows calculating, using a mathematical formula, the distance to another GPS position.

The classes `LeastUsedStationComparator and MostUsedComparator` use the `compare` method on the number of bike returns made between stations.

The `Manager` class enables supervision of the system. It notably contains the MyVelib database in the form of `HashMaps` `stations` and `users`. It is capable of sorting its data and reporting the state of a station or user object in the system.

The classes `OptimalRidePlanning`, `PreferPlusStationRidePlanning` & `AvoidPlusStationRidePlanning` implement the `RidePlanningStrategy` according to the strategy pattern. By sorting stations based on their distances and taking into account a user's desire to favor, disfavor, or be indifferent towards plus stations, these classes are able to propose a route between two stations to a user.

The `User` class is the main class of MyVelib and serves to describe a user. Each user is characterized by their name, UUID, GPS coordinates, credit card identifier, their card, what they have paid, the number of minutes they have in credit, the number of trips they have made, their bike, and whether they are currently using it. Its methods enable them to take a bike from a station if possible and to return a bike to a station after using and paying for it.

3.3 Implementation Challenges

Thanks to our prior reflection on the UML diagram, we did not encounter any major difficulty during the implementation of our code. We had a good structure from the start and then benefited from the examples covered in class.

One of the points that gave us a hard time was the implementation of `sortStation`. We only dealt with it at the end of our project and had to rethink the structure a bit. Indeed, to solve this issue we used comparators. Thus, the code was located in the core while we needed it in the CLUI.

4 JUnit Tests

4.1 Approach

One of the objectives of this project was to acquire software engineering habits. While not using a test-driven development approach, we paid particular attention to testing our code during development.

To do this, we used the JUnit framework, which allows verifying the correct functioning of classes and methods. The list of these tests can be found in the `test` folder of our project.

4.2 Scope of Tests

In this section we will exhaustively list the tests that have been performed to show the scope of the code they covered.

- `DockingStationTest`

`CountFreePlaceOfA4parkingFreeStationStandard()` and `Count-FreePlaceOfA4parkingFreeStationPlus()` which verify that the correct number of spots is assigned to a standard or plus station.
`DoesTheStationHasElectrical()` checks if an electric bike can be added to a station.
`bikePickedlsEqualToTheBikeOfTheStation() & bikePickedLeavesTheP-arkingEmpty()` verify that when a user picks a bike, the bike is assigned to them and the docking spot becomes free.
`countFreePlacesReturnsCorrectNumberOfFreePlaces() & count-FreePlacesReturnsZeroWhenNoFreePlaces()` check that the correct number of empty slots is counted.
`whereBikeTypeWhenNoElectricalBikesAvailable() &, whereBikeType-WhenMechanicalBikeExists() &, whereBikeTypeWhenNoMechanicalBike-sAvailable() & whereBikeTypeWhenElectricalBikeExist()` verify that we are able to report the presence of different bike types in the station.

- `TestBicycleFactory`

`ifBicycleRequiredlsElectricalTheReturnedlsElectrical() & if-BicycleRequiredlsMechanicalTheReturnedlsMechanical() & ifBicy-cleRequiredNotMecaOrElectricalReturnNull()` verify that the correct bike type is created.
`testMechanicalBicycleAcceptVisitor() & testElectricalBicycleAc-ceptVisitor()` verify the implementation of the visitor pattern.

- `TestCardFactory`

`ifCardsRequiredIsVlibreReturnVlibre() &, ifCardsReguiredlsV-maxReturnVmax &, ifCardsRequiredlsNotVmaxNorVlibreReturnNoCard()` verify that the correct card type is created.
Tests of the form `testPrice...` check that the price and time-credit balance are correct for different combinations of bike, card, trip duration, and initial time-credit balance.
`testStationPlusFirstHour() & testStationPlusAfterFirstHour()` check the correct pricing for plus stations.

- `TestGPS`

`getDistanceWhenCoordinatesAreTheSame() & getDistanceBetweenT-woGPSCoordinates()` verify that distance calculations are correct.

- `TestPlanningStrategy`

`testStandardStationlsClosest4StationsClosestIsStation3()` checks that the optimal route is chosen.
`testPreferPlusStationInf() &, testPreferPlusStationSup() &, tes-tAvoidPlusStationinf() & testAvoidPlusStationSup()` verify that the route corresponds to the desired strategy regarding plus stations.

`testNoDesiredBikeAvailable() &, testNoStationAvailable() & test-NoStationUtile()` verify that there are no bugs when no route is available.

- `TestPlanningUser`

`ifUserlsAlreadyRentingABikeSendMessageWarning()` verifies that you cannot rent two bikes at the same time.
`testnbRides()` verifies the correct increment of the ride counter.
`testReturnMechanicalBike() & testReturnElectricalBike()` verify that the correct bike type is returned to the station.
`testReturnWildBike` verifies the pricing when a bike is left outside a station.

4.3 Problematic Tests

The many tests described above were crucial for verifying the capabilities and limits of our code.

For example, a series of tests that was particularly useful for detecting our failures were the tests `testNoDesiredBikeAvailable() &, testNoStation-Available() & testNoStationUtile()`. Indeed, this showed that we had not taken into account the possibility of no possible route. In this case, one might expect a user-facing message such as “No Vélib available at the moment.”

Tests of the form `testestPrice...` allowed us to detect many cases that we had not considered. This was for example the calculation of Balance Time Credit when the paid hour was not fully used or when stopping at a plus station, as well as the price calculation when a discount was applied thanks to the time-credit balance.

Moreover, this task showed us the difficulty of performing tests that isolate a single action on complex methods. For example, it was difficult to test the effects of a reservation on a user because we had to create a new environment each time with a User, a card, and a trip, for many different combinations.

5 Use Case Scenarios

5.1 Scenarios

Scenarios are run by the `runtest TestNumber` command. The command information is also provided in the console by the `help` function.

All scenarios start by loading `MyVelib.ini`, which first generates a network named `velib`, to which three users—Gaspard, Jeanne, Balthazar—are added with the different possible card cases: Vlibre, Vmax, and NoCard, respectively.

1. `runtest 1`: The first test demonstrates the use of `offline`, `display`, and `online`. A velib station is taken offline, and the system state is displayed. It is then brought back online, and another takes offline, displaying the result again.

2. `runtest 2`: The second test aims to demonstrate a real situation of renting and returning velibs. A user rents a velib at a first station, then returns it at a second station after 130 minutes. They then rent another velib at a third station and return it again to the second station. The stations are then sorted according to `LeastUsed` and the sorted stations are displayed.

3. `runtest 3`: Test 3 simply highlights the `planningRide` function by looking for a mechanical velib from the GPS point (0.1,0.5) to (1.0,0.2). The result is then displayed directly on the console.

4. `runtest 4`: The purpose of this scenario is to highlight the rental cost of a velib applied to the user. A user rents and returns a velib, and then the user's properties are displayed.

6 Conclusion

6.1 Summary

In this project, we coded an object-oriented Java application MyVelib to support a user during the use of a velib.

The main features we implemented allow: - Renting a mechanical or electric bike using a Vlibre card, a Vmax card, or no card.
- Planning the best possible route, favoring or avoiding certain stations.
- Charging a user.
- Monitoring a user's balance or the status of a station.

This project was very educational in terms of different Java programming techniques and gave us an overview of the challenges in Software Engineering.

6.2 Possible Improvements

• Open-closed principle and design patterns: As mentioned in the discussion on architecture, some parts of our code do not respect the open-closed principle. Thus, to make our code more modular and easier to modify, certain design patterns could be used. In particular, the Model-View-Controller pattern should be implemented in the Manager class. In addition, an observer should be implemented to facilitate interactions between the bicycle and station classes. This design pattern would allow direct reporting of modifications to a bike contained in a station.

• Parallel programming: Our code is not yet very heavy, so running on a single processor is not a handicap. However, it could grow, and it would therefore be wise to start looking at how it could be parallelized to allow multithreading. Indeed, this design flaw affects, for example, the game Minecraft, which started as a small program but is now handicapped by its lack of multithreading. However, one disadvantage of multithreading is the risk of conflicts on shared variables. Therefore, more tests would be required for our program.

• Tests and scenarios: In order for this application to be available for commercial use, more tests will be needed to certify the robustness of our code. Thus, in the future we should consider adding even more tests. In addition, there are still certain exceptions that have not yet been taken into account. This is the case, for example, when there is no route that favorably uses a bike between the departure and arrival points.

Also, we could create a 5th scenario to highlight the features of plus stations and the cost of leaving a bike in the wild. We would follow a rather erratic user, with a standard station and a plus station at equal distance. First they would take a route preferring the plus station, then the same route avoiding it. Finally, they would abandon their bike outside a station and we would display the price of their trip.

• Graphical User Interface: With a view to commercial deployment, our program should be made more user-friendly for a general user. Thus, much progress can still be made on the User Interface, which is why we would propose adding a Graphical User Interface. This would allow the user to perform actions directly in a window with buttons. For example, one could imagine a button "Load Velib Card" that would increase a user's balance or a JRadioButton to choose between an electric or mechanical bike ride.

• User Experience: For now, all commands are launched from the terminal, which is quite hostile for the user. Thus, we could add listeners to improve their experience. For example, to choose a destination, a window representing a map could be generated by the Graphical User Interface, and then one would simply click on their destination. The x,y position of the cursor would give the GPS arrival position.

• Cross-platforming: Finally, this program is not practical for a real situation because it requires a computer. More realistically, a user would want to access MyVelib on their way with their phone. However, to turn our software into an application, would we have to rewrite it entirely in Swift for iOS? Regarding the UI, indeed we would have to start from scratch. There is no JVM installed on iOS. On the server side, however, it will be very easy to implement since it is one of the standard languages for this use. More in-depth security-oriented tests would then be needed.

CentraleSupélec