myVelib - a bike sharing system

Author :
BADAOUI Abdennacer
MRABET Hatim

Supervisors :
Mr. Paolo BALLARINI
Mr. Arnault LAPITRE

Table des matières

1 Introduction

This technical report presents the development of myVelib, a Java framework designed to manage a bike sharing system inspired by successful implementations like Velib in Paris. The project is divided into two parts : **Part 1** focuses on the core infrastructure development, while **Part 2** concentrates on designing and developing a command line user-interface for the myVelib system.

The main objective of this project is to develop a Java framework, myVelib, that addresses the requirements of managing a bike sharing system. The framework will support the core functionalities of the system, such as bike and station management, user registration and interaction, and maintenance operations. Additionally, a user-friendly interface will be designed to enhance the user experience and enable seamless interaction with the myVelib system.

2 myVelib core

The first part of the project involves the design and development of the core infrastructure for the myVelib system. This core functionality encompasses the following key requirements :

2.1 Docking stations

The `DockingStation` class represents a docking station in the MyVelib system. It manages the parking slots and bikes available at the station. Here is a description of the class and its methods :

2.1.1 Attributes

The `DockingStation` class has the following attributes :

— **`NumberOfParkingSlots`** : An integer representing the total number of parking slots in the docking station.
— **`StationLocation`** : An instance of the `Location` class representing the location of the docking station.
— **`ratioOccupied`** : A double value indicating the ratio of occupied parking slots in the docking station.
— **`ratiofree`** : A double value indicating the ratio of free parking slots in the docking station.

— `ratioElectricalBikes` : A double value indicating the ratio of electrical bikes in the docking station.
— `StationStatus` : A string representing the status of the docking station (e.g., "Online", "Offline").
— `StationType` : A string specifying the type of the docking station (e.g., "Standard", "Plus").
— `generatorStationId` : A static integer representing the generator station ID (used to assign unique IDs to docking stations).
— `StationId` : An integer representing the ID of the docking station.
— `parkingSlots` : An ArrayList of `ParkingSlot` objects representing the parking slots in the docking station.
— `numberOfRents` : An integer representing the number of bike rental operations from this docking station.
— `numberOfReturns` : An integer representing the number of bike return operations to this docking station.

2.1.2 Methods

The `DockingStation` class provides the following methods :
— `DockingStation(numberOfParkingSlots, ratioOccupied, ratiofree, ratioElectricalBikes, stationLocation, stationStatus, stationType)` : The constructor of the `DockingStation` class initializes the docking station with the specified parameters. It creates parking slots based on the ratios and initializes the bike distribution accordingly.
— `showDockingStationState()` : This method displays the state of the docking station, including the station type, status, location, and information about each parking slot (ID, status, and bike information if occupied). It also provides a report with the number of occupied slots, electrical bikes, mechanical bikes, free slots, and out-of-order slots.
— `rentFromStation(user, bikeType)` : This method allows a user to rent a bike of the specified type from the docking station. It checks if the station is offline and throws an exception if that is the case. It searches for an available bike of the specified type in the parking slots and performs the rental operation.
— `returnedToStation(user)` : This method is used when a user returns a bike to the docking station. It checks if the station is offline and throws an exception if that is the case. It searches for a free parking slot in the docking station and updates the slot status and bike information accordingly.

— **`hasBike(type)`** : This method checks if the docking station has a bike of the specified type available for rent.
— **`hasFreeSlot()`** : This method checks if the docking station has a free parking slot available.
— Getter and setter methods for various attributes.

The `DockingStation` package also includes other classes such as `EndDockingStation` and `StartDockingStation`, which implements the `RidesPlanning` interface and provides methods for finding the nearest docking station based on location and bike type, and comparators (`LeastOccupiedStationComparator` and `MostUsedComparator`) for sorting docking stations based on usage statistics.

The UML diagram below describes the different parts of this package :

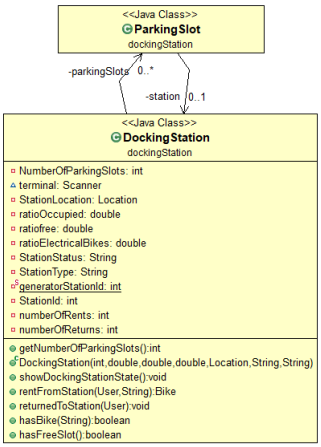

FIGURE 1 – UML Diagram of dockingStation and ParkingSlot class

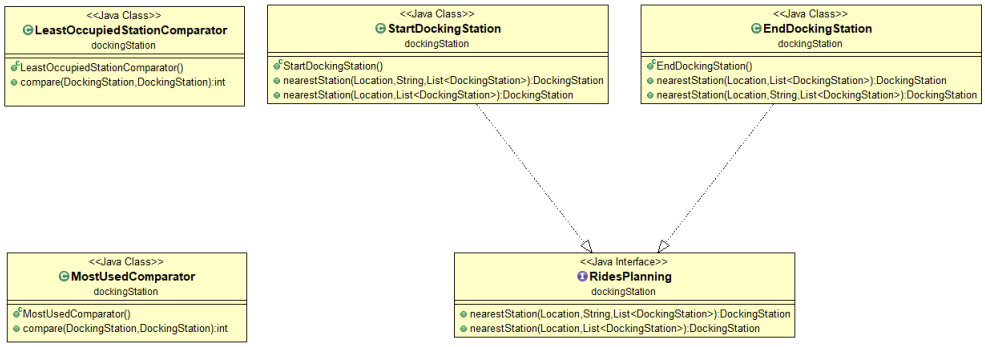

FIGURE 2 – UML Diagram of RidePlanning and Comparators

2.2 Parking Slots

this class represents the parking slots of a docking station, it has the following attributes :

- **DockingStation**(it represents the docking station in which this parking slot exists)
- **slotStatus** (it's a string that could take the values 'occupied' or 'free')
- **bike** (an attribute of type bike that takes null if there is no bike in the parking slot, and if there's a bike, it takes as value the bike parked there)
- **slotId**

The UML diagram of this class is the following :

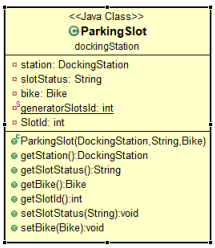

FIGURE 3 – UML Diagram of ParkingSlot class

2.3 Bike

For this class, we used a **Factory Pattern**, to handle the creation of either mechanical bikes or electrical bikes. A mechanical (or electrical) bike has the following atrributes :

- **bikeID**
- **location**
- **dockingStation**

The bike class contains many methods, including its getters and setters. It also contains a **rent** method, that allows to rent the bike by assigning the bike to the user who rents the bike using the setRentedBike() of the User class. The bike class also includes a **returnBike** method that allows to return the bike to a station.

The definition of each interface and class of our Factory pattern will be described in the following UML diagram :

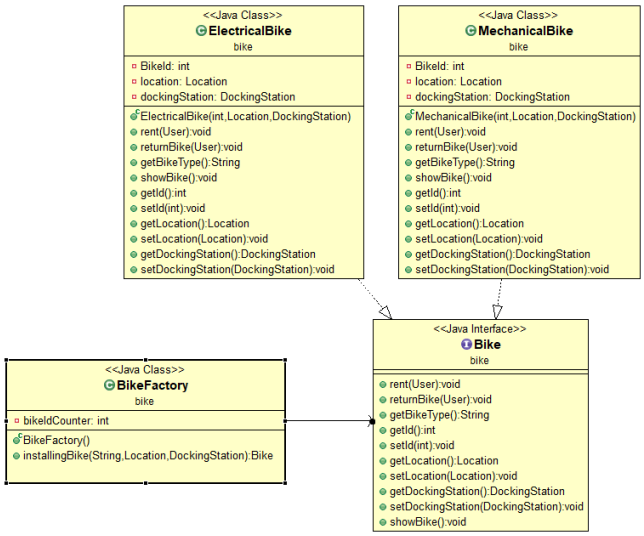

FIGURE 4 – UML Diagram of Bike Factory design

2.4 User

2.4.1 User Class

The `User` class represents a user in the MyVelib system. It contains various attributes and methods to manage user information and interactions with the system.

Attributes of the `User` class :

— **`userName`** : A string representing the name of the user.
— **`id`** : A static integer used to generate a unique ID for each user.
— **`unique_id`** : An integer representing the unique ID of the user.
— **`userLocation`** : An instance of the `Location` class representing the current location of the user.
— **`creditCard`** : An instance of the `CreditCard` class representing the user's credit card information.
— **`registrationCard`** : An instance of the `RegistrationCard` class representing the user's registration card information.
— **`balanceOfTotalCharges`** : A double value representing the total balance of charges for the user.
— **`hasRentedBike`** : A boolean indicating whether the user has currently rented a bike.
— **`rentedBike`** : An instance of the `Bike` class representing the bike currently rented by the user.

— `NumberOfRides` : An integer indicating the total number of rides taken by the user.
— `totalTimeSpentOnRides` : A double value representing the total time spent on rides by the user.
— `startStation` : An instance of the `DockingStation` class representing the starting docking station for the user's ride.

Methods of the `User` class :

— `User(name, userLocation, creditCard, registrationCard)` : The constructor method initializes a `User` object with the specified name, location, credit card, and registration card.
— `hasRentedBike()` : This method returns a boolean value indicating whether the user has currently rented a bike.
— `sethasRentedBike(bo)` : This method sets the `hasRentedBike` attribute to the specified boolean value.
— `setRentedBike(bike)` : This method sets the `rentedBike` attribute to the specified `Bike` object and updates the `hasRentedBike` attribute accordingly.
— `getRentedBike()` : This method returns the `Bike` object currently rented by the user.
— `getStartStation()` : This method returns the starting docking station for the user's ride.
— `setStartStation(station)` : This method sets the starting docking station for the user's ride to the specified `DockingStation` object.

Overall, the User class provides functionalities to manage user information such as name, location, credit card, and registration card. It also keeps track of the user's rented bike, number of rides, total time spent on rides, and balance of charges. Users can interact with the system by renting bikes, returning bikes, and accessing their ride history.

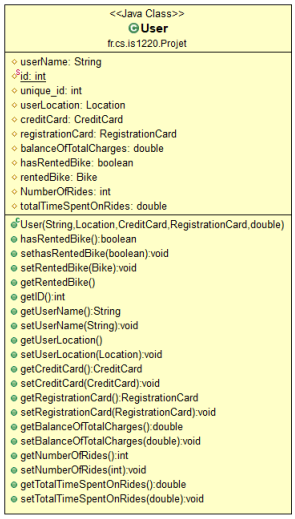

FIGURE 5 – UML Diagram of User class

2.5 Cards

The card system implementation consists of several classes that form the core of the system. The key classes and their functionalities are as follows :

2.5.1 CreditCard Class

The `CreditCard` class represents a credit card and stores important details such as the owner's name, card number, expiration date, and secret code. It provides getter and setter methods to access and modify these attributes. Additionally, it overrides the `toString()` method to generate a string representation of the credit card object, facilitating convenient display and debugging.

2.5.2 RegistrationCard Interface

The `RegistrationCard` interface defines a contract for registration card classes in the system. It specifies methods for retrieving the card type, card ID, adding time credit balance, and retrieving the current time credit balance. All registration card classes in the system must implement this interface to ensure consistency and interoperability.

2.5.3 RegistrationCardFactory Class

The `RegistrationCardFactory` class follows the **Factory Design Pattern** and serves as a central component for creating registration cards. It contains a method `createRegistrationCard(String cardType)` that takes a card type as input and returns an instance of the corresponding registration card. Currently, it supports the creation of `Vlibre` and `Vmax` registration cards based on the provided card type.

2.5.4 Vlibre Class

The `Vlibre` class represents a specific type of registration card called "Vlibre." It implements the `RegistrationCard` interface and provides implementations for the defined methods. Each `Vlibre` instance has a unique ID, which is incremented each time a new card is created. Additionally, it maintains a time credit balance attribute, allowing for the addition and retrieval of time credit.

2.5.5 Vmax Class

The `Vmax` class represents another type of registration card called "Vmax." Similar to `Vlibre`, it implements the `RegistrationCard` interface and provides implementations for the defined methods. Each `Vmax` instance possesses a unique ID, which is incremented upon card creation. It also tracks the time credit balance, allowing for time credit addition and retrieval.

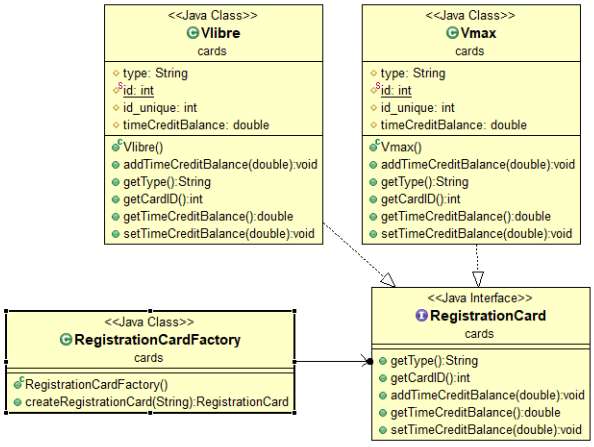

FIGURE 6 – UML Diagram of Cards Factory design

2.6 Ride

The `Ride` class and related classes are integral parts of the system and handle the logic and cost calculation for rides. The key classes and their functionalities are as follows :

2.6.1 Ride Class

The `Ride` class represents a ride taken by a user on a bike. It contains attributes such as the user, bike, starting and ending docking stations, starting and ending locations, ride duration, cost strategy, and cost. The class provides a constructor to initialize the ride object with the necessary information. Additionally, it has a method `calculateCost()` that calculates the cost of the ride based on the ride duration, bike type, card type, and whether the bike was returned to a docking station. The `determineCostStrategy()` method determines the appropriate cost strategy based on the docking stations involved.

2.6.2 RideStrategy Interface

The `RideStrategy` interface defines the contract for different cost strategies used in calculating the cost of a ride. For computing the cost, we used **Strategy Design Pattern**, to respect the different strategies of the cost. This interface includes a single method `calculateCost()` that takes the ride duration, bike type, card type, and return status as parameters and returns the calculated cost. Classes implementing this interface provide different cost calculation algorithms based on specific conditions.

2.6.3 DockingStationCostStrategy Class

The `DockingStationCostStrategy` class is an implementation of the `RideStrategy` interface. It calculates the cost of a ride when the bike is rented and returned to a docking station. The `calculateCost()` method checks the card type and delegates the cost calculation to specific methods based on the card type and bike type. It returns the calculated cost.

2.6.4 FreePositionCostStrategy Class

The `FreePositionCostStrategy` class is another implementation of the `RideStrategy` interface. It calculates the cost of a ride when the bike is rented from a free position and returned to a docking station. The class takes an instance

of the `DockingStationCostStrategy` as a constructor parameter and applies a 10 % discount to the cost calculated by the docking station strategy.

2.6.5 RoadParkingCostStrategy Class

The `RoadParkingCostStrategy` class is yet another implementation of the `RideStrategy` interface. It calculates the cost of a ride when the bike is not returned to a docking station but left parked in the road. The class also takes an instance of the `DockingStationCostStrategy` as a constructor parameter and applies a 10 % malus to the cost calculated by the docking station strategy.

These classes work together to determine the appropriate cost calculation strategy based on the ride conditions and calculate the cost accordingly. The `Ride` class encapsulates the ride-related information and uses the selected cost strategy to calculate the cost of the ride. The cost strategies provide flexibility in handling different ride scenarios and allow for easy extension with additional strategies if needed.

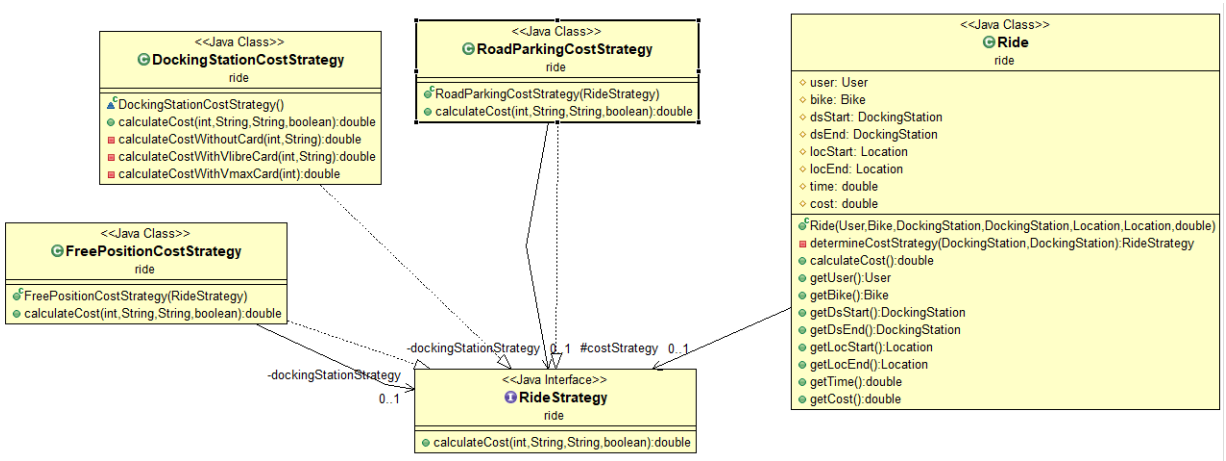

FIGURE 7 – UML Diagram of Ride package

2.7 MyVelib

The `MyVelib` class represents the main class of the MyVelib system, which manages docking stations, users, and rides. It includes various methods for system management and user interactions.

2.7.1 MyVelib Class

The `MyVelib` class has the following attributes :

— **`dockingStations`** : A list of `DockingStation` objects representing the docking stations in the system.
— **`users`** : A list of `User` objects representing the users registered in the system.
— **`NumberOfStations`** : An integer indicating the total number of docking stations in the system.
— **`TotalParkingSlots`** : An integer representing the total number of parking slots in all docking stations.
— **`ratioOccupied`** : A double value indicating the ratio of occupied parking slots in each docking station.
— **`ratioElectrical`** : A double value representing the ratio of electrical bikes in each docking station.
— **`side`** : A double value representing the size of the area in which the docking stations are randomly located.
— **`name`** : A string specifying the name of the MyVelib system.

The `MyVelib` class provides the following methods :

— **`MyVelib(name, numberOfStations, totalParkingSlots, ratioOccupied, ratioElectrical, side)`** : The constructor of the `MyVelib` class initializes the system with the specified parameters. It randomly generates the locations of docking stations based on the given `numberOfStations` and `side`.
— **`State()`** : This method displays the state of the MyVelib system, including the total number of stations, the number of stations by type (standard or plus), and the number of stations in different statuses (on service or offline). It also shows the state of each docking station individually.
— **`newRide(User user, String biketype, Location LocEnd, double time)`** : This method creates a new ride for the specified user, bike type, end location, and duration. It checks if the user already has a rented bike and throws an exception if that is the case. It finds the nearest starting and ending docking stations, rents a bike from the starting station, and creates a new ride object.
— **`dropbike(Ride ride)`** : This method is used to drop the bike at the end of the ride. It returns the bike to the ending docking station and updates the user's ride history.
— **`addDockingStation(DockingStation dockingStation)`** : This method

adds a new docking station to the MyVelib system.

— `addUser(User user)` : This method adds a new user to the MyVelib system.

— `UserBalance(User user)` : This method displays the balance information for the specified user, including the number of rides, total amount of charges, time credit (if applicable), and total time spent on rides.

— `StationBalance(DockingStation station)` : This method displays the balance information for the specified docking station, including the number of rents and returns operations.

— `sort(String method)` : This method sorts the docking stations based on the specified sorting method. It supports two methods : "MostUsed" and "LeastOccupied". The sorted stations are displayed with their corresponding station IDs.

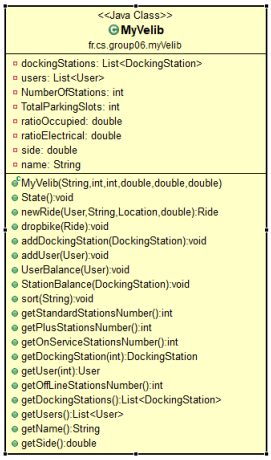

FIGURE 8 – UML Diagram of myVelib class

3 myVelib user interface

3.1 CLUI

The `MyVelibCLUI` class provides a command-line user interface (CLUI) for interacting with the MyVelib system that I have developed. This CLUI allows users to perform various operations and access different functionalities of the system using simple commands.

1. **Setup Command :** The `setup` command enables the creation of a new MyVelib network. Users can specify the network name, the number of stations, the number of slots per station, and various

ratios related to the network configuration, such as the occupancy ratio and the electrical bikes ratio. Alternatively, users can create a network with default parameters.

2. **Add User Command :** The `addUser` command allows users to add new users to the MyVelib system. Users can provide the user's name, the type of registration card (e.g., "Vlibre" or "Vmax"), and the name of the MyVelib network to which the user will be associated. The command handles the creation of a registration card and assigns it to the user, and then adds the user to the specified network.
3. **Offline and Online Commands :** The `offline` and `online` commands enable users to change the status of a docking station within a MyVelib network. By providing the network name and the station ID, users can mark a station as either offline or online, respectively. These commands are useful for managing the availability and maintenance of docking stations.
4. **Rent Bike Command :** The `rentBike` command allows users to rent a bike from a specific station. Users need to provide their user ID and the station ID from which they want to rent the bike. The command checks the availability of bikes at the station and assigns a bike to the user if one is available. It also handles the associated operations, such as updating the user's rented bike status and generating a ride record.
5. **Return Bike Command :** The `returnBike` command enables users to return a rented bike to a specific station. Users need to provide their user ID, the station ID where they want to return the bike, and the duration of the ride. The command handles the necessary calculations, including ride cost, and updates the user's information and the station's availability.
6. **Display Station and User Commands :** The `displayStation` and `displayUser` commands provide information about a specific station or user within a MyVelib network. By providing the network name and the station/user ID, users can retrieve details such as the station's balance (available bikes and parking slots) or the user's balance (rented bikes, ride history, etc.). These commands offer insights into the system's status and user activities.
7. **Sort Station Command :** The `sortStation` command allows users to sort the docking stations within a MyVelib network based on a specific policy. Users provide the network name and the desired sorting

policy, such as "mostOccupied" or "leastOccupied." The command rearranges the stations according to the chosen policy, facilitating better station management and analysis.

8. **Runtest Command :** The `runtest` command enables users to execute a predefined set of test cases that are specifically designed to evaluate all the features of our MyVelib system. This command provides a convenient way to validate the functionality and performance of the system by running various scenarios and assessing the system's behavior under different conditions. The test cases cover a comprehensive range of operations and interactions within the MyVelib system, ensuring thorough testing and reliable evaluation of its features.

We also added a method called `run`, that handles scanning the input of the user, and that executes the correponding command.

We finally added a main class, that should be run to be able to interact with our user interface.
The `MyVelibCLUI` class provides a comprehensive set of commands that cover essential functionalities of the MyVelib system. With these features, users can easily set up a network, add users, manage station statuses, rent and return bikes, retrieve station and user information, and perform station sorting. The CLUI enhances the user experience by providing a convenient and efficient way to interact with the MyVelib system through the command line.

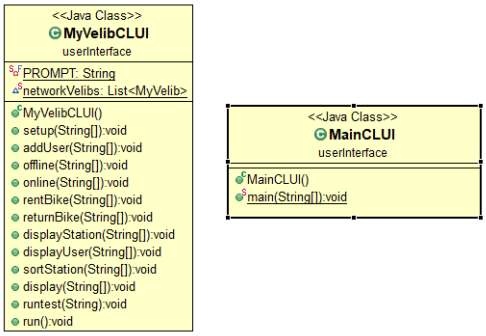

FIGURE 9 – UML Diagram of CLUI

3.2 GUI

In addition to the command-line user interface (CLUI), we have enhanced our project by introducing a simple graphical user interface (GUI) called MyVelibGUI. The MyVelibGUI provides a more user-friendly and intuitive way for users to interact with the MyVelib system.
The MyVelibGUI features a clean and intuitive design, with a text input field where users can enter commands and a text area that displays the output of the executed commands. The GUI supports essential functionalities such as setting up the system, adding users, marking stations as offline or online, renting and returning bikes, sorting stations, and displaying station and user information.

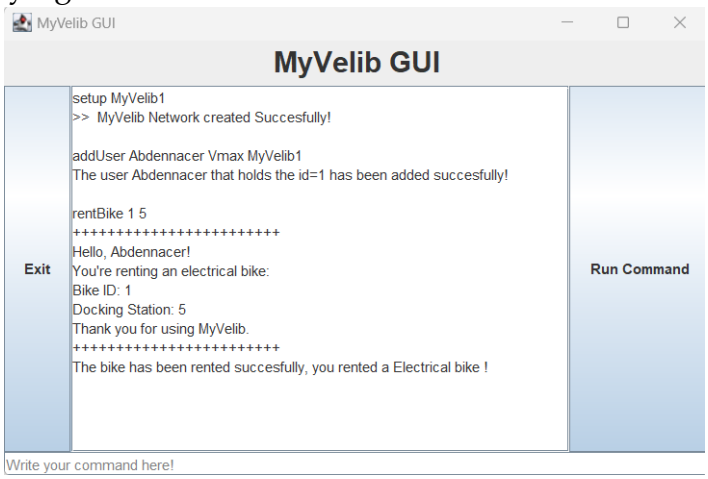

FIGURE 10 – Graphical User Interface

4 Evaluation of our final product :

4.1 Test scenario cases :

To evaluate the functionalities, we defined some test case scenarios that run the output of several commands grouped in some text files.
We even added an initial configuration file called **my _velib.ini**, which allows initializing a default MyVelib system directly after running our program. The default network has :
— 10 stations
— 10 parking slots per station
— 0.5 ratio of occupied parking slots distributed uniformly, which

means 5 bikes per station
— 0.5 ratio of electrical bikes
— A side of 4 km

4.1.1 Test case scenario 1 :

This test case scenario is very basic. It allows us to evaluate the default network setup by adding a user called **Hatim**, who owns a **Vmax** card, to the default network. This user rents a bike with ID 1 from the station with ID 5. We then display the information of this user and the whole station.

4.1.2 Test case scenario 2 :

This test case scenario uses more commands than the previous one. We first add 2 users to the default network. The first user, named **Abdennacer**, owns a **Vlibre** card, while the second one called **Hatim** has no registration card. **Abdennacer** rents a bike from station 5. Then, we turn station 8 **offline**. After that, we display Abdennacer's information to check if the rental count has been updated. He then returns the bike to station 8 after a duration of 1000, and we check if the return count has been updated by displaying the user's information again.
To check if the user has already returned his bike, we use the command **returnBike** on different stations to see if it's possible. Normally, he is not supposed to be able to do that. Each time we return the bike, we add a different duration to see if the user's information is being updated even if he has already returned the bike. We find that the information does not change.
After this, we display station 5's statistics, and we should find 1 rent since there has been only one rent operation from this station. All other statistics must be null. We then display station 2's statistics, and we find that all its statistics are set to 0 since there have been no operations in this station.
Finally, we sort stations using the two possible comparators. For the first criteria (most used), either station 5 or 8 should appear first. For the least occupied, station 5 should be first. We finally display the default network's information.

4.1.3 Test case scenario 3 :

This is the most robust test case scenario since it covers all possible exceptions. We first set up a new Velib system called **NewVelib** with 5 stations, 25 parking slots distributed uniformly between the stations, a side length of **4 km**, and a total of 25 bikes, which means all parking slots are occupied. We then add 6 users : **Hatim1**, **Hatim2**, **Hatim3**, **Hatim4**, **Hatim5**, and **Abdennacer1**, each of them having a Vlibre card. **Hatim1** to **Hatim5** rent bikes from station 11 (since the IDs are incremented for all Velibs, the first station of our new Velib system has an ID equal to 11 [10 + 1], the second one has an ID equal to 12, and so on). Normally, since all the stations have exactly 5 bikes, there will be 5 bikes in station 11. However, when **Abdennacer1** tries to rent a bike from this station, he won't be able to do so.
Afterward, **Abdennacer1** rents a bike from station 12. **Hatim1** then tries to return his bike to station 3 (remember : all the stations are fully occupied before a bike is rented from them). Since there are no available parking slots in station 3, **Hatim1** won't be able to return his bike there. He will then return it to station 2 instead (since **Abdennacer1** has already rented a bike from it, there is a free parking slot available). All the other users return their bikes to station 11 after a duration of 1000, except for **Abdennacer1**, who tries to rent another bike while still using the one he rented before. He won't be able to do so because our system forces users to return the bike before renting a new one. **Abdennacer1** will then return his bike to station 11, like his friends, after a duration of 1000.
Next, station 11 goes offline, and **Hatim1** tries to rent a bike from it but fails since the station is offline. Afterward, the same station goes back online, and **Hatim1** is able to rent his bike from it. He returns it to the same station after a duration of 1000.
REMARK : After each return, we notice that the cost differs among users, even when they have the same duration for their ride. This is related to the fact that the electrical bike is more expensive than the mechanical one.

```
Step .17

 Duration :1000.0 min.          Bike Type: Electrical
============================================================
    Bike Returned ! Thank you Hatim2 for using MyVelib ;)
============================================================
 The cost of your Ride is = 30.666666666666668 euros.

Step .18

 Duration :1000.0 min.          Bike Type: Mechanical
============================================================
    Bike Returned ! Thank you Hatim3 for using MyVelib ;)
============================================================
 The cost of your Ride is = 15.333333333333334 euros.
```

FIGURE 11 – Difference between costs for electrical and mechanical bikes

After this, we display the statistics of station 11. Normally, if we keep count, it should have **6 rents and 6 returns**. Then we display the statistics of **Hatim1**, and we find 2 rides with a total duration of 2000 minutes. Finally, we sort the stations based on the most used criteria, and station 11 should be first, which is what we get.

5 The guide to use the user interface

To use the user interface, the **Main.CLUI.java** file, that exist in the **fr.cs.group06.myVelib.userinterface** package. After running this file, the **Terminal** appears. First, a default MyVelib network is initialized, that has the properties defined in the **my_velib.ini** configuration file.To execute the commands, the user should follow the steps below :

— `setup <name> <nstations> <TotalNumberofSlots> <ratioOccupiedParkingSlots> <ratioElectricalBikes> <Side>` : Creates a myVelib network with the given name. The network consists of `nstations` stations, with a total of `TotalNumberofSlots` parking slots distributed unifromly between the stations. The stations are arranged in a uniform manner over an area, which is squared with a side length of `s`. the `ratioOccupiedParkingSlots` helps to find the number of bikes by multpilying this ration by the total number of parking slots, these bikes are distributed unifromly between the the stations, we also use the `ratioElectricalBikes` to specify the ratio of electrical bikes in the Velib system. (**REMARK : <ratioOccupiedParkingSlots> <ratioElectricalBikes> <Side> are doubles, don't forget for <Side> to write 4.0 instead of 4)**

— `addUser <userName,cardType, velibnetworkName>` : Adds a user with the given name `userName` and card type `cardType`(Vlibre, Vmax, none)

to the myVelib network `velibnetworkName`. The card type can be "none" if the user has no card.

— `offline <velibnetworkName, stationID>` : Puts the station with ID `stationID` of the myVelib network `velibnetworkName` offline.

— `online <velibnetworkName, stationID>` : Puts the station with ID `stationID` of the myVelib network `velibnetworkName` online.

— `rentBike <userID, stationID>` : Allows the user with ID `userID` to rent a bike from station `stationID`. If no bikes are available, the command should behave accordingly.

— `rentBike <userID, GPS_Position,>` : Allows the user with ID `userID` to rent a bike parked at a given GPS position.(**REMARK : GPS_Position is written in the following format : x,y where x and y are doubles)**

— `returnBike <userID, stationID, duration>` : Allows the user with ID `userID` to return a bike to station `stationID` for a given duration. If no parking bay is available, the command should behave accordingly. This command should display the cost of the rent.

— `returnBike <userID, GPS_Position, duration>` : Allows the user with ID `userID` to return a bike in a given GPS position for a given duration. If no parking bay is available, the command should behave accordingly. This command should display the cost of the rent.

— `displayStation<velibnetworkName, stationID>` : Displays the statistics of station `stationID` of a myVelib network `velibnetwork`.

— `displayUser<velibnetworkName, userID>` : Displays the statistics of user `userID` of a myVelib network `velibnetwork`.

— `sortStation<velibnetworkName, sortpolicy>` : Displays the stations in increasing order with respect to the sorting policy of the user `sortpolicy`. (**REMARK : possible policies : mostused leastoccupied)**

— `display <velibnetworkName>` : Displays the entire status (stations, parking bays, users) of the myVelib network `velibnetworkName`.

— `runtest <testScenario.txt>` : Allows the user to run the predefined test case scenario explained above.
(REMARK : could be either 1, 2 or 3 since we defined 3 test case scenarios)

6 Splitting work between the two members of the group :

Hatim	User package Cards Package Bike package JUnit Test for the classes made User inteface command (rentBike, returnBike, setup, DisplayStation)
Abdennacer	Ride package Docking Station package Location package JUnit Test for the classes made User interface command (Online, addUser, Offline, DisplayUser, Sort

7 Advantages and limitations

Our project boasts several advantages in handling various scenarios encountered within the bike-sharing system. For instance, it effectively manages situations where a user is prohibited from renting a second bike, parking at an offline station, or parking at a station with no available slots. Moreover, the inclusion of the Ride class enables our system to propose personalized plans to users based on their location. By determining the nearest online docking station with their preferred bike and an end docking station with a free slot, we provide users with convenient route options. Additionally, our system accurately calculates user and station statistics, ensuring the proper tracking of user and station statuses. Furthermore, it allows for a more tailored setup by incorporating ratios of electric bikes, occupied slots, and free slots, facilitating the calculation of out-of-order slots that cannot be used. Our project offers several advantages in handling mistyping errors and logical situations encountered by users through the Command Line User Interface (CLUI). The CLUI incorporates robust error handling mechanisms, effectively minimizing the impact of mistyped commands or incorrect inputs. It provides intelligent feedback and suggestions to users, helping them rectify their mistakes and ensure proper command execution. Additionally, the CLUI handles logical situations with precision, such as enforcing restrictions on renting multiple bikes or parking at offline or full stations. By anticipating and addressing these logical scenarios, our system ensures smooth operation and a seamless user experience.
However, there is a limitation regarding users dropping off their bikes

outside of docking stations. To address this, we have implemented a special "null" docking station that accounts for these cases. Despite this limitation, we are still able to track the locations of bikes through their assigned positions, ensuring comprehensive monitoring of the system. Another limitation of our project is the simplicity of the Graphical User Interface (GUI). While the GUI effectively serves its purpose, there is room for improvement in terms of aesthetics, functionality, and user experience. Enhancing the GUI with more visually appealing elements, intuitive navigation, and additional features would contribute to a more polished and user-friendly interface. By investing in the development of a more sophisticated GUI, we can further enhance the overall usability and attractiveness of our system.

8 Conclusion

In summary, our project successfully addresses the challenges of managing a bike-sharing system by providing a comprehensive solution. The Command Line User Interface (CLUI) efficiently handles mistyping errors and logical situations, ensuring a smooth user experience. The integration of the Ride class allows for personalized planning, suggesting the nearest available docking stations and bikes based on user location. The system accurately calculates statistics, tracks user and station statuses, and allows for a more specific setup. However, there is room for further improvement to make the system even more robust by handling more complicated scenarios and incorporating additional functionalities. With continued development, our project has the potential to enhance the bike-sharing experience and meet the evolving needs of users.