Report - MyVelib project

Génie logiciel orienté objet

Séléna Jean-Mactoux
Lucas Beziers

May 28th 2023

Contents

1 Introduction

The myVelib project aims to develop a Java framework for managing a bike sharing system similar to Velib in Paris. This system allows city inhabitants to rent bicycles and freely cycle around a metropolitan area. To accomplish this, the project involves designing and implementing two main parts: the myVelib core and the myVelib user-interface.

Part 1: myVelib core The myVelib core focuses on the fundamental infrastructure of the bike sharing system. The key components include docking stations, various types of bicycles (both mechanical and electrically assisted), users with registration cards, and a maintenance crew responsible for bike collection and replacement. The core infrastructure should facilitate seamless interaction among these elements.

Part 2: myVelib user-interface The myVelib user-interface serves as the means for users to interact with the bike sharing system. This component requires designing and implementing an intuitive and user-friendly interface using the Java framework. The user-interface should enable users to perform essential tasks, such as viewing available bicycles, renting bikes, checking the status of docking stations, and managing their interactions with the system.

To enhance the understanding of the model's structure and its different classes, we have prepared a Javadoc documentation that covers all aspects of our project. To access this documentation, please open the file **index.html** located in the "doc" folder of the project. It provides valuable information on the organization and functionality of the model, allowing you to navigate through its components effectively.

2 UML diagram

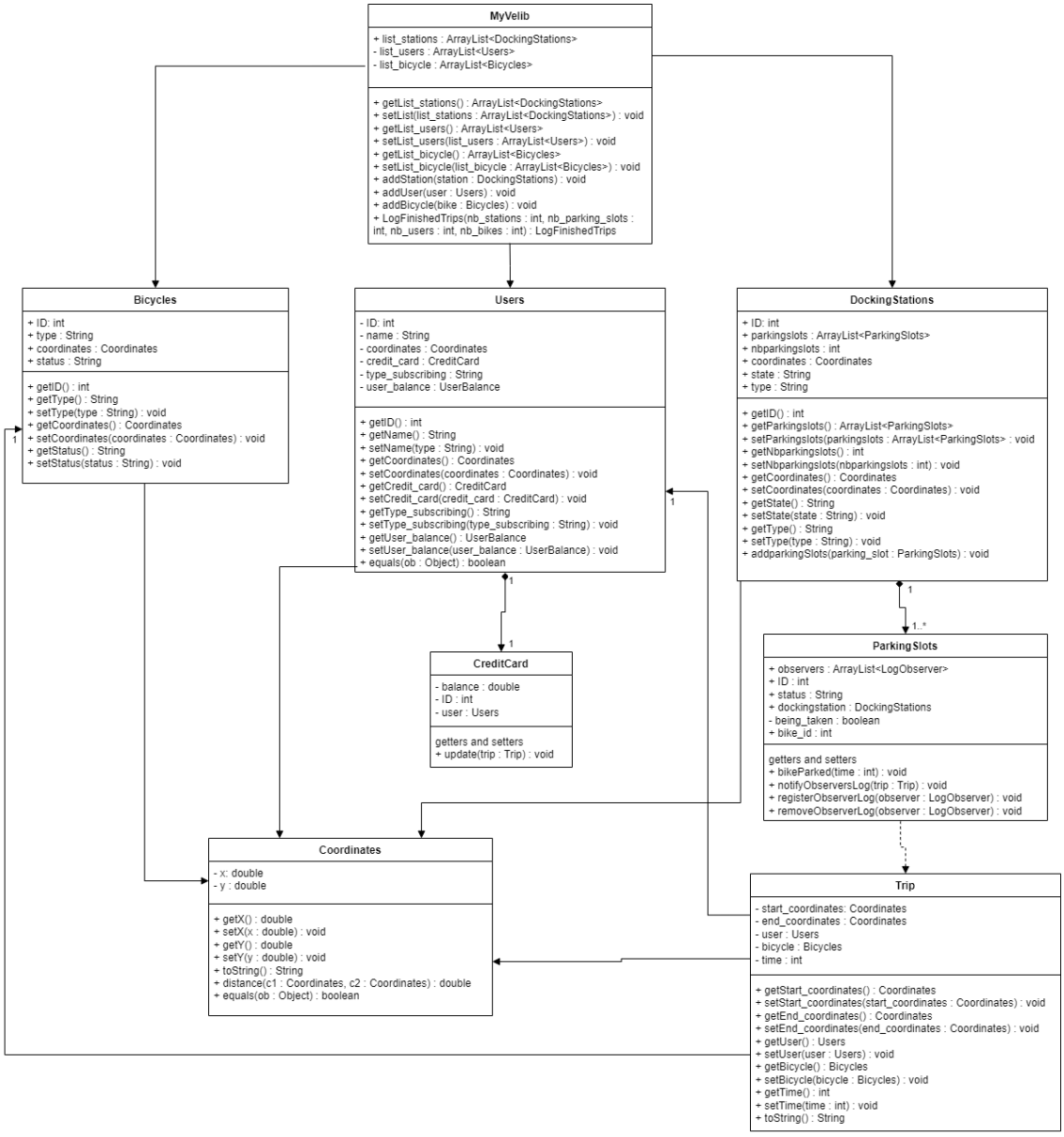

Figure 1: UML diagram showing main classes

This diagram shows how the main classes interact with each other. MyVelib corresponds to the whole model, hence it has attributes corresponding to the list of all stations, the list of all users and the list of all bicycles. It is needed to instantiate it only once.

2.1 Bicycles

The class Bicycles models a bike in our model:

- **ID**: Its unique ID.
- **type**: Can be *electrical* or *mechanical*.
- **coordinates**: The coordinates of the bike. If the bike if parked in a docking station, then its coordinates are the same as the station ones.
- **status**: Can be *available* or *not_available* regarding if someone is driving it or not.

2.2 Users and credit cards

The class Users models a user of the system in our model:

- **ID**: Its unique ID.
- **name**: Its name.
- **coordinates**: Its current coordinates.
- **credit_card**: The credit card associated to the user.
- **type_subscribing**: Can be *nocard*, $Vlibre$ or $Vmax$
- **user_balance**: Its current time balance.

Since users are objects, we created a new method *equals*() to compare two of them: the method retrieves the ID of another user and compare it with the first one. Each user is associated with one and only one credit card. The credit card has its own class CreditCard since we didn't want to overload the class Users with too many attributes. Hence the CreditCard class is described as follow:

- **balance**: Its current money balance.
- **ID**: Its unique ID.
- **user**: The user associated with this card.

2.3 Docking stations and parking slots

The class DockingStations models a docking station in our model:

- **ID**: Its unique ID.
- **parkingslots**: The list containing all its parking slots.
- **nbparkingslots**: Its number of parking slots.
- **coordinates**: Its coordinates.

- **state**: Can be *service* or $offline$.
- **type**: Can be *standard* or *plus*

Each docking station contains different parking slots with their unique IDs. Hence we decided to create a class ParkingSlots in order to distinguish one slot from another. Even if this class is important, we moved it to a package different from main in order to make clearer the observer pattern (Figure 6). This class is described as follow:

- **observers**: The list of observers for this slot. It is explained in details in section 5.
- **ID**: Its unique ID.
- **status**: Its number of parking slots.
- **dockingstation**: The docking station containing the slot.
- **being_taken**: Boolean that is put *true* when the bike of the slot is taken then returns to $false$
- **bike_id**: The ID of the bike parked on the slot (0 if there is no bike).

2.4 Trip

The class Trip corresponds to a trip: it allows us to manipulate this concept easily in the system since it is used for pricing and also stored in a log. This class is described as follow:

- **start_coordinates**: The coordinates of the starting point on the bike.
- **end_coordinates**: The coordinates of the ending point on a bike.
- **user**: The user during the trip.
- **bicycle**: The bike used during the trip.
- **time**: The duration of the trip.

We can use the attribute *time* since we have to instantiate a trip only when it is finished.

2.5 Coordinates

We also decided to create a specific class for coordinates since it allows to manipulate space and distances more easily with it. The UML diagram is sufficiently explicit to understand the structure of this class.
Something noticeable is that since coordinates are objects, we created a new method $equals()$ to compare two points: this method retrieves the x and y coordinates from another point and compare them with the coordinates of the first point.

3 Main characteristics

3.1 Package organisation

We divided our classes into different packages in order to clarify the interactions between them:

clui: This package contains all the commands that are necessary for the CLUI to facilitate the interaction with the MyVelib system. It also includes command "help" indicating all the available commands.

junit_tests: This package contains all the JUnit tests, mainly for core classes and some for clui classes.

test: This package contains some tests different and more complex that JUnit ones. They were helpful when coding classes and looking at their interactions by creating more complex situations.

main: This package contains the main classes as described in Figure 1. It does not contain ParkingSlots since they're involved strongly in the log observer pattern that we implemented in the next package.

balance_update_from_log: This package contains classes involved in the balance observer pattern (see Figure 5. It also includes the class UserBalance containing statistics for each user.

log_update_from_parking_slots: This package contains classes involved in the log observer pattern (see Figure 6

planning: This package is made only for the class TripPlanningMethods designed for calculating the best trip regarding the choices of the user. All the bonuses methods for planning rides are also here.

pricing: This package contains classes which are necessary for pricing a trip once it is finished. We used an interface in order to create more pricing strategies easily (see Section 5.2).

renting: This package contains the class OnGoingTrip representing a trip while it is not finished and the class LogOnGoingTrips for keeping them in a log. An OnGoingTrip is created and stored in the log when a user starts a trip.

sorting: This package contains classes allowing to sort stations. In order to respect the open-close principle, each sorting algorithm corresponds to one distinct class.

terminals: This package contains terminals allowing to simulate the interaction between a user and the model. These terminals deferred from the CLUI since CLUI simulates an administrator choosing what is happening in the model while each of the terminals in this

package simulate a situation happening for a user in our model like renting a bike or planning a ride.

3.1.1 TripPlanning

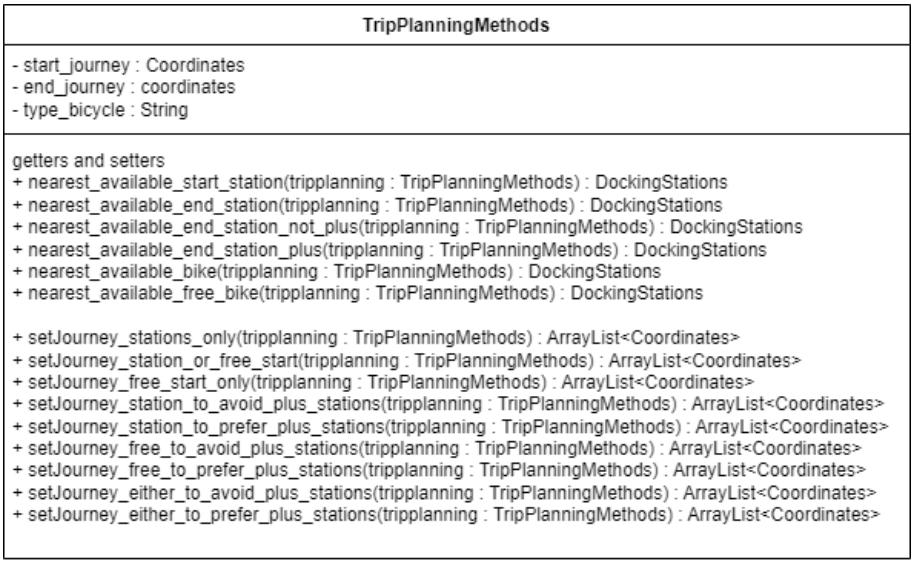

Figure 2: UML diagram for the class TripPlanningMethods

We have implemented optional ride-planning policies that can be utilized in our myVelib design. These policies are not mandatory but provide additional flexibility to the users. The two optional ride-planning policies are as follows:

- **Avoid "plus" stations**: This policy aims to minimize the walking distance for users. However, in this case, the return station cannot be a "plus" station.
- **Prefer "plus" stations**: With this policy, the return station should be a "plus" station. If there exists a "plus" station within a distance no further than 10% of the closest station to the destination location, it is considered for the return. If no such "plus" station exists, this policy behaves normally and functions as a minimal walking distance policy.
- **Prefer a bike parked in the street**: With this policy, the user wants to rent a bike parked in the street and not in a station because it is less expansive. If not such bike exists the trip returned is null and the terminal returns "No such trip is possible".

These optional ride-planning policies enhance the user experience by providing them with the flexibility to customize their ride preferences based on their individual needs and preferences.

3.1.2 Ongoing trips

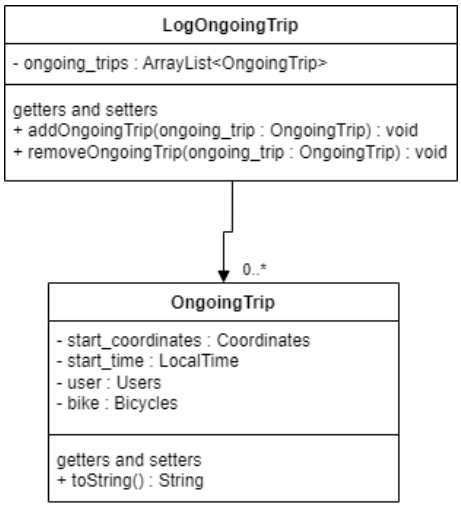

Figure 3: UML diagram for the package *renting*

We implemented this package because in our structure, it was needed to differentiate finished trips from ongoing trips. Thus, OngoinTrip class doesn't have *end_coordinates* attribute and *time* is replaced by *start_time*. Every OngoingTrip is stored in LogOngoingTrip when created. When the user finishes his trip, we get his OngoinTrip by looking at his bike ID in LogOngoingtrip and then transform it into a Trip (by setting *end_ccordinates* with its current location and *time* by calculating the difference between current time and *start_time*) while adding it to LogFinishedTrips. Then thanks to the balance observer pattern (Figure 5), it updates the balance of the station (if parked in a station) and the balance of the user during the trip.

3.2 Terminals

We kept in mind that we were creating a system that had to communicate with users, that is why, in addition to the command line interface requested, we implemented terminals to dialogue with the user to recreate the renting operations and trip planning operations.

These terminals are inside the "terminals" package.

In each of these terminals, every time the terminal asks something to the user, the user has to enter something in the console. The code is robust : it does not crash when the user enters something not expected. The terminals throws an error and asks the user to enter something special. Example in the next figure :

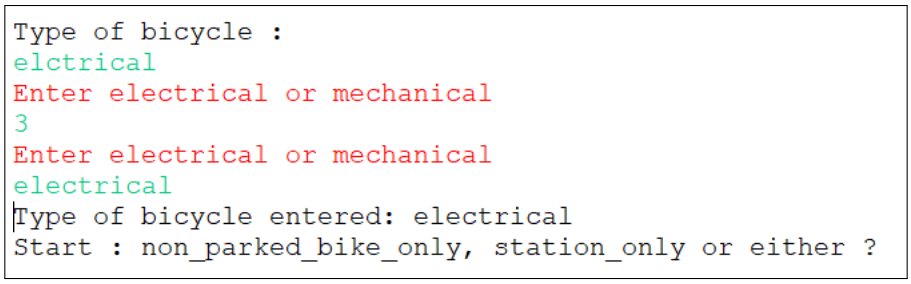

Figure 4: Error when entering the type of bike

3.2.1 TripPlanningTerminal

This file allows to plan a trip. It asks the user to enter the start coordinates of the user's journey , the end coordinates of the user's journey, the type of bike they want, the type of start they want (from a station : **"station_only"** option, from a bike parked in the streets : **"non_parked_bike_only"** option, or either : "either" option) and finally the type of station they want at the end (if they want to avoid plus stations : **"avoid_plus_stations"** option, if they prefer plus stations : **"prefer_plus_stations"** option, or either : **"either"** option).

3.2.2 TripPlanningClient

This class is only used to create a method used in the Client class explained below. This method allows to ask the user if they want to plan a trip or not. The terminal continues to ask the user if they want to plan a trip until the user responds "yes" or "no". The planned trip is then returned if wanted by the user.

3.2.3 RentingTerminalMethods

This class allows to create all the necessary methods for the renting terminal.

User_identification : Asks the user to identify themselves (by credit-card or Vlib-card)

Renting_allowed : Verifies if the user is not already renting a bike, if so, the terminal throws the error **"Cannot rent more than one bike at a time"**)

Renting : Asks the user to chose their bike (identifying it by the bike's id), and updates the status of the parking slot if the bike was in a parking slot. We assumed that the user would use the terminal when in front of a bike to rent (either at a station or in the streets). This method also updates the LogOngoingTrips to be able to retrieve the important information when the user returns the bike.

Returning : It allows the user to return a bike.
If the user is at a station (their coordinates match) the method searches for a free slot. If it is found, the statuses of the bike and the slot are updated, the ongoing trip is removed from the log of ongoing trips and the final trip is added to the log of finished trips so that

the station balance and the user balance are updated.
If the user is not at a station the terminal asks the user if they want to stop the trip over and over until the user says "yes". When they do, their coordinates are entered as end coordinates of the trip and the different logs are updated.
The fact that the terminal asks the same question over and over again is a choice we made because a real application would too, with a button "stop".

Terminal : This method simulates the terminal entirely, using the methods defined previously, and the start coordinates, the end coordinates, and the duration of the ride.
Because we do not have access to the coordinates of the user in real time, we had to ask the user for the end coordinates of their trip during the renting, which is not really realistic but it is the choice we made to simplify the terminal.
Moreover, because we do not want to wait for an hour to simulate a real trip, we allow the user to enter a duration of the ride. If the user enters 0, the time of the machine will be used (just like the time of the beginning of the trip is marked as the machine time at the time of the rent).

3.2.4 RentingTerminal

This terminal asks the user if they want to rent a bike.

If the user says "yes", the terminal calls the "terminal" method of "RentingTerminal-Methods". If the user says "no", the program stops. Finally if the user enters anything else, the terminal asks to enter yer or no.

3.2.5 Client

The terminal client allows to use the trip planning and the renting terminal, just like an application for the user.

It uses the TripPlanningClient so it asks the user if they want to plan a trip. Then it asks the user if they want to keep the planned trip so that they do not have to enter the start and end coordinates again. If the user did not plan a trip they only have to enter "no". Then it asks the user for the duration of the wanted ride (entering 0 uses the time machine at the time of returning). If the user did not plan a trip or they did not want to do the planned trip, the RentingTerminal is called.

4 Design decisions

We decided that creating different MyVelib systems at a time to represent different cities was not necessary in this project so we set all the attributes of MyVelib to static. To be able to create different MyVelib we could have not set the attributes of MyVelib to static and created a class to store the list of all the MyVelib systems.

In the command line interface we chose to match what was expected as commands like for example "setup myvelib". As a result the order of the arguments for a command matter and the system will not catch the problem if the arguments are not in the right order.

In addition to our project's objectives, we aimed to implement an open-close approach to enhance its flexibility and expandability. To achieve this, we devised a comprehensive framework called the "sorting" package, which simplified the process of creating new sorting algorithms. By employing this package, the system administrator only needs to create a new class for each unique algorithm, thus streamlining the development process.
Furthermore, this open-close approach was also evident in our "pricing" package. To facilitate the addition of new pricing strategies, we leveraged the strategy pattern (refer to Figure 5.2). By utilizing the "PricingStrategy" interface, integrating a new pricing strategy merely requires the creation of a new class dedicated to the strategy and the addition of a single line of code within the "Pricing" class. This design choice allows for seamless expansion and modification of pricing strategies, empowering administrators to adapt to evolving business needs with minimal effort.

5 Design patterns

5.1 Observer patterns

The Observer pattern in Java establishes a way for objects to be notified automatically of any changes in the state of another object. It allows multiple "observers" to subscribe and receive updates when the "subject" object changes, promoting loose coupling and flexibility in the code. We decided to implement this pattern twice in our model.

5.1.1 Balance observer pattern

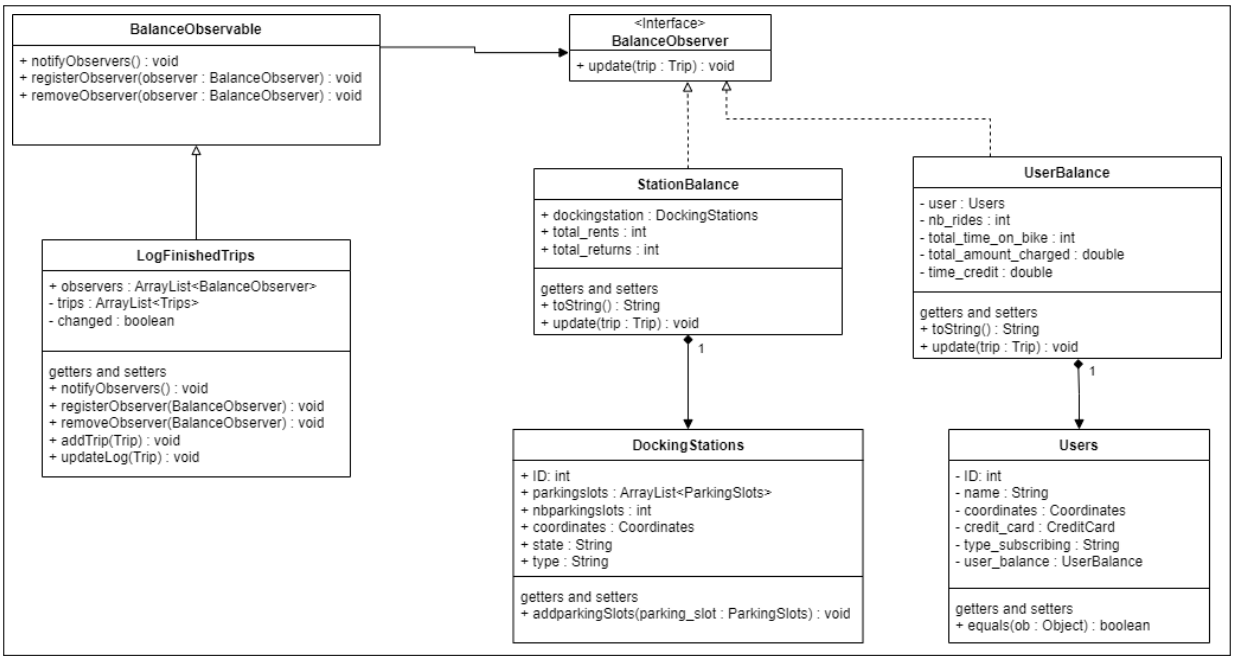

Figure 5: Balance observer pattern

We implemented the balance observer pattern to ensure the accurate updating of user balances and station balances when a new trip is added to the log of finished trips. This pattern involves the use of observers and an observable component. The UserBalance and StationBalance classes serve as the observers in this pattern. They are responsible for monitoring and reflecting changes in the balances of individual users and stations, respectively. Whenever a new trip is added to the log of finished trips, these observer classes are notified and take appropriate actions to update the relevant balances.

The LogFinishedTrips class acts as the observable component in the balance observer pattern. It maintains a log that records all completed trips. When a new trip is added to this log, the LogFinishedTrips class notifies the observers, i.e., the UserBalance and StationBalance classes, about the change.

Upon receiving the notification, the UserBalance observer updates the balance of the corresponding user based on the details of the newly finished trip. It ensures that the user's balance accurately reflects the completed trip and any associated charges or deductions. Similarly, the StationBalance observer updates the balance of the relevant station. It takes into account factors such as the number of trips completed at the station and any fees or revenue associated with those trips. This ensures that the station's balance is updated accurately to reflect the latest state.

5.1.2 Log observer pattern

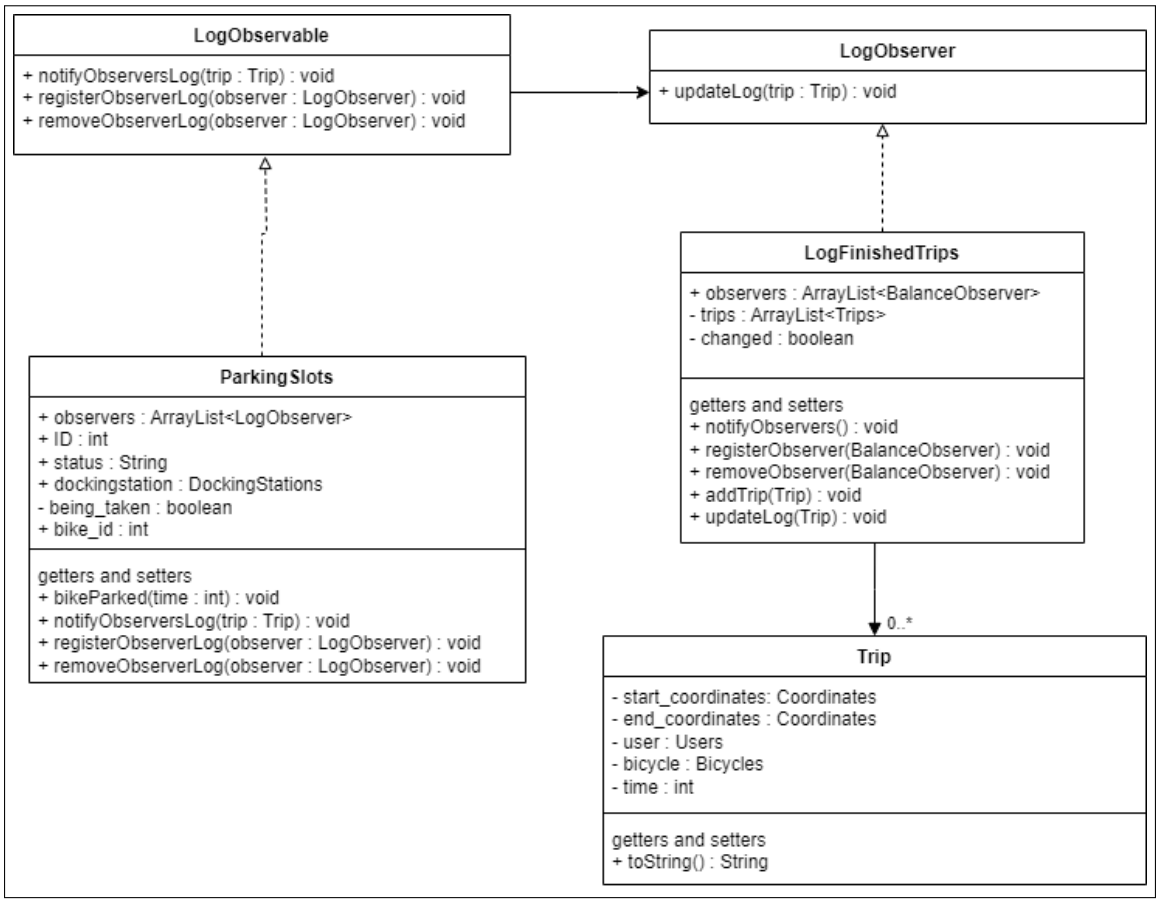

Figure 6: Log observer pattern

We implemented the log observer pattern to handle updates to the log of finished trips when a bike is parked in a station. This pattern involves the use of an observable component, which in this case was the parking slots, and the LogFinishedTrips class as the observer.

The idea behind this pattern is that whenever a bike was parked in a station, the corresponding parking slot will act as the observable and notify the LogFinishedTrips observer about the update. The LogFinishedTrips class will then handle adding the relevant trip details to the log of finished trips.

We implemented this observer pattern early on in the project, before developing the terminal methods and the command-line interface for renting and returning bikes. Our intention was to simulate a real-world scenario where a slot being taken would automatically trigger an update in the log. However, as we progressed further in the project, we realized that this observer pattern was not necessary and could have been avoided. We came to the understanding that we could simply use the existing addTrip method within the returning

methods of the bikes to directly add the trip to the log without the need for the observer pattern.

5.2 Strategy pattern

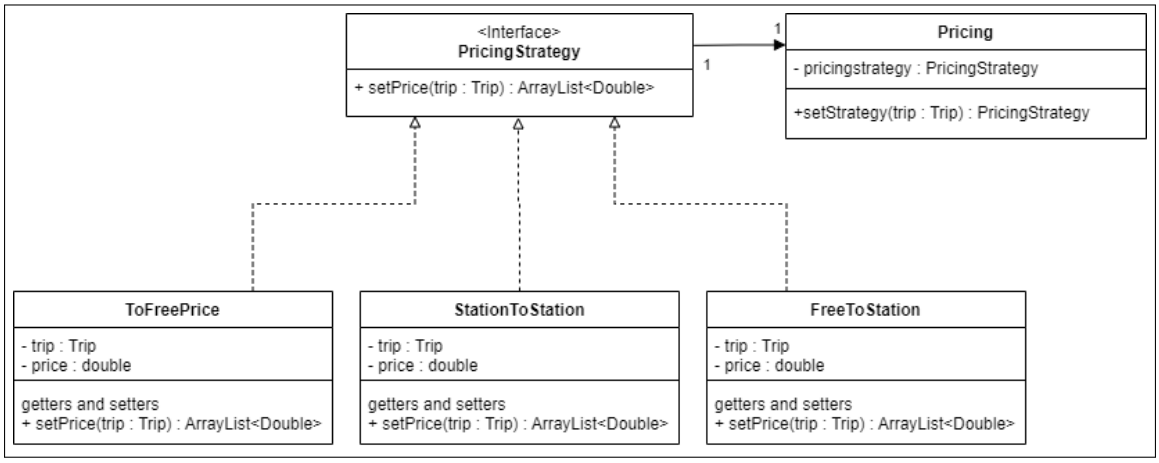

Figure 7: Pricing using strategy pattern

We used a strategy pattern for the pricing of different trips. There are 3 strategies for the pricing : from station to station (StationToStationPrice), from a bike parked in the street or a station to a street (ToFreePrice) or from a bike parked in the street to a station (FreeToStationPrice). The only method in the interface PricingStrategy is the setPrice method. In the class Pricing, the method setStrategy looks at the start and end coordinates of the trip entered in argument and searches if the coordinates match those of a station in the MyVelib system. The method then sets the right pricing strategy.

6 Commands added compared to the required ones

6.1 Help command : help

This command displays all the different commands possible and clarify possible doubts about the arguments. Its Javadoc files uses HTML to make it the clearer as possible.

6.2 SetUp command that adds users : setup <name> <nstations> <nslots> <s> <nbikes> <nusers>

This commands allows to set up a myvelib with users without having to ass them manually. The users have random names from a pre-defined list, a unique id and random types of subscribing

6.3 Plus command : plus <velibnetworkName> <stationID>

The plus command allows the user to set a station to plus.

6.4 Standard command : standard <velibnetworkName> <stationID>

The standard command allows the user to set a station to standard.

6.5 Enter_manually command : enter_manually

This command allows to enter manually the commands and test the commands without using the testScenario files.

6.6 TripPlanningClui command : tripplanningclui <velibnetworkName> <startcoordinates> <endcoordinates> <typeofbike> <starttype> <strategyforpluststations>

This command allows to plan a trip and displays the planned trip. It implements the different options to pick a bike from a station or from the street, to avoid or to prefer plus stations.

7 Tests

7.1 Test scenarii for the command line interface

In order to see the different scenarii, please:

1. Launch Clui.java : from fr.cs.group17.myVelib.bin enter java clui.Clui
2. Enter as an argument **runtest testScenarioX.txt** with X between 1 and 3

The produced scenario will appear in the file "testScenarioXoutput.txt"

Moreover if you decide to enter commands manually, please:

1. Launch Clui.java : from fr.cs.group17.myVelib.bin enter java clui.Clui
2. Enter as an argument **enter_manually**
3. Enter your commands line by line as specified by the command **help**
4. Enter **stop** to end the interaction

It is normal that the response of entered commands do not appear in the terminal. The output of each command is written in the file "manualoutput.txt".

7.1.1 Test scenario 1

The first scenario demonstrates the handling of errors.
The first 11 lines of instructions are returning errors: several commands are returning errors due to non-existing entities or incorrect parameters. The MyVelib system reported that there was no such type of subscribing ("no"), no MyVelib network with the specified name ("velib" and "myvelib"), no station with certain IDs (7, 42, 99, 1, and 38), and no user with specific IDs (1 and 2). Stations were not able to be set as offline or online because of their non-existence or offline status.
After that it manages to add the user Patrick and shows some errors about renting and returning a bike. It finally displays information about the state of the system and the commands available using the CLUI through the command "help"

7.1.2 Test scenario 2

The scenario demonstrates the setup and usage of the MyVelib system, including creating stations, renting and returning bikes, managing user balances, and displaying system information. It specifically shows the added command of the setup with users.
Firstly we set up the MyVelib system with the following parameters:

- **Name**: myvelib
- **Number of users**: 10
- **Number of stations**: 10
- **Number of parking slots per station**: 10
- **Number of bikes**: 70
- **Size of the side of the area**: 4

We used the "Online" command and the added "Plus" to make sure the stations were online and to show that the time credit is added properly. Then we show the rents, there might be errors because no slot is available, that is normal since the setup is random, the user just has to run the test scenario again.
Finally we display stations information, users information and then the whole myvelib system details. There is also the help command to show the other commands for the CLUI.

7.1.3 Test scenario 3

The third scenario demonstrates how using the planning system works within a MyVelib model: the user's interaction with the trip planning CLUI involves specifying start and end stations, preferences for electric bikes and plus stations, and constraints regarding bike availability. The system responds by providing information about free bike coordinates and the suggested end station based on the given inputs.
The first command indicates that the user wants to plan a trip using an electric bike starting

from a station and ending in another station specifying the starting coordinates of the user. It returns that the user should start at station 1 and end in the same station, hence no trip for this user.
The second command indicates that the user at the same location wants to plan a trip using an electrical bike starting anywhere and ending preferably in a plus station. It returns that the user should start with a bike from the street (giving its coordinates) and end at station1.
The third command indicates the same thing but starting with a bike from the street. Logically it returns the same trip than the previous situation since it was already giving a bike parked in the street.

7.2 Test scenario for the terminals

The terminals are unable to read commands from a file, but we provided examples of commands in the Annex (those are the same as the ones in the testTerminals.txt file in our project folder) to help gain some time. In this file there are different scenarii, they are only examples of what is possible as a lot of combination are possible.

7.3 Junit tests

There is a total of 79 JUnit tests:

- 62 tests for core classes
- 17 tests for clui classes

They were useful during the coding of our classes since we used them in two different ways. The first way corresponds to classes where we applied Test Driven Development , which means that we finished the tests before the class. For example it was the case for the class Coordinates since its methods and attributes were really predictable. Nonetheless we followed this method only at the beginning of the project since there was too many structures changes and dependencies later on, making this kind of development too difficult to continue. Most of the time, it was useful for verifying that class that we made were working well in our model. For instance it was thanks to tests that we figured out that comparing users necessitates to create a new method $equals()$ since they are objects.

7.4 Situations when the test allowed to correct a bug

All along the project, in addition to the Junit Tests, we used test classes with little scenarios to test the basic functions of the program. Some of these tests can be seen in the "test" package (we deleted the tests along the way so there are only the most recent ones, that test the entire patterns for example). We chose to leave them in the project to show their importance as every one of them allowed us to understand some errors.

8 How to test

8.1 Command line interface

After opening the project, running the Clui main method (from fr.cs.group17.myVelib.bin enter java clui.Clui) will initialize the MyVelib system with the name myvelib thanks to the my_velib.ini file. The initial MyVelib has : 10 stations, 10 parkingslots per stations, 70 bikes, no users, and is a square of side size 4km.

Then if the user wants a different MyVelib, they have to use the command line interface and use the commands setup with either :

- only the name wanted

- or the name, the number of stations, number of slots per stations, the size of the square, and the number of bikes

- or the name, the number of stations, number of slots per stations, the size of the square, the number of bikes and the number of users

To use the command line interface, one has to call the Clui main method with as arguments either :

runtest testScenario1.txt (the .txt is important because we use the argument "testScenario1.txt" to call the file and write the output file) to run the test scenario 1 and write the outputs in the file testScenario1output.txt, not in the terminal. The same goes for the other test scenarios.

or **enter_manually** to be able to use manually the commands and test the functionalities without using the test files. The outputs are entered in the manualoutputs file and are not written in the terminal, just like the test sceanrios.

If no arguments are entered when calling the Clui, only the initialisation will be done and the program returns : "Enter 'runtest' followed by the name of the file, 'enter_manually' or 'stop'" and the program will loop until the user enters one of these 3 keys.

Help command : At any time, the user can use the help command (either as an argument when calling the Clui.java file, when entering the commands manually or inside a file) to see how to write the different commands and arguments.

8.2 Terminals

To test the terminals the user can run the Client file in the package "Terminals" and copy paste the commands in Annex (those are the same as the ones in the testTerminals.txt inside the project folder) or enter them manually when the terminal asks so. These commands are only examples as there are a lot of different cases to test (different type of bikes, different coordinates, different startegies of parking, different strategies towards plus stations etc).

9 Error detection

As explained in the beginning of the terminals part in the main characteristics, our code is robust : it will search for the right type of command and throw an error asking the user to enter the right thing, it will not crash.

The only case we encountered of the program crashing that we did not fix was when the user does not set up the MyVelib system before entering commands in the Clui or the terminals. We did not fix it because normally the user will initialize correctly the MyVelib system by lauching the Clui before doing anything else.

10 Limitations

- One cannot create different MyVelib systems as explained in the design choices part. If the user uses the MyVelib setup functions several times the myvelib will be reset but the unique identification numbers will not. In the second scenario for example, because the MyVelib is set 2 times (at initialisation and in the testScenario2.txt file), the ids start from 10 and go up to 20 instead of from 1 to 10.

- The order of the arguments for a command matter and the system will not catch the problem if the arguments are not in the right order.

- In the terminals we have the warnings about the scanner objects not being closed. That is because we use terminals as methods and if we close the scanners inside the methods then we cannot use another terminal.

11 Workload split

	code	Junit tests
Main classes	Lucas	Séléna
Balance observer patterns	Lucas	Séléna
Strategy pattern	Séléna	Lucas
Trip planning	Séléna	Lucas
Terminals	Séléna	Lucas
Clui	Séléna	Lucas

Table 1: Workload split

We split the work regarding our interests and availability. During the whole project, the one who coded was not the one to test. We did not use a git, we simply sent the files to each other using teams.

We implemented the classes and methods gradually. Indeed thanks to the UML diagram we knew which patterns or methods would use which. We started with the set up and main classes, then the pricing strategy pattern, then the balance observer pattern that uses the pricing, then the log of ongoing trips and the log observer pattern that completes the balance observer, then the terminals. The trip planning could be done separately as well as the sorting strategies. We then implemented the command line interface.

We imagined the model's structure together. Looking at the aftermath of this project, Séléna wrote more classes while Lucas did more tests and docs. The report was mainly written by Lucas but each of us wrote about what we did so Séléna wrote about the terminals, error detection etc

12 Perspectives

What would be interesting would be to be able to create several MyVelib systems to represent different towns for example.

To further improve the terminals we could also make it possible for them to read instructions from a file, it would simplify their use.

Also integrating a database into the system could optimize data storage and management, especially for a large system like this and even more if we plan to deploy it into different cities.

13 Conclusion

This extensive project has provided us various challenges that tested our skills and knowledge. One crucial realization we made was the significance of planning in the development of different classes and designing the system using UML diagrams while embarking on the coding phase. It also taught us to write proper code since it was even more important while making the clui classes.

Throughout the project, we encountered specific challenges that required us to revise a part of the code architecture multiple times like making pricing strategies and trip planning package. These experiences deepened our understanding of these concepts and emphasized the importance of adaptability when dealing with intricate project requirements.

This project has served as a platform for us to consolidate our technical knowledge, acquire new skills, and sharpen our ability to tackle complex problems as a team.

14 Annex : example of commands to test the terminals

// example of commands to plan a trip, use this trip and rent a bike

yes

1

2

7

8

electrical

station_only

either

yes

90

Vlib-card

3

4

// example of commands to just rent a bike

no

no

1

2

6

8

90

credit-card

2

7

no

yes