MyVelib Application – Java Project

Oussama El M'tili

June 2020

Contents

1 Introduction

I decided to work alone in this project. Firstly we will see the design and implementation of the core. Then the design and implementation of the interface using mainly the factory pattern. Everything is done trying to respect at most the open close principle. The interaction between the core and the interface responds to the **MVC** pattern.

For the rest of the document, VL user, VM user and Credit user will respectively refer to user with a Vlibre card, user with a Vmax card and user with no card. Operation refers to a rent or return.

2 Core

2.1 Strategy Pattern

The pricing differs for each type of user. Furthermore the pricing can differ throughout time. Some updates on the algorithm can happen. The good design for this is the **Strategy pattern**. That means we have to create 3 subclass (one for each type of user) and create a pricing interface with the method *price(int minutes)*.

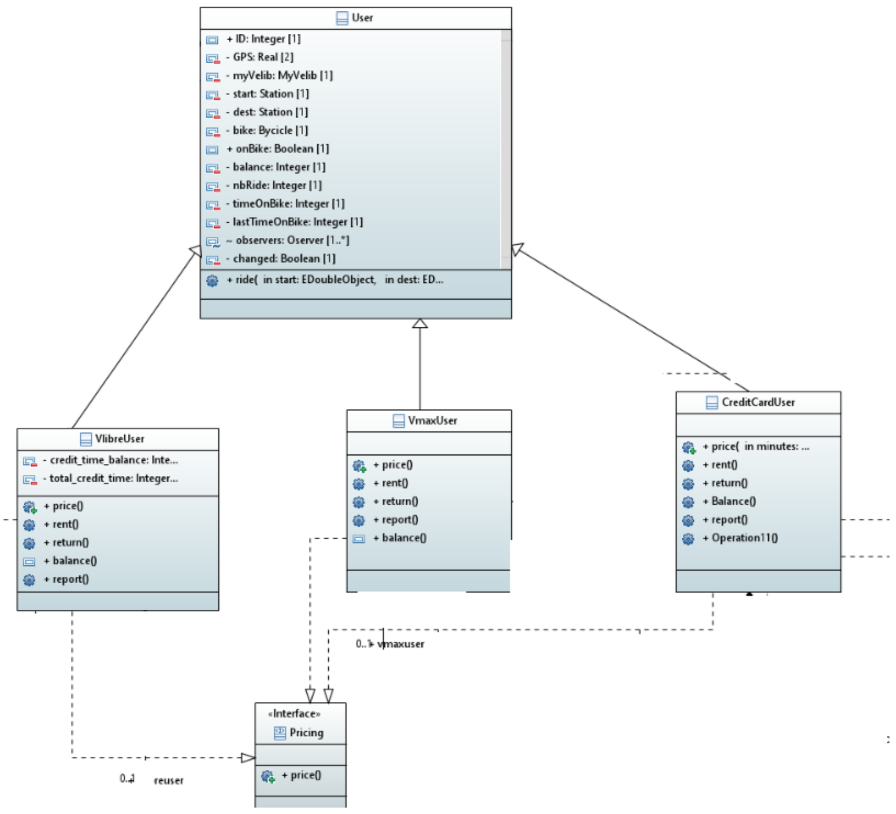


Figure 1: Strategy Pattern

2.2 Observed Pattern

Each time an operation is done, all the velib network should be notify and should update. The perfect design pattern for that is the Observed pattern. So we need to create a **MyVelib** class which will represent a single velib network. **This will be the Observable**. MyVelib will then have an **update** method. The thing is we want to update several message depending on type of user. For instance, we want to print the credit time balance of a VL user. In order to respect the open close principle, we should not modify the **update** method if there is a (imagine) a new type of card. Thus for the moment there needs to be 3 **update** methods (one for each user). So, the VL users, VM users and Credit users will be the **Observers**.

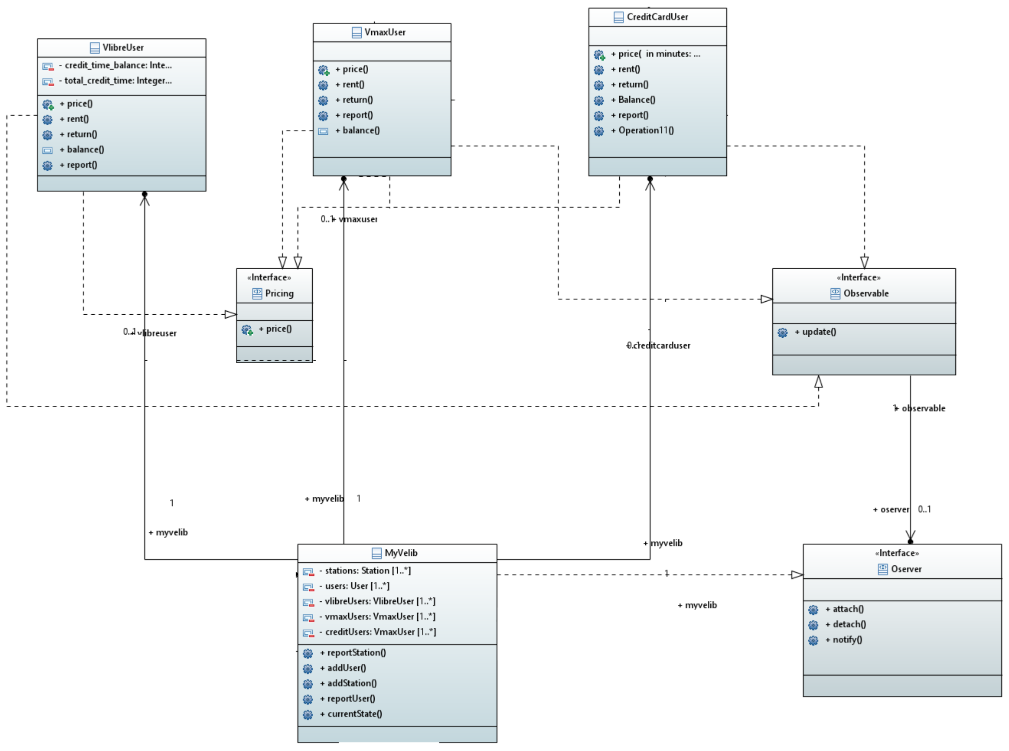


Figure 2: Observed Pattern

2.3 Classes

For the moment, **MyVelib** class contains **users, VL users, VM users, Credit users and stations**. The bikes are included in Station class. But Let's not forget the interface which shall have the ability to create several velib network. Hence we need to create a class **MyVelibCreation** that create and store **MyVelib** objects. Finally here are the several class of the core: Bicycle, Station, User, VlibreUser, VmaxUser, CreditCardUser, MyVelib, MyVelibCreation.

2.4 Main methods

There are rent, return, ride and statistics methods.

2.4.1 rent, return, ride

Those are performed by users. Following the Observed pattern we made, we cannot implement rent and return directly in the **User** class. Indeed those methods need to use the **notify()** method which does not exist for the User class. So we need to put them in the 3 subclass. We see here a **limit to our strategy** : we are typing the exact same code in three different class. However it still respects the open close principle.
The **ride** method does not need to be notified, so we can put it directly in the **User** class.

2.4.2 Statistics

The statistic are applied to station and users and are the same (by their calling). A great way to implements these methods is using a **Statistic interface**. Again the statistic differs regarding the type of user. So in order to respect the open close principle and not modifying the balance/report methods if (imagine) a fourth card appears; each user subclass should implement the **Statistic interface**.

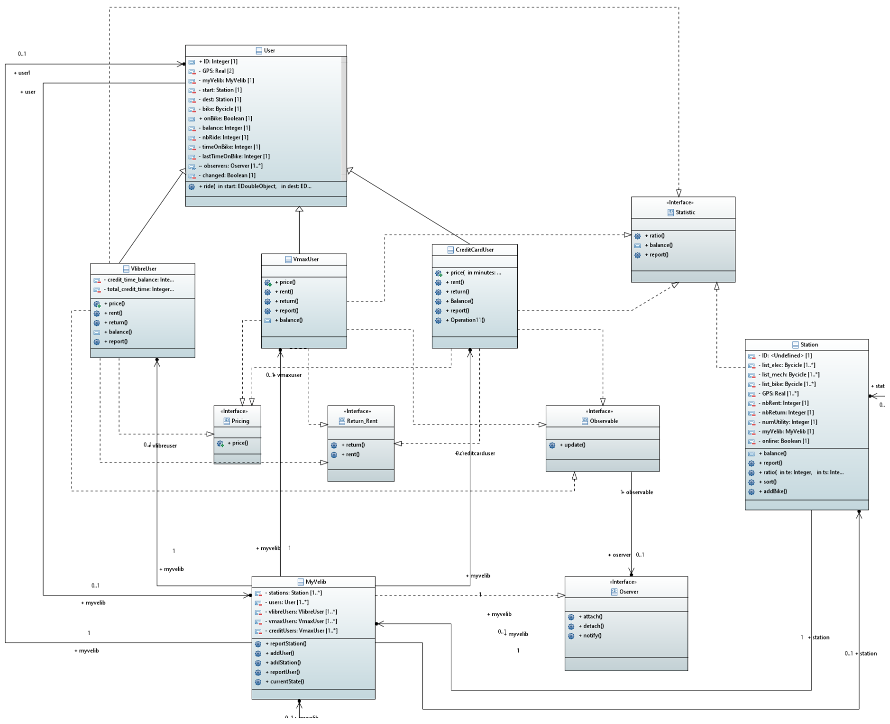


Figure 3: Methods implementation

2.5 Handling time

Handling time is crucial since the rent and return methods are based on it, as of the occupation rate. The thing is we cannot import a clock since we are not in interactive mode at this point. Let's take a look at the occupation rate of a given station.

$$r(t_e, t_s) = \frac{\sum_{i=1}^{N} t_i^{\Delta}}{N\Delta}$$

Then Let call

- $t_1^{rent} < \quad ... \quad < t_{r_\Delta}^{rent} \quad time \;\; of \;\; rent \;\; during \;\; [t_e, t_s]$
- $t_1^{Return} < \quad ... \quad < t_{R_\Delta}^{Return} \quad time \;\; of \;\; return \;\; during \;\; [t_e, t_s]$
- $N_{t_e} = N_{t=0} + (number\, of\, return\, before\, t_e) - (number\, of\, rent\, before\, t_e) \quad number\, of\, bikes\, available\, at\, t_e$

We have then

$$r(t_e, t_s) = \max\left(\frac{N_{t_e} - (r_\Delta + R_\Delta)}{N}, 0\right) + \frac{(\sum_{i=1}^{r_\Delta} t_i^{rent}) - r_\Delta t_e}{N\Delta} + \frac{R_\Delta t_s - (\sum_{i=1}^{r_\Delta} t_i^{Return})}{N\Delta}$$

So we only need to create 2 attributes *listTimeRent, listTimeReturn* for Station Class and update those attributes each time *rent* and *return* methods are called. We will express the time in minutes. Each velib network has its own time. $t = 0$ is when a velib network is created.

Sorting Strategy

There are 3 sorting strategies:

- USED : number of time the station has been used.
- DISTANCE : regarding a station s_0, how far are the other station from s_0. (We use euclidian distance). This is only for rides planning.
- OCCUPIED : Global Average Occupation.

Concerning the last one, what I have done is getting the last time the velib network had been used t_{max}. Then I sort all the station regarding their occupation rate in $[0, t_{max}]$. Nevertheless, for this to work, I need at least one operation (rent or return) in the velib network. So if this cannot be the case, we will just compare the stations regarding their initial number of bikes.

You will find all the used **comparators** in the package **comparator**.

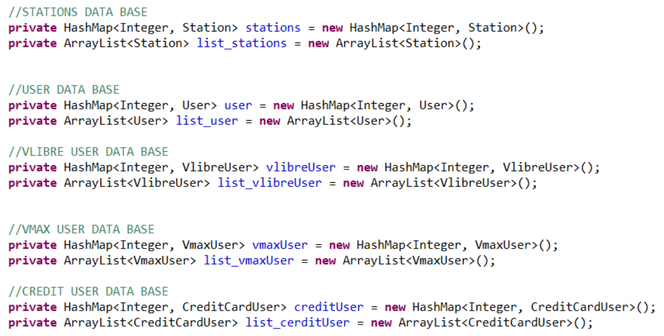
2.6 Handling Errors

We handled errors throwing exception to our methods. You can find in the javadoc all the Exception added to the project. However, here is a small description of the one used in core package.

- The user cannot rent a bike if he is already renting one.
- The user should choose a valid type of bike (electric or mechanic).
- The user cannot return a bike to the wanted station if this one is full.
- The user cannot rent an electric bike from the wanted station if this one has no electric bicycle.
- The user cannot rent an mechanic bike from the wanted station if this one has no electric bicycle.
- The user cannot rent a bike at time lesser than the last time he returned a bike. The user cannot return a bike at time lesser than the last time he rent a bike.
- The user cannot return a bike if he is not already renting one.
- The user cannot perform operations from an offline station.
- The type of card must be VLIBRE, VMAX or NONE.
- The policy for sorting station must be USER or OCCUPIED.

2.7 Handling Data

To make a link between a user and its ID, a station and its ID, a velib network and its name, a user and its name we use **Hash maps**.



```java
//STATIONS DATA BASE
private HashMap<Integer, Station> stations = new HashMap<Integer, Station>();
private ArrayList<Station> list_stations = new ArrayList<Station>();


//USER DATA BASE
private HashMap<Integer, User> user = new HashMap<Integer, User>();
private ArrayList<User> list_user = new ArrayList<User>();

//VLIBRE USER DATA BASE
private HashMap<Integer, VlibreUser> vlibreUser = new HashMap<Integer, VlibreUser>();
private ArrayList<VlibreUser> list_vlibreUser = new ArrayList<VlibreUser>();


//VMAX USER DATA BASE
private HashMap<Integer, VmaxUser> vmaxUser = new HashMap<Integer, VmaxUser>();
private ArrayList<VmaxUser> list_vmaxUser = new ArrayList<VmaxUser>();

//CREDIT USER DATA BASE
private HashMap<Integer, CreditCardUser> creditUser = new HashMap<Integer, CreditCardUser>();
private ArrayList<CreditCardUser> list_cerditUser = new ArrayList<CreditCardUser>();
```

Figure 4: Data Base for **MyVelib** Class

3 User Interface

3.1 Factory Pattern

I want to separate the process of user typing his command to the one of creating new object. This is the problem faced by the **factory pattern**. For the moment we thus need 2 class : **MyVelibApplication** class with the **main** method and the **CommandLine** class that translates the input the user typed into a command. However, for this translation to be I need to translate all the methods from **MyVelibCreation** and adapt them to have in input a array of string. I have done that through the **Command class**. The **drawback** of this design is that we have to write almost twice the same thing.
Moreover, I do not exactly follow the pattern we have seen in lecture since i don't use "product interface".
Let see an example to show how this design almost respect the open-close principle. Imagine we want to create a new command. I will then create a new method in **MyVelibCreation** class. Then I'll need to adapt it in the **Command** class. So for now I didn't modify a single thing. However my **CommandLine** class only contains one method to handle all the user input. So i will need to modify this method. Here is the **limit**. However this would be the only modification.

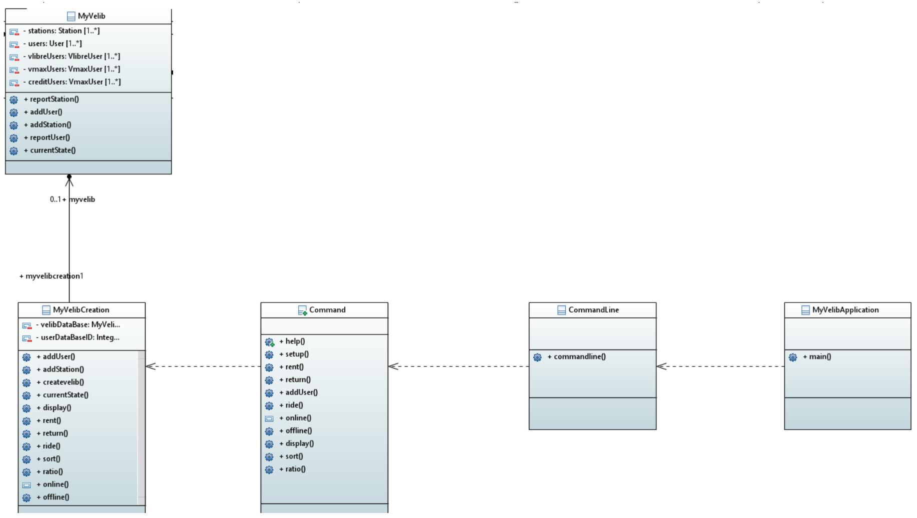
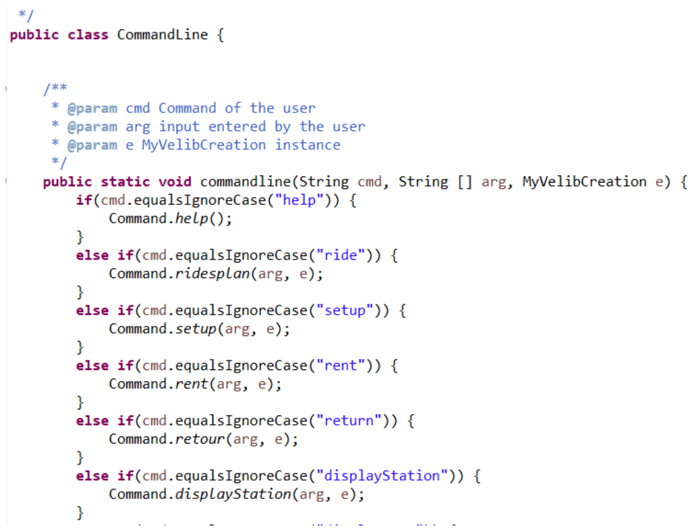


Figure 5: Design of the User Interface



```
 */
public class CommandLine {


    /**
     * @param cmd Command of the user
     * @param arg input entered by the user
     * @param e MyVelibCreation instance
     */
    public static void commandline(String cmd, String [] arg, MyVelibCreation e) {
        if(cmd.equalsIgnoreCase("help")) {
            Command.help();
        }
        else if(cmd.equalsIgnoreCase("ride")) {
            Command.ridesplan(arg, e);
        }
        else if(cmd.equalsIgnoreCase("setup")) {
            Command.setup(arg, e);
        }
        else if(cmd.equalsIgnoreCase("rent")) {
            Command.rent(arg, e);
        }
        else if(cmd.equalsIgnoreCase("return")) {
            Command.retour(arg, e);
        }
        else if(cmd.equalsIgnoreCase("displayStation")) {
            Command.displayStation(arg, e);
        }
```

Figure 6: Extract from the CommanLine class

3.2 Handling Errors

The only errors of the user interface are typing errors.

- Valid velib network name.
- Valid user ID.
- Valid Station ID.
- Valid Command.

Again, you can find all the Exception in the exception package.

4 Result

4.1 MVC Pattern

The project follows the **MVC pattern.** which is great for separating design concerns in GUI development. The **Observable** is **MyVelib** class, the **Observers** are the 3 subclass of user and the controller is **MyVelibApplication**.
And we see that the time our design does not respect the open close principle is due to the loop of the factory design.

4.2 MyVelibApplication

I suggest you to run the application through **Eclipse**.
MyVelibApplication supports 2 modes :

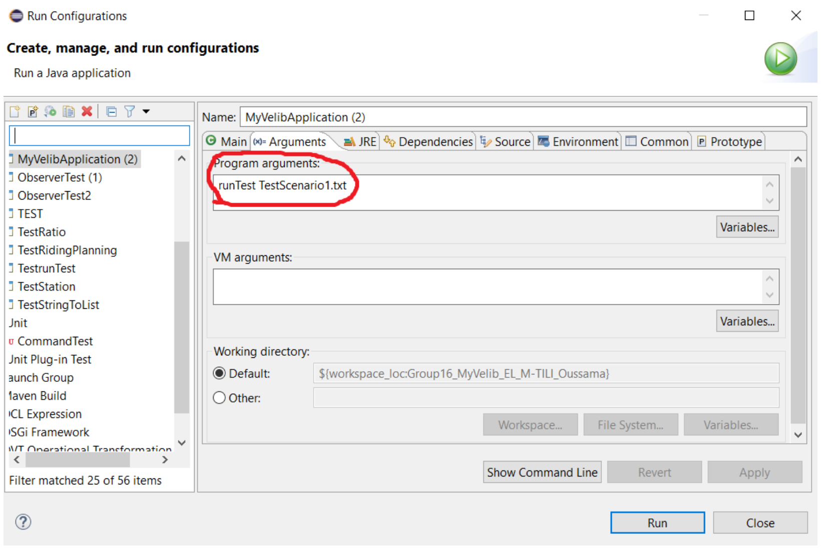
- An Interactive mode.
- A Test mode.

4.2.1 Test mode

To access this mode, open the project in Eclipse and go to:

```
src > userInterface > MyVelibApplication > Run As > Run Configurations > Arguments
```

You should see this window :


Run Configurations

Create, manage, and run configurations

Run a Java application

Name: MyVelibApplication (2)

Program arguments: runTest TestScenario1.txt

VM arguments:

Working directory:
[✓ Default:] ${workspace_loc:Group16_MyVelib_EL_M-TILI_Oussama}
Other:

Filter matched 25 of 56 items

Figure 7: runTest launch

In **arguments** just write

```
runTest <TestScenario file>
```

This will directly print the result in a outputi.txt file.

Description of the different *TestScenario* files

- TestScenario1.txt : run the scenario described on the project instructions.
- TestScenario2.txt : Highlights errors handling of section 2.6.
- TestScenario3.txt : Highlights errors handling of section 3.2.

The **ini** file is well loaded in both mode. The name of the basic velib network it loads is "default".

4.2.2 Interactive mode

If you are not conviced by the test I have made you can go to the interactive mode to type your command yourself.
To do so, follow the previous guideline and type **I** in the Arguments section and clic run.
You can tap **help** to show all the available commands and their descriptions. You are now in a real CLUI.

5 Remark

I suppose you noticed my **UML diagram** is not really well filled. Indeed the methods take in most cases several arguments. This is due to my lack of time (I was alone in this project) and of technique in Papyrus. Nevertheless, I assumed it didn't bother the global design of the application and its understanding.
For a more detailed description you can take a look at the java doc.