

CENTRALESUPÉLEC

OBJECT ORIENTED SOFTWARE ENGINEERING

myVelib - a bike sharing application

Authors:

BAUDINET Charles
LOPES Matheus Elyasha

Group: 10

June 2020



CentraleSupélec

Contents

1	Introduction	2
2	System description and requirements	3
2.a	UML diagrams	5
3	Design decisions	6
3.a	IDE	6
3.b	VCS (Version Control System)	6
3.c	Design patterns that we tried to use during the development	6
3.d	Git flow workflow	7
3.e	Pair programming	8
3.f	Test Driven Development/Design (TDD) and Behavior Driven Development (BDD)	8
3.g	Testing	8
4	Advantages and limitations of the solution	9
4.a	Advantages	9
4.b	Limitations	9
5	Test scenarios	11
5.a	Setting up a myVelib system (or network)	11
5.b	Rental of a bicycle	11
5.c	Simulation of a planning ride	11
5.d	Visualization (user, station and system)	12
5.e	Computation statistics	12
6	How to test our project	14
6.a	Examples of utilization of the myvelib CLI	14
7	Documentation	21
8	Project coverage (testing)	22
9	Conclusion	23

1 Introduction

A bike sharing system (like, for example, Velib in Paris) allows inhabitants to rent bicycles and cycle around a metropolitan area. Such a system consists of several interacting parts including: the renting stations (displaced in key points of a metropolitan area), different kinds of bicycles (mechanical and electrically assisted), the users (which may have a registration card), the maintenance crew (responsible for collecting/replacing broken down bicycles), etc.

During the SG8, we have created a JAVA framework, hereafter called **myVelib**.

The project consists of two parts:

- Part 1: myVelib core: design and development of the core Java infrastructure for the system
- Part 2: myVelib CLI (command-line interface): design and development of a user-interface for the system

2 System description and requirements

The system description was given by a PDF file, with some points that could have been better explained. We feel that sometimes, the desired feature was not exactly useful to the myVelib framework and maybe if we have other arguments in some methods it could have been better designed.

Nevertheless, we followed the document given by the teaching team. And we figured out the following strategy to deal with the project.

First, we created a **README.md** file at GitHub (please, see the VCS section of this document). Then, we listed all the requirements, and we checked each requirement when we finished the following feature.

In this way, we can say that we implemented a kind of agile method to deal with the requirements list and we invite the reader to go to our repository page to see a better and explained version of our implementation, or just read the java documentation that is way better explained (link in the documentation section).

Our solution is given with two main packages:

- core package
- cli package

And here, we chose to use the **CLI** abbreviation, that stands for command-line interface.

The basic structure of the packages can be seen in the figures 2.1, 2.2 and 2.3.

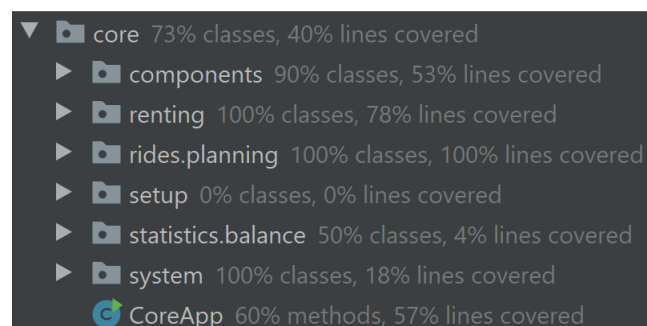


Figure 2.1: core package

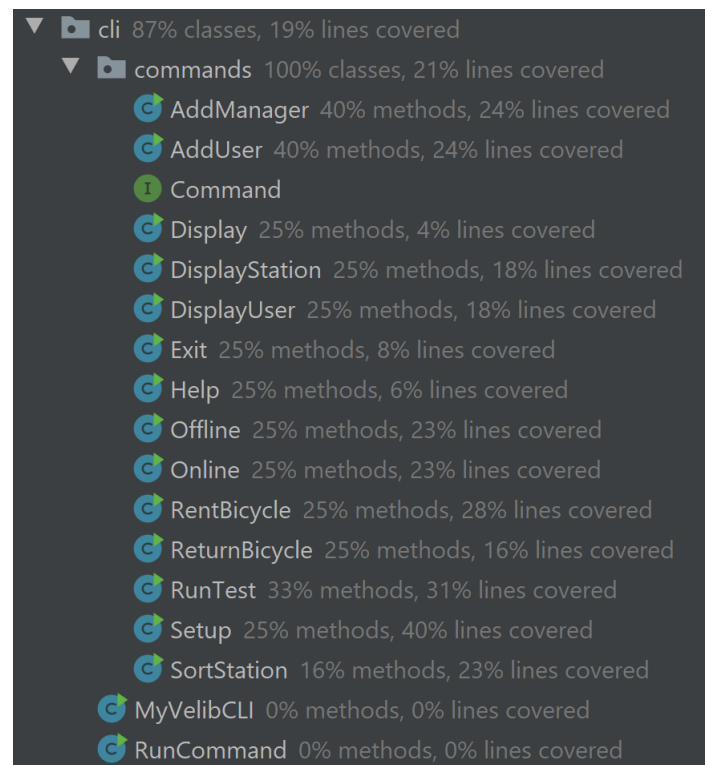


Figure 2.2: cli package

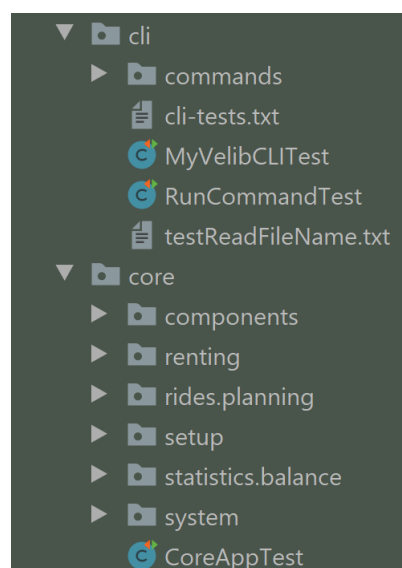


Figure 2.3: test package

We know that you (the testers) would like to see our classes implementation with the code, that is why we will not give a detailed image of our packages implementation, that is straightforward.

For **CLI**, we only have the commands package that implement the commands, states package that represent the states of the myvelib interpreter, and the cli main application classes that execute the user interface.

For **core**, we divided in a components package, that contains all the model classes and their behavior, a renting package, that contains all renting algorithms, a rides.planning package that contains all the planning algorithms, a setup package that contains all the algorithms to setup a myvelib network, a statistics package to deal with statistics, a system package to define a system and its factory, and finally, the Main core application.

2.a UML diagrams

All uml diagrams are in the folder **model** in the project folder (they are in .uml and .png format). And they all have been generated with the help of IntelliJ IDEA Ultimate, the IDE that we used for the project. Nevertheless, all files were exported to a Papyrus project and we checked to make sure that the project would run in any local machine.

In order to see the uml diagrams (we opted to not show them here due to the report format), just go to the folder and open the files.

3 Design decisions

3.a IDE

During the project, we have used **IntelliJ IDEA Ultimate** to develop our code. All the UML diagrams were generated by the IDE from our actual models. This shows that our class diagrams are indeed exactly what we have coded.

3.b VCS (Version Control System)

During the execution of the project, we have used **git** to code all the packages. All the code is hosted at **GitHub** in two repositories with the **MIT License**.

- <https://github.com/elyasha/myVelib>:

This is the repository with all the source code and the documentation, that is hosted in GitHub pages. The link to the documentation is given in the documentation section.

- <https://github.com/elyasha/myVelib-coverage>:

This is the repository with all the coverage test report, which can be visualized locally or on GitHub pages. The link to the coverage test report is given in the coverage section.

3.c Design patterns that we tried to use during the development

Before starting to code, we first tried to find some design patterns to code our application to be stable, easy to maintain and easy to understand.

To keep track of all design patterns used in the project, we created a file **design_patterns.md** in the root of the **myVelib** repository. We invite the reader to go to GitHub and see this file, if there are any new releases, this document can be outdated.

Nevertheless, a list, that may become outdated, is:

- Factory pattern (for class creation)

- Strategy pattern (for classes that use multiple algorithms - renting planning)
- Visitor pattern (ride payment)
- Singleton pattern (to classes that will have only one instance)
- Observer pattern (terminal must be informed if the station is on service or not)

3.d Git flow workflow

For this project, we used the git flow workflow to deal with all the branches in the source code repository.

An image to (better) explain the workflow of the group is given in figure 3.1.

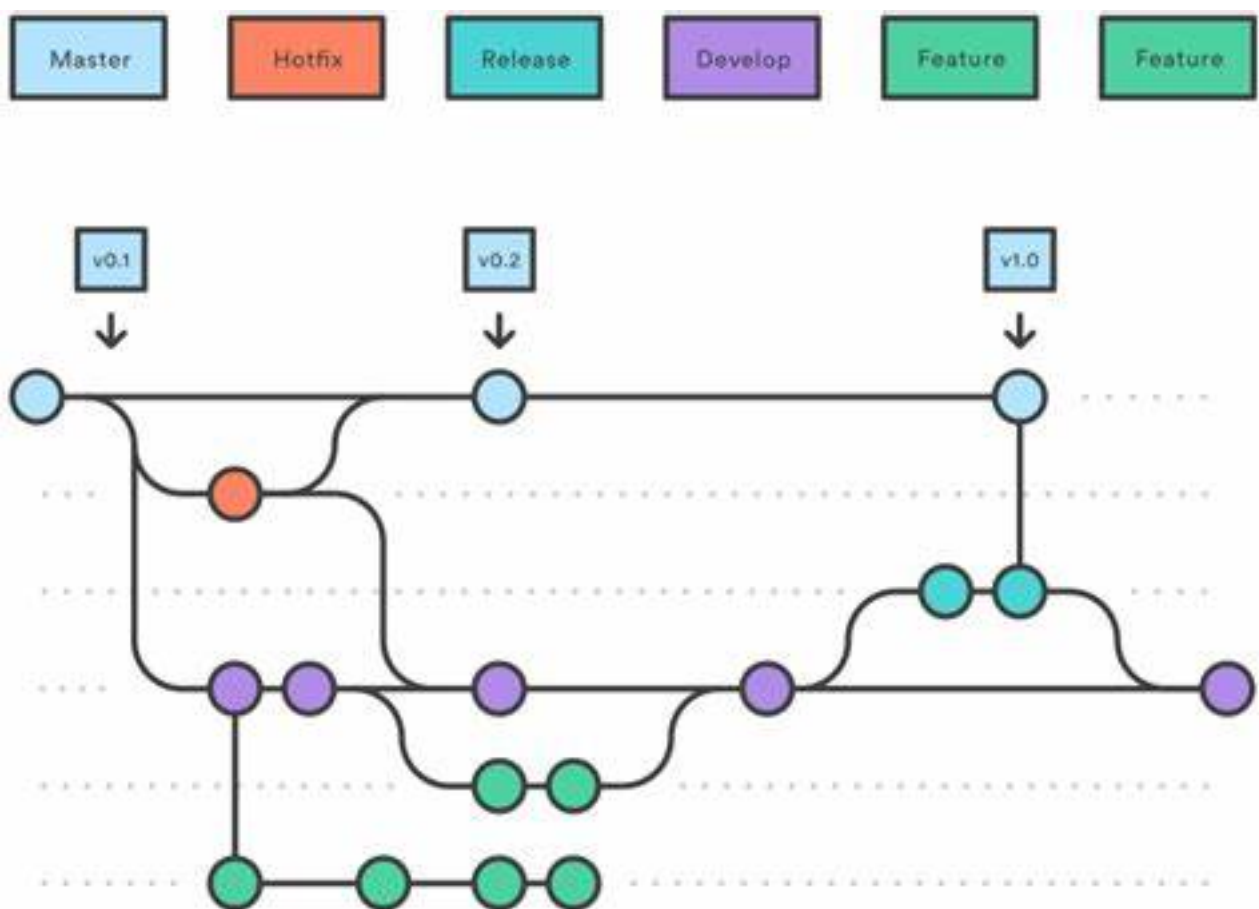


Figure 3.1: Workflow used during development

3.e Pair programming

Even if, during the course, the concept of **pair programming** had not been well explained. We decided to code all the application during hours that we could both code and discuss about our code, looking for better implementations, better algorithms and more robust methods for our classes and UML diagrams.

In this way, we can say that the work was well divided but two. For further analysis about contributions to the **myVelib** and **myVelib-coverage** repositories, please see the history of commits at GitHub.

3.f Test Driven Development/Design (TDD) and Behavior Driven Development (BDD)

We have studied both ways (TDD and BDD) to do the project. We tried to keep our discipline and follow the TDD principles. But in the end, our development process is more likely to be compared to a BDD approach.

Also, the BDD approach is very important, because our code style and development process was first oriented to the behavior of our code when considering the test scenarios. We tried to have in our code base only the needed code to pass the tests for different scenarios, that is to say, rental of a bicycle, drop a bicycle in a station, creation of a user in a myVelib system, and so on. Also keeping in mind, the deadline that was imposed and what was possible to do by the due time. (This kind of framework is better develop in longer periods of time, without the pressure of the final exams)

Of course that due to the fact that we used more the BDD approach, the quantity of unit tests were not the same as if we were using the TDD approach. But this is not a huge problem, because what needs to be solid is the design itself. For that, we followed the **SOLID**, **KISS**, **DRY** and **YAGNI** principles and appropriate design patterns.

3.g Testing

For this project, we decided to use **junit4** in order to standardize the test package.

4 Advantages and limitations of the solution

4.a Advantages

First of all, the main advantage of this JAVA framework is that the documentation is well written, we tried to apply a good software design approach during all the development. That will be able to see with a possible maintenance of this framework after the deadline of the project by the original developers. We would like to add another features in order to make the project even better.

The fact that the project is indeed well organized in its package structure makes it really simple to understand, debug or even contribute at GitHub. The CLI is a good point to mention because we tried to implement it using good design patterns. Which makes the project really easy to contribute due to the open-closed principle that was kept in mind when designing the system.

This fact can be seen, for example, in the CLI package. To add a new command, it suffices to add a new class with an "execute" method (or main method) and add the command in an appropriate switch case.

4.b Limitations

In the limitation side, we may say that due to the fact that we used the Point class, that implements a point with two integers. The coordinates in our project are supposed to be integers, which can be improved for later releases of the software. Another limitation is the fact that we did not implement a graphical user interface (GUI), mainly because the course has been shorted due to the sanitary crisis and to time restrictions at the final exam period.

Another limitation is the fact that we did not try to implement a payment package because we were not going to deploy the application (in the end, this is an academic project). So maybe, in the future, we can work in a payment package with credit card verification, and even create a price API that will be called each time the returnBicycle method is invoked.

We have coded the rentingApp so that the user can rent an electric or mechanical bike. However, the CLI requested in the requirements file did

not give as an argument the type of bike we wanted. So we decided to give a mechanical bike by default when the user of CLI runs the `rentBicycle` command in the CLI.

5 Test scenarios

In the project, and due to specifications give by the requirements file, we have created five different test scenarios, that are going to be discussed now.

All these tests scenarios are well documented in the **eval/scenarioDescriptions** folder in our project GitHub page, or in the source code project.

It is basically, some steps that are executed with the **runTest** command, when we pass a test file path as an argument.

The test files associated with these test scenarios can be found in the folder **eval/tests**. They are all with the **.txt** extension.

Also, a good remark is that in our project, due to the main operating system (used by the developers) be Windows, we decided to call the initialization file **myvelibInitialization.txt**, due to bad interpretation of the **.ini** extension. The testers are free to change (or refactor) the file name with the **.ini** extension.

Also, to be better self explained we called the files using camelCase notation with the scenario's name, whether than using numbers that have no meaning.

5.a Setting up a myVelib system (or network)

The first test scenario. It has less than 10 commands to test, and is very simple. It just call the setup method and the addUser method.

5.b Rental of a bicycle

The second test scenario. It has a couple more lines, but it is very simple and test the rentBicycle method that was written by us

5.c Simulation of a planning ride

Well, we did not created a planRide command, because it was not well designed in the pdf file whether or not a user can plan a ride without renting a bicycle. Then we decided to implement planRide on the fly with the rentBicycle command. That is why in this test scenario we call the

rentBicycle command. And finally we can see the results of the planning package with its algorithms.

5.d Visualization (user, station and system)

This test scenario is very interesting because it allows us to see the call of the toString method for each object. This test will use the implementation of our methods: **report**, **userReport**, **stationReport**.

5.e Computation statistics

The largest test scenario and the scenario that was very important to inform us when to stop writing code. Given that the statistics were the latest features that we implemented, this is the test scenario that will use the largest part of our code base.

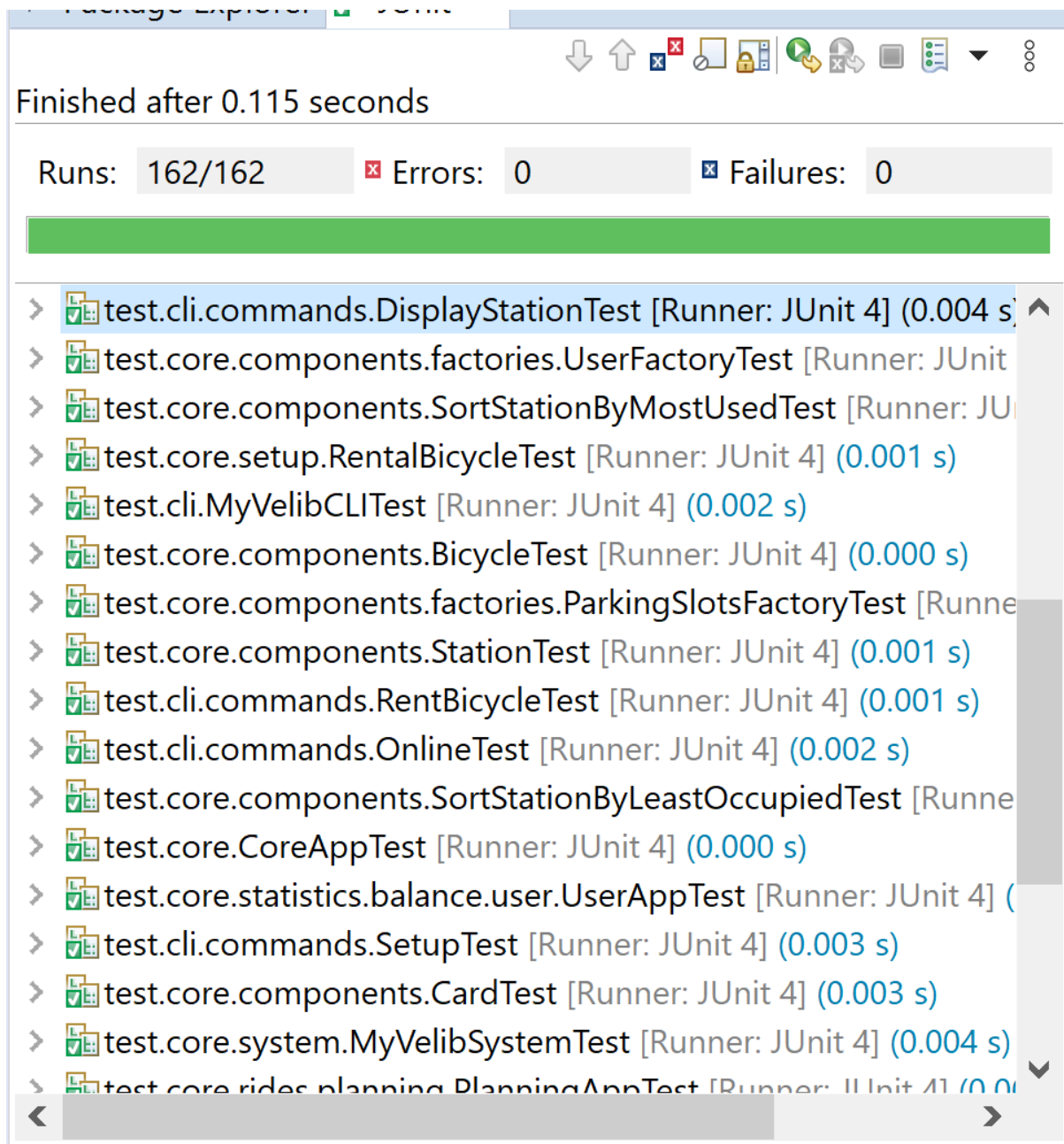


Figure 5.1: Test suite that passes (162/162 tests)

6 How to test our project

First, make sure the **JUnit 4** library is in the project and that you are running with **jdk 11 (we used 11.0.7)**. Normally, we manually added these libraries.

To test our project, it is really easy, just run the **runTest** command with an argument that is the text file path on your system.

In a IDE, you can just execute the **RunTest** class with some (consistent) argument. That is to say, the file path on your system. On Windows, it is something like "**C:\...**"

Or, of course, you can execute the myVelib CLI. Executing the cli, the initialization file will be executed, it will create a myvelib system (or network) that is called "myvelib" and then you can start using the interactive command-line. For example, you can execute the runTest command inside the cli.

We are going to add some screenshots that show how to use our cli in the next subsection.

6.a Examples of utilization of the myvelib CLI

```

Run: myVelib
MyVelibCLI
"C:\Program Files\Java\jdk-11.0.7\bin\java.exe" ...
CLI to interact with myVelib
myVelib Framework v1.0 [create with win32-x64 and java11.0.7]
The setup command!
>>>
command without args myVelib
The addUser command!
>>>
command with args myVelib
The addUser command!
>>>
command without args myVelib
The addUser command!
>>>
command without args myVelib
The addManager command!
>>>
command with args myVelib
The addManager command!
>>>
command without args myVelib
The addManager command!
>>>
command with args myVelib
The addUser command!
The system does not exist
>>>
command without args myVelib
The addUser command!
There is an error in the card type!
Please, add a coherent card type (Vmax, Vlibre, None)
There is a problem with the arguments passed!
Please add some (consistent) argument! For help: myVelib help [COMMAND]
>>>
|

```

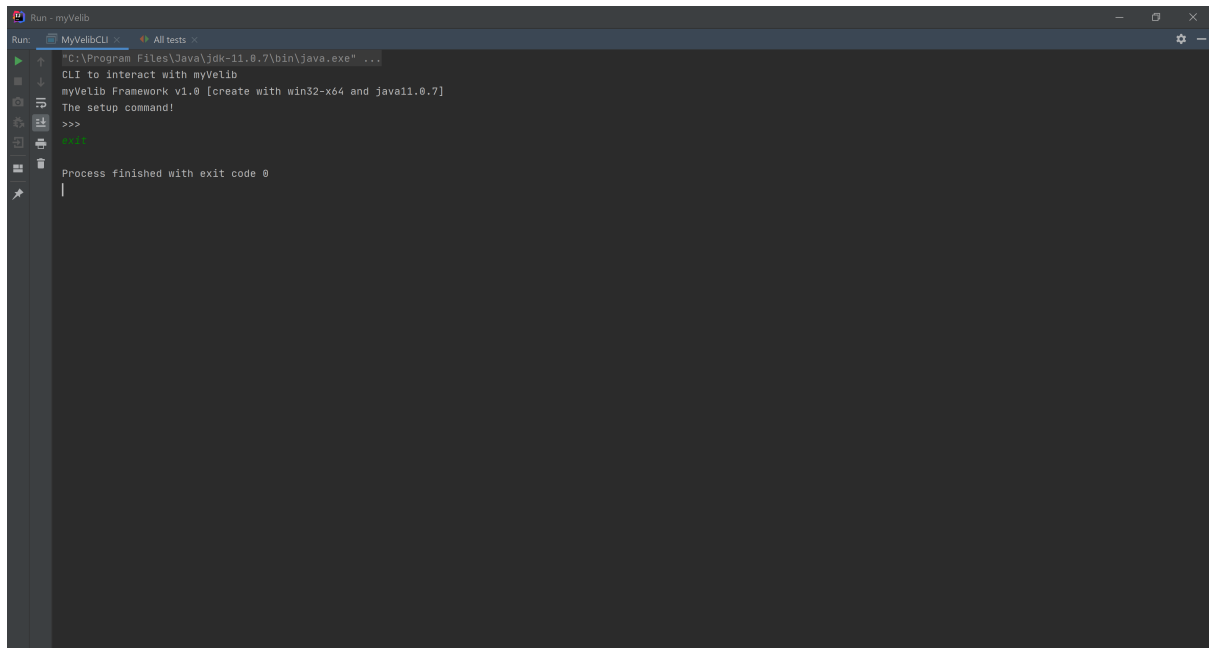
Figure 6.1: Executing the addUser and addManager commands in the CLI

```

Run: myVelib
MyVelibCLI
"C:\Program Files\Java\jdk-11.0.7\bin\java.exe" ...
CLI to interact with myVelib
myVelib Framework v1.0 [create with win32-x64 and java11.0.7]
The setup command!
>>>
command myVelib
The display command!
SYSTEM REPORT: myVelib
MyVelibSystem{stations=[Station{id=1, coordinate=java.awt.Point[x=58,y=657], onService=true, parkingSlots=[ParkingSlot{id=6, state=0}, ParkingSlot{id=7, state=0}, ParkingSlot{id=8, state=0}, ParkingSlot{id=9, state=0}], ...}
>>>
command without args
The display command!
The system does not exist
>>>
command without args
The displayStation command!
USER REPORT: Matheus
User{id=1, name='Matheus', coordinate=java.awt.Point[x=58,y=272], card=null, creditCardNumber=0, timeCreditBalance=0.0, allMoneyCharged=0.0, currentMoneyCharged=0.0, money=100.0, ...}
>>>
command without args
There is a problem with the arguments passed!
Please add some (consistent) argument! For help: myVelib help [COMMAND]
>>>
command without args
The displayStation command!
STATION REPORT: 1
Station{id=1, coordinate=java.awt.Point[x=58,y=657], onService=true, parkingSlots=[ParkingSlot{id=6, state=0}, ParkingSlot{id=7, state=0}, ParkingSlot{id=8, state=0}, ParkingSlot{id=9, state=0}, ...}
>>>
|

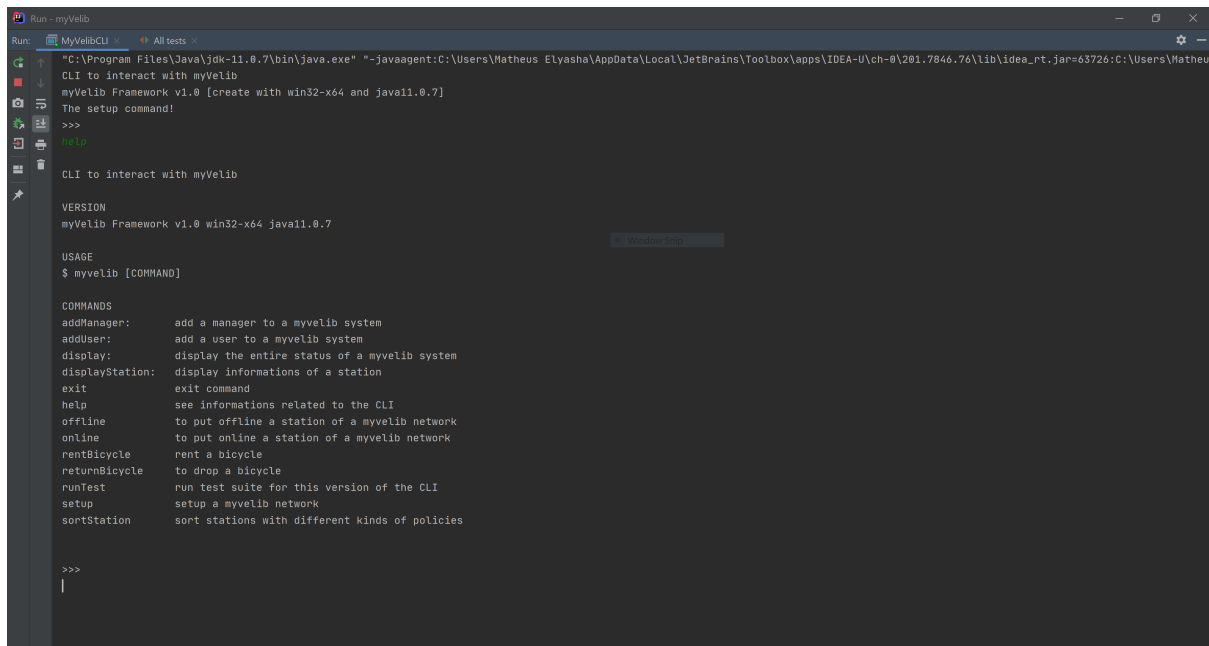
```

Figure 6.2: Executing the display, displayStation and displayUser commands in the CLI



```
Run: myVelib
MyVelibCLI
" C:\Program Files\Java\jdk-11.0.7\bin\java.exe" ...
CLI to interact with myVelib
myVelib Framework v1.0 [create with win32-x64 and java11.0.7]
The setup command!
>>>
exit
Process finished with exit code 0
```

Figure 6.3: Executing the exit command in the CLI



```
Run: myVelib
MyVelibCLI
" C:\Program Files\Java\jdk-11.0.7\bin\java.exe" "-javaagent:C:\Users\Matheus\Ellyasha\AppData\Local\JetBrains\Toolbox\apps\IDEA-U\ch-0\201.7846.76\lib\idea_rt.jar=63726:C:\Users\Matheus\Ellyasha\AppData\Local\JetBrains\Toolbox\apps\IDEA-U\ch-0\201.7846.76\bin\java.exe" ...
CLI to interact with myVelib
myVelib Framework v1.0 [create with win32-x64 and java11.0.7]
The setup command!
>>>
help
CLI to interact with myVelib

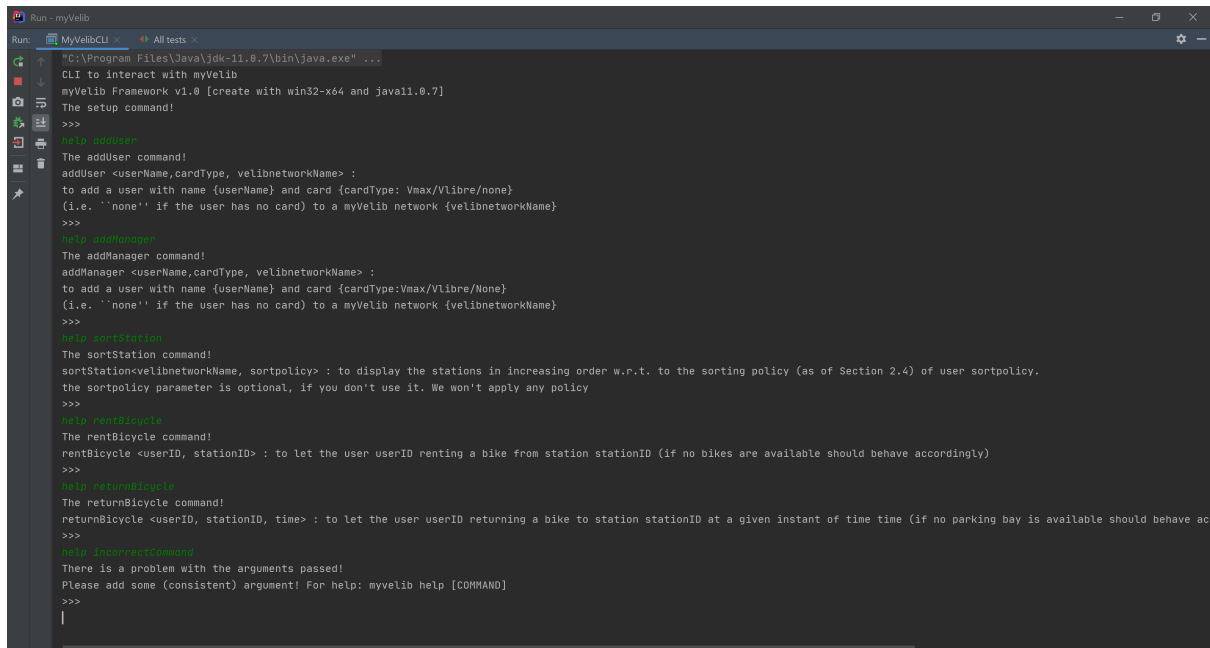
VERSION
myVelib Framework v1.0 win32-x64 java11.0.7

USAGE
$ myVelib [COMMAND]

COMMANDS
addManager:    add a manager to a myVelib system
addUser:       add a user to a myVelib system
display:       display the entire status of a myVelib system
displayStation: display informations of a station
exit           exit command
help           see informations related to the CLI
offline        to put offline a station of a myVelib network
online         to put online a station of a myVelib network
rentBicycle    rent a bicycle
returnBicycle  to drop a bicycle
runTest        run test suite for this version of the CLI
setup          setup a myVelib network
sortStation    sort stations with different kinds of policies

>>>
```

Figure 6.4: Executing the help command in the CLI

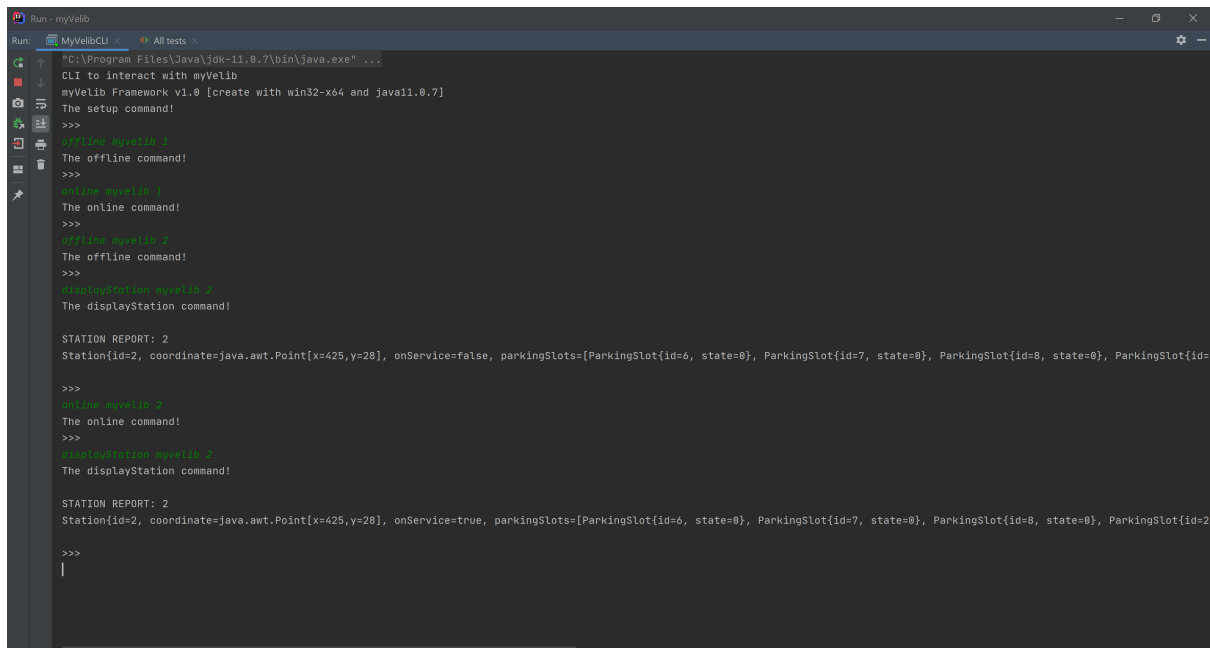


```

Run: myVelib
MyVelibCLI
All tests
"C:\Program Files\Java\jdk-11.0.7\bin\java.exe" ...
CLI to interact with myVelib
myVelib Framework v1.0 [create with win32-x64 and java11.0.7]
The setup command!
>>>
The addManager command!
addManager <userName,cardType, velibnetworkName> :
to add a user with name {userName} and card {cardType: Vmax/Vlibre/none}
(i.e. ''none'' if the user has no card) to a myVelib network {velibnetworkName}
>>>
The addManager command!
addManager <userName,cardType, velibnetworkName> :
to add a user with name {userName} and card {cardType:Vmax/Vlibre/None}
(i.e. ''none'' if the user has no card) to a myVelib network {velibnetworkName}
>>>
The sortStation command!
sortStation<velibnetworkName, sortpolicy> : to display the stations in increasing order w.r.t. to the sorting policy (as of Section 2.4) of user sortpolicy.
the sortpolicy parameter is optional, if you don't use it. We won't apply any policy
>>>
The rentBicycle command!
rentBicycle <userID, stationID> : to let the user userID renting a bike from station stationID (if no bikes are available should behave accordingly)
>>>
The returnBicycle command!
returnBicycle <userID, stationID, time> : to let the user userID returning a bike to station stationID at a given instant of time time (if no parking bay is available should behave accordingly)
>>>
There is a problem with the arguments passed!
Please add some (consistent) argument! For help: myvelib help [COMMAND]
>>>
|

```

Figure 6.5: Executing the help command with some arguments in the CLI



```

Run: myVelib
MyVelibCLI
All tests
"C:\Program Files\Java\jdk-11.0.7\bin\java.exe" ...
CLI to interact with myVelib
myVelib Framework v1.0 [create with win32-x64 and java11.0.7]
The setup command!
>>>
The offline command!
>>>
The online command!
>>>
The offline command!
>>>
The displayStation command!
>>>
STATION REPORT: 2
Station{id=2, coordinate=java.awt.Point[x=425,y=28], onService=false, parkingSlots=[ParkingSlot{id=6, state=0}, ParkingSlot{id=7, state=0}, ParkingSlot{id=8, state=0}, ParkingSlot{id=9, state=0}]
>>>
The online command!
>>>
The displayStation command!
>>>
STATION REPORT: 2
Station{id=2, coordinate=java.awt.Point[x=425,y=28], onService=true, parkingSlots=[ParkingSlot{id=6, state=0}, ParkingSlot{id=7, state=0}, ParkingSlot{id=8, state=0}, ParkingSlot{id=9, state=0}]
>>>
|

```

Figure 6.6: Executing the online and offline commands in the CLI

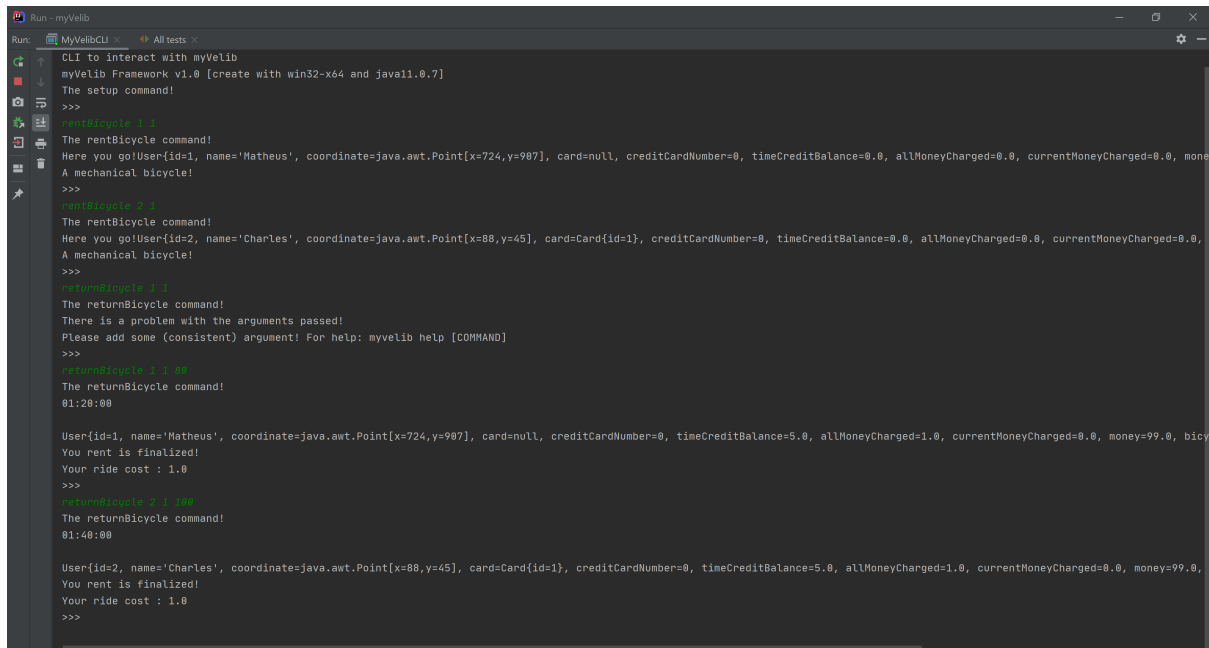


Figure 6.7: Executing the `rentBicycle` and `returnBicycle` commands in the CLI

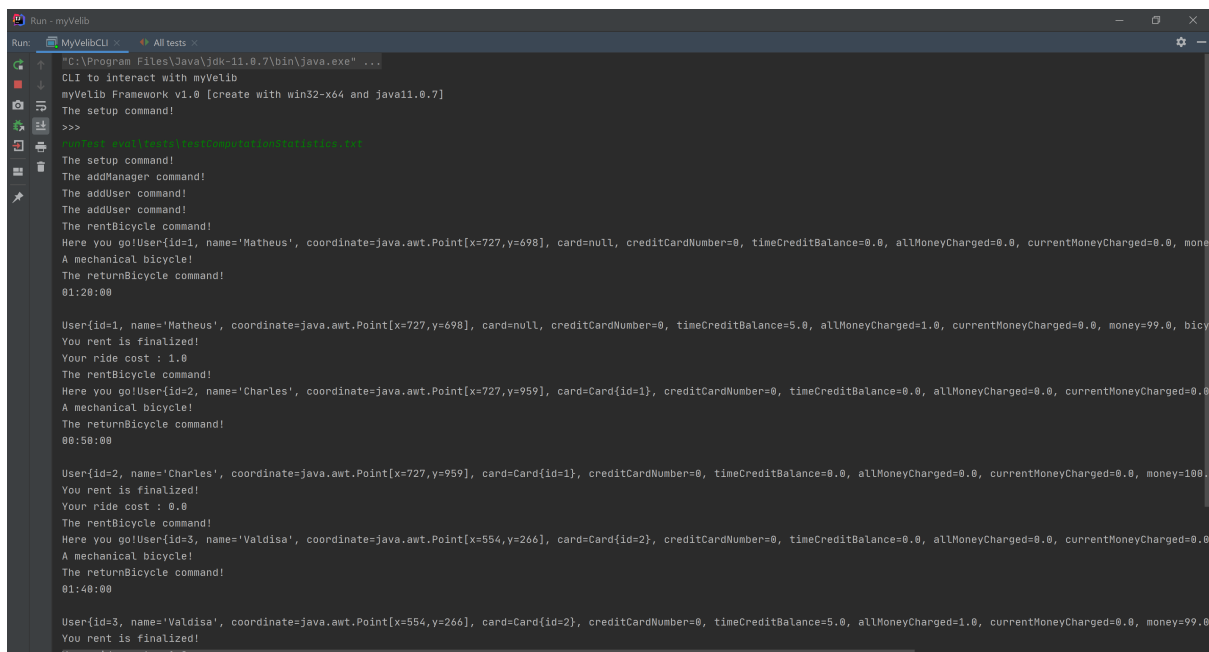


Figure 6.8: Executing the runTest command in the CLI

```

Run: MyVelibCLI < All tests <
runTest tests{nameOfTheTest}.txt
The file cannot be found! If you are running tests, please use (without the {}) and change nameOfTheTest):
runTest eval\tests{nameOfTheTest}.txt
>>>
The setup command!
The addUser command!
The addUser command!
The addUser command!
The rentBicycle command!
Here you go!User{id=1, name='Matheus', coordinate=java.awt.Point[x=894,y=589], card=null, creditCardNumber=0, timeCreditBalance=0.0, allMoneyCharged=0.0, currentMoneyCharged=0.0, money=92.0, bicycle=1}
A mechanical bicycle!
The returnBicycle command!
00:20:00

User{id=1, name='Matheus', coordinate=java.awt.Point[x=894,y=589], card=null, creditCardNumber=0, timeCreditBalance=5.0, allMoneyCharged=8.0, currentMoneyCharged=8.0, money=92.0, bicycle=1}
You rent is finalized!
Your ride cost : 8.0
The rentBicycle command!
Here you go!User{id=1, name='Matheus', coordinate=java.awt.Point[x=894,y=589], card=null, creditCardNumber=0, timeCreditBalance=5.0, allMoneyCharged=8.0, currentMoneyCharged=8.0, money=92.0, bicycle=1}
A mechanical bicycle!
The returnBicycle command!
01:00:00

User{id=1, name='Matheus', coordinate=java.awt.Point[x=894,y=589], card=null, creditCardNumber=0, timeCreditBalance=5.0, allMoneyCharged=8.0, currentMoneyCharged=8.0, money=92.0, bicycle=1}
You rent is finalized!
Your ride cost : 0.0
The rentBicycle command!
Here you go!User{id=2, name='Charles', coordinate=java.awt.Point[x=174,y=126], card=Card{id=1}, creditCardNumber=0, timeCreditBalance=0.0, allMoneyCharged=0.0, currentMoneyCharged=0.0, money=99.0, bicycle=1}
A mechanical bicycle!
The returnBicycle command!
01:20:00

User{id=2, name='Charles', coordinate=java.awt.Point[x=174,y=126], card=Card{id=1}, creditCardNumber=0, timeCreditBalance=5.0, allMoneyCharged=1.0, currentMoneyCharged=8.0, money=99.0, bicycle=1}
You rent is finalized!
Your ride cost : 1.0
>>>

```

Figure 6.9: Executing the runTest command with other file in the CLI

```

Run: myVelib
CLI to interact with myVelib
myVelib Framework v1.0 [create with win32-x64 and java11.0.7]
The setup command!
>>>
The setup command!
>>>
The display command!

SYSTEM REPORT: myvelib2
MyVelibSystem{stations=[], users={}}

>>>
The setup command!
>>>
The display command!

SYSTEM REPORT: myvelib3
MyVelibSystem{stations={Station{id=11, coordinate=java.awt.Point[x=876,y=295], onService=true, parkingSlots={ParkingSlot{id=14, state=0}, ParkingSlot{id=15, state=0}, ParkingSlot{id=16, state=0}}, users={}}

>>>
There is a problem with the arguments passed!
Please add some (consistent) argument! For help: myvelib help [COMMAND]
>>>
The setup command!
>>>
The display command!

SYSTEM REPORT: 1
MyVelibSystem{stations=[], users={}}

```

Figure 6.10: Executing the setup command in the CLI



7 Documentation

For the documentation, the best source of (up-to-date) information, we have just used the **javadoc** command that is built-in with **jdk 11**, the main pre-requisite of our projet.

It was a difficult task to write the documentation for all the classes in our application, but due to good practices when writing good code, we have chosen to well document all our code.

The documentation can be found in the following link:

<https://elyasha.github.io/myVelib/>

8 Project coverage (testing)

For the initial part of the project, we have tried to use the Test Drive Development (TDD) approach. But, it is really difficult to keep this discipline during the period of final exams while being really productive. That is the reason why we could not guarantee a **100 %** coverage report and the fact that inspired us to go for the BDD (behavior driven development).

Nevertheless, the main methods and classes are tested and we believe that even if we did not have enough time to write all tests, our application is well written and have good design. That is to say, the developers believe that the applications is good to be shipped as a MVP or as a Java package in a near future after a second iteration of the code source given a release 2.0.

The project's coverage report was generated with the help of **IntelliJ IDEA Ultimate**, the IDE that was mainly used to conclude the project.

The coverage report can be found in the following link:

<https://elyasha.github.io/myVelib-coverage/>

Here, we may recommend, using the local version of the coverage report that is better to visualize, due to missing libraries in GitHub pages, that usually does not show the good design of the coverage report online.

9 Conclusion

With this final project, we have used almost all concepts learned during the lectures and tutorials. In this sense, we feel very confident to say that the process of software design was well learned by us.

We could think about the requirements, start writing our code and seeing the development of our design during the last month. We invite the reader to take a look at our commit history at [GitHub](#) to see how much we were able to develop our tech skills and finish this impressive project.