Object-Oriented Software Engineering (2EL1520)

Engineering curriculum: 2^{nd} year

Final Project: myVelib

Professors:

Paolo BALLARINI — paolo.ballarini@centralesupelec.fr
Arnault LAPITRE — arnault.lapitre@centralesupelec.fr

Authors:

Osman MONLA (2101461) — osman.monla@student-cs.fr
Louis LHOTTE (2101405) — louis.lhotte@student-cs.fr

2022/2023 – 2nd Semester

Contents

1 Introduction

This work represents the finality of the Object-Oriented Software Engineering (EL1520) course at CentraleSupélec. This project - myVelib - consists of writing an application for bike sharing. It is designed for city inhabitants to allow them to rent bikes in order to move around a metropolitan area. The appropriate way to attack this project can be deduced from a study of its complexity. This project is about making a lot of different objects interact with each other where each object has its own properties (attributes) and functionalities (methods). Thus, object-oriented programming seems to be an effective paradigm to adopt.
The complexity of a problem or any matter comes, by definition, from the high number of interacting parts and their dependencies on each other. Indeed, myVelib handles interactions between *Users*, that wish to move from one point to another in the city by renting *Bicycles* that come in two types: **mechanical** or **electrical**; *Docking stations* where the users may rent an available bicycle of the desired type from a *Parking Slot* or return one they finished using it to a *Parking Slot*. The renting, the returning and the cost operations are all handled by a *Terminal* in the *Docking Station*.
Hence, to accurately describe the objects in interaction, we modelled them using different classes, some of which are composed of one another, and some of which are used to create subclasses. The goal was to build an application with clean code that applies the principles of Object-Oriented Programming such as code reuse, inheritance, Polymorphism and the Open-Closed principle.
A thorough reading of the system requirements led us to decide what we should use as for the implementation of the methods needed to make this system works. Thus, design patterns were used where we judged important, possible and beneficial in order to facilitate an extension/update to the application.
One should note that this final project is far from being ready to be used by an End-Customer as there is no Graphical User Interface or any front-end development (no HTML, nor CSS, nor JavaScript).

2 Task distribution

In this small section, we'll have an overview of the contribution of each member of this group. Both, Osman MONLA and Louis Lhotte, worked using GitHub using Java as a programming language, and IntelliJ as an IDE. The link to the GitHub repository, created by Louis, can be found here. Note that WinUX-Sudo (found in the commits of this project) and Osman Monla are the same person.
Louis LHOTTE handled the implementation of the Cost method using the **Observer Pattern** along with the Rent Bike method, the JUnit tests and the base case scenario.
Osman MONLA handled the implementation of the planning function using a **Stategy Pattern**, along with the Park Bike method, the User and Station Balances, and the Station Sorting policies using a **Stategy Pattern**.
Both, Osman MONLA and Louis LHOTTE, worked on the architecture of this final project - myVelib - thus, the UML diagram.

3 Classes

In this section, we are going to explore the classes that we implemented in order to achieve the goal of the myVelib application.

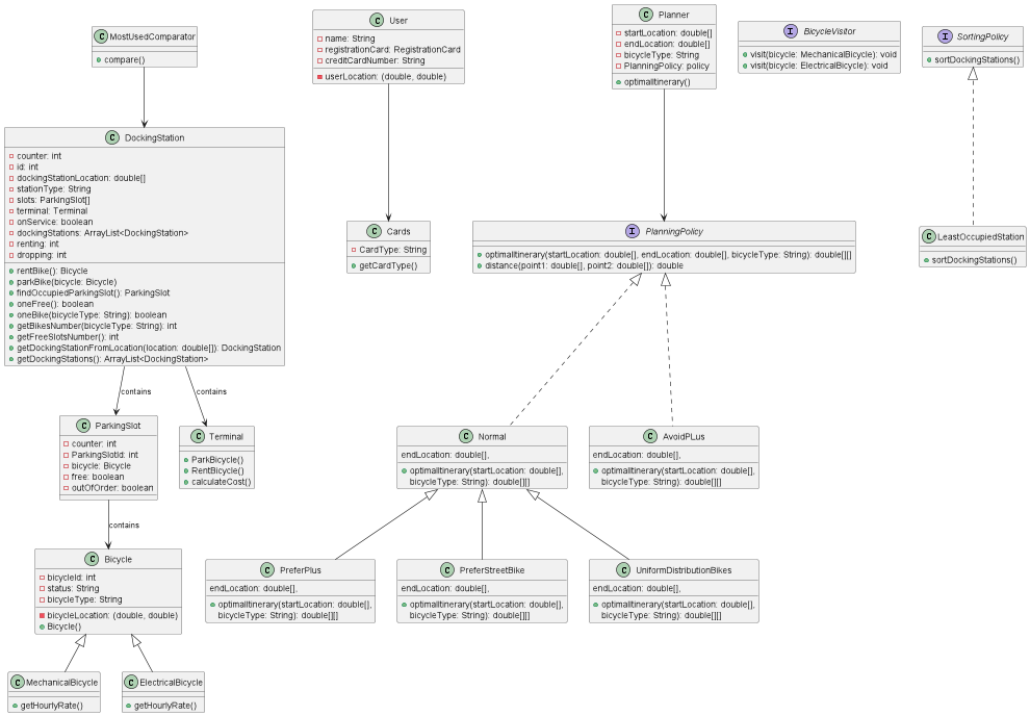

Figure 1: UML diagram of core classes

3.1 User

We start by the **User** class. This class represents a user of the myVelib system. Each User is characterized by:

- a unique ID. (int)
- a name. (String)
- a credit card. (String)
- a GPS location. (double[])
- (optionally) a registration card. (registrationCard)

Some additional statistics can be found as attributes of a User:

- Time credit balance. (integer)
- total charges of a user. (double)
- total number of rides of a user. (integer)

- total rent time of a user. (integer)
- Time credit **earned**. (integer)
- Current rented Bicycle (Bicycle).
- Rent date time (=null if the user is not renting a Bicycle) (LocalDateTime)
- Park date time (LocalDateTime)

This class is equipped with a 3-argument constructor that initializes the *name*, the *location*, and the *credit card*, and another 4-argument constructor that initializes the same before-mentioned attributes in addition to the *registration card*. All of the other fields are initialized to their default value, except for the *ID* field.
The *ID* field is set by a **static** counter that increases by 1 at each call of any of the 2 constructors mentioned above. The first ID takes the value 1.

Note: All of the classes that contain an *ID* field, treat the *ID* field as such.

3.2 Bicycle

Now, we move to the **Bicycle** class. This class models the bicycles used in the myVelib system. Each bicycle is characterized by:

- a unique ID. (integer)
- a GPS location. (double[])
- a type. (String)

Although, we could have handled the type with an **Enum** class, we decided to code the type of the bicycle with a **String-type** attribute. The bicycle ID is handled in the same way as the User ID is handled.
Furthermore, as mentioned in the system requirements, bicycles can be found not only in Docking Stations but also on the street. Hence, in order to keep track of all the bicycles that are not parked in Docking Stations but rather parked on the street, we created a **static ArrayList** of bicycles that may get bigger at each call of the constructor (at creation, a bicycle object is added to this ArrayList called streetBicycles only if the location of the bicycle at creation is different from all Docking Station locations).
The constructor of Bicycle takes 2 arguments: the GPS location, and the Bicycles type.

Note: The **Bicycle** is not designed to be used to instantiate objects but rather a superclass to derive two new classes: MechanicalBicycle and ElectricalBicycle.

3.2.1 MechanicalBicycle

As mentioned in the note above, we created a subclass of **Bicycle** called **MechanicalBicycle** that has one additional field hourlyRate that will be initialized in a 1-argument constructor to the value of 1.0. The constructor takes as argument a GPS location and inherits the constructor of **Bicycle** via **super(bicycleLocaiton, "Mechanical")**.

3.2.2 ElectricalBicycle

Implementing a new subclass of **Bicycle** - **MechanicalBicycle** - won't be done without creating another subclass of **Bicycle** called **ElectricalBicycle**. It also contains one (more) field called hourlyRate that will be initialized in a 1-argument constructor to the value of 2.0. The constructor takes as argument a GPS location and inherits the constructor of **Bicycle** via **super(bicycleLocaiton, "Electrical")**.

Note: The *hourlyRate* field is used to compute the cost of a ride.

3.3 ParkingSlot

ParkingSlots are the places where **Bicycle**s are (preferably) parked. They are located in Docking Stations. Each parking slot is characterized by:

- a unique ID. (integer)
- a bicycle. (Bicycle)
- a free status (indicates whether the parking slot is free or not). (boolean)
- an out-of-order status (indicates whether the parking slot is out-of-order or not). (boolean)

Obviously, we could have reduced the number of fields by replacing the last by one of class **Enum** indicating the status: FREE, TAKEN, OUT_OF_ORDER. But, we decided to go with 2 boolean fields. The ID fields is handled the same way as in the classes talked about above. This class is equipped with a 3-argument constructor whose arguments are: a **bicycle**, a boolean value **free** that indicates whether the parking slot is free or not, and another boolean value that indicates whether the given parking slot is operational or out-of-order.

3.4 Terminal

The **terminal** is where most of the interaction between a user and the application happens. A terminal, in our design of myVelib, does not have any attributes. Hence, its constructor is the default constructor. It is a way to access most of the functionalities of myVelib. It implements the BicycleVisitor interface that we created in order to use the **Visitor** interface to implement the **cost** function of a ride.

3.5 DockingStation

The **DockingStation** is where a user would (most probably) head to to rent a bike or to park one. Each **DockingStation** object is characterized by:

- a unique ID. (integer)
- a GPS location. (double[])
- a type. (String)
- an Array of parking slots. (ParkingSlot[])
- a terminal (all docking station **may** share the same terminal object). (Terminal)
- a boolean attribute indicating whether or not the docking station is operational called *onService*. (boolean)

Some additional statistics can be found as attributes of a DockingStation object:

- the number of renting operations (integer).
- the number of dropping operations (integer).

In addition to all those instance attributes, we keep track of all docking stations created by creating a class attribute of type ArrayList that gets bigger at each call of the constructor (a DockingStation object is added to this list at creation). Finally, the constructor takes 4 arguments:

- a GPS location. (double[])
- a type. (String)
- an Array of parking slots. (ParkingSlot[])
- a terminal (all docking station **may** share the same terminal object). (Terminal)

A DockingStation object is on-service at creation and has 0 for dropping and renting operations.

3.6 Cards

This small section deals with registration cards. As mentioned in the system requirements and above, a user may or may not have a registration card, however having one has consequences on the cost. A registration card can be of 2 types: VLIBRE or VMAX. Hence, we modelled this choice by String-type attribute (even though we could have done it using an **Enum** class). The constructor takes one argument: the type of the card. As mentioned in the User section, this Card is a field for the user.

Note: In this project, we could have created this class as a nested one in the User class because only the user can have them and use them. But creating a separate class for the cards makes the code more resistible to changes.

3.7 Other

In the following sections, we will explore the other classes and interfaces that we created to implement the required functionalities of the myVelib system. The purpose of those additional classes and interfaces is to make the code follow the Open-Closed principle.

4 Ride Costs

To calculate the cost of a rental for a user, the calculateCost method in the Terminal class follows a specific methodology:

1. Retrieve the rented bicycle from the user object.
2. Initialize the cost variable to 0.
3. Ensure that the user and their registration card are not null.
4. Determine the user's registration type and time credit balance.
5. Set the free hour duration to 60 minutes.
6. Determine the hourly rate based on the type of bicycle rented.
7. Calculate the cost based on the user's registration type:
   - For users with "VLIBRE" registration:
     (a) Calculate the duration excess by subtracting the free hour duration from the rental duration, ensuring a minimum value of 0.
     (b) Calculate the effective duration by subtracting the time credit from the duration excess, ensuring a minimum value of 0.
     (c) Calculate the cost by multiplying the effective duration (converted to hours) by the hourly rate.
     (d) Adjust the time credit by subtracting the duration excess.
     (e) Update the user's time credit balance.
   - For users with "VMAX" registration:
     (a) Calculate the duration excess by subtracting the free hour duration from the rental duration, ensuring a minimum value of 0.
     (b) Calculate the effective duration by subtracting the time credit from the duration excess, ensuring a minimum value of 0.
     (c) Calculate the cost as a fixed rate of 1 euro per hour of the effective duration.
     (d) Adjust the time credit by subtracting the duration excess.
     (e) Update the user's time credit balance.
   - For users with other registration types:
     (a) Calculate the cost by multiplying the total duration (converted to hours) by the hourly rate.
8. Return the calculated cost.

This methodology ensures a systematic approach to calculating ride costs. The algorithm is designed to handle various registration types and accurately determine the cost based on the rental duration and the hourly rate of the rented bicycle.

By following this methodology, the calculateCost method provides reliable and consistent cost calculations for rentals in the myVelib system.

5 Planner

In this section, we cover the planning policies mentioned in the system requirements. myVelib is equipped with a functionality that helps users plan a ride giving a starting GPS location, an end GPS location and a given type of bike: mechanical or electrical. The planner doesn't give the trajectory to follow but rather - given a certain policy - the start and end location of a docking station.
Given the diversity of the existing policies and the incompatibility among them, we decided to implement a **Strategy Patter** to let the user (or the system managers) choose one of the existing implementations of the algorithm.
This was done by creating a **PlanningPolicy** interface in which is defined a method (with no body) called *optimalItinerary*, a default method called *distance* which computes the Manhattan distance between 2 points given as argument (this distance is chosen because it is more adapted than the Euclidean one in a city where the taxicab geometry is adopted).
The object that we will have to create to find the "optimal" itinerary is of type **Planner** whose attributes are:

- a start GPS location. (double[])
- a end GPS location. (double[])
- a type of bicycle. (String)
- a planning policy. (PlanningPolicy)

In the following sections, we present the different classes that adopt this interface.

5.1 Normal planner

The start and end GPS location **given by the user** doesn't have to correspond to docking station locations. Thus, every planning policy - except for **"PreferStreetBike"** - should return as output 2 GPS locations that would correspond to locations of some docking stations.
The **Normal** policy constitutes the most basic policy where system will search for the nearest docking station to the starting GPS location (given by the user) and that contains at least one bike of the desired type; and the nearest docking station to the end GPS location (given by the user) and that contains at least one free parking slot. The method - *optimalItinerary* - will return the GPS location of those two docking stations.

5.2 UniformDistributionBikes planning policy

In this section we explore a policy that tries to ensures a uniform distribution of bicycles among the different docking stations. In order to do that, we start by finding the **Normal** solution. Then, we explore the docking stations that are "close" to the start docking station. By "close", we mean that the distance between the start GPS location (given by the user) and the docking station we're searching for should not exceed 1.05 times the distance between the start GPS location (given by the user) and the docking station given by the **Normal** solution. Of course we're not just trying to find a docking station that is farther from the user. The

docking station we're searching for must have more bikes than the docking station given by the **Normal** solution. That's for the starting side. Concerning the end docking station, it must follow the same distance criteria as for the start docking station. However, the end docking station we're searching for must have more free parking slots than the end docking station given by the **Normal** solution.
In order to facilitate the coding procedure, and to benefit from the Object-Oriented Programming paradigm, we decided to extend this class to the **Normal** class (i.e.) **UniformDistributionBikes** is a subclass of **Normal**.

5.3 PreferPlus planning policy

In this section, we explore another policy that does not require a uniform distribution of bikes among the docking stations. But rather, it states that the user **prefer** to return the bicycle to a "plus" station. Hence, to compute the solution we proceed in a similar way as in UniformDistributionBikes. We start by taking the solution given by the **Normal** class. To do that, we extend **PreferPlus** to **Normal** (i.e.) **PreferPlus** is a subclass of **Normal** (and implements the **PlanningPolicy** interface). Then, we scan the neighborhoods of the end docking station location given by the Normal solution. We authorize a 10% increase in the distance to the end GPS location given by the user. If no "plus" station is found in this area, the function would output the "standard" station it found.

5.4 PreferStreetBike planning policy

The **PreferStreetBike** is similar to **PreferPlus**, in the way that the user **prefer** to pick a street bicycle and is not limited only to this choice. This means that if no street bicycle is found, the optimal itinerary is that of **Normal**. Hence, this class will extend the **Normal** class (i.e.) **PreferStreetBike** is a subclass of **Normal**. In all cases, whether there were some street bicycle or not, the end location that the function should output is always the same (equal to the one given by **Normal**). To code this algorithm, we call the *optimalItinerary* given by **Normal** via **super** and then we check if there are street bikes by checking the size of the *streetBicycles* ArrayList. If it empty, then the solution is the one given by **Normal**. If not, we keep the end location and we search for the nearest street bicycle to the start location given by the user.

5.5 AvoidPlus planning policy

The **AvoidPlus** may seem similar to the **PreferPlus**, however some differences should be noted. In this case, the user does not only prefer not to drop the bicycle in a "plus" station, but rather the user would prefer drop the bicycle on the street. This means that if no "standard" docking station with a free parking slot is found (anywhere) the user will drop the bicycle on the street. Given this big constraint we won't be able to benefit from we coded in **Normal**, hence we will not extend this class to **Normal**.

6 Bike rent and Bike park

7 Statistics

This section covers some additional functionalities that myVelib supports.

7.1 UserBalance and StationBalance

In particular, myVelib keeps record, for every user, of:

- The total number of rides.
- The total number spent on bicycles.
- The total number of charges for all rides.
- The total time-credit earned by a user.

To memorize all of those values **for every user**, we created an attribute - for each statistic - in **User** that is updated at each call of the *parkBicycle* method of **Terminal**. Cf. User's section on additional statistics.

In addition, myVelib also keeps record of important statistics on **each** docking station. In particular, each **DockingStation** object contains a total number of renting operations attribute and a total number of dropping operations attribute that are updated at each call of the *rentBike()* and the *parkBike(Bicycle)* methods respectively of the **DockingStation** class.

7.2 SortingPolicy

In this section, we move to the sorting policies of the docking stations. The system requirements suggest two, incompatible, sorting strategies (i.e.) two different implementations of the *sortDockingStations* method. Knowing that, in the future, myVelib managers may wish to add new strategies and in order to prevent the code from breaking down, we decided to implement the **Strategy Pattern** to manage the sorting of the *dockingStations* ArrayList.
Therefore, we created the *SortingPolicy* interface which only declares the *sortDockingStations* method, and two classes that implements it where each one correspond to one policy.
The first policy consists of sorting the docking stations with respect to the sum of renting and dropping operations of each docking station. To achieve this, we created another class **MostUsedComparator** that implements the **Comparator** interface and overrides the *compare* method. In the **MostUsedStation** class, where we implement the *sortDockingStations* method, we use the *sort* method of the **Collections** class where we put two arguments: the *dockingStations* ArrayList and the an object of the **MostUsedComparator** class.
We proceed in exactly the same way for the second policy which consists on sorting the docking stations by the difference: $renting - dropping$, where renting and dropping are fields for each docking station that code the total number of renting operations and of dropping operations for each docking station respectively. The related class for this policy is named: **LeastOccupiedStation**, while the class that implements the **Comparator** interface is named: **LeastOccupiedComparator**.

8 JUnit tests

JUnit Tests JUnit tests were implemented to ensure the correctness and reliability of the system. The tests were written using the JUnit testing framework, and they cover various aspects of the system's functionality.

In the `TerminalTest` class, multiple tests were created to validate the `calculateCost` method. These tests cover different scenarios, such as different types of registration cards (e.g., "VLIBRE," "VMAX," or "OTHER") and different types of bicycles (e.g., mechanical or electrical). The expected costs are compared using the `assertEquals` method from the JUnit framework, providing a tolerance of ± 0.01.

The `PlannerTest` class includes tests for the `optimalItinerary` method of the `Planner` class. These tests validate the functionality of the method under different scenarios. For example, the "normal" test checks if the optimal itinerary is calculated correctly when using a normal planning policy. The "preferplus" test verifies the behavior when using a "PreferPlus" policy, and the "preferstreetbike" test checks the result with a "PreferStreetBike" policy. The expected itineraries are compared using assertions provided by the JUnit framework.

Additionally, the `UserTest` class contains tests for the `getName`, `setName`, and `addToTotalCharges` methods of the `User` class. These tests ensure the correctness of basic user-related operations. Assertions are used to compare the expected values with the actual values obtained during the tests.

Overall, the implemented JUnit tests contribute to the overall quality assurance of the system by validating its behavior and ensuring the accuracy of the implemented functionalities.

9 Command Line User interface (CLUI)

The MyVelib network utilizes a Command Line User Interface (CLUI) to interact with users. The `Command` class represents a command in the CLUI and is responsible for evaluating and executing user commands. Users can perform various actions through the CLUI, such as setting up the network, adding users, renting and returning bikes, and displaying network information.

9.1 MyVelib Network

The `MyVelibNetwork` class represents the overall MyVelib network. It manages the network's name, users, and docking stations. This class provides methods to access and manipulate the network's data, allowing for tasks such as adding and removing users, managing docking stations, and retrieving network information.

9.2 MyVelib

The `MyVelib` class represents an individual MyVelib network. Each network has a unique name and maintains a list of users and docking stations associated with it. This class provides methods to access and manage the network's data, including adding and removing users and docking stations, as well as retrieving network information specific to that network.

Command Syntax: `command-name <arg1> <arg2> ... <argN>`

9.3 Available Commands

- `setup <velibnetworkName>`: Create a myVelib network with the given name. The network consists of 10 stations, each with 10 parking slots. Stations are arranged on a square grid of side 4km. Initially, 75% of bikes are randomly distributed across the stations.
- `setup <name> <nstations> <nslots> <s> <nbikes>`: Create a myVelib network with the given name, consisting of `nstations` stations, each with `nslots` parking slots. Stations are distributed as uniformly as possible over either a circular area of radius `s` or a square area of side `s`. The network is initially populated with `nbikes` bikes randomly distributed across the stations.
- `addUser <userName, cardType, velibnetworkName>`: Add a user with the specified name and card type ('none' if the user has no card) to the myVelib network.
- `offline <velibnetworkName, stationID>`: Put the specified station offline in the myVelib network.
- `online <velibnetworkName, stationID>`: Put the specified station online in the myVelib network.
- `rentBike <userID, stationID, bicycleType>`: Allow the specified user to rent a bike from the specified station. If no bikes are available, appropriate action is taken.
- `rentBike <userID, GPS_Position>`: Allow the specified user to rent a bike from a given GPS position.
- `returnBike <userID, stationID, duration>`: Allow the specified user to return a bike to the specified station after the given duration. If no parking bay is available, appropriate action is taken. The command also displays the cost of the rent.
- `displayStation <velibnetworkName, stationID>`: Display the statistics of the specified station in the myVelib network.
- `displayUser <velibnetworkName, userID>`: Display the statistics of the specified user in the myVelib network.
- `sortStation <velibnetworkName, sortpolicy>`: Display the stations in increasing order based on the specified sorting policy of the user.
- `display <velibnetworkName>`: Display the entire status of the myVelib network, including stations, parking bays, and users.
- `exit`: Exit the myVelib network.

Overall, the MyVelib network provides a range of functionalities through its classes, allowing users to interact with the system, manage network components, and perform various tasks related to bike rental and network administration.

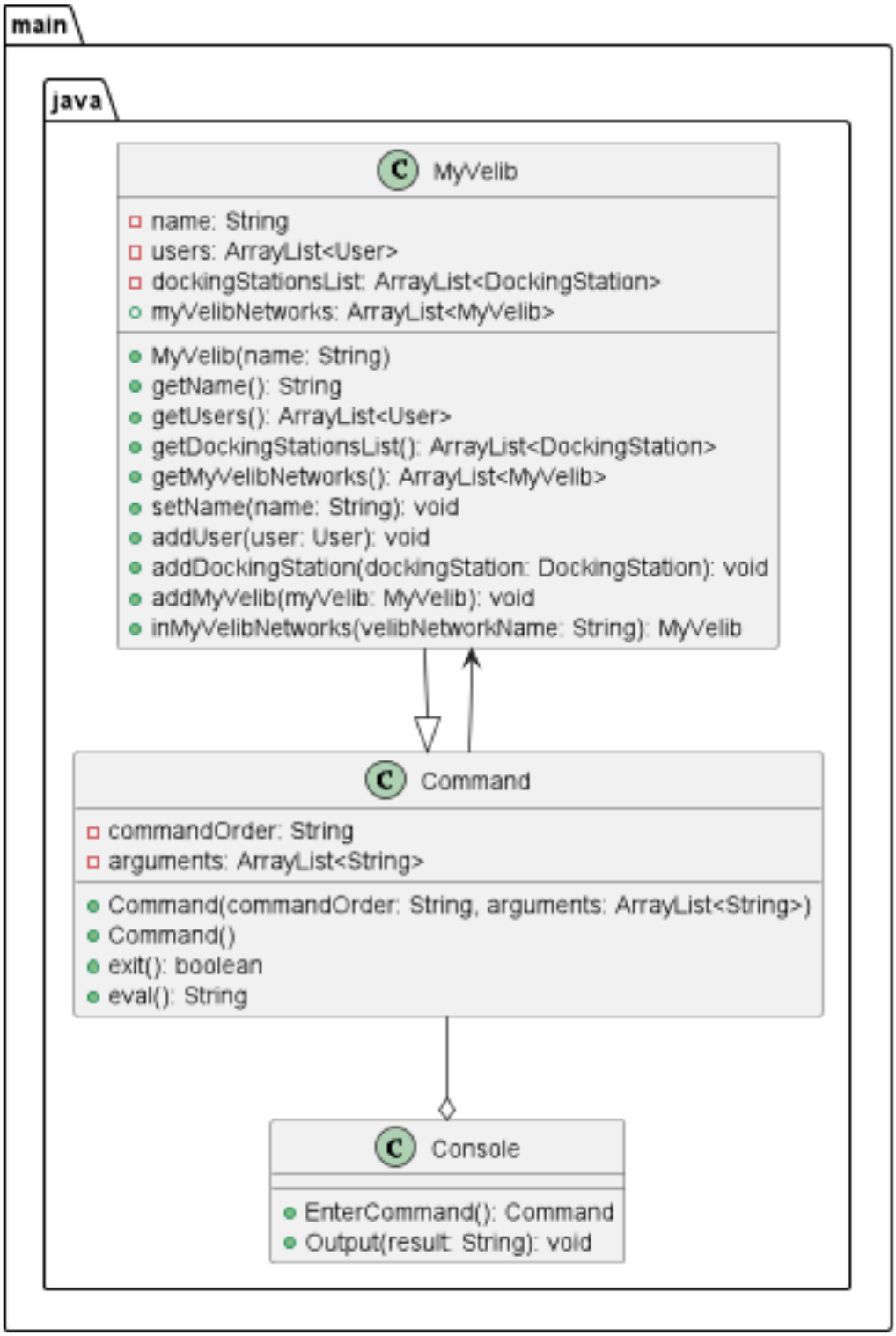

Figure 2: UML diagram of CLUI classes