CentraleSupélec | université PARIS-SACLAY

2EL1520 – Object Oriented Programming in Java

Project Report of a

Bike Sharing System - myVelib

Presented By
JABBOURI Layal
PASQUALOTTO Marco

Presented To
DR. BALLARINI Paolo
DR. MARCADET Dominique

CentraleSupélec

28 May 2023

Contents

Introduction

The goal of this project is to develop a Java application called **myVelib** for a bike sharing system, similar to popular systems like Velib in Paris. A bike sharing system enables residents to easily rent bicycles and navigate a metropolitan area. It involves various components that interact with each other, such as docking stations strategically located throughout the city, different types of bicycles including mechanical and electrical, and registered users with membership cards, etc...
This application will provide the necessary functionality to support the core operations of a bike sharing system, including bike rental, station operations, etc...
By developing this Java application, we aim to create a flexible and scalable solution that can be easily adapted and customized to meet the specific needs of different bike sharing systems. The framework will adhere to industry best practices, including the utilization of design patterns, adherence to open-close principles, and implementation of test scenarios.
Throughout this report, we will discuss the design and implementation details of the myVelib app, highlighting the key components, design patterns utilized, open-close principles applied, test scenarios considered, supported commands, as well as classes and interfaces.

Design Decisions and UML

Classes

In general, we have the following major classes in our system:

- **MyVelib**: This class represents the system netowork. Each network consists of instances of **DockingStation**, where each docking station has a number of **ParkingSlot** instances. The network also includes instances of **User** and **Bike**.
- **Card**: this is an abstract class that represents a card that users can have. It serves as the base class for different types of cards, such as **CreditCard** and **RegistrationCard**.
- **Bike**: This class represents a bike in the system.
- **DockingStation**: This is an abstract class that represents a docking station in the network. It consists of multiple **ParkingSlot** instances. It has two concrete classes which are **StdStation** and **PlusStation** representing two different types of stations.
- **ParkingSlot**: This class represents a parking slot within a docking station.
- **User**: This class represents a user in the system. It contains user information, including a **CreditCard** instance. Users can have 1 to 2 Card instances, one of which is always a **CreditCard**, and the second one is an optional **RegistrationCard**.
- **BikeRide:** This class is instantiated every time a user rents a bike from a station. It is used to describe the ride using its properties such as its starting point, destination, the corresponding user, its duration, etc....
- **Location:** It is a simple class with name, longitude, and latitude to get the precise location of the user/station/bike

In addition, we have a specialized class called **UserWithRegistrationCard**. This class inherits the User class and extends its functionality. It includes the **RegistrationCard**, which is an abstract class that serves as a base class for different types of cards, such as **VlibreCard** and **VmaxCard**.

We also added 3 classes to be able to generate unique identifiers for some classes, which are:

- **BikeIDGenerator**
- **StationIDGenerator**
- **UserIDGenerator**

These classes are to be further discussed in the Singleton Design Patterns Section.

Some might wonder why the parking slot id is not generated with a singleton class as well, and that's because the parking slots are unique within a station, which means that parking slots can have same numeric identifier if they belong to different stations. So, this implementation is a lot easier with the use of static variable *counter* in **ParkingSlot** instead of a singleton class.
To note that the cost of the bike ride will differ not only based on the ride duration, but also on the type of user riding the bike.

Interfaces

Moreover, we have two major interfaces, **RidePlanning** and **CostStrategy**. To note that the interfaces in the UML software that we used (starUML) are represented by a circle which is a bit different than the representation of interfaces in papyrus.
These two interfaces will be used, respectively, to plan the ride of the user according to different strategies, for example the closest station to the starting and ending point, and to calculate the cost of the ride based on its duration as well as the type of the user.
These interfaces will be discussed in detail in the Strategy Design Pattern Section.

Enums

Since enums are useful whenever you need to represent a fixed set of distinct values or options, we used them to represent the BikeType which can be either MECHANICAL or ELECTRICAL, as well as the SlotStatus (status of a parking slot) which can be FREE, OCCUPIED by a bike, or OUT_OF_ORDER.

To note that the remaining classes will be discussed in the Design Patterns Section.

For a detailed overview of the attributes and operations of each component, please refer to the provided UML diagram. It provides a visual representation of the class structure and their relationships within the system.

Design Patterns

Singleton Pattern

User ID generation

Since each user must have a unique identifier, we need a generator of unique identifiers.
For this purpose, we defined a class UserIDGenerator with a private integer *"num"* used to generate each unique ID.
To guarantee that no duplication happens, we must guarantee that only one UserIDGenerator instance may exist, and this could be done using the getInstance() method.
Kindly check the code of UserIDGenerator class for the detailed implementation.

Station ID generation

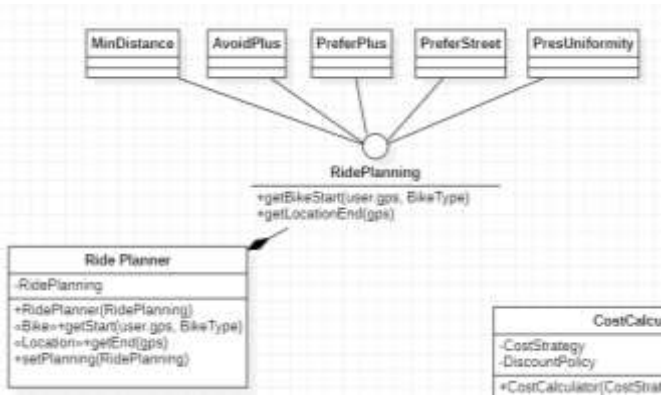
For the same purpose, we created a StationIDGenerator class, to ensure that each station in the system has a unique ID.

Bike ID generation

Another class called BikeIDGenerator is implemented to verify the uniqueness of the ID of all the bikes in the system.

Strategy Pattern

Ride planning



According to the requirements, the system has to be equipped with a functionality that helps users to plan a ride which works as follows:
If a user wants to rent a bike, we access his location automatically, yet he still has to provide the type of bike that he wishes to use (ELECTRICAL or MECHANICAL). The system should be able to compute the optimal location corresponding either to a docking station, or to a free position, from which the user can go and rent the bike.
If a user wants to return a bike to a specific destination to be provided, the system should be able to compute the optimal location of the docking station relative to the provided destination.

The optimal locations in both cases can be computed based on different strategies, such as:

- Minimum distance strategy.
- Prefer plus stations strategy.
- Avoid plus stations strategy.
- Prefer a bike parked in the street strategy.

We have an algorithm for computing the starting and ending locations, therefore we created strategies for this computation

Every ride planner has to have a ride planning behavior (composition relation).
The RidePlanning is an interface having two different functions:

- getBikeStart(Location,BikeType)
- getLocation(Location)

Each of these methods has a different behavior according to the strategy to be applied.
Therefore, concrete classes (MinDistanceStrategy, PreferPlusStrategy, etc....) representing different strategies will implement the RidePlanning interface and will implement its methods in their own way.

For instance, the MinDistanceStrategy will implement:

- *getBikeStart(Location, BikeType)* by taking the location of the user and the type of bike (arguments) he/she needs and will return the closest bike of the desired type.
- *getLocationEnd(Location)* will be implemented in such a way to return the location of closest docking station to the destination provided as an argument.
  The return type of this function is chosen to be a Location of a docking station and not a Bike since the ride planner should advise the user to always dock their bike in a docking station.

To note that we have not implemented all of the previous strategies. But in case we want to, the implementation is quite simple since it does not involve any change in the client code, so we just need to provide the implementation of the strategy and that would be it.

Cost calculation

The cost of the bike ride should be computed once the user returns the bike. This cost is dependent on two factors other than the ride duration, which are:

- The user and the type of card that he/she has (*Figure 1*)
  For this reason, we decided to have a CostStrategy interface implemented by different concrete classes that return the initial cost.
- The starting and ending locations of the bike (*Figure 2*)
  For this reason, we decided to have a DiscountPolicy interface implemented by 3 concrete classes that return a factor that could be either 1, 0.9 (in case of a discount), and 1.1 (in case of a malus).

These strategies are encapsulated within a class called CostCalculator which acts as a separate component that performs the calculations.

In the bike ride, since we have access to the user and starting and ending locations, we can select the cost strategy and discount strategy to be applied according to the following:

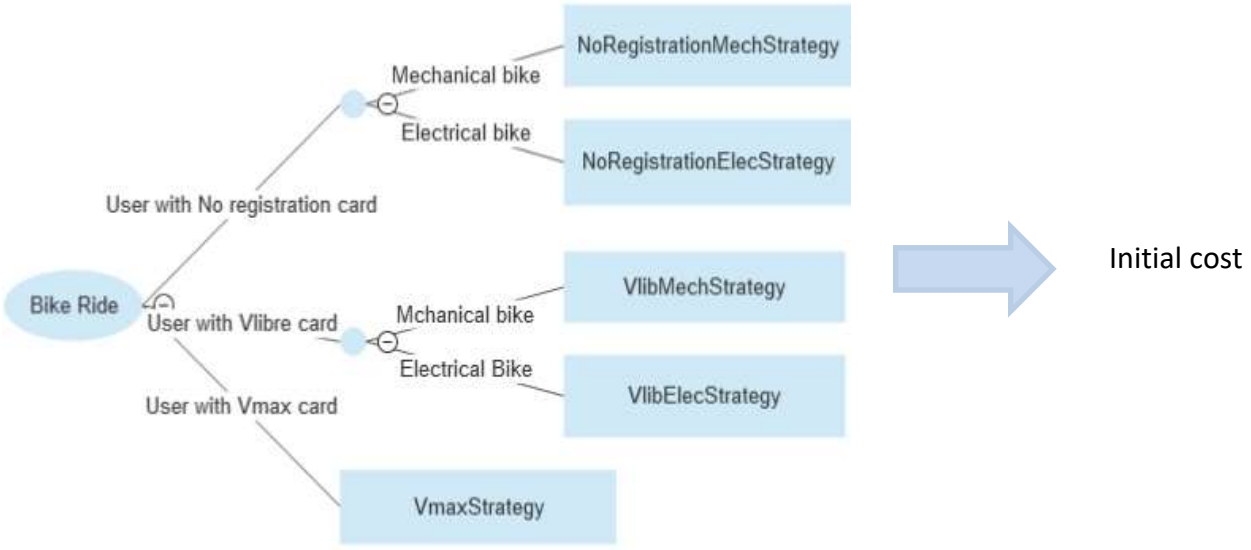


Figure 1

Start \ End	Docking Station	Road
Docking Station	Factor = 1 NoDiscount	Factor = 1.1 LeaveOnRoad
Road	Factor = 0.9 ReturnedToStation	

Figure 2

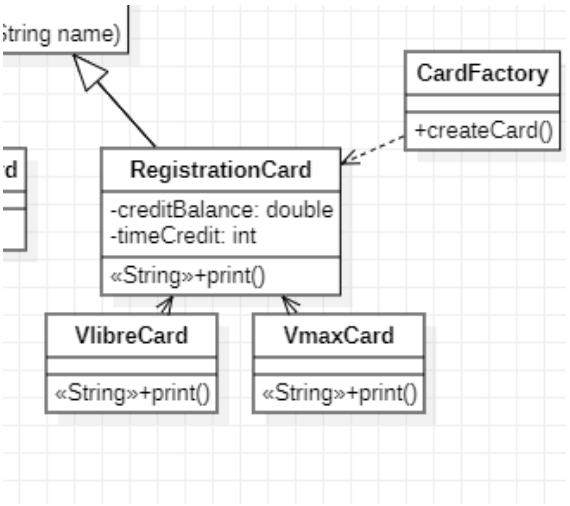
$$final\ cost = factor * initial\ cost$$

Simple Factory Pattern

Card Creation

We used the simple factory pattern in which we have the RegistrationCard as an abstract class containing a print() method to display the card info, and two concrete classes VlibreCard and VmaxCard overriding this method. Using this technique, we ensure an easy extension of the system in case new membership card are available.

To note that we thought of the possibility of using the abstract factory pattern instead of simple factory since we have the card that might be of different types, the registration card can have different types as well. But we realized that it was not efficient in our case since there can be only 2 types of cards (credit and registration) and it makes no sense to consider any further extension in this case.



Stations Sorting

Most used station

The most used station is the one with the highest number of renting and dropping operations. For this purpose, we introduced two attributes to the DockingStation class, of type int, corresponding to the number of rents and drops and are incremented, usually and respectively, when the function rent() and return() is called. To note that not every time a user rents and drops a bike these integers are incremented, since there is a possibility that the bike is rented/dropped from/on road.

To be able to retrieve the most used station, a MostUsedComparator class was created which

implements the comparator interface of type DockingStation.
In this class, we implement the **compare** operation by making it return an integer.
This integer is calculated by:

- Summing the number of rents and returns for a first station s1.
- Summing the number of rents and returns for a second station s2.
- Taking the difference of the two results.

3 scenarios can take place:

- If the integer is positive, then s1 is more used than s2.
- If the integer is equal to zero, then s1 and s2 are equally used.
- If the integer is negative, then s1 is used less than s2.

Least occupied station

The "Least Occupied Station" feature involves sorting stations based on the difference between the number of dropping and renting operations. This difference serves as an indicator of a station's level of occupation.
As in the previous section, a LeastOccupiedComparator was created which implements the comparator interface of type DockingStation. The compare operation is implemented, and an integer is returned corresponding to the indicator.

CLUI Commands

Objective

In this section, we will tackle the realization of a command-line user interface for the developed application.
The goal is to allow the user to interact with the system using some specific commands previously implemented in the code. For this purpose, we created a **MyVelibCLUI** class responsible for implementing and processing each of the commands supported by our application.

Commands

Our system supports the following commands:

Syntax	Arguments	Data Types
setup	<networkName> <nstations><nslots> <s> <nbikes>	<String> <int> <int> <int> <int>
addUser	<username> <cardType> <networkName>	<String> <String> <String>
offline	<networkName> <stationID>	<String> <int>
online	<networkName> <stationID>	<String> <int>
rentBike	<userID> <stationID>	<int> <int>
returnBike	<userID> <stationID> <duration>	<int> <int> <int>
displayStation	<networkName> <stationID>	<String> <int>
displayUser	<networkName> <userID>	<String> <int>
sortStation	<networkName> <sortpolicy>	<String> <String>
display	<networkName>	<String>

help	none	
runtest	<fileName>	<String>

To note that the duration in the returnBike command is in minutes, and we chose it to be an integer.

- The help command displays all possible commands.
- The runtest command run a test-scenario file.
- The description of all the remaining commands can be found in the requirements of the project section 3.1.

Command Processing

Brief Working Principle

For the system to be able to process the command provided by the user, it considers each line a separate command. It scans it and splits the obtained string into an array of substrings based on a specified delimiter: the space character, therefore the command and the arguments should be separated by a space.
The obtained result is processed as follows:

- Check whether the command name is valid or not.
- Check whether the command corresponds to the provided number of arguments or not.
- Convert the arguments from String to their expected data type.

Try – Catch

To note that the Try-Catch comes into play in this section to verify the validity of the data type of the arguments. For instance, if the expected data type of an argument is an integer whereas the user provided a string of characters, the Integer.ParseInt will throw an exception that can be handled in the catch block.

Test Scenarios

Overview

A text file called test-scenario file is created containing a collection of CLUI commands that can be executed to reproduce a specific test scenario. This includes creating a network, setting it up, adding users and stations with parking slots, etc....
The goal of test scenarios is to validate and verify the functionality and behavior of our application under different conditions. They help ensure that the system behaves as expected and meets the specified requirements.

Undergone Tests

testScenario1.txt

The goal of this test is to make sure that:

- The user cannot return a bike before renting one.
- Cannot rent two bikes at a time.

setup network1 3 10 5 20	network1 having 3 stations of 10 slots each, 20 bikes
addUser user1 none network1	user1 added to network1 (id = 0 since first user)
0 0	User location updated
returnBike 0 0 10	user1 tries to return a bike before renting one
rentBike 0 0	user1 rents a bike from station of id = 0
rentBike 0 0	user1 tries to rent another bike
returnBike 0 0 10	user1 returns the bike to station 0 after 10 min
rentBike 0 0	user1 rents another bike
returnBike 0 1 10	user1 returns the bike

testScenario2.txt

The goal of this test is to make sure that:

- Users cannot rent a bike from an offline station.
- Users cannot return a bike to an offline station.

setup network1 2 10 5 10	network1 created having two stations
addUser user1 none network1	user1 added to network1
0 0	
addUser user2 none network1	user2 added to network1
0 0	
displayStation network1 0	Display of the first station of network1 -> Online
rentBike 0 0	User1 rents a bike from the first station
offline network1 0	The first station is made offline
returnBike 0 0 10	user1 tries to return the bike to the offline station
rentBike 1 0	uer2 tries to rent a bike from the offline station
online network1 0	The offline station is now back online
returnBike 0 0 10	user1 returns the bike to the station
rentBike 1 0	user2 rents a bike from the station
displayStation network1 0	Display of the first station of network1 -> back Online

testScenario3.txt

The goal of this test is to verify that:

- A system cannot have a number of bikes > number of parking slots.
- Users cannot rent a bike from an empty station.

setup network1 1 3 5 6	Setting up network with nb of bikes 6 > nb of slots 1*3
setup network1 2 2 5 6	Setting up network with nb of bikes 6 > nb of slots 2*2
setup network1 1 10 5 2	network1 created with 1 station, 10 parking slots and 2 bikes
addUser user1 none network1	user1 added to network1
0 0	

addUser user2 none network1	ser2 added to network1
0 0	
addUser user3 none network1	user3 added to network 1
0 0	
rentBike 0 0	user1 rents a bike from the station
rentBike 1 0	user2 rents a bike from the station
rentBike 2 0	user3 tries to rent a bike from the empty station
returnBike 0 0 10	User1 return the bike to the station

testScenario4.txt

The aim of this test is to:

- Verify the validity of commands, their number of arguments, and their data types.
- Verify that we always check for the existence of users, stations, networks, and slots before using them.

setuppp network1 1 2 10 20	No setuppp command
setup network1 1 2 20	Setup requires 5 arguments and not 4
setup network1 1 2 abcd 20	The side length should be an integer and not a string
setup network1 2 1 10 2	network1 created with 2 stations, 1 parking slot each, and 2 bikes
addUser user1 none network2	user1 being added to non-existing network
addUser user1 none network1	user1 added to network1
0 0	
addUser user2 none network1	user2 added to network1
0 0	
addUser user3 none network1	user3 added to network1
0 0	
displayStation network2 0	Displaying a station of a network that does not exist
displayStation network1 3	Displaying a non-existing station (only 2 stations in network1)
displayStation network1 0	Displaying first station of network 1
displayUser network1 0	Displaying user1 of network1
display network2	Displaying non-existing network
display network1	Displaying network1
rentBike 0 0	user1 rents a bike from station1
rentBike 1 0	user2 tries to rent a bike from station1 -> no bikes since only 1 slot
returnBike 0 1 10	user1 tries to return the bike to station2 which is full (1 bike in 1 slot)
returnBike 0 0 10	user1 return the bike to station 1

testScenario5.txt

The goal of this part is to make sure the MostUsedComparator and the LeastOccupiedComparator work fine.

This was done in the following way:

- A network1 was created with 3 stations.
- 10 users added to the network1.
- 5 users rented a bike from the first station.
- 3 users rented a bike from the second station.
- 2 users rented a bike from the third station.
- The 3 stations were sorted using the least occupied policy. Expected result: station 1, 2, then 3.
- 4 users returned their bikes to the second station.
- The 3 stations were sorted using the most used policy. Expected result: station 2, 1, then 3.

testScenario6.txt

The goal of this test is to verify the cost calculation when dealing with different types of users (with and without registration cards).
This was done in the following way:

- A network1 was created with one station and 10 slots.
- 3 users were added:
  - user1 without a registration card.
  - user2 with vlibre registration card.
  - user3 with vmax registration card.
  - The 3 users rented their bikes from the same location, spent the same duration (100 min), and returned the bikes to the exact same stations.
  - The cost which is calculated in the return method should be different for each user. Expected result: cost for user1 > cost for use2 > cost for user3

Execution

In order to execute the test scenarios above, we use the *runtest <fileName>* command previously described. e.g.:

runtest "testScenario1.txt"

JUnit tests

JUnit is a widely used framework for writing and running automated tests in Java. It provides a simple and standardized way to define and execute unit tests for individual units of code, such as methods or classes.
We used the JUnit tests to debug our code. We tested all the main functions of our program and fixed a lot of bugs. In combination with the test Scenario, the JUnit has been a great tool to build a robust program.

Advantages and Limitations

Advantages

The code we wrote provide a simple implementation of the Vlib system, implementing all the main functionalities and giving the possibility of extensions. The use of Patters respecting the OPEN-CLOSE principle give the possibility to add more cost strategies, types of discount and different types of cards.

Limitations

As said the code wrote is simple and this means that some functionalities are missing e.g. for now we have only one network.

Possible improvements

We can add the possibility to have more networks, allowing each network to manage a different zone.
We can add a GUI, to give the user a better experience while using the software.

Tasks Distribution

Distribution of the work concerning the main classes

	Design	Code	JUnit test
Card classes	Marco, Layal	Marco	Marco
User	Marco, Layal	Layal, Marco	Layal
MyVelib	Marco, Layal	Marco	Layal
RidePlanner	Marco, Layal	Layal	Marco
Cost Calculator	Marco, Layal	Layal	Layal
Ride	Marco, Layal	Layal	Marco
Location	Marco, Layal	Marco	Marco

The report was written by both parties.

Conclusion

The goal of this project was to develop a Java application called myVelib for a bike sharing system, similar to popular systems like Velib in Paris. The project allowed us to refine our Java programming skills, implement various types of Strategy Patterns, and develop a complex program that required debugging and JUnit testing. The project demanded a lot of work, but we are satisfied with the final result, and we hope that you will be too.