Project work
`myVelib` - a bike sharing system (v1)

CentraleSupélec

Hand-out: May 06, 2020
Due : **June 3, 2020**
Programming Language: Java

1 Overview

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 kind of bicycles (mechanical and electrically assisted), the users (which may posses a registration card), the maintenance crew (responsible for collecting/replacing broken down bicycles), etc. Based on the systems requirements (see Section 2) you are required to develop a Java framework, hereafter called `myVelib` (see details about naming of the project under Eclipse in Section 6.1) , for managing bike sharing.
The project consists of two parts:

Part 1: `myVelib` **core**: design and development of the core Java infrastructure for the `myVelib` system (based on requirements given in Section 2)..

Part 2: `myVelib` **user-interface**: design and development of a user-interface for the `myVelib` system (see Section 3).

2 System description and requirements

You are required to develop a suitable JAVA design of the core classes for the `myVelib` system based on the characteristics described in this section.

2.1 Components of the `myVelib` system

- **Station**: a station is where bicycles can be rented and dropped. It consists of a number of parking slots where bikes are stored and of a terminal which users can

interact with in order for renting bicycles. Parking slots can be be occupied by a bicycle, free or out-of-order. A station is located in a specific place specified by GPS coordinates. A station can be on service or offline: if it is offline all of its parking bays as well as the terminal cannot be used. There exist two types of stations, a "standard" type, and a "plus" type. When a user who hold a Velib card drops a bicycle to a "plus" station it earns 5 minutes credits in its time balance. Each station has a unique numerical ID and so each parking slot (within a station) has a unique numerical ID.

- **Bicycle**: there exists two kind of bicycles, mechanical and electrical. Each bicycle has a unique numerical ID.
- **User**: has a name, a unique numerical ID, a geographical position (GPS coordinates) a credit card and might have a registration card. In case a user holds a card she has also a time-credit balance (expressed in minutes) representing the credit gained by returning bicycles to "plus" stations. The time credit is used to compute the actual cost of a bike ride. The user has also a balance of total charges representing the total amount of money she has been charged for for using bicycle of the `myVelib` system.
- **Cards**: there are two kinds of registration cards: `Vlibre` and `Vmax`

2.1.1 Costs for using bicycles

On returning of a velib to a station the system will (automatically) compute the corresponding cost of the ride based on the ride duration (in minutes), the kind of bike, and on the type of card a user has. Specifically:

- if a user has no card the cost is 1 Euro per hour (for mechanical bikes) and 2 Euro per hour (for electrical bikes)
- if a user has a `Vlibre` card the cost is: 0 Euro for the first hour then 1E per each successive hour (for mechanical) and 1 Euro for the first hour then 2E per each successive hour (for electrical bikes). If a ride lasts longer than 60min the actual time balance exceeding 60min is computed by deducing from the user's time-credit (if any). For example a user's time-credit is 20min and the ride lasts 75min, then the user won't be charged any extra min (beyond 1h) but it's time-credit will be updated to 5min (deducing the 15min in excess to the free hour). On the other hand if the ride lasts 135min, the user will be charged for 135-20 = 115min, hence for the 55min in excess of an hour, and so on.
- if a user has `Vmax` card the cost is: 0 Euro for the first hour then 1E per each successive hour (independently of the kind of bike)

2.2 Rides planning

The `myVelib` system must be equipped with a functionality that helps users to plan a ride from a starting location to a destination location. Given the starting and destination GPS coordinates the ride planning functionality will identify the "optimal" start and end stations from/to where the bike should be taken/dropped according to the following criteria:

- the start, respectively the end, station, for a ride should be as close as possible to the starting, respectively to the destination, location of the ride.
- the start station should have one bike of the desired kind (electrical or mechanical) available for renting
- the end station should have at least one free parking slot.

Bonus points: if you apply an open-close design for realising the ride planning functionality you will gain some bonus points. Furthermore if you equip your design with any of the optional ride-planning policies described below you will also gain some additional points.

Optional ride-planning policies: the following ride-planning policies are to be considered as optional (you can, if you want, let your `myVelib` design support them however they are not mandatory).

- **avoid "plus" stations**: like minimal walking distance but return station cannot be a "plus" station
- **prefer "plus" stations**: with this policy the return station should be a "plus" station (given a "plus" station no further away than 10% of the distance of the closest station to the destination location exists). If no such a "plus" station exists then this policy behaves normally (as a minimal walking distance).
- **preservation of uniformity of bicycles distribution amongst stations**: with this policy the choice of the source and destination station is affected by the number of available bikes (at source station) and free slots (at destination). Specifically, let s_0 be the closest station to the starting location with at least one available bike of the wanted kind, and s_d be the station closest to the destination location with at least one free parking slot. Then if a station s'_0 whose distance from the starting location is no more than 105% the distance of s_0 from the start location has a larger number of bikes (of the wanted kind) than those available at s_0 it should be selected in place of s_0. Similarly if a station s'_d (whose distance from the destination location is at most 105% of the distance of s_d from the destination location) has a larger number of free parking slots than s_d it should be selected as the destination station in place of s_d.

2.3 Rental and returning of a bicycle

To rent a bicycle a user must get to one station, identify herself (either through a velib-card or through a credit-card) and pick up one of the available bikes. A user can only rent at most one bicycle (i.e. if she has a bicycle and has not yet returned it, she cannot rent a second one). To return a bicycle a user must park it to a free (and on-duty) parking bay of some station. When the bike is returned the cost for the ride is computed and user is automatically charged (if a charge applies).

2.4 Computing statistics and sorting of stations

The `myVelib` system should support the following functionalities for computing relevant statistics:

- **User balance**: should allow to retrieve, for any user of the `myVelib` system, the number of rides, the total time spent on a bike, the total amount of charges for all rides performed by a user, as well as the time-credit earned by a user
- **Station balance**: should allow to retrieve the total number of rents operation, as well as of return operations performed on the station. It should also allow to compute the average rate of occupation of a given station over a given time window $[t_s, t_e]$ of width $\Delta = t_e - t_s$, where the rate of occupation of a station is given by:

$$\frac{\sum_{i=1}^{N} t_i^{\Delta}}{\Delta \cdot N}$$

  where N is the number of parking slots in the station, t_i^{Δ} is the time the i-th slot has being occupied during the time window $[t_s, t_e]$. Notice that in the computation of occupation rate of a station if a parking slot is out-of-service then it should be accounted as being occupied (independently of whether it holds a bicycle or not).

Furthermore `myVelib` should support different policies for **sorting stations** including those based on the following criteria:

- **most used station**: stations are sorted w.r.t. the total number of renting + dropping operations
- **least occupied station**: stations are sorted w.r.t. the rate of occupation (ratio between free time over occupied time of parking bays). This allows for figure out policy to increase the use of less occupied stations (for example by electing the least occupied stations to the "plus" category so to attract users to drop bikes).

Remark (an OPEN-CLOSE solution). Your design should match as much as possible the open-close principle. Using of design patterns should be properly documented in the project report explicitly describing to to fulfil which requirement of the `myVelib` system a design pattern has been applied.

2.5 Use case scenario

In order to further guide you in the process of designing/developing the core infrastructure for the `myVelib` system you should take into account the following use case scenario which describe a few examples of realistic interactions between the user of the `myVelib` system and the `myVelib` framework.

Setting up of `myVelib`

1. the user sets up an instance of the `myVelib` system: he/she create a `myVelib` with N stations, summing up to a total of M parking slots, the 70% of which is occupied by available bicycles (30% of which are electrical the rest mechanical). The N stations should be placed randomly (in a uniform manner) over a square surface of area 10×10 square kilometres (roughly the surface of Paris *intramuros)*.

2. *the user add some user to the `myVelib` system some of which are card holder, some not.*

Rental of a bike

1. a user (card holder or not) rent a bike of a given type from a given station at a given moment in time

2. the user returns the bike to another station after a given duration (expressed in minutes).

3. the user get charged for the corresponding amount (possibly 0) and possibly receive a time-credit (possibly 0)

Visualisation of user, station and system state

1. a manager (or a user) of `myVelib` request to see the current state of a user with given ID

2. the system displays a report of the requested user (including its balance, time credit, number of rides, etc).

3. a manager (or a user) of `myVelib` request to see the current state of a station with given ID
4. the system displays a report of the requested station (including num. of free/occupied slots, current state, its occupation etc)
5. a manager (or a user) of `myVelib` request to see the current state of a the entire system
6. the system displays a summary report of the system (including list of online/offline stations, list of users of `myVelib` , etc.)

Simulation of a planning ride

1. a user at a given position require a ride planning to reach a destination position
2. the user receives the source and destination stations
3. the user retrieve a bicycle from the source station of the planned ride at a given instant of time
4. the user that is holding a bicycle returns it (in a given free slot) at a given station at a given instant of time.

Computation of statistics

1. the `myVelib` system stores relevant data in form of records representing N rental bike simulations (by different users, on different stations, and with different duration)
2. the statistics (computed w.r.t. the rental records added in previous step) for each user are displayed
3. the statistics (computed w.r.t. the rental records added in previous step) for each station are displayed
4. stations are displayed sorted w.r.t. the most used station (first)
5. stations are displayed sorted w.r.t. the least occupied station (first)

3 Part 2: myVelib user interface

The part 2 of the project is about realising a user interface for the myVelib-app. The user interface consists a command-line user interface (CLUI). For insights about CLUI see https://en.wikipedia.org/wiki/Command-line_interface and some examples in Java https://dzone.com/articles/java-command-line-interfaces-part-29-do-it-yoursel

3.1 myVelib command line user interface

The command line interpreter provides the user with a (linux-style) terminal like environment to enter commands to interact with the myVelib core. A command consists of the command-name followed by a blank-separated list of (string) arguments:

```
command-name <arg1> <arg2> ...  <argN>
```

a command without argument is denoted command-name <>.

For example the myVelib CLUI command for setting up a myVelib system consisting of 10 stations each of which has 10 parking bays and such that the initial population of bicycle is 75% of the total number of parking bays in the system would be as follows:

```
setup 10 10 0.75
```

The list of commands supported by the myVelib CLUI must include the following ones:

- setup <velibnetworkName>: to create a myVelib network with given name and consisting of 10 stations each of which has 10 parking slots and such that stations are arranged on a square grid whose of side 4km and initially populated with a 75% bikes randomly distributed over the 10 stations
- setup <name> <nstations> <nslots> <s> <nbikes>: to create a myVelib network with given name and consisting of nstations stations each of which has nslots parking slots and such that stations are arranged in as uniform as possible manner over an area you may assume either being circular of radium s or squared of side s (please document what kind of area your implementation of this command takes into account and how stations are distributed over it).Furthermore the network should be initially populated with a nbikes bikes randomly distributed over the nstations stations
- addUser <userName,cardType, velibnetworkName> : to add a user with name userName and card cardType (i.e. ''none'' if the user has no card) to a myVelib network velibnetworkName

- `offline <velibnetworkName, stationID>` : to put offline the station `stationID` of the `myVelib` network `velibnetworkName`
- `online <velibnetworkName, stationID>` : to put online the station `stationID` of the `myVelib` network `velibnetworkName`
- `rentBike <userID, stationID>` : to let the user `userID` renting a bike from station `stationID` (if no bikes are available should behave accordingly)
- `returnBike <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). This command should display the cost of the rent
- `displayStation<velibnetworkName, stationID>` : to display the statistics (as of Section 2.4) of station `stationID` of a `myVelib` network `velibnetwork`.
- `displayUser<velibnetworkName, userID>` : to display the statistics (as of Section 2.4) of user `userID` of a `myVelib` network `velibnetwork`.
- `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`.
- `display <velibnetworkName>`: to display the entire status (stations, parking bays, users) of an a `myVelib` network `velibnetworkName`.
- to be completed ...

It should be possible to write those commands on the CLUI and to run the commands in an interactive way: the program read the commands from `testScenarioN.txt` (see Section 4.2), pass them on to the CLUI, and store the corresponding output to `testScenarioNoutput.txt`.

Error messages and CLUI. The CLUI must handle all possible types of errors, i.e. syntax errors while typing in a command, and misuse errors, like for example trying to rent a bike from a station which is offline or that has no available bikes.

4 Project testing (mandatory)

In order to evaluate your implementations we (the Testers of your project) require you (the Developers) to equip your projects with both standard **Junit tests** (for each class) and a **test scenario**, described below. Both JUnit tests and the test scenario are mandatory parts of your project realisations, as we (the Testers) will resort to both of them to test your implementations.

4.1 Junit tests

Each class in your project must contain JUnit tests for the most significant methods (i.e. excluded *getters* and *setters*).
Hint: if you follow a Test Driven Development approach you will end up naturally having all JUnit tests for all of your classes.

4.2 Test scenario

In order to test your solution you are required to include in the project

- one initial configuration file (called `my_velib.ini`), automatically loaded at starting of the system,
- at least one test-scenario file (called `testScenario1.txt`).

Configuration file. An initial configuration file must ensure that, at startup (after loading this file) the system contains at leas the "standard" setup corresponding to the default version of the CLUI command `setup`.

Test-scenarion file. A test-scenario file contains a number of CLUI commands whose execution allows for reproducing a given test scenario, typically setting up a given configuration of the `myvelib` system (i.e. creation of some velib network, adding of some users, simulation of some rental/returning of bikes, simulation of planning of a ride, computation of statistics for the stations and the users, etc.). You may include several test-scenario files (e.g. `testScenario1.txt`, `testScenario2.txt`, ...). For each test-scenario file you provide us with you MUST include a description of its content (what does it test?) in the report. We are going to run each test-scenario file through the `runtest` command of the CLUI (see CLUI commands above):

```
runtest  testScenario1.txt
```

5 A roadmap to design and implementation

We briefly remind you the roadmap you should follow to develop your solution to the `myVelib` system.

1. carefully read and analyse the `myVelib` requirements. From the requirements start identifying the relevant classes, the relationship between classes (inheritance, composition), whether a class should be abstract, should be an interface or a generics. At the same time try and see if you can map any of the specification requirement into some design pattern. does a requirement map in one problem corresponding to a design pattern seen during the course?

2. sketch a first UML class diagram for the classes you identified from step 1 (again consider applying design patterns): **DO NOT START CODING BEFORE CAREFULLY WORKING OUT A UML CLASS DIAGRAM DESIGN**.
3. progressively fill in each identified class with the necessary attributes and necessary methods (signature) and updating the class diagrams (possibly modifying/adding new relationship between classes)
4. once you are quite confident about the structure of a class you identified in steps 1,2 and 3 start coding it by implementing each attribute and each method.
5. test each class you have been coding by developing unit-tests (JUnit). This can be done in reversed order if you adopt Test Driven Development

6 Deadlines and submitting instructions

The project work is to be done in teams of 2 students. Each team must hand in (in the dedicated Project Assignment on Edunao) the following material:

- an Eclipse project (exported into a .zip file) containing the code for your implementation (see details below for how to name the project and what the project should contain)
- a written report on the design and implementation of the corresponding part

6.1 Project naming

The Eclipse projects you submit must be named as follows:

- `Group`N`_MyVelib_student1Name_student2Name` exported into file `Group`N`_MyVelib_student1Name_student2Name.zip`

thus, if Group1 is formed by students Alan Turing and John Von Neumann, they will have to create, on Eclipse, a project called `Group1_MyVelib_Turing_VonNeumann`, which they will export into a file named, `Group1_MyVelib_Turing_VonNeumann.zip`.

6.2 Eclipse project content

Each Eclipse project should contain all relevant files and directories, that is:

- .java files logically arranged in dedicated packages
- a "test" package containing all relevant junit tests

- a "doc" folder containing the javadoc generated documentation for the entire project. The generation of javadoc is essential for the evaluation of the project. Think about writing javadoc comments as soon as you start writing your code: at the end is not useful for you and it is time consuming.
- a "model" folder containing the papyrus UML class diagram for the project The UML class diagram must be attached to the project as an image (we encourage you to use papyrus but you can design and produce this file using any tool you like).
- an "eval" folder containing the following:
  - **MANDATORY**: an initial configuration file "my_velib.ini", automatically loaded at starting, which contains a list of users and meals/dishes that ensure the system is not empty after startup.
  - **MANDATORY**: at least a file "testNinput.txt" (the one that is described in this project) which contains a sequence of CLUI commands that allows to test the main functionalities of the `myVelib`-app following the test cases described in Section 4

IMPORTANT REMARK: it is the students responsibility to ensure that the project is correctly named as described above and that it is correctly exported into a .zip archive that correctly works once is imported back into Eclipse. A PROJECT THAT IS NOT PROPERLY NAMED OR THAT CANNOT BE STRAIGHTFORWARDLY IMPORTED IN ECLIPSE WILL NOT BE EVALUATED (HINT: do verify that export and re-import works correctly for your project work before submitting it).

7 Writing the report (team work)

You also have to write a final report file describing your solution. The report **must** include a detailed description of your project and must comprehend the following points:

- main characteristics
- design decisions
- used design patterns: you should clearly describe which pattern you used to solve which problem
- advantages/limitations of your solution
- **MANDATORY: the test scenario description to your advantage**

- **MANDATORY: how to test your realisation to your advantage** : guide the hand of the corrector with the text to be pasted into the console to launch the test scenario(s) with runTest
- how the workload has been split and its realisation (who did what in a commented table with mandatory columns design — code — JUnit test and lines class or task)
- etc.

The quality of the report is an important aspect of the project's mark, thus it is warmly recommended to write a quality report. The report should also describe how the work has been divided into task, and how the various tasks have been allocated between the two members of a group (e.g. task1 of `myVelib`-core → responsible: Arnault, task2 of `myVelib`-core →responsible: Paolo, etc.). Also the writing of the report should be fairly split between group's members with each member taking care of writing about the tasks he/she is responsible for. In a good report there is no code listing but some code, or better, UML Class diagram, can be inserted in order to comment special algorithms or issues (do not abuse of this for code).

8 Project grading

The project is graded on a total of 100 points for mandatory features + 20 bonus points (bonus points will complement points lost elsewhere). The guidelines of marks breakdown is given below:

- `myVelib` Core functionalities (max 40 point)
- `myVelib` CLUI functionalities (max 20 points)
- JUnit tests (max 15 points): `myVelib` core (9pt), `myVelib` CLUI (6pt)
- UML (max 10 points): `myVelib` core (8pt), `myVelib` CLUI (2pt)
- Final report (between -15 and +15 points depending on quality)

Each part of the solution (i.e. `myVelib` CORE, `myVelib` CLUI) will be evaluated according to two basic criteria:

- requirements coverage: how much the code meets the given project requirements (described in Section 2)
- Code quality: how much the code meets the basic principles of OOP and software design seen throughout the course (i.e. object oriented design, separation of concerns, code flexibility, application of design patterns, etc.)

- the quality of the report describing each part of the project

The grade will be determined based on the project as final implementation and the final report as submitted.

Remark: the above grading scheme is meant to give an idea of the relative importance of each part of the project. It will be used as a guideline throughout the marking but it does not constitute an obligation the marker must stick to. The marker has the right to adapt the marking criteria in any way he/she feels like it is more convenient in order to account for specific aspects encountered while marking a particular solution.

9 General Remarks

While working out your solution be aware of the following relevant points:

- Design your application in a modular way to support separation of concerns. For example, `myVelib` core should not depend on the `myVelib` command-line user interpreter.
- The system should be robust with respect to incorrect user input. For example, when a user tries to import files from a nonexistent directory, the `myVelib` should not crash.
- **application of design patterns**: flexible design solutions must be applied whenever appropriate, and missing to apply them will affect the evaluation of a project (i.e., a working solution which doesn't not employ patterns will get points deducted). Thus, for example, whenever appropriate decoupling approaches (e.g. visitor pattern) for the implementation of specific functionalities that concern `myVelib` must be applied.

10 Questions

Got a question? Ask away by email:

Paolo Ballarini: paolo.ballarini@centralesupelec.fr
Arnault Lapitre: arnault.lapitre@centralesupelec.fr

or post it on the Forum page on Edunao.