IS1220 - Final Project
`myUber`: a car-ride sharing system (v4)

CentraleSupélec

Hand-out: October 8, 2018
Due Part 1: ~~November 12, 2018~~ **November 19, 2018**
Due Part 2: ~~November 26, 2018~~ **December 3, 2018**
Programming Language: Java

1 Overview

Uber is a ride-sharing system which allows inhabitants of a metropolitan area to get a ride on a car driven by a professional driver. The Uber system consists of several parts including: the cars (which circulate on the metropolitan area), the drivers, the customers (the persons registered to the system and that can book a car ride). Based on the systems requirements (given in Section 2) you are required to develop a Java framework, called `myUber` for representing and managing the Uber ride-sharing system.
The project consists of two parts:

Part 1: `myUber` **core**: design and development of the core Java infrastructure for the `myUber` system.

Part 2: `myUber` **user-interface and simulator**: design and development of a user-interface and simulator for the `myUber` system.

2 System description and requirements

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

2.1 Components of the `myUber` system

- **Car**: a car is used by a driver to take customers (passengers) around for a ride. A car can be owned by several drivers that can take turn at the steering wheel. There

are at least 3 kind of cars: `Standard` (standard cars), `Berline` (i.e. luxury cars) and `Van` (that can carry up to 6 passengers). A car has a fixed number of seats to carry passengers (4 seats for standard and berline cars 6 for Van cars) and a unique alphanumerical ID with the following form: `Standard`N, `Berline`N and `Van`N where N is a unique number for the corresponding kind of car. A car has GPS coordinate indicating its current position. The commissioner of `myUber` that you are in charge to develop requires that new kind of cars should be added easily in the system if required.

- **Driver**: a `myUber` car driver has a name, a surname and a unique numerical ID. His/her state can be: offline (i.e. not working), on-duty (i.e. ready to take a ride), on-a-ride (driving a customer somewhere) or off-duty (i.e. working but on a pause, that is, not driving and not available to take a ride).
- **Customer**: a `myUber` customer has a name, a surname, a unique numerical ID, a geographical position (GPS coordinates), a credit card number, and a message box where incoming messages from `myUber` are stored.
- **Rides**: `myUber` system shall offer registered customers with a service for booking a ride according to the following specifications:

  s1: a customer can book a ride from one starting point (a physical position) to a destination point.

  s2: `myUber` shall offer the following kind of rides for booking: An `UberX` ride which is an economy, not-shared, ride on a "standard" car, a `UberBlack` ride which is a ride on a luxury car, a `UberVan` which is non-shared ride on a car which can carry up to 6 persons and a `UberPool` ride for a ride on a standard car, shared with other passengers with different destinations (this is a the cheapest type of ride).

  s3: `myUber` should be flexible w.r.t. to the kind of rides offered (i.e. the offered rides should be modifiable easily).

  s4: a ride status can be in either of the following states: `unconfirmed` (the ride request has been sent to `myUber` but waiting to be allocated to a driver), `confirmed` (the ride request has been accepted by a driver), `ongoing` (the customer has boarded a car and the ride is taking place), `canceled` (the ride request has been cancelled by the user before before boarding the car), `completed` (the ride has been completed),

2.2 Booking of a car ride

The `myUber` framework shall allow customers to book a ride and car drivers to accept a booking request according with the following use case scenario:

- a customer shall enter a given destination (GPS) in the `myUber` system
- `myUber` then sends back a price list to the requesting customer indicating the cost of the ride (details in Section 2.3) for each kind of ride (`UberX`, `UberBlack`, `UberVan`, `UberPool`)
- the customer then select the kind of ride he/she wants to book.
- `myUber` then looks up for nearby available cars that are compatible with the kind of ride selected by the customer and forward the ride request to the drivers up until the booking request is accepted by a driver.
- a car driver may accept an incoming booking request if 1) the ride type is compatible with the car; 2) the car has sufficiently many free seatss to allocate the booking request (relevant only for `UperPool` rides).
- once a booking request is accepted by the driver, it is registered in the `myUber` "book of rides". The entries of such book must indicate, for each ride, the identity of the driver, that of the car, that of the booking customer, the trajectory (starting and ending point) the length of the ride, as well duration (pick up time and arrival time). The duration should be given in function of the traffic at the moment the ride has been booked and by using the average speed values given in Table 1.

	low-traffic	medium-traffic	heavy-traffic
`average speed`	15 Km/h	7.5 Km/h	3 Km/h

Table 1: Average speed of a ride for different traffic conditions.

- on termination of a ride the customer may send to `myUber` an evaluation mark (from 1 to 5 stars) to express its level of satisfaction w.r.t. to the driver and the quality of the ride (1 star meaning very unsatisfied, 5 stars meaning very satisfied).

2.2.1 On `uberPool` rides

The behaviour of `myUber` in case of `uberPool` ride is described below:

- when a customer selects the `uberPool` kind of ride its request is added to a *poolRequests* list which contains the currently received (and not yet served) requests for a `uberPool` ride. Such list of *poolRequests* is passed on to drivers who may accept to serve the requests in the *poolRequests*. A *poolRequests* may contains a maximum of 3 ride requests (from 3 different customers).

- a sole `uberPool` request is held on as long as at least another `uberPool` request has arrived.
- `myUber` looks up for nearby available cars that may serve the *poolRequests*. To establish in which order "nearby" car should receive proposal to serve a *poolRequest* `myUber` uses the following criteria to assign a "cost" for a car to serve a *poolRequests*: the "cost" for car c to serve rides in a *poolRequests* is given by the sum of the distances in the minimal pick-up/drop-off trajectory (see description below) for the rides in the *poolRequests*. Thus `myUber` will forward a *poolRequests* to cars following the cost ordering of each to serve the *poolRequests*.
- minimal pick-up/drop-off trajectory: let c be position of the car and $p1$, $p2$, $p3$, respectively $d1, d2, d3$, the pickup, respectively, drop-off positions for 3 customers in a *poolRequests*. Let assume that $p1$ is closest to c (than $p2, p3$), that $p3$ is closest to $p1$ than $p2$, that $d2$ is closest to $p2$ than $d1, d3$, that $d1$ is closest to $d2$ than $d3$, then the minimal pick-up/drop-off trajectory is given by the sequence $p1$-$p3$-$p2$-$d2$-$d1$-$d3$ and its "cost" (w.r.t. car c) is the sum of the distances between points in the sequence plus the distance of the car from the first point in the sequence.

2.3 Costs of a car ride

The `myUber` framework shall be equipped with a functionality for computing the a ride fare. This should include a **fare computation function** which depends on several factors, including: the kind of ride booked (i.e. `UberX`, `UberVan`, `UberBlack`, `UberPool`), the distance of the ride, the traffic condition at the moment the ride is booked (assuming there are three possible traffic status: `light`, `medium`, `heavy`). The specifications for the **fare computation function** are as follows:

s4: the fare of a ride is given by the product of a *basicRate* times the length of the ride time a factor that account for the traffic condition:

$$rideFare(rideType, length, traffic) = basicRate(rideType, length) \cdot length \cdot trafficRate(rideType) \quad (1)$$

where $basicRate(rideType, length)$ is given in Table 2 and the $trafficRate(rideType)$ is given in Table 3.

kind of ride	$length < 5km$	$5 < length < 10$	$10 < length < 20$	$length > 20$
`uberX`	3.3	4.2	1.91	1.5
`uberBlack`	6.2	5.5	3.25	2.6
`uberPool`	2.4	3	1.3	1.1
`uberVan`	6.2	7.7	3.25	2.6

Table 2: Basic rates in Euro/Km in function of the ride length (l given in Km).

kind of ride	low-traffic	medium-traffic	heavy-traffic
`uberX`	1	1.1	1.5
`uberBlack`	1	1.3	1.6
`uberPool`	1	1.1	1.2
`uberVan`	1	1.5	1.8

Table 3: Traffic factor for computing the ride fare.

Remark: The commissioner of `myUber` requires that the system should allow for extending the computation cost functions available for `myUber`.

2.3.1 Traffic state

The `myUber` systemn should account for 3 traffic states *low-traffic*, *mid-traffic*, *high-traffic* (whose corresponding average speed is given in Table 1). The current state of the traffic depends on the current time of the day and obeys the probability distributions given in Table 4:

time interval	low-traffic	medium-traffic	heavy-traffic
$[22h00, 7h00]$	0.95	0.04	0.01
$[7h00, 11h00]$	0.05	0.20	0.75
$[11h00, 17h00]$	0.15	0.70	0.15
$[17h00, 22h00]$	0.01	0.04	0.95

Table 4: Probability distribution for the traffic state

The probability laws in Table 4 should be used to establish the traffic condition at the moment of reservation of a ride.

2.4 Computing relevant statistics

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

- **Customer balance**: should allow to retrieve the number of rides, the total time spent on a uber car, the total amount of charges payed by a given customer (for all rides performed).
- **Driver balance**: should allow to retrieve the total number of rides a driver performed over a given time interval, as well as the total amount of money cashed by the driver through the rides. It should also allow to compute some key performance indicators (KPI) for a driver, like: i) the *on-duty rate of driving*, i.e. the ratio between the time a driver has been driving around customers and the time it has been on duty; ii) the

rate of activity, i.e. the ratio between the time a driver has spent on-duty or driving over the total time the driver has been in service (i.e. not offline).

- **System balance**: should allow to retrieve the total number of rides for all drivers, as well as the total amount of money cashed in by all drivers

Furthermore `myUber` should be equipped with different policies for **sorting drivers and customer** including those based on the following criteria:

- **most frequent customer**: customers are sorted w.r.t. the total number of completed rides
- **most charged customer**: customers are sorted w.r.t. the total amount of charges for all completed rides
- **least occupied driver**: drivers are sorted w.r.t. their occupation rate (ration between the on-duty time over the total ride time). This allows to potentially setting up ride allocation policies that increase the use of less occupied drivers (for example by allocating booking request to the least occupied drivers first).
- **most appreciated driver**: drivers are sorted w.r.t. their level of appreciations (in terms of feedback received by customers).

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 `myUber` system a design pattern has been applied.

2.5 Use case scenario

Below you find a list of use case scenarios that describe how a user of the `myUber` should interact with the `myUber` system to test its functioning. You should take into account those scenarios while designing/developing the core infrastructure as well as the user interface for `myUber`. In the remainder "user" is referred to the user of the `myUber` system.

Setting up of `myUber`

1. the user sets up an instance of the `myUber` system: he/she creates a `myUber` with N_c cars (of different kind), N_d drivers, N_u customers. An initial configuration (in terms position of cars and customers as well as of the state of the drivers) shall be given as well.

Simulation of a UberX, UberBlack or UberPool car ride

1. a customer at a given position requires a ride to reach a destination position
2. the customer receives the price list for each kind of ride for the desired destination and chose either a UberX, UberBlack or UberPool ride type.
3. the booking request is assigned by `myUber` to a driver (during a given traffic condition)
4. the ride takes place, the customer get charged, the customer expresses its satisfaction mark

Simulation of a UberPool car ride

1. a customer $c1$ at a given position $p1$ requires a ride to reach a destination position $d1$
2. a customer $c2$ at a given position $p2$ requires a ride to reach a destination position $d2$
3. customers $c1$ and $c2$ receive the price list for each kind of ride for their respective desired destination
4. both $c1$ and $c2$ select the UberPool ride.
5. the booking request is presented by `myUber` to drivers (during a given traffic condition) and one driver accept to serve the uberPool call
6. the ride takes place, the customers get charged, the customers express their satisfaction mark

Computation of statistics

1. the `myUber` book of rides is populated with records representing N completed rides (by different customers, on different cars, and with different drivers)
2. the statistics (computed w.r.t. the ride records added in previous step) for each customer are displayed (most frequent customer and most charged customer)
3. the statistics (computed w.r.t. the ride records added in previous step) for each driver are displayed (most appreciated driver, least occupied driver)

3 Part 2: `myUber` user interface

For the second part of the project you have to realise a *user interface* (UI) for the `myUber`-app. The goal of the UI is to provide the user of the `myUber` with a framework to run pertinent use case scenarios. In practice the UI will consists of a **mandatory command-line user interface** (CLUI), and, for the more motivated ones, an optional graphical user interface (GUI), whose implementation will be based on the CLUI (i.e. the GUI can be seen as a layer of software based on the commands provided by the underlying CLUI). Remark: the (non-mandatory) implementation of a GUI, will gain extra points, which can re-enforce points lost elsewhere.

3.1 `myUber` command line user interface

The CLUI is a command interpreter program that provides the user with a (linux-style) terminal like environment to enter commands to interact with the `myUber` core. The CLUI takes as input a specific *command* (see list below) and execute it producing an output. In the remainder we describe commands with the following syntax:

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

where `command-name` is the name of the command and arguments are denoted (as a blank separated list) of items where each argument is denoted in between angular brackets.

Remark: a command without argument is denoted `command-name <>`. Furthermore notice that the angular brackets are only used in this document to distinguish arguments of a CLUI command from the command name, an actual CLUI command is invoked by typing in the command name followed by the list of arguments once the CLUI is launched.

List of mandatory commands. The CLUI must support the following list of commands:

- `init <fileName>.ini`: to create a `myUber` system initialised by the execution of the CLUI commands stored in the file named `fileName.ini` (i.e. the file extension is `.ini` see example below).
  output: the state of the system, including the list of cars (with their relevant information), the list of drivers (with their relevant information) and the list of customers (with their relevant information, see command `displayState<>`).
- `setup <nStandardCars> <nBerlinCars> <nVanCars> <nCustomers>`: to create a `myUber` consisting of `nStadardCars` standard cars , `nBerlinCars` berlin cars, `nVanCars` van cars and `nCustomers` customers. The command also create a driver for each car. Drivers and customers names/surnames are given symbolically by the command (e.g.

`driver1name`, `customer2name` etc). Drivers are initially `offduty`. The initial position of each car and customer is decided randomly imagining they are spread over a square area of size 100, with $(0, 0)$ being the centre of the square.

- `addCustomer <customerName> <customerSurname>`: to add a customer with name `customerName` and surname `customerSurname`.
  output: the list of customers (with their relevant information).

- `addCarDriver <driverName> <driverSurname> <carType>`: to add a driver with given name and surname and a car of type `carType` and and associate the newly created driver to the newly created car (where `carType` is either `standard`, `berline`, `van`). The added driver is initially *offline*
  output: the list of car and the list of drivers (with their relevant information).

- `addDriver <driverName> <driverSurname> <carID>`: to add a driver with given name and surname and to an existing car with ID `carID`.
  output: the list of car and the list of drivers (with their relevant information).

- `setDriverStatus <driverName> <driverSurname> <status>`: to set the status of a driver with given name and surname to a given value `status` (where `status` is either `offduty`, `onduty`, `offline`).
  output: the list of drivers (with their relevant information).

- `moveCar <carID> <xPos> <yPos>`: to set the position of car `carID` to co-ordinates `xPos,yPos`.
  output: the list of car (with their relevant information).

- `moveCustomer <custID> <xPos> <yPos>`: to set the position of customer `custID` to co-ordinates `xPos,yPos`.
  output: the list of customers (with their relevant information).

- `displayState <>` : display summary information of the current state of the system, including the list of cars (with their position), the list of drivers (with their status, their statistics, etc), the list of customers (with their position, their statistics, etc).
  output: the list of car the list of drivers the list of customers (with their relevant information).

- `ask4price <customerID> <destination> <time>` : to let the customer `customerID` to ask for the price of ride to a given `destination` (where `destination` is a GPS coordinate) using `time` as time the price estimation is done (where `time` is either an integer value within $0 \leq$ `time` ≤ 23, or `time`< 0 in which case the current time of execution of the `ask4price` command is considered as the time for the estimation of the price).

output: the list of prices for each kind of ride to reach the given destination at the given time of the day.

- `simRide <customerID> <destination> <time> <rideType> <driverMark>`: to simulate a ride request and ride execution for a given customer (i.e. `customerID`) wanting a ride of a given type (`rideType`) to a given destination (`destination`) at a given time (`time`) and expressing a given evaluation (`driverMark`) for the driver taking the ride.
  Output: a summary message which includes relevant information about the ride execution, such as the identity of the driver and car (driverID, carID) selected by the system of the requested ride, the time of departure and arrival to destination (timeDeparture, timeArrival) the amont the customer is charged with (e.g. costOfRide).

- `simRide_i <customerID> <destination> <time>`: to simulate a ride request and ride execution. This command is the *interactive* version of `simRide`, as the user, has to interactively exchange information with the CLUI in order to chose the the kind of ride, express the evaluation for the driver, etc. The execution of `simRide_i` requires the following input/output steps:

  - **output-step1**: the list of prices for each kind of ride type (`uberX`, `uberBlack`, etc) to reach `destination` (as with command `ask4price`).
  - **input-step2**: the CLUI asks the user to select the kind of ride for `customerID` (by entering the name of the type of ride). On receiving this input the system allocate a car and let the ride takes place.
  - **output-step3**: an output message with summary information about the ride that just took place (e.g. driverID, customerID, carID, rideType, timeDeparture, timeArrival)
  - **input-step3**: the CLUI asks the user to enter the evaluation for the driver with which the sutomer took the ride (such evaluation is stored by the system as part of the driver statistics).
  - **final-output**: the current state of the system (cars, drivers, customers) is displayed at the end of the execution of the `simRide` command.

- `displayDrivers<sortpolicy>` : to display the drivers in the `myUber` system in increasing order w.r.t. to the sorting policy (where `sortpolicy` can be either `mostappreciated`, or `mostoccupied`).
  output: the list of drivers (with their relevant information) sorted according to the chosen policy

- `displayCustomers<sortpolicy>` : to display the customers in the `myUber` system in increasing order w.r.t. to the sorting policy (where `sortpolicy` can be either `mostfrequent`, or `mostcharged`).

output: the list of customers (with their relevant information) sorted according to the chosen policy

- `totalCashed<>` : to display the total amount of money cashed by all drivers in the `myUber` system.
  output: the total amount of money cashed by all drivers.

Other commands: in your implementation of the CLUI you are free to add other commands, apart from those described above. If you do so you are required to carefully describe any extra command (their syntax and informal semantics), together with examples of their application, in the project report.

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.

Example of `.ini` file. An example of commands to be included in a `.ini` file to be passed on as argument to the `setup` command of the CLUI is given below:

```
setup 10 5 5 5 // to create an Uber system with 10 standard cars, 5 berline cars, 5 vans and 5 users
addCustomer Alan Turing // to add a customer named Alan Turing
addDriver James Hunt 3 // to add a driver named James Hunt to car number 3
moveCar 2 25 35 // to move car number 3 to co-ordinates 25 35
moveCustomer 1 20 30 // to move customer number 1 to co-ordinates 20 30
ask4price 1 45 82 // to ask the price of a ride for customer number 1 to destination given by co-
    ordinates 45 82
simRide 1 45 82 -1 uberX 5 // to simulate a ride for customer 1 to go to (45,82) at current time with
    uberX and giving 5 as evaluation to the driver
simRide 2 22 33 -1 uberX 3 // to simulate a ride for customer 2 ..
simRide 3 61 4 -1 uberBlack 4 // to simulate a ride for customer 2 ..
display // to display current state of the system
displayDrivers mostappreciated // to display drivers sorted w.r.t. appreciation
displayDrivers mostoccupied // to display drivers sorted w.r.t. occupation
totalCashed // to display total amount cashed by all drivers
```

3.2 Hints

1. Develop a set of JUnit tests prior to the development, try to work using the Test-Driven-Development (TDD) approach.

2. for GUI related problems it might be useful to have a look to online documentation for Java Swing, for example `http://docs.oracle.com/javase/tutorial/uiswing/dnd/index.html` and in particularly for GUI reactivity `http://docs.oracle.com/javase/tutorial/uiswing/concurrency/index.html`

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

These are standard tests whose goal is simply to test the main functionalities of the various part of the project (i.e. testing of each method of each class). Each class in your project must contain a JUnit test for each method (or at least for the most significant methods). **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 (temporary definition, to be amnded)

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

- one initial configuration file (called `my_uber.ini`), automatically loaded at starting of the system,

- at least one test-scenario file (called `testScenario1.txt`).

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`

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 `myUber` system (i.e. creation of some `myUber` framework, adding of some customers, drivers, cars, simulation of some ride requests, simulation of ride allocation to a driver, computation of statistics for drivers, customers, 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 `myUber` system.

1. carefully read and analyse the `myUber` 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)
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

All project work must be done in teams of 2 people. The project itself is divided in 2 parts, each with its own deadline:

- **Part 1, the `myUber` core**: due ~~November 12, 2018~~ (i.e. $\sim$ 6 weeks work)
- **Part 2, the `myUber` user-interface and simulator**: due ~~November 26, 2018~~ (i.e., $\sim$ 2 weeks work) which can consist in either:

  - a command-line shell: allowing the user to interact with the `myUber` core through a number of pre-defined commands
  - a graphical user interface: allowing the user to interact with the `myUber` core through a GUI

At each deadline you must hand in:

- an Eclipse project 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

Note that the proposed project structure allows for incremental development (you can first work out part 1, and then part 2 which will be based on part 1).

6.1 Project naming

You must hand over your work for both parts of the projects through Claroline (Assignments section) in the form of an Eclipse project (exported into a .zip file). The Eclipse projects you submit must be named as follows:

- `Group`N`_Project_IS1220_part1_student1Name_student2Name` (the project covering part 1), exported into file `Group`N`_Project_IS1220_part1_student1Name_student2Name.zip`
- `Group`N`_Project_IS1220_part2_student1Name_student2Name` (the project covering part 2), exported into file `Group`N`_Project_IS1220_part2_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_IS1220_Project_part1_Turing_VonNeumann`, which they will export into a file named, `Group1_IS1220_Project_part1_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_uber.ini", automatically loaded at starting, which contains a list of drivers, cars, and customers that ensure the system is not empty after startup.
  - **MANDATORY**: at least a file "testScenario1.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 `myUber`-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 `myUber`-core → responsible: Arnault, task2 of `myUber`-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). You can find a reference guide for writing your report at: `http://www.cs.ucl.ac.uk/index.php?id=4060`.

8 Project grading

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

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

Each part of the solution (i.e. `myUber` CORE, `myUber` CLUI, and also `myUber` GUI) 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 by ~~November 26, 2018~~. However, each team has to provide an implementation and a report summarizing the current design and project status at each intermediate deadline (~~November 12, 2018~~). The implementation at the intermediate deadlines has to be runnable (they must compile and run).

IMPORTANT: if a group does not submit part 1 (but only part 2) some points will be deducted from the final mark.

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, `myUber` core should not depend on the `myUber` 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 `myUber` 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 `myUber` must be applied.

10 Questions

Got a question? Ask away by email:

Paolo Ballarini: paolo.ballarini@ecp.fr
Arnault Lapitre: arnault.lapitre@gmail.com

or post it on the Forum page on Claroline.