IS1220 - Final Project
`MyFoodora` - Food Delivery System

CentraleSupélec

Hand-out: March 13, 2017
Due Part 1: April 10, 2017
Due Part 2: April 23, 2017
Programming Language: Java

1 Overview

The goal of this project is to develop a software solution, called `MyFoodora`, whose functionality are similar to that of nowadays **Food Delivery Systems** such as, for example, *Foodora* (`www.foodora.fr`) or *Deliveroo* (`www.deliveroo.fr`). `MyFoodora` has to be conceived as a platform that give customers access to a set of restaurants from which they can order food according to different modalities (i.e. *à-la-carte* or by selection of a pre-compiled meals). Registered customers make their own selection and they are charged a total fee which is cashed by `MyFoodora` and that includes also the cost of delivery. The total fee is broken down into: 1) the price of the order (which is set by each restaurant) plus 2) a *service-fee* which is set by the `MyFoodora` manager. In order to guarantee an income `MyFoodora` also apply a *markup percentage* ("*percentage de marge*") to the price of an order. The *markup percentage* represents the percentage of money retained by `MyFoodora` from the price of an order placed to a given restaurant. The `MyFoodora` system must then payback both the restaurants as well as the courriers. Restaurants compile their own menus and may set up special offers. Once an order is placed (and payed for) the system is in charge for managing the delivery of the order, that is, the system will find an available deliverer amongst a fleet of available couriers. Customers may register to fidelity plans that allows them to obtain discounts and/or access to special offers menus.
The project consists of two parts:

Part 1: `MyFoodora` **core**: design and development of basic functionalities for the `MyFoodora` system (i.e. designing of the Java infrastructure for the `MyFoodora` system);

Part 2: `MyFoodora` **interface**: design and development of an interface which allows the actor(s) to interact with the `MyFoodora` system.

The requirements for both parts of the project are given in Section 2.

2 Project requirements

The following requirements describe the main functionalities that the `MyFoodora` system should guarantee. The designer has to find out a proper Java design so that all requirements are fulfilled by the resulting software implementation.

2.1 Part 1: `MyFoodora` core

Restaurants, menus, meals. Each restaurant offers a menu whose items are split in three categories: *starters*, *main-dishes*, *desserts*. Each menu's item has a price and is either of type *standard* or *vegetarian*. Furthermore some menu's items may also be classified as *gluten-free*. Client may order *à-la-carte* by choosing any combination of items from a restaurant's menu. Furthermore each restaurant offers a set of predefined meals, that can be selected by all clients. Meals may also distinguished in three kinds: *standard*, *vegetarian*, *gluten-free*, and accordingly they must consist of combination of items of the same category (e.g. a vegetarian meal must contain only vegetarian items). Meals are also distinguished into *half-meals*, and *full-meals*. An *half-meal* consists of two items: either a starter and a main-dish or a main-dish and a dessert. A *full-meal* consists of a starter a main-dish and a dessert. The default price of a meal is obtained by applying a 5% discount on the total price given by the sum of each meal's item price. At least one meal amongst those offered by a restaurant is offered as a *meal-of-the-week* special offer. The *meal-of-the-week* offer has a discounted price, which is given by application of a *discount-factor* which can be chosen by the restaurant and that, by default, is set to 10% of the total price of each item composing the meal.

Users: managers, restaurants, couriers, customers. The `MyFoodora` system should support the following kinds of user: *managers*, *restaurants*, *couriers* and *customers*, described as follows:

- **Managers**: `MyFoodora` managers have a name, surname, a unique ID and username, the latter used for logging in the system. They are in charge of overseeing the `MyFoodora` system, hence they can:
  - add/remove any kind of user (restaurant, customers and/or couriers) to/from the system.
  - activate/disactivate any kind of user (restaurant, customers and/or couriers) of the system

  Also managers may perform a number of operations related to assessing the business performances of the `MyFoodora` system including:

  - changing the *service-fee* percentage and/or the *markup percentage* ("*percentage de marge*") and/or the *delivery-cost*
  - computing the total income and/or profit over a time period
  - computing the average income per customer (i.e., the total income divided by the number of customers that placed at least one command) a time period
  - determining either the *service-fee* and/or *markup percentage* and/or the *delivery-cost* so to meet a *target-profit* (see **target profit policies** below)
  - determining the most/least selling restaurant
  - determining the most/least active courier of the fleet
  - setting the current *delivery-policy* used by `MyFoodora` to determine which courier is assigned to deliver an order placed by a customer

- **Restaurants**: have a name, a unique ID, an address (expressed as a two dimensional co-ordinate) and username, the latter used for logging in the system. They are in charge for:
  - editing the restaurant menu (adding/removing items)
  - creating/removing different meals (half or full meal, vegetarian, gluten-free and/or standard meals).
  - establishing the *generic discount factor* (default 5%) to apply when computing a meal price
  - establishing the *special discount factor* (default 10%) to apply to the meal-of-the-week special offer.
  - sorting of shipped orders with respect to different criteria (see below)

- **Customers**: have a name, a surname, a unique ID, an address (expressed as a two dimensional co-ordinate), an email address, a phone number and a username, the latter used for logging in the system. They can
  - place orders: this includes choosing a selection of items *à-la-carte* or one or more meals offered by a given restaurant, and paying the total price for the composed order.
  - register/unregister to/from a fidelity card plan
  - access the information related to their account: including history of orders, and points acquired with a fidelity program
  - give/remove consensus to be notified whenever a new special offer is set by any restaurant

- **Couriers**: have a name, surname, a unique ID, a position (expressed as a two dimensional co-ordinate), a phone number, a counter of delivered orders and a username, the latter used for logging in the system. They can:
  - register/unregister their account to the `MyFoodora` system.
  - set their state as *on-duty* or *off-duty*
  - change their position
  - accept/refuse to a delivery call (received by the `MyFoodora` system)

The `MyFoodora` system. This is the core part of the system. It stores the entire state of the system (i.e., registered managers, restaurants, customers and couriers), the history of completed orders, as well as the profit-related information, such as: the *service_fee*, the *markup_percentage* ("*percentage de marge*") and the *delivery_cost* (e.g. when a customer place an order the core system is in charge for allocating a courier for delivering the order, etc). Furthermore it provides basic services that user's specific operations rely upon. Specifically the `MyFoodora` system allows for:

- setting of the *service-fee*, the *markup percentage* ("*percentage de marge*") and the *delivery_cost* values
- allocating of a courier to an order placed by a customer (by application of the current **delivery policy**, see below details of supported policies)
- notifying users that gave consensus to receive special offers notifications, of a new special offer set by a restaurant
- computing the *total income* (i.e. the sum of all completed orders) as well as the *total profit* of the system, knowing that the the profit of a single order is given by:

  $$profit\_for\_one\_order = order\_price \cdot markup\_percentage + service\_fee - delivery\_cost$$

- chose the ***target profit policy*** (see below) used to optimise the profit-related-parameters (*service-fee*, *markup percentage*, and the *delivery_cost*)

Policies supported by `MyFoodora` system.

To help optimising the functioning of the system `MyFoodora` should support different kinds of policies. Specifically:

- **Target profit policies**. Allow managers for reasoning about different ways of meeting a given *target profit*. `MyFoodora` should support the following policies:

  - **targetProfit_DeliveryCost**: based on the last month total income (i.e. number of completed orders), and given a *service_fee* value, and a *markup_percentage* value compute the *delivery_cost* value to meet a given *target_profit*.
  - **targetProfit_ServiceFee**: based on the last month total income (i.e. number of completed orders), and given a *markup_percentage* value, and a *delivery_cost* value, compute the *service_fee* value to meet a given *target_profit*.
  - **targetProfit_Markup**: based on the last month total income (i.e. number of completed orders), and given a *service_fee* value and a *delivery_cost* value, compute the *markup_percentage* value to meet a given *target_profit*.

- **Delivery policies**. Allow for allocating a courier to an order to be delivered. `MyFoodora` should support the following policies:
  - **fastest delivery**: the courier which has the shortest distance to cover to retrieve the order from the chosen restaurant and delivering it to the customer is chosen
  - **fair-occupation delivery**: the courier with the least number of completed delivery is chosen

- **Shipped order sorting policies**. Allow restaurants and managers to sort the shipped orders according to different criteria. `MyFoodora` should support the following policies:
  - **most/least ordered half-meal**: display all half-meals sorted w.r.t the number of shipped half-meals
  - **most/least ordered item *à la carte***: display all menu items sorted w.r.t the number of time they been selected *à la carte*

Pricing and fidelity cards. The pricing policy depends on the kind of fidelity card that a user owns. The `MyFoodora` handles different types of cards, including the following three types: **basic fidelity card**, **point fidelity card** and **lottery fidelity card**. The final price is then calculated according to the following rules:

1. **basic fidelity card** is the card given by default, at registration, to any user. This card simply allows to access to special offers that are provided by the restaurant
2. **point fidelity card**. A client can select to have this fidelity card. Instead of having the special offer she will gain a point for each 10 euros spent in the restaurant. Once she will reach 100 points she will receive a 10% discount on the next order.
3. **lottery fidelity card**. A member that has this card will not access to any offer nor gain any points but will have a certain probability to gain her meal for free each day[1]

[1] In the initial implementation you can decide the lottery outcome at the startup of the system.

Remark (an OPEN-CLOSE solution). The main principle to follow while designing your software solution to meet the `MyFoodora` requirements given above is that of the OPEN-CLOSE principle. Therefore you should apply as much as possible the principle for flexible code design (i.e. design patterns).

A roadmap to design and implementation We briefly remind you the roadmap you should follow to develop your solution to the `MyFoodora` system.

1. carefully read and analyse the `MyFoodora` requirements (from the requirements start identifying potential application of design patterns: which pattern ? to solve which problem ?)
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

2.2 Use case scenario

The following use case scenario describe examples of how the `MyFoodora` should function.

Startup scenario

1. the system loads all registered users: at least 2 manager (the CEO and his deputy i.e. ajoint), 5 restaurants and 2 couriers, 7 customers, 4 full-meals per restaurant...
2. the system sends alerts to the customers that agreed to be notified of special offers

Register a user

1. a user start using the system because she wants to register
2. the user inserts his first-name, his last-name, his username, his address, his birth-date...

3. the user starts inserting a contact info with the type and the value (e.g. email, phone)
    - the user repeats step 3 since he ends to inserts his contact info
4. if the user is a customer she sets the agreement about the special offer contact (by default it is no)
5. the user is a customer selects the contact to be used to send the offers (by default it is the e-mail if exists)
6. if the user is a courier he sets his current duty status (default off-duty)
7. the user specify to save the account

Login user

1. a user wants to login
2. the user inserts username and password
3. the system handles the login and presents to the user the available operations according to his role

Ordering a meal

1. a client start using the system because she wants to order a meal
2. the client inserts his credentials (username and password)
3. the system recognizes the client and proposes the available restaurants
4. the client select a restaurant and compose an order either by selecting dishes *à la carte* or by selecting meals from the restaurant menu. For each item in the order the client specifies the quantity.
5. Once the order is completed the client selects the end
6. the system shows the summary of the ordered dishes and the total price of the order taking into account the pricing rules

Inserting a meal or dish in a restaurant menu

1. a restaurant person start using the system because she wants to insert a new meal
2. she inserts the restaurant credentials (username and password)

3. the system recognizes the restaurant and shows the list of dishes and meals in the menu
4. the restaurant selects the insert new meal (or dish) operations
5. the restaurant inserts the name of the new meal (or dish) to be added and specify whether it is an half-meal or a full-meal or a meal-of-the-week
6. in case of a dish the restaurant specify the unit price and the category its category (starter, main dish, dessert)
7. in case of meal
   - the restaurant inserts the dishes of the meal
   - the restaurant compute and save the price of the meal
8. the restaurant saves the new created meal (or dish) in the menu

Adding a *meal of the week* special offer

1. a restaurant staff starts using the system and inserts the restaurant credentials
2. the system shows all restaurant's available meals
3. the restaurant selects the meal to be set as *meal of the week*
4. the system automatically updates the price of selected *meal of the week*, by application of *special discount factor*
5. the system notifies the users (that agreed to be notified of special offers) about the new offer

Removing a *meal of the week* special offer

1. a restaurant staff starts using the system and inserts the restaurant credentials
2. the system shows all restaurant's available meals
3. the restaurant selects a meal in the *meal of the week* list and selects the remove from its special offer state.

2.3 Part 2: `MyFoodora` user interface

The part 2 of the project is about realizing a user interface for the `MyFoodora`-app. The user interface consists of two parts: the command-line user interface (CLUI), which is mandatory, and the the graphical user interface (GUI), which is optional (but will gain extra points, which can re-enforce points lost elsewhere).

2.3.1 `MyFoodora` 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 `MyFoodora` core. A command consists of the `command-name` followed by a blank-separated list of (string) arguments. For example:

```
registerRestaurant "TourDargent" "45.1,66.2" "12345678"
```

In particular command-line interpreter should feature the following list of commands, whose syntax is denoted as follows:

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

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

- `login <username> <password>` : to allow a user to perform the login (**remark**: a Myfoodora manager user with username: `ceo` and password:`123456789` is assumed to exist)
- `logout <>` : to allow the currently logged on user to log off
- `registerRestaurant <name> <address> <username> <password>` : for the currently logged on manager to add a restaurant of given name, address (i.e. address should be a bi-dimensional co-ordinate), username and password to the system.
- `registerCustomer <firstName> <lastName> <username> <address> <password>` : for the currently logged on manager to add a client to the system
- `registerCourier <firstName> <lastName> <username> <position> <password>` : for the currently logged on manager to add a courier to the system (by default each newly registered courier is `on-duty`).
- `addDishRestauarantMenu <dishName> <dishCategory> <foodCategory> <unitPrice>` : for the currently logged on restaurant to add a dish with given name, given category (starter,main,dessert), food type (standard,vegetarian, gluten-free) and price to the menu of a restaurant with given name (this command can be executed by a restaurant-user only)

- `createMeal <mealName>` : for the currently logged on restaurant to create a meal with a given name
- `addDish2Meal <dishName> <mealName>` : for the currently logged on restaurant to add a dish to a meal
- `showMeal <mealName>` : for the currently logged on restaurant to show the dishes in a meal with given name
- `saveMeal <mealName>` : for the currently logged on restaurant to save a meal with given name
- `setSpecialOffer <mealName>` : for the currently logged on restaurant to add a meal in meal-of-the-week special offer
- `removeFromSpecialOffer <mealName>` : for the currently logged on restaurant to reset a special offer
- `createOrder <restaurantName> <orderName>` : for the currently logged on customer to create an order from a given restaurant
- `addItem2Order <orderName> <itemName>` : for the currently logged on customer to add an item (either a menu item or a meal-deal) to an existing order
- `endOrder <orderName> < date>` : for the currently logged on customer to finalise an order at a given date and pay it
- `onDuty <username>` : for the currently logged on courier to set his state as on-duty
- `offDuty <username>` : for the currently logged on courier to set his state as off-duty
- `findDeliverer <orderName>` : for the currently logged on restaurant to allocate an order to a deliverer by application of the current delivery policy (remark: this is just an extra facility of the CLUI, that will allow us to test whether deliverer allocation works properly. The actual allocation of a deliverer should be automatically triggered by the system on completion of an order by a customer).
- `setDeliveryPolicy <delPolicyName>` : for the currently logged on myFoodora manager to set the delivery policy of the system to that passed as argument
- `setProfitPolicy <ProfitPolicyName>` : for the currently logged on myFoodora manager set the profit policy of the system to that passed as argument
- `associateCard <userName> <cardType>` for the currently logged on myFoodora manager to associate a fidelity card to a user with given name

- `showCourierDeliveries <>` for the currently logged on myFoodora manager to display the list of couriers sorted in decreasing order w.r.t. the number of completed deliveries
- `showRestaurantTop <>` for the currently logged on myFoodora manager to display the list of restaurant sorted in decreasing order w.r.t. the number of delivered orders
- `showCustomers <>` for the currently logged on myFoodora manager to display the list of customers
- `showMenuItem <restaurant-name>` for the currently logged on myFoodora manager to display the menu of a given restaurant
- `showTotalProfit<>` for the currently logged on myFoodora manager to show the total profit of the system since creation
- `showTotalProfit <startDate> <endDate>` for the currently logged on myFoodora manager to show the total profit of the system within a time interval
- `runTest <testScenario-file>` for a generic user of the CLUI (no need to login) to execute the list of CLUI commands contained in the `testScenario` file passed as argument
- `help <>` for a generic user of the CLUI (no need to login) to display the list of available CLUI commands (a command per line) with an indication of their syntax

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 3.2), 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 order a meal which is not contained in the menu, or modifying a meal that is not in the menu, etc.

2.3.2 `MyFoodora` graphical user interface

The goal of this part is to realize an interactive Graphical User Interface (GUI) for the `MyFoodora` core. The GUI must support all operations implemented in Part 1. Navigation should be possible using both mouse and keyboard. For this part of the project you only have to support a single user; multi-user support is not required. You are encouraged to design a nice and user-friendly interface. An interactive GUI must provide for example the possibility to select a menu, to visualize this menu (using a click on the name of the menu,

or on a button, ...), etc. Bonus points are given for a responsive GUI that does not stall during long-running operations, like search of meals in a menu.

`MyFoodora` GUI requirements

1. The browser should support all operations from Part 1 (`MyFoodora` core). For example, users should be able to create a new restaurant, to add a menu to a restaurant, to define a meal deal for a restaurant, to add users, to add courriers, etc.) all of which without using console commands.
2. The browser should support the login of the user showing the available operations only to a certain category of users (a client cannot insert a new meal).
3. The browser should support mouse navigation. The required operations are the same as in requirement.

2.4 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`

3 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.

3.1 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.

3.2 Test scenario

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

- one initial configuration file (called `my_foodora.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 least: 2 managers (the CEO, and his deputy), 5 restaurants and, per restaurant, 4 full-meals, 2 couriers, 7 customers... This `my_foodora.ini`'s file can be seen as a scenario file containing only user registration, meal creation and dishes adding, ...'s' commands.

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 `myFoodora` system (i.e. creation of some resraurants, each with its one menu and meals offers, creation of some customers, as well as some courriers, simulation of some usage of the system with creation of orders by customers and delivery of orders, calculation of profits for myFoodora, 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
```

The mandatory `testScenario1.txt` must test each and every command of the CLUI (except obviously `runtest`). Furthermore it should develop a meaningful scenario like that depicted in the following example:

- logon as manager
- **adding commands to show/print all items (user, meals, ...) in the initial configuration**
- register a few restaurants to the system
- register a few customers
- register a few couriers
- display the list of restaurant
- display the list of customers
- logout manager

- logon as restaurant (for each new restaurant)
- add (and populate) a menu to each restaurant (this requires login as restaurant manager)
- add (half and/or full) meal to each restaurant's menu, setting some special-offer and meal-of-the-week meal deal
- display the menu of each restaurant
- logout restaurant
- logon as customer (for few customers)
- simulate placing of a few orders by few customers (concerning a few restaurants)
- logout
- logon as manager
- compute/display the total profit of the system (since creation and over a given time-interval)
- change the profit policy of the system and re-compute the total profit of the system (since creation and over the same time-interval as before)
- display the list of courriers sorted according to their operativity (num. of completed delivery)
- **adding commands to completely simulate special offer and its notification for:**
  - **meal of the day/week**
  - **birthday celebration**

  This scenario requires you to manage explicitly time advancement (day change with a dedicated Thread ?)
- logout
- ...

4 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 `MyFoodora` core**: due **April 10, 2017** (i.e. $\sim$ 4 weeks work)
- **Part 2, the `MyFoodora` user-interface**: due **April 23, 2017** (i.e., $\sim$ 2 weeks work) which can consist in either:
  - a command-line shell: allowing the user to interact with the `MyFoodora` core through a number of pre-defined commands
  - a graphical user interface: allowing the user to interact with the `MyFoodora` 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).

4.1 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`.

4.2 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_foodora.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 `MyFoodora`-app following the test cases described in Section 3

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).

5 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 `MyFoodora`-core → responsible: Arnault, task2 of `MyFoodora`-core →responsible: Vladimir, 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/current_students/msc_cgvi/projects/project_report_structure/`.

6 Project grading

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

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

- `MyFoodora` GUI (max 20 points, bonus)

Each part of the solution (i.e. `MyFoodora` CORE, `MyFoodora` CLUI, and also `MyFoodora` 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 April 23, 2017. However, each team has to provide an implementation and a report summarizing the current design and project status at each intermediate deadline (April 10, 2017). 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.

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

8 Questions

Got a question? Ask away by email:

Paolo Ballarini: paolo.ballarini@ecp.fr
Arnault Lapitre: arnault.lapitre@gmail.com
Celine Hudelot: celine.hudelot@ecp.fr

or post it on the Forum page on Claroline.