Faire un fichier test scénario pour ne pas rentrer toutes les commandes à la main (3/4 fichiers tests scénarios)
Implémenter les différentes méthodes (par exemple si addMeal n'est pas dans un fichier scénario, Lapitre condisère que cette méthode n'existe pas

Par exemple 100 tests environ

design pattern dans chapitre 3 sur l'implémentation
Pour que tous les clients soient au courant qu'il y a un nouveau menu

Noms auteurs dans la java doc

Commentaires pour la java doc

Project work
`myFoodora` - Food Delivery System

CentraleSupélec

Rapport : introduction (ce qu'on va faire)
conclusion (ce qui a été fait, avec +/- de difficultés, si on est allé plus loin --> perspectives futures (application android, iphone ...)
rapport au plus 20 pages au moins 10

Hand-out: May 9, 2025
Due : **June 4, 2025**
Programming Language: Java

src.gen : uniquement cela qui est regénéré
src : pas regénéré

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 the core Java infrastructure for the `myFoodora` system (based on requirements given in Section 2).

Part 2: **`myFoodora` user-interface**: design and development of a user-interface for the `myFoodora` system (see Section 4).

2 System description and requirements

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

2.1 Menus and meals

Restaurants, which are amongst the basic components of the `myFoodora` systems (see description below), sell dishes to customers. Dishes offered by a restaurant are collected in a menu and also may be arranged in specific combinations to form meals which are offered at special fares, as an alternative to ordering dishes *à la carte*. Menus and meals are characterised as follows:

- **Menus**. 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.

- **Meals**. Furthermore each restaurant offers a set of predefined meals, that can be selected by the 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.

2.2 Users of the `myFoodora` system

The `myFoodora` system shall support different types of users, such as:

- **Restaurants**: a restaurant is characterised by a name, a unique ID, a location (expressed as a two dimensional co-ordinate) and a username and password, the latter used for logging in the system. Furthermore a restaurant has a menu and some meals (see Menu and Meal description). A restaurant has the responsibility to:
  - edit the restaurant menu (adding/removing dishes)

  - create/remove 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.

- **Customers**: a customer is characterised by 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. A customer has the responsibility to:
  - 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**: a courier is characterised by a name, surname, a unique ID, a position (expressed as a two dimensional co-ordinate) representing the physical position where the courier is currently located, a phone number, a counter of delivered orders and a username, the latter used for logging in the system. A courier has the responsibility to:
  - set their state as *on-duty* or *off-duty*
  - change their position
  - accept/refuse to a delivery call (received by the `myFoodora` system)

- **Managers**. A `myFoodora` manager has the responsibility of performing "management" related operations on the system, for example, adding other users to the system, or computing relevant statistics about the `myFoodora` usage, and so on (see details below). A manager of `myFoodora` is characterised by a name, surname, a unique ID and username, the latter used for logging in the system. A manager is in charge of overseeing the `myFoodora` system, hence it 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. Furthermore a manager 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

2.3 The actual `myFoodora` system

attributs en italique

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 *serviceFee*, the *markupPercentage* ("*percentage de marge*") and the *deliveryCostt* (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 *deliveryCost* 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:

$$profitForOneOrder = orderPrice \cdot markupPercentage + serviceFee - deliveryCost$$

- chose the ***target profit policy*** (see below) used to optimise the profit-related-parameters (*serviceFee*, *markupPercentage*, and the *deliveryCost*)

3 Adding sophisticated functionalities

The `myFoodora` systems should be equipped with necessary means to support the following functionalities

3.1 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 *serviceFee* value, and a *markupPercentage* value compute the *deliveryCost* 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 *markupPercentage* value, and a *deliveryCost* value, compute the *serviceFee* 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 *serviceFee* value and a *deliveryCost* value, compute the *markupPercentage* value to meet a given *target_profit*.
- **Delivery policies**. Delivery policies are responsible for allocating once amongst the *on-duty* courier to an order placed by a Customer and need to be delivered. `myFoodora` should support the following policies:
  - **fastest delivery**: selects 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**: selects the courier with the least number of completed delivery is chosen
- **Shipped order sorting policies**. These policies should be available to both restaurants and managers of the `myFoodora`. They allow one to sort the shipped orders according to different criteria. The client requires that `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*

3.2 Pricing and fidelity cards

The pricing policy are used by the `myFoodora` system to determine the price of an order placed by a Customer. The price depends on the kind of fidelity card that Cusomer owns. The `myFoodora` shall handle 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]

Bonus points (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 `myFoodora` system a design pattern has been applied. You will gain bonus points depending on how much your solution complies with *open-close* principle.

3.3 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 birthdate...
3. the user starts inserting a contact info with the type and the value (e.g. email, phone)

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

- 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 customer start using the system because she wants to order a meal
2. the customer inserts his credentials (username and password)
3. the system recognizes the customer and proposes the available restaurants
4. the customer 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 customer specifies the quantity.
5. Once the order is completed the customer 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.

4 Part 2: `myFoodora` user interface

The part 2 of the project is about realising a user interface for the `myFoodora`-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`

You can start off from a **toy example of CLUI written in Java and whose code is available on Edunao**. `https://centralesupelec.edunao.com/pluginfile.php/526474/mod_resource/content/1/CLI_toy.java`

4.1 `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:

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

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

For example the `myFoodora` CLUI command for setting up a `myFoodora` system consisting of 4 restaurants (each of which has a menu and some meals) a manager, 3 customers, 5 couriers should be:

```
setup 4 3 5
```

Notice that the address (i.e. location) of restaurants and customers, as well as the position of each courier should be set randomly within an hypothetical square surface of given side.

4.1.1 `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 5.2), 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.

5 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.

5.1 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.

5.2 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`).

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 `myfoodora` system (i.e. creation of some foodora 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
```

6 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 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

7 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:

- **JAVA code**: in form of 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)
- **UML class diagrams**: one ore more UML class diagrams that illustrate the design of your code solution (UML class diagrams should be included, into a dedicated folder, within the Eclipse project)
- **Report**: a written report on the design and implementation of the corresponding part

7.1 Project naming

The Eclipse projects you submit must be named as follows:

- `fr.cs.groupNN.myFoodora` exported into file `GroupNN_MyFoodora_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 `Group01_MyFoodora_Turing_VonNeumann`, which they will export into a file named, `Group01_MyFoodora_Turing_VonNeumann.zip`.

7.2 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 5

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).

8 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: 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).

9 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:

- `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)

Each part of the solution (i.e. `myFoodora` CORE, `myFoodora` 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.

10 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.

11 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.