6 User interface

This section describes the second part of the project. As first the interpretation and execution of the commands of the *command line interface* will be clarified. Then the implemenation of the *graphical* user inteface will be described. Finally, explanations will be given about what was added or changed to the project requirements.

6.1 Command line interface

Firstly, a note has to be made on the general choice of implementation of the command line interface. Following UNIX command line style, one should be able to run a command with multiple arguments in the following style `command -arg1_type arg1 -arg2_type arg2`. In the `MyFoodora` case, it seems more adequate to have one syntax for each command, being `command <arg1> <arg2>`, which is proposed in the project requirements. The program will thus be able to give information on the commands when needed and we expect the user to quickly remember the commands, as there are only few of them for each user type.

The choice of implementation described in the first paragraph leads us to the use of an `HashMap<String, Integer>` to store the list of commands and their associated number of arguments, as we only have *one* number of arguments for one command, except for the `showTotalProfit` method which is handled simply by using `null` when no dates are given.

In terms of effeciency, using a parallel `HashMap` to store the commands names and their associated number of needed arguments allows to check, if the given command name exists in $\mathcal{O}(1)$ time on average and in $\mathcal{O}(\log n)$ on worst case [3].

6.1.1 Separing interpretor from processor

Thinking about the design, we quickly realised the need to separate the *request* (ie. the given command) from its actual *execution*. This idea follows the open/close principle and has similar advantages as design patterns (note that we discovered the *command design pattern* [8] which is somewhat similar). It indeed allows to easily change the behaviour of the program if one wants to change the syntax of the commands (*request* class) or what a particular command does (*execution* class). This leads to the implementation of the `CommandLine` and the `CommandProcessor` classes.

In order to make use of commands in an efficient way, a `Command` class is designed. It consists of a name and an array of strings containing its arguments. This allows the interpretor and processor to easily communicate between themselves and with the real user. The reader is advised to have a look at figure 3 containing the UML diagram of the relationship between the three listed above classes. One will notice the use of the Singleton pattern for the `CommandProcessor` and `CommandLine`. This is justified by the fact that only one of each will be required at the same time and it does not created problems with JUNIT tests as those are made using concrete scenarios generated by `.txt` files of commands.

6.1.2 The `CommandLine` class

The `CommandLine` can be seen as the *interpreter*. It will be the link class between the real user and the system. It can either interpret commands given directly as input with the `launchFromInput()` method, or listed in a `.txt` file with the `launchFromFile` method. Once a String, that should represent a command, is given to those methods, they will call the `getInputInfoAndProcessCmd` method whose role is to check if the input

1. corresponds to an existing command,
2. has the "<>" argument declaration,
3. has the right number of arguments associated with this command.

If one of those conditions is not satisfied, a message containing the error description will be send back to the `launch` method which will print it out. On the other hand, if the conditions are satisfied, the method creates a new `Command` object and passes it to the `processCmd` method of its `CommandProcessor` attribute.

6.1.3 The `CommandProcessor` class

The `CommandProcessor` can be seen as the *executor*. When its `processCmd` receives a `Command` (which is valid thanks to the `CommandLine`), it executes the behavior associated with the given command. Its attributes mainly consist of the `Core` system, as most of the method will need it and of a `DishFactory` and a `MealFactory` that will be used to produce meals and dishes efficiently.

The methods of this class will mostly consists of translation of user commands into `Core` commands. It can easily be noticed that not all user commands are available in the core but are directly applicable to the `current_user` of the core. For example when a courier wants to set his status to *avaible*, one simply executes `core.getCurrent_courier().setAvailable(true)`.

The following is about how the *creation of meals and orders* is handled. As we are required to create meals and orders using multiple commands in the following pattern

1. create a new `Meal` or `Order` giving basic information
2. add dishes (resp. items) to `Meal` (resp. `Order`)
3. validate steps 2 and 3 by using a `save` like function

we used *global* `private` variables to handle this and therefore have an `ArrayList <Meal>` to store the potentials meals and an `Order` object. Note that there is no need to store multiple orders, nor is there a need to give an order name since a `Customer` will only place one order at a time (see paragraph **??** for a more detailed explanation of order handling).

On the next page the reader will find the UML diagram of the general structure of the CLUI and the listing 3 containing the application of the Singleton pattern of the `CommandLine` class.

Listing 3: Singleton pattern with CommandLine class.

```
private CommandLine() {
  cmd_processor = CommandProcessor.getInstance();
  command_list = ParseCommands.parseCommands("src/txtFILES/
     mf_commands.txt");
  for(Command cmd : command_list) {
    command_hm.put(cmd.getName(), cmd.getNb_args());
  }
}
private static class CommandLineHolder {
  private static final CommandLine INSTANCE = new CommandLine
     ();
}
public static CommandLine getInstance() {
  return CommandLineHolder.INSTANCE;
}
```

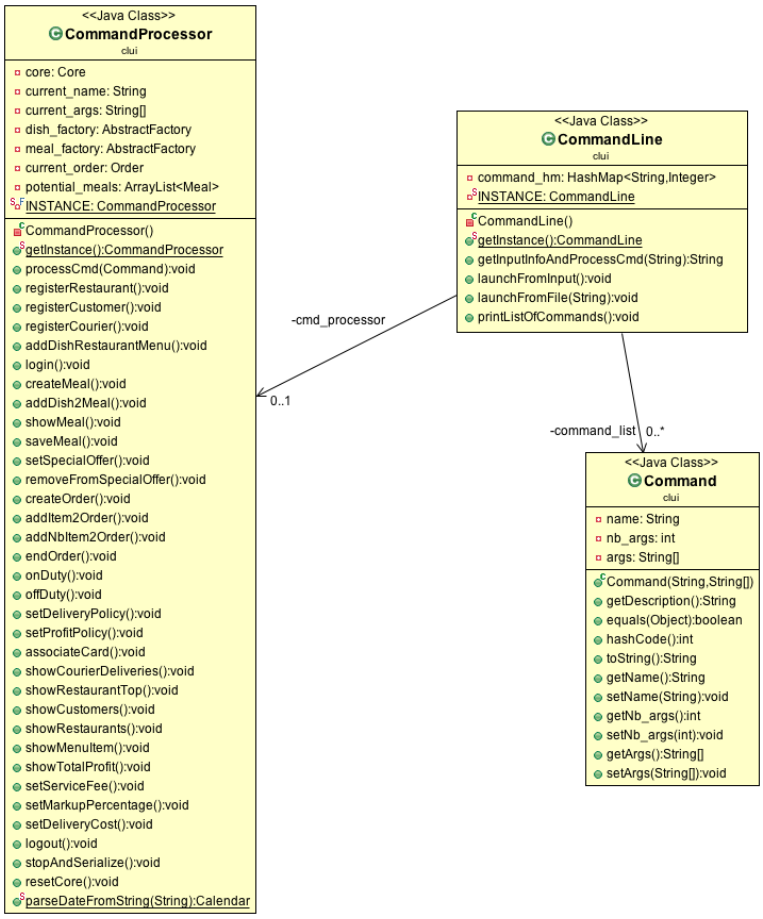

Figure 3: UML diagram : of the command line user interface.