CentraleSupélec

université PARIS-SACLAY

CentraleSupélec

Rapport projet MyFoodora

Object Oriented Software Programming

Groupe 15

Réalisé par :
Mathias Thirion
Maxime Leboeuf

9 juin 2024

Table des matières

Chapitre 1

Introduction

Notre rapport étant écrit en français, afin d'éviter les amalgames avec les traductions (en français, meal : menu et menu : menu), nous garderons les appellations en anglais, qui se trouvent être de toute façon le nom de nos classes. Ainsi, dans ce rapport **Menu** : menu global du restaurant, et **Meal** : menu précis (menu du jour, menu enfant etc...)

L'objectif du projet myFoodora était de réaliser une application de restauration à distance (similaire à Uber eats par exemple). Les fonctionnalités à inclure dans cette application variaient et devaient inclure différents utilisateurs. La gestion côté restaurant comportait la création de **Menu**, et de leur contenu (**Dish** et **Meal**). Les clients devaient pouvoir commander à partir des **Menu**. Les commandes étaient ensuite attribuées à des livreurs. Enfin, des administrateurs peuvent gérer la partie administrative, politique et statistique de l'application.

Nous avons structuré myFoodora en plusieurs parties, une pour chaque fonctionnalité. Nous détaillerons chacune de ces fonctionnalités dans la première partie du projet, puis nous détaillerons leur implémentation. Nous essaierons d'exercer un regard critique sur notre réalisation, sur ce que nous aurions pu ajouter, et sur les choix que nous avons fait par rapport au sujet.

Pour démarrer l'interface de l'application, allez dans le dossier clui, puis dans mainCLUI. Vous pouvez lancer l'application ! Ensuite, si vous le souhaitez, et comme le message de bienvenue vous l'indique, vous pouvez taper `startup` pour initialiser le système avec des restaurants, users,.... Vous pouvez également taper `help` pour avoir la liste des commmandes.

Chapitre 2

Caractéristiques du projet

2.1 Users

Le système MyFoodora doit être utilisé par différents types d'utilisateurs, dits **User**. Ces utilisateurs sont : les **Manager**, les **Customer** (clients), les **Restaurant** et les **Courier** (livreurs). Tous ces utilisateurs ont des droits différents dans le système permettant de réaliser diverses opérations qui seront détaillées par la suite, et dont les Manager sont les administrateurs.

Dans cet univers, les **Restaurant** offrent aux **Customer** un **Menu**, composé lui même de **Dishes** et de **Meals**. Un **Meal** est soit un **FullMeal**, soit un **HalfMeal**, composé pour un **FullMeal** d'un **Starter**, un **MainDish** et un **Dessert**. Un **HalfMeal** est lui, comme son nom l'indique, composé d'un **Starter** et d'un **MainDish** ou d'un **MainDish** et d'un **Dessert**. Ces Meal sont proposés à la commande par les **Customer**, et ils peuvent également commander "à la carte" en choisissant individuellement des Dish.

Ainsi, le principe du système **MyFoodora** doit être que les **Customer** puissent commander à manger dans un Restaurant, en choisissant ses plats et ses menus, et qu'un livreur soit assigné à cette commande. Additionnellement, le système doit pouvoir supporter des fonctionnalités spéciales, comme le choix d'une politique de livraison différente selon les envie du Manager, ainsi que des politiques de fidélité différentes selon les clients.

Voici le détail des fonctionnalités de chaque utilisateur :

Manager

- ajouter ou supprimer des **User**
- gérer les politiques
- gérer les frais du système (livraison, ...)
- afficher des statistiques (**Restaurant** le plus demandé, **Courier** le plus actif, ...)
- accéder aux revenus du système

Customer

- commander à un restaurant (**Order**)
- changer de carte de fidélité
- pour la fidélité basique : accepter ou non les notifications d'offres spéciales

Restaurant

- ajouter ou supprimer des **Dish** et des **Meal**
- ajouter ou retirer des **Meal of the Week**
- changer les politiques de réduction pour la fidélité basique

Courier

- changer d'état (**OnDuty**/**OffDuty**)
- choisir d'accepter ou refuser une demande de livraison envoyée par le **Core**
- mettre à jour sa position après livraison

2.2 Core

Le **Core** est la partie principale de l'application. Il regroupe les différentes composantes, et permet l'interaction entre les données de l'application et l'interface utilisateur.

En effet, le **Core** stocke les données des différents utilisateurs enregistrés, et leur permet d'interagir. En particulier, il permet aux restaurants de modifier leur **Menu** (en utilisant les fonctionnalités du **Restaurant**) et aux clients de placer des **Order**. L'une des principales problématique du **Core** était de pouvoir caractériser ces **Order**. Nous avons choisi de diviser le système central de l'application, en créant une classe **Order** pour stocker les données de chaque commande, permettant un accès et une gestion des commandes grandement facilités.

Celle-ci offre donc la possibilité d'accéder :

- au **Customer** ayant passé la commande
- au **Restaurant** chez qui la commande est faite
- au **Courier** à qui la commande a été attribué et qui a accepté de la livrer
- le contenu de la commande ainsi que son prix
- ce que la commande rapporte à **myFoodora**

Le **Core** permet ensuite aux **Customer** de placer ces **Order** et de les manipuler, et les attribue à un **Courier** lorsque les clients ont terminé. Il suit pour cela la **DeliveryPolicy** choisie par les **Manager** de l'application (celles-ci seront détaillées dans la partie suivante).

Enfin, le **Core** garde en mémoire :

- les **User** enregistrés
- les **Order** reçus n'ayant pas été attribués, ainsi que l'historique des **Order** livrés
- le **User** connecté à l'application

Chapitre 3

Design

3.1 Factory pattern

En lisant le sujet et en posant le problème sur papier, il nous est apparu évident qu'un *Factory pattern* allait être nécessaire pour permettre de créer des objets **Meal** et **Dish**. Ces deux objets étant essentiellement différents, il s'est avéré que le pattern le plus utile dans ce cas était le **Abstract factory pattern**. Ainsi, le système est indépendant de la manière avec laquelle les objets sont créés, et peut simplement créer des **Meal** et des **Dish** avec une seule **Factory** qui crée elle même des "sous-**factories**".

Cependant, l'utilisation de ce pattern ne vient pas uniquement avec des avantages. En effet, l'utilisation de l'*abstract factory pattern* rend l'ajout de nouvelles familles d'objet compliqué, car il faut *Override* toutes les méthodes de l'interface **AbtractFactory** dans toutes les **Factories**. C'est, en un sens, un problème que nous avons rencontré lorsque nous avons décidé de rajouter un constructeur dans l'une des classes et que nous avons dû tout modifier en conséquence.

Ce pattern reste, cela dit, puissant, et son diagramme UML peut être aperçu ci-dessous :

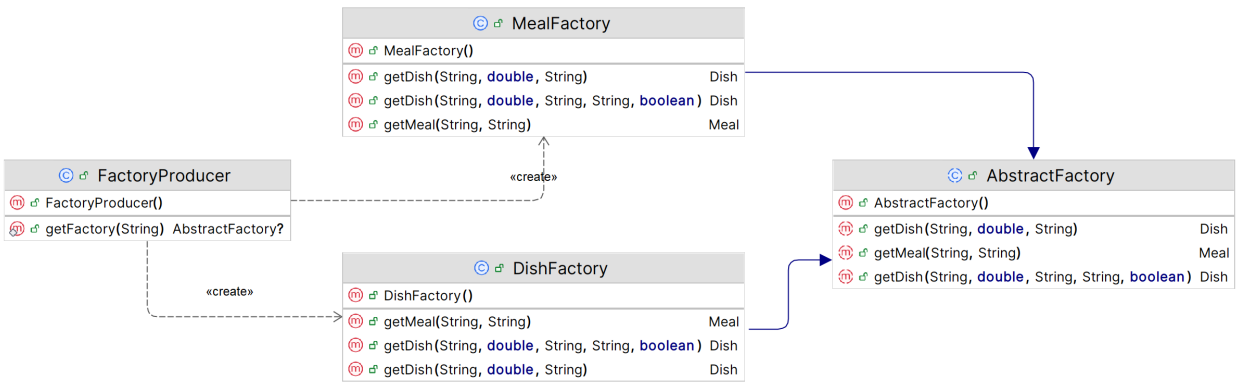

FIGURE 3.1 – Diagramme UML de l'Abtract Factory Pattern dans notre projet

3.2 Strategy patterns

Un autre pattern utilisé dans notre projet est le **Strategy pattern**. Ce pattern est utilisé à deux reprises dans notre projet, pour gérer les différentes politiques implémentées. En effet, et comme évoqué plus tôt dans ce rapport, le système doit être régi par plusieurs politiques, dont une politique de livraison et une politique de fidélité. Le strategy pattern est donc très utile dans ce cas, car il permet d'appliquer à un objet (par exemple le prix dans le cas de la fidélité) un calcul particulier selon les attributs donnés, tout en utilisant une seule méthode pour le réaliser, peu importe la politique choisie.

Ainsi, dans le cadre des politiques de livraison, deux sont implémentées : "**Fastest Delivery**" et "**Fair-Occupation Delivery**". La première permet de trouver le livreur le plus proche du restaurant pour lui communiquer la commande, tandis que la deuxième permet de trouver le livreur ayant réalisé le moins de commande pour la lui envoyer, pour favoriser "l'équité" parmi les livreurs.

Enfin, l'autre politique est la politique de fidélité. Chaque utilisateur peut posséder une carte de fidélité parmi trois proposées : la **Basic** qui permet simplement d'être notifié de quel **Meal** est le **MealOfTheWeek** et de profiter d'une petite réduction. La **Points** qui permet de récolter un point tous les 10€ d'achat, avec une réduction de 10% tous les 100 points, et la **Lottery**, qui permet d'obtenir sa commande gratuitement avec une certaine probabilité.

Les diagrammes UML de ces design patterns sont visibles ci-dessous :

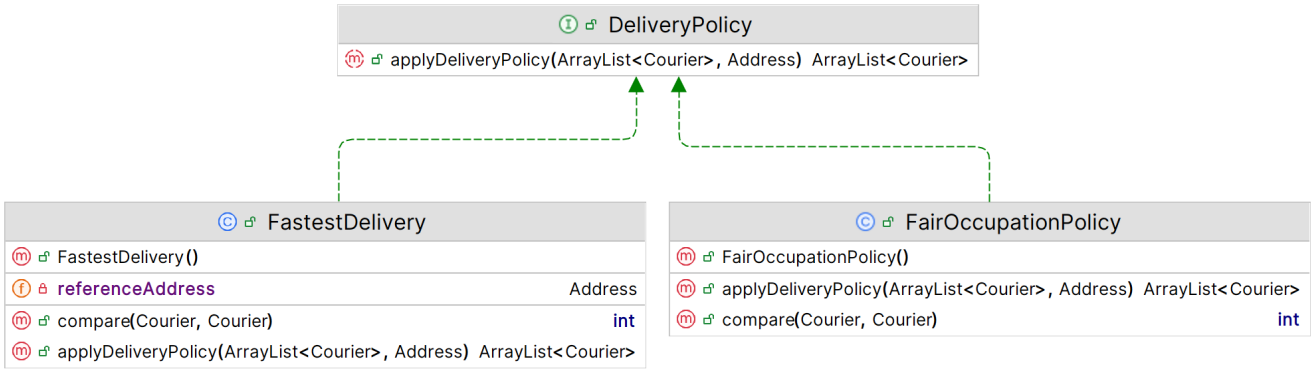

FIGURE 3.2 – Diagramme de la **Delivery Policy**

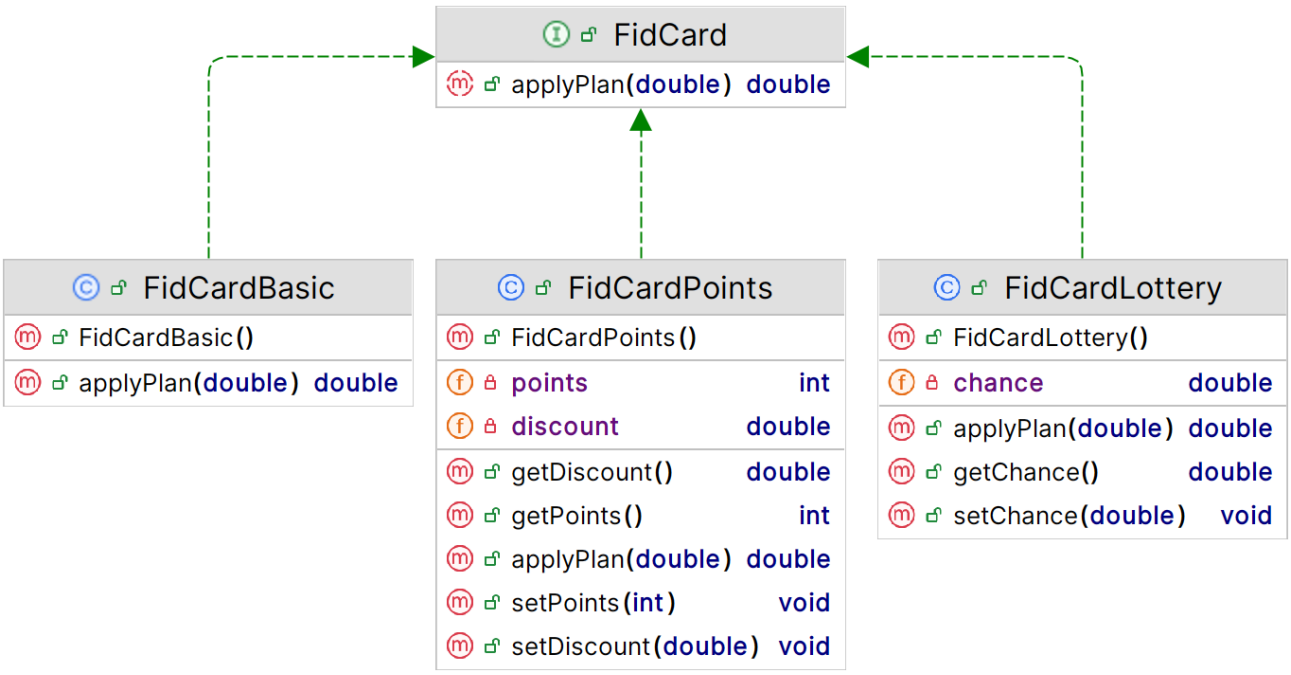

FIGURE 3.3 – Diagramme de la **Fidelity Card Policy**

3.3 Observer pattern

Nous avons aussi observé que l'utilisation de l'**Observer pattern** pouvait convenir à une partie de l'application. En effet, les **Customer** devaient être notifiés en cas de publication de **Meal of the week** par un **Restaurant**, et pouvoir accepter ou non de recevoir ces notifications. L'**Observer pattern** permettant justement de notifier une classe en fonction des variations de l'état d'une autre classe, nous avons décidé de l'utiliser.

L'**Observer** était de manière évidente le **Customer**, puisque c'est lui qui était supposé être notifié d'une nouvelle offre spéciale. Cependant, l'**Observable** nous a posé plus de problème.

La solution qui nous est d'abord apparue a été de catégoriser comme **Observable** la classe **Restaurant**, puisque ce sont ces derniers qui publient les offres spéciales. Cependant, cela a rapidement posé plusieurs problèmes :

- les **Restaurant** n'avaient pas accès aux **Customer**
- la communication entre les deux catégories d'utilisateurs était peu adaptée dans le modèle suivi
- les **Customer** devaient être notifiés d'offres venant de tous les **Restaurant**, pas seulement de **Restaurant** en particulier

Aussi, nous avons décidé que le moyen le plus adapté serait de mettre le **Core** en tant qu'**Observable**, car cela palliait aux problèmes évoqués, en particulier au niveau de l'accès aux données. L'inconvénient de ce choix était que l'implémentation était plus tardive, et permettait moins de visibilité sur l'utilisation du pattern.

Ainsi, les **Customer** peuvent s'enregistrer en précisant qu'ils souhaitent recevoir des notifications et changer d'avis. Les **Restaurant** peuvent ajouter des **Meal** à la liste des **Meal of the week**, ce qui notifie automatiquement les **Customer** enregistrés.

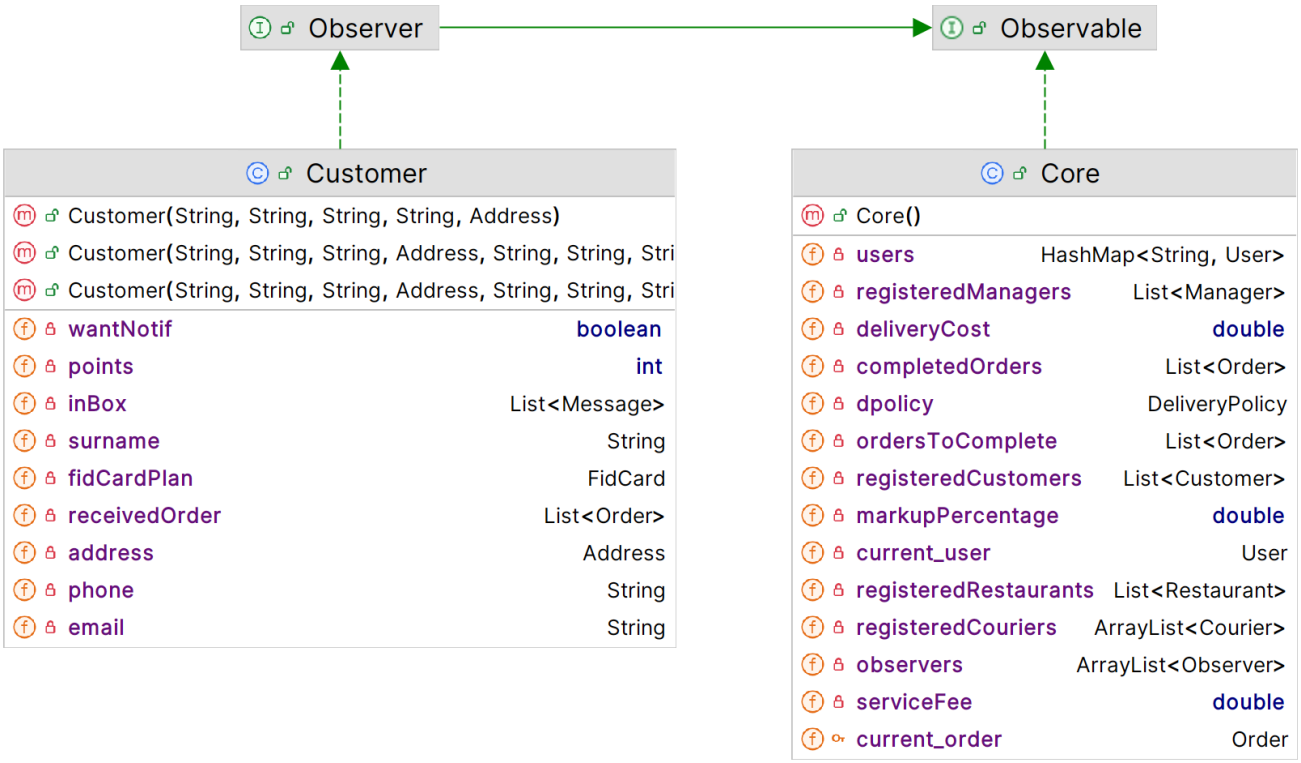

FIGURE 3.4 – Diagramme de l'**Observer**

Chapitre 4

Code

4.1 Restaurants et RestaurantComponents

Les restaurants avaient la possibilité de mettre à jour leur **Menu** en créant différents **Dish** et **Meal**. Aussi, et comme détaillé dans la partie sur le **Factory Pattern**, nous avons isolé ces **RestaurantComponents**. Ce package contient les classes représentant **Dish** et ses sous-classes, **Meal** et ses sous-classes, ainsi que les différentes **factories**.

Une classe **Menu** regroupe toutes ces composantes et les stockent dans des **ArrayList** pour représenter le menu d'un restaurant. Il inclue aussi le **Meal of the week**. Le sujet étant parfois ambigu sur la forme que devait prendre celui-ci, nous avons pensé que ce qui répondait le mieux au consigne était une liste de **Meal of the week**, dans laquelle on pouvait ajouter ou retirer des **Meal** déjà présent dans le **Menu**.

Les **factories** ont été utilisées dans la classe **Restaurant**. Pour ajouter des éléments du menu d'un restaurant, on appelle ces **factories** pour les créer. La première réflexion lors de l'implémentation ayant été que l'utilisateur des classes représentant ces items étant le **Restaurant**, il était logique d'utiliser les **factories** à cet endroit. Cependant, avec un peu de recul, le **Menu** sert d'interface entre les items et le **Restaurant**, et je pense qu'il aurait été plus adapté d'écrire les méthodes de modification du menu directement dedans en utilisant les **factories**.

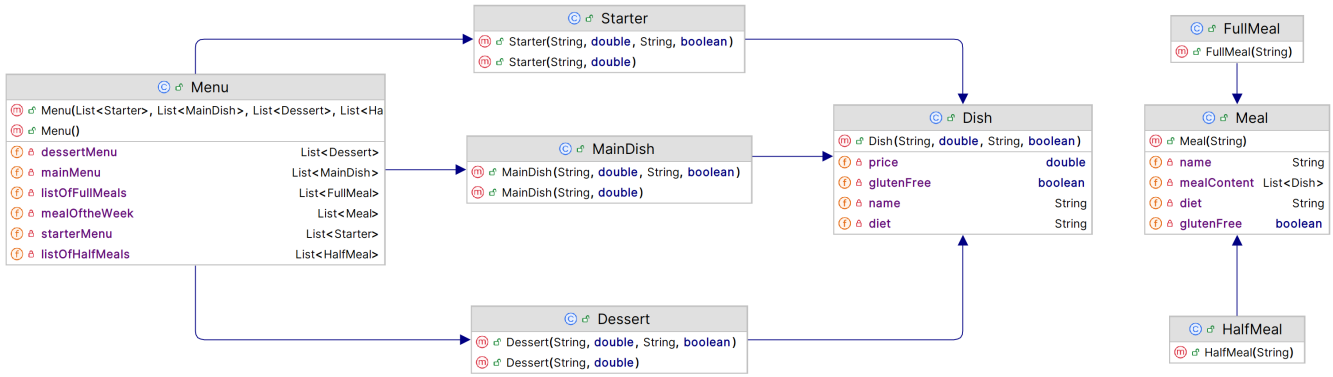

FIGURE 4.1 – Diagramme partiel du package **restaurantComponents**

4.2 Users

Nous avons créé une superclasse **User** héritée par chaque type d'utilisateur, regroupant les informations basiques et communes à chaque sous-classe. Chaque sous-classe d'utilisateur implémente les fonctions utiles au type d'utilisateur qu'il représente (modifier les menus pour **Restaurant**, accepter ou refuser des **Order** pour les **Courier**...). La javadoc indique les fonctionnalités de chaque utilisateur.

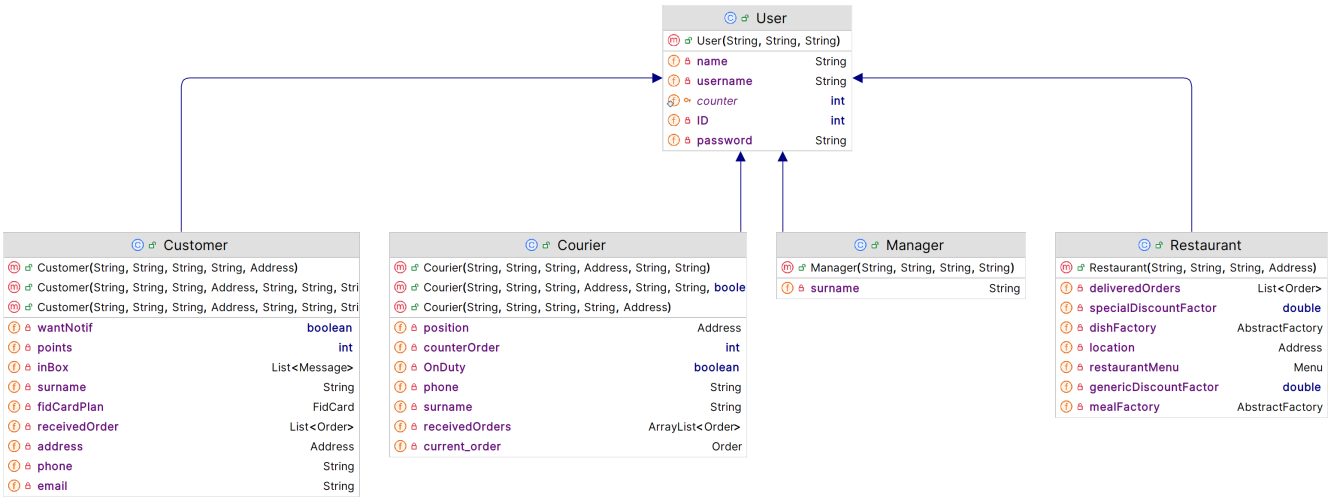

FIGURE 4.2 – Diagramme partiel du package **User**

4.3 System

Comme précisé dans la partie **Caractéristiques**, le package **System** est découpé entre la classe **Order** (dont les fonctionnalités sont aussi décrites précédemment) et la classe **Core**.

Le **Core** permet notamment, grâce à la méthode treatOrder, d'attribuer un **Courier** à un **Order**. Pour ce faire, il suit la **deliveryPolicy** choisie et stockée dans une variable dédiée. Le problème qui s'est rapidement posé à nous a été d'inclure le fait d'attendre qu'un **Courier** accepte un **Order** lui ayant été attribué.

En effet, la logique suivi lors d'une commande est la suivante :

- un **Order** est créé, son contenu mis à jour, puis finalisé par un **Customer**
- ce dernier est ajouté à la liste des **Order** à attribuer, et la fonction treatOrder se lance
- elle trouve un **Courier** disponible selon la **deliveryPolicy**
- si aucun **Courier** n'est disponible, arrête l'algorithme
- si un **Courier** a été trouvé, on attend qu'il accepte un **Order** ou non
- on réattribue ensuite les **Order** restants en conséquence

En terme d'implémentation, cette logique était moins simple. Nous avons donc choisi de créer une commande permettant d'accepter un **Order**. Ainsi, l'algorithme attribue autant d'**Order** que

possible, puis s'arrête. La fonction est ensuite appelée lorsqu'un **Courier** change son état (**onDuty** ou **offDuty**), accepte un **Order** (et par conséquent refuse les autres lui ayant été attribué). Cela permet d'attribuer les **Order** restant au fur et à mesure.

Les fonctions utilisées par l'interface sont aussi écrites dans le **Core**. En particulier, une variable *current_user* permet de stocker l'utilisateur connecté à l'application via la fonction **Login**.

Chapitre 5

Interface

L'interface réalisée devait être une interface en ligne de commande (CLUI) permettant à l'utilisateur d'entrer une série de **Command** et d'interagir avec l'application.

Pour réaliser cette interface utilisateur, nous avons utilisé une *interface* **Command**, implémentée par chaque commande, qui ont été réalisées sous la forme d'une classe par commande.
Celles-ci ont ensuite été stockées dans la classe **CommandHandler** dans une *HashMap* mappant les noms des commandes aux classes les représentant.

La classe **MainCLUI** contient la méthode *main* : elle est utilisable pour lancer l'application, et prend en entrée ce que l'utilisateur tape. Elle sépare la 1ere entrée des autres (séparées par un espace) qui sont stockées dans une liste d'arguments. Le **CommandHandler** est ensuite appelé pour traiter la commande.
Cette classe possède une méthode permettant de traiter les commandes mises en entrée : elle récupère la commande correspondante, puis exécute celle-ci en fournissant aussi les arguments correspondants. Cette classe possède également une méthode permettant de lire des commandes depuis un fichier texte, ce qui permet notamment d'implémenter la commande `startup` qui permet d'initialiser le système en lui fournissant des restaurants avec des menus etc...Cette méthode permet également de réaliser des fichiers texte de test, dont le fonctionnement sera détaillé plus tard.
A l'origine, le parsing des arguments des commandes nécessitait de mettre chaque argument sous forme de `String` comme cela : `registerRestaurant "Burger King" "bk" "45,66" "bkpass"`. Nous trouvions ce procédé fastidieux, et nous avons donc procédé à un plit par espace suivi d'une conversion en String pour s'abroger de ce procédé. Cependant, cela empêchait à présent d'entrer des arguments à deux mots comme `"Burger King"`, qui seraient alors traités comme deux arguments différents. Après un peu de recherche sur internet, nous avons trouvé une source (lien) qui nous a permis d'adapter le code pour parser la commande en cherchant si certains arguments sont entre guillemets. Les commandes peuvent donc prendre la forme `registerRestaurant "Burger King" bk 45,66 bkpass`.

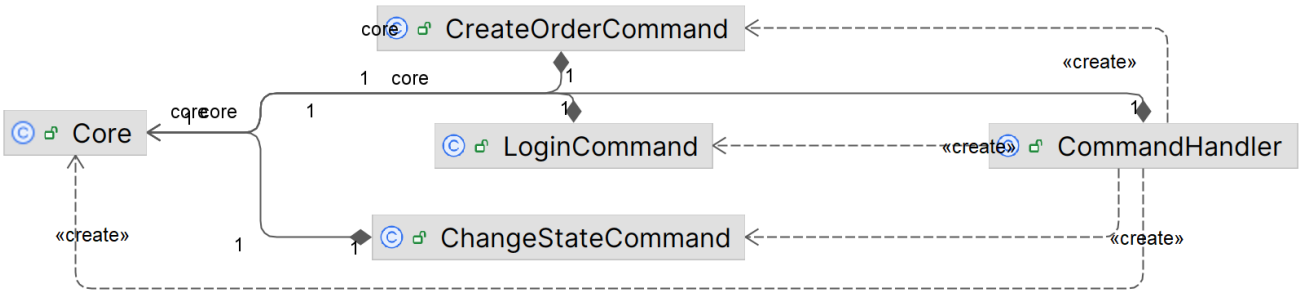

FIGURE 5.1 – Diagramme du package **clui** contenant quelques exemples de commande

Chapitre 6

Tests

6.1 Tests JUnit

Durant la conception de notre système, nous n'avons pas utilisé la méthode du **Test Driven Development (TDD)**. En effet, bien que cela soit une bonne pratique, nous ne nous sentions pas de le faire car nous avons beaucoup tatonné dans plusieurs directions au début du projet pour trouver par où commencer, puis par où poursuivre (nous avons d'ailleurs posé plusieurs questions en TD dans ce sens). Nous ne savions donc pas vraiment comment tester efficacement nos classes alors que nous n'avions pas tout conçu avant. Cependant, nous avons quand même réalisé des tests pour toutes les méthodes principales des classes principales, dans le package **tests**.

Ainsi, nous avons utilisé les tests JUnit en imaginant des cas et en créant des objets (**Restaurants**, **Dishes**, ...) temporaires pour tester les méthodes principales, en indiquant les valeurs attendues. Nous avons notamment utilisé ChatGPT pour être plus rapides et efficaces dans la création d'exemples, car c'est une tâche fastidieuse que nous estimons utile de déléguer à ChatGPT dans ce cas, avec les bonnes consignes évidemment.

6.2 Tests avec commandes

Nous avons écrit des fichiers textes lisibles par la commande *runtest* pour réaliser une série de commandes d'un coup. Ces fichiers textes respectent entre autres les scénarios proposés dans les use case du sujet. Nous détaillerons les actions réalisées par chacun de ces fichiers :

RegisterUser
`runtest registerUser.txt`

Ce fichier permet de tester la création d'utilisateurs. Il crée un **Customer** (sans être connecté) avec la commande *register* permettant à un client de s'enregistrer lui-même. Celle-ci le connecte directement. Il rentre ensuite son numéro de téléphone, puis son adresse mail, et indique qu'il souhaite être notifié d'offres spéciales, puis il se déconnecte.

Le CEO (existant à la création de l'application) se connecte pour enregistrer un **Courier**. Celui-ce se connecte, indique son numéro de téléphone, puis indique qu'il est *onDuty*.

AddItemMenu
`runtest additemmenu.txt`

Ici, on vérifie que les commandes pour ajouter des item au menu d'un restaurant fonctionne. Pour ce faire, on lance le *startup*, pour avoir un restaurant préexistant, puis on se connecte à celui-ci. On ajoute à son menu des **Dish**, on en retire un. On ajoute un **Meal** et on le remplit, on affiche son contenu, pui on enlève un plat et on supprime le **Meal**.

MealOfTheWeek
`runtest mealoftheweek.txt`

On vérifie le bon fonctionnement de l'ajout d'offres spéciales et les notifications.

On lance le *startup* puis on se connecte en tant que restaurant. On ajoute un **Meal** à l'offre spéciale, puis on se déconnecte. On se connecte ensuite à un **Customer** ayant précisé vouloir recevoir des notifications, ce qui affiche les messages reçus (montrés à la connexion). Vous devez donc voir un message indiquant l'existence de l'offre spéciale.

On se déconnecte, puis on se reconnecte en tant que **Restaurant** pour retirer l'offre spéciale.

CreateOrder
`runtest createOrder.txt`

On teste ici la création et le bon fonctionnement des fonctions côté **Customer** des **Order**.

On initialise, puis on se connecte en tant que **Customer**. On crée un **Order**, on y ajoute des **Dish** et des **Meal**, puis on finalise l'**Order**.

OrderAllocation
`runtest orderallocation.txt`

Ce fichier teste la fonctionnalité la plus technique de l'application : l'attribution des **Courier**.

On initialise avec *startup* et en créant un certain nombre d'**Order**, pour différents **Customer** et **Restaurant**.

On se connecte ensuite en tant que différents **Courier**, pour voir les **Order** qui leur ont été attribués.

La politique par défaut étant **FastestDelivery**, l'algorithme attribue les **Order** 0 et 4 à alikhan, les 1 et 5 à sofiarossi, et les 2 et 3 à annapetrov. alikhan accepte l'**Order** 0, et le 4 est réattribué à sofiarossi. Celle-ci accepte le 5, et ses deux autres sont réattribués à miguelsantos. Alikhan indique qu'il a livré l'**Order** 0 (il est donc à nouveau disponible, et à l'adresse du client). Miguelsantos accepte le 4. Alikhan reçoit alors le 1. Sofiarossi livre son **Order**. Annapetrov accepte le 3, le 2 est donc attribué à alikhan, qui accepte le 1, et le livre. Miguelsantos aussi.

On se connecte aussi en tant que manager, et on affiche les **Courier** dans l'ordre du nombre de commandes. On se connecte ensuite en tant que **Restaurant**, et on vérifie l'allocation de l'**Order** 0 (attribué à sofiarossi après la mise à jour des positions.

6.3 Utilité des tests

Cette section a pour but de témoigner de l'utilisé des tests. En effet, quasiment chaque test réalisé (que ce soit **JUnit** ou avec **Commandes**) a mené à des erreurs, soit de syntaxe dans le code, soit de conception des méthodes. Ces tests nous ont donc permis de corriger de nombreuses erreurs de notre part que Eclipse n'a pas remarqué.

Nous avons réalisé ces tests durant tout le projet, en testant quasiment chaque classe immédiatement après les avoir implémenté, ce qui a permis de construire le code en permanence sur des bases sûres, plutôt que de s'emballer dans la conception alors que le système sous-jacent est erroné.

Chapitre 7

Déroulement du projet

Le but de cette partie est de rendre compte du fonctionnement de notre binôme dans la conception de ce projet et de discuter de nos méthodes utiliser.

Tout d'abord, concernant la collaboration à deux sur un projet de code, nous avons immédiatement pensé à utiliser git, qui est évidemment fait pour ce cas. Cependant, nous avons au début essayé d'intégrer git à l'IDE Eclipse, et nous avons recolté beaucoup d'erreurs. Étants encore à ce moment là au début du projet, nous avons décidé de ne pas s'y attarder trop longtemps, car nous avions beaucoup de travail qui nous attendait. Nous n'avons donc pas utilisé git, et nous nous sommes contentés de travailler ensemble dans le salon (nous habitons ensemble) en s'envoyant les fichiers par messagerie. Ce fonctionnement n'était pas parfait mais il a suffi sans trop créer de problèmes.

Ensuite, nous souhaitions évoquer les méthodes utilisées pour implémenter tout le projet dans le code. Nous avons évidemment utilisé beaucoup de ressources sur internet (documentations, StackOverflow, etc...) mais également ChatGPT. ChatGPT possède ses limites, mais a été tout de même utile pour debugger rapidement certains problèmes, pour poser des questions sur l'utilisation d'objets que nous ne connaissions pas, et pour générer des objets pour les tests comme évoqué plus tôt.

Un tableau de répartition du travail peut être trouvé à la page suivante.

Tâche	**Personne**
Dish et sous-classes	Maxime
Dish et **Meal** Factory	Maxime
Meal et sous classes	Mathias
Menu	Mathias
User et sous-classes	Mathias + Maxime
Address	Maxime
Core et **Order**	Mathias + Maxime
Observer pattern	Mathias
CLUI CommandHandler et setup	Mathias
Execution fichiers textes	Maxime
Commandes	Mathias + Maxime
Strategy pattern pour les **Policy**	Maxime
Tests JUnit	Mathias + Maxime
Tests fichiers	Mathias
Javadoc	Mathias + Maxime
Diagrammes UML	Maxime

FIGURE 7.1 – Répartition du travail dans le binôme

Chapitre 8

Conclusion et discussion

Pour conclure, nous avons réussi à implémenter quasiment tout le système. Il ne manque en effet que quelques fonctionnalités mineures que nous avions pas le temps d'ajouter, et qui impliquaient d'utiliser un pattern déjà utilisé.

Nous aimerions également, dans cette partie, évoquer quelques choix que nous avons faits durant la conception du projet.

Tout d'abord, le sujet était flou sur l'ajout de la possibilité qu'un plat soit sans gluten ou non, et nous ne savions pas bien s'il fallait ajouter **gluten-free** comme un régime comme les autres (**Standard** ou **Vegetarian**), ou comme quelque chose à part entière. Cela dit, dans la réalité, un plat peut être par exemple végétarien et sans gluten, donc nous avons ajouté la possibilité de sans gluten dans un attribut à part entière représenté comme un booléen.

Ensuite, pour le traitement des commandes, le sujet souhaitait que le système permette à l'utilisateur de créer des commandes avec un nom, ce qui permet ensuite d'ajouter un objet à la commande en indiquant son nom. Cette fonctionnalité permet à l'utilisateur de faire plusieurs commandes différentes en même temps. Cependant, nous avons estimé que ce cas n'était pas courant, et nous n'avons pas implémenté cela, en se contentant de permettre à l'utilisateur d'ajouter des objets à sa commande directement après l'avoir créée.

Enfin, nous aimerions remercier Paolo Ballarini et Arnault Lapitre pour leur aide apportée.