
Compte-rendu Ingénierie logicielle

Fley Nicolas – Ravoux Corentin

IS1220 mai 2017

CONTENTS

1	Introduction	3
2	Design	5
2.1	Le bloc utilisateur	6
2.2	Design pattern rattachés au client	8
2.3	Gestion des plats et menus	10
2.4	Mise en place des Strategy pattern	11
2.5	Interface utilisateur	12
2.6	Coeur du système d'information	13
2.7	Autres Classes	13
3	Réalisation et discussions	14
3.1	Réalisation du programme	14
3.2	Problèmes rencontrés	15
3.3	Limites du programme, pistes d'amélioration	15
4	Tests	17
5	Lancement du programme et Scenario	18
5.1	prise en main	18
5.2	lancement des scénarios	20
6	Conclusion	22

1 INTRODUCTION

Le but du projet est de développer un logiciel qui sert de système d'information pour une entreprise de livraison de nourriture appelé MyFoodora. La livraison se fait par des coursiers en vélo qui peuvent s'inscrire et se désinscrire du programme. Les plats sont préparés par des restaurants qui s'inscrivent dans ce système. Les manager de ce système doivent être capables de le gérer au maximum. Enfin, les clients doivent pouvoir commander les plats proposés par les restaurants inscrits.

Une fois que le choix du client est réalisé, il doit payer le prix composé d'un total du prix de la commande fixé par le restaurant, du prix de service de livraison qui constitue le salaire du coursier et le pourcentage de marge sur le prix de la commande qui permet à MyFoodora d'assurer une rentrée d'argent. Le système doit aussi prendre en compte le coût de la livraison (salaire coursiers, amortissement équipements, ..) dans le calcul du profit.

Les restaurants doivent avoir la possibilité de rentrer des menus et des plats dans le système d'exploitation. Ils peuvent aussi mettre en place des offres spéciales avec des réductions plus importantes. Ces offres seront notifiées aux clients qui ont accepté de recevoir des notifications.



Pour réaliser ce projet, nous avons d'abord construit un diagramme UML de la structure du système d'information. Nous avons ensuite implémenté notre solution puis réalisé des tests pour chaque classe. Enfin, nous avons mis en place un scénario de test qui utilise une interface utilisateur sous forme de commande pseudo-Linux que nous avons conçus.

Le système d'information est consisté d'un cjur MyFoodora et de nombreuses autres classes qui représente les objets qui interagissent avec lui. Un Main permet de lancer le programme et l'interface utilisateur, il se trouve dans le package Core.

Pour avoir une solution la plus flexible possible, nous avons fait en sorte de respecter au maximum le principe OPEN-CLOSE (limitation du nombre de modification internes du code s'il

est amélioré). Pour cela, nous avons implémenté un nombre suffisant de classe pour décrire toutes les interactions possible et nous avons mis en place des Design pattern bien précis pour répondre de la meilleure façon possible aux problématiques du sujet tout en respectant ce principe.

2 DESIGN

Les figures 2.1 et 2.2, sont les deux parties du diagramme UML que nous avons mis en place pour créer la structure du système d'information.

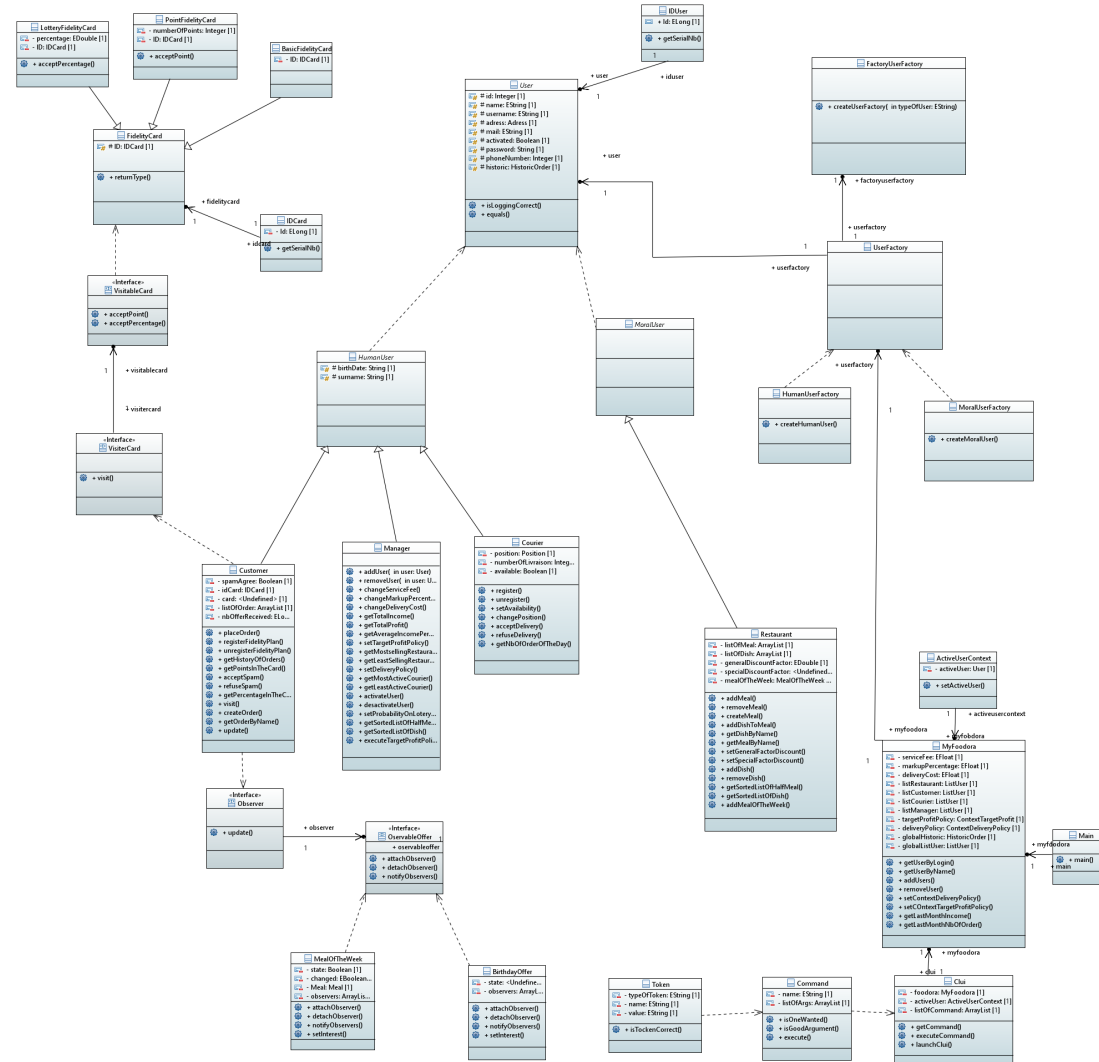


Figure 2.1: UML partie 1

Après une lecture de l'énoncé, nous avons rapidement dégagé les classes les plus importantes qui constituent les acteurs dans le système d'information (comme les Users), les classes utiles à la réalisation des Design Pattern, et les classes auxiliaires qui apporte de la clarté au code et qui permet une meilleure hiérarchie.

humain) et à l'ajout de nouveaux types (clients, managers, ...). Une usine d'usine (abstraite) va permettre de créer des usines de chaque genre d'utilisateurs. Une fois qu'une de ces deux usines est créée, elle permettra à son tour de créer des utilisateurs grâce à toutes les fonctions implémentées à l'intérieur de ces classes.

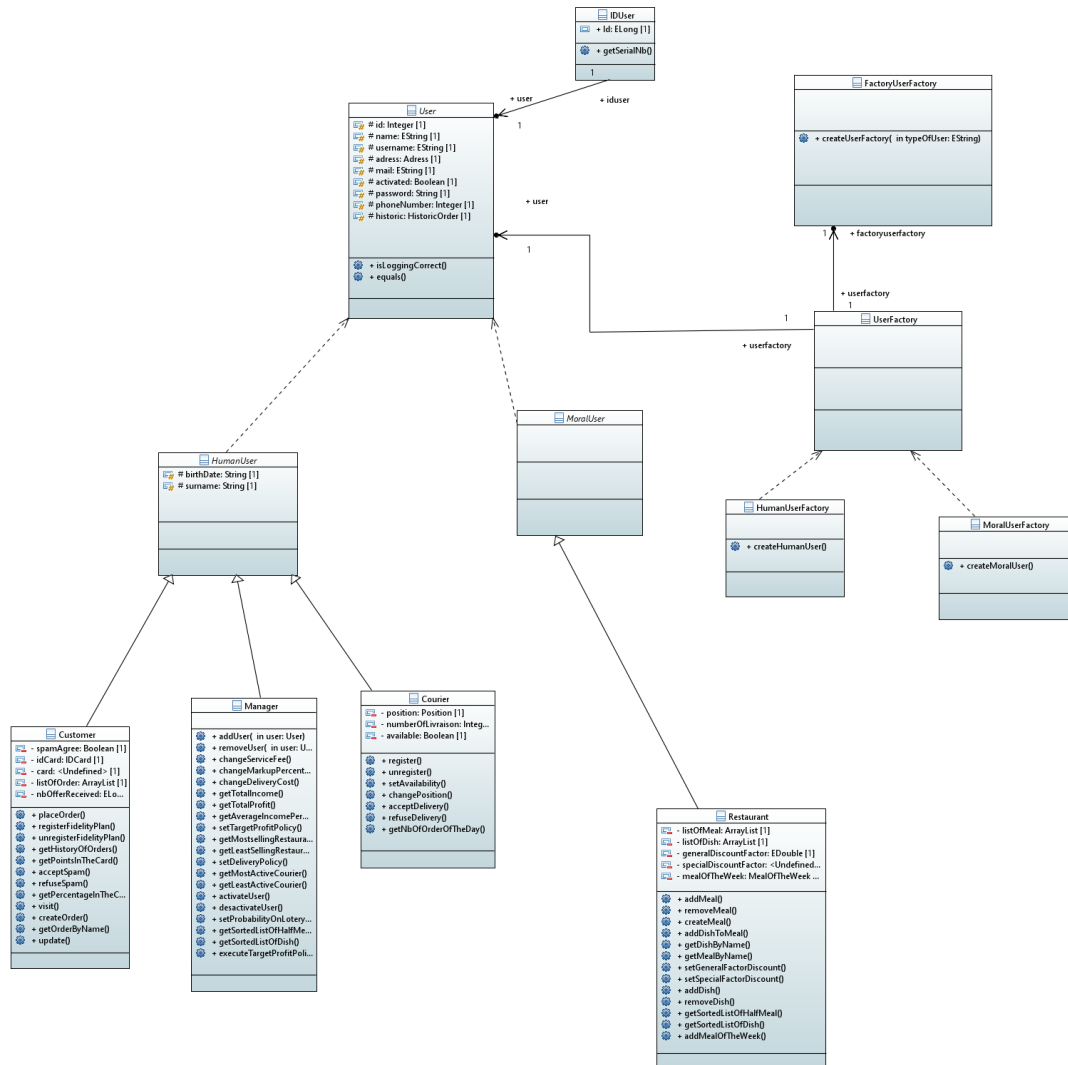


Figure 2.3: Bloc Utilisateur et son Factory Pattern

Pour créer les ID des utilisateurs nous avons décidé de mettre en place un Singleton pattern qui permet à MyFoodora d'avoir une seule et unique occurrence d'une ID pour chaque utilisateur, c'est nécessaire pour pouvoir utiliser l'ID comme une clé primaire. Ce singleton est donné par la Classe IDUser.

La classe User permet de définir la fonction login nécessaire pour chaque utilisateur pour entrer dans le système. Dans notre projet nous gérons la connexion d'un seul utilisateur à la fois. Dans les conditions réelles, il faudrait utiliser du multi-threading pour que différents utilisateurs puissent interagir avec le système simultanément.

La classe Manager contient les méthodes principales à la gestion du système, il permet notamment d'ajouter ou de retirer des utilisateurs, d'afficher les statistiques nécessaires à la bonne gestion de l'entreprise (activité des coursiers, ventes, profits, ventes des restaurants, ...) et la modification des différentes composantes du système (pourcentage de marge, coûts dédiés à la livraison, coût de service, stratégies de livraison, de profit visé). Il peut également consulter à tout moment les plats et les menus à deux pièces les plus vendus dans chaque restaurant.

La classe Restaurant contient les fonctions principales pour ajouter des plats à sa carte, pour ajouter des repas (full ou half), en indiquant de quel type de nourriture il s'agit (normal, végétarien, sans gluten). Il y a également les fonctions qui permettent au restaurant d'ajouter les offres (repas de la semaine par exemple) avec les promotions associées. Il peut aussi regarder à tout moment la liste par ordre croissant de menus à deux pièces les plus vendus et de plats à la carte les plus vendus. De plus, le restaurant peut à tout moment décider quelle réduction il accorde sur ces offres et sur ces menus classiques

La classe Courier contient les fonctions qui lui permettent d'indiquer s'il est disponible et s'il accepte la livraison d'une commande et s'il la refuse. MyFoodora doit aussi lui permettre de s'inscrire et se désinscrire du système quand il le souhaite. Il peut également changer sa position. Dans un programme réel se changement de position se ferait automatiquement en utilisant le GPS du téléphone du coursier par exemple ou en lui fournissant une puce GPS intégrée au vélo.

Enfin, la classe Client lui permet de commander les menus qu'il souhaite dans les restaurants qu'on lui propose après avoir affiché ceux qui étaient disponibles dans ce restaurant. Il lui permet également de souscrire à un plan de fidélité (carte de fidélité basique, à point ou en loterie). L'attribut spamAgree permet de savoir si le Client accepte de recevoir des offres de la part de MyFoodora comme les offres de la semaine ou les offres spécial anniversaire. Les clients peuvent aussi changer leur attribut SpamAgree par le biais d'une fonction.

2.2 DESIGN PATTERN RATTACHÉS AU CLIENT

Pour mettre en place le système d'offres, nous avons décidé de mettre en place un Observer pattern (cf fig. 2.4). Il permet aux offres qui sont des Observables de notifier les clients qui ont mis leur accord grâce à l'attribut SpamAgree. Avec ce design pattern, on peut ainsi ajouter de nouvelles offres en touchant au minimum au code déjà existant, il suffit de créer une nouvelle classe caractéristique de l'offre.

Avec ce système, lorsqu'un restaurant ajoute une offre, les clients ayant accepté le spam sont automatiquement notifiés de l'offre. Dans la réalité, il faudrait bien sur envoyer ces offres par mail.

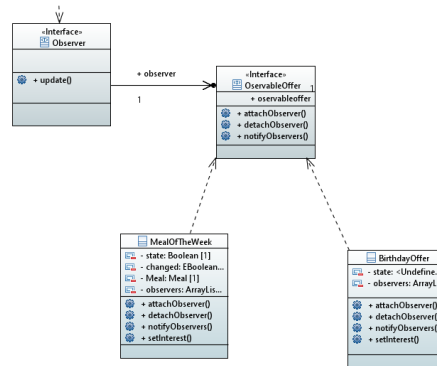


Figure 2.4: Observer Pattern

Pour le système de fidélité, il a fallu mettre en place trois type de systèmes de fidélité, matérialiser sous forme de carte de fidélité : une basique, une à point et une utilisant un système de loterie permettant au client d'avoir une chance d'avoir son repas gratuit lors de sa commande. Pour mettre en place ce système nous avons créé une classe abstraite Card et trois classes concrètes qui étendent Card (cf fig. 2.5).

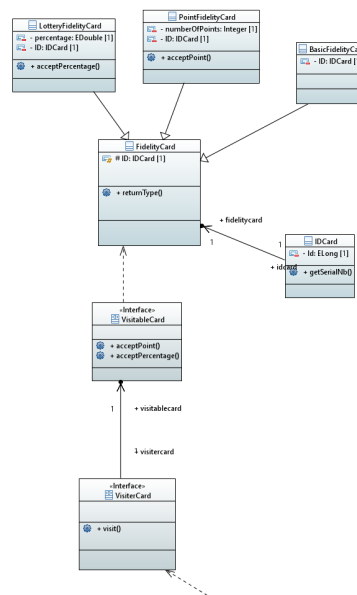


Figure 2.5: Cartes de fidélité

Ce système de carte de fidélité est facilement extensible si on veut rajouter un nouveau type de carte de fidélité dans le code. Nous avons également mis en place un Visitor pattern pour obtenir plus facilement le nombre de points sur la carte à point et pour pouvoir respecter le principe d'OPEN-CLOSE si l'on rajoute une autre carte utilisant un autre système de points.

L'ID de chaque carte est assuré d'être unique grâce au Singleton Pattern IDCard.

2.3 GESTION DES PLATS ET MENUS

Pour gérer les plats et les menus, nous avons décidé de créer deux classes abstraites Meal et Dish qui permettent d'étendre des classes de plats et de menus concrets (cf fig. 2.6). Meal et Dish sont eux-mêmes des extensions de la classe abstraite Item qui permet de regrouper les arguments communs et de pouvoir créer des listes de Meal et Dish. Grâce à ce système, on peut facilement ajouter des nouveaux types de plats (en plus de dessert, dish et starter) et des nouveaux types de menus (en plus de half et full) avec des modifications minimales de code. Les classes concrètes des menus sont des menus à 2 pièces (HalfMeal) ou à trois pièces (FullMeal) qui constituent les différents types de menus. Les classes concrètes des plats sont Dessert, Starter et MainDish qui permettent leur positionnement dans le menu lorsqu'ils sont ajoutés à celui-là. Chaque menus et plats possède un argument type de nourriture qui permette de dire si cet item est normal, sans gluten ou végétarien.

Pour créer ces différents plats et menus nous avons mis en place des factory Pattern assez simples qui permettent également le respect des principes OPEN-CLOSE.

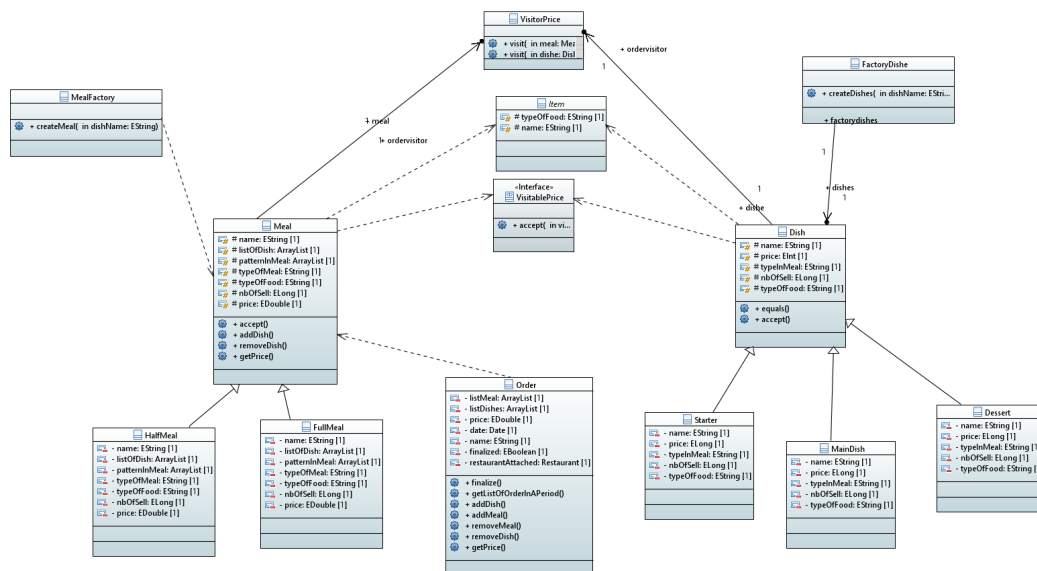


Figure 2.6: Structure des plats, menus et Orders

Les menus sont enfaite des compositions particuliers de plats qui permettent d'avoir une réduction. Une fois que le client a fait sa commande, on regroupe les différents menus et plats commandé dans une commande symbolisée par la classe Order. La classe Order a donc des fonctions permettant de lui ajouter les plats et menus et d'être finaliser à la fin de la commande du client.

Pour déterminer le prix de la commande et les prix des différents plats et menus, il a fallu mettre en place un pattern qui nous permettait de calculer le prix de chacun des éléments cités de façon différentes. Le Visitor Pattern est un excellent choix pour cela car il permet notamment de pouvoir calculer un autre prix sur une autre classe de façon différent tout en respectant le principe OPEN-CLOSE. Ainsi, on peut calculer le prix de la commande que le client doit payer.

2.4 MISE EN PLACE DES STRATEGY PATTERN

Il a également fallu mettre en place des Strategy Pattern (cf fig. 2.7) pour déterminer les stratégies à mettre en place par les managers pour faire fonctionner le système à leur guise. Le cahier des charges nous impose de pouvoir mettre en place différentes stratégie pour la livraison et pour le système de mise en place des prix.

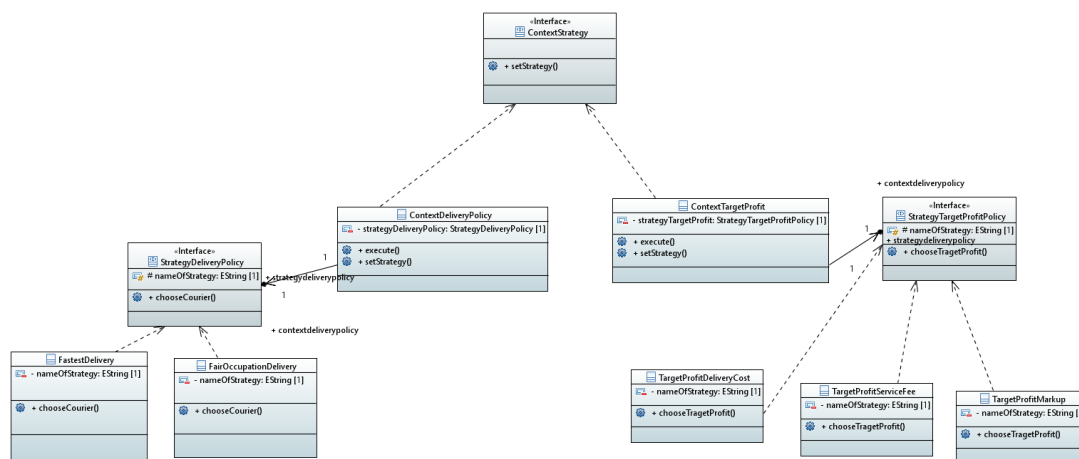


Figure 2.7: Strategy Pattern

Ainsi, on met en place un Strategy Pattern pour déterminer quel coursier va être choisi pour la livraison. On met en place une classe abstraite qui regroupe les différentes stratégies et qui possède les fonctions communes à toutes les stratégies. Ensuite on crée une classe de contexte qui permet de choisir la stratégie à mettre en place selon un contexte donné par le manager. On code les différentes stratégies concrètes qui étendent la classe abstraite. Les stratégies pour la livraison choisit le coursier selon s'il est le moins occupé ou le plus près. Le

manager peut ensuite modifier ce système de choix grâce aux fonctions qui lui sont associées.

On fait la même chose pour les stratégies de prix avec un schéma de stratégie similaire. Cette fois, cette stratégie permet au manager de déterminer quel paramètre il doit imposer (coût de livraison, coût de service, pourcentage de marge) pour que MyFoodora réalise le profit souhaité en se basant sur les ventes du mois dernier.

2.5 INTERFACE UTILISATEUR

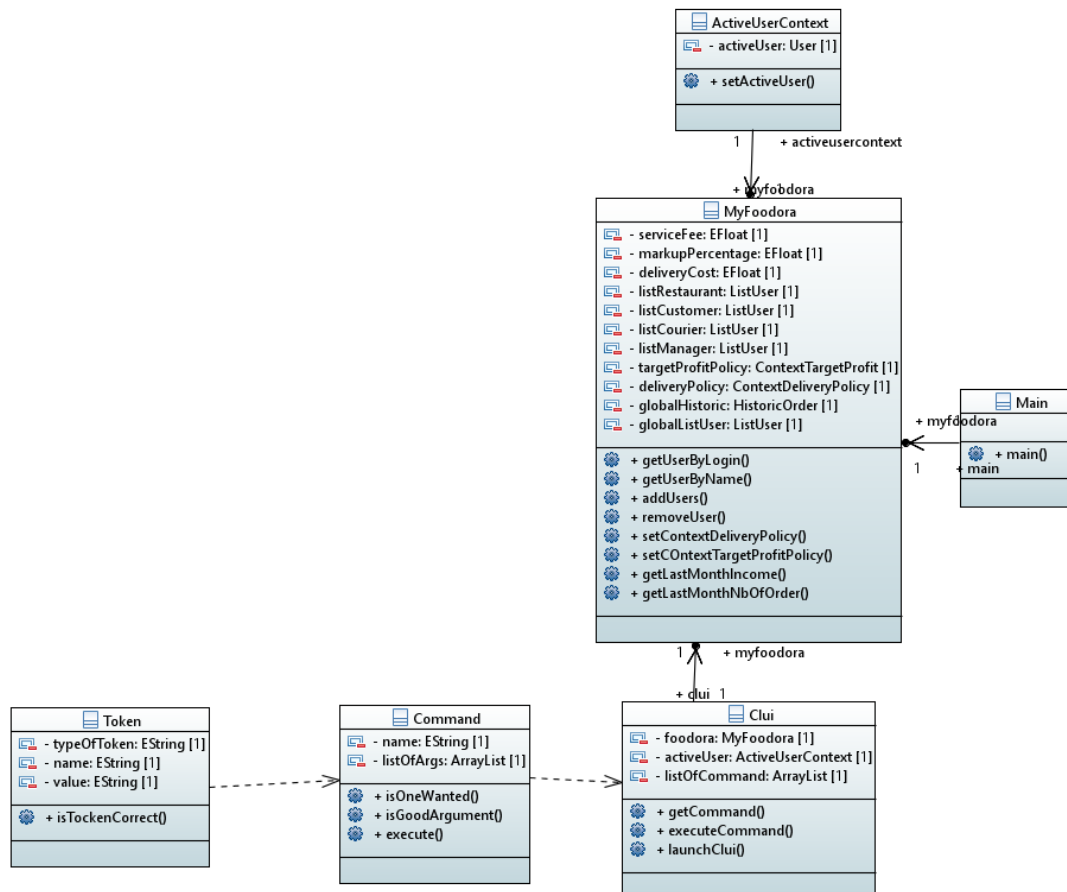


Figure 2.8: Coeur du programme et interface utilisateur

Nous avons dû mettre en place une interface utilisateur qui permet aux différents utilisateurs du système d'information de communiquer avec lui. Pour cela, nous avons créé une un package CLUI (Command Line User Interface) (cf fig. 2.8) qui contient les classes nécessaire à la bonne interprétation des commandes du cahier des charges et qui fait office de console Linus-Like pour pouvoir taper les commandes.

Nous avons créé une classe `Command` qui permet de décrypter une commande en un nom et une liste d'arguments. Nous avons également créé une classe `Clui` qui permet le lancement d'une console sur l'interpréteur Java et qui récupère chaque commande pour les transcrire en fonction à exécuter par le système d'information. Cette classe regroupe toutes les commandes possible dont notamment la commande `help <>` qui permet d'afficher de toute les affichées. Nous avons également créé une classe `Token` qui permet de récupérer les arguments des commandes tapées pour les mettre sous forme de `Token`. Cette classe permet de valider si le nombre de token et si les arguments sont les bons associées à une commande en particulier.

De plus, nous avons mis dans ce package des fonctions qui permettent de récupérer rapidement la confirmation ou la réponse d'un utilisateur à une question.

2.6 COEUR DU SYSTÈME D'INFORMATION

Le coeur du système contient une classe `MyFoodora` (cf fig. 2.8) qui symbolise le système d'information lui-même. Cette classe contient tous les arguments nécessaires pour que le programme tourne. Cette classe contient comme arguments des listes d'utilisateurs `ListUser`, une pour chaque type d'utilisateur et une qui contient tous les utilisateurs à la fois. Il enregistre aussi le pourcentage de marque, le coût de livraison, le coût de service, les stratégies de livraisons et de profit. Et il contient aussi la liste de toutes les commandes qui ont été passé durant le lancement du programme. Cette classe enregistre à chaque instant l'état du système d'information.

Cette classe contient aussi les fonctions utiles au bon fonctionnement du programme. Il permet d'ajouter et d'enlever des utilisateurs dans le système, de mettre en place les différentes stratégies détaillées plus haut, de récupérer les utilisateurs avec leur login et d'obtenir les statistiques de vente du système.

Nous avons également codé un `Main` qui permet le lancement du programme. Son utilisation est détaillée dans le paragraphe Lancement du programme et Scenario.

2.7 AUTRES CLASSES

Nous avons également codé d'autres classes qui nous permettent de mieux structurer le code et d'obtenir un meilleur contrôle sur les objets utilisés. Nous avons décidé pour cela de créer les classes `Date`, `Adress`, `Position`, `Period`, `HistoricOrder` et `ListUser`.

Nous avons également décidé de programmer des classes `Exception` qui permettent de mieux maîtriser les erreurs de saisies et les erreurs lorsque le programme tourne pour par exemple éviter que le programme plante et pour qu'il renvoie une erreur à la place.

3 RÉALISATION ET DISCUSSIONS

3.1 RÉALISATION DU PROGRAMME

Nous avons codé notre programme grâce à une méthode assez classique de programmation. Nous avons d'abord élaboré un design partiel mais assez complet de l'architecture du programme grâce à un diagramme UML sur Papyrus. Une fois l'architecture mis en place, nous avons codé certaines parties du programmes chacun de notre côté que nous avons sauvegardé sur Internet grâce à une programmation de type Git permettant la pérennisation de notre code et un partage des tâches sans trop de conflits.

Une fois la structure à peu près réalisée, nous avons codé les fonctions du programme. Puis, nous avons tester les parties du programme que l'autre avait fait pour permettre de comprendre mieux la partie du code que nous n'avions pas codé et pour essayer d'éprouver au maximum les fonctions que l'autre a programmé. Pendant que nous programmions, nous avons également renseigné la Javadoc du programme.

Enfin, nous avons programmé l'interface utilisateur ensemble et le Main. Puis nous avons réalisé les commandes de l'interface utilisateur et nos différents scénarios pour mettre le programme en situation de test sous condition réel.

De plus, nous avons écrit le rapport sous LaTeX en se répartissant les tâches : celui qui a réalisé le design d'une partie devait décrire sa solution.

La table 3.1 indique la répartition du travail lors de ce projet.

Table 3.1: Répartition du travail

Partie du code	Architecture	Design	Tests
Utilisateurs	N/C	C	N
Pattern Utilisateurs	N/C	C	N
Plats et menus	N/C	N	C
Strategy Pattern	N/C	C	N
Interface	N/C	N	C
Coeur	N/C	N/C	C
Autres	N/C	N/C	N/C

La notation N signifie que la partie concernée a été (en grande partie) réalisées par Nicolas Fley et C par Corentin Ravoux. N/C signifie que la répartition des tâches sur la partie concernée a été équitable.

3.2 PROBLÈMES RENCONTRÉS

Nous n'avons pas réussi à mettre en place l'offre d'anniversaire qui demandait une gestion du temps en Multi-Threading. Ce n'est pas le multi-threading qui nous a posé problème mais plutôt le système de notification des utilisateurs, c'est-à-dire, quand et par quel moyen nous envoyons l'offre au Client. Dans une version plus avancée du programme, nous aurons pu réaliser ce système d'offre mais nous avons manqué un peu de temps.

Nous avons également eu du mal à mettre en place les Strategy Pattern car le choix des caractéristiques de MyFoodora en fonction du profit visé et le choix des coursiers pour la livraison sont deux fonctions qui en appellent beaucoup d'autres. Nous avons passé beaucoup de temps sur ces stratégies mais nous avons réussi à les programmer car elles permettent d'avoir un système d'information complet avec des fonctions assez avancées.

Nous avons également passé beaucoup de temps sur le système de prix des plats, menus et Order. Il a fallu que la classe qui nous permette de mettre un prix sur ces articles puisse s'adapter à chacun d'entre eux. Nous y sommes également parvenus avec du temps.

De plus, la principale difficulté que nous avons rencontrée a été la mise en commun du travail et le recodage de fonction mal codées voir même le changement de structure du code. Malgré le fait que nous utilisions Git pour fusionner notre code, il y a eu beaucoup de fois où nos modifications se chevauchaient et la mise en commun du travail devenait donc très fastidieuse et chronophage. Malgré le fait que nous ayons fait une architecture très complète sous UML de notre solutions, nous nous sommes rendu compte qu'il était impossible de prévoir complètement cette architecture avant de commencer à coder. Nous avons donc dû à plusieurs reprises modifier notre architecture et certaines fonctions ce qui a été très chronophage également.

Nous avons également rencontré des problèmes face au manque de temps. En effet, la mise en place des commandes de l'interface utilisateur, les tests JUnit et les scénarios sont des parties du programme qui prennent beaucoup de temps.

3.3 LIMITES DU PROGRAMME, PISTES D'AMÉLIORATION

Notre programme est assez complet mais présente des limites qui ne lui permettent pas d'être utilisable par une entreprise. Dans ce paragraphe, nous allons donner quelques pistes d'amélioration de notre programme qui aurait pu lui permettre d'être commercialisable.

Tout d'abord, par manque de temps, il nous a été impossible de créer une interface visuelle GUI pour les utilisateurs du programme. Pour qu'un produit soit commercialisable, à part pour les logiciels très spécialisés destinés à des connaisseurs en informatique (par exemple, les codes de calculs, ...), il faut qu'un logiciel présente une interface visuelle claire, simple à utiliser et esthétique.

De plus, notre programme est conçu de telle sorte que l'on ne peut pas avoir plusieurs Utilisateurs du système connectés en même temps. Nous aurions pu régler ce problème en implémentant une méthode de multi-threading ce qui se fait bien sous Java. Pour cela, il aurait également fallu prévoir les parties du code qui aurait pu être en conflit entre les différents utilisateurs qui utilisent simultanément ce logiciel. Ce problème se résout facilement par la mise en place de Lock ou par l'utilisation des méthodes/blocs/variables synchronise.

Le programme ne comprend également pas d'offre spéciale anniversaire. Nous aurions en faire une ainsi que d'autres offres du même type souvent présente dans les menus du restaurant. De plus, dans un vrai système d'information, il faudrait envoyer ces offres au client par mail ou par le biais d'une application.

Pour améliorer le programme, il est également possible de rendre la classe MyFoodora serialisable. Cela permettrait d'enregistrer le système à un moment donné pour permettre de faire une sauvegarde du système à un instant et reprendre le système dans cette configuration. Cela permettrait de pouvoir stopper le système sans perdre les données qui lui sont associées et permettre de faire une sauvegarde.

4 TESTS

Nous avons essayé de regrouper les différentes classes dans des packages pour une meilleure lisibilité du code.

Pour réaliser des tests, nous avons créé un package regroupant tous les JUnit test de notre programme. Pour chaque package, nous avons créé une classe de test et nous en avons créé plusieurs pour les packages qui contiennent beaucoup de classes avec des fonctions à tester (par exemple les Utilisateurs).

Pour chaque test, nous avons testé les fonctions les plus importantes de chaque classe, c'est-à-dire les fonctions qui possédaient une interaction avec d'autres classes que celle dans laquelle elle se trouvait. Nous avons testé les fonctions permettant de vérifier le bon fonctionnement des design Pattern. Nous avons également testé toutes les classes qui définissaient une structuration importante de la classe dans laquelle elle se trouvait. Enfin, nous avons testé les Exceptions et les fonctions principales de l'interface Utilisateur. Avec ces critères, la plupart des fonctions ont été testées à part les plus simples comme les Getters et les Setters.

Pour ces tests, nous avons essayé de réaliser des petits Main. C'est-à-dire que nous avons fait en sorte que chaque fonction test se déroule comme s'il se déroulait comme une petite partie du Main principale. A chaque nouvelle fonction test, nous avons essayé, autant que faire se peut, d'appeler une instance du cœur du programme surtout pour tester les classes Utilisateurs qui possèdent beaucoup de fonctions très développées.

Les classes les plus difficiles à tester sont les classes Utilisateurs qui possèdent beaucoup de fonctions assez complexes ainsi que les stratégies de livraison et de choix de caractéristiques par rapport au profit souhaité. Par exemple pour les managers, il a fallu tester toutes les fonctions qui interagissent beaucoup avec les autres classes. Nous avons testé l'ajout et le retrait d'utilisateurs, les différents changements des caractéristiques de MyFoodora et l'obtention des statistiques utiles à la gestion du système d'information. Les stratégies ont également été dures à tester car les fonctions mises en place dans ces stratégies sont assez complexes.

5 LANCEMENT DU PROGRAMME ET SCENARIO

5.1 PRISE EN MAIN

Le programme est composé de 14 package : Cards, Cli, Commands, Core, DeliveryStrategy, Exception, Item, Offers, Order, Others, StrategyProfit, Test, User, UserFactoryPattern.

La structure des cartes de fidélités et du Visitor Pattern associé est dans le package Cards. Tout ce qui concerne l'interface utilisateur se trouve dans le package CLi notamment la classe qui contient l'interface elle-même. Toutes les commandes sont recensées sous forme de classe dans le package Commands. Les deux stratégies pattern qui permettent de choisir automatiquement le coursier et qui détermine les caractéristiques du système à partir du profit visé sont implémenté dans les packages DeliveryStrategy et StrategyProfit. Toutes les exceptions permettant de mieux gérer les erreurs de saisie sont dans le package Exception. La gestion des plats et des menus sont dans le package Item. Les offres avec le Observer Pattern sont dans le package Offers. Les commandes réalisées par les clients sont dans le package Order. Les utilisateurs et leur Factory Pattern associé sont dans les packages User et UserFactoryPattern. Enfin, tous les tests sont implémenté dans le package Test.

Le coeur de programme est contenu dans Core. Pour lancer le programme, il faut ouvrir la classe Main de ce package et faire Run.

Une fois le Main lancé, l'interface Utilisateur se lance dans l'interface I/O de Java. Pour tester le programme, le testeur pourra alors taper la commande :

help

Cette commande permet d'afficher toutes les commandes disponibles du système. Le testeur pourra alors tester s'il le souhaite quelques commandes ou lancer un scénario comme indiqué dans le paragraphe précédent.

Liste des commandes disponibles :

- login <username> <password>
- logout <>
- registerRestaurant <name> <address> <username> <password>
- registerCustomer <firstName> <lastName> <username> <address> <password>
- registerCourier <firstName> <lastName> <username> <position> <password>
- addDishRestaurantMenu <dishName> <dishCategory> <foodCategory> <unitPrice>
- createMeal <mealName>

- addDish2Meal <dishName> <mealName>
- showMeal <mealName>
- saveMeal <mealName>
- setSpecialOffer <mealName>
- removeFromSpecialOffer <mealName>
- createOrder <restaurantName> <orderName>
- addItem2Order <orderName> <itemName>
- endOrder <orderName> <date>
- onDuty <username>
- offDuty <username>
- findDeliverer <orderName>
- setDeliveryPolicy <delPolicyName>
- setProfitPolicy <ProfitPolicyName>
- associateCard <userName> <cardType>
- showCourierDeliveries <>
- showRestaurantTop <>
- showCustomers <>
- showMenuItem <restaurant-name>
- showTotalProfit<>
- showTotalProfit <startDate> <endDate>
- runTest <testScenario-file>
- help <>

5.2 LANCEMENT DES SCÉNARIOS

Pour lancer un scénario il faut utiliser la commande :

runtest <testScenario – file>

En renseignant le nom du fichier scénario dans la commande, cela lance les commandes référencées dans le fichier scénario.

Un premier fichier ini permet de charger une configuration de base du système. Pour cela le testeur peut taper dans l'interface utilisateur :

runtest scenario/my_foodora.ini

Ce fichier permet d'ajouter au système un manager de plus que le CEO déjà lancé par le Main, cinq restaurants, trois coursiers et sept clients. Il permet également d'ajouter des plats et des menus aux cartes des restaurants ajoutés.

Une fois ce scénario d'initialisation lancé, le testeur peut lancer le scénario test n°1 qui permet de tester toutes les commandes du programme. Le lancement de ce scénario se fait en tapant la commande suivante :

runtest scenario/testScenario1.txt

Ce test est très élaboré et suit la procédure suivante :

- Le CEO se log et affiche la liste des clients, des coursiers et des restaurants triés par ordre de nombre de ventes.
- Ensuite, le CEO enregistre quelques autres utilisateurs : deux restaurants, deux coursiers et deux clients.
- Le CEO affiche une nouvelle fois la liste des restaurants triés et des clients avant de se déconnecter.
- Un des restaurants du système se log et ajoute à sa carte plusieurs plats.
- Il crée plusieurs menus et ajoute les plats associés à chaque menu avant de se déconnecter (le login d'une nouvelle entité implique le logout de l'entité connectée précédente grâce à la classe `ActiveUserContext`)
- Les autres restaurants qui sont dans le système se connectent également et ajoutent de même des plats et des menus dans leur carte avant de se déconnecter.
- Chacun des restaurants se connecte à leur tour et ajoute une offre menu de la semaine avant de se déconnecter.

- le CEO se connecte une seconde fois et affiche la liste des restaurants triés pour voir leurs menus et plats.
- Le programme associe des cartes de fidélités à deux clients pour pouvoir tester ces cartes.
- Un client se connecte au système, crée une commande et ajoute les items qu'il désire à sa commande avant de finaliser sa commande.
- Un autre client fait la même chose avec plusieurs commandes différentes.
- Une nouvelle commande est réalisée par le client précédent pour tester le calcul du profit.
- A ce moment-là, aucun coursier n'est disponible
- le CEO se connecte et libère un coursier. Le système de choix de coursier et de livraison se réalise automatiquement et le client est livré.
- Le CEO souhaite voir le profit total, et l'affiche une nouvelle fois en changeant les stratégies de profit.
- Il affiche aussi le profit réalisé sur une certaine période.
- Il affiche ensuite la liste des coursiers et la liste des coursiers classés par ordre croissant puis décroissant de nombre de livraison.
- le CEO libère un autre coursier et voit comment les livraisons avancent en affichant la liste des coursiers avec leur nombre de livraison
- Il fait de même en rendant ce coursier non disponible.
- Enfin, le CEO affiche la liste des clients avec le nombre de livraison qui a été réalisé.

6 CONCLUSION

Nous avons réussi à mettre en place une première version assez solide d'un système d'information assez complexe.

Ce programme pourra être amélioré sur quelques aspects mais il est solide dans le sens où il n'y a pas besoin de modifier beaucoup l'architecture du projet pour obtenir un projet commercialisable.

Ce projet nous a permis de mieux appréhender le codage d'un vrai programme et de nous améliorer beaucoup en programmation orienté objet. Il a été utile pour acquérir une même maîtrise de Java et de la logique de conception d'un logiciel. Cela sera sans aucun doute très utile voir nécessaire dans la suite de notre cursus.