Theocharis Zachariadis
Anastasia Bouzdine

Mars-Avril 2017

[IS 1220 - RAPPORT FINAL]

MY FOODORA- FOOD DELIVERY SYSTEM

CentraleSupélec

SOMMAIRE

PARTIE 1

1. Introduction

Notre projet, intitulé « *MyFoodora* » représente un système de livraison de repas entièrement codé en Java. Il mêle donc à la fois une réflexion sur la structure de la solution nécessaire pour répondre aux exigeances d'une plateforme de commande/livraison, mais également une application techniquer directe par un code source en Java.

Ce projet a été extrêmement enrichissant tant au niveau de la réflexion qu'il entraîne (quelles sont les structures et les design pattern en Java les plus adaptées?) qu'au niveau des compétences techniques qu'il exige (implémentation des classes, stratégies).

Bien évidemment, la structure même exigée de la solution a nécessité un travail incrémental. Nous avons tout d'abord chacun tracé notre représentation du système avec les liens entre les différents interlocuteurs, puis confronté nos points de vue sur le sujet. La principale difficulté a résidé dans le choix parmi la multitude de solutions possibles. Bien que nous soyons guidé dans l'approche du système, il pouvait exister de nombreuses solutions répondant aux exigeances du système – le choix devait prendre en compte la flexibilité permise par la solution choisie.

Chaque design pattern étudié en classe pouvait être utilisé à de multiples fins : le plus dur a été de choisir les plus adaptés et d'évaluer leur nécessité. Ensuite, une fois le squelette du système tracé, la répartition des tâches nous a permis de conserver une certaine liberté dans l'implémentation des méthodes.
Ainsi, nous avons pu petit à petit rajouter des méthodes de manière incrémentale à partir des classes basiques développées, et relier les diverses instances du système entre elles.

Nous avons ainsi appris l'importance de fixer un cadre commun (le squelette du système), que nous pourrions ensuite développer chacun à notre intérêt. Le fait de partager le travail a mis en exergue la nécessité de commenter à chaque fois le code créé pour permettre au partenaire de suivre son évolution.
Tester notre programme nous a paru être une étape difficile, car elle demande une rigueur irréprochable et surtout de la motivation – après avoir codé une implémentation du système pendant quelques heures, il apparaît difficile de devoir recoder toutes ses méthodes pour la tester, bien que cela soit indispensable pour tout bon programmeur.

Aussi, l'objectif de ce rapport est de présenter à la fois notre solution – *Comment elle répond aux attentes et exigeances du système ? Quels patterns avons-nous choisi et pourquoi ?* - mais également les difficultés que nous avons pu rencontrer et la manière dont nous nous avons du nous adapter.

Nous commencerons par introduire le contexte du projet, et la manière dont nous l'avons analysé, pour ensuite développer les choix que nous avons effectués (design/structures), l'implémentation de notre solution, les tests effectués et les résultats finalement obtenus.

2. Contexte

L'ubérisation de notre société est un phénomène récent, mais qui ne cesse de prendre de

l'ampleur avec le développement de nouvelles technologies de plus en plus pointues. Aussi, des plateformes de mise en contact direct émergent un peu partout, permettant aux professionnels et aux clients d'utiliser des services de manière quasi-instantanée.

De tels systèmes, comme par exemple Deliveroo ou encore Foodora, connaissent un succès grandissant du fait de leur simplicité d'organisation et flexibilité.

L'objectif de ce projet a été de développer une solution codée en Java pour décrire et organiser un système complet de livraison de repas sur l'exemple même de Deliveroo et Foodora.
Il s'agit, par ce projet, de développer des compétences de programmation indispensables en Java et de les appliquer sur un cas concret, le système *MyFoodora* créé dans ce projet.

En quelques mots, il s'agit de modéliser un tel système par le biais de plusieurs utilisateurs reliés entre eux à travers le cœur du système MyFoodora :
- les restaurants qui proposent une carte composée de menus (complet ou demi) et d'items végétariens, standards et sans gluten, qu'il est possible de sélectionner.
- le client qui passe la commande et choisit ses produits, et peut bénéficier d'une carte de fidélité
- un coursier qui se charge de livrer la commande du restaurant au client
- le(s) manager(s) qui gérent le système et peuvent vérifier, calculer les différentes variables du système (profit, recettes, marge...) et choisir des stratégies pour augmenter leur rendement, ou encore trier les produits « phares » du restaurant.

Le cœur du système se charge de relier ces différents interlocuteurs et permet leur interaction pour en faire une véritable plateforme de commande, paiement, livraison.

Répondre aux exigeances du système, c'est à dire connecter les utilisateurs entre eux de manière quasi-instantanée, a nécessité de dessiner un squelette de notre système (sur l'exemple des diagrammes UML) pour définir les interactions au sein du système, avant de commencer à coder.

Nous avons donc réfléchi dans la partie suivante au design que l'on choisirait pour implémenter notre solution, et qui correspond à l'analyse du sujet que nous avons faite.

3. Analyse et Conception

Nous avons analysé le problème sous trois angles différentes, nous permettant à chaque fois de compléter notre vision : la carte du restaurant, les utilisateurs du système et le système en lui-même.

- *Carte du restaurant*

Elle propose des « meals » (full ou half- entrée/plat ou plat/dessert) qui sont des combinaisons d'items, eux aussi disponibles séparement à la carte. Ainsi, chaque item possédant un type particulier (végétarien ou standard) ainsi qu'une contenance en gluten qui lui est propre. Donc les meals possèdent à leur tour un type, qui dépend de la composition de leurs items.

Pour répondre à cette exigence, il nous a semblé logique de créer des classes abstraites, et de relier les items et les meals par un **Observer Pattern** : cela étant, toute modification d'un item entraînerait automatiquement la mise à jour du meal le contenant.

Un **Factory Method Pattern** a également été proposé pour la création des meals et items, car il s'agit là d'un pattern qui permet plus de flexibilité dans l'ajout de nouveaux items et de meals.

<<UML diagramme – Meals>>

De par le statut un peu particulier du menu spécial de la semaine, nous avons finalement choisi de construire une classe à part pour « Meal Of The Week » bien que dans l'idée initiale nous pensions en faire une sous-classe de Meal ou RestaurantMenu.
Nous nous sommes inclinés vers ce choix pour la simple raison que le Meal Of The Week peut être une combinaison d'items – et c'est d'ailleurs la solution que nous avons privilégié, puisque une combinaison d'items est aussi un meal.
En faire une classe à part permet au Restaurant d'éditer son menu spécial facilement, et indépendamment de sa carte « RestaurantMenu ».

- *Les utilisateurs de MyFoodora*

Quatre instances utilisent principalement *MyFoodora* : les managers, les clients, les restaurants et les coursiers. Bien que chaque instance possède des caractéristiques qui lui sont propres, et des actions qu'elle seule peut effectuer, ces instances dérivent toutes d'une classe abstraite « Utilisateur » car elles partagent des caractéristiques communes (le login, le mot de passe, l'identifiant, l'adresse par exemple).

<<UML diagramme - Users>>

De la même manière, un **Factory Method Pattern** permet de créer facilement ces utilisateurs à partir d'une *User Factory* (abstraite), et les classes Factory concrètes (Restaurant, Manager, Customer, Courier), et d'éditer les utilisateurs de manière flexible.

En ce qui concerne les caractéristiques propres de chaque utilisateur, nous ne détaillerons pas ici les attributs et méthodes propres à chacun car elles suivent la logique proposée dans les consignes du projet.
Toutefois, nous expliquons notre choix concernant la classe Manager : nous avons choisi de ne pas surcharger cette classe avec les méthodes propres du Manager (calculer le profit total, déterminer les frais de service suivant une stratégit de profit particulière, etc.) pour la conserver lisible.
Ces méthodes font partie du système interne « *MyFoodora* » car elles en utilisent des attributs principaux (profit de MyFoodora, frais de service, pourcentage de marge) – seul le manager peut y accéder à condition d'être connecté (if current_user instanceof Manager).

Par contre, nous soulignons le lien existant entre Restaurant et Customer par l'interface **ObserverC** qui permet de notifier tous les clients ayant accepté les notifications d'une nouvelle offre spéciale, c'est à dire de la mise en ligne par le restaurant du menu spécial de la semaine.
L'interface ObservableC n'a pas paru nécessaire et a permis de simplifier le code, puisque les Observer (clients) sont uniquement « notifiés » par le restaurant qui lance le menu spécial – il n'y a pas d'Observable concret.

- *Le système interne « MyFoodora »*

Il s'agit là de la partie qui nous a paru la moins évidente, car c'est réellement de ce système interne que dépendent les connexions entre nos classes.
Le système *MyFoodora* doit en premier lieu contenir une archive de tous les utilisateurs, stockée sous forme de listes, mais également permettre à tout nouvel utilisateur de se connecter au système et d'accéder aux informations qui lui sont réservées.

<<UML diagramme- MyFoodora core>>

Bien sûr un tel système ne peut pas exister en plusieurs « exemplaires » ; d'où la nécessité d'en faire un Singleton. Or réaliser un Singleton suppose y accéder uniquement par la méthode getInstance(), et nécessite un constructeur privé.
La majeure difficulté de ce choix, bien qu'indispensable, se pose au niveau des tests, qui doivent générer à chaque fois une instance pour vérifier son bon fonctionnement, ce qui entraîne parfois des blocages.

Par ailleurs, il était nécessaire de bloquer l'accès à certaines informations si l'utilisateur connecté n'était pas destiné à y accéder : comment faire ?
Nous avons privilégié une solution simple (if current_user instanceof ..., else return « No access »), mais qui ne permet pas à plusieurs utilisateurs d'instances différentes d'être connecté (pas de multithreading) et peut donc limiter fortement l'accès : chaque utilisateur doit donc se connecter puis se déconnecter à tour de rôle.

Bien sûr, toutes les méthodes exigées par le système (calcul du bénéfice, des recettes, choix des politiques de livraison, des politiques d'atteinte de bénéfice) que l'on ne détaillera pas ici, font partie intégrante du système et sont accessibles aux managers.

Une méthode cruciale de *MyFoodora* repose bien sur l'attribution d'une commande à un coursier en tenant compte de la politique de livraison choisie.
Cela a nécessité la création d'une classe *Order* et d'une classe *DeliveryRequest*, qui permettait de relier le Client et le Coursier par le biais d'un booléen d'acceptation ou non de la livraison par le coursier, une fois la commande payée.

Pourquoi distinguer ces deux classes ?
Cela nous a permis de séparer une prise de commande d'une commande achevée et donc payée : distinguer la commande en cours (modifiable) de la commande terminée, et donc prête à être acheminée.
La classe *Order* contient toutes les informations liées à la commande comme sa composition, sa date, son prix intermédiaire, le bénéfice qu'en tire *MyFoodora* et le restaurant, le prix final à payer et surotut la méthode de paiement qui redirigie automatiquement vers le système pour allouer la commande payée à un coursier.

- *Les politiques utilisées par MyFoodora*

Il nous paraît important de souligner les stratégies choisies pour permettre au système *MyFoodora* de fonctionner suivant différentes politiques.

En ce qui concerne les cartes de fidélité, nous avons choisi de procéder sur le modèle des **Strategy pattern** : une interface (FidelityCard) déclinée sur les trois possibilités *(Basic, Point, Lottery)*, et intimement liée aux classes Customer (qui détient la carte) et Order (chaque commande fait gagner des points, ou le prix de la commande selon une certaine probabilité).

De la même manière, les politiques de livraison sont regroupées sous une interface *DeliveryPolicy* déclinée en FastestDelivery ou FairOccupation. Suivant la politique choisie, la méthode allocateCourier(Order o) diffère alors légèrement dans le code.

Pour les politiques de tri, **Sorting policies,**

<<Sorting policies- explanation>>

Enfin, les politiques d'atteinte d'un bénéfice **(target profit policies)** sont simplement des méthodes accessibles aux managers dans le système interne *MyFoodora*, car leurs attributs sont des attributs du système et donc facilement accessibles depuis *MyFoodora* package.

Une fois ce design là proposé, les diagrammes UML provisoires tracés en fonction des Design et Strategy Pattern choisis, les classes et interfaces peuvent être construites progressivement avec leurs attributs et méthodes, tout en remaniant le diagramme UML général du système si nécessaire.

<<UML Diagramme final>

4. Implémentation

L'implémentation du code à partir du diagramme UML construit nous a alors paru plus facile, puisque les points critiques avait déjà été abordés (Meal Of The Week par exemple) et nous nous étions concertés sur les différents Design.
A partir des modèles vus en cours *(Factory Patter, Observer Pattern, Strategy Pattern)*, nous avons créé toutes les classes de manière incrémentale (attributs/méthodes principales) tout en veillant à bien commenter le code pour pouvoir générer une Java-doc facilement.
Nous nous sommes équitablement répartis le travail (l'un responsable des utilisateurs, l'autre de la carte du restaurant, puis enfin le système interne qui nécessitait l'intrication des deux).

Une fois le principal créé, nous avons pu rajouter l'implémentation des classes et méthodes plus difficiles mais nécessaires (par exemple la classe Order, la méthode allocateCourier(Order o), les connexions/déconnexions au système).
Des remaniements ont parfois été nécessaires, comme l'introduction supplémentaire dans le cœur du système *MyFoodora* d'un message de notification des clients de la part du restaurant, si jamais ce dernier venait à modifier son menu spécial (exigence du système) en étant connecté.

De nouvelles relations sont apparues entre les classes (DeliveryRequest entre Order et Courier, initialement non confirmée), et nous avons peu à peu complexifié le code en introduisant des méthodes plus complexes, pas forcément indispensables au bon fonctionnement du système mais néanmoins exigées par le cahier des charges : les target profit policies, delivery policies et le système de cartes de fidélité.

Nous avons implémentées ces classes et méthodes en dernier, car elles n'étaient pas obligatoires pour le bon fonctionnement de *MyFoodora*, et permettaient donc de travailler et tester notre code tant qu'il était encore allégé.

Pendant l'implémentation du code, à force de multiplier les boucles à condition (if...else...), nous nous sommes rendus compte de la nécessité de créer des exceptions pour mieux tenir compte des exceptions et erreurs de notre code JAVA, et surtout éviter d'alourdir le code avec des boucles et conditions démultipliées.

5. Gestion des exceptions

Bien que nous n'ayons pas directement pensé aux exceptions que nous devrions créer pour notre code, nous avons rapidement été confrontés à ce problème lors de notre implémentation.

Nous avons repéré certaines méthodes qui pourraient générer des exceptions *(par exemple un pourcentage supérieur à 1, un nom d'utilisateur déjà existant, un coursier non disponible, une date non valide, un utilisateur absent du système, un menu non existant, etc)*, et nous avons créé un package regroupant les différentes exceptions héritées de **Exception**.
Chaque bloc de code pouvant générer une exception a ensuite été isolé et l'exception capturée à travers le bloc *try{...} catch{...}*, puis traitée afin d'afficher un message d'erreur.

La liste des exceptions gérées du package *MyFoodoraException*, non exhaustive, est présentée ci-dessous :

<<Exceptions du système- liste, exemples :>>

-NoCourierAvailableException
-EmptyOrderException
-BadMenuException
-WrongItemNameException
-WrongMenuNameException
-WrongRestaurantNameException
-NoItemOrdered
-NoMealOrdered
-AccessRestrictedException
-AlreadyDeliveredException
-WrongPercentageException
-UnregisteredUserException
-AlreadyExistingUsernameException

6. Tests

Cette partie est réservée aux différents tests sur notre programme. Après avoir implémenté nos classes, nous avons commencé à tester chacune d'elle et générant des **Junit Test** sur les méthodes, getters et setters de nos classes.

Bien que cette étape soit indispensable à tout bon programmeur, elle nous a paru très coûteuse en temps, car elle nécessite d'écrire un test pour chaque méthode de chaque classe créée, ce qui finit vite par être considérable.

Aussi, nous sommes conscients des limites de nos tests, en ce sens qu'ils ne couvrent pas l'intégralité de notre code, et omettent donc des parties entières de nos classes.

Comme suggéré, chaque création de méthode ou classe doit être directement suivie de l'implémentation d'un test pour la vérifier ; toutefois, soucieux de fournir rapidement en squelette implémenté de notre système, nous avons d'abord codé un premier jet rapide de toutes nos classes, et c'est seulement après que nous avons commencé les tests.

Les tests réalisés se trouvent dans le package *MyFoodoraTests*, et leurs résultats sont plutôt concluants.

7. Résultats

Nous avons répondu au scénario type du cahier des charges par l'implémentation de notre système :

<<Screenshot Use case scenario>>

Nous vous proposons de juger nos résultats à travers :
- un **fichier initial** *my_foodora.ini* qui permet l'initialisation des utilisateurs et de quelques menus du système.
- un **fichier test-scenario** *testScenario1.txt* réservé à la partie 2 pour les commandes CLUI.
Pour le lancer, il suffit d'utiliser la commande *runtest* du CLUI et de rentrer le nom du fichier testScenario.

8. Conclusion et pistes d'amélioration

A travers ce projet, nous avons fourni une structure de type « plateforme » capable de relier entre eux des restaurants, clients, coursiers et managers de manière quasi-instantanée pour répondre à un objectif de livraison de repas à domicile.

Nous avons décrit ici l'analyse et la conception de notre système puis son implémentation en JAVA, et enfin les tests et résultats obtenus qui ont permis de corriger notre implémentation et d'en assurer l'efficacité et robustesse.

Bien que conscients des limites de notre système et des pistes d'amélioration qui peuvent y être apportées, notre implémentation actuelle permet de répondre à la majeure partie des exigeances du système, et fournit une plateforme simple d'utilisation et flexible.

Toutefois, nous pourrions suggérer d'y apporter quelques améliorations en garantissant par exemple l'accès à plusieurs utilisateurs en même temps.
Cela impliquerait un programme multi-threading, et donc capable d'assurer la sécurité et de gérer la concurrence entre les Threads, notamment par l'utilisation de Locks ou de méthodes synchronisées.

Par ailleurs, nous pourrions étendre notre système de manière à fournir une véritable plateforme de paiement par carte bancaire dès qu'une commande est passée par le client.

Nous pourrions également facilement élargir notre système à travers de nouveaux produits (notamment grâce aux Factory method Pattern), comme par exemple dans certains restaurants, qui vendent des produits de qualité utilisés pour la conception de leurs plats (huile d'olive, farine intégrale, fromages frais, jambons locaux), ou encore des boissons pour accompagner le repas.

L'envergure de ce projet nous a permis d'approfondir nos connaissances dans des domaines de code variés et nécessaires à la création d'un code JAVA flexible et efficace (design pattern, exceptions, tests, serialization) pour permettre de répondre au cahier des charges exigeant et complet du système.

La mise en forme de la plateforme a sollicité un important travail de réflexion et la confrontation de nos points de vue de manière récurrente, pour mettre en place le système le plus adapté possible.
Ainsi, le travail d'équipe qui a résulté de ce projet nous a permis d'avancer de manière coordonnée et efficiente, mais surtout d'acquérir plus de maturité tant au niveau technique (implémentation en Java) qu'au niveau créatif (analyse et conception d'un système).

ANNEXES