Introduction

L'objectif de ce projet est de créer un système permettant l'interaction entre des restaurants, des clients et des coursiers : les clients passent des commandes aux restaurants qui produisent des plats livrés par un coursier. Au delà du caractère fonctionnel de notre programme, nous avons aussi tenté d'utiliser le plus possible les design patterns pour rendre noter code le plus extensible possible.

Table des matières

1 Analyse des contraintes et design du programme

1.1 Le système central (Main.MyFoodora) : singleton pattern

Cette classe est censée stocker l'historique entier du système. Elle a donc un rôle central, et nous avons choisi de l'utiliser comme un pivot permettant d'accéder aux instances des objets dont on a besoin.

Pour ce faire, la classe MyFoodora doit donner facilement accès aux listes telles que la liste des coursiers de garde depuis l'instance d'un Restaurant ou bien aux nombres tels que le profit total depuis une instance de Manager. Ces attributs doivent être accessibles de quasiment n'importe quelle classe, et ne doivent pas dépendre de l'instance de la classe MyFoodora.

Nous avons trouvé deux solutions :

— soit définir tous les attributs nécessaires (historique des commandes, utilisateurs enregistrés, etc.) de MyFoodora comme statiques ;

— soit utiliser le singleton pattern de manière à ce qu'une seule instance de la classe MyFoodora n'existe.

Nous avons choisi la deuxième option car elle était probablement plu simple à implémenter et surtout plus élégante. Ainsi, une grande partie des classes de notre programme ont *l'unique* instance de MyFoodora comme attribut privé. Il faut absolument déclarer l'instance de MyFoodora comme privée, sinon n'importe quel utilisateur peut accéder au coeur du système.

1.2 Traitement du temps : une classe Main.Date avec des attributs statiques et non statiques

La notion du temps est très présente dans le cahier des charges du système. Par exemple, le système doit stocker l'historique des commandes et être capable de calculer le revenu sur une période donnée. Les clients doivent être notifiés quand il y a de nouveaux plats de la semaine ou bien quand c'est leur anniversaire.

Nous avons considéré plusieurs possibilités pour modéliser cette notion de temps dans le programme.

D'emblée nous voulions considérer le temps comme un thread, qui avance d'un jour toutes les minutes, en utilisant par exemple un compteur qui s'incrémente toutes les minutes grâce à la méthode Thread.sleep(). Cependant, cette approche ne garantit pas que le temps avance régulièrement, cela dépend de la fréquence à laquelle le thread dédié est dans l'état running. Une autre solution pourrait être alors de faire que le thread vérifie régulièrement l'heure donnée par l'ordinateur, et qu'il la convertisse en une date.

Ce genre d'approche pourrait toutefois être légèrement ennuyeuse pour le correcteur, car il ne pourrait donc pas gérer le temps comme bon lui semble. C'est pour cela que nous implémenté la classe Date. Cette classe Date est en fait hybride. Elle présente deux types d'attribut :

— des attributs statiques qui correspondent à la date courante du système. Elle peut bien entendu être modifiée à loisirs (on peut avancer d'un jour ou plusieurs jours à tous moments grâce aux méthodes Date.goTomorrow() et Date.advanceTime(int nbDays)).

— des attributs non statiques qui permettent de dater des événements dans le passé (pour la création d'un historique par exemple).

Notre classe Date permet également de convertir le compteur de jours depuis l'instant zéro (arbitrairement placé au samedi 1er avril 2017) en une date classique avec le nom du jour et le nom du mois. Pour des raisons d'irrégularité du calendrier grégorien, l'utilisateur sera limité à naviguer entre le 1er mars 2017 et le 31 décembre 2017. Par défaut, la date initiale est le lundi 10 avril 2017.

1.3 Les utilisateurs (package Users)

Comme tous les utilisateurs ont besoin de s'authentifier avec un username et un mot de passe, nous les avons défini comme attributs communs de tous les utilisateurs. Nous avons créé une classe abstraite User qui possède trois attributs protégés : un ID, un nom d'utilisateur et un mot de passe. Cette classe est abstraite car elle n'a pas besoin d'être instanciée.

De façon à ce que le programme reconnaisse un utilisateur grâce à son username, nous devons nous assurer que chaque username est caractéristique de son propriétaire. C'est pourquoi les utilisateurs sont stockés dans une HashMap où le username est la clef.

1.4 Les politiques (package Policies)

Le cahier des charges demandait de mettre en place trois types de politiques différentes : DeliveryPolicy, ShippedOrderSortingPolicy et ProfitPolicy. De manière générale, nous avons implémenté ces politiques en respectant le principe open-close, c'est-à-dire pour qu'un développeur ultérieur puisse aisément ajouter une nouvelle politique de profit sans avoir besoin de modifier le code déjà fait. Nous avons donc utilisé le strategy pattern, qui permet d'externaliser la mise en oeuvre de l'action déterminée par la politique choisie.

Chaque catégorie de Policy est donc une interface qui est implémentée par des politiques concrètes, qui pourront être complétées par d'autres par un autre développeur si besoin.

1.4.1 DeliveryPolicy

Tout d'abord, nous mentionnerons que la DeliveryPolicy choisie doit être connue de toutes les instances de coursiers. Nous avons donc mis la DeliveryPolicy choisie comme attribut de l'instance unique de MyFoodora. Il faudra toutefois prendre garde à ne pas déclarer de valeur de la DeliveryPolicy à l'initialisation de MyFoodora, car on a alors une situation de deadlock puisque DeliveryPolicy a bien entendu MyFoodora comme attribut également. C'est pour cela qu'il faut bien définir la DeliveryPolicy avant de passer toute commande, vu que la DeliveryPolicy n'a pas de valeur par défaut.

Pour le reste, les deux politiques demandées reposent sur la maximisation ou la minimisation d'un critère dans la liste des coursiers en service. La possibilité laissée aux coursiers de refuser une commande est implémentée de façon un peu artificielle, car on ne peut pas attendre que le coursier se connecte au système pour réaliser toute commande. Nous avons donc mis une probabilité faible que le coursier refuse d'effectuer la course.

Il serait plus judicieux de créer une liste triée selon le critère (distance ou occupation) que juste chercher l'extremum.

1.4.2 ProfitPolicy

Le statut de cette interface est un peu différent. En effet, le choix d'une ProfitPolicy a un effet immédiat sur les ProfitFigures (pourcentage de marge, coût de livraison, frais de service). Il n'y a donc pas vraiment besoin de la mettre en attribut du système. Le calcul du ProfitFigure manquant se fait simplement par la résolution d'une équation du premier degré.

1.5 Les produits (package Food)

Nous avons décidé de créer une classe abstraite de laquelle héritent les deux catégories de produits existants (Dish et Meal). Elle permet de regrouper les attributs communs tels que le prix et le nom. Elle comprend aussi les méthodes de l'interface Visitable, c'est-à-dire accept(Visitor).

On ainsi la présence du visitor pattern qui permet de faire le calcul du prix dans la classe Restaurant en utilisant les facteurs de réduction du restaurant au moment de la commande.

1.5.1 Dish

Cette classe est assez simple. Le seul attribut supplémentaire est le type du plat, qu'on a codé sous la forme d'un enum. Les classes Starter, MainDish et Dessert héritent de la classe Dish.

1.5.2 Meal

La classe Meal est la mère de HalfMeal et FullMeal. Les attributs supplémentaires sont la liste des plats du repas ainsi que le type du repas. Pour le type on a considéré qu'un repas ne pouvait être d'un type spécifique donné uniquement si tous les plats sont de ce type, sans quoi le repas est standard. Ceci est codé dans la méthode findMealType. En ce qui concerne le prix, c'est tout simplement la somme des prix des items.

En ce qui concerne HalfMeal et FullMeal, il faut respecter la cohérence d'un repas (pas de menu avec deux desserts par exemple). Nous avons donc créé une exception InvalidMeal pour le cas où on construit un Meal avec une ArrayList contenant un repas invalide. Cependant, la création d'un constructeur sans ArrayList a rendu cette exception obsolète, que nous avons quand même laissé dans le code.

1.6 Le processus de commande

Nous touchons ici le processus fondamental du programme : le passage d'une commande par un client. Nous avons créé une classe dédiée nommée Food.Order. Ses attributs sont tous les éléments nécessaires à la caractérisation d'une commande : un ID, la liste des items commandés, le client, le restaurant, la date de la commande, le prix ainsi que le livreur. Le prix est calculé grâce au visitor pattern pour les DiscountFactors (avec la méthode Customer.accept(Restaurant)) et grâce au strategy pattern pour la remise liée aux cartes de fidélité. Tout cela est regroupé dans la méthode Order.computeOrderPrice.

Lorsqu'un client passe commande, une instance d'Order est donc automatiquement créée. La commande n'est cependant pas immédiatement ajoutée dans l'historique car il y a risque (faible) de ne pas trouver de coursier. Ce n'est que lorsqu'on a trouvé un coursier qu'on peut considérer la commande comme validée et l'ajouter à l'historique des commandes.

1.7 Fidélisation des clients

Les clients sont fidélisés par les deux dispositifs suivants.

1.7.1 Les notifications

Les notifications fonctionnent grâce au pattern observer. Les observables sont les restaurants, qui notifient les observeurs (les clients qui ont accepté de recevoir des notifications) chaque fois qu'on met un nouveau plat de la semaine ou bien qu'on change les facteurs de remise.

Il y a aussi la notification d'anniversaire, qui a été permise en notifiant les restaurants à chaque changement de date, qui notifient alors les clients dont c'est l'anniversaire.

1.7.2 Les cartes de fidélité (package Fidelity

La carte de fidélité pouvant avoir plusieurs types : (lottery, basic ou point). Nous en avons fait une classe abstraite, que les classes LotteryFidelityCard, BasicFidelityCard, et PointFidelityCard étendent.

Une carte de fidélité n'est valable que dans un restaurant, le client doit donc avoir une carte par restaurant. Pour se faire, nous avons rajouter l'attribut FidelityCardList à la classe Customer, la structure de donnée que nous avons choisi

est une HashMap, cela nous permet de trier les cartes de fidélités par restaurant, un client ne pouvant avoir qu'une carte par restaurant. Les keys de cette HashMap sont donc les restaurants et les Values sont les cartes de fidelity.

Par défaut tout client possède une carte de fidélité basique, il a donc fallu modifier le constructeur du client pour lui créer des cartes de fidélités basique pour tous les restaurants du système et modifier le constructeur du restaurant pour ajouter des cartes de fidélité basique pour celui-ci chez tous les clients du système. Ce choix de dire que tout client possède une carte de fidélité nous permet de ne pas vérifier si le client possède une carte de fidélité auprès du restaurant à chaque fois qu'il fait une commande.

Ensuite ces cartes de fidélité sont utilisées via la méthode useCard(). Pour que cette méthode soit utilisée à chaque commande, elle est placée directement dans le constructeur de la classe Order sous la forme d'un coefficient de réduction qu'elle applique au prix de la commande. La méthode renvoie donc un double. Cela permet en même temps de traiter la carte de lotterie : si le client obtient une commande gratuite, useCard renvoie 0 ce qui correspond à une réduction de 100

1.8 Diagramme UML

Voici le diagramme UML quasiment exhaustif de notre programme :

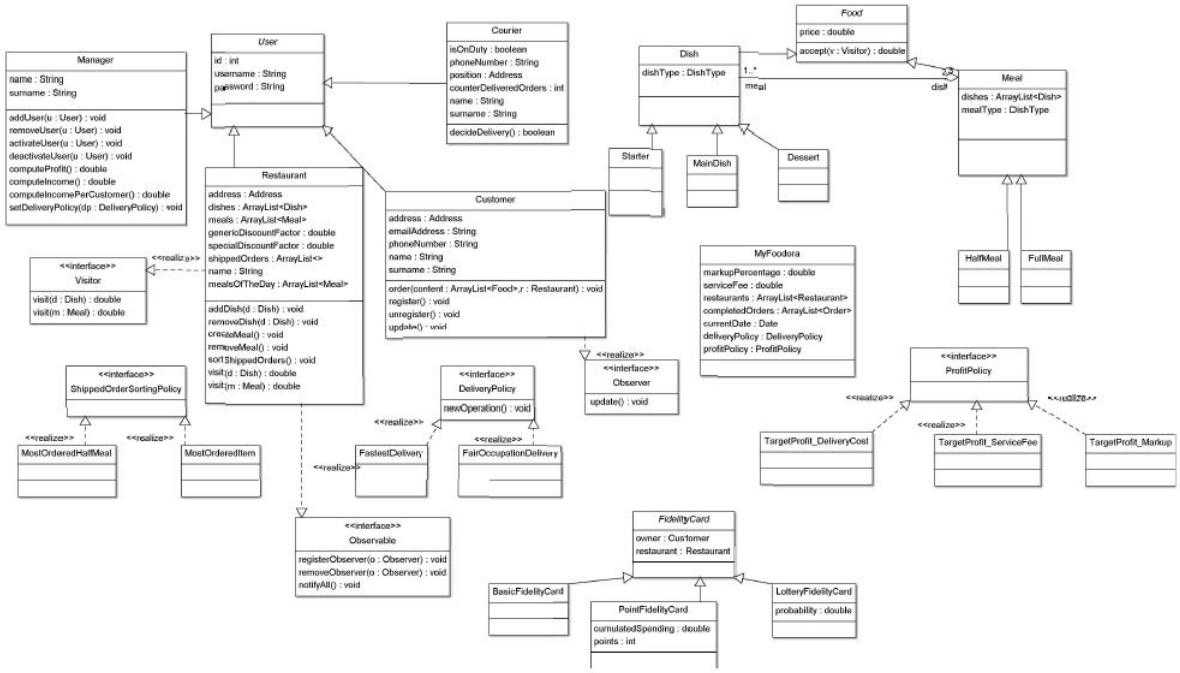

FIGURE 1 – Diagramme UML

On y voit apparaître les différents design patterns expliqués dans toute cette section : singleton pattern pour MyFoodora, observer pour les notifications, strategy pour les policies, visitor pour le calcul des prix.

2 Implémentation et tests

2.1 Organisation de notre travail

Nous avons respecté la consigne « celui qui code ne teste pas ». Cela a permis aux deux membres du projets de s'intéresser à chaque classe pour pouvoir maîtriser l'ensemble du projet. Cela a aussi permis d'apporter un regard neuf sur chaque partie du code et ainsi de corriger certaines erreurs.

Package	Implémentation	Test
Fidelity	Quentin	Nathan
Main	Nathan 50% Quentin 50%	Quentin 50% Nathan 50%
Policies	Nathan	Quentin
Users	Nathan	Quentin

On remarque que Nathan a plus implémenté, et que Quentin a plus testé. C'est notamment dû au fait que Nathan avait plus de temps à consacrer au début et Quentin à la fin. Ce rapport sera inversé dans la seconde partie du projet.

Outils utilisés : GitHub et Trello

Pour pouvoir coder en parallèle, nous avons dû utiliser deux outils principaux. Tout d'abord Github, sur lequel on avait créé le projet avec 3 branches : une branche « master », une branche « modifNathan » et une branche « modifQuentin ». Cela nous permet d'avancer sur nos tâches chacun de notre côté sur les branches qui nous sont réservées. Puis à intervalle de temps régulier, typiquement une journée, nous synchronisons nos branches avec la branche master. Github est pour cela très pratique car il permet de visualiser rapidement les modifications entre les versions. De plus, grâce aux extensions pour Github dans Eclipse nous pouvions coder directement sur notre IDE coutumière Eclipse. Nous utilisons aussi Trello qui est un système de « To Do List » un peu plus élaboré. Il nous permet de segmenter le travail en tâches que nous nous répartissons et sur lesquelles nous pouvons voir l'état d'avancement. Nous pouvons aussi y rajouter des commentaires pour signaler lorsque à autre lorsque nous faisons face à une difficulté.

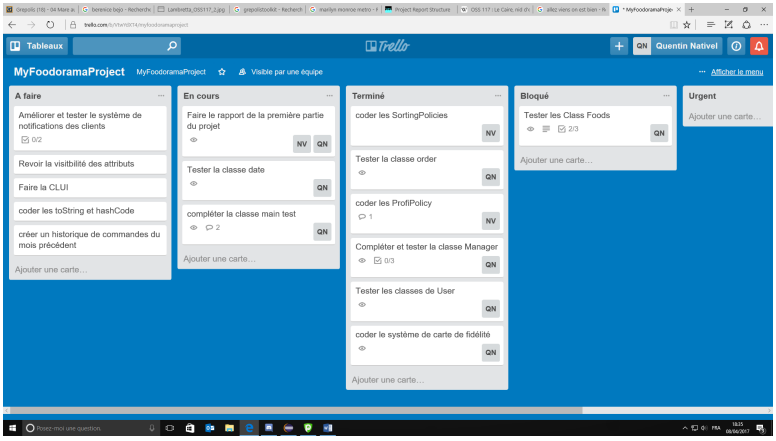

FIGURE 2 – Tableau Trello

Grâce à ces outils, la communication était bonne dans l'équipe et le fait de devoir travailler à deux ne nous a pas posé de difficulté particulière.

2.2 Implémentation et problèmes rencontrés

Nous avons divisé les classes en plusieurs packages logiques : Food, Users, Policies, Fidelity et Main.

2.2.1 Remplacement des ArrayList par des HashMap

Initialement, nous avons massivement utilisé la structure ArrayList pour stocker les listes d'utilisateurs ou bien les listes de plats dans un restaurant. En testant le programme, nous nous sommes rapidement rendu compte qu'il y aurait besoin de pouvoir accéder simplement à une instance grâce à un identifiant (le username pour les Users et le nom pour les plats). Nous avons donc remplacé progressivement toutes les ArrayList par des HashMap. La difficulté que nous avons rencontrée est pour les plats, qui peuvent avoir un nom de plusieurs mots. L'utilisation seule de la commande String.split n'est pas suffisante, et on peut apercevoir comment on a réglé ce problème dans le code de MainTest.

2.2.2 Deadlock à l'initialisation de l'instance de MyFoodora

De nombreuses classes ont l'instance de myFoodora comme attribut par défaut. Il est donc essentiel de ne pas en initialiser quand on crée l'instance de MyFoodora, sans quoi ni l'instance de MyFoodora ni les objets susmentionnés ne pourront être créés. C'est pour cela par exemple qu'il n'y a pas de valeur par défaut de la DeliveryPolicy.

2.2.3 Les méthodes de tri et de maximisation

Nous avons utilisé des approches de tri et de maximisation plus ou moins évoluées selon les cas. Dans les cas de maximisation nous avons en général codé à la main une fonction maximum par souci d'efficacité (FastestDelivery par exemple), dans les cas de tri nous avons utilisé la méthode Collections.sort(ArrayList, Comparator). Il y a également le problème de deux instances différentes d'un item mais qui ont les mêmes attributs. Nous avons décidé d'implémenter la méthode equals dans la classe Food de manière à ce que deux items soient égaux tout simplement s'ils ont le même nom.

2.2.4 MainTest : une première ébauche d'interface client

Nous avons réalisé une classe MainTest, et c'est sans doute celle-là qui intéressera le plus le correcteur. On remarquera la présence de la classe ConfigInitiale, dont la méthode statique launch() pourra être lancée au début de tout test pour établir un historique (placé au mois de mars pour pouvoir utiliser les fonctions de ProfitPolicy). Conformément au cahier des charges, on peut effectuer tous les use cases des pages 7 à 9. C'est notamment lors de l'implémentation de cette classe MainTest qu'on a réalisé l'importance de pouvoir désigner des instances par un identifiant, par exemple quand le client choisit un plat dans le menu d'un restaurant.

2.3 Tests JUnit

Le test du programme a été effectué en deux parties. Chaque classe a d'abord été testée avec JUnit Test Case puis nous avons réalisé des tests globaux à l'aide des cas d'utilisation fournis dans l'énoncé, ceux-ci sont présents dans la classe « MainTest ». Globalement, il y a eu très peu de corrections à effectuer le code fonctionnait souvent du premier coup. En effet, nous avons en partie adopté l'approche Test Driven Method, c'est-à-dire que nous testions les fonctionnalités du code tout en l'écrivant, ce qui a donné un code avec très peu d'erreur.

Pour les tests JUnit, nous ne testions que les méthodes les plus importantes de la classe. Nous ne testions pas les getters et les setters, ni les constructeurs. De même que, grâce aux différents patterns utilisés, la plupart des méthodes étaient simples, elles n'étaient souvent composées que d'un appel à une autre méthode d'une autre classe. Nous testions alors la méthode originelle et nous ignorions les doublons. Pour réaliser ces différents tests, nous avons créé une classe « configInitiale » qui construisait le système « My Foodora » à l'aide de la méthode launch. Dans cette configuration initiale, l y avait un coursier, un client, un restaurant, un manager ainsi que quelques plats. Le but étant de modéliser un système très simple qu'on puisse appeler dès que nous voulions effectuer un test Les tests ne se terminent pas toujours par des assertions mais aussi souvent par des prints sur la console. Et nous regardions manuellement si le résultat affiché était juste. Cette façon de procéder était souvent plus rapide et flexible qu'une assertion. Nous avons aussi ajouté des prints à divers endroits du code pour suivre plus précisément son exécution et vérifier par quelles étapes passait le programme.

Une des erreurs faites a été la suivante : la formule pour calculer la politique de profit « Mark_up » était fausse, une partie de la formule calculait le profit global tandis que l'autre calculait le profit par client. Il a donc fallu créer une méthode computeIncomeBis() qui retourne un tableau constitué du revenu et du nombre de clients qui ont commandé quelque chose durant la période demandée (un mois pour ce cas).

Globalement, nous avons compris l'intérêt de bien anticiper dans le processus de codage, car tout changement peut se révéler fastidieux : changement du nom d'une méthode, de son type de retour. Seul le changement de paramètre n'est pas trop gênant grâce à la possibilité de surcharger les méthodes en Java.

3 Comment tester notre programme ?

Il suffit de lancer le main de la classe MainTest pour tester le programme. Chaque fois que le correcteur se demandera quelle commande rentrer, il pourra rentrer "help". On a précisé uniquement l'intitulé des commandes, les arguments sont les mêmes que ceux de l'énoncé du projet.

Quelques rappels

- le programme commence automatiquement à la date du 10 avril 2017. Pour changer la date, il faut être manager ;
- le programme demande si on déjà un compte. On pourra dire oui (taper 1) ;
- le programme charge automatiquement une configuration initiale (issue de ConfigInitiale.launch()). Le correcteur pourra en voir les détails en utilisant la commande "show all" quand il est connecté en tant que manager.

3.1 Test du manager

On pourra se connecter avec le username "ceo" et le mot de passe "123456789". On pourra tester les fonctions suivantes :

- afficher ce qui est stocké par le système (`show orders`, `show customers`, `show couriers`, `show restaurants`, `show all`)
- enregistrer un utilisateur : par exemple, pour enregistrer un nouveau client on pourra taper `registerCustomer Cyprien Dupres cdupres 3,4 1234`
- faire avancer le temps d'un jour (`goTomorrow`)

3.2 Test d'un restaurant

On pourra se connecter avec le username "asiat" et le mot de passe "1234" (traiteur asiatique). On pourra tester les fonctions suivantes :

— ajouter un plat (exemple : `addDishRestaurantMenu smokedTuna starter Standard 7.5`)
— ajouter/retirer un plat de la semaine (exemple : `setSpecialOffer Full meal`)
— la fonction créer un menu n'est pas encore codée pour la partie interface
—

3.3 Test d'un client

On pourra se connecter avec le username "ddescamps" et le mot de passe "1234" (Diane Descamps). On pourra tester les fonctions suivantes :

— passer une commande : `createOrder asiat` puis `addItem2Order Nem` puis `addItem2Order Beef with onions` puis `endOrder`.

3.4 Test d'un coursier

On pourra se connecter avec le username "jdupont" et le mot de passe "1234" (Jean Duont). On pourra tester les fonctions suivantes :

— se mettre en service : `onDuty`.

4 Conclusions et limites

Nous avons réalisé un programme qui, nous l'espérons, satisfait toutes les contraintes de la partie 1 du cahier des charges. Nous avons tenté de respecter le plus possible le principe open-close. Ainsi, il sera aisé pour un développeur ultérieur d'ajouter une politique de choix des chiffres du profit ou de choix du livreur. Il sera aussi simple d'ajouter un nouveau type de carte de fidélité. Il sera assez simple d'ajouter un nouveau type d'Item (comme sous-classe de Food), même s'il faudra modifier l'interface Visitor pour ajouter un visit(newItemType) et ajouter un attribut liste de ce nouvel item dans la classe Restaurant. En revanche, il sera compliqué d'ajouter un nouveau type de User. L'ajout d'une interface graphique ergonomique sera très bénéfique et sera fournie en partie 2 de ce projet.