

Rapport de projet

Software Engineering

MyFoodora

COLAS Gaël
PRABAKARAN Sylvestre

Introduction	3
Sujet du projet	3
Notations	3
Objectifs et démarche	4
Objectif du projet	4
Grands axes de développement	4
Démarche	5
Architecture de l'application	6
Sérialisation des classes	6
Gestion des cartes proposées	6
Gestion des utilisateurs	8
Gestion des commandes et des livraisons	12
Gestion économique	14
Gestion globale	15
Exceptions à gérer	17
Tests effectués	18

Introduction

Sujet du projet

L'objectif du projet est de mettre en place une application de commande en ligne de nourriture auprès de restaurants. Cette application fait usage d'une plateforme MyFoodora qui est utilisée par différents utilisateurs et de différentes formes. Cette plateforme doit respecter de nombreuses contraintes qui sont présentées dans le sujet du projet. Ce cahier des charges ne sera pas présenté dans le rapport.

Notations

Des notations en couleurs différentes ont été utilisées dans ce rapport afin de donner plus de clarté à la lecture. Voici leurs significations :

- **Classe** : nom d'une classe ou d'une interface
- **variable** : nom d'une variable ou d'un attribut
- **fonction** : nom d'une fonction ou d'une méthode
- **Design pattern** : nom d'un design pattern présenté dans le cours

Objectifs et démarche

Objectif du projet

Comme le montre le cahier des charges, la plateforme MyFoodora se doit de respecter des contraintes précises. Il a fallu donc mettre en place une architecture orientée objet précise et lui permettre malgré tout d'être modifiable ou améliorable sans effort important de réécriture du code source. C'est ce problème qui s'est présenté comme le plus important lors du développement de ce projet.

Grands axes de développement

Après lecture détaillée du cahier des charges et réflexion en binôme, il fallu séparer le projet en grandes parties qui sont plus ou moins indépendantes les unes des autres :

- Gestion des cartes proposées : La plateforme MyFoodora doit permettre de mettre à disposition des clients des variétés de plats qui peuvent se présenter sous forme d'entrée (Starter), de plat principal (MainDish) ou de dessert (Dessert). Chacun de ces plats possède des caractéristiques (Végétarien, sans gluten, etc...). Toutes ces informations nécessitent d'être implémentées sous forme de classes différentes mais respectant une certaine hiérarchie étant donnée les points communs.
- Gestion des utilisateurs : La plateforme permet à plusieurs utilisateurs de l'utiliser mais de manière différente. Chaque utilisateur a une identité et un objectif différent dans l'utilisation de la plateforme. Ainsi, chaque type d'utilisateur doit être implémenté par une classe différente car chaque utilisateur peut effectuer des tâches très différentes.
- Gestion des commandes et livraisons : Chaque client peut commander parmi les cartes proposées dans la plateforme et se faire livrer. Les livraisons reposent sur des politiques d'affectation. Il faut donc implémenter une solution qui utilise les positions pour affecter des livreurs à chacune des livraisons.
- Gestion économique : L'application doit se charger de calculer différentes grandeurs économiques telles que les profits ou les chiffres d'affaire. Il faut permettre aussi à l'application d'appliquer des stratégies pour optimiser le profit des mois suivants.
- Gestion globale : Le coeur de l'application est représenté par la classe **MyFoodora** comme demandé dans le cahier des charges. C'est la partie du projet qui demande le plus de travail dans la mesure où toutes les fonctions et classes créées précédemment devront être utilisées dans cette classe.
- Gestion de l'interface : Afin de pouvoir être utilisée, la plateforme doit posséder une interface qui se doit d'être facile d'utilisation et ergonomique (dans le cas où l'interface serait graphique notamment). Ce sera la grande partie du travail à faire une fois que la gestion globale avec la classe **MyFoodora** aura été implémentée.

Démarche

Ce projet a nécessité une certaine coordination afin de mener à bien et efficacement le développement de l'application. Nous nous sommes séparés les tâches et avons travaillé sur un fichier collaboratif grâce à Git.

Concernant l'avancement temporel du projet, la phase de réflexion s'est faite très tôt à travers beaucoup de diagrammes UML qui ont convergé vers le choix des solutions qui ont été implémentées ensuite. Toute la partie codage s'est faite de manière organisée et toujours en vérifiant régulièrement la cohérence des avancements entre nos deux travaux. Le travail en terme de temps s'est toujours équitablement réparti entre les deux étant donné que le temps de travail se faisait en commun.

Le planning de travail peut être décrit selon le tableau suivant :

Gaël COLAS	Sylvestre PRABAKARAN
Réflexion commune sur le choix de l'architecture	
Codage de toutes les classes concernant l'implémentation des FoodItem	Codage de toutes les classes concernant l'implémentation des User
Mise en place de toutes les Strategy Pattern avec les classes TargetProfitPolicy , DeliveryPolicy et SorterFoodItem et associées à leurs Factory Pattern .	Codage de la classe Order et mise en place des Observer Pattern pour les classes Customer et Courier avec la classe Board .
Début de mise en place de la classe MyFoodora	
Ajout et implémentation de toutes les méthodes correspondant aux actions faites par chaque User .	Appel des méthodes correspondant à chaque action des User dans la classe MyFoodora .
Implémentation des tests liés à toutes les classes héritant de User et correction de tous les bugs constatés.	Implémentation des tests liés aux classes héritant de FoodItem et aux Order et correction de tous les bugs constatés.
Rédaction du rapport	
Début de codage de l'interface en console avec MyFoodoraClient .	

Architecture de l'application

Les réflexions de départ ont donc permis de mettre en place une organisation de l'architecture des classes notamment grâce à un travail en diagrammes UML. Malgré tout, beaucoup de problèmes se sont posés pendant l'implémentation des solutions initialement envisagées et les diagrammes ont donc également évolué durant l'avancement du projet. Le travail a été séparé selon les secteurs présentés dans la partie précédente.

Sérialisation des classes

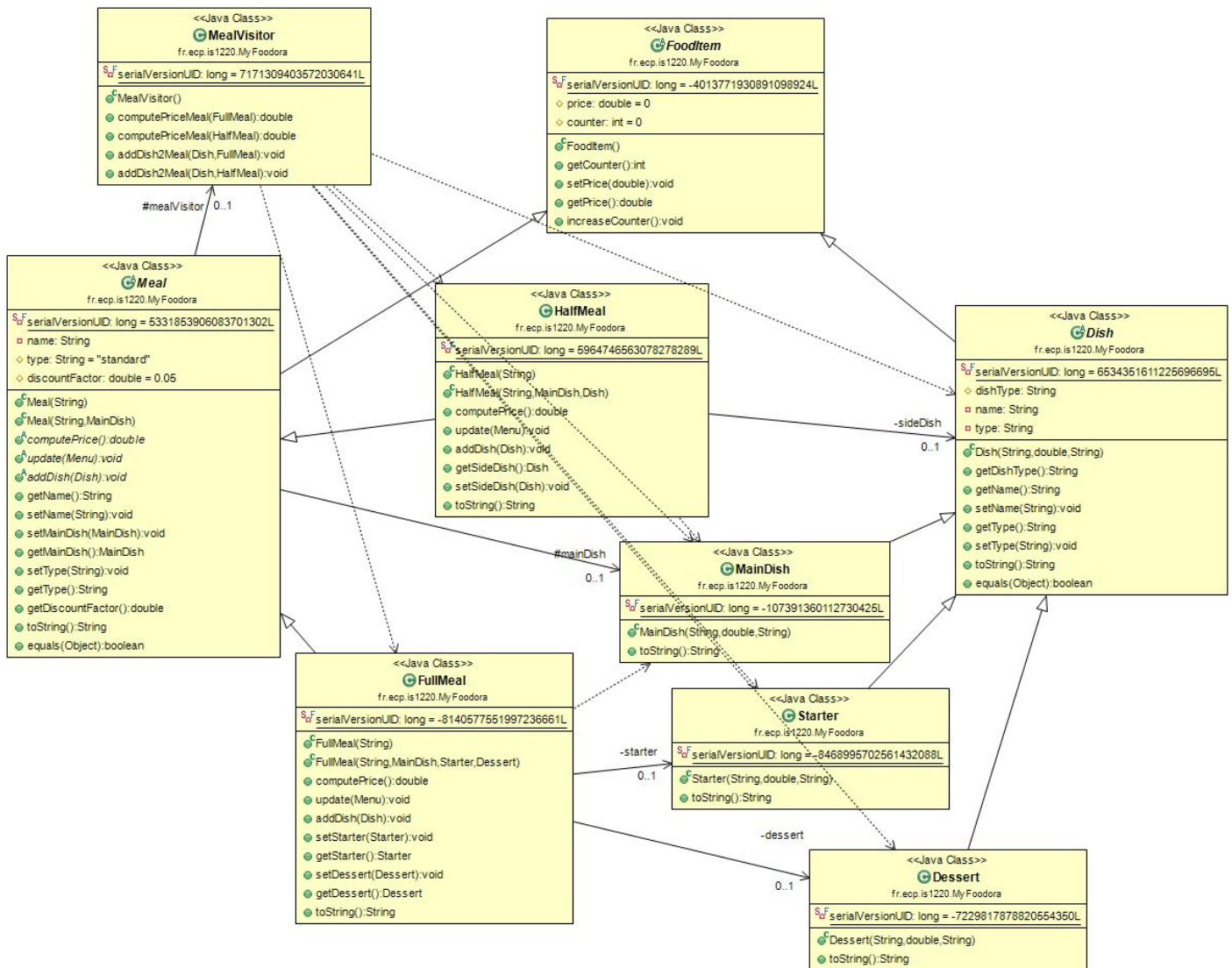
Avant toute chose, nous savons que la plateforme doit pouvoir être sauvegardée puis chargée en permanence lors de son utilisation sur le serveur. Une solution qui a été envisagée pour répondre à ce problème est de mettre en place la sérialisation des classes. Ainsi, toutes les classes doivent hériter de l'interface `java.io.Serializable` et posséder une clé de sérialisation unique du nom de `serialVersionUID` et de type long. Ainsi il faut implémenter cette solution notamment sur toutes les classes parentes du projet. C'est ce qui sera représenté par la suite dans tous les diagrammes UML avec une valeur générée aléatoirement par Eclipse.

Gestion des cartes proposées

En ce qui concerne la nourriture, les cartes des restaurants proposent deux types d'éléments. D'une part, elle propose des plats variés (classe `Dish`) et d'autre part, des menus (classe `Meal`) qui sont constitués de ces plats. Pourtant la classe `Dish` et la classe `Meal` ont un point commun : les deux classes possèdent un prix caractérisé par un attribut decimal `price` et un compteur qui comptera le nombre de fois où ces plats auront été commandés. Ce compteur est l'attribut entier `counter`. De plus, lorsqu'un client fera une commande, il pourra ajouter à sa commande aussi bien des menus que des plats, et le prix total de la commande sera calculé en gérant les deux types d'éléments sans les différencier. C'est pourquoi il a été décidé de faire une classe abstraite parente des classes `Dish` et `Meal` : La classe abstraite `FoodItem`.

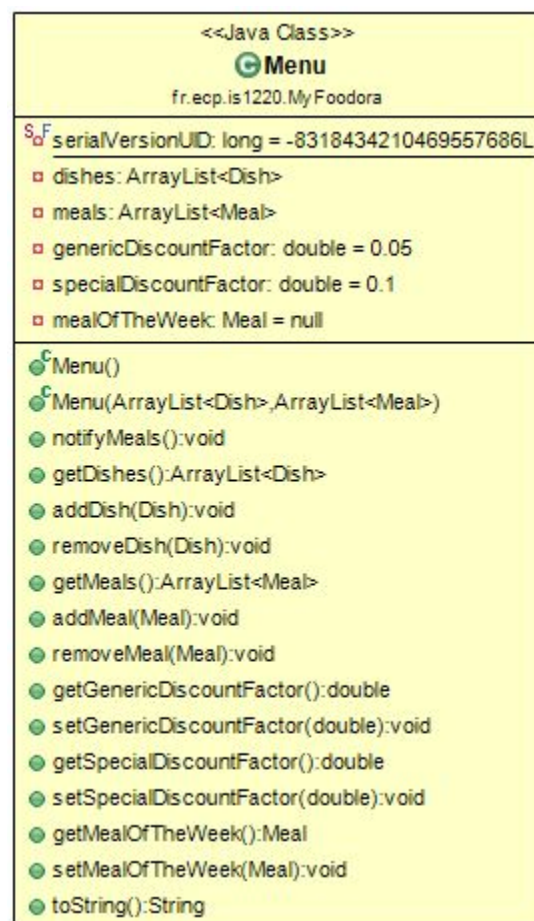
Concernant les entrées, plats et desserts, ces classes héritent de la classe `Dish` et sont respectivement les classes : `Starter`, `MainDish` et `Dessert`. Concernant les menus, ceux-ci peuvent se présenter sous forme de menus réduits ou de grands menus qui sont représentés par les classes `HalfMeal` et `FullMeal`.

On note aussi que les classes `HalfMeal` et `FullMeal` sont similaires dans leur utilisation et font appel aux mêmes méthodes `computePrice` pour calculer le prix du menu ou `addDish2Meal` mais qui ne fonctionnent pas de la même façon en fonction du cas où le menu serait réduit ou grand. On fait donc appel à un `Visitor Pattern` en créant une classe `MealVisitor` qui contient ces méthodes.



Enfin, chaque restaurant de la plateforme MyFoodora propose une carte. Chacune de ces cartes est représentée par un objet de la classe **Menu**. La classe **Menu** a pour attributs une liste de **Dish** et une liste de **Meal**. Mais elle est également caractérisée par des grandeurs afin de définir le prix des menus qui sont les attributs décimaux **genericDiscountFactor** et **specialDiscountFactor**. Cette carte dispose également, comme la plateforme MyFoodora l'impose, d'un "menu de la semaine" qui est un attribut de type **Meal** nommé **mealOfTheWeek**. Cependant, à chaque fois que le **genericDiscountFactor**, le **specialDiscountFactor** ou le **mealOfTheWeek** sera changé, il faudra mettre à jour tous les prix de tous les **Meal** de la carte. Ainsi, une structure en **Observer Pattern** a été adoptée ici.

Observer	Que doit-il observer ?	Observable	Implémentation
Meal	Le repas doit observer les changements des valeurs de coût dans la classe Menu .	La classe Menu : Lorsqu'elle met à jour le genericDiscountFactor ou le specialDiscountFactor ou le mealOfTheWeek .	Lorsque les valeurs en question sont modifiées, la méthode notifyMeals de la classe Menu est appelée et tous les Meal qui sont contenus dans la carte appellent la méthode computePrice .



Gestion des utilisateurs

La plateforme MyFoodora permet l'accès de plusieurs types d'utilisateurs. Comme présenté dans le cahier des charges, il y a les managers (**Manager**), les clients (**Customer**), les restaurants (**Restaurant**) et les livreurs (**Courier**).

Ces types de clients possèdent des points communs, ils sont tous définis par un index d'identification (noté **uniqueID**), un identifiant de connexion (noté **userName**), un mot de passe (noté **password**), un nom (noté **name**) et un nom de famille (noté **surname** sauf pour les restaurants). C'est pourquoi une classe parente des classes d'utilisateurs a

été faite : la classe abstraite `User`. On peut aussi noter que chaque utilisateur a donc un type qui doit être connu pour choisir les fonctions auxquelles il a accès, c'est pourquoi un attribut chaîne de caractère nommé `type` renseigne le type de l'utilisateur.

Chaque action que peut faire un utilisateur est implémentée sous forme de méthode dans la classe correspondant au type d'utilisateur (par exemple : il y a une méthode pour ajouter des utilisateurs dans la classe `Manager` car les managers peuvent ajouter des utilisateurs). Chacune des classes qui héritent de `User` a ses spécificités.

La classe `Manager` correspond aux utilisateurs qui ont tous les droits sur la plateforme MyFoodora. C'est pourquoi cette classe possède en attribut l'objet de type `MyFoodora` qui correspond à toutes les informations de la plateforme et au coeur de l'application. Toutes les fonctions de la classe `Manager` auront donc accès et pourront modifier les informations de la plateforme (par exemple ajouter des utilisateurs, modifier les valeurs économiques...).

Dans les classes `Restaurant`, `Courier` et `Customer`, il n'y a pas d'attribut de type `MyFoodora` afin de respecter une encapsulation car ces classes ne doivent pas avoir de libre accès à toutes les informations de la plateforme. Cependant, certaines des actions faites par ces utilisateurs doivent modifier les informations de la plateforme MyFoodora pour fonctionner, c'est pourquoi les méthodes correspondant à ces actions prennent un objet de type `MyFoodora` en paramètre. Ces trois classes possèdent également des adresses. C'est pourquoi une classe `Position` a été créée : elle est composée de deux entiers et caractérise une adresse.

La classe `Restaurant` possède un attribut de type `Menu` car chaque restaurant propose une carte. Un numéro de téléphone a aussi été ajouté parmi les attributs de la classe `Courier`.

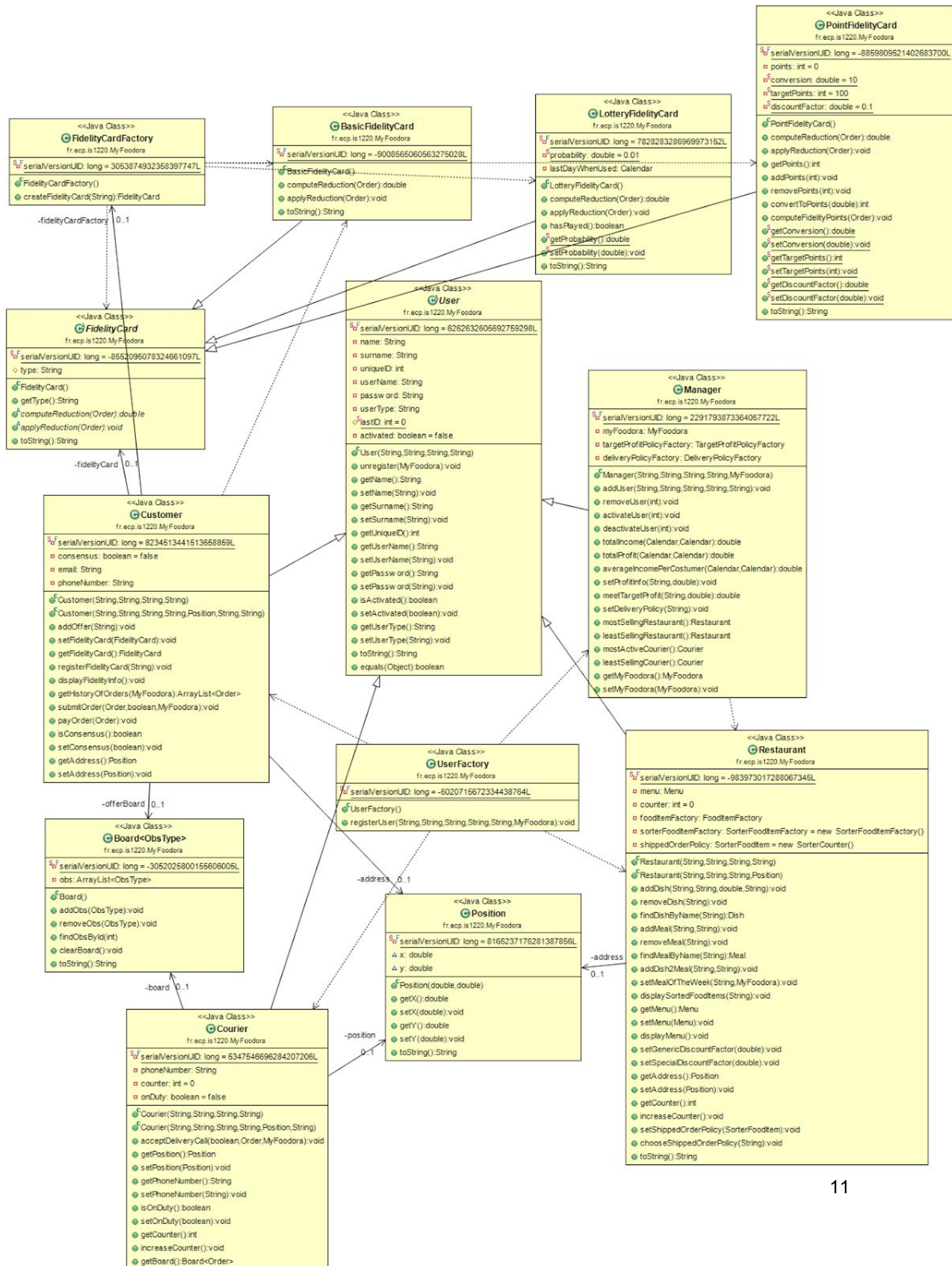
Les classes `Courier` et `Customer` doivent, en plus, observer des informations : c'est pourquoi il a été choisi ici d'adopter un `Observer Pattern` afin de répondre à ce besoin.

Voici plus précisément pourquoi il a fallu utiliser ce pattern dans chacun des cas :

Observer	Que doit-il observer ?	Observable	Implémentation	
Custom er	Le client doit recevoir les nouvelles offres de la plateforme et les afficher une fois que celui-ci se connecte.	Dans la classe MyFoodora : Des chaînes de caractères offer	Dans les deux cas, un système de tableau d'affichage (création d'une classe Board) est utilisé. Lorsqu'une Observable apparaît, celle-ci est ajoutée à tous les tableaux d'affichage de tous les Observers jusqu'à ce que ceux-ci les lisent.	Le tableau d'affichage est affiché à chacune des connexions du client. Une fois que celui-ci a lu les offres, celles-ci sont effacées du tableau d'affichage.
Courier	Le livreur doit recevoir les nouvelles demandes de livraison envoyées par MyFoodora pour ensuite les accepter ou non.	Dans la classe MyFoodora : Des commandes (classe Order)		Le tableau d'affichage est affiché dès qu'une nouvelle commande est demandée d'être livrée. Celle-ci disparaîtra lorsque le livreur a validé ou refusé cette demande.

Le cahier des charges demande également de mettre en place un système de cartes de fidélité pour les clients. Ceux-ci sont représentés par trois types : les cartes de fidélité basiques (classe **BasicFidelityCard**), les cartes de fidélité à points (classe **PointFidelityCard**) et les cartes de fidélité à loterie (classe **LotteryFidelityCard**). Ces trois classes sont utilisées différemment mais chaque client ne peut avoir qu'une seule carte de fidélité. C'est pourquoi la classe des positions avec notamment l'adresse du client et l'adresse du livreur. C'est pourquoi la classe **Customer** possède un attribut de type **FidelityCard** qui est une classe abstraite parente de tous les types de cartes de fidélité.

Enfin, afin de prévoir l'inscription des utilisateurs à la plateforme MyFoodora, il faut permettre à une seule méthode notée **registerUser** de renvoyer n'importe quel type d'utilisateur lorsqu'il faudra les stocker dans la plateforme. C'est pourquoi un **Factory Pattern** est utilisé ici : La classe **UserFactory**, à travers sa méthode **registerUser** ajoute à la classe **MyFoodora** les **Users** du type correspondant à la demande d'inscription.



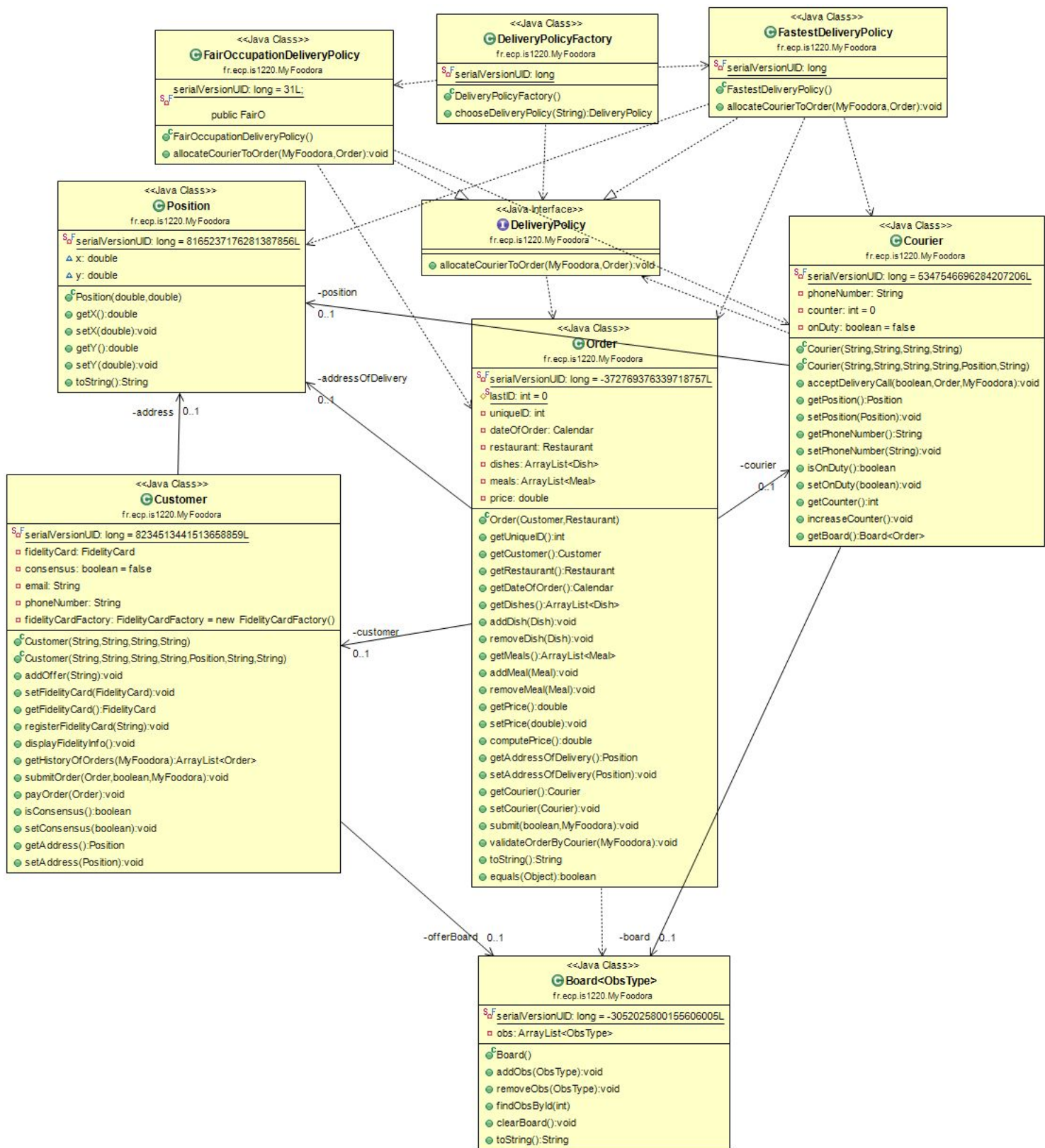
Gestion des commandes et des livraisons

La plateforme MyFoodora doit pouvoir gérer les commandes et les livraisons automatiquement en fonction de critères différents.

Chaque commande est faite par un client et est caractérisée par la classe `Order`. Cette classe comporte des attributs qui sont la date de la commande représentée par un objet de la classe `Calendar`, ainsi qu'une liste de `Dish`, une liste de `Meal` et un prix caractérisé par une variable `price` qui quantifie le prix total de la commande. Enfin, elle possède trois attributs correspondant aux trois utilisateurs qui concernent la commande : le client qui effectue la commande de classe `Customer`, le restaurant qui propose les plats choisis par le client de classe `Restaurant`, et le livreur qui s'occupera de livrer la commande `Courier`.

La classe `Order` contient aussi des méthodes importantes : des méthodes pour ajouter des plats à la commande tels que les méthodes `addDish` et `addMeal`, ainsi que des méthodes pour valider la commande : `submit` pour que le client valide sa commande (et ainsi augmenter tous les compteurs correspondant à chacun des plats) et `validateOrderByCourier` pour que le livreur valide l'affectation de la commande (et ainsi augmenter le compteur de livraisons du livreur correspondant).

Le cahier des charges demande de pouvoir affecter des livreurs à chacune des commandes mais en respectant des stratégies différentes : soit en choisissant le livreur qui est le plus proche du client soit en choisissant le livreur qui a fait le moins de livraisons pour que tous les livreurs aient fait le même nombre de livraisons. Pour répondre à ce besoin, un `Strategy Pattern` est utilisé grâce à la classe `DeliveryPolicy` qui consiste à choisir une politique d'affectation. Les classes qui héritent de `DeliveryPolicy` sont les classes `FairOccupationDeliveryPolicy` et `FastestDeliveryPolicy`. Cependant, pour permettre l'ajout possible d'autres politiques d'affectation tout en respectant le principe d'ouverture-fermeture, une structure en `Factory Pattern` est utilisée avec la classe `DeliveryPolicyFactory` et sa méthode `chooseDeliveryPolicy` qui permet de gérer la création des `DeliveryPolicy` en fonction d'une chaîne de caractère en paramètre qui renseigne quelle politique d'affectation l'utilisateur souhaite.

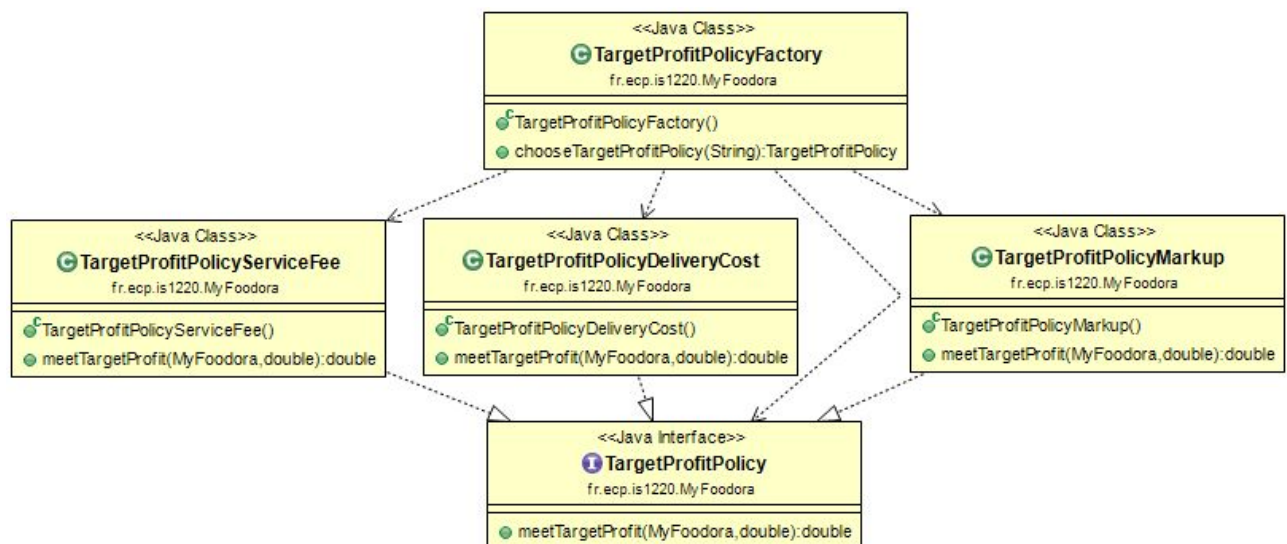


Gestion économique

Parmi toutes les classes présentées jusqu'à présent, certaines d'entre elles ont affaire avec des problématiques financières. On peut notamment penser aux prix à travers l'attribut `price` des classes `FoodItem` (et les classes qui en héritent) et `Order`. Il y a également tous les coefficients qui influencent ces prix tels que les attributs `genericDiscountFactor` et `specialDiscountFactor` de la classe `Menu`. Il y a également des attributs qui seront gérés dans la classe `MyFoodora` tels que le `serviceFee` (le prix payé par un restaurant à chacune des commandes pour pouvoir faire parti de la plateforme MyFoodora), le `markupPercentage` (le pourcentage de supplément à payer par un client lorsqu'il fait une commande) et le `deliveryCost` (le coût d'une livraison à payer par MyFoodora).

Le cahier des charges demande aussi de permettre à la plateforme de calculer le profit et le chiffre d'affaire. Il a été décidé de faire ce calcul entre deux dates en utilisant des objets de la classe `Calendar`. En effet ces calculs sont faits grâce aux méthodes `totalIncome` et `totalProfit` qui prennent en paramètre les deux dates `Calendar`. Ainsi, grâce à ces fonctions, on peut calculer n'importe quel profit ou chiffre d'affaire.

Ces fonctions sont appelées afin de répondre au deuxième problème posé par le cahier des charges : pouvoir ajuster les paramètres pour atteindre un certain profit. Cependant, les stratégies pour le faire sont différentes et c'est pourquoi un **Strategy Pattern** est aussi utilisé dans ce cas. L'interface `TargetProfitPolicy` avec sa méthode `meetTargetProfit` est utilisée et les classes `TargetProfitPolicyDeliveryCost` (ajustement du coût de livraison en fonction du profit à atteindre), `TargetProfitPolicyMarkupPercentage` (ajustement du pourcentage de marque ajouté sur les commandes en fonction du profit à atteindre) et `TargetProfitPolicyServiceFee` (ajustement du prix de service en fonction du profit à atteindre) en héritent. De même que pour les classes `DeliveryPolicy`, un **Factory Pattern** a été utilisé avec la classe `TargetProfitPolicyFactory` et sa méthode `chooseDeliveryPolicy`.



Gestion globale


Une fois que toutes ces classes ont été mises en place, on peut passer au développement du coeur de la plateforme : la classe `MyFoodora`. Cette classe définit toute la plateforme et c'est donc elle qui contient en quelque sorte toute la base de donnée. Cette classe a donc pour attribut une liste de tous les `User` dans un `ArrayList`. Cette liste de `User` contient donc en particulier tous les `Restaurant` et donc toutes les cartes proposées par ceux-ci avec les `FoodItem` qui les composent. Ainsi, il ne reste plus qu'à stocker toutes les informations administratives : une archive de toutes les commandes qui ont été faites avec un `ArrayList` de `Order`, et toutes les grandeurs économiques telles que le `serviceFee`, le `markupPercentage` et le `deliveryCost`.

Il faut d'abord mettre en place la sérialisation de la plateforme afin de la sauvegarder puis de la recharger lorsqu'elle serait interrompue. C'est pourquoi une méthode `saveMyFoodora` est mise en place, elle prend en charge la sérialisation et la sauvegarde dans un fichier nommé `"MyFoodora.ser"`. Une méthode statique notée `loadMyFoodora` qui renvoie un objet de la classe `MyFoodora` permet de lire ce fichier sérialisé. Ainsi, la plateforme peut maintenant être sauvegardée.

Il faut également gérer l'inscription et la connexion des utilisateurs à la plateforme. La connexion se fait à travers la méthode `login` de la classe `MyFoodora`. Cette méthode cherche le `User` qui correspond avec les `userName` et `password` donnés en paramètre. Ce `User` sera récupéré et stocké dans une variable de l'interface de l'utilisateur. Il aura ainsi accès à toutes les actions que son rôle lui permet de faire. Pour l'inscription à la plateforme, la méthode `register` de `MyFoodora` est utilisée, celle-ci fait appel à la classe `UserFactory` présentée précédemment.

Dans la classe `MyFoodora` sont aussi implémentées toutes les méthodes de calcul financier présentées dans la partie précédente.

Enfin, toutes les fonctions de la gestion de la liste des `User` sont implémentées comme par exemple les méthodes `addUser` (pour ajouter un nouvel utilisateur hors inscription), `removeUser` (pour supprimer un utilisateur), `findUserByUniqueID` (pour trouver l'utilisateur correspondant à un certain index d'identification).

<<Java Class>>	
	
fr.eop.is1220.MyFoodora	
S.F	serialVersionUID: long = -6956532311204306476L
□	users: ArrayList<User>
□	completedOrders: ArrayList<Order>
□	serviceFee: double
□	markupPercentage: double
□	deliveryCost: double
□	targetProfitPolicy: TargetProfitPolicy = new TargetProfitPolicyServiceFee()
□	deliveryPolicy: DeliveryPolicy = new FastestDeliveryPolicy()
□	shippedOrderPolicy: SorterFoodItem = new SorterCounter()
□	userFactory: UserFactory
C	MyFoodora(double, double, double)
●	saveMyFoodora():void
S	loadMyFoodora():MyFoodora
●	register(String, String, String, String, String):void
●	login(String, String):User
●	new Offer(String):void
●	totalIncomeLastMonth():double
●	totalIncome(Calendar, Calendar):double
●	totalProfitLastMonth():double
●	totalProfit(Calendar, Calendar):double
●	averageIncomePerCostumer(Calendar, Calendar):double
●	getUsers():ArrayList<User>
●	setUsers(ArrayList<User>):void
●	displayUsers():void
●	addUser(User):void
●	removeUser(int):void
●	findUserByUniqueID(int):User
●	getCompletedOrders():ArrayList<Order>
●	setCompletedOrders(ArrayList<Order>):void
●	addCompletedOrders(Order):void
●	getServiceFee():double
●	setServiceFee(double):void
●	getMarkupPercentage():double
●	setMarkupPercentage(double):void
●	setDeliveryCost(double):void
●	getDeliveryCost():double
●	getTargetProfitPolicy():TargetProfitPolicy
●	setTargetProfitPolicy(TargetProfitPolicy):void
●	getDeliveryPolicy():DeliveryPolicy
●	setDeliveryPolicy(DeliveryPolicy):void
●	getShippedOrderPolicy():SorterFoodItem
●	setShippedOrderPolicy(SorterFoodItem):void
●	getUserFactory():UserFactory

Exceptions à gérer

Afin de minimiser le risque d'erreur, plusieurs types d'exceptions ont du être créés. Ceux-ci sont répertoriés dans le tableau suivant :

Nom de l'exception	Classe qui l'utilise	Méthodes qui la lancent	Occurence
<code>AccountDeactivatedException</code>	<code>MyFoodora</code>	<code>login</code>	Lorsqu'un utilisateur souhaite se connecter alors que son compte est désactivé
<code>FoodItemNotFoundException</code>	<code>Restaurant</code>	<code>findDishByName</code> <code>findMealByName</code> <code>removeMeal</code> <code>removeDish</code> <code>addDish2Meal</code>	Lorsqu'un utilisateur souhaite gérer un <code>FoodItem</code> en envoyant un nom d'item qui n'existe pas.
<code>IdentificationIncorrectException</code>	<code>MyFoodora</code>	<code>login</code>	Lorsqu'un utilisateur souhaite se connecter avec le mauvais <code>userName</code> ou le mauvais <code>password</code> .
<code>NonReachableTargetProfitException</code>	<code>TargetProfitPolicy</code>	<code>meetTargetProfit</code>	Lorsqu'un manager souhaite atteindre un certain bénéfice alors qu'aucun moyen ne le permet.
<code>NoPlaceInMealException</code>	<code>Meal</code>	<code>addDish</code>	Lorsqu'un restaurant souhaite ajouter une entrée, un plat ou un dessert dans un menu qui en contient déjà un.
<code>OrderNotFoundException</code>	<code>Board</code>	<code>findObsById</code>	Lorsqu'un livreur souhaite valider une commande en donnant un numero d'identification qui ne correspond à aucune commande de son <code>Board</code> .
<code>UserNotFoundException</code>	<code>MyFoodora</code>	<code>removeUser</code> <code>findUserByUniqueID</code>	Lorsqu'un utilisateur n'est pas trouvé dans la base de donnée.

Tests effectués

Afin de vérifier le code alors qu'il n'y a pas encore d'interface, il faut effectuer des tests. Plusieurs tests ont été réalisés pour les classes les plus importantes et cela a permis de détecter de nombreuses erreurs de codage de l'application. A partir de ce moment, la démarche de développement est passée en mode Test-Driven Development.

Afin de réaliser des tests, il fallu soumettre chacune des classes à un environnement adéquat, c'est pourquoi la classe `MyFoodoraExample` a été créée : elle permet de créer une plateforme avec déjà plusieurs utilisateurs et plusieurs cartes proposées. Cette plateforme est stockée dans le fichier "`MyFoodora.ser`". Ce fichier est ensuite importé à chaque fois qu'une classe est testée en utilisant l'expression `@BeforeClass`.

Voici dans le tableau ci-dessous la liste des tests qui ont été faits :

Classe testée	Nom du test	Objectif
MyFoodora	<code>testSaveMyFoodora</code>	Tester la sauvegarde de <code>MyFoodora</code> grâce à la sérialisation.
	<code>testLoadMyFoodora</code>	Tester le chargement de <code>MyFoodora</code> grâce à la sérialisation.
	<code>testRegister</code>	Tester l'inscription d'un utilisateur.
	<code>testLogin</code>	Tester la connexion d'un utilisateur.
	<code>testLoginWhenWrongIdentification</code>	Tester la gestion de l'exception lorsqu'un utilisateur se trompe de <code>userName</code> ou de <code>password</code> .
	<code>testLoginWhenAccountDeactivated</code>	Tester la gestion de l'exception lorsqu'un utilisateur souhaite se connecter sur un compte désactivé.
	<code>testDisplayUsers</code>	Tester l'affichage des utilisateurs.
	<code>testFindUserByUniqueIDWhenNotInTheSystem</code>	Tester la gestion de l'exception lorsqu'un utilisateur non existant est recherché.
Courier	<code>testGetBoard</code>	Tester le stockage et l'affichage correct du <code>Board</code> correspondant aux commandes en attente du livreur.
	<code>testAcceptDeliveryCall</code>	Tester les conséquences lorsqu'un livreur accepte de livrer une commande.
Restaurant	<code>testFindDishByNameWhenInMenu</code>	Tester l'affichage d'un plat cherché par un utilisateur.

	testFindDishByNameWhenNotInMenu	Tester la gestion de l'exception lorsqu'un utilisateur fait une faute de frappe pour chercher un plat.
	testAddDish	Tester la création d'un nouveau plat de la carte.
	testRemoveDish	Tester la suppression d'un plat de la carte.
	testFindMealByName	Tester la recherche d'un menu dans la carte.
	testAddMeal	Tester la création d'un nouveau menu de la carte.
	testRemoveMeal	Tester la suppression d'un menu dans la carte.
	testAddDish2Meal	Tester le bon ajout d'un plat à un menu de la carte.
	testDisplaySortedFoodItems	Tester l'affichage des plats en fonction des tris et de leurs politiques d'affectation.
	testDisplayMenu	Tester l'affichage de la carte du restaurant.
Customer	testDisplayFidelityInfo	Tester l'affichage des informations sur la fidélité.
	testGetHistoryOfOrders	Tester l'historique des commandes.
	testSubmitOrder	Tester les conséquences d'une validation d'une commande par un client.
	testPayOrder	Tester l'affichage de la phase de paiement.
	testRegisterFidelityCard	Tester le changement de carte de fidélité d'un client.
Manager	testRemoveUser	Tester la suppression d'un utilisateur par un manager.
	testActivateUser	Tester l'activation d'un utilisateur par un manager.
	testDeactivateUser	Tester la désactivation d'un utilisateur.
	testTotalIncome	Tester le calcul du chiffre d'affaire.
	testTotalProfit	Tester le calcul du bénéfice.
	testAverageIncomePerCustomer	Tester le calcul du chiffre d'affaire moyen par client.
	testMostSellingRestaurant	Tester le calcul du classement des meilleurs restaurants.

	<code>testLeastSellingRestaurant</code>	Tester le calcul du classement du pire restaurant.
	<code>testMostActiveCourier</code>	Tester le calcul du livreur le plus actif.
	<code>testLeastActiveCourier</code>	Tester le calcul du livreur le moins actif.
Dish	<code>testGetDishType</code>	Tester la récupération du type de plat.
	<code>testToString</code>	Tester l'affichage d'un plat.
	<code>testEqualsObject</code>	Tester l'égalité de deux plats.
Meal	<code>testMealString</code>	Tester l'affichage des menus.
	<code>testComputePrice</code>	Tester le calcul du prix d'un menu en fonction des paramètres.
	<code>testUpdate</code>	Tester la mise à jour d'un menu lorsqu'une carte le met en menu de la semaine.
	<code>testAddDish</code>	Tester l'ajout d'un plat au menu.
	<code>testAddDishWhenMealFull</code>	Tester l'ajout d'un plat et la gestion de l'exception lorsque le menu est plein.
Order	<code>testToString</code>	Tester l'affichage d'une commande en texte.
	<code>testSubmit</code>	Tester les conséquences d'une validation de commande.
	<code>testValidateOrderByCourier</code>	Tester les conséquences d'une validation de livraison de commande par un livreur.
Board	<code>testToString</code>	Tester l'affichage d'un Board contenant des offres pour les clients.
	<code>testClearBoard</code>	Tester le nettoyage du Board une fois qu'un client l'a vu.