

*Rapport de projet*

***Software Engineering***

# **MyFoodora**

COLAS Gaël  
PRABAKARAN Sylvestre

<b>Introduction</b>	<b>3</b>
Sujet du projet	3
Notations	3
<b>Objectifs et démarche</b>	<b>4</b>
Objectif du projet	4
Grands axes de développement	4
Démarche	5
<b>Architecture de l'application</b>	<b>7</b>
Sérialisation des classes	7
Gestion des cartes proposées	7
Gestion des utilisateurs	10
Gestion des commandes et des livraisons	13
Gestion économique	15
Gestion globale	16
<b>Exceptions à gérer</b>	<b>18</b>
<b>Tests effectués</b>	<b>19</b>
<b>Command Line User Interface</b>	<b>22</b>
Implémentation	22
Commandes	23
Cas Tests	25

# 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 dernière 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 :

Partie 1 : MyFoodora core (jusqu'au 11/12/2016)

Gaël COLAS	Sylvestre PRABAKARAN
Réflexion commune sur le choix de l'architecture	
Conception de la première version de l'UML	
Mise en place du projet Eclipse sous Git (hébergé sur GitLab)	
Codage de toutes les classes concernant l'implémentation des <b>FoodItem</b> ( <b>Meal</b> , <b>Dish</b> ) et le <b>Factory Pattern</b> associé ( <b>FoodItemFactory</b> )	Codage de toutes les classes concernant l'implémentation des <b>User</b> ( <b>Customer</b> , <b>Courier</b> , <b>Restaurant</b> , <b>Manager</b> ) et le <b>Factory Pattern</b> associé ( <b>UserFactory</b> )
Conception et codage de la classe <b>MealVisitor</b> , <b>Visitor Pattern</b> pour les classes <b>Meal</b>	
Mise en place de toutes les <b>Strategy Pattern</b> avec les classes <b>TargetProfitPolicy</b> , <b>DeliveryPolicy</b> et <b>SorterFoodItem</b> et leurs <b>Factory Pattern</b> associés.	Codage de la classe <b>Order</b> et mise en place des <b>Observer Pattern</b> pour les classes <b>Customer</b> et <b>Courier</b> avec la classe <b>Board</b> . Mise en place de la sérialisation
Début de mise en place de la classe <b>MyFoodora</b>	
Implémentation des méthodes correspondant à chaque action des <b>User</b> dans la classe <b>MyFoodora</b> .	Ajout et implémentation de toutes les méthodes correspondant aux actions faites par chaque <b>User</b> .
Implémentation des tests (dans les classes JUnit associées)	Implémentation des tests (dans les classes JUnit associées)

liés aux classes héritant de <code>FoodItem</code> et aux <code>Order</code> et correction de tous les bugs constatés.	liés à toutes les classes héritant de <code>User</code> et correction de tous les bugs constatés.
Rédaction du rapport	
Rédaction des parties : <ul style="list-style-type: none"> <li>- "Gestion des cartes proposées"</li> <li>- "Gestion économique"</li> <li>- "Gestion des exceptions"</li> <li>- "Exceptions à gérer"</li> <li>- "Tests effectués"</li> </ul> Mise à jour de l'UML lié au travail sur ces parties du code.	Rédaction des parties : <ul style="list-style-type: none"> <li>- "Sérialisation des classes"</li> <li>- "Gestion des utilisateurs"</li> <li>- "Gestion des commandes"</li> <li>- "Exceptions à gérer"</li> <li>- "Gestion globale"</li> </ul> Mise à jour de l'UML lié au travail sur ces parties du code.
Début de codage de l'interface en console avec <code>MyFoodoraClient</code> .	

## Partie 2 : MyFoodora user interface (jusqu'au 03/01/2017)

Gaël COLAS	Sylvestre PRABAKARAN
Conception de la vision globale du fonctionnement de l'interface (architecture client)	
Implémentation des commandes <code>register</code> et <code>login</code> .	Implémentation des commandes pour les classes : <ul style="list-style-type: none"> <li>- <code>Courier</code></li> <li>- <code>Customer</code></li> <li>- <code>Restaurant</code></li> <li>- <code>Manager</code></li> </ul> Et rédaction des fonctions <code>work</code> correspondantes.
Rédaction de l'exemple stocké dans la classe : <code>MyFoodoraExample</code> Et du cas test : " <code>testScenario1.txt</code> "	
Implémentation de la prise en charge des cas test par l'interface : commande <code>runTest</code>	
Rédaction de l'UML lié à l'interface	
Rédaction de la partie du rapport : Cas Test	Rédaction de la partie du rapport : Command Line User Interface

## 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.

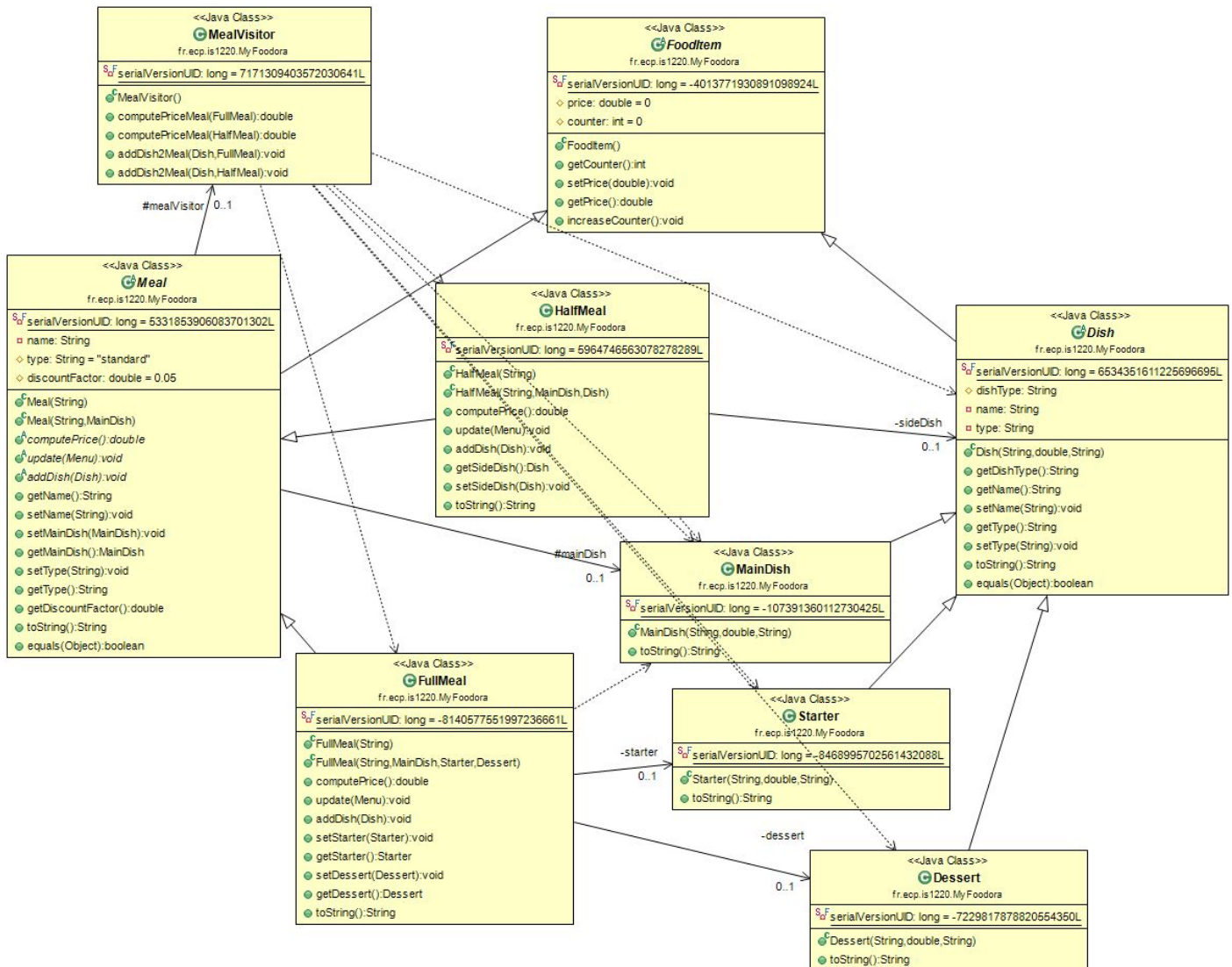
### Gestion des cartes proposées

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`.

Afin de respecter le principe d'ouverture-fermeture dans la création des `FoodItem`, nous avons eu recours à un `Factory Pattern`. Nous avons implémenté une `FoodItemFactory` qui grâce à ses méthodes `createDish` et `createMeal` prend en charge la création de tous types de `FoodItem`. Notre structure est ainsi facilement extensible à de nouveaux types de `FoodItem` (par exemple un Café) sans avoir besoin de changer le code client.

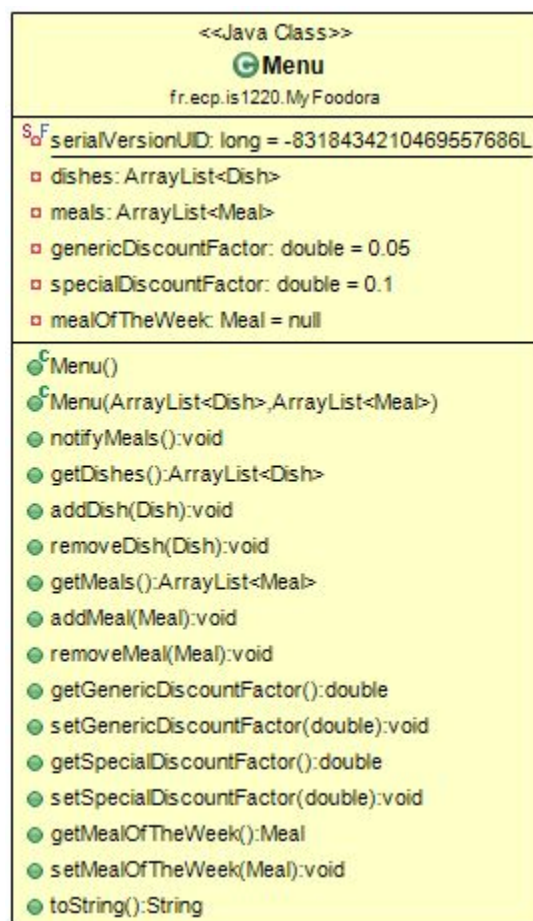
On note aussi que les classes `HalfMeal` et `FullMeal` sont similaires dans leur utilisation et font appel aux mêmes méthodes `computePrice` pour calculer le prix du menu ou `addDish2Meal` mais qui ne fonctionnent pas de la même façon en fonction du cas où le menu serait réduit ou grand. On fait donc 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 <code>genericDiscountFactor</code> ou le <code>specialDiscountFactor</code> ou le <code>mealOfTheWeek</code> .	Lorsque les valeurs en question sont modifiées, la méthode <code>notifyMeals</code> de la classe Menu est appelée et tous les Meal qui sont contenus dans la carte appellent la méthode <code>computePrice</code> .



## 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**.

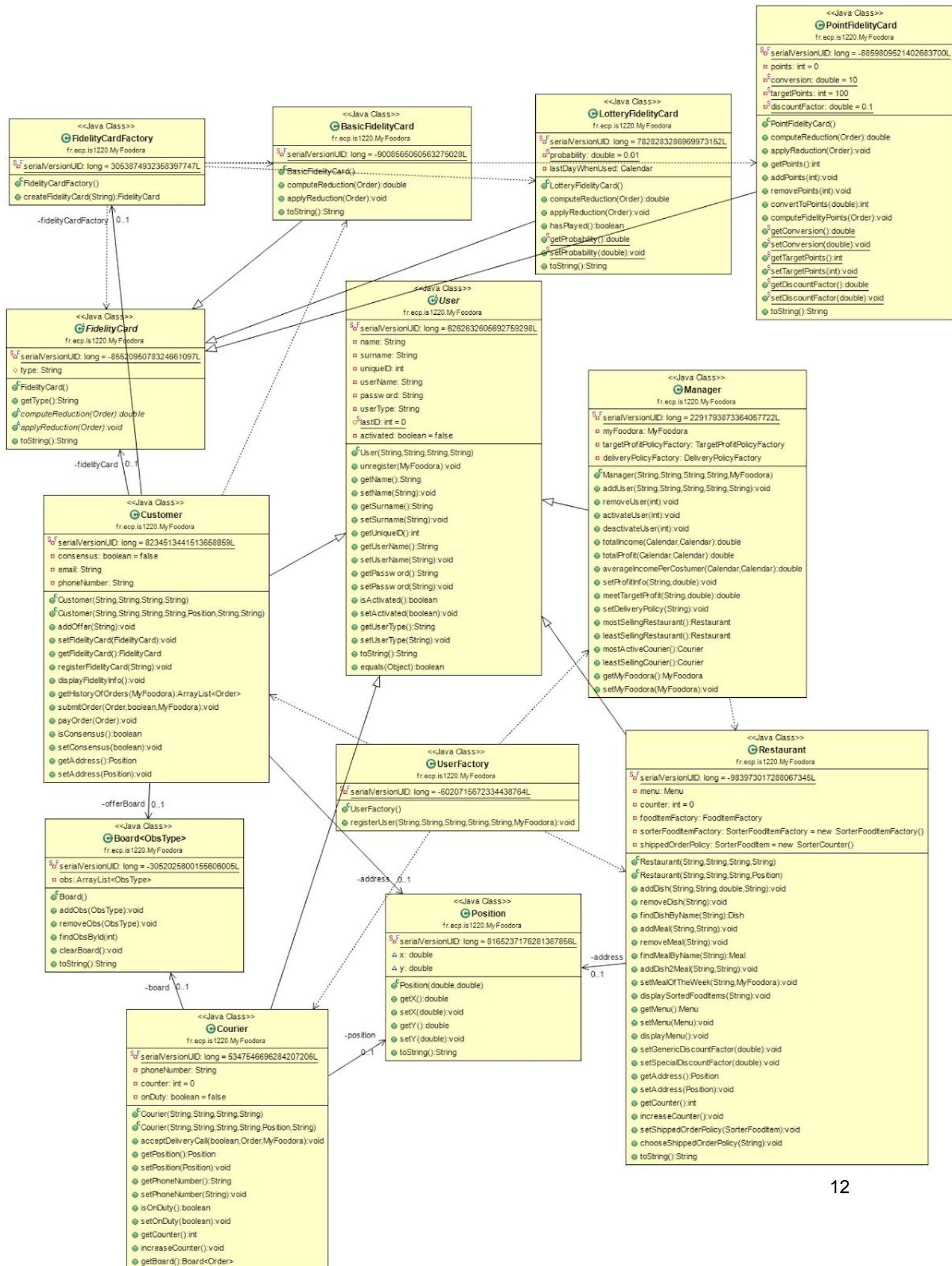
Les classes **Courier** et **Customer** doivent, en plus, observer des informations : c'est pourquoi il a été choisi ici d'adopter un **Observer Pattern** afin de répondre à ce besoin.

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 <code>MyFoodora</code> : Des chaînes de caractères <code>offer</code>	Dans les deux cas, un système de tableau d'affichage (création d'une classe <code>Board</code> ) 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 <code>MyFoodora</code> : Des commandes (classe <code>Order</code> )		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.

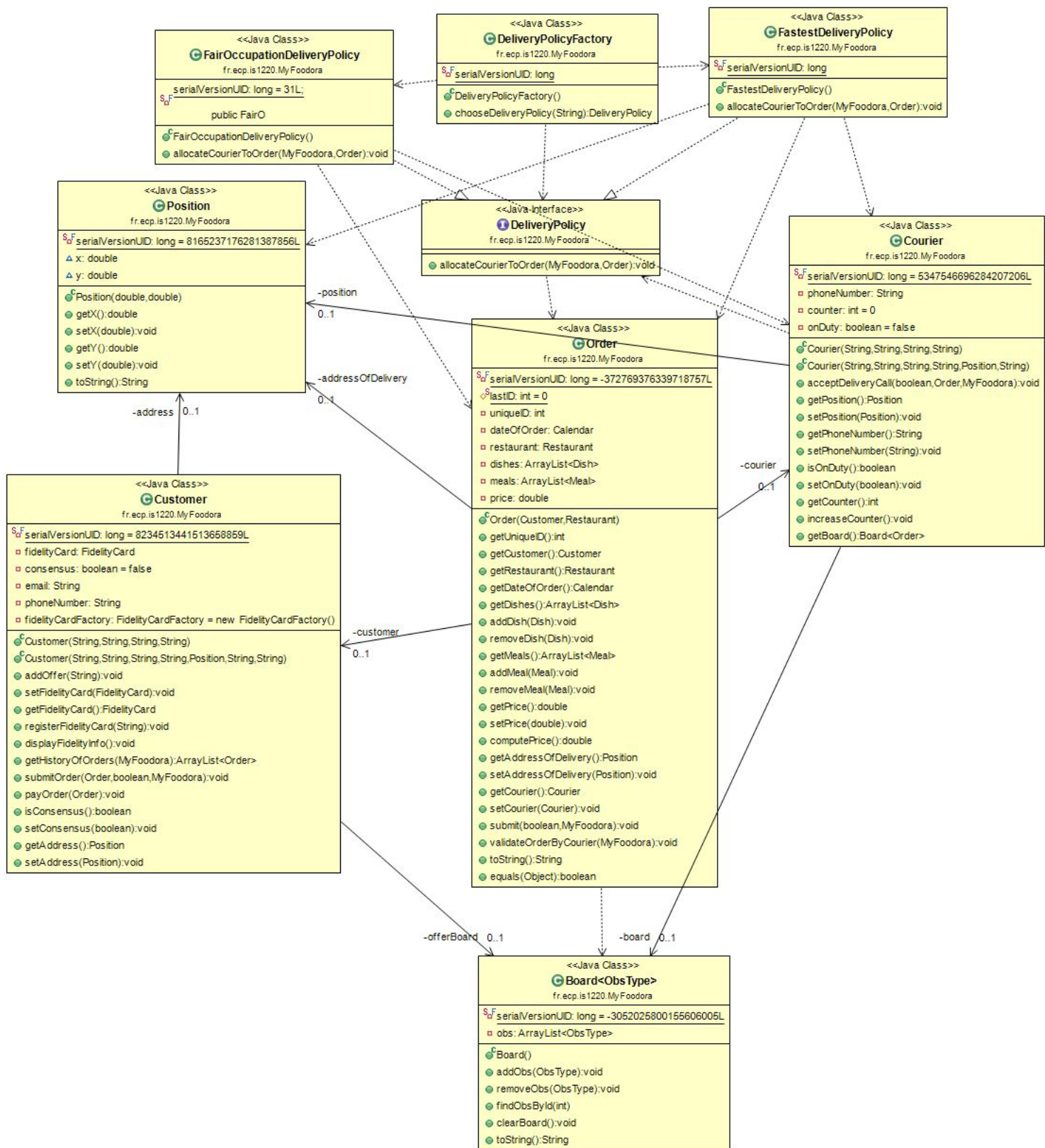
Chaque commande est faite par un client et est caractérisée par la classe `Order`. Cette classe comporte des attributs qui sont la date de la commande représentée par un objet de la classe `Calendar`, ainsi qu'une liste de `Dish`, une liste de `Meal` et un prix caractérisé par une variable `price` qui 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.

Voici le déroulement d'une commande traitée par MyFoodora :

- 1) Un client se connecte et passe une commande de plats à un restaurant
- 2) Il valide sa commande et applique ou non la réduction liée à son programme de fidélité
- 3) Un courrier est alloué à la commande selon la politique : le plus proche courrier ou celui qui a réalisé le moins de commande dans les courriers actifs
- 4) La commande s'affiche sur le board du courrier correspondant
- 5) Les conteurs des plats sont incrémentés
- 6) Le courrier se connecte et voit l'appel sur son board qu'il accepte ou non
- 7) Si il accepte : la commande est supprimée de son board et ajoutée aux commandes terminées. Le conteur du courrier est incrémenté
- 8) Si il refuse : un autre courrier est alloué

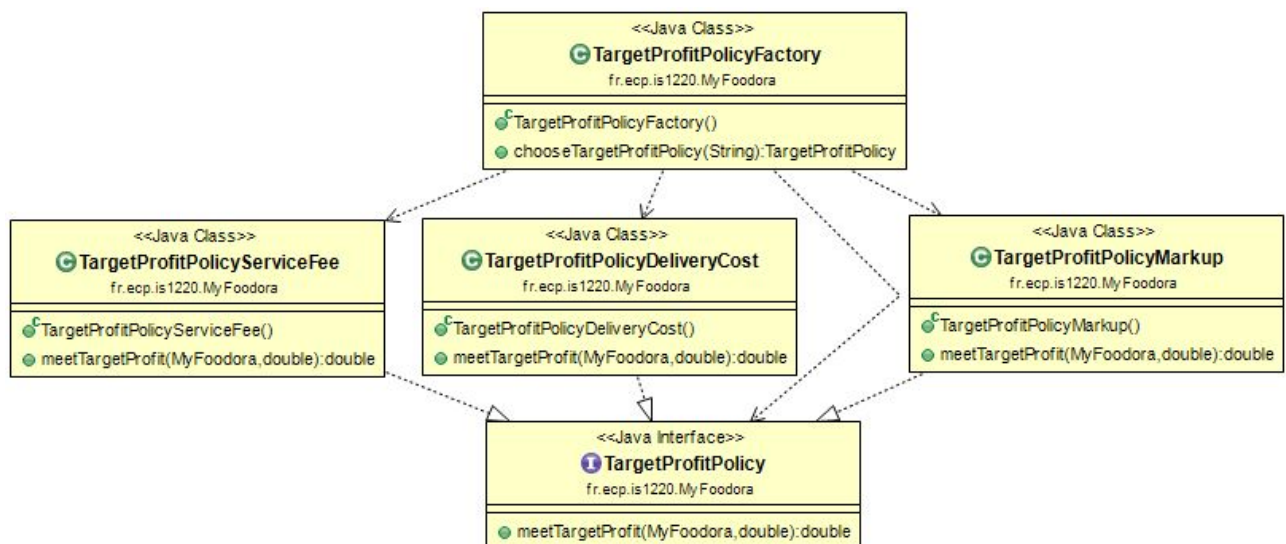


## Gestion économique

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. Par défaut, comme il a été requis, un manager existe : c'est le CEO (name = "", surname = "", username = "ceo", password = "123456789"). 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.

Comme toutes les fonctions disponibles pour un type de `User` donné sont stockées comme méthode de la classe, cette gestion permet de s'assurer qu'un `User` n'a accès qu'aux fonctions qui lui sont autorisées (par rapport à son type).

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 dû être créés. Ceux-ci sont répertoriés dans le tableau suivant :

Nom de l'exception	Classe qui l'utilise	Méthodes qui la lancent	Occurrence
<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 <b>Board</b> contenant des offres pour les clients.
	<code>testClearBoard</code>	Tester le nettoyage du <b>Board</b> une fois qu'un client l'a vu.



# Command Line User Interface

## Implémentation

Nous avons implémenté l'interface utilisateur de **MyFoodora** dans la classe **MyFoodoraClient**. Nous avons pu l'implémenter avec quelques modifications mineures du coeur de l'application liées à la nouvelle version du sujet, notre architecture est donc bien adaptée à la résolution du problème.

Les utilisateurs interagissent avec l'interface de 2 façons possibles :

- En tapant les commandes directement dans la console
- En remplissant un fichier texte des commandes souhaitées, puis en exécutant celles-ci au moyen de la commande "runtest testScenarioN.txt" ; les résultats sont ensuite stockés dans le fichier de sortie : "*testScenarioOutput.txt*".

Le fonctionnement de la méthode **login** de **MyFoodora** décrite plus haut permet de s'assurer qu'un utilisateur après s'être identifié avec son login et son mot de passe n'a accès qu'aux méthodes autorisées (propres à sa classe). La liste des méthodes auxquelles il a accès, ainsi que leur description, sont disponibles à tous moments par l'appel à la commande **help** de l'interface.

L'interface **MyFoodoraClient** fonctionne de la façon suivante :

- 1) L'utilisateur se connecte en utilisant la méthode **login <username> <password>**
- 2) L'interface récupère le **User** associé
- 3) L'interface lance la fonction **work** correspondant au type de **User**

Pour respecter le principe d'Open-Close, nous avons en effet séparé les fonctions des différentes classes du **main** de l'interface. Pour chaque type de User, les commandes auxquelles il a accès sont stockées dans la fonction **work** correspondante (**workCustomer** pour les **Customer** par exemple). Ainsi l'ajout d'un nouveau type de **User** ne nécessite pas de modifier le **main**.

Nous avons implémenté dans l'interface toutes les commandes demandées, ainsi que toutes celles qui nous semblaient adaptées à notre architecture de MyFoodora core. Elles sont détaillées plus concrètement dans notre cas test de la partie suivante.

## Commandes

Remarque : il faut lancer la classe `MyFoodoraInitialization` dans le package `MyFoodoraTests` pour avoir une version vierge de `MyFoodora` avant de lancer le cas test.

Pour entrer des arguments `String` dans les commandes, ceux-ci doivent être délimités par des guillemets comme dans l'exemple de l'énoncé :

*`registerRestaurant "TourDargent" "45.1,66.2" "12345678"`*

Cette notation permet de traiter des noms de plats avec des espaces comme :

*`addDishRestaurantMenu "maki surimi" "mainDish" "standard" "8"`*

Nous avons implémenté toutes les commandes demandées en énoncé. Nous nous parlerons donc pas de celles qui ont été implémentées telles quelles : mêmes arguments et mêmes outputs. Nous décrirons ici les nouvelles commandes que nous avons implémentées, ainsi que celles dont nous avons modifiées les arguments.

Concernant les `User`:

Nous avons changé l'ordre des paramètres des “register” pour que cela soit adapté à la syntaxe du core de `MyFoodora` qui nous semblait plus adaptée. Voilà les nouvelles commandes disponibles pour le Manager :

- `registerRestaurant <name> <username> <password> <address>`
- `registerCustomer <firstName> <lastName> <username> <password> <address>`
- `registerCourier <firstName> <lastName> <username> <password> <position>`

Nous avons également fait en sorte que les `Manager` puissent s'inscrire sur la plateforme sans avoir besoin de passer par un `Manager`. Voici la commande à utiliser sur l'écran d'accueil :

- `register <firstName> <lastName> <username> <password> <address>` : register as customer into the system

Concernant les `Manager`:

Nous avons revu la gestion de la politique économique. Si le `Manager` veut modifier un des paramètres économiques (“profitInfo” = “markup” percentage, “serviceFee” or “deliveryCost”), il utilise les fonctions suivantes :

- `meetTargetProfit <profitInfo> <targetProfit>` : compute the value of the parameter “profitInfo” needed to reach a given “targetProfit”
- `setProfitInfoValue <profitInfo> <value>` : set the “profitInfo” at the new value “value”

Nous avons également implémentées les commandes qui étaient déjà prévues dans le core de `MyFoodora` :

- `activateUser <username>` : activates the user
- `deactivateUser <username>` : deactivates the user

Concernant les `Restaurant`:



Nous avons implémentées les commandes qui étaient déjà prévues dans le core de MyFoodora :

- showSortedMeals <> : display all the meals of the Menu sorted w.r.t. the number of time they have been picked
- showSortedDishes <> : display all the dishes of the Menu sorted w.r.t. the number of time they have been picked

Concernant les [Customer](#):

Nous avons implémentées les commandes qui étaient déjà prévues dans le core de MyFoodora :

- registerFidelityCard <cardType> : for the currently logged customer to register to a new fidelity program  
(“cardType” = “basic”, “point”, “lottery” or “none” to unregister)
- displayFidelityInfo <> : for the currently logged customer to display the information of his fidelity program  
(number of points for a points fidelity program)
- historyOfOrders <> : for the currently logged customer to display the history of all his past orders

Concernant les [Courier](#):

Nous avons enlevé des arguments inutiles : pas besoin de l’username du [Courier](#) puisqu’il est déjà loggé dans le système. Voilà les nouvelles commandes :

- onDuty <>
- offDuty <>

Nous avons également ajouté d’autres commandes pour répondre aux besoins énoncés dans la partie 1 et prévu dans le core de MyFoodora.

- acceptDeliveryCall <orderId> <answer> : for the currently logged courier to accept/refuse a delivery call for the order of orderId.  
(answer = “yes” or “no”)

Concernant les [Order](#):

Dans notre modèle, les [Order](#) ne sont pas repérés par un nom, mais par un ID unique. On retrouve ensuite les [Order](#) grâce à la méthode `findOrderByID`, le paramètre “orderName” n’a donc pas de signification dans notre architecture et a été enlevé des commandes. La date a également été enlevé des paramètres car elle est récupéré directement lors de la validation grâce à la classe Java : [Calendar](#).

De plus, dans notre architecture, un [Customer](#) ne peut réaliser qu’un seul [Order](#) à la fois, il n’y a donc pas besoin de repérer [Order](#) auquel on applique les fonctions puisqu’il s’agit de [Order](#) actif.

- createOrder <restaurantName>
- addItem2Order <itemName>
- endOrder <applyReduction> : submit the order to today's date and applies the order depending on the applyReduction value “yes” or “no”

## Cas Tests

Pour tester le fonctionnement de l'interface utilisateur et des différentes commandes CLUI (command line user interpreter), la création d'un cas test était nécessaire. Celui-ci, stocké dans le fichier *testScenario1.txt* du dossier *eval*, teste les différentes commandes CLUI de la classe [MyFoodoraClient](#) dans le scénario décrit ci-dessous.

Type utilisateur actif	Action	Ligne de commande CLUI
Manager	Login	login "ceo" "123456789"
	Ajout de 2 restaurants au système	registerRestaurant "Sushi Saka" "saka" "password" "12.4,147.2"
		registerRestaurant "Pizza Hut" "hut" "dateofbirth" "125.3,78.3"
	Ajout de 2 clients	registerCustomer "Jamie" "Oliver" "joliver" "chef123" "12.2,45.3"
		registerCustomer "Gordon" "Ramsay" "gramsay" "chef456" "45.8,145.3"
	Ajout de 3 courriers	registerCourier "Forrest" "Gump" "fgump" "shrink" "784.2,12.3"
		registerCourier "Usain" "Bolt" "ubolt" "jamaik" "123.54,76.2"
		registerCourier "Speedy" "Gonzalez" "speedgonz" "ch33se" "12.3,125.3"
	Affichage des restaurants	showRestaurants
	Affichage des clients	showCustomers
	Logout	logout
Restaurant	Login	login "saka" "password"
	Ajout de plats au menu	addDishRestaurantMenu "maki surimi" "mainDish" "standard" 8
		addDishRestaurantMenu "maki radis" "mainDish" "glutenFree" 7.5
		addDishRestaurantMenu "sushi avocat" "mainDish" "vegetarian" 8.5

		addDishRestaurantMenu "maki surimi" "mainDish" "standard" 8
		addDishRestaurantMenu "miso" "starter" "vegetarian" 3
		addDishRestaurantMenu "boule coco" "dessert" "vegetarian" 5
	Ajout de menus au menu	createMeal "S3" "full"
		addDish2Meal "miso" "S3"
		addDish2Meal "sushi avocat" "S3"
		addDish2Meal "boule coco" "S3"
		showMeal "S3"
		createMeal "M7" "half"
		addDish2Meal "miso" "M7"
		addDish2Meal "maki surimi" "M7"
	Choix du menu de la semaine	setSpecialOffer "S3"
	Affichage du menu	showMenuItem
	Changement du Generic Discount Factor	setGenericDiscountFactor "0.055"
	Vérification que le menu a bien été mis à jour	showMenuItem
	Logout	logout

Restaurant	Login	login "hut" "dateofbirth"
	Ajout de plats au menu	addDishRestaurantMenu "4 formaggi" "mainDish" "vegetarian" "12"
		addDishRestaurantMenu "margherita" "mainDish" "standard" "13.5"
		addDishRestaurantMenu "grecini" "starter" "vegetarian" "4.5"
		addDishRestaurantMenu "arancine" "starter" "vegetarian" "3"
		addDishRestaurantMenu "tiramisu" "dessert"

		"vegetarian" "6.5"
	Ajout de menus au menu	createMeal "complet" "full"
		addDish2Meal "4 formaggi" "complet"
		addDish2Meal "arancine" "complet"
		addDish2Meal "tiramisu" "complet"
		showMeal "complet"
		createMeal "midi" "half"
		addDish2Meal "margherita" "midi"
		addDish2Meal "grecini" "midi"
	Choix du menu de la semaine	setSpecialOffer "midi"
	Affichage du menu	showMenuItem
	Changement du Generic Discount Factor	setSpecialDiscountFactor 0.09
	Vérification que le menu a bien été mis à jour	showMenuItem
Customer	Logout	logout
	Register	register "Auguste" "Gusteau" "agusteau" "chef789" "78.9,1.23"
	Login	login "agusteau" "chef789"
	Changement de programme de fidélité	registerFidelityCard "lottery"
	Affichage des données de fidélités	displayFidelityInfo
	Choix d'un restaurant pour commander	createOrder "Pizza Hut"
	Remplissage de la commande	addItem2Order "midi"
		addItem2Order "complet"
		addItem2Order "tiramisu"
	Soumission de la commande avec réduction	endOrder "yes"

	Logout	logout
Customer	Login	login "joliver" "chef123"
	Changement de programme de fidélité	registerFidelityCard "point"
	Affichage des points de fidélités	displayFidelityInfo
	Choix d'un restaurant pour commander	createOrder "Sushi Saka"
	Remplissage de la commande	addItem2Order "S3"
		addItem2Order "M7"
		addItem2Order "miso"
		addItem2Order "boule coco"
	Soumission de la commande sans réduction	endOrder "no"
	Logout	logout
Courier	Login	login "ubolt" "jamaik"
	Acceptation d'une livraison	acceptDeliveryCall "1" "yes"
	Refus d'une livraison	acceptDeliveryCall "2" "no"
	Mise dans l'état "fin de service"	offDuty
	Logout	logout
Courier	Login	login "speedgonz" "ch33se"
	Refus d'une livraison	acceptDeliveryCall "2" "no"
	Logout	logout
Courier	Login	login "fgump" "shrink"
	Acceptation d'une livraison <b>On vérifie que la commande a été transmise au livreur actif suivant</b>	acceptDeliveryCall "2" "yes"
	Mise dans l'état "fin de service"	offDuty
	Logout	logout
Courier	Login	login "ubolt" "jamaik"

	Mise dans l'état "début de service"	onDuty
	Logout	logout
Courier	Login	login "speedgonz" "ch33se"
	Mise dans l'état "début de service"	onDuty
	Logout	logout
Manager	Login	login "ceo" "123456789"
	Calcul et affichage du profit du système depuis la création	showTotalProfit
	Calcul et affichage du profit du système sur le mois précédent	showTotalProfit "01/01/2017" "31/01/2017"
	Changement de la politique de livraison	setDeliveryPolicy "fairOccupation"
	Logout	logout
Customer	Login	login "joliver" "chef123"
	Affichage de l'historique des commandes	historyOfOrders
	Affichage des points de fidélités	displayFidelityInfo
	Choix d'un restaurant pour commander	createOrder "Sushi Saka"
	Remplissage de la commande	addItem2Order "M7"
		addItem2Order "miso"
		addItem2Order "boule coco"
	Soumission de la commande avec réduction	endOrder "yes"
	Logout	logout
<p>Commentaire :</p> <p>Dans la politique de livraison précédente : "livraison la plus rapide", comme c'était Usain Bolt qui était le plus proche du restaurant "Sushi Saka", c'est à lui que revenait la livraison.</p> <p>Maintenant la politique de livraison est : "occupation égale des livreurs".</p> <p>Comme Speedy Gonzalez est celui qui n'a pas encore fait de livraison, c'est à lui que revient la commande.</p> <p>On vérifie ci-dessous que Usain Bolt n'a rien sur son Board et que Speedy Gonzalez si.</p>		
Courier	Login	login "ubolt" "jamaik"

	Logout	logout
Courier	Login	login "speedgonz" "ch33se"
	Acceptation d'une livraison	acceptDeliveryCall "3" "yes"
	Logout	logout
Restaurant	Login	login "saka" "password"
	Affichage des menus triés selon le nombre de fois où ils ont été commandés	displaySortedMeals
	Affichage des plats triés selon le nombre de fois où ils ont été commandés	showSortedDishes
	Logout	logout
Manager	Login	login "ceo" "123456789"
	Calcul et affichage du profit du système depuis la création	showTotalProfit
	Calcul et affichage du profit du système sur le mois précédent	showTotalProfit "01/01/2017" "31/01/2017"
	Changement de la politique de profit	meetTargetProfit "markup" "20"
		setProfitInfoValue "markup" "0.12"
	<p>Commentaire : notre fonction meetTargetProfit prend en donnée le revenu du mois précédent pour calculer la valeur de "ProfitInfo" qui permettrait d'obtenir le profit souhaité ce mois-ci.</p> <p>Comme dans notre initialisation de MyFoodora, il n'a pas eu de commande au mois précédent, le revenu du mois précédent est nul.</p> <p>La fonction renvoie donc une exception : "NonReachableTargetProfit"</p>	
	Affichage du nouveau profit total	showTotalProfit
	Affichage de la liste des clients <b>dont ceux inscrits seuls</b>	showCustomers
	Affichage de la liste des courriers selon leur nombre de livraisons	showCourierDeliveries
	Affichage de la liste des restaurants triés selon leur nombre de commandes	showRestaurantTop
	Désactivation d'un courrier	deactivateUser "ubolt"

	Logout	logout
Courier	Login On vérifie qu'il a bien été désactivé	login ""ubolt"" "jamaik"
	Fermeture de MyFoodora	close