Rapport OOSE - Projet MyFoodora

Alexandre Gravereaux
Hugues d'Hardemare

Génie Logiciel Orienté Objet

06/06/2025

Professors :
Paolo Ballarini
paolo.ballarini@centralesupelec.fr
Arnaud Lapitre

Contents

1 Introduction

Ce projet a pour objectif de concevoir et développer une application de livraison de repas en ligne, baptisée *myFoodora*, s'inspirant des plateformes existantes telles que *Uber Eats* ou *Deliveroo*. L'application est entièrement développée en Java et accessible via une interface en ligne de commande. Elle vise à simuler un environnement numérique centralisé où interagissent plusieurs types d'utilisateurs : gestionnaires, restaurateurs, clients et livreurs. Chaque acteur dispose d'une interface dédiée, lui permettant de gérer ses propres fonctionnalités spécifiques : création de menus pour les restaurants, passage de commandes pour les clients, organisation des livraisons pour les coursiers, et supervision globale pour les gestionnaires.

Le système central, *myFoodora*, assure la coordination des différentes entités. Il gère les données des utilisateurs, applique les politiques de livraison et de rentabilité, et notifie les utilisateurs des offres spéciales ou évolutions du service. L'architecture logicielle permet la gestion de repas à la carte ou préconfigurés (standard, végétariens, sans gluten), l'activation de programmes de fidélité, ainsi que le suivi des performances économiques du système (marges, bénéfices, coûts).

Le présent rapport a pour vocation de fournir une vue d'ensemble du projet : il décrit le fonctionnement général de l'application, présente deux scénarios de test illustratifs, détaille les choix techniques réalisés durant le développement, et rend compte de l'organisation du travail collaboratif au sein de l'équipe. Le développement s'est appuyé sur l'environnement de développement Eclipse.

2 Fonctionnement de l'application

2.1 Accessibilité et lancement de l'application

L'application fonctionne grâce à une interface en ligne de commande (CLUI). L'utilisateur interagit avec le système en entrant des commandes dans le terminal.

Afin d'accéder à cette interface, on exécute le programme `Main.java` qui ouvre le terminal. On peut ensuite générer un environnement ayant un manager, r restaurants, c clients et l livreurs grâce à la commande `setup r c l`. La commande `setup` permet de créer un manager et autant de restaurants, de clients et de livreurs que nous le désirons. Pour chacun de ces utilisateurs, les attributs sont initialisés de la manière suivante :

- **manager** : le manager est initialisé avec l'identifiant `"ceo"`, le mot de passe `"123"`, le prénom `"John"` et le nom de famille `"Doe"`.

- **restaurants** : les restaurants ont comme noms `"Restaurant1"`, ..., `"Restaurantn"`, comme identifiants `"rest1"`, ..., `"restn"`, comme mots de passe `"restpass1"`, ..., `"restpassn"` et des adresses choisies aléatoirement dans un carré de côté de longueur 100. Chaque restaurant se voit attribué trois entrées, trois plats et trois desserts. Nous avons effectué ce choix de sorte à ce que chaque catégorie de plat, à savoir standard, végétarien ou sans gluten, soit représentée. Les noms donnés aux entrées sont `starter_1`,

`starter_2` et `starter_3`, ceux donnés aux plats `main_1`, `main_2` et `main_3`, et, de la même manière, les noms donnés aux desserts sont `dess_1`, `dess_2` et `dess_3`. Ensuite, pour avoir une carte qui soit la plus réaliste possible, nous avons attribué à ces plats des prix générés aléatoirement, avec des fourchettes de prix étant différentes pour les entrées, pour les plats et pour les desserts. Aussi, chaque restaurant se voit attribué 3 menus, nommés selon la même logique que précédemment `meal_1`, `meal_2` et `meal_3`. Chaque menu a une catégorie différente : l'un est standard, un autre végétarien et le dernier sans-gluten. Enfin, nous attribuons à tous ces menus les plats de la même catégorie.

- **clients** : les clients ont les prénoms `"Customer1"`., ... `"Customern"`, comme noms de famille `"Lastname1"`, ... `"Lastnamen"`, comme identifiants `"cust1"`, ... `"custn"`, comme mot de passes `"custpass1"`, ... `"custpassn"`, comme localisation une position générée aléatoirement, de la même façon que pour les restaurants, puis une adresse mail et un numéro de téléphone inutiles dans la suite du projet. Pour initialiser la base de données des commandes, chacun de ces clients a effectué un nombre compris entre 0 et 20 commandes à une date choisie au hasard dans les trois mois précédents la date actuelle, dans les restaurants (eux aussi choisis aléatoirement dans la liste des restaurants déjà établie). De plus, les clients peuvent choisir de recevoir ou non des notifications relatives aux offres proposées par les restaurants. Dans cette initialisation, cela est également déterminé aléatoirement, donc certains peuvent les recevoir et d'autres non.

- **livreurs** : les livreurs ont comme noms `"Courier1"`, ... `"Couriern"`, comme identifiants `"courier1"`, ... `"couriern"`, comme mots de passe `"courpass1"`, ... `"courpassn"` et de même que précédemment, une localisation aléatoire. Les livreurs sont aussi caractérisés par un état « occupé » ou « libre », cet état étant initialisé au hasard pour chacun des livreurs.

Tous les utilisateurs possèdent aussi un identifiant qui lui est propre, pour faciliter son identification. Enfin, la commande setup initialise aussi les paramètres de l'application, en fixant à 20 pourcent le pourcentage de marge, à 3 euros les frais de livraison et à 2,5 euros les frais de fonctionnement de l'application.

La liste des commandes est spécifiée à la section ...

2.2 Structure de l'application

Nous avons organisé le projet selon une architecture modulaire, répartie en plusieurs packages contenant des classes aux responsabilités distinctes. Voici une synthèse des principaux packages et classes du projet :

- Le package **fidelity** regroupe les classes liées aux programmes de fidélité, telles que `FidelityCard` (interface), et ses implémentations `BasicFidelityCard`, `LotteryFidelityCard`, et `PointFidelityCard`.
- Le package **food** contient les classes représentant les repas et plats, notamment `Dish`, `Meal`, ainsi que ses sous-classes `FullMeal` et `HalfMeal`.

- Le package **policies** est subdivisé en sous-packages pour gérer les différentes politiques de l'application :
  - **delivery** : classes pour la gestion des politiques de livraison, comme `DeliveryPolicy`, `FairOccupationDeliveryPolicy`, et `FastestDeliveryPolicy`.
  - **ordersorting** : classes définissant l'ordre de traitement des commandes, telles que `OrderedDishPolicy` et `OrderedMealPolicy`.
  - **profit** : classes gérant les stratégies de profit, par exemple `ProfitPolicy`, `MarkupProfitPolicy`, et `ServiceFeeProfitPolicy`.
- Le package **system** contient les classes centrales pour le fonctionnement de l'application, comme `MyFoodoraSystem`, `MyFoodoraCLI` pour l'interface en ligne de commande, et `Order` représentant une commande.
- Le package **users** définit les différents types d'utilisateurs du système. Il comprend la classe abstraite `User` et ses extensions : `Manager`, `Restaurant`, `Courier`, et `Customer`.
- Le package **test** regroupe les tests unitaires importants du projet, organisés par classe testée, tels que `BasicFidelityCardTest`, `DishTest`, `UserTest`, etc.

2.3 Explication des différentes commandes

Nous rappelons ici les commandes disponibles dans l'application (certaines peuvent avoir été modifiées).

- `login <username> <password>` : permet à un utilisateur de se connecter
- `logout` : permet à l'utilisateur connecté de se déconnecter
- `registerCustomer <firstName> <lastName> <username> <address> <password>` : permet à un manager de créer un client
- `registerCourier <firstName> <lastName> <username> <position> <password>` : permet à un manager d'enregistrer un livreur
- `registerRestaurant <name> <address> <username> <password>` : permet à un manager d'enregistrer un nouveau restaurant
- `addDishRestaurantMenu <dishName> <dishCategory> <foodCategory> <unitPrice>` : permet à un restaurant d'ajouter un plat à son menu
- `createMeal <mealName> <foodCategory>` : permet à un restaurant de créer un repas

- `addDish2Meal <dishName> <mealName>` : permet à un restaurant d'ajouter un plat à un repas
- `saveMeal <mealName>` : permet à un restaurant d'enregistrer un repas
- `setSpecialOffer <mealName>` : permet à un restaurant de désigner une offre spéciale
- `removeFromSpecialOffer <mealName>` : permet à un restaurant de retirer une offre spéciale
- `createOrder <restaurantName> <orderName>` : permet à un client de créer une commande
- `addItem2Order <orderName> <itemName>` : permet à un client d'ajouter un article à une commande
- `endOrder <orderName> <date>` : permet à un client de finaliser une commande
- `onDuty <username>` : permet à un livreur de se rendre disponible
- `offDuty <username>` : permet à un livreur d'indiquer qu'il n'est plus en service
- `findDeliverer <orderName>` : permet à un restaurant de désigner un livreur pour une commande
- `setDeliveryPolicy <PolicyName>` : permet au manager de changer la politique de livraison
- `setProfitPolicy <ProfitPolicyName>` : permet au manager de changer la politique de profit
- `associateCard <username> <cardType>` : permet au manager d'associer une carte de fidélité à un client
- `showCourierDeliveries` : affiche la liste des livreurs classés par nombre de livraisons
- `showRestaurantTop` : affiche la liste des restaurants classés par commandes livrées

- `showCustomers` : affiche la liste des clients
- `showMenuItem <restaurantName>` : affiche le menu d'un restaurant
- `showTotalProfit` : affiche le profit total depuis la création
- `showTotalProfit <startDate> <endDate>` : affiche le profit total sur une période
- `runtest <testScenario-file>` : exécute les commandes depuis un fichier de scénario
- `help` : affiche l'aide pour l'utilisation des commandes disponibles

3 Scénarios de test

Nous avons élaboré plusieurs scénarios de test permettant de tester des parcours utilisateurs typiques, qu'il s'agisse de customer, de restaurants ou bien de managers. L'ensemble des scénario crées se situe dans le dossier eval/ à la racine du projet.

Afin de lancer un scénario, il suffit d'exécuter le fichier `Main.java` et de rentrer dans le terminal:
`runtest eval/testScenario1.txt`

De même, on peut tester les autres scénaris.
Remarque: nous n'avons pas réussi à configurer un fichier .ini.

4 Choix d'implémentations

Nous avons majoritairement suivi les consignes de l'énoncé, tout en prenant certaines libertés dans l'organisation du code, en nous appuyant sur le diagramme UML que nous avons conçu en amont du développement. Ce diagramme nous a servi de guide pour structurer notre application de manière cohérente et modulaire.

Par exemple, nous avons choisi de représenter le menu non pas comme une classe à part entière, mais comme un simple attribut au sein de la classe `Restaurant`. Cela nous a semblé plus pertinent, le menu étant directement lié au restaurant qui le propose, sans nécessiter une abstraction supplémentaire.

À l'inverse, nous avons décidé de créer une classe `Order` à part entière afin de mieux encapsuler les informations liées aux commandes (plats commandés, client, date, statut, etc.) et de faciliter leur gestion dans l'application.

Enfin, bien que le sujet évoque la notion de repas, nous avons fait le choix de modéliser plus finement les types de repas en définissant deux sous-classes : `HalfMeal` et `FullMeal`. Cela nous a permis de mieux représenter les différences de structure et de prix entre les repas partiels et

les repas complets.

Ces décisions nous ont permis de maintenir une architecture claire, évolutive et conforme aux principes de la programmation orientée objet.

5 Principes de développement

Lors du développement de *MyFoodora*, nous avons appliqué le principe "open-closed" de la programmation orientée objet, selon lequel le code doit être ouvert à l'extension mais fermé à la modification. Cela permet de faire évoluer l'application sans modifier le code existant.

Par exemple, la classe abstraite `User`, héritée par `Customer`, `Courier`, `Manager` et `Restaurant`, centralise les comportements communs aux utilisateurs. Si l'on souhaite ajouter un nouveau type d'utilisateur, il suffit de créer une nouvelle sous-classe sans toucher au reste du code.

Nous avons également défini une interface `FidelityCard` implémentée par plusieurs stratégies concrètes de cartes de fidélité (par exemple : `PointCard`, `LotteryCard`, `BasicCard`), ce qui permet de modifier ou d'ajouter de nouvelles politiques de fidélité sans toucher au reste du code.

Le design de l'application repose sur la séparation des responsabilités à travers plusieurs classes : une couche CLI (`MyFoodoraCLI`) pour l'interaction utilisateur, une couche métier pour les entités et leurs comportements, et un gestionnaire central (`MyFoodora`) qui agit comme un contrôleur. Cela nous a permis d'assurer une architecture claire et évolutive.

Afin de garantir la robustesse et la maintenabilité du code, nous avons écrit des tests unitaires en utilisant le framework `JUnit`. Chaque composant critique de l'application a été testé de manière indépendante, ce qui permet de détecter rapidement les régressions lors des modifications.

Enfin, toutes les fonctions de notre code sont documentées à l'aide de docstrings claires, facilitant ainsi la compréhension et la réutilisation du code par d'autres développeurs.

6 Conclusion

Nous avons choisi de travailler en binôme de façon très collaborative. La majorité du travail a été effectuée lors de séances de travail communes organisées dans notre colocation (ce qui est une chance pour nous deux de pouvoir travailler en projet ensemble), ce qui nous a permis de réfléchir ensemble à la structure globale de l'application, aux choix d'implémentation, et de faire régulièrement des revues de code. Le fait de vivre ensemble a grandement facilité notre collaboration, notamment en dehors des horaires de cours. Une fois l'architecture définie, nous avons réparti les tâches de manière équitable : l'un se chargeait, par exemple, de l'implémentation de l'interface en ligne de commande pendant que l'autre travaillait sur les tests unitaires. Nous avons également alterné à la fin du projet pour la conception des scénarios et la rédaction du rapport. Cette méthode de travail nous a permis de rester alignés tout au long du projet et de bénéficier pleinement des idées de chacun.

Le projet *MyFoodora* nous a permis de mettre en œuvre les principes fondamentaux de la programmation orientée objet et de construire une architecture claire, modulaire et évolutive. Nous avons appliqué le principe « open-closed » en concevant des classes extensibles comme `User` ou `FidelityCard`, et avons respecté une séparation des responsabilités rigoureuse entre les différentes couches de l'application. Toutes les fonctions ont été documentées à l'aide de docstrings, et des tests unitaires JUnit ont été mis en place pour assurer la fiabilité du code.

Néanmoins, certains aspects restent à améliorer. L'interface se limite actuellement à une ligne de commande, ce qui n'est pas optimal pour une utilisation grand public. De plus, la persistance des données n'a pas été complètement développée, limitant le réalisme de l'application dans un contexte de production.

À l'avenir, le projet pourrait évoluer vers une application mobile (Android et iOS), avec une interface graphique plus intuitive et une intégration backend complète (API REST, base de données, géolocalisation en temps réel, etc.). Ces perspectives offriraient une base solide pour transformer *MyFoodora* en une véritable application de livraison utilisable au quotidien.