SophiaTech Eats

Rapport Équipe B - 2023

- **Quality Analyst :** Lucie ANDRES & Karim CHARLEUX
- **Product Owner :** Nina BOULTON
- **Software Architect :** Axel DELLILE
- **DevOps :** Damien STENGEL

Table des matières

Périmètre fonctionnel

1.1. Hypothèses de Travail

- Les commandes sont considérées comme livrées en 10 minutes.
- Une commande est prête également en 10 minutes. (Une évolution possible serait de laisser le restaurant spécifier le temps de préparation).
- Il n'y a pas de fermeture au sein d'une journée pour un restaurant (pas de pause).
- Une fois la commande en cours de préparation, pour éviter le gaspillage de nourriture, il devient impossible de l'annuler.
- Une commande préparée sera récupérable par le livreur même si le restaurant est fermé, toujours afin d'éviter le gaspillage (on considère qu'il a un pass ou une clé pour rentrer quand même).
- Un utilisateur non connecté ne verra pas l'option pour passer une commande.
- Le livreur peut choisir de prendre en charge une commande dès que la commande entre en préparation
- Le livreur doit accepter une commande lui-même
- Présence d'un récapitulatif commande à la fin de la création d'une commande par le client.
- Possibilité de supprimer ou d'ajouter de nouveaux menus à la commande à cette étape.
- Passé l'étape du paiement, une commande ne peut plus être modifiée, seulement annulée.

1.2 Extensions choisies et éléments spécifiques

Nous avons mis en œuvre avec succès l'extension obligatoire [EX1] Diversité des commandes ainsi que [EX2] Ristournes des restaurateurs -.

- [EX1] Diversité des commandes :

Nous avons intégré une gamme diversifiée de types de commandes pour répondre aux besoins variés des utilisateurs. Notre système gère désormais plusieurs formats de commandes :

- Commandes simples (un restaurant, un usager),

- Commandes multiples (plusieurs restaurants, un usager),
- Commandes de groupe (plusieurs restaurants, plusieurs usagers),
- Commandes buffets (un restaurant offrant des buffets, un staff universitaire, un usager destinataire),
- Commandes afterworks (un restaurant avec des menus afterworks, un usager commandeur, et un nombre de participants prévu).

Les commandes buffet et afterwork ne nécessitent ni paiement ni livraison. Nous avons conçu cette fonctionnalité de manière modulaire, en utilisant des templates méthodes, permettant ainsi une adaptation aisée en cas d'exigences métier.

- [EX2] Ristournes des restaurateurs :

On a réussi à implémenter la feature donc lorsqu'un utilisateur fait 10 commandes dans un intervalle de 15 jours, durant les 15 jours suivants, il aura une ristourne de 5% et c'est bien évidemment prolongeable.

1.3 Points non implémentés relativement la spécification et des extensions choisies

Concernant les extensions prévues, l'extension [EX5] Signalement des Usagers, n'a pas été implémentée dans les délais. Nous n'avons pas pu livrer cette fonctionnalité à temps. La principale raison de cette omission est un conflit interne sur les critères d'acceptation de cette US.

En effet, nous voulions implémenter une interdiction de commande après un nombre répété de signalement, sauf que pour certain l'interdiction devait être définitive, alors que pour d'autres elle ne devait durer qu'une semaine avant suppression des signalements.

La pull request est encore ouverte : "[US18] Late user reporting#42 #55"
Et le développement est sur la branche : "D-feature-18"

Actuellement, l'absence de cette extension n'a pas d'impact majeur sur l'avancement du projet, bien que sa mise en place aurait pu améliorer l'expérience utilisateur pour les livreurs.

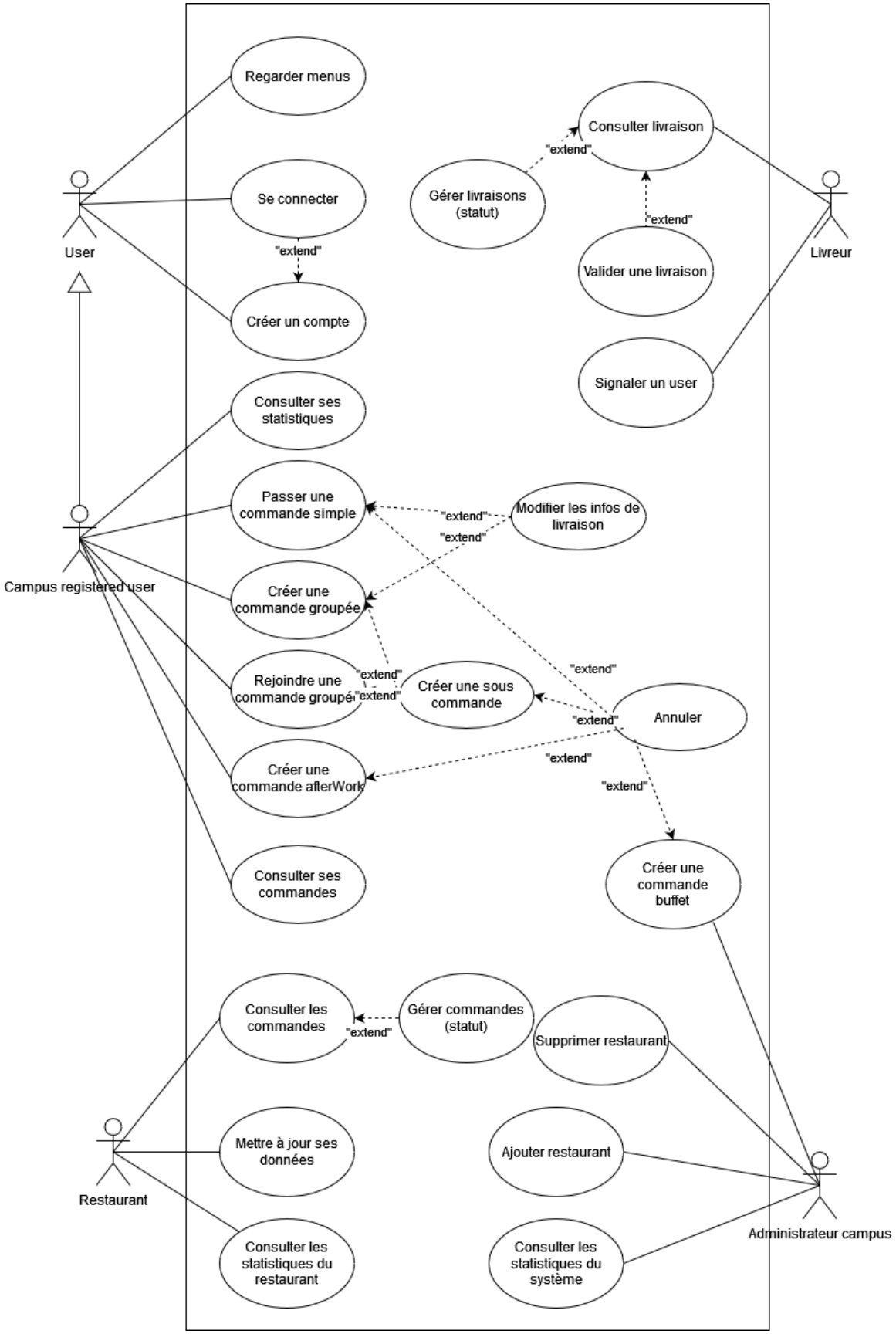
Nous n'avons pas non plus pu fournir des statistiques détaillées, ce qui aurait été appréciable. Nous avons cependant quelques statistiques de base telles que le nombre de commandes par statut, le nombre de restaurants.

Conception

2.1 Glossaire

- **Administrateur** (du campus) : Personne occupant un rôle administratif, responsable de superviser et gérer les activités du campus.
- **API** (Application Programming Interface) : ensemble de protocoles et d'outils permettant différentes applications logicielles pour communiquer et interagir entre elles.
- **Campus :** Zone centrée sur PNS, qui se trouve à 10 minutes à pied du centre (PNS, les IUT, l'INRIA, les Algorithmes, Luciole et St-Philippe).
- **Commande :** demande faite par un utilisateur du campus pour des menus alimentaires spécifiques provenant d'un ou plusieurs restaurants.
- **Utilisateur du campus :** Individu au sein de la communauté du campus, notamment les étudiants, le personnel et les membres du corps enseignant.
- **Livreur :** Utilisateur de l'application chargé de livrer les commandes, il est externe aux restaurants et possède un compte personnel
- **Menu :** Liste de produits alimentaires proposés par un restaurant sur commande
- **Utilisateur :** Usager de l'application n'ayant qu'un accès aux fonctionnalités limité.
- **Utilisateur connecté :** Utilisateur appartenant au campus, possédant un compte auquel il est connecté, et pouvant profiter des fonctionnalités de commande sur l'application.
- **Restaurant :** Désigne le personnel du restaurant qui interagit avec le système.
- **Systèmes de paiement externes :** fait référence aux plateformes de paiement tierces, telles que PayPal et Google Pay, utilisées pour traiter les transactions financières.
- **Temps de préparation :** temps estimé nécessaire à un restaurant pour préparer un menu

2.2 Diagramme de cas d’utilisation et User Stories



- **User**

Regarder menu : [US1] Placing a simple order #1
Se connecter : [US5] Sign in / login for user and addition of constraint for...
Créer un compte : [US5] Sign in / login for user and addition of constraint for...

- **Campus registered user**

Créer une commande groupée : [US3] Create or join a group order...
Rejoindre une commande groupée : [US3] Create or join a group order...
Passer une commande simple : [US1] Placing a simple order #1
Créer une commande afterwork : [US16] After Work Orders#40

- **Restaurant**

Mettre à jour ses données : [US2] Updating menus, [US9] Schedule management...
Consulter les commandes : [US4] Restaurant managing orders #8,
Gérer les commandes : [US4] Restaurant managing orders #8

- **Administrateur campus**

Supprimer restaurants : [US8] Add campus administrators...
Ajouter restaurants : [US8] Add campus administrators...
Consulter les statistiques du systèmes : [US8] Add campus administrators...
Créer une buffet order : [US15] Buffet Orders #39

- **Livreur**

Consulter livraisons : [US7] Add the delivery system#10
Gérer livraisons : [US7] Add the delivery system#10
Valider une livraison : [US7] Add the delivery system#10
Signaler un user : [US18] Late user reporting #42

2.3 Diagramme de classes

Pour le diagramme de classes, vous pouvez voir un diagramme général qui omet les méthodes et certaines classes non essentielles à la compréhension (logger par exemple). Il n'est peut-être pas très lisible, il y a un certain nombre de classes malheureusement.

Ce qu'il faut retenir de la structure, c'est une entrée dans le modèle objet via la classe STEats. Qui possède plusieurs registry servant d'intermédiaire pour chaque type d'utilisateur afin de réaliser les fonctionnalités dont il a besoin.

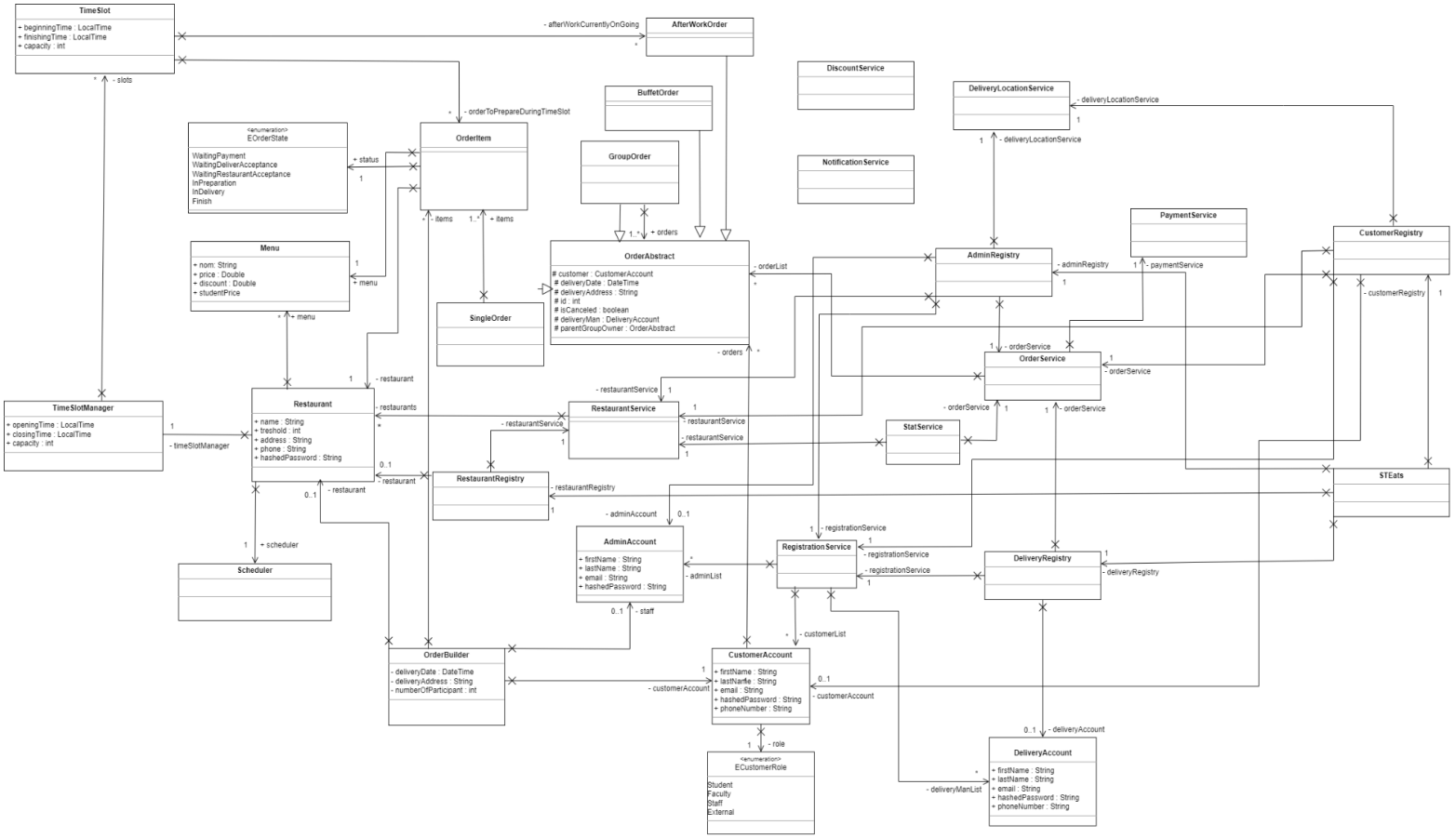


Image 2 : Diagramme de classe général

Vous pouvez également voir ci-dessous une version plus approfondie (et lisible) sur l'un des packages, celui des order. Pour voir plus en détail le fonctionnement de cette partie du projet qui est à mon sens un peu plus complexe.

Plus de détails sont apportés sur la partie parlant des design pattern, mais ici on éviter une certaine quantité de duplication de code bien que ça ne soit pas parfait. On observe par exemple que certaines méthodes sont développées en double tel que calculatePriceWithCreditDeduction dans BuffetOrder et SingleOrder. Celà étant essentiellement dû au manque de temps.

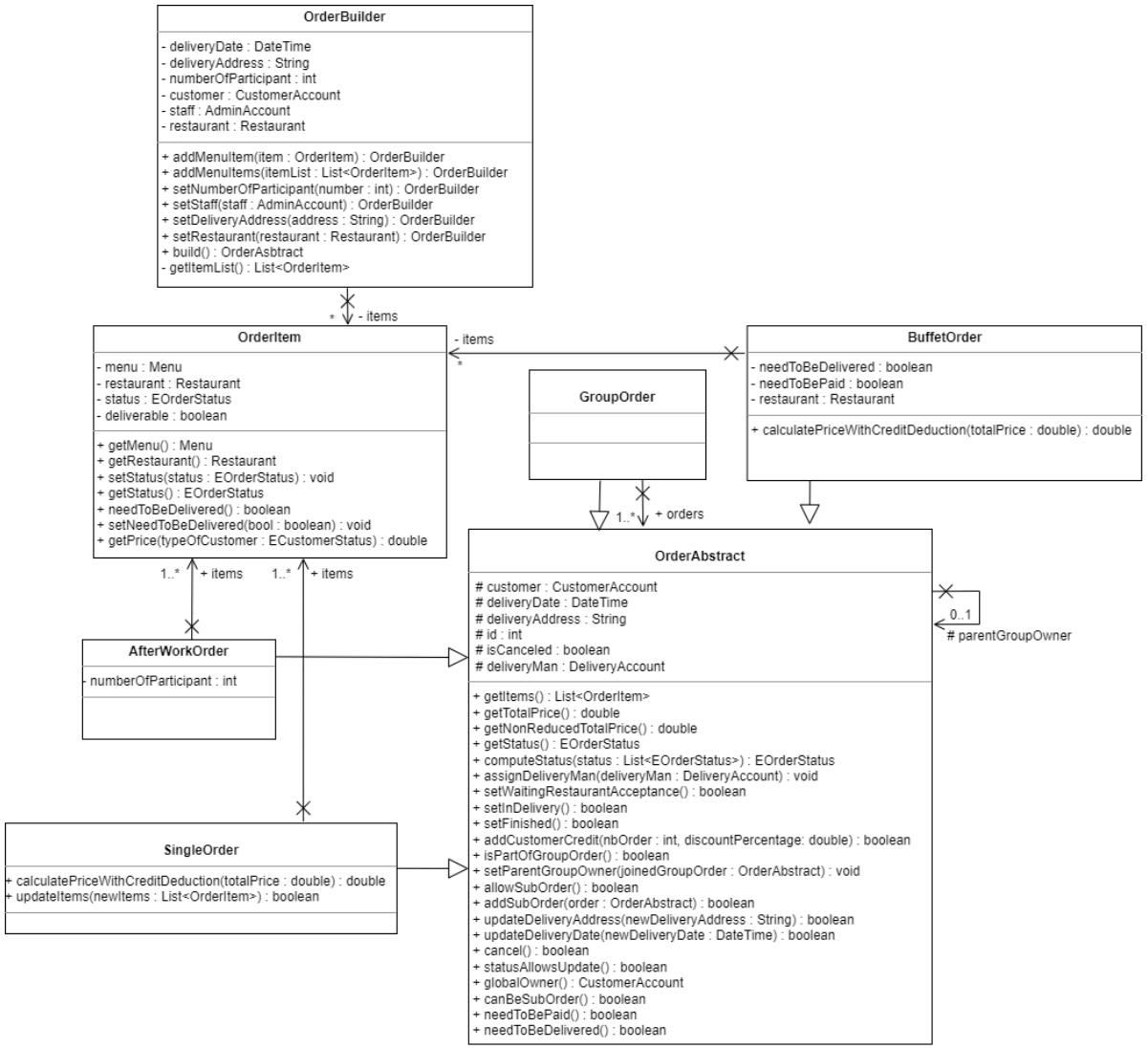
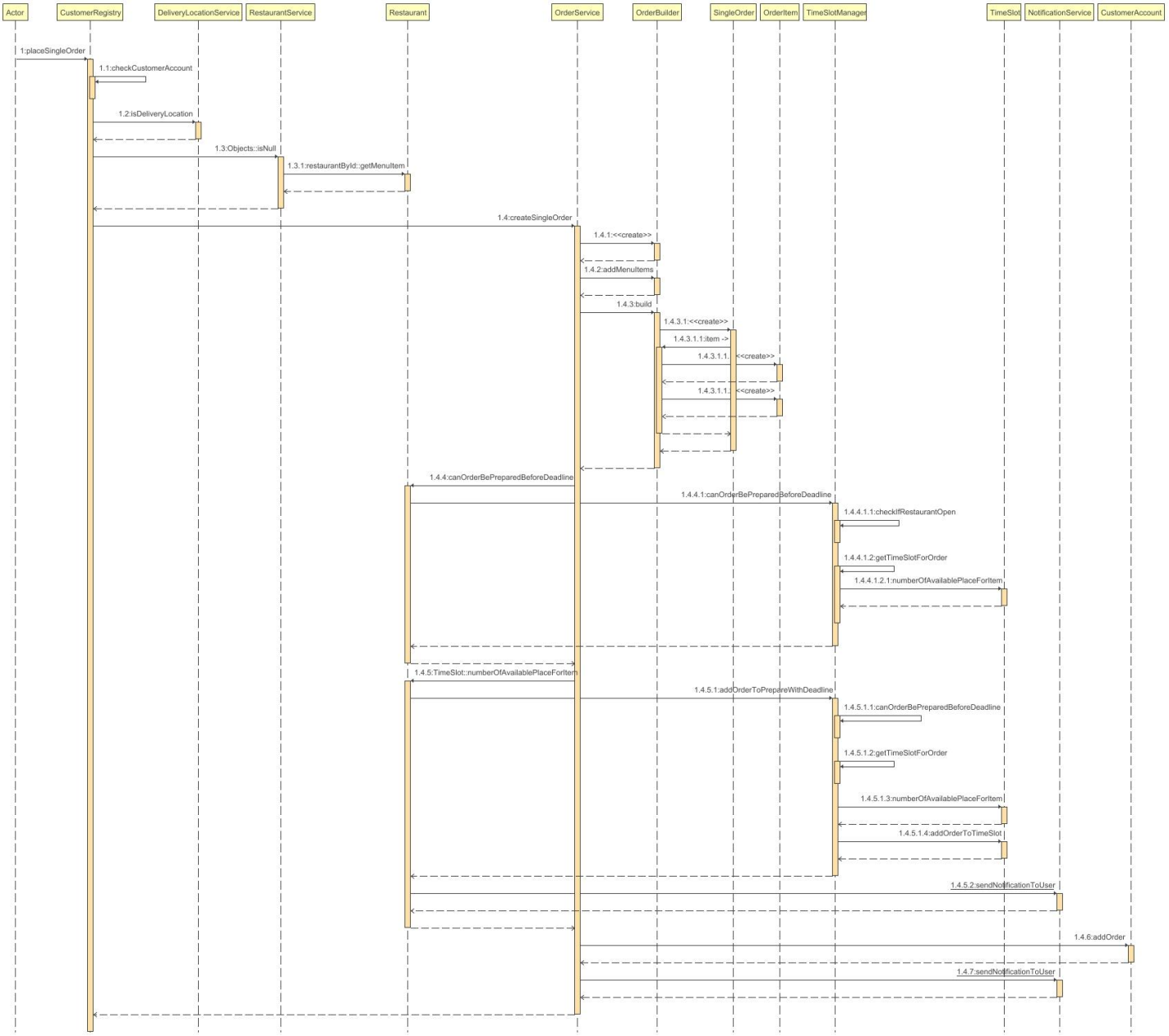


Image 3 : Diagramme de classe pour le package order

N'étant pas réellement pertinent à mon sens je n'ai pas inclus d'autres diagrammes de classes.

2.4 Un diagramme de séquence



Design Patterns appliqués ou pas

3.1. DPs en détail

Plusieurs design pattern ont été implémentés dans le projet. Pour chacun d'entre eux, je vous renverrai vers les diagrammes de classes de la section 2.3.

Pattern Builder : Ici la classe OrderBuilder (package order) permet d'instancier selon les méthodes appelées avant d'appeler la méthode build un type de classe héritant d'OrderAbstract différente. Celà apporte à la fois de la généricité dans l'instantiation des order mais également un peu plus de légèreté, car on évite l'appel à des constructeurs à rallonge tout en simplifiant l'utilisation côté client.

Pattern Composite : C'est un pattern que l'on a implémenté dès la V1 (package order) qui malheureusement est devenu un peu moins efficace depuis la V2 à cause de la feature sur la diversité des orders. Cela nous facilite la gestion des commandes de groupes par exemple. Cependant, l'implémentation de la V2 sans duplication de code dû à l'utilisation de ce pattern fut compliquée (je détaille ce propos lors de la conclusion sur l'architecte).

Pattern template method : Pour cette partie, toujours dans le package order pour discerner si une commande a besoin d'être livrée/payée, on a utilisé des templates méthodes afin de simplifier les vérifications et de par conséquent fusionner des méthodes de orderService et de réduire la duplication de code. C'est la solution que l'on a trouvée couplée au pattern composite qui nous a permis de faire la v2.

3.2. Autres DPs

Bien que l'on ait déjà implémenté certains patterns, dû principalement au manque de temps (et parfois à une utilisation qui complexifierait davantage le code pour très peu de bénéfice) nous avons dû faire l'impasse sur d'autres design pattern.

Pattern state : Ici, c'est le manque de temps qui a joué, l'idée aurait été de simplifier la manipulation des commandes en créant différent state pour remplacer l'enum EOrderStatus.

Pattern Observer : Alors ici, c'était un manque de conviction en l'efficacité du pattern. L'idée ici aurait été de rendre les commandes *observables* et de rendre les classes associées aux restaurants, livreurs et client *observer* afin d'automatiser le traitement des commandes.

D'autres patterns seraient sûrement pertinents pour ce projet, cependant nous n'avons pas porté notre attention sur les autres potentielles améliorations qu'ils nous auraient apportées.

Qualité des codes et gestion de projets

- **Nature et Couverture :**

Notre stratégie de tests a été exhaustive, pour chaque user story, on s'est appuyé sur des tests d'intégration avec le framework Cucumber et des tests unitaires avec JUnit et Mockito.

- Les tests d'intégration avec Cucumber permettent de couvrir un maximum de scénarios utilisateur de l'application, comme la gestion de tous les types de commandes, la connexion des différents types d'utilisateurs, ou la gestion des restaurants. Ces tests ont été effectués pour valider chaque fonctionnalité du service demandé.
- Les tests unitaires avec JUnit ont assuré la fiabilité des composants individuels du code. Les tests unitaires JUnit avec Mockito, nous a permis de bien isoler la partie du code qu'on teste dans le but de ne pas dépendre de systèmes externes ou d'autres classes, ainsi, il est possible de détecter de façon plus locale les erreurs possibles de notre système.

- **Qualité du Code et Points d'Amélioration :**

La robustesse des tests unitaires et d'intégration a été un point fort, assurant une base de code fiable. Nous avons mis en place des contrôles stricts pour garantir la robustesse des entrées des méthodes sensibles, on minimise ainsi les erreurs utilisateur potentielles. Évidemment, cela peut restreindre légèrement l'expérience utilisateur, mais ces mesures assurent un fonctionnement fiable du système soutenu par des mécanismes de gestion d'erreurs efficaces. En effet, on a bien pris soin de décrire au plus clair possible la cause de l'erreur avec la bonne exception pour guider au mieux l'utilisateur qui l'utilise.

Afin d'améliorer nos tests, on pourrait isoler un peu plus nos tests unitaires, je m'explique. Certains de nos tests sont très larges et test à la fois les cas d'erreurs et les cas passant. Cela complexifie parfois la recherche d'erreurs, mais par manque de temps, on a jugé cela moins crucial.

On pourrait également faire un scénario transversal qui couvrirait un maximum de fonctionnalités afin de fournir une meilleure démo, mais le temps nous a manqué pour cela.

- **Automatisation des branches :**

Les Github actions nous ont offert une automatisation des tests JUnit et Cucumber sur toutes les pull request allant vers develop ou main.

On a également profité d'un serveur de Maxime Billy un autre étudiant de la promotion afin d'avoir des scans SonarQube. Couplé à Jacoco celà nous offre des vérifications bloquante si elle échoue, et visible sur les PR depuis github directement, avec différent palier d'exigences :

- 80% de couverture de tests sur les nouvelles lignes de code.
- Un faible taux de duplication de code induit par le nouveau code.
- Un faible nombre de code smell et aucun code smell high ou critical.

Ces différentes automatisations sont essentielles afin d'assurer la production d'un code de qualité tout en assurant sa non-régression.

Rétrospective et Auto-évaluation

- **Quality Analyst : Lucie ANDRES & Karim CHARLEUX**

  En tant qu'analyste qualité, nous avons beaucoup appris en travaillant sur ce projet. Notre rôle a consisté à réaliser des tests d'intégration avec Cucumber et des tests unitaires utilisant JUnit et Mockito. Cela a permis d'assurer la couverture des scénarios utilisateurs. Et cela a aidé à identifier et isoler efficacement les erreurs, contribuant ainsi à améliorer la robustesse et la qualité du code.

  On s'est par ailleurs chargé de bien vérifier que chaque pull request est bien couverte grâce aux tests avec au moins un test cucumber qui marche couvrant le scénario associé à l'user story ainsi que des tests unitaires qui passent également au vert.

  Par ailleurs, nous nous sommes directement chargés de la rédaction des scénarios et des tests afin d'en augmenter l'étendue.

- **Product Owner : Nina BOULTON**

  Pour notre Product Owner l'important était d'assurer le lien entre les besoins des utilisateurs, la vision du produit et le travail de l'équipe de développement. Il s'agissait de comprendre les besoins et de créer une vision claire pour le développement, en assurant la cohérence pour garantir l'efficacité du projet.

  Pour cela, il traduit les besoins en user stories claires et compréhensibles pour l'équipe de développement, définit des milestones pour marquer les étapes importantes du projet, et utilise les boards GitHub pour suivre et organiser le travail de l'équipe.

  En utilisant ces outils, il facilite la communication, la planification, et s'assure que le développement se concentre sur la réalisation des objectifs du produit et la satisfaction des utilisateurs.

- **Software Architect : Axel DELILLE**

  Côté architecture, il y a plusieurs points à aborder. Tout d'abord, la rédaction des diagrammes de classes a entièrement été prise en charge par ses soins.

  Ensuite, avant chaque prise de ticket, il a fait une petite session d'échange afin d'expliciter les besoins en termes technique, tout en expliquant les points clés nécessaires au bon déroulement du ticket.

  Par ailleurs, il s'est rendu disponible dans le but d'assister ses pairs lors des développements posant problèmes.

Il a géré la majorité des pulls request pour la partie implémentation de code, couplé à la vérification des QA celà permettait d'avoir une vue d'ensemble des problèmes possibles et des points à rectifier.

Ensuite, il y a deux moments importants à revoir. Le premier étant le premier gros refactor qui a eu pour but :

- De retirer le rôle de god class de STEats au profit de la création des divers registry.
- Un clean du code du pattern composite (OrderAbstract, SingleOrder, GroupOrder)
- Une refonte du workflow de passage des commandes via le registry pour permettre de passer une commande simple avec plusieurs éléments d'un coup.

Le second a eu lieu peu après le début de la V2 afin d'étudier les solutions pour implémenter BuffetOrder et AfterWorkOrder, en effet avant le choix de la template method, une branche a été créé dans le but de comparer les diverses solutions, la première était de faire :

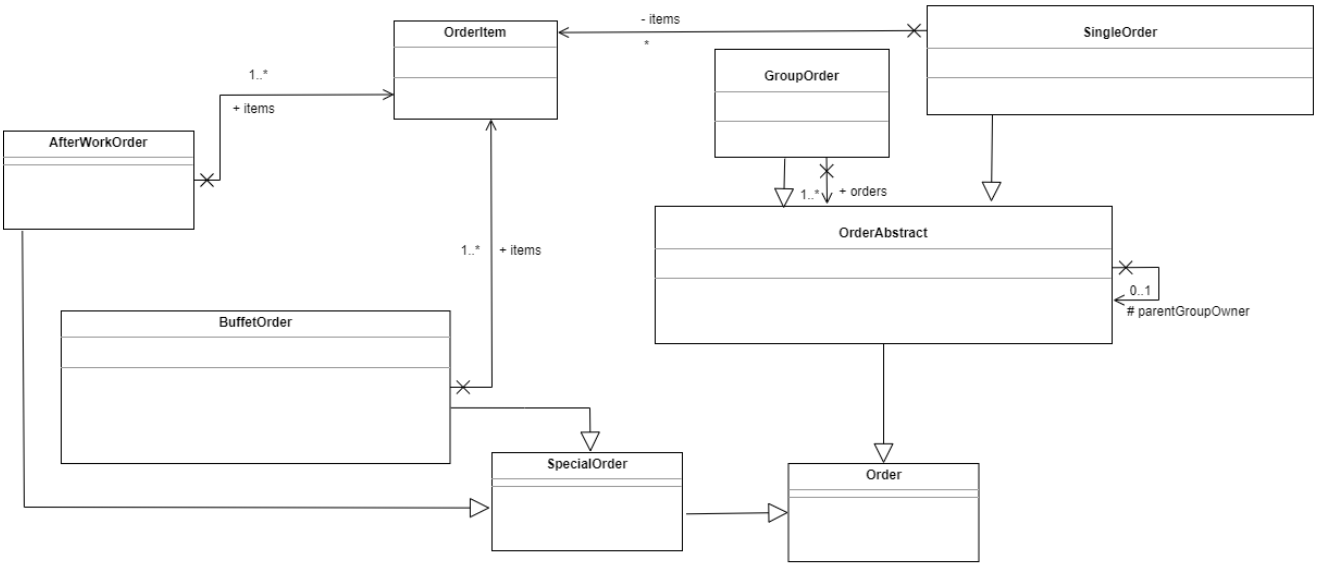


Image 5 : Ébauche diversité order, diagramme de classe

Sauf que faire cela résulterait en de la duplication de code dans la totalité du modèle objet sans apport particulier. C'est à ce moment qu'est apparue l'idée de la template method.

- **DevOps : Damien STENGEL**

  Côté DevOps ce fût dans un premier temps un travail de mise en place du github afin que les branches soit sécurisé et que la forme des commit soit commune à l'ensemble du groupe. Par la suite quelque changement sur la CI/CD déjà présente a été faite afin d'avoir une branche develop la plus saine possible. Ensuite, grâce à la machine d'un étudiant de la promotion nous avons pu implémenter SonarQube dans notre projet afin de faciliter le travail des QA et permettre d'avoir une vision d'ensemble du projet, cela nous a permis d'améliorer considérablement la qualité de code du projet.

- **Bilan de fonctionnement de l'équipe**

  Notre équipe a su surmonter les difficultés, le dialogue n'a pas toujours été simple, les tickets n'étaient pas toujours fait en temps et en heure. Une grosse charge de travail a été effectuée par une partie de l'équipe. Cependant, après quelques discussions, cela s'est rééquilibré légèrement au début de la V2.

  Une différence de compréhension s'est faite sentir sur l'implémentation de certaines US, beaucoup de peer programming a été nécessaire à certains moments. Cependant, le groupe est en bonne voie de progression, chacun a appris lors de ce projet en tant que développeur, mais également au travers du rôle qui lui a été assigné.

  On ne peut pas dire que c'est parfait, c'est faux. Mais je pense qu'étant étudiants, une marge de progression est tout à fait normale et montre la bonne volonté de chaque acteur du groupe.

- **Auto-évaluation**

  Lucie ANDRES : 90 /100
  Nina BOULTON : 90 /100
  Karim CHARLEUX : 100 /100
  Axel DELILLE : 100 /100
  Damien STENGEL : 90 /100