POLYTECH NICE SOPHIA

UNIVERSITÉ CÔTE D'AZUR

Rapport de conception logicielle

Equipe H

Sagesse ADABADJI : PO

Sara TAOUFIQ : SA

Jean Paul ASSIMPAH : QA

Selom ADZAHO : QA

Abenezer YIFRU : OPS

2024/2025

Sommaire

1. PÉRIMÈTRE FONCTIONNEL : HYPOTHÈSES, LIMITES, EXTENSIONS, POINTS FORTS ET POINTS FAIBLES. (2 pages Max)

1.1 Hypothèse de travail

Hypothèse 1 : Seuls les restaurants disponibles sont proposés au client. Un temps de préparation moyen de quinze minutes est prévu pour une commande dans un restaurant, ce qui sert pour la gestion de la capacité.

Hypothèse 2 : L'affichage des heures de livraisons possibles se fait en fonction des horaires du restaurant.

Hypothèse 3 : Les heures de livraison ont un intervalle de 15 minutes qui correspond au temps de préparation moyen d'une commande

Hypothèse 4 : Lorsque l'utilisateur sélectionne une heure de livraison, le temps de préparation total de sa commande doit correspondre à l'heure de livraison choisie.

Hypothèse 5 : L'utilisateur peut changer l'heure de livraison mais dans la limite de celles disponibles

Hypothèse 6 : L'utilisateur choisit un moyen de paiement parmi ceux prédéfinis

1.2 Limites identifiées.

Lorsqu'une personne passe une commande en ayant choisi une date de livraison au préalable, les menu Items ont donc pu être disponible en fonction de la capacité du restaurant mais s'il attend l'heure de livraison avant de valider sa commande...

1.3 Stratégies choisies et éléments spécifiques.

Notre stratégie repose sur une approche proactive qui nous permet d'éviter les situations imprévues ou difficiles à gérer, telles que des commandes impossibles à honorer par le restaurant ou des livraisons non réalisables.

Pour notre strategy pattern, nous avons distingué deux familles de stratégies : celle qui influence le solde du client, comme la **StatusStrategy** et la **PercentageStrategy**, et celle qui agit directement sur la commande elle-même, en modifiant la quantité d'items, comme la **Buy1Get1**. La différence entre ces stratégies devient particulièrement visible au moment de leur application, notamment lors de la confirmation de commande et lors du paiement. Sans gestion différenciée, des problèmes tels qu'un solde infini pour le client pourrait subvenir.

Gestion de la Capacité d'un Restaurant

1. Détermination de la Capacité du Restaurant

    a. **Capacité basée sur le personnel et le temps de préparation** : La capacité est calculée en fonction du nombre de membres du personnel disponibles dans un créneau et du temps de préparation moyen des items de menu. Cela permet de déterminer le nombre optimal de commandes gérables dans les délais.
    b. **Disponibilité des restaurants** : En fonction de la capacité calculée, seuls les restaurants capables de répondre à la demande apparaissent comme disponibles pour la période choisie, évitant ainsi les surcharges.

2. Affichage des Horaires de Livraison et Menu Items

    a. **Horaires de livraison** : Les créneaux disponibles sont affichés selon les horaires d'ouverture et la capacité restante de chaque restaurant.
    b. **Items de menu** : Les articles disponibles sont ajustés en fonction de l'heure de livraison choisie.

3. Gestion des Commandes en Cours

    a. **Liste temporaire des commandes PENDING** : Une commande avec statut PENDING est créée lorsqu'un utilisateur sélectionne un restaurant, une adresse de livraison et un créneau horaire. Cette liste temporaire permet de suivre les créneaux horaires déjà remplis.
    b. Les commandes PENDING sont supprimées une fois validées et payées.
    c. *Amélioration future :* Suppression automatique des commandes PENDING non validées après un certain délai.

1. Contrôle Dynamique des Temps de Préparation et des Créneaux de Livraison

    d. **Temps de préparation contrôlé** : Le temps de préparation pour les items choisis doit être compatible avec l'heure de livraison, assurant un service dans les délais.
    e. **Mise à jour des créneaux** : Si l'utilisateur modifie ses items ou quantités et que le temps de préparation ne permet plus de respecter le créneau sélectionné, une nouvelle heure de livraison compatible est proposée.

4. Mise à Jour de la Capacité

    a. **Ajustement automatique après validation et paiement** : La capacité est mise à jour pour refléter la charge de travail actualisée. Le statut de la commande passe de PENDING à CONFIRMED, et elle est retirée de la liste temporaire. La capacité tient compte du temps total de préparation et du nombre d'items.

Limites de l'Approche pour la gestion de la capacité

1. **Dépendance à la Prévision**
    a. **Incertitude de la Demande** : La capacité d'un restaurant repose sur des prévisions de commandes, mais la demande réelle peut varier considérablement, rendant la prévision des besoins en personnel difficile.
2. **Complexité Logistique**
    a. **Gestion des Commandes Concurrentes** : La gestion en temps réel de la capacité est compliquée lorsque plusieurs utilisateurs passent des commandes simultanément, surtout si les informations de disponibilité ne sont pas mises à jour instantanément.
    b. **Retards de Livraison** : Des temps de préparation estimés mal calculés peuvent provoquer des retards de livraison, impactant l'expérience client.
3. **Inflexibilité des Horaires**
    a. **Limites des Heures de Livraison** : L'affichage des heures de livraison disponibles peut restreindre la flexibilité des utilisateurs si les horaires ne correspondent pas à leurs besoins.
    b. **Adaptation aux Changements d'Horaires** : Modifier l'heure de livraison après la sélection des items peut rendre difficile le réajustement des temps de préparation et de livraison.

2. CONCEPTION UML

2.1 Glossaire

Internet user : Désigne l'ensemble des utilisateurs non enregistrés et enregistrés qui utilisent notre application.

Menu : Liste des plats proposés par un restaurant, comprenant le prix de chaque plat.

Restaurant Manager : Utilisateur du système chargé de gérer les commandes effectuées au sein d'un restaurant, et de mettre à jour les horaires et le nombre de staffs du restaurant.

Discount : Offre proposée par le restaurant, entre réduction et Menu Item offert.

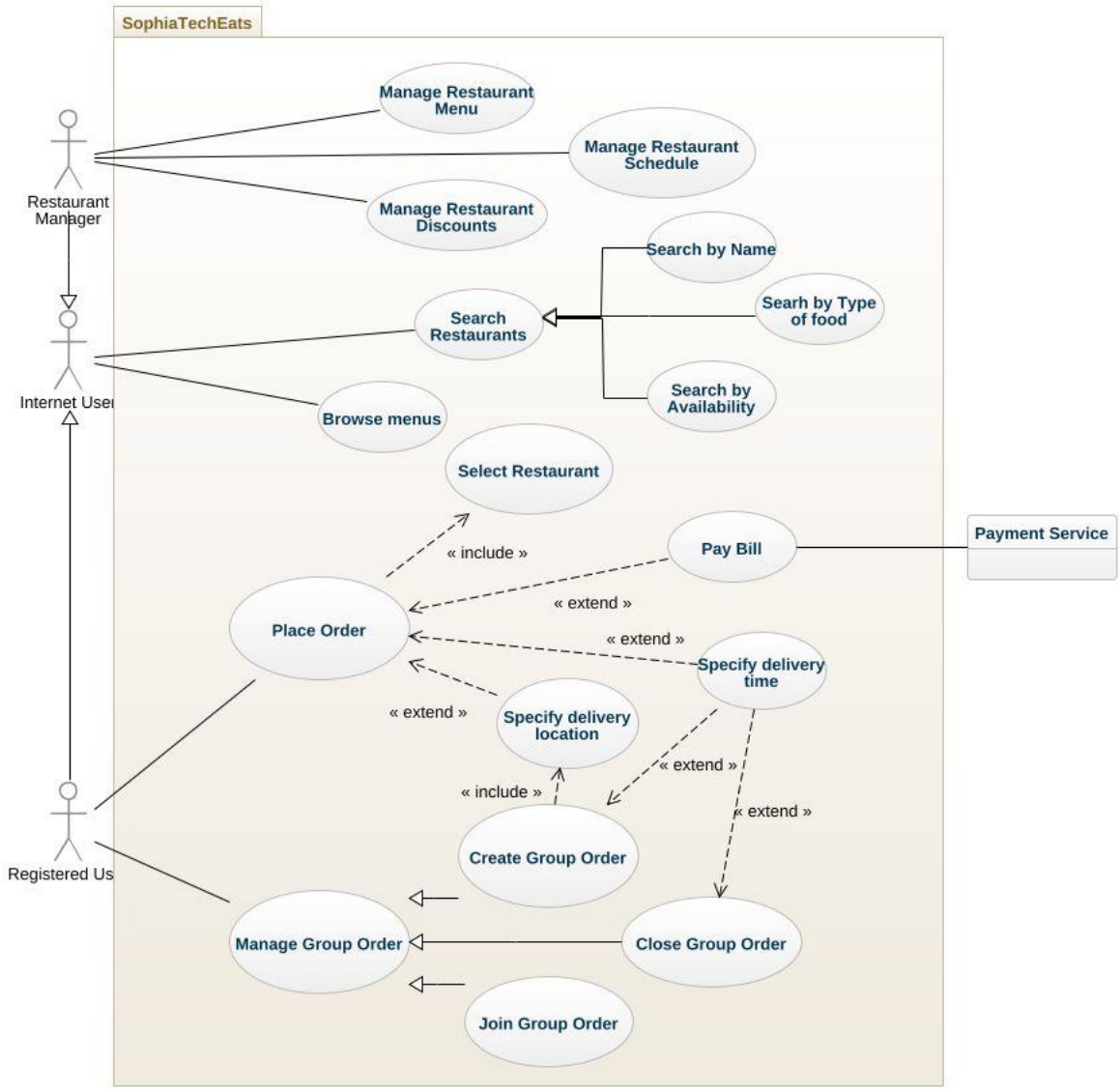
Schedule : Ensemble des créneaux horaires définis pour les commandes et la gestion d'employés travaillant dans un créneau bien défini.

Group order : Un ensemble de commandes individuelles passées par plusieurs personnes, regroupées dans une seule commande, et livrée par un seul livreur.

Capacité d'un restaurant : Nombre de commandes pouvant être prises en charge dans un créneau horaire donné.

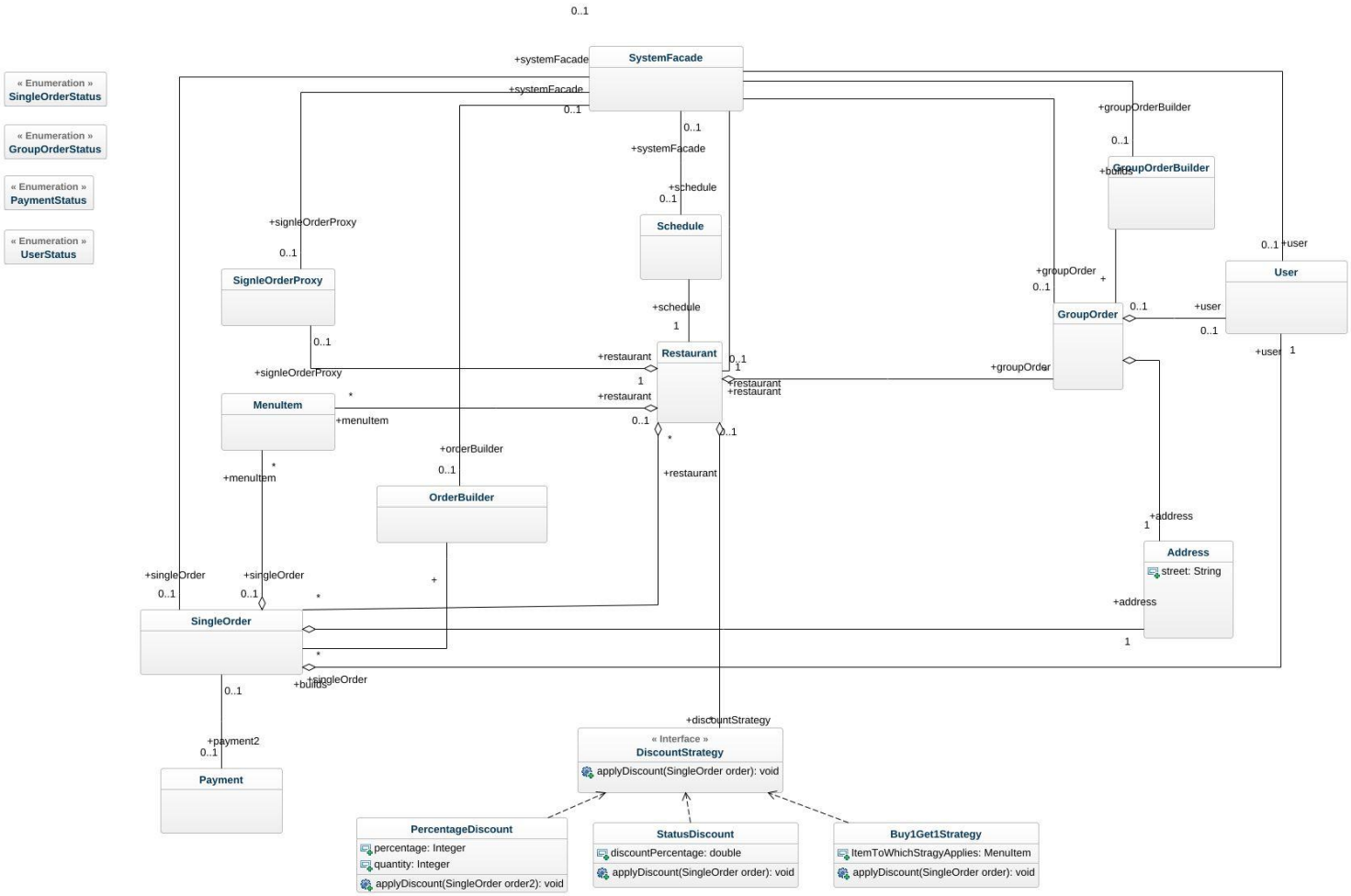
Order Item: C'est le menu item et quantité choisie par l'utilisateur pour sa commande

2.2 Digramme de cas d'utilisation

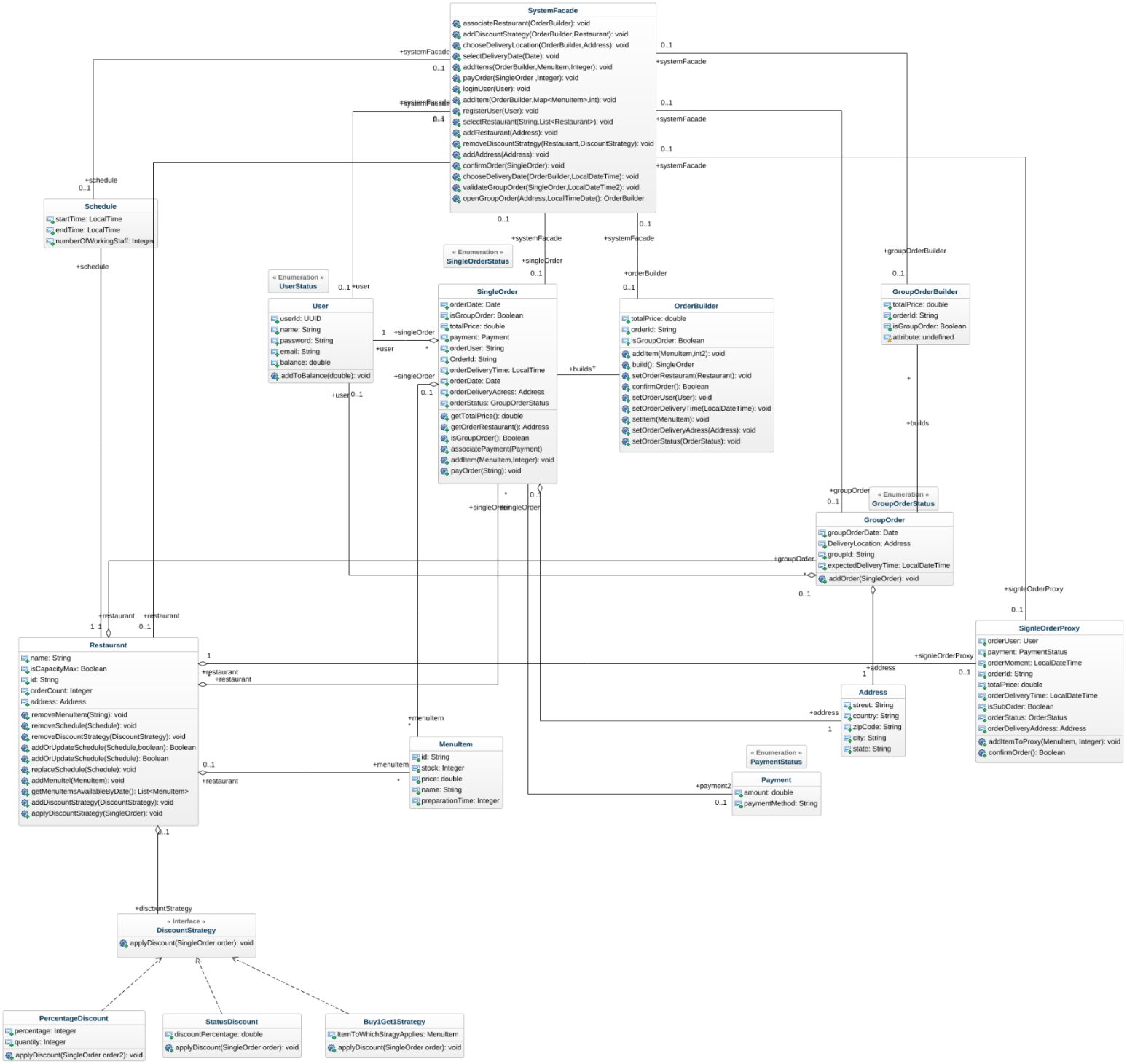


2.3 Digramme de classe

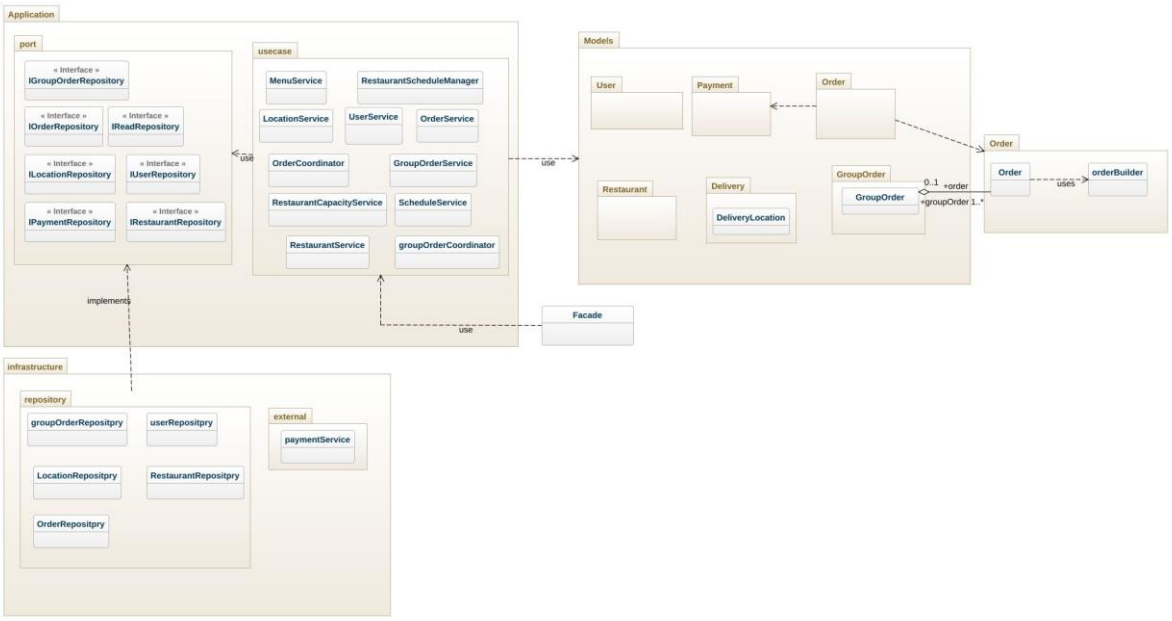
2.3.1-Vue simplifiée des modéles de notre diagramme de classe



2.3.2-Vue détaillée des modèles du diagramme de classe.



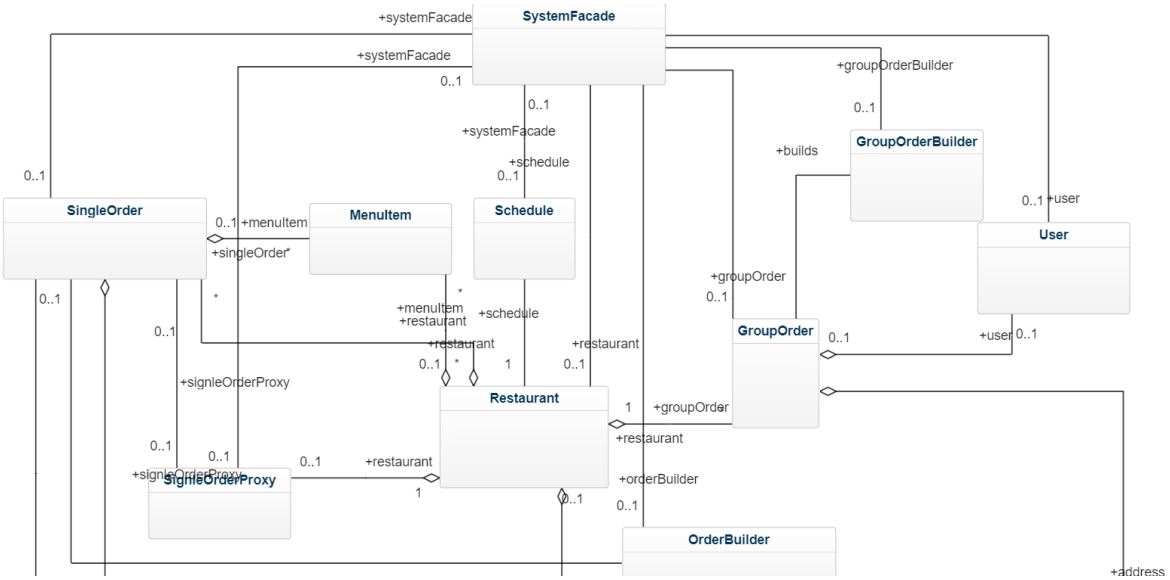
2.3.4-Vue de diagramme de classe sous forme de package.



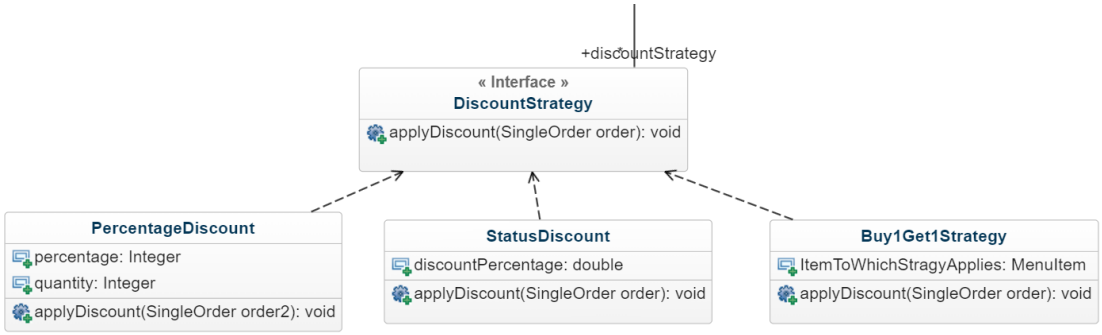
2.4 Design pattern

2.4.1-Façade :

On savait qu'il nous fallait une **façade** dans notre projet. Elle est passée par plusieurs évolutions. Au départ, elle contenait des managers comme attributs, puis nous sommes passés aux entités elles-mêmes, pour finalement opter pour le passage des entités en paramètres.

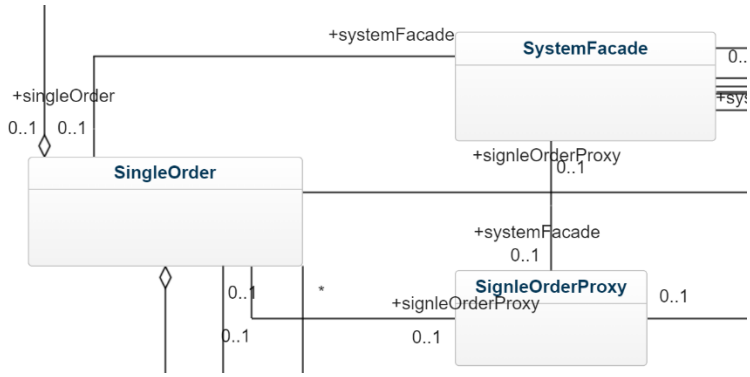


2.4.2-Strategy:



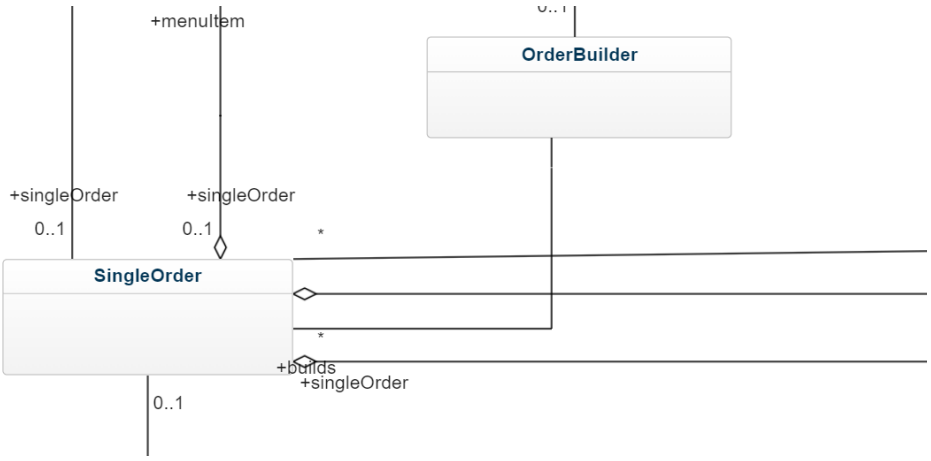
2.4.3-Proxy :

Nous avons initialement envisagé d'utiliser un proxy pour centraliser les contrôles liés à la création des commandes. Cependant, après avoir révisé notre architecture, nous avons constaté que ces contrôles étaient déjà efficacement gérés par plusieurs services distincts. Le proxy introduisait donc une redondance et une duplication de logique inutiles dans le code. En conséquence, nous avons choisi de le retirer pour simplifier l'architecture et améliorer la maintenabilité du système.

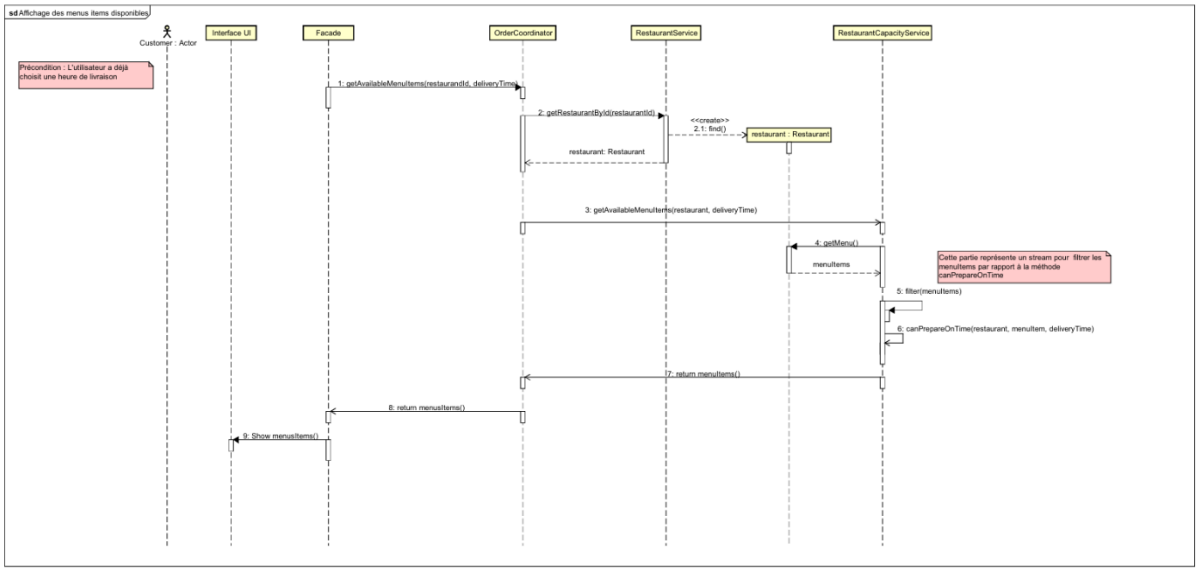


2.4.4-Builder :

Après avoir hésité entre le **Builder,** le **Composite** et le **Factory,** nous avons choisi le Builder, qui s’adaptait mieux à notre refactoring et permettait une construction progressive des commandes.

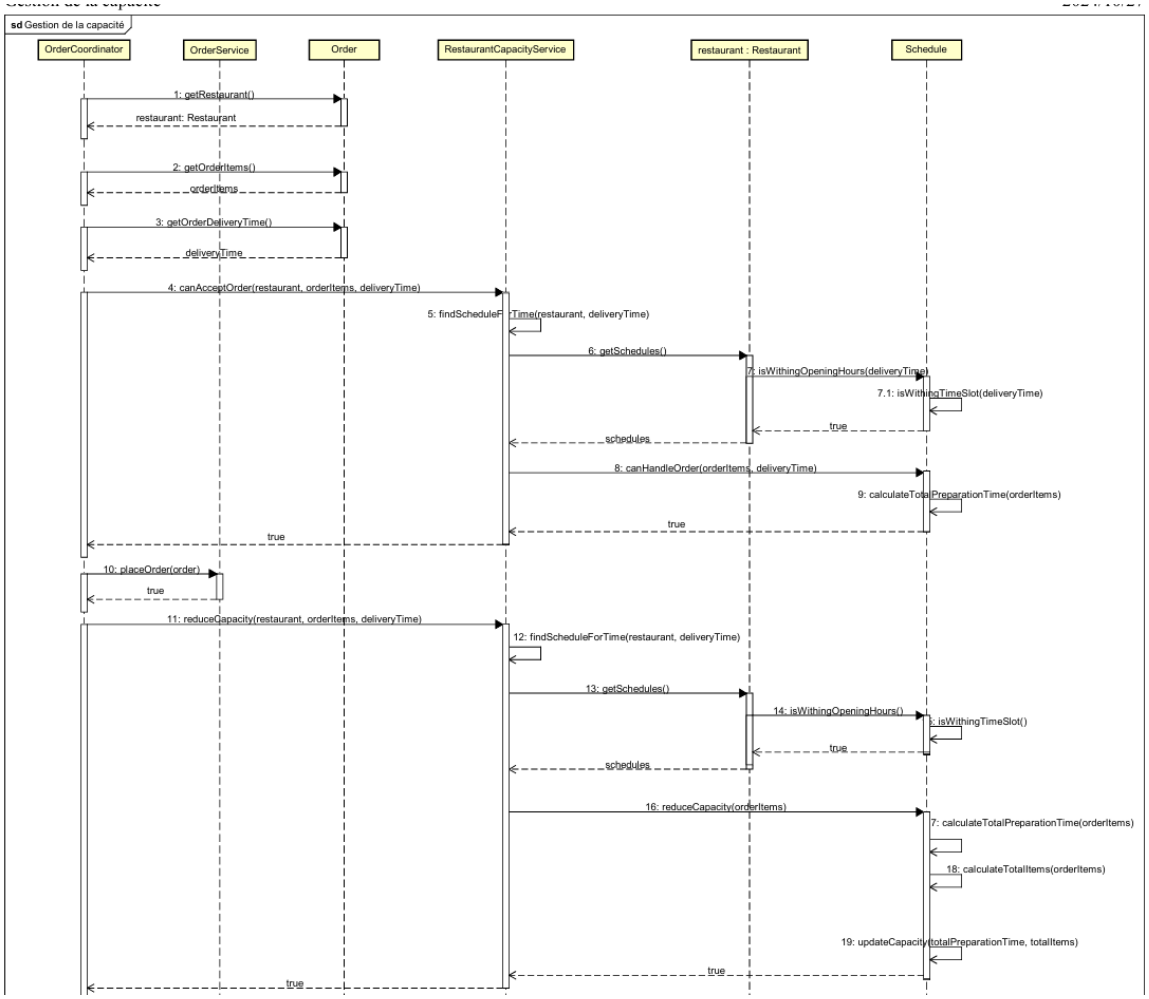


2.5 Diagramme de séquence (Affichage des menus items disponibles)



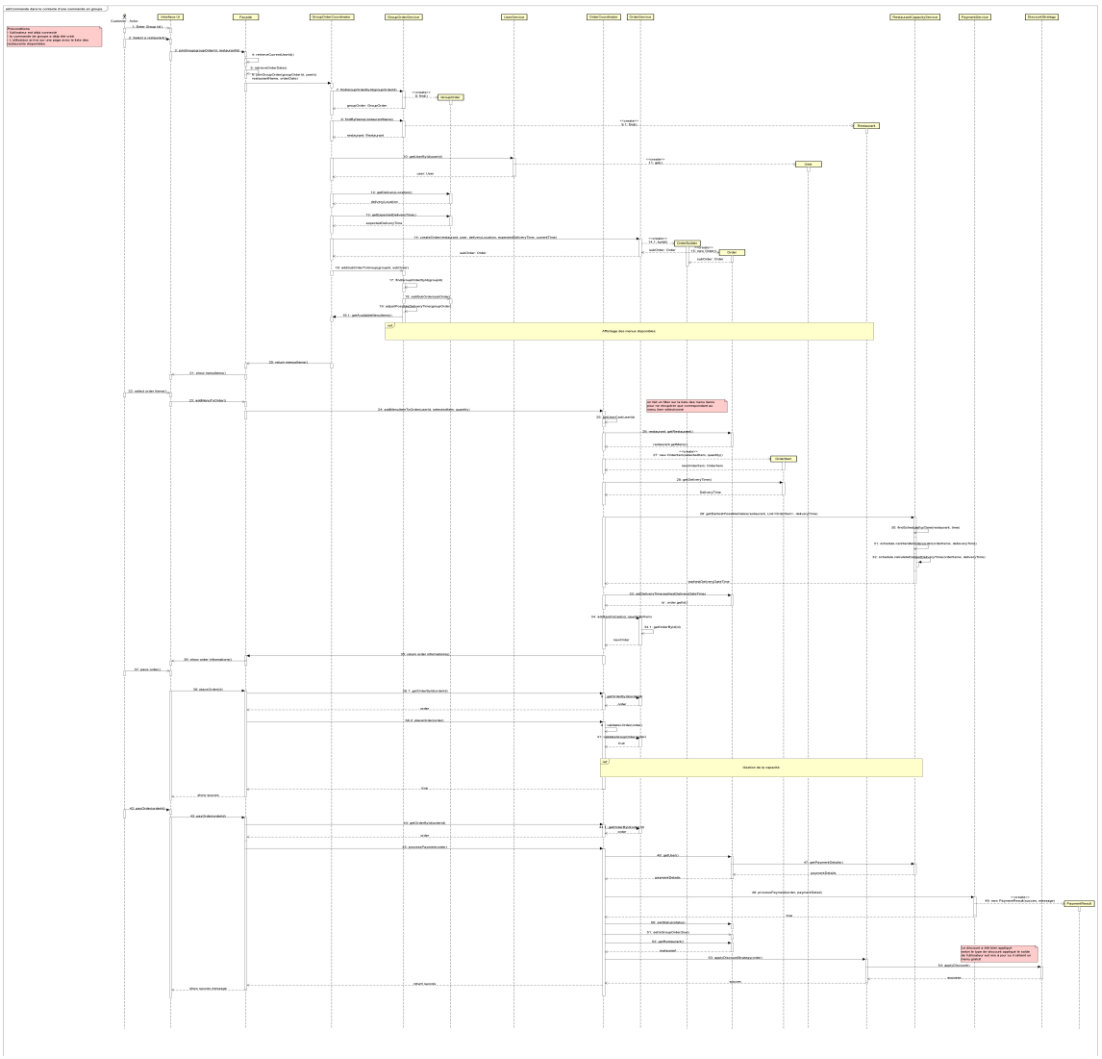
2.6 Diagramme de séquence (Gestion de la capacité)

Notre gestion de capacité intervient à plusieurs étapes du processus de commande : depuis le choix du restaurant (seuls les restaurants disponibles et ayant une capacité suffisante sont affichés) jusqu'à la validation de la commande. Dans le diagramme, nous avons cependant modélisé uniquement la gestion de la capacité lors de la validation de la commande. Nous partons ainsi du principe que le restaurant sélectionné par l'utilisateur n'a pas encore atteint sa limite de commandes pour le créneau horaire choisi.

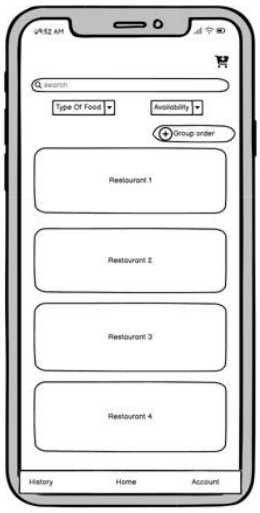


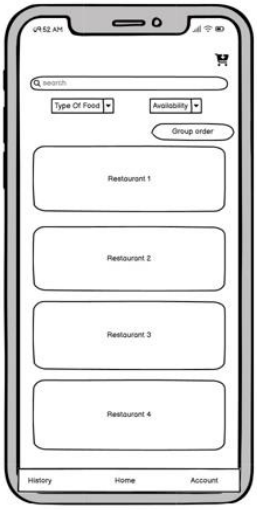
2.7 Diagramme de séquence (Commande individuelle dans le contexte d'un groupe de commandes)

<u>Notre diagramme de séquence n'étant pas assez visible dans le rendu, vous trouverez une copie pdf dans le dossier doc qui sera joint au rendu</u>



3. MAQUETTE





4. QUALITE DU CODE ET GESTION DE PROJET .

4.1 Types de tests, leurs qualités et leur couverture.

Dans le projet, deux types de tests principaux ont été utilisés pour assurer la qualité et la fiabilité du système :

- **Tests Cucumber :** ils ont été utilisés pour valider les fonctionnalités d'après les exigences des utilisateurs, en se basant sur les **critères d'acceptation** définis dans les histoires utilisateurs. Ils ont permis de garantir que le comportement du système correspond aux attentes fonctionnelles.
    - **Exemples et scénarios** : Chaque fonctionnalité est testée à l'aide de plusieurs scénarios représentatifs, incluant des **exemples** spécifiques, c'est-à-dire des ensembles de données qui alimentent les tests Cucumber pour s'assurer que l'application fonctionne correctement avec différentes entrées.
    - **Gestion des cas d'erreur** : Pour les fonctionnalités complexes (comme la gestion des commandes), les scénarios sont divisés pour tester les comportements normaux, ainsi que les **cas d'erreurs spécifiques**.
    - **Couverture** : Ils couvrent toutes les principales fonctionnalités de l'application, garantissant que celles-ci répondent aux besoins des utilisateurs.

- **Tests unitaires JUnit :** ils sont utilisés pour valider le comportement **interne du code** en testant chaque méthode ou classe de manière isolée.
    - **Portée** : ils vérifient que chaque méthode retourne le bon résultat pour différentes entrées, incluant des cas limites.
    - **Couverture** : Ils permettent de tester toutes les parties essentielles du code et de prévenir l'apparition de nouveaux bogues lors des modifications.

Méthodologie de test :

Pour chaque nouvelle fonctionnalité, des tests Cucumber et JUnit sont écrits. Cela permet de valider le fonctionnement à la fois du point de vue utilisateur et au niveau du code.

- **Prévention des régressions** : Grâce à l'intégration continue, les tests sont automatiquement exécutés à chaque modification du code, ce qui permet de détecter rapidement les erreurs et d'éviter les régressions.

4.2 Avis sur la qualité du code

La qualité du code dans ce projet repose sur plusieurs points essentiels :

- **Design patterns :** L'utilisation de patterns comme Façade a permis de structurer le code de manière modulaire et efficace, rendant le projet maintenable et évolutif.

- **Lisibilité :** Le code est bien écrit avec des noms clairs pour les variables, méthodes et classes, ce qui facilite la collaboration et la compréhension.
- **Contrôles robustes :** Des mécanismes de gestion d'erreurs et de validation des données, renforcés par des tests unitaires et d'intégration, garantissent la fiabilité du système.
- **Modularité :** La séparation claire des fonctionnalités dans des classes distinctes permet une grande modularité, facilitant ainsi les ajouts ou modifications de fonctionnalités sans impacter l'ensemble du système.

Enfin, bien que certaines améliorations soient possibles, comme la gestion plus rigoureuse des messages d'erreur ou la réduction de duplications de code, l'ensemble du projet montre une solide maîtrise des bonnes pratiques de développement. Ces améliorations, si elles sont prises en compte pourraient encore améliorer la qualité déjà satisfaisante du code.

4.3 Gestion de projet

Dès le début, pour la gestion des branches, nous avons adopté une stratégie Git basée sur GitFlow. La branche main est dédiée aux versions stables et prêtes pour la production. Aucune modification ne peut y être directement intégrée sans validation préalable. La branche develop est utilisée pour intégrer les nouvelles fonctionnalités. Elle représente un environnement de travail stable mais non destiné directement à la production.

Les nouvelles fonctionnalités sont développées dans des feature branches spécifiques tirés à partir de develop, chacune correspondant à une tâche ou une user story. Ce modèle garantit que plusieurs développeurs peuvent travailler simultanément sur des fonctionnalités différentes sans interférer les uns avec les autres.

Chaque fonctionnalité est développée dans une branche dédiée, à partir de la branche develop. Une fois la fonctionnalité terminée, une pull request est soumise pour fusionner la feature branch dans develop. À ce stade, des revues de code et des tests automatisés sont exécutés avant d'autoriser la fusion. Une fois validée, la branche feature est fusionnée dans develop, garantissant que seul du code testé et revu est intégré.

Au début du projet, nous avons constaté que certaines issues tentaient de résoudre trop de fonctionnalités d'un coup, rendant difficile la gestion et le suivi de leur avancement. Cela engendrait également des problèmes lors des revues de code, où de nombreux changements étaient mélangés, compliquant la validation de chaque partie.

Pour améliorer cette situation, nous avons introduit une approche de découpage vertical des fonctionnalités. L'idée était de diviser les grandes tâches en plus petits morceaux, chacun correspondant à une fonctionnalité distincte et directement testable. Par exemple, au lieu d'une seule issue pour " Gestion de l'information sur les restaurants (heures d'ouverture et menu)", nous avons créé deux issues séparées : l'une pour la gestion des horaires et l'autre pour la gestion du menu.

Chaque issue devient alors plus facile à gérer, à tester, et à valider individuellement. Cela a permis de mieux suivre l'avancement des fonctionnalités tout en facilitant les revues de code, car chaque pull request était beaucoup plus concise et précise. Ce découpage vertical nous a également aidés à mieux prioriser les fonctionnalités critiques pour le projet et à livrer plus rapidement des versions fonctionnelles aux utilisateurs.

5. RESTROSPECTIVE ET AUTO-EVALUATION (2 pages)

5.1 Bilan sur notre travail d'équipe

Nous sommes globalement satisfaits du projet. Chacun a respecté son rôle et accompli ses tâches dans l'ensemble, à l'exception du Product Owner qui, malheureusement, n'a pas rempli son rôle, ce qui a grandement influencé notre avancement.

Cependant, nous avons rencontré quelques défis en termes de coordination et d'organisation, ce qui a parfois impacté le respect des délais et occasionné de légers retards. Néanmoins, cela nous offre des pistes d'amélioration pour les prochains projets.

5.2 Rôle du product owner (Sagesse ADABADJI)

En tant que Product Owner, j'ai été responsable de piloter et de coordonner les différentes étapes du développement du système.

Nous avons identifié les besoins des parties prenantes du système, notamment les utilisateurs invités, les utilisateurs du campus et les gestionnaires de restaurant. Pour chacun de ces besoins, traduits en cas d'utilisation, j'ai ensuite formulé avec des critères d'acceptation de fonctionnalité et priorisé les user stories en fonction de leur valeur ajoutée afin de constituer et prioriser le backlog produit. J'ai enfin élaboré un planning de développement structuré en sprints, avec des objectifs clairs et réalistes pour chaque itération.

J'ai participé à la vérification des fonctionnalités implémentées en m'assurant que les scénarios de tests étaient conformes. À la fin de chaque sprint, les livrables ont été validés en vérifiant leur conformité par rapport aux objectifs du sprint et aux exigences de l'étude de cas.

Compte tenu de certains retards de conformité de fonctionnalités, nous avons veillé à adapter le backlog et les objectifs des sprints. Des améliorations sont tout de même envisagées en ce qui concerne la communication régulière avec l'équipe de développement et la transparence totale sur l'avancement du projet.

5.3 Rôle du software architect (Sara TAOUFIQ)

En tant que Software Architect, j'ai veillé à assurer l'intégrité de notre architecture de la manière suivante :

Notre **diagramme de classe** nous a accompagné tout au long du projet en évoluant et s'adaptant à chaque changement qu'on envisageait de faire.

Dès le début, nous avons structuré notre système en respectant le principe de **responsabilité unique**, avec chaque classe ayant un seul et unique rôle clairement défini.

Ensuite viens l'intégration des designs pattern, l'implémentation des différents designs s'est déroulée sans difficulté majeure, à l'exception du pattern **"Facade",** qui a nécessité un refactoring pour répondre aux besoins spécifiques du projet.
Ce refactoring, en plus d'autres précédemment discutée, nous a couté du temps, qui aurait pu être utilisé pour réaliser d'autres améliorations.

Au début, notre architecture était centrée autour de nos modèles et se focalisait principalement sur eux. Nous avons dû refactorer l'ensemble pour intégrer les éléments nécessaires à notre projet, tels que des repositories, des interfaces, ainsi qu'une séparation en sous-systèmes.

En tant que Software Architect, j'admets que ces refactorings auraient pu être évités si j'avais disposé des connaissances nécessaires dès le début du projet.

5.4 Rôle du Quality Assurance Engineer (Selom Ami ADZAHO & Jean Paul ASSIMPAH)

En tant que Quality Assurance Engineers (QAs) de l'équipe, notre rôle consistait à assurer la qualité globale du projet en testant nos fonctionnalités de manière exhaustive pour garantir la fiabilité et la robustesse du code.

Nous avons veillé à ce que chaque fonctionnalité soit rigoureusement testée à travers des tests fonctionnels (Cucumber) et des tests unitaires, non seulement pour vérifier leur bon fonctionnement, mais aussi pour identifier les bugs et gérer les cas d'erreurs. Cette méthodologie nous a permis d'atteindre un taux de couverture des tests de **81%** avec un grand nombre de scénarios testés. L'outil **SonarLint** a été aussi utilisé pour améliorer la qualité du code en détectant rapidement les problèmes.

En rétrospective, il aurait été préférable d'assurer une meilleure structuration du plan de tests dès le départ. Idéalement, chaque scénario Cucumber aurait dû être immédiatement

suivi de tests JUnit pour garantir la robustesse du code à chaque étape, ce qui n'a pas toujours été systématiquement fait.

Enfin, bien que nous n'ayons pas pu couvrir tous les tests nous-mêmes, **l'implication de l'ensemble de l'équipe** a été précieuse pour garantir que le projet atteigne un haut niveau de qualité. Cette collaboration a permis de renforcer l'efficacité des tests et d'améliorer la qualité globale du produit.

5.5 Rôle du OPS (Abenezer YIFRU)

Dans le cadre du project, en tant qu'Ops, nous avons pris en charge l'aspect qualité et gestion des processus automatisés liés aux tests, et à la gestion des branches.

La qualité des tests a été surveillée grâce à des outils tels que JaCoCo pour la couverture et une instance de SonarQube déployé sur le cloud pour l'analyse statique du code. Ces outils ont permis de détecter et corriger des failles potentielles et des duplications de code.

Chaque nouvelle fonctionnalité a été développée dans des feature branches individuelles, puis fusionnée dans develop via des pull requests. Nous avons mis l'accent sur une revue de code avant tout merge sur les branches principales (main et develop). Cette revue s'assurait que le code respecte les standards de qualité définis et évitait l'introduction de bugs.

Nous avons configuré un pipeline d'intégration continue (CI) à l'aide de GitHub Actions. Ce pipeline s'exécute automatiquement à chaque pull request ou validation de code, incluant les étapes suivantes : build du projet, exécution des tests unitaires et tests d'intégration, et analyse statique avec SonarQube pour garantir la qualité du code. Cette automatisation a réduit le temps nécessaire à la validation et intégration des nouvelles fonctionnalités.

5.6 Auto-Evaluation

Sagesse ADABADJI	Sara TAOUFIQ	Jean Paul ASSIMPAH	Abenezer YIFRU	Selom ADZAHO
100	***100***	***100***	***100***	***100***

Nous avons décidé de nous attribuer la même note à chacun, car tout le monde a été impliqué dans le projet malgré les nombreuses difficultés que nous avons rencontrées.