Rapport D1 Team K

512 Eats

Équipe :

- ALLAIN Emma : QA
- BACON Roxane : OPS
- FADDA RODRIGUEZ Antoine : SA
- LEFÈVRE Clément : PO

Sommaire

I. Périmètre fonctionnel

I.1 Hypothèses de travail

Pour mener à bien notre projet et répondre aux exigences, nous avons formulé plusieurs hypothèses de travail pour clarifier certains points :

- **H1. Localisation pré-enregistrée** : Les localisations sont supposées être déjà enregistrées dans la base de données, sans création en temps réel.
- **H2. Plat uniquement** : Les restaurants ne proposent que des plats individuels (entrée, plat, dessert, boisson) sans formules, laissant aux clients le soin de composer leur repas.
- **H3. 1 livreur <=> 1 commande** : Chaque commande (subOrder) est prise en charge par un seul livreur, sans affectation multiple.
- **H4. Horaires d'ouverture** : Les restaurants sont ouverts tous les jours de la semaine, du lundi au vendredi, avec des horaires fixes.
- **H5. Temps moyen de préparation d'une commande** : Dans chaque restaurant, toutes les commandes ont un temps de préparation moyen identique prédéfini par le manager du restaurant.
- **H6. Temps maximum de préparation** : Le temps maximal pour préparer une commande est supposé être de 30 minutes.
- **H7. Temps moyen de livraison** : Le temps de livraison estimé pour une commande est de 20 minutes.

I.2 Limites identifiées

En fonction des hypothèses posées et des exigences initiales, certaines limites ont été identifiées :

- **L1. Livraisons non optimisées** : Le trafic n'est pas pris en compte, et chaque livreur est affecté à une seule commande, ce qui n'optimise pas les livraisons dans le cas de commandes individuelles non groupées.
- **L2. Pas de formules** : Chaque client doit composer son repas en sélectionnant les produits individuellement sur la carte.
- **L3. Non responsabilité du paiement**: Le processus de paiement n'étant pas à prendre en charge, malgré une couverture de cas d'erreur pour les objets envoyés au processus de paiement, nous ne pouvons pas garantir le bon déroulé d'un processus externe.
- **L4. Préavis de commande de 50 min minimum** : Pour assurer la préparation de la commande, celle-ci doit être faite au moins 50 minutes à l'avance car il faut prendre en compte le temps de livraison (20 minutes) et le

temps de préparation maximum (30 minutes dans le créneau horaire du restaurant).

I.3 Stratégies choisies et éléments spécifiques

Nous avons mis en place une stratégie bien particulière pour la gestion des commandes en fonction des temps de préparation de celles-ci. Cette stratégie permet un compromis entre les utilisateurs et les restaurateurs, et donc éviter de créer une frustration importante d'un côté comme de l'autre.

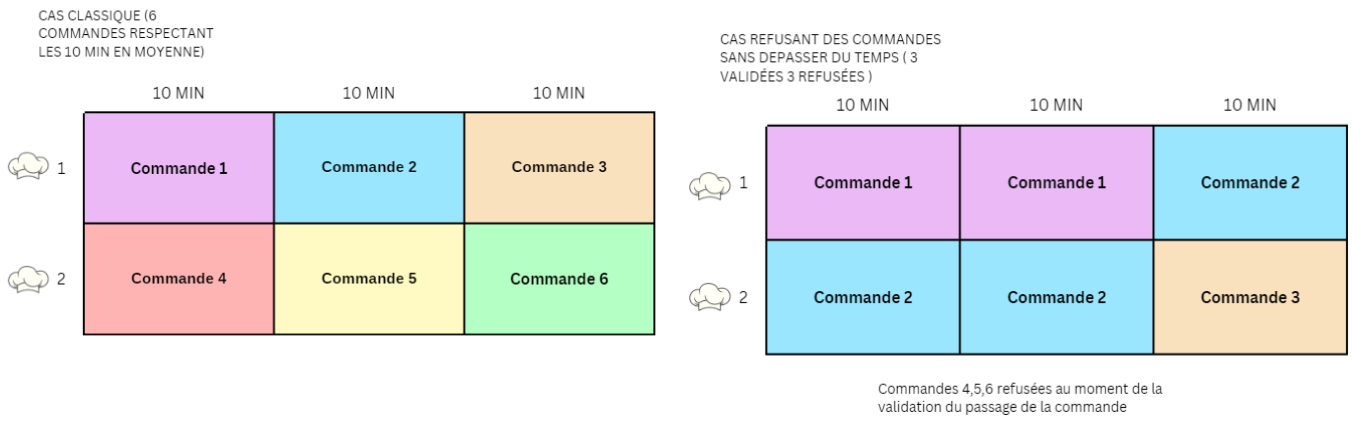
Nous sommes partis de l'exigence R2 qui fixe la durée d'un créneau (time slot) à 30 minutes.

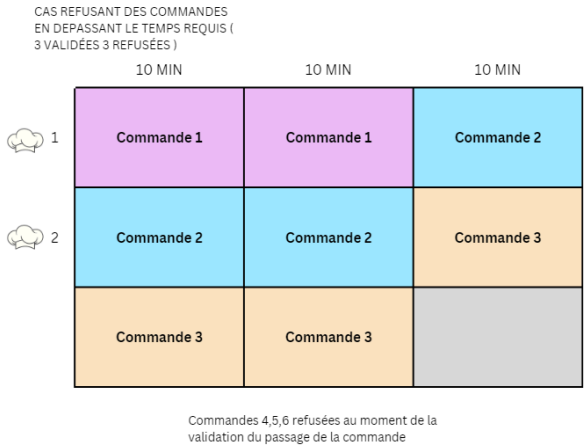
Exemple de fonctionnement :

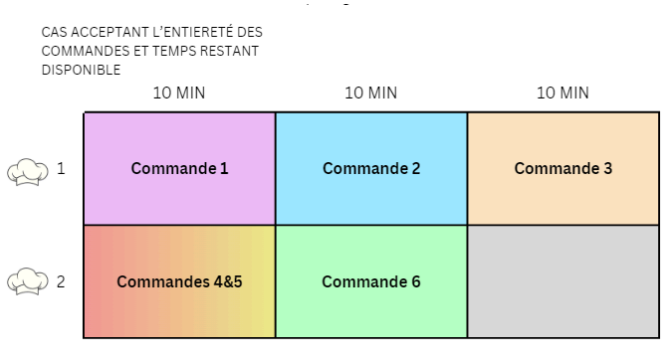
En ayant un temps moyen de préparation de 10 minutes par commande, avec une seule personne pour les préparer, nous autorisons 3 commandes passée par TimeSlot (30 minutes/10 minutes * 1 préparateur(s) de commande). Si nous avons 2 préparateurs de commande, alors nous permettrons à 6 utilisateurs enregistrés de passer une commande, etc.

Remarque:

Si il y a trop de demandes pour un restaurant à un horaire précis, alors nous n'autorisons plus la création de commande pour ce restaurant. Reprenons notre exemple avec 2 préparateurs de commandes, soit 6 commandes possibles. Si les 3 premières prennent l'entièreté du temps disponible, alors seulement 3 utilisateurs ayant déjà créé une commande seront bloqués avant de pouvoir valider celle-ci. (voir schéma ci-dessous). A l'inverse, peut être que les 6 commandes seront courtes et du temps pourrait être perdu, ou bien que le temps de préparation de la dernière commande acceptée peut être plus long que le temps disponible, ce qui mettrait en retard le restaurant.







II. Conception UML

II.1 Glossaire

Plat / Dish : utilisé pour désigner un plat, un accompagnement, une boisson ou un dessert qu'un restaurant propose.

Type de nourriture / Food type: correspond à tous les types de nourriture que l'on peut retrouver dans le restaurant : végan, asiatique, sans gluten etc...

SubOrder : C'est une commande qui n'a pas de localisation et qui est créée dans un groupe. La commande sera livrée à la localisation indiquée dans la commande de groupe (group order) liée.

Commande individuelle / Individual order: C'est une seule commande créée par une seule personne avec sa propre localisation.

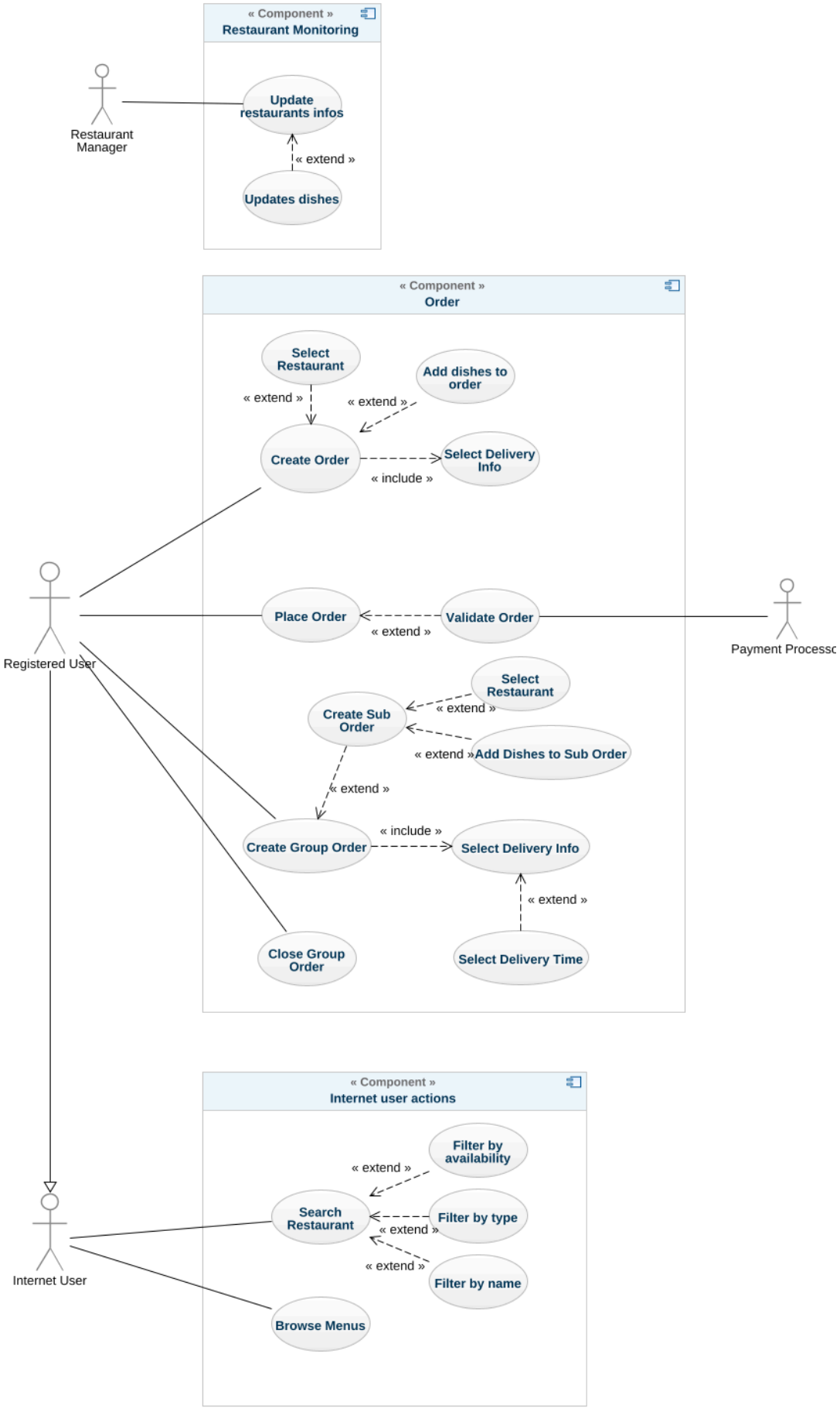
Commande de groupe / Group order: C'est un groupe créé par une personne. D'autres personnes peuvent le rejoindre et commander sur plusieurs restaurants et recevoir leurs commandes au même endroit en même temps.

Informations du restaurant : les informations du restaurant comprennent les horaires d'ouverture et de fermeture, les plats que proposent le restaurant, les type de nourriture qu'il propose ainsi que le temps moyen de préparation d'une commande

Menu : la liste des plats (*dishes)* proposés par un restaurant.

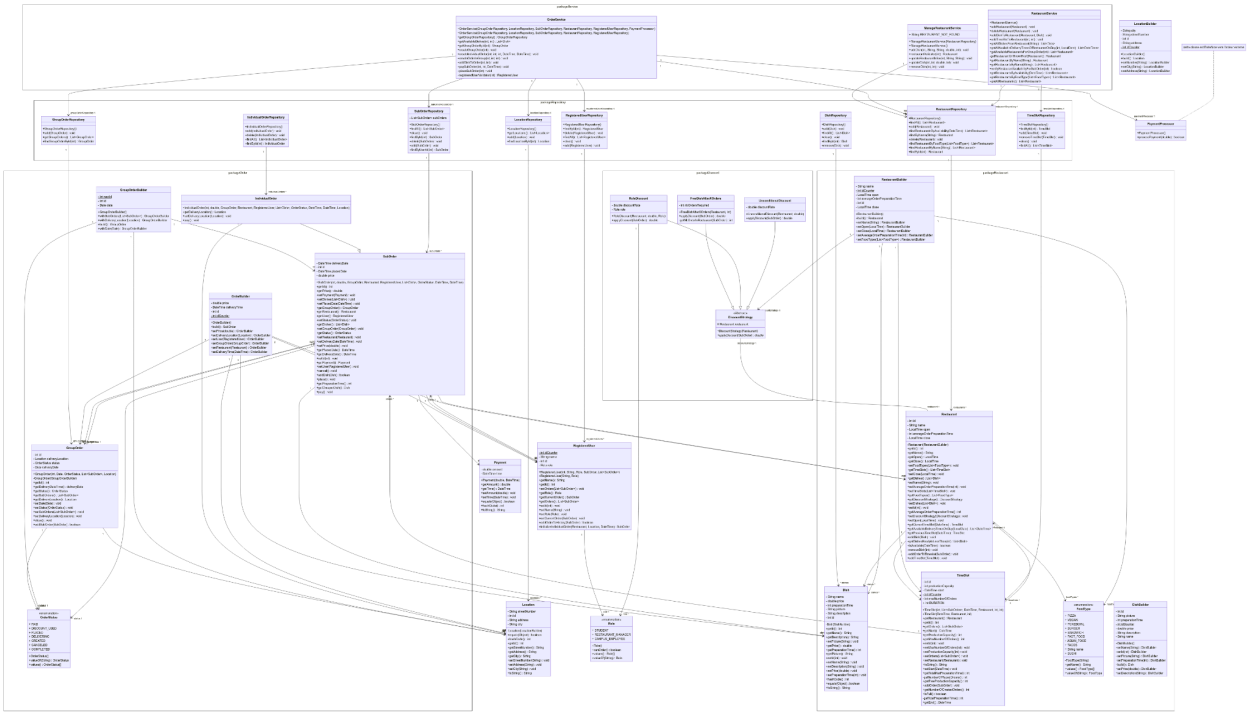
Discount : une méthode de promotion propre à un restaurant.

II.2 Diagramme de cas d'utilisation



II.3 Diagramme de classes

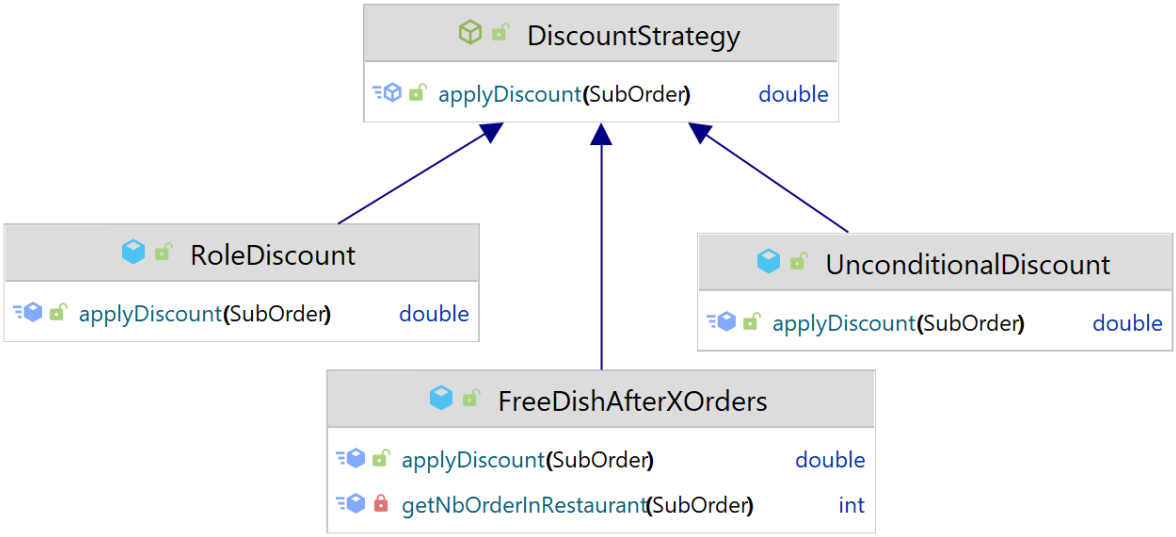
Le diagramme de classe ci-dessous a été généré à partir de notre code. Nous avons décidé de le compléter pour le rendre cohérent avec le diagramme de séquence disponible plus bas. Notre diagramme de classe n'est donc pas tout à fait réaliste car nous n'avons pas eu le temps d'implémenter la fonctionnalité représentée par le diagramme de séquence. Cependant, pour le rendre plus lisible, nous avons décidé de regrouper les classes dans différents packages et proposer une version Markdown ici.



II.4 Design patterns

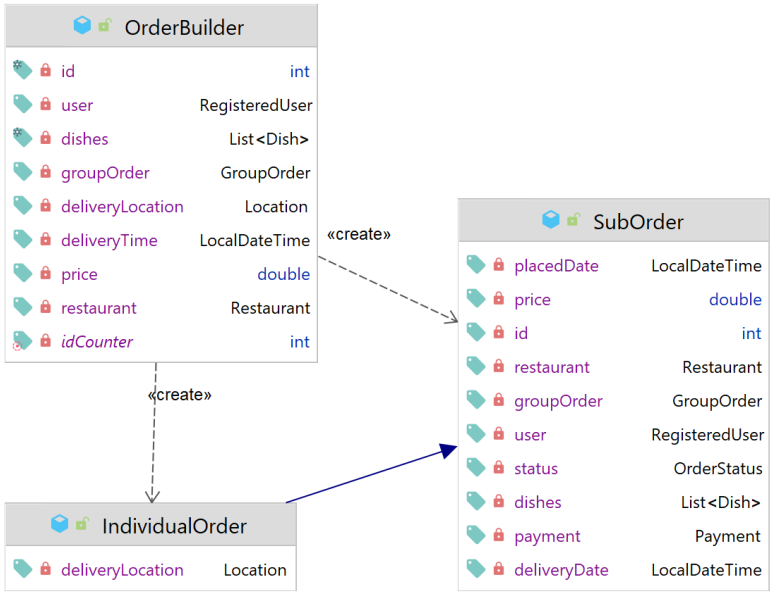
II.4.a Strategy

Nous avons choisi d'implémenter la fonctionnalité des ristournes personnalisées à l'aide du design pattern strategy. Celui-ci nous permet d'avoir différentes implémentations pour une même méthode, et ainsi différents types de ristourne de manière tout à fait transparente. Ainsi, chaque manager de restaurant pourra choisir la stratégie de ristourne à appliquer dans son restaurant. Techniquement, chaque Restaurant possède une DiscountStrategy, qui pourra être modifiée par le manager via le RestaurantService.

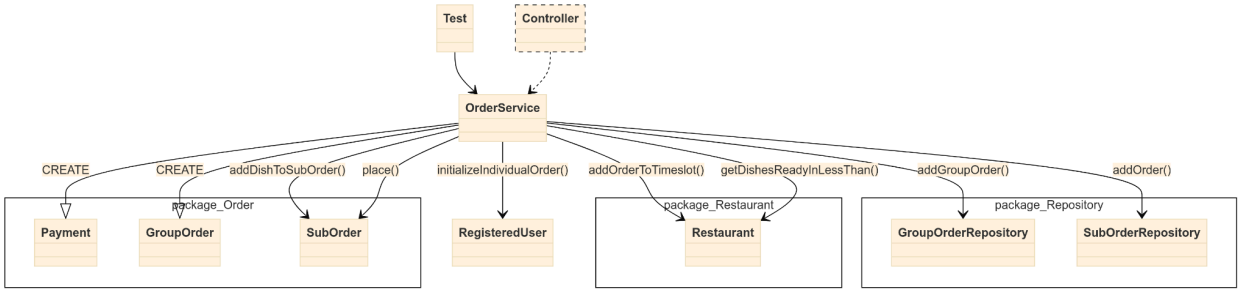


II.4.b Builder

Pour simplifier la construction de nos objets dans l'ensemble de notre projet, nous avons utilisé à plusieurs reprises le design pattern builder. Il nous permet de créer des objets simplement sans avoir de constructeur prenant de nombreux paramètres. Ci-dessous se trouve l'exemple de notre OrderBuilder, qui nous permet de construire de manière transparente des SubOrders ou des IndividualOrders en fonction des paramètres données. Si le builder se voit attribuer une GroupOrder, il construira une SubOrder, sinon, s'il reçoit une deliveryLocation, alors il construira une IndividualOrder.



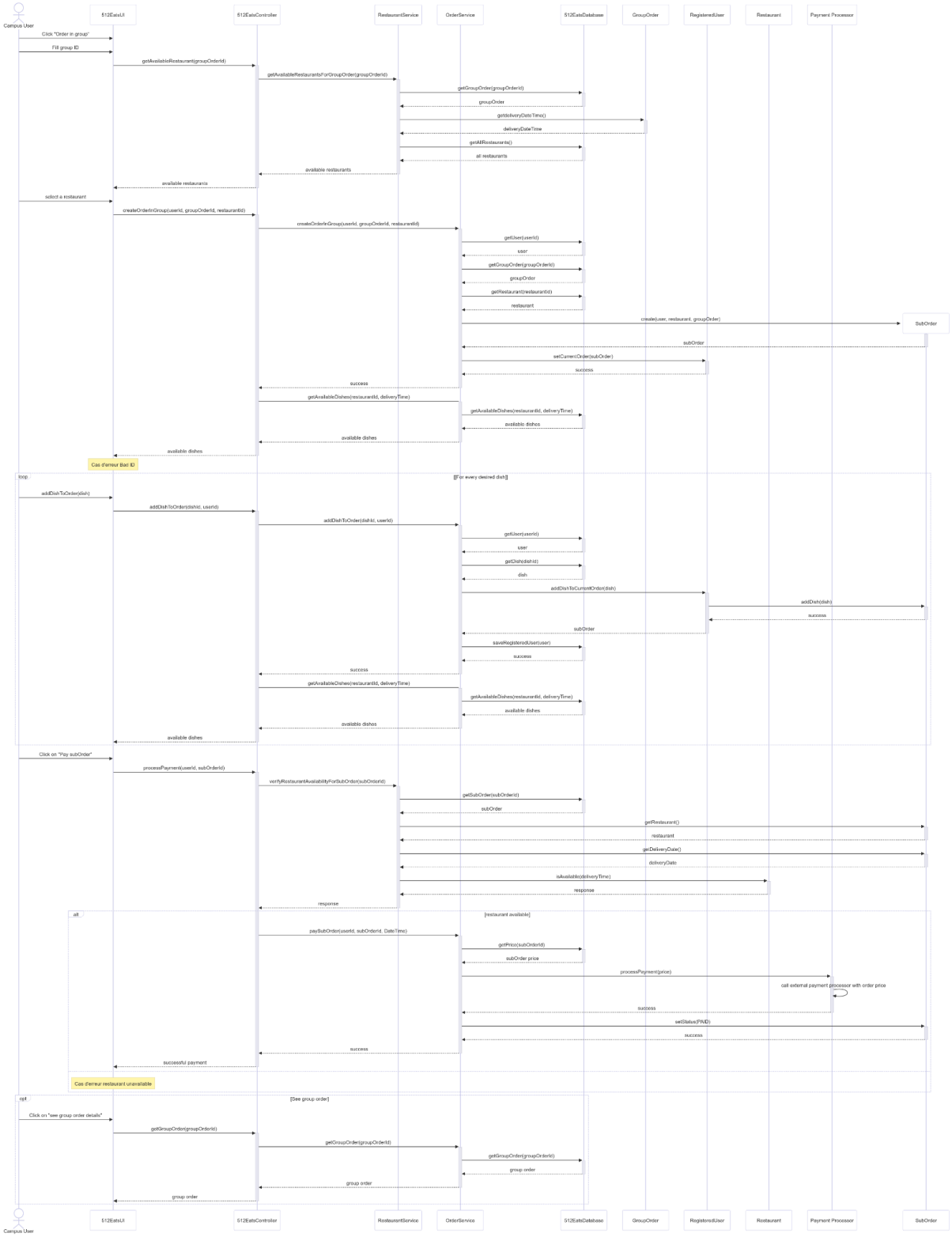
II.4.c Façade



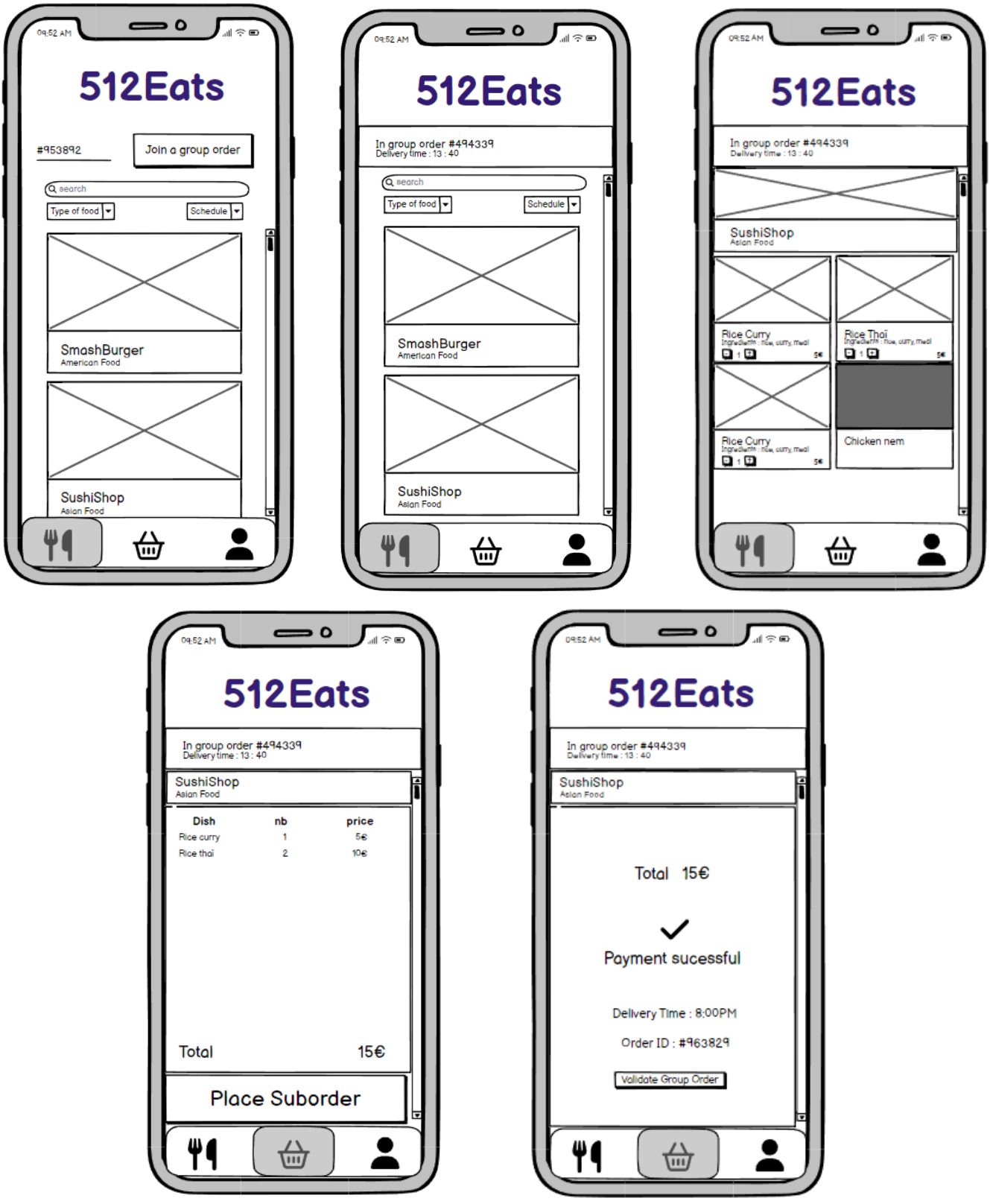
Nous avons utilisé le design pattern façade pour mettre en place des services. Ceux-ci simplifieront grandement l'implémentation futur de contrôleurs REST. Ainsi, nous avons par exemple l'OrderService, qui gère toutes les fonctionnalités liées aux commandes. Le diagramme ci-dessous montre les classes et méthodes utilisées par ce service.

II.5 Diagramme de séquence

Ci-dessous, le diagramme de séquence représentant le passage d'une commande dans le contexte d'une commande de groupe. Nous n'avons malheureusement pas eu le temps d'implémenter cette fonctionnalité. Une version Markdown du diagramme est disponible ici


Campus User
512EatsUI
512EatsController
RestaurantService
OrderService
512EatsDatabase
GroupOrder
RegisteredUser
Restaurant
Payment Processor
SubOrder
Click "Order in group"
Fill group ID
getAvailableRestaurant(groupOrderId)
getAvailableRestaurantsForGroupOrder(groupOrderId)
getGroupOrder(groupOrderId)
groupOrder
getdeliveryDateTime()
deliveryDateTime
getAllRestaurants()
all restaurants
available restaurants
available restaurants
select a restaurant
createOrderInGroup(userId, groupOrderId, restaurantId)
createOrderInGroup(userId, groupOrderId, restaurantId)
getUser(userId)
user
getGroupOrder(groupOrderId)
groupOrder
getRestaurant(restaurantId)
restaurant
create(user, restaurant, groupOrder)
subOrder
setCurrentOrder(subOrder)
success
success
getAvailableDishes(restaurantId, deliveryTime)
getAvailableDishes(restaurantId, deliveryTime)
available dishes
available dishes
available dishes
Cas d'erreur Bad ID
loop
[For every desired dish]
addDishToOrder(dish)
addDishToOrder(dishId, userId)
addDishToOrder(dishId, userId)
getUser(userId)
user
getDish(dishId)
dish
addDishToCurrentOrder(dish)
addDish(dish)
success
subOrder
saveRegisteredUser(user)
success
success
getAvailableDishes(restaurantId, deliveryTime)
getAvailableDishes(restaurantId, deliveryTime)
available dishes
available dishes
available dishes
Click on "Pay subOrder"
processPayment(userId, subOrderId)
verifyRestaurantAvailabilityForSubOrder(subOrderId)
getSubOrder(subOrderId)
subOrder
getRestaurant()
restaurant
getDeliveryDate()
deliveryDate
isAvailable(deliveryTime)
response
response
alt
[restaurant available]
paySubOrder(userId, subOrderId, DateTime)
getPrice(subOrderId)
subOrder price
processPayment(price)
call external payment processor with order price
success
setStatus(PAID)
success
success
successful payment
Cas d'erreur restaurant unavailable
opt
[See group order]
Click on "see group order details"
getGroupOrder(groupOrderId)
getGroupOrder(groupOrderId)
getGroupOrder(groupOrderId)
group order
group order
group order

III. Maquettes



IV. Qualité des codes et gestion de projets

IV.1 Tests

Pour tester notre projet, nous avons implémenté des tests fonctionnels en utilisant Cucumber et des tests unitaires avec Junit.

Nous avons, avec les tests cucumbers, tester divers scénarios et avons également mis en place de la gestion d'erreur. Pour chaque erreur nous avons implémenté des messages personnalisés pour pouvoir directement les injecter dans notre front-end de l'application au moment de l'implémentation.

Concernant les tests unitaires, nous avons, par manque de temps, fait le choix de tester uniquement les classes et méthodes fonctionnellement complexes comme les méthodes de la classe TimeSlot qui se charge de la gestion du temps.

IV.2 Qualité du code

IV.2.a Code de bonne qualité

Pour mettre en avant la qualité de notre code, nous pouvons parler de divers sujets notamment :

- L'utilisation de divers design patterns qui nous permet d'avoir de bonnes structures et lisibilité.
- L'utilisation de petites méthodes plutôt que des grosses méthodes à cognitivité complexe pour une meilleure lisibilité.
- L'architecture globale est bien pensée, avec des packages clairement définis, facilitant la navigation et la compréhension du projet.
- La présence d'une javadoc assez détaillée pour les méthodes non triviales.
- Les scénarios de tests couvrent un large éventail de cas, y compris la gestion des erreurs, garantissant ainsi une certaine robustesse.

IV.2.b Code à améliorer

Nous sommes conscients que nous avons des parties de code à améliorer. Nous pouvons développer certains points comme par exemple :

- Nous n'avons pas eu le temps d'implémenter le design pattern proxy afin de vérifier le droit d'un utilisateur à passer commande ou non, nous l'avons fait d'une manière moins optimisée.

- Nous donnons trop de responsabilité à notre “registeredUser” qui utilise l’OrderBuilder alors la responsabilité devrait revenir à l’OrderService.
- Nous avons donné un peu trop de responsabilités à la classe OrderService, car plusieurs de ses méthodes devrait plutôt être exposées par d’autre services : `getAvailableDishes()` devrait être exposé par le RestaurantService et `paySubOrder()` devrait être exposé par un PaymentService.
- Notre classe IndividualOrder n’est pas vraiment pertinente car elle n’a qu’un seul attribut en plus que sa classe mère : SubOrder. Il faudrait analyser cette partie de la conception pour régler ce problème.
- La complexité cognitive de certaines méthodes est trop élevée et a été détectée par notre extension Sonar Lint, qui mériterait un refactor.

IV.3 Gestion du dépôt

En ce qui concerne la gestion du dépôt et du développement. Un Pipeline vérifiait nos développements en lançant un maven test.

Nous avons adopté une stratégie de branche “git flow” avec une branche “main”, une branche “dev” et des branches feature et fix pour chaque implémentation de fonctionnalités ou de correction de bugs.

Pour chaque branche feature ou fix nous faisions une Pull Request qui devait être relue et approuvée par au moins un membre de l’équipe pour que le merge sur dev soit possible.

V. Rétrospective et Auto-évaluation

V.1. Recul

Au cours de ce projet, nous avons appris divers principes que nous remettrons en pratique dans le futur. Tout d'abord nous avons mesuré l'intérêt de cerner le besoin utilisateur afin de ne pas produire "trop" ou "inutilement". Nous avons tout autant pris en compte l'importance qu'avait la modélisation dans un projet d'une ampleur comme celui-ci, afin de pouvoir travailler librement et efficacement. Ces mesures nous permettent alors de travailler en autonomie sans pour autant laisser place à l'interprétation de chacun qui pourrait apporter des quiproquos au sein du projet.

L'auto-évaluation a favorisé l'évolution de notre projet en nous offrant de nouveaux points de vue grâce aux retours des autres équipes, ce qui a permis de corriger et d'affiner notre modélisation. Seuls les retours jugés pertinents ont été pris en compte. D'un point de vue individuel, évaluer un autre groupe a renforcé notre compréhension des notions et nous a aidé à éviter les erreurs identifiées chez les autres.

V.2. Missions accomplies

Emma Allain QA : Sur le projet nous avons implémenté différents types de tests afin de s'assurer du bon fonctionnement de notre développement/ projet. Nous avons mis en place des tests cucumber avec des scénarios divers recouvrant les exigences et extensions implémentées pour le côté fonctionnel. De plus à ces tests cucumber nous avons implémenté des tests unitaires pour s'assurer du bon fonctionnement de nos méthodes et de nos algorithmes. Enfin toutes nos user stories en must have étaient implémentées et testées lors de notre présentation. Nous avons donc bien priorisé nos tâches.

Roxane Bacon OPS : Nous avons adopté une approche qui vise à assurer la qualité de notre projet, avec des tests Cucumber pour les scénarios fonctionnels et des tests unitaires pour valider nos méthodes et algorithmes. Le suivi de l'avancement était géré via un board GitHub Project en Kanban, organisé en sprints hebdomadaires avec des milestones, des user stories et des issues labellisées par priorité. Nous avons mis en place une stratégie de branches git flow : une branche dev (devenue notre branche par défaut afin que les issues soit fermées automatiquement lorsqu'une pull request sur dev est acceptée) pour tester les nouvelles fonctionnalités avant leur intégration dans main, garantissant ainsi la stabilité. De plus, nous avons implémenté

une stratégie de nommage : “feature/...”, “fix/...”, etc, visant à organiser intelligemment les branches. La répartition des tâches était équilibrée, ce qui a permis de respecter les délais et de gérer efficacement le développement.

Antoine Fadda Rodriguez SA : Pour assurer une architecture de qualité au projet, nous avons commencé par nous documenter sur les principaux design patterns. De cette manière, nous avons été en capacité d’identifier les endroits où ils allaient être utiles. En collaboration étroite avec le PO, nous avons aussi réalisé une importante phase de conception. Celle-ci s’est avérée plutôt efficace car, depuis la dernière analyse ayant eu lieu après l’auto-évaluation, notre architecture n’a pas subi de changement majeur. De plus, selon Sonarqube, nous avons moins d’une heure de dette technique liée à quelques soucis de maintenabilité (constructeur prenant trop de paramètres, attributs pas encore utilisés, ...)

Clément Lefèvre PO : Nous avons structuré le travail de développement en veillant à ce que chaque fonctionnalité soit correctement définie et priorisée pour répondre aux besoins du projet. Nous avons élaboré chaque user story en utilisant les templates d'issues fournis, garantissant une définition claire des critères d'acceptation et des objectifs. Nous avons veillé à ce que les user stories soient bien comprises par l'équipe avant le début de chaque sprint, facilitant ainsi une planification efficace et une exécution fluide.

Le suivi de l'avancement a été assuré via un board GitHub Project en Kanban, où nous avons travaillé en sprints d'une semaine, permettant une gestion agile de nos développements. Chaque sprint était associé à une milestone, et les issues étaient labellisées selon leur priorité et leur nature (feature, test, user story), ce qui a permis un suivi clair et une transparence dans l'avancement des travaux.

Nous nous sommes également assurés que la répartition des tâches soit équitable au sein de l'équipe, tout en veillant à ce que les deadlines soient respectées.

V.3. Auto-évaluation / implication

Concernant l'attribution des points, ayant tous travaillés équitablement, nous avons décider d’attribuer les points de manière homogène:

Emma: 100pts / Roxane: 100pts / Antoine: 100pts / Clément: 100pts