CONCEPTION LOGICIELLE

Cookie Factory

POLYTECH NICE SOPHIA | UNIVERSITÉ CÔTE D'AZUR

TEAM M

DEAL Emma – LORCERY Morgane – MARINI Claire – MEHDI Khadidja – PRINCE Jules

Sommaire

1. UML

A) Diagramme de cas d'utilisation

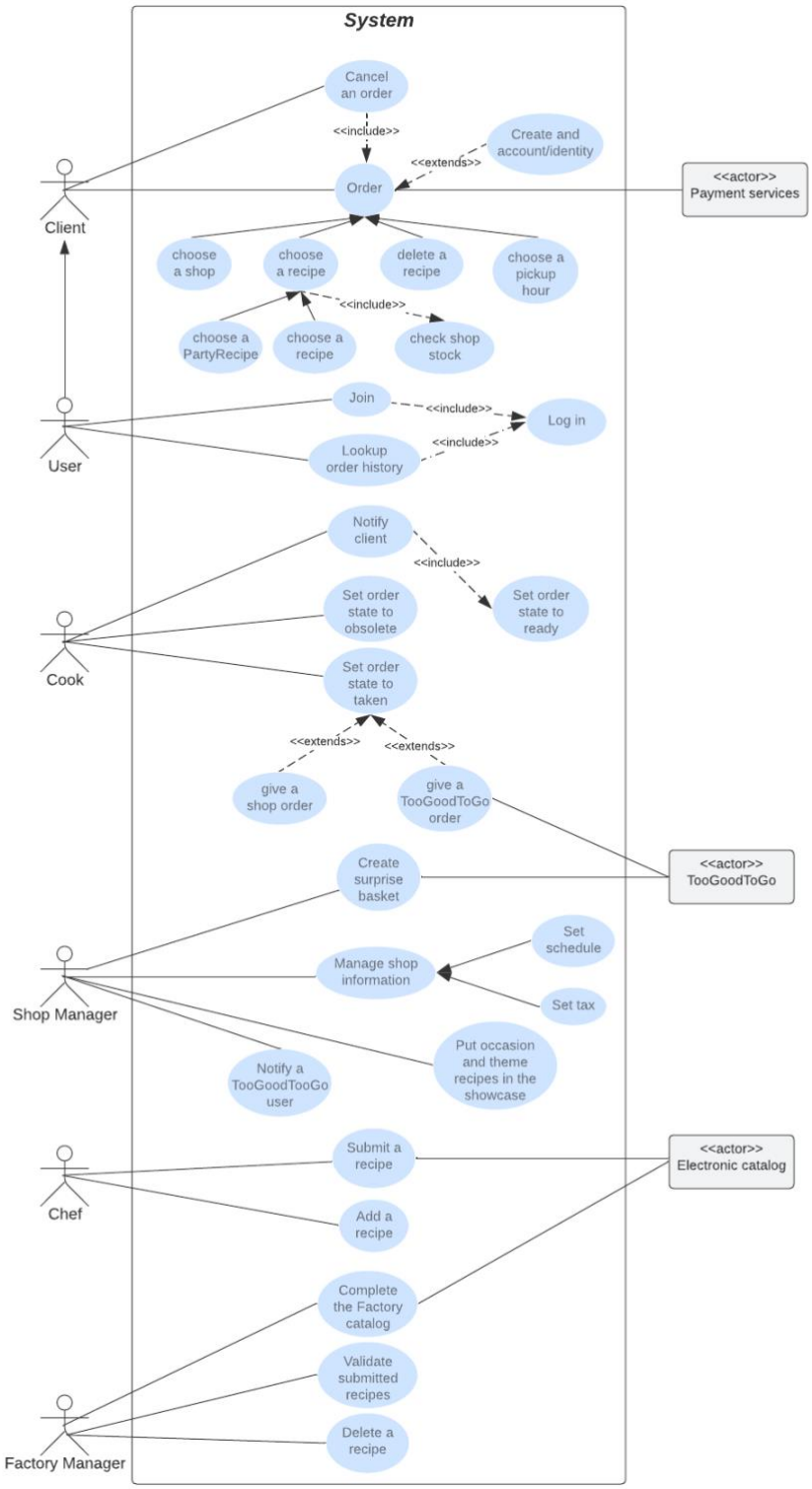

Figure 1: Diagramme de cas d'utilisation

Hypothèses

- Un client n'est pas connecté et peut seulement passer commande. À la récupération de sa commande, il sera supprimé de la "base de données" par le manager RGPD. Quant à l'utilisateur, qui est connecté, il restera en mémoire dans la base de données client et un historique de commandes lui sera associé.
- Dès lors qu'une commande est obsolète depuis 2h, celle-ci est transformée en panier TooGoodToGo par le Shop Manager du magasin concerné. Lors de la création, l'API TooGoodToGo est mise au courant de l'existence du panier. Les utilisateur ayant accepté de recevoir des notification pour les panier TooGoodToGo sont également avertis par le Shop manager.
- Les Cuisiniers de chaque magasin et associés aux commandes gèrent le fait de notifier les utilisateurs une fois que la commande est prête. Ils changent également l'état des commandes en "**OBSOLETE**" si personne ne vient les récupérer après 2h. Si un client vient récupérer une commande, il pourra le faire auprès du Cuisinier en se munissant de l'ID de sa commande, ainsi le Cuisinier passera la commande en "**TAKEN**".
- La Factory possède un catalogue d'ingrédients disponibles. Ce catalogue référence les ingrédients disponibles dans un catalogue extérieur. C'est le Factory Manager qui assure le remplissage du catalogue de la Factory.
- Les Chefs sont responsables de l'ajout et de la suppression d'une recette chaque mois.
- Les particularités de chaque magasin (taxe, horaires, occasions) sont gérées par le Shop Manager.
- Chaque magasin possède une "vitrine" qui permet d'exposer seulement les recettes, occasions et thèmes disponibles dans le magasin en question selon le stock et les cuisiniers qui y travaillent à ce moment-là (notamment pour les thèmes). C'est également le Shop Manager qui gère cela.

Commande

Une commande se déroule ainsi :

- Le client choisit un magasin (il est logique de commencer par cette étape afin de proposer par la suite au client seulement des recettes, thèmes, occasions et ingrédients qui sont disponibles dans le magasin où il veut récupérer sa commande).
- Une fois le magasin choisi, une liste de recettes provenant de la "vitrine" lui est proposée, ainsi que les thèmes, occasions et ingrédients (qui sont utiles pour personnaliser une Party Cookie Recipe) disponibles.

Ensuite, il peut :

- Ajouter une recette à son panier (dans la quantité qu'il désire). Le stock du magasin est alors décrémenté.
- Supprimer une recette de son panier. Le stock est alors réincrémenté.
- Ajouter une Party Cookie Recipe (personnalisée ou non).

Enfin, il pourra choisir une heure de retrait et payer sa commande.
Il peut ensuite l'annuler si sa préparation n'a pas commencé. Ou alors, il peut la récupérer au comptoir du magasin en se munissant de l'ID de sa commande, quand celle-ci sera prête.

À savoir :
- Le mécanisme pour passer les commandes en obsolète et notifier les clients sont des "Cron Spring" qui vérifient l'état des commandes notées prêtes.
- Le mécanisme de création de paniers TooGoodToGo est aussi un Cron, ainsi que celui responsable de la création/suppression de recettes chaque mois.
Cela est bien plus simple et efficace que de demander l'intervention de l'homme pour ces actions répétitives.

B) Diagramme de classes

Le diagramme de classes se trouve en **Annexe 1 : Diagramme de classes**.

C) Diagramme de séquence

User story : Passer une commande

Pour une question de visibilité, le diagramme de séquence a été divisé en deux parties. Une première partie qui concerne l'ajout de recette dans le panier de l'utilisateur. En second, la partie où l'utilisateur choisit l'heure de récupération de sa commande et le paiement de cette dernière. Les diagrammes sont donc à lire dans l'ordre.

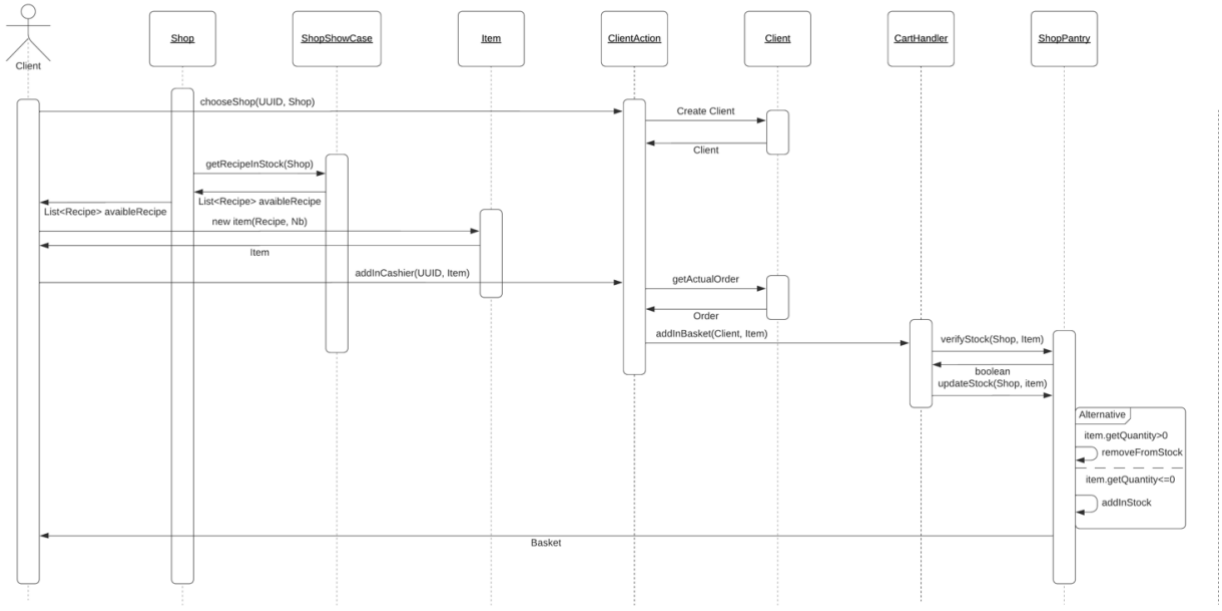

Figure 2: Première partie du diagramme de séquence

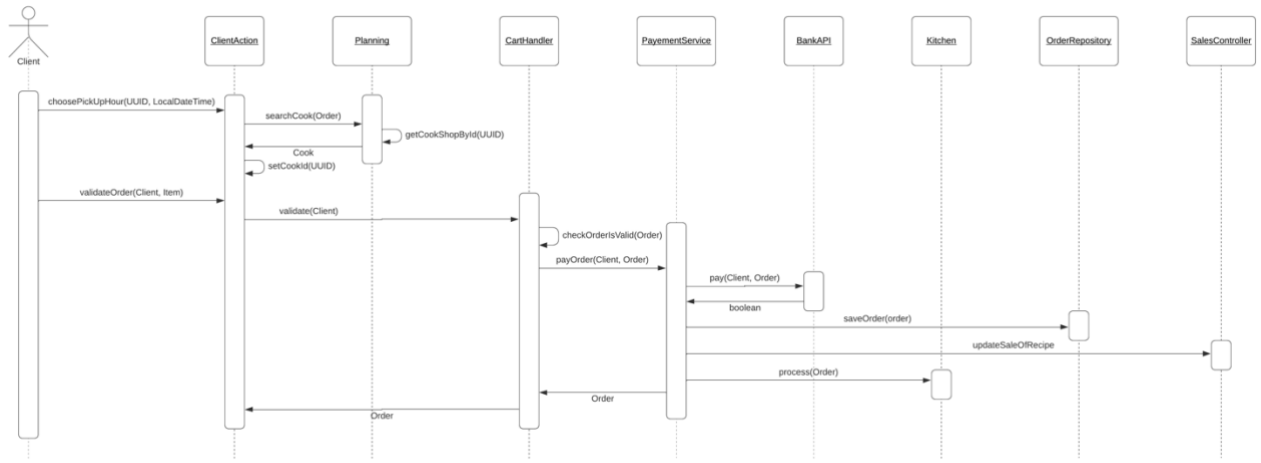

Figure 3: Deuxième partie du diagramme de séquence

2. Patrons de conception

A) Pattern Observer

Pour notre projet, nous avons décidé d'utiliser un patron de conception de comportement, l'**Observer**, dans le cadre de TooGoodToGo. Les *observers* sont les utilisateurs authentifiés de notre application, qui vont être notifiés par les magasins auxquels ils se sont abonnés lorsqu'un nouveau panier surprise est disponible. Ce patron de conception met en jeu 3 composants. Le premier est SurpriseBasketService, qui a pour responsabilité de créer des panier TooGoodToGo à partir de commandes obsolètes et de transmettre des informations à l'API TooGoodToGo. C'est aussi lui qui notifie les utilisateurs voulant recevoir des notifications lorsqu'un nouveau panier TooGoodToGo est disponible. Pour envoyer ces notifications, il va utiliser les deux autres composants :

- **ShopSubscriberService** qui permet d'ajouter ou de supprimer des abonnés à un magasin et de notifier ces derniers grâce au composant NotificationObserverService.
- **NotificationObserverService**, qui permet d'ajouter une notification a tous les utilisateurs ayant souscrit à un magasin.

Cela nous a donc permis qu'un objet soit capable de notifier d'autres objets sans que ces objets ne soient fortement couplés et de simplifier le code.

B) Pattern Factory Method

Nous avons aussi décider d'implémenter une Factory Method sur la création d'ingrédients. Dough, Topping et Flavor sont des sous-classes de la classe mère Ingredient. Cette Factory nous permet donc d'instancier dynamiquement ces sous-classes par le biais d'un IngredientCreator.

Les appels directs de construction d'objet ont été remplacé par des appels à une méthode de fabrique spéciale, qui est ici la classe IngredientCreator.

C) Autres pattern

D'autres patrons de conception nous paraissaient utiles mais ayant déjà bien avancés sur le projet lorsqu'ils nous ont été présentés, nous avons préféré ne pas tout refaire au risque de perdre trop de temps à refaire le code et les tests liés. C'est le cas du patron de conception State que nous aurions pu instaurer pour pouvoir changer le comportement d'une commande quand son état (*taken*, *obsolete*...) change, sans pour autant en changer son instance.

3. Rétrospective

A) Diagramme de composants

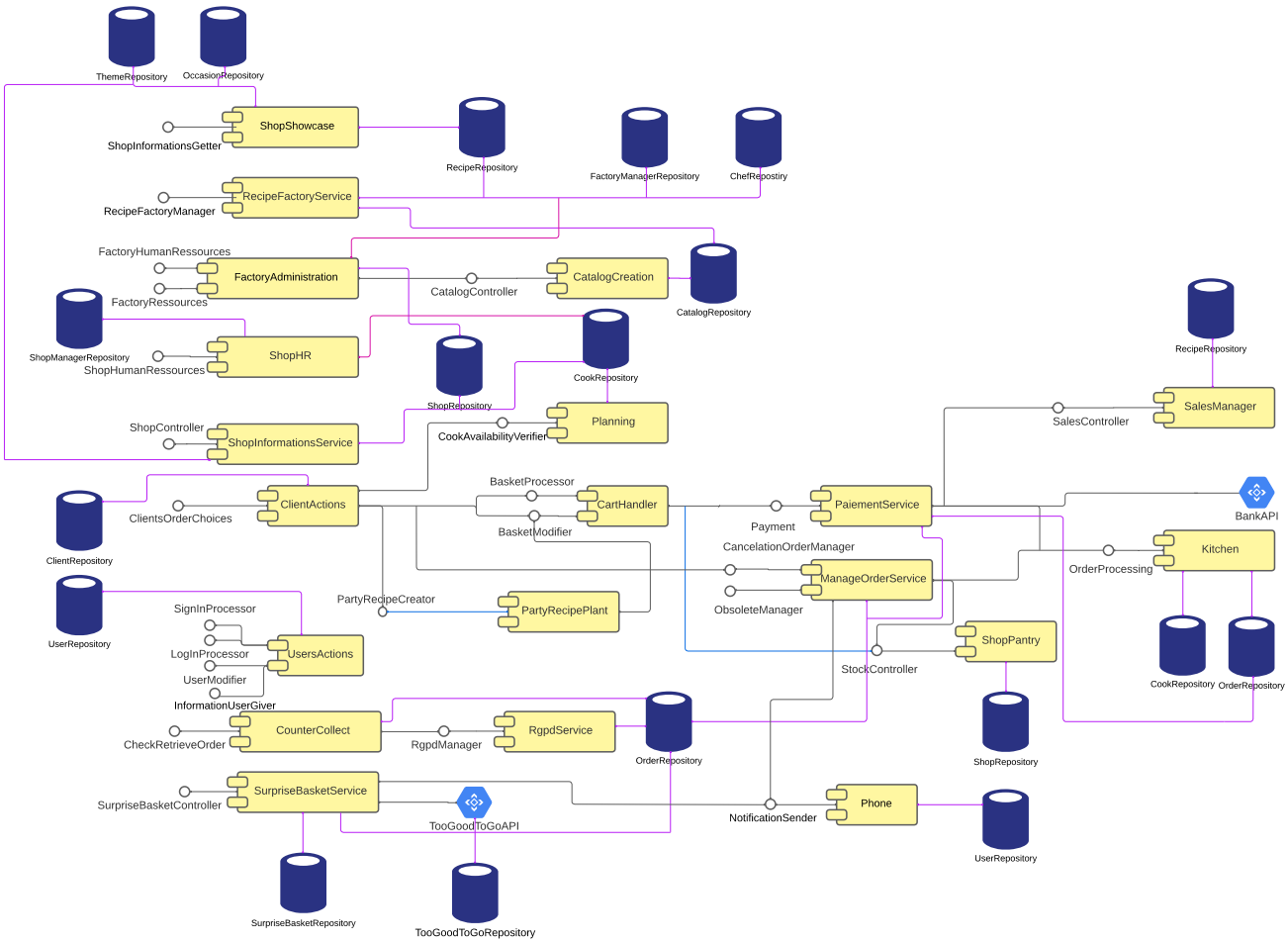

Figure 4: Diagramme de composants

B) Présentation de l'évolution du projet à travers les sprints

Pour arriver au résultat final du projet, nous avons fonctionner à l'aide de sprint, avec 1 sprint par semaine. Le but étant qu'à chaque fin de sprint, le projet soit fonctionnel et qu'il avance de manière à ajouter des fonctionnalités utiles et nécessaires.

1) Lors du premier sprint, le but était donc de mettre en place l'architecture du projet en réalisant les fonctionnalités de base. C'est-à-dire qu'il n'existait qu'un magasin, qu'un cuisinier, qu'un client, qu'une recette, … Le but étant simplement qu'un chef puisse proposer une recette et que le manager puisse la valider pour que le chef puisse l'ajouter à la Factory. Ensuite, le but était qu'un client puisse passer une commande avec un magasin et une recette (ajoutée à la Factory) puis la récupérer, sans gérer les étapes de préparation pour le moment.
2) Lors du deuxième sprint, nous avons ajouté des fonctionnalités comme la gestion des stocks ou le fait qu'une commande soit associée à un cuisinier disponible (c'est-à-dire qu'il n'ait pas de commande à préparer pour la même heure, sans prendre en compte le temps de préparation). Nous avons aussi ajouté des améliorations aux fonctionnalités de bases implémentées au sprint précédent comme l'ajout de la taxe, la limite d'ingrédients pendant la création d'une recette ou encore le fait qu'un client puisse annuler sa commande
3) Lors du troisième sprint, nous avons commencé à aller plus loin dans les fonctionnalités en ajoutant tout ce qui était en rapport avec la gestion du temps comme la prise en compte du temps de préparation des cookies ou l'ajout des horaires d'ouverture du magasin. Nous avons aussi inclus la gestion des commandes obsolètes ainsi que la gestion des annulations ou encore les notifications et des petites améliorations afin que les fonctionnalités précédentes correspondent entièrement aux attentes.
4) Lors du quatrième sprint, nous avons ajouté des fonctionnalités manquantes afin que notre projet respecte bien toutes les demandes. Nous avons donc ajouter l'historique des commandes, l'adhésion au Loyalty Program, la suppression des recettes invendues et la création d'une nouvelle recette tous les mois. Nous avons aussi pu ajouter le catalogue d'ingrédients et tout ce qui concerne l'authentification (inscription et connexion) des utilisateurs. Puis, nous avons implémenté les fonctionnalités supplémentaires, c'est-à-dire TooGoodToGo ainsi que la gestion des PartyRecipe.
5) Lors du cinquième sprint, nous avons ajouté les dernières fonctionnalités manquantes et fini celles qui n'avaient été entièrement implémentées dans les sprints précédents. Nous avons aussi pu faire des modifications suite aux conseils que nous avons reçus à travers l'oral de présentation du projet. Puis, nous avons amélioré et ajouté des tests Cucumber pour couvrir au maximum notre projet avant le passage vers Spring.

C) Passage vers Spring

Évoluer vers une approche composants

Pour passer notre projet selon la logique Spring, nous sommes partis sur un projet vierge configuré afin d'avoir un projet Spring. Nous avons réfléchi aux différentes responsabilités que nous pouvions dégager de notre ancien projet et comment regrouper ses responsabilités afin d'en faire des composants, ainsi que dégager les fonctions qui seront rattachées à chacune de ces responsabilités. Ensuite nous avons essayé de dégager les composants "façade", c'est-à-dire ceux qui seront directement déclenchés par l'action d'un humain (*exemple : clic d'un chef, utilisateur, etc..)* et les composants intermédiaires qui eux sont responsables de traitements plus complexes et sont appelés par les composants "façade".

Nous avons ensuite crée un diagramme de composants avec, en première ligne, les composants "façade" puis les composants intermédiaires et leurs interactions avec les repository. La création d'un diagramme de composant nous a grandement aidé à nous diriger vers la version la plus optimale possible. En effet, il nous a permis de constater que certaines de nos dépendances entre composants n'étaient pas logiques. L'ajout des dépendances aux repository a également fait évoluer nos composants et leurs besoins. Grâce au visuel, on se rend vite compte si un composant touche à un repository alors que cela ne parait pas logique au vu de ses responsabilité et que dans ce cas les responsabilités sont mal distribuées.

Après avoir défini toutes les responsabilités et établi lesquelles de nos anciennes fonctions leur correspondaient, nous avons pu commencer à migrer notre code. Nous avons d'abord créer toutes les interfaces et tous les composants liés, puis nous avons migrer le code responsabilité par responsabilité. Nous avons testé indépendamment chaque composant, ce qui nous fait un total de 109 tests de composants.

À chaque étape, nous avons pu poser des questions lors des séances dédiées au projet, ce qui nous a permis de rectifier peu à peu le code. Par la suite, nous avons refait progressivement tous nos tests Cucumber. Nous comptabilisons au total 184 tests, dont 73 tests Cucumber et 111 tests de composants.

Responsabilité de chaque composant

ShopShowCase est un composant dont la responsabilité est, pour chaque magasin, d'exposer les recettes et ingrédients disponibles selon le stock du magasin concerné. Il permet aussi d'exposer les thèmes (selon les cuisiniers disponibles) et les occasions proposées par le magasin. Ainsi, lorsqu'un client passera une commande auprès de ce magasin, il aura seulement à faire un choix parmi une sélection disponible, c'est la "vitrine du magasin".

RecipeFactoryService a pour responsabilité la création de recettes pour la Factory. En effet, une recette peut avoir 3 états (à valider/*TO_BE_VALIDATED*, validée/*VALIDATED*, disponible/*AVAILABLE*). Au travers de ce composant, les recettes seront créées, validées puis ajoutées aux recettes de la Factory. Toutes ces recettes sont dans le recipeRepository et seul leur champ état/*state* est modifié à chaque étape.

FactoryAdministration a pour responsabilité l'embauche du personnel de la Factory, l'ajout de magasins ainsi que de remplir le catalogue des ingrédients disponibles dans la Factory (grâce au composant **CatalogCreation**).

ShopPersonnalService a pour responsabilité l'embauche du personnel propre à chaque magasin (exemple : shopManager, cook).

ShopInformationService a pour responsabilité la modification des informations de chaque magasin (comme la taxe, les horaire, les thèmes et les occasions).

UserAction a pour responsabilité toutes les actions qu'un utilisateur peut faire à travers l'interface de l'application, comme se connecter, s'inscrire, voir son historique ou encore adhérer au programme de fidélité.

CounterCollect permet la récupération des commandes prêtes ainsi que la suppression des clients dans le repository pour correspondre aux normes RGPD (grâce au composant **RgpdService**).

ClientAction a pour responsabilité toutes les actions qu'un client et un utilisateur peut faire à travers l'interface de l'application dans le but de réaliser une commande. Voici les différentes actions et leur ordre durant une commande :

- Il peut choisir un magasin

Une fois le magasin choisi, une liste de recettes provenant de la "vitrine" lui est proposée ainsi que les thèmes, occasions et ingrédients disponibles (utile pour personnaliser une PartyRecipe).

- Il peut maintenant :
  - Ajouter une recette à son panier (dans la quantité qu'il désire). La modification du panier est gérée par le composant CartHandler. Le stock du magasin est alors décrémenté (grâce au composant ShopPantry).
  - Supprimer une recette de son panier, le stock est alors incrémenté.
  - Ajouter une PartyRecipe (personnalisée ou non). La création de la PartyRecipe est gérée par le composant **partyRecipeCreator** et ajoutée au panier par le composant **CartHandler**.

- Ensuite, il peut choisir une heure de retrait, puis un cuisinier est assigné à la commande selon l'heure choisie. Ceci est géré par le composant **Planning**.
- Enfin, il peut valider son panier, ceci est pris en charge par le composant **CartHandler**, puis délégué au composant **PaiementService** afin que le paiement soit effectué et que la commande soit ainsi ajoutée au repository des commandes.
- Les instructions pour la préparation de la commande partent en cuisine avec le composant **Kitchen**, et les recettes concernées par la commande voit leur nombre de ventes augmenter grâce au composant **SalesManager**.

Lorsqu'une commande est validée, le client/utilisateur peut annuler cette commande si elle n'a pas encore été lancée en cuisine, ce qui implique le composant **ManagerOrderService**.

ManagerOrderService est responsable de l'annulation des commandes et de remettre en stock les ingrédients de la commande dans le magasin concerné. Mais il gère aussi tout ce qui concerne une commande non récupérée, comme les notifications (gérées par le composant **Phone**), ainsi que le besoin de passer la commande en obsolète.

SurpriseBasketService a pour responsabilité de créer des paniers TooGoodToGo à partir de commandes obsolètes, et de transmettre des informations à l'API TooGoodToGo. Il notifie également les utilisateurs qui ont voulu recevoir des notifications pour les paniers TooGoodToGo.

Les notifications TooGoodToGo sont implémentées avec le pattern Observer et, afin de réaliser cette tâche, nous avons 2 composants impliqués :

- **ShopSubscriberService** qui permet d'ajouter/supprimer des *subscribers* à un magasin (c'est-à-dire des utilisateurs qui veulent être notifiés si le magasin a de nouveaux paniers surprises). Il permet également de notifier ces *subscribers* grâce au composant **NotificationObserver**.
- **NotificationObserver** qui permet d'ajouter une notification à tous les utilisateurs qui ont voulu souscrire au magasin concerné par le panier surprise.

4. Auto-évaluation

A) Répartition du travail

Concernant la répartition du travail, chaque semaine nous rédigions une liste des issues que nous allions réaliser dans la semaine, correspondant aux User Stories. Nous nous mettions d'accord pour attribuer une note sur 4 pour la difficulté à chaque issue. La note de 1 correspondant à une tâche très simple et/ou très rapide à faire, et la note de 4 correspondant à une tâche plus compliquée et/ou longue à faire. Ainsi, à la fin de chaque séance chaque issue

était attribuée à un membre du groupe de manière à ce que chacun ait le même niveau de difficulté et la même quantité de travail à faire dans la semaine.

Les issues était créées au mieux afin d'éviter les dépendances entre elles, afin que lors de la répartition aucun membre ne dépende du travail d'un autre. Cela n'était pas toujours possible, notamment dans le cas de la gestion du temps où tout le monde avait besoin de la même base pour réaliser sa partie.

Concernant l'implémentation, nous avons créé une branche *develop* dès le début du projet afin de toujours garder la branche main fonctionnelle. Puis, chacun créait une branche dans laquelle il travaillait, en utilisant une branche par issue. Lorsqu'un membre du groupe avait fini son issue et réaliser les tests associés, il demandait la review d'un ou plusieurs autre(s) membres du groupe qui pouvait la merge vers la branche *develop*. Cela permettait d'éviter des potentiels conflits entre les branches et de vérifier que les autres membres du groupe étaient d'accord avec la méthode utiliser ou encore de vérifier que tout fonctionnait bien.

B) Ce qui a fonctionné

La manière de travailler du groupe a plus que bien marché, nous avons trouvé un fonctionnement qui permettait à chacun de trouver sa place dans le projet.

Dans un premier temps, la manière de répartir le travail chaque semaine a permis à chacun des membres de travailler de manière équitable et d'avoir une charge de travail cohérente avec le temps imparti. De plus, nous avons beaucoup profité des sessions de TD pour discuter de nos idées sur la conception du projet et le travail à faire, chacun pouvant donner sa vision du projet ou d'une issue en particulier. Certaines issues plus compliquées ont même été réalisées à plusieurs, ce qui a permis d'aller plus vite et d'oublier le moins d'éléments possibles mais aussi d'avoir différents points de vue sur la manière d'implémenter.

La bonne cohésion au sein du groupe et la communication nous ont aussi beaucoup apporté, notamment dans les situations de dépendance entre plusieurs issues. Nous communiquions pour indiquer au groupe où en était le travail et les éléments apportés qui pouvait servir à d'autres membres.

Ce qui nous a aussi permis d'avancer et d'aller aussi loin est le fait que chaque membre du groupe testait le code associé à son issue sur sa branche avant de le pousser sur la branche commune. Cela permettait de vérifier que tout fonctionnait au fur et à mesure pour éviter des mauvaises surprises lorsqu'une nouvelle fonctionnalité était ajoutée au projet.

C) Les améliorations possibles

Cependant, certains choses auraient pu être faites différemment pour un rendu de projet encore plus complet ou pour un meilleur fonctionnement tout au long de l'implémentation.

Dans un premier temps, avec du recul, nous pensons ne pas nous être assez appuyé sur la partie UML du projet, c'est-à-dire les différents diagrammes. Cela aurait pu nous permettre de mieux visualiser l'architecture de notre projet pour peut-être optimiser notre code.

Aussi, nous avons fait de nombreux tests tout au long du projet, ce qui est positif. Cependant, nous aurions pu mieux nous organiser dans la création de *feature* pour les tests. En effet, chaque membre réalisant des tests pour chacune de ses issues, beaucoup de *features* ont été créées. A la fin du projet, nous nous sommes retrouvés avec beaucoup de *features*, dont certaines qui n'étaient pas forcément utiles, ou encore d'autres qui pouvaient être regroupés. Cela a créé de la duplication de code et nous ajouté une charge de travail supplémentaire afin de nettoyer cela avant le rendu final.

Pour finir, dans un souci de répartition équitable et cohérente du travail, nous avons dû faire des choix concernant les issues à réaliser chaque semaine. C'est ainsi que certaines fois nous avons implémenté des fonctionnalités moins importantes avant d'en implémenter des plus importantes. Nous aurions donc peut-être pu tenter de découper certaines fonctionnalités autrement pour mieux les répartir.

D) Attribution de points

Pour cette partie auto-évaluation, nous nous sommes mis d'accord avec tous les membres du groupes pour attribuer un certain nombre de points à chacun. Cela permet de réaliser une note pour chaque membre du groupe représentative des ressentis de chacun. Ainsi, nous avons pu répartir les 500 points (100 points par membre) de la manière suivante :

Membre du groupe	**Points accordés**
Claire	140
Emma	90
Jules	90
Khadidja	90
Morgane	90

Annexes

Annexe 1 : Diagramme de classes

L'annexe du diagramme de classes inclut le diagramme complet afin d'avoir une vision globale de l'architecture. On y retrouve également le même diagramme coupé en 3 parties afin que ce soit plus lisible.

Figure 5 : Diagramme de classes complet

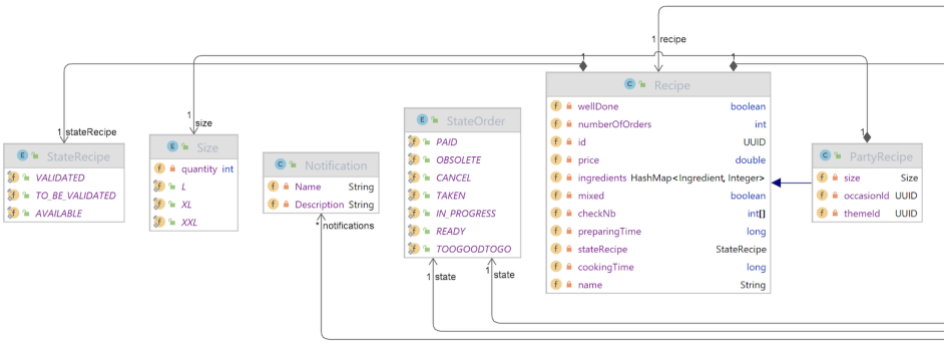

Figure 6: Première partie du diagramme de classes

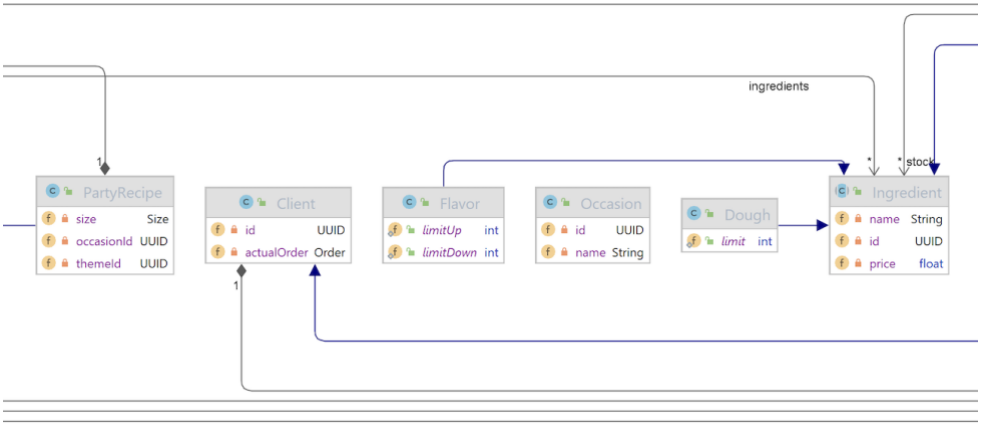

Figure 7: Seconde partie du diagramme de classes

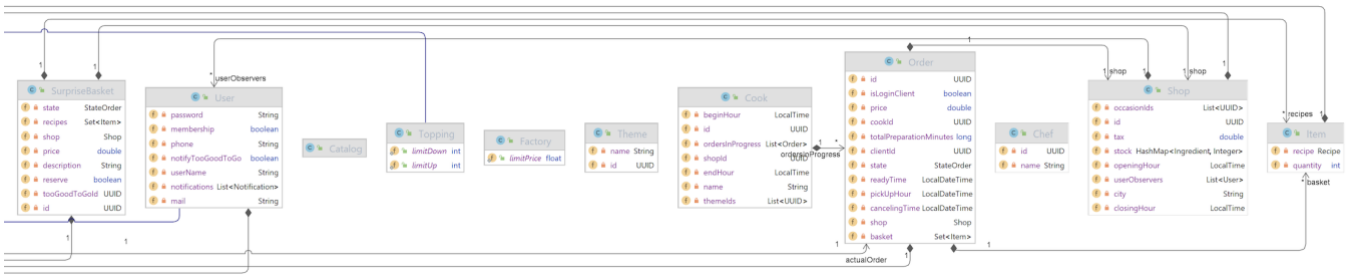

Figure 8 : Troisième partie du diagramme de classes