Bonnet Kilian - Imami Ayoub - Le Bihan Léo - Karrakchou Mourad - Derain Émile

Conception logicielle

The Cookie Factory

Team-K

I. Diagrammes

Vous pouvez retrouver les différents diagrammes ci-dessouus en qualité original à cette adresse : https://github.com/PNS-Conception/cookiefactory-22-23-k/tree/main/docs

Diagramme de cas d'utilisation

Les cas d'utilisation n'ayant pas de liaison avec un acteur sont des cas exécutés par le système lui-même. Par exemple, "Verrouiller le cookie" est réalisé par le système afin de réserver le cookie à ce client dans le but de laisser le temps au client de finaliser sa commande.

Une commande ne peut être annulée uniquement si sa préparation n'a pas encore commencé. Lorsque une nouvelle recette est ajoutée, le système vérifie si ses ingrédients sont bien présents dans le catalogue des partenaires, dans le cas contraire la recette n'est pas ajoutée, comme indiqué dans le cahier des charges.

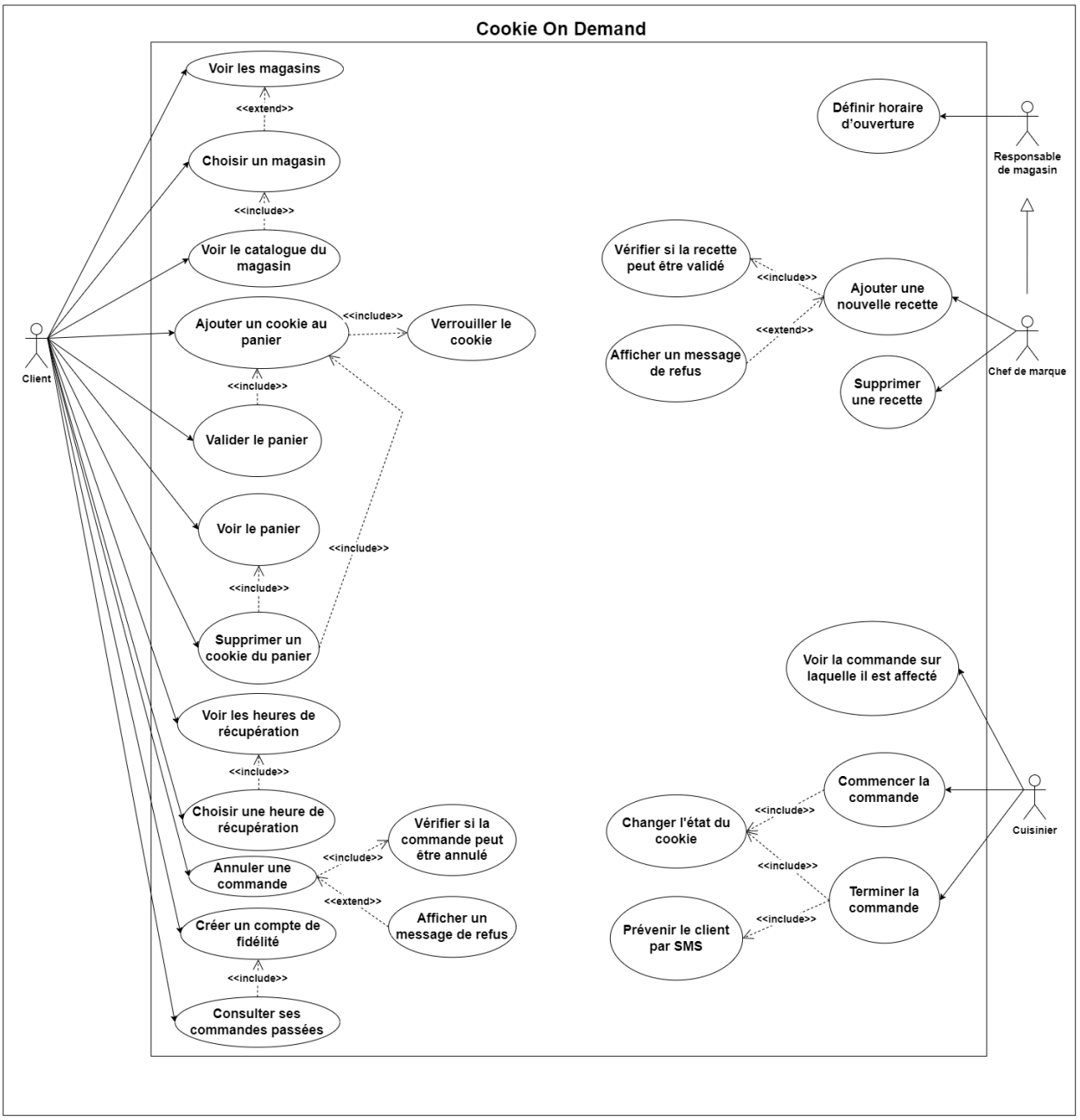


Diagramme de classes

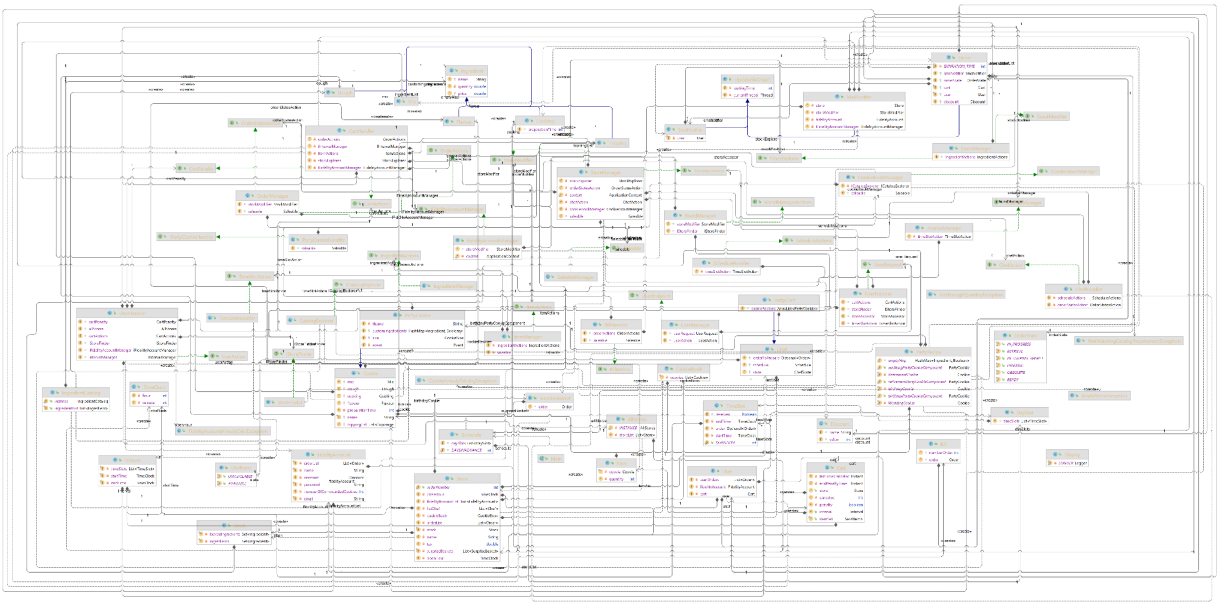
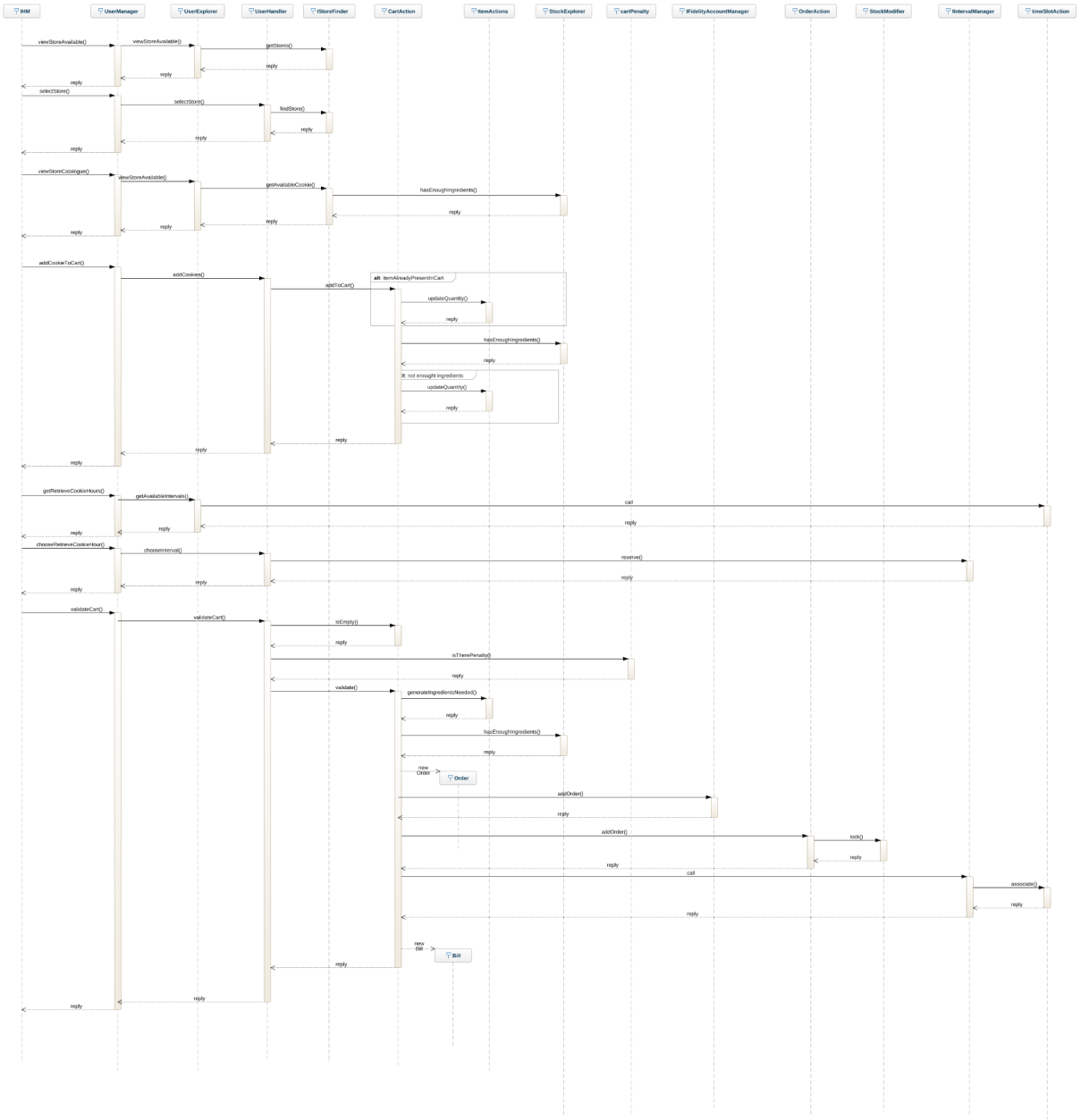


Diagramme de séquence



II. Patron de conception

Singleton

Le design pattern Singleton est utilisé pour garantir qu'une classe n'a qu'une seule instance et qu'il existe un point d' accès global à cette instance. Cela peut être utile dans plusieurs cas de figure :

La classe *IngredientCatalog* est le classe représentant la liste des ingrédients présents dans le catalogue en ligne. En utilisant le patron de conception Singleton, nous nous assurons ainsi qu'il n'y a qu'une seule instance de cette classe et que tous les objets de l'application utilisent la même instance.

La classe *StoreLocation* est la classe représentant la liste des différents magasins. Cette classe, à l'instar de *IngredientCatalog*, est une classe qui représente un objet unique. Cet objet doit également être accessible depuis n'importe quel endroit de l'application. Le pattern Singleton vous permet de définir un point d'accès global à cet objet, ce qui peut simplifier la gestion de la recherche de magasin dans l'application Cookie On Demand.

StoreLocation et *IngredientCatalog* sont des classes nécessitant être instanciées et initialisées avant d'être utilisées. Nous devons par exemple remplir *StoreLocation* de quelques magasins ou encore *IngredientCatalog* qui nécessite quelques ingrédients au préalable. En utilisant le pattern Singleton, nous nous assurons que ces instances sont créées et initialisées avant d'être utilisées.

Façade

Le patron de conception Façade est utilisé pour simplifier l'interface d'un système complexe en fournissant une interface unique qui offre un accès simplifié aux fonctionnalités du système. Cela a pu s'avérer particulièrement utile pour les interfaces *UserRequest* et *UserAction* qui permettent de centraliser les différentes interactions qu'un utilisateur peut avoir avec notre système.

Simplification de l'interface : en utilisant une classe Façade, nous pouvons ainsi offrir une interface simple et uniforme pour accéder aux fonctionnalités de notre système. Cela rend le code plus facile à comprendre et à utiliser pour les personnes en charge d'utiliser notre système.

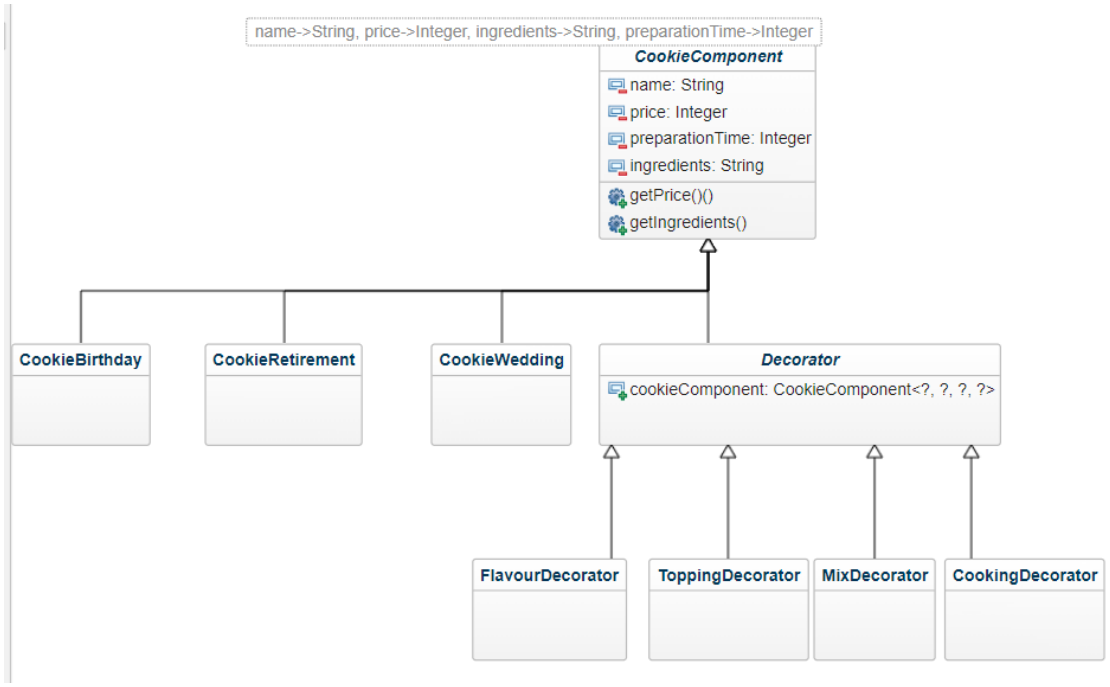
Amélioration de la lisibilité du code : en utilisant une classe Façade, nous regroupons ainsi différents appels à différentes parties de notre système dans un seul endroit. Cela permet de rendre le code plus lisible et plus facile à comprendre.

Meilleure flexibilité : en utilisant une classe Façade, nous pouvons changer l'implémentation de notre système sans affecter une potentielle application qui serait branchée à notre système. Cela signifie que nous pouvons ajouter de nouvelles fonctionnalités ou modifier l'implémentation existante sans avoir à modifier le code des clients ou de toutes autres personnes s'étant branché à notre système.

Décorateur

Afin d'implémenter l'extension de code "Party Cookie", nous avons pensé à utiliser le patron de conception décorateur. Cette extension permet au client de commander des cookies spéciaux de tailles variables pour différentes occasions avec différents thèmes. Le magasin propose aussi de créer des Party Cookie à partir de cookies existants en ajoutant ou en enlevant des ingrédients à volonté. C'est donc le fait de pouvoir ajouter / supprimer des ingrédients autant que l'utilisateur le souhaite qui nous a conduit sur la piste du patron de conception Décorateur. De cette manière on pourrait créer des Party Cookie simplement et leurs compositions ainsi que leurs prix seraient mis à jour dynamiquement à chaque modification.

Diagramme de classe du design pattern décorateur:



Cependant après le début de l'implémentation nous avons fait le choix de ne pas implémenter le pattern décorateur. En effet après réflexion nous avons décidé d'opter plutôt pour de l'héritage car cela nous permet d'avoir une implémentation plus légère pour le même résultat.

III. Rétrospective

Passage à une approche à composant

Pour faire évoluer la conception de notre application vers une approche à composants avec Spring, nous avons dans la majeure partie des cas suivi les étapes suivantes :

Rechercher des classes à transformer en composant

1. Identifier les parties de notre application qui étaient réutilisées dans plusieurs parties de l'application.

2. Rechercher les parties du code qui sont complexes et qui peuvent être divisées en sous-parties indépendantes afin qu'elles puissent être mises à jour ou modifiées indépendamment les unes des autres afin d'améliorer la modularité.

3. Rechercher les parties du code qui sont difficiles à tester ou qui nécessitent de nombreux tests.

Transformer ces parties de code en interface afin de créer afin d'obtenir la liste des méthodes fournies par nos composants.

Créer des modules indépendants pour chacune de ces parties, en utilisant Spring pour gérer les dépendances et l'injection de dépendances entre les modules.

Tester chaque module indépendamment pour nous assurer que ces derniers fonctionnent correctement et qu'ils puissent être réutilisés facilement.

Combiner les modules en utilisant Spring pour gérer les communications entre les modules et pour injecter les dépendances nécessaires dans chaque module.

Tester l'application dans son ensemble à l'aide des scénarios Gherkins afin de s'assurer qu'elle fonctionne correctement.

Remarques

Nous estimons que la qualité de notre code est satisfaisante. Actuellement, nous sommes à 92% de couvertures de classe, soit 80% de lignes couvertes par nos tests. En ce qui concerne nos composants SpringBoot, nous sommes parvenus à obtenir une couverture de 100% des classes et de 86% des lignes. Le code étant couvert par des tests unitaires JUnit, ainsi que, et majoritairement par nos scénarios Gherkins.

L'approche en composants consiste à créer des modules logiciels réutilisables et indépendants qui peuvent être combinés pour créer une application complète. Durant notre intégration en utilisant SpringBoot nous avons relevés les avantages suivant :

Réutilisabilité : Nous avions dans notre code certaines parties dupliquées : nous pouvons par exemple citer la méthode pour récupérer un magasin qui avait été écrite deux fois. Grâce à l'approche orientée composant, certains de nos composants peuvent être réutilisés dans différentes applications, ce qui peut réduire le temps et les coûts de développement.

Modularité : Après avoir eu la nécessité d'ajouter de nouvelles fonctionnalités à nos composants SpringBoot, nous avons constaté que ces composants étaient indépendants les uns des autres et pouvaient être modifiés ou mis à jour sans affecter le reste de l'application. Cela a rendu le développement et la maintenance de notre application plus facile.

Responsabilités

ItemManager

Le composant *ItemManager* a pour responsabilité de gérer les actions sur les articles (items). Un article est une association entre un cookie et une quantité. Ces actions sur les articles sont mettre à jour la quantité d'un article, générer les ingrédients nécessaires pour un article ou pour une liste d'articles. Ce composant fournit des méthodes et ne nécessite aucune injection de composant en contrepartie.

CatalogExplorer

Le composant *CatalogExplorer* a pour responsabilité de gérer l'exploration du catalogue d'ingrédients commun à tous les magasins de CookieFactory. En utilisant ces méthodes, il est possible de vérifier la disponibilité d'un ingrédient dans le catalogue d'ingrédients et de récupérer les ingrédients nécessaires pour la préparation d'un cookie. Ce composant fournit des méthodes et ne nécessite aucune injection de composant en contrepartie.

CartHandler

Le composant *CartHandler* gère les actions du panier d'achat, comme ajouter ou supprimer des articles du panier, valider le panier et annuler une commande. Afin de s'assurer qu'il y ait les ingrédients nécessaires dans un magasin pour pouvoir ajouter un cookie *CartHandler* nous avons besoin de lui fournir *StockExplorer*. Etant donné que *CartHandler* à une méthode pour valider le panier et donc de créer une commande, nous avons également besoin de lui fournir *OrderActions*. Afin de pouvoir manipuler de items : c'est-à-dire ajouter et supprimer des items du panier, nous avons besoin de fournir à notre component l'interface *ingredientActions*.

CookieBookManager

Le composant *CookieBookManager* a comme responsabilité de gérer un livre de recettes de cookies. Elle fournit des méthodes qui permettent d'ajouter ou de supprimer une recette de cookies dans le livre de

recettes, de récupérer la liste des cookies disponibles dans le livre de recettes ou encore de récupérer une recette de cookie en particulier en fonction de son nom. Le composant s'appuie sur l'interface *CatalogExplorer* pour vérifier que les ingrédients d'une recette de cookie sont bien présents dans le catalogue d'ingrédients avant de l'ajouter au livre de recettes. Si un ingrédient n'est pas présent dans le catalogue, une exception *NotMatchingCatalogRequirementException* est levée. Si la recette de cookie que l'on essaie d'ajouter au livre de recettes existe déjà, une exception *CookieAlreadyExistingException* est levée.

IngredientManager

La responsabilité de cette classe est donc d'apporter des fonctionnalités de manipulation d'ingrédients. Plus précisément, elle propose deux méthodes : *multiplyQuantity* et *increaseQuantity*. La première permet de définir la quantité d'un ingrédient en multipliant sa quantité initiale par un facteur donné, tandis que la seconde permet d'ajouter une quantité donnée à la quantité actuelle de l'ingrédient.

OrderManager

La responsabilité de cette classe est d'apporter des fonctionnalités de manipulation de commandes. Cette classe dépend de plusieurs autres composants : *StockManager*, *CartHandler* et *ChefAction*. Elle utilise ces composants pour réaliser certaines de ses fonctionnalités, comme la réservation d'ingrédients dans le stock ou la gestion de la préparation des commandes par les chefs.

StockManager

Le composant *StockManager* gère l'exploration et la modification du stock d'ingrédients dans les magasins couverts par notre application. Il implémente les interfaces *StockExplorer* et *StockModifier*, qui définissent respectivement des méthodes permettant de rechercher et de vérifier la disponibilité d'ingrédients dans le stock, ainsi que des méthodes pour bloquer, débloquer et ajouter des ingrédients au stock.

ChefHandler

Le composant *ChefHandler* est un composant qui dépend des composants *ScheduleActions* et *OrderStatesAction*. Elle a pour responsabilité de gérer les actions sur les chefs d'un magasin, comme mettre à jour l'emploi du temps d'un chef, associer un commande à un chef, vérifier si un chef est disponible ou pas, démarrer le travail d'un chef sur une commande, récupérer les plages horaires disponibles pour un chef et mettre à jour la commande en cours d'un chef.

TimeSlotHandler

Le composant *TimeSlotHandler* gère les actions liées aux créneaux horaires dans notre application. Il implémente l'interface *TimeSlotAction*, qui définit des méthodes permettant de récupérer des intervalles disponibles dans un horaire, de vérifier si un créneau horaire est disponible, de dissocier un créneau horaire d'une commande et d'associer un créneau horaire à une commande. Le composant dépend du composant *ChefAction*, qui est injecté dans l'instance du composant via l'annotation @Autowired.

IntervalManager

Le composant *IntervalManager* gère la réservation, la validation et la libération d'intervalles de temps des cuisiniers dans notre application. Il implémente l'interface *IntervalManager*, qui définit des méthodes permettant de réserver, valider et libérer des intervalles. Ce composant dépend du composant *TimeSlotHandler*, qui est injecté dans l'instance du composant via l'annotation *@Autowired*. Cela signifie que le composant *IntervalManager* dépend d'un bean Spring capable de fournir une implémentation de *TimeSlotAction*, qui doit être enregistré dans le contexte de Spring avant que le composant puisse être utilisé.

ScheduleHandler

Le composant *ScheduleHandler* gère les actions de planification dans une application. Il implémente l'interface *ScheduleActions*, qui définit des méthodes permettant de récupérer des intervalles disponibles dans un horaire, de récupérer un créneau horaire d'un horaire, et de récupérer la commande à préparer pour un moment donné dans un horaire. Le composant dépend du composant *TimeSlotAction*, qui est injecté dans l'instance du composant via l'annotation *@Autowired*. Cela signifie que le composant *ScheduleHandler*

dépend d'un bean Spring capable de fournir une implémentation de *TimeSlotAction*, qui doit être enregistré dans le contexte de Spring avant que le composant puisse être utilisé.

StoreFinder

Le composant *StoreFinderest* une implémentation de l'interface *StoreFinder*. Cette interface permet de trouver des magasins par leur nom et de récupérer la liste de tous les magasins disponibles. Le composant *StoreFinder* est initialisé avec une liste de trois magasins, situés dans les villes d'Antibes, de Nice et de Sophia. La méthode *findStore* parcourt cette liste et renvoie le magasin qui a le nom spécifié en entrée, s'il existe. Si aucun magasin n'a ce nom, la méthode lance une exception *InvalidStoreException*. La méthode *getStores* renvoie simplement la liste de tous les magasins disponibles .Le composant est annoté avec *@Component*, ce qui signifie qu'il est géré par Spring et peut être injecté dans d'autres composants ou beans de l'application grâce à l'annotation *@Autowired*.

BillHandler

Le composant *BillHandler* est une implémentation de l'interface BillAction. Cette interface permet de calculer le prix total d'une commande et de générer un reçu pour une commande donnée. Le composant *BillHandler* est initialisé avec une instance de l'interface *OrderActions*, qui est injectée grâce à l'annotation *@Autowired*. La méthode *calculateTotalPrice* utilise cette instance pour appeler la méthode *computeTotalPrice* de l'interface *OrderActions* et renvoyer le prix total de la commande. La méthode *returnBill* génère un reçu en chaîne de caractères pour la commande donnée. Le reçu inclut le nom du magasin, la liste des produits et leur prix, ainsi que le prix total de la commande. Si la commande possède un discount, le reçu inclut également le nom et la valeur du discount.

StoreManager

La classe *StoreManager* implémente deux interfaces: *StoreModifier* et *StoreAccessor*. Ces interfaces définissent les méthodes qui peuvent être appelées pour modifier ou accéder à aux données concernant les magasins couverts par notre application. La classe *StoreManager* a plusieurs responsabilités. Elle fournit une implémentation de méthodes qui permettent de changer l'heure d'ouverture et de fermeture d'un magasin, de récupérer les cookies disponibles dans un magasin, d'ajouter ou de retirer des chefs ou des cookies dans un magasin, de changer le prix des ingrédients dans un magasin, d'ajouter des comptes de fidélité dans un magasin et de traiter les commandes passées dans un magasin. En outre, la classe *StoreManager* dépend de plusieurs autres classes pour accomplir ces tâches. Elle utilise une instance de *ChefAction* pour mettre à jour l'horaire des chefs d'un magasin, une instance de *StockExplorer* pour vérifier si un magasin a suffisamment d'ingrédients pour préparer un cookie, une instance de *StoreFinder* pour trouver un magasin, une instance de *CookieBookManager* pour gérer le livre de recettes d'un magasin et une instance de *OrderStatesAction* pour traiter les états des commandes d'un magasin.

UserExplorer

Le composant *UserExplorer* à pour objectif de récupérer toutes les informations à présenter à l'utilisateur. Il propose donc plusieurs méthodes qui permettent à un utilisateur de naviguer dans l'application et d'interagir avec elle. Le composant va communiquer avec les interfaces suivantes : *CartActions* afin de récupérer les informations du panier, *TimeSlotAction* afin de connaître les heures de récupération disponible au client, le *IStoreFinder* afin de connaître l'ensemble des magasin de la firme auprès desquels le client pourrait passer commande, et enfin le *StoreAccessor* afin de récupérer le catalogue de cookie présenté par un magasin.

UserHandler

Le composant UserHandler représente toutes que l'utilisateur va pouvoir réaliser.
Ce composant contient plusieurs méthodes permettant à l'utilisateur de faire différentes opérations telles que : Ajouter ou retirer des cookies de son panier en utilisant les méthodes addCookies et removeCookies. Choisir une boutique en utilisant la méthode selectStore qui utilise l'interface IStoreFinder pour trouver la boutique correspondant à un nom donné. Réserver un intervalle de temps en utilisant la méthode chooseInterval qui utilise l'interface IIntervalManager pour réserver l'intervalle. Valider le panier en utilisant la méthode validateCart qui vérifie si le panier est vide ou s'il y a une pénalité avant de valider le panier. S'inscrire à un compte de fidélité en utilisant la méthode subscribeToFidelityAccount qui crée un

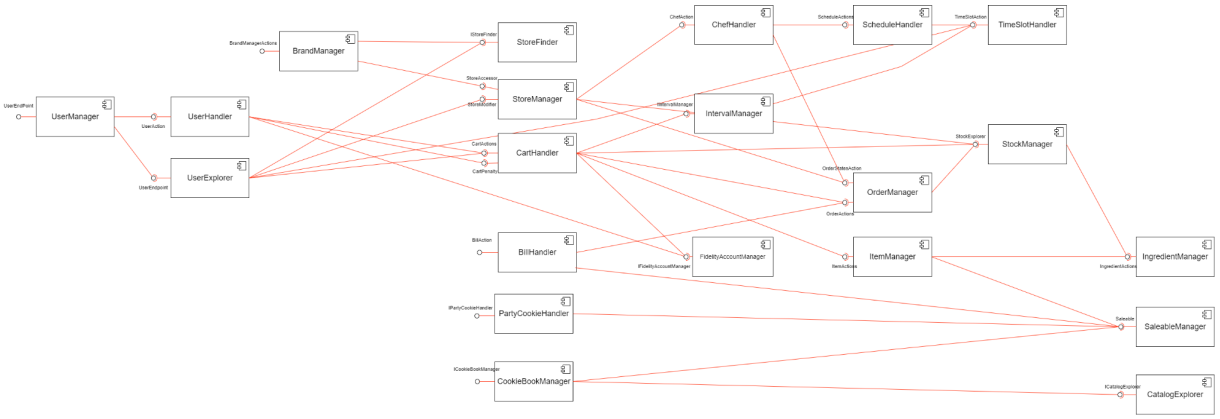
nouveau compte de fidélité pour l'utilisateur. Annuler une commande en utilisant la méthode cancelOrder qui vérifie si la commande peut être annulée avant de l'annuler

UserManager

Le composant *UserManager* de Spring Boot est une classe qui implémente l'interface *UserEndpoint* et qui s'occupe de gérer les actions effectuées par un utilisateur et de répondre aux requêtes de l'utilisateur. Il est une façade, il permet de fournir des points d'entrée simplifiée afin d'utiliser notre système.

Ce composant contient plusieurs méthodes qui permettent à l'utilisateur de faire différentes opérations telles que : Afficher les boutiques disponibles en utilisant la méthode viewStoreAvailable qui utilise l'interface UserRequest pour récupérer la liste des boutiques disponibles. Choisir une boutique en utilisant la méthode selectStore qui utilise l'interface UserAction pour sélectionner la boutique. Afficher le catalogue de la boutique en utilisant la méthode viewStoreCatalogue qui utilise l'interface UserRequest pour récupérer le catalogue de la boutique sélectionnée par l'utilisateur. Ajouter un cookie au panier en utilisant la méthode addCookieToCart qui utilise l'interface UserAction pour ajouter le cookie au panier de l'utilisateur. Valider le panier en utilisant la méthode validateCart qui utilise l'interface UserAction pour valider le panier de l'utilisateur. Retirer un cookie du panier en utilisant la méthode removeCookieFromCart qui utilise l'interface UserAction pour retirer le cookie du panier de l'utilisateur. Récupérer les heures de récupération des cookies en utilisant la méthode getRetrieveCookieHours qui utilise l'interface UserRequest pour récupérer les heures de récupération disponibles pour la boutique sélectionnée et le panier de l'utilisateur. Choisir une heure de récupération des cookies en utilisant la méthode chooseRetrieveCookieHour qui utilise l'interface UserAction pour réserver l'heure de récupération. Annuler une commande en utilisant la méthode cancelOrder qui utilise l'interface UserAction. Créer un compte de fidélité en utilisant la méthode createFidelityAccount qui utilise l'interface UserAction afin de créer le compte de fidélité pour l'utilisateur. Récupérer les commandes précédentes de l'utilisateur en utilisant la méthode getPreviousOrders qui utilise l'interface UserRequest pour récupérer l'historique des commandes du compte de fidélité de l'utilisateur.

Diagramme de Composants



IV. Auto-évaluation

Globalement, lors des réunions du mardi, nous vérifiions l'avancement du projet et s'il y avait des implémentations à modifier ou à améliorer. Nous corrigions ensuite le code si cela était nécessaire.

Nous avons divisé notre Kanban en 3 parties: In progress, Tests et Done. Chacun choisissait une issue et la déplaçait donc dans "In progress" en faisant des commits au fur et à mesure que l'issue avançait. Puis, une fois l'implémentation faite, la personne chargée de l'issue la déplaçait dans "Tests" et effectuait les tests sur cette partie du programme. Une fois la nouvelle fonctionnalité testée, elle est déplacée dans dans "Done" et alors seulement nous pouvions close l'issue.

Pendant la semaine, tous les membres étaient conscients du travail à faire et chacun travaillait et gérait son temps comme bon lui semblait. Nous nous posions souvent des question entre nous pour connaître l'avis de chacun, obtenir de l'aide ou une information.

Nous nous sommes délégués le travail de façon à ce que chaque membre du groupe travaille sur les différentes parties, UML, Gherkin, Design Pattern et passage à Spring. De ce fait chaque membre de l'équipe connaissait les différents aspects de notre programme ainsi que les différentes notions abordées en cours.

Nom	**Note**
Mourad	100
Kilian	100
Ayoub	100
Emile	100
Léo	100