POLYTECH NICE SOPHIA

UNIVERSITÉ CÔTE D'AZUR

Rapport
Conception logiciel - Cookie Factory

Membres du groupe :

Zinedine CHELGHAM
Thomas BILLEQUIN
Tobias ERPEN
Benoit GAUDET
Ayman BASSY

17/12/2022

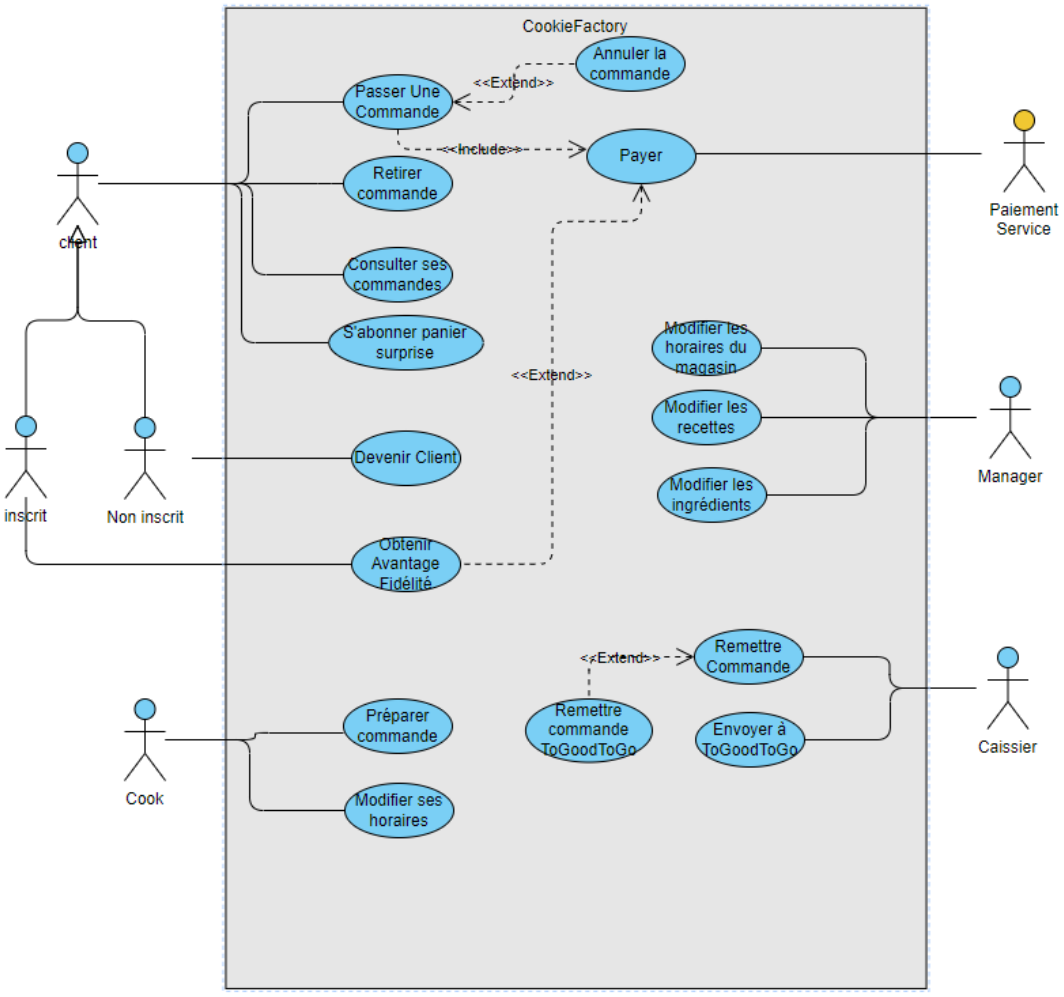
POLYTECH NICE SOPHIA

UNIVERSITÉ CÔTE D'AZUR

Table des matières

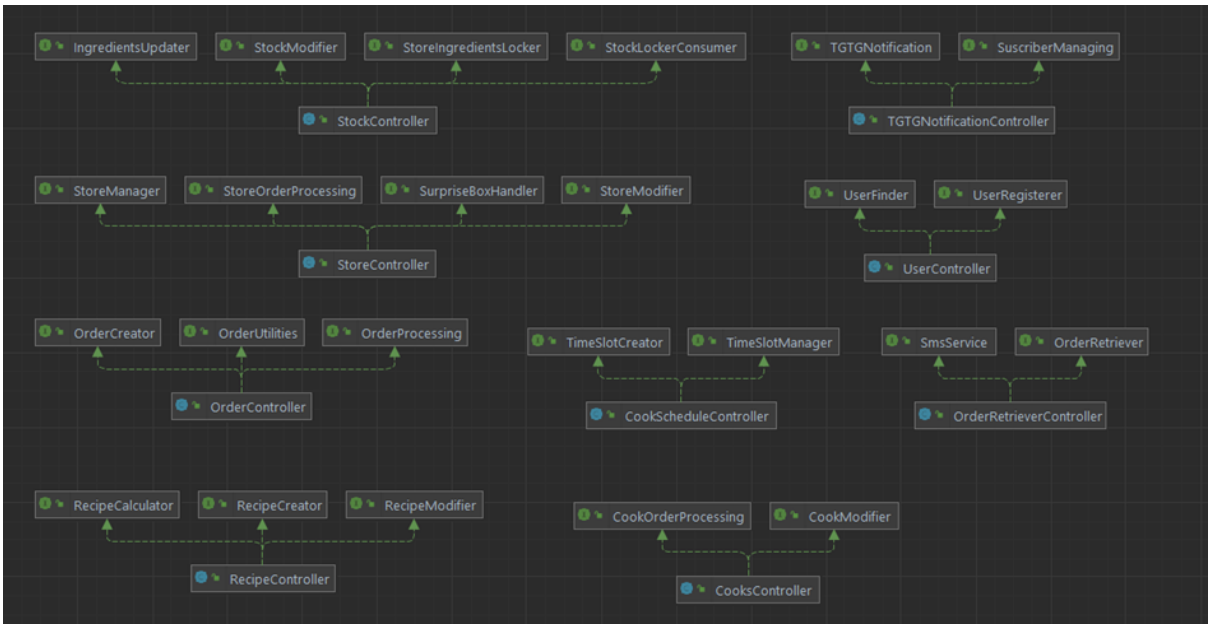
UML

Diagramme de cas d'utilisation

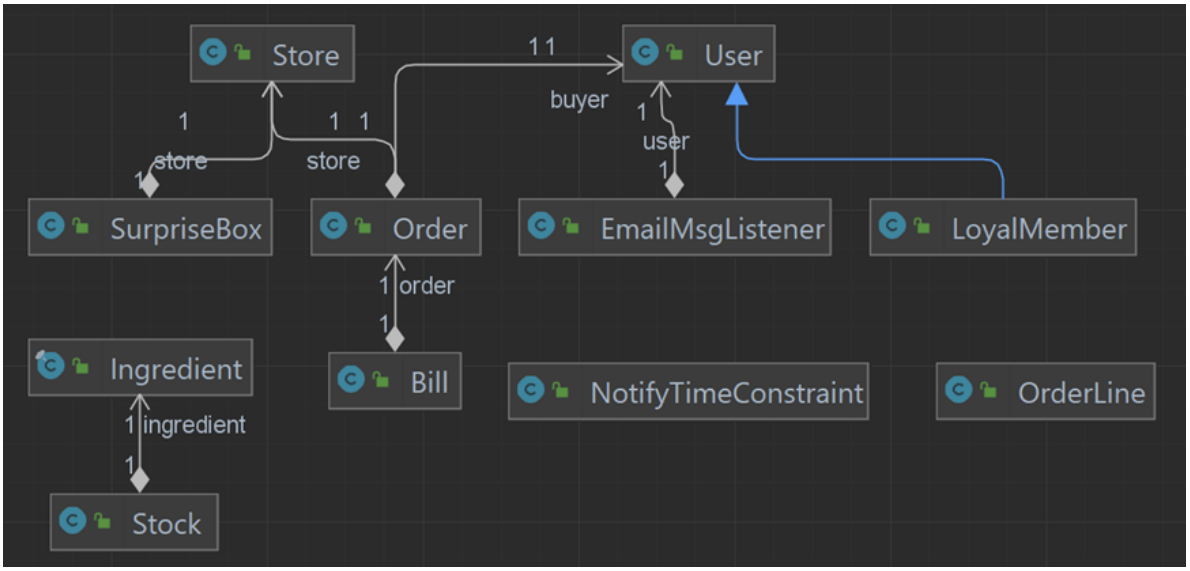


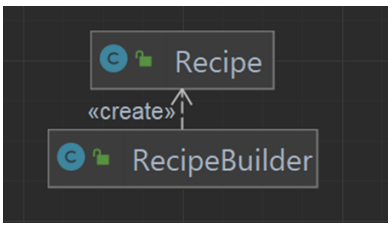
Diagrammes de classe

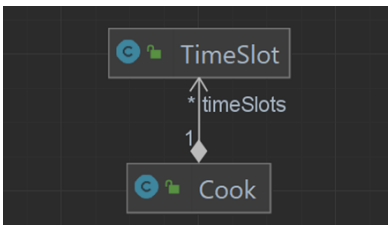
Composants et interfaces



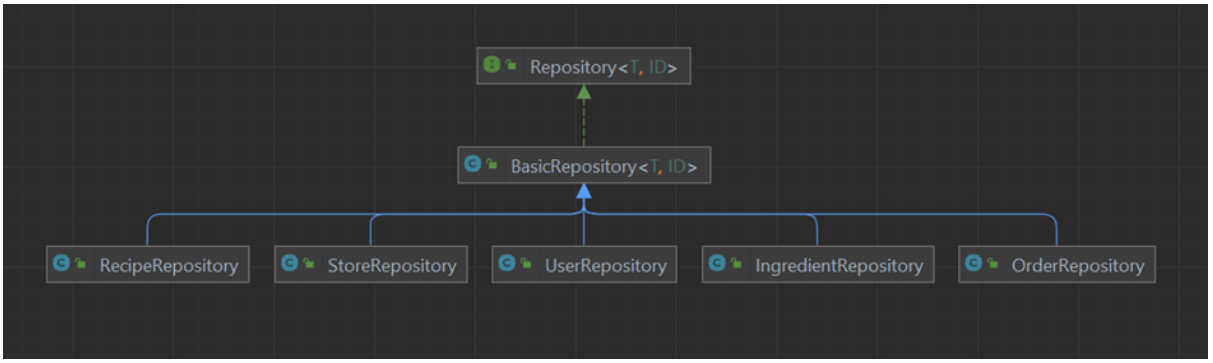
Model







Repositories



Services

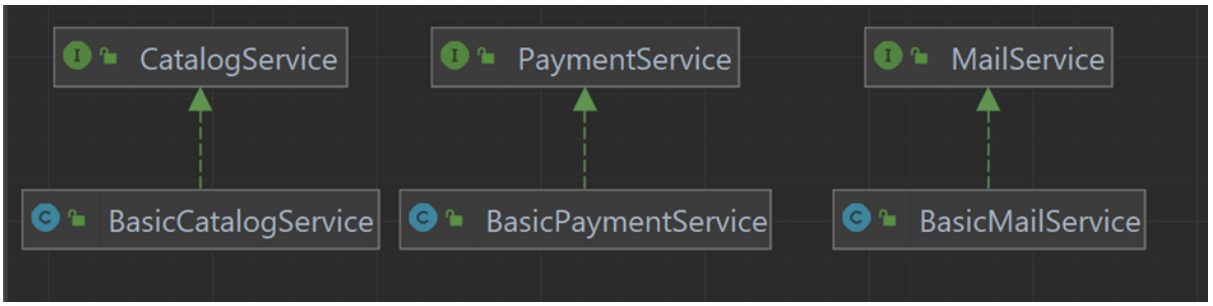
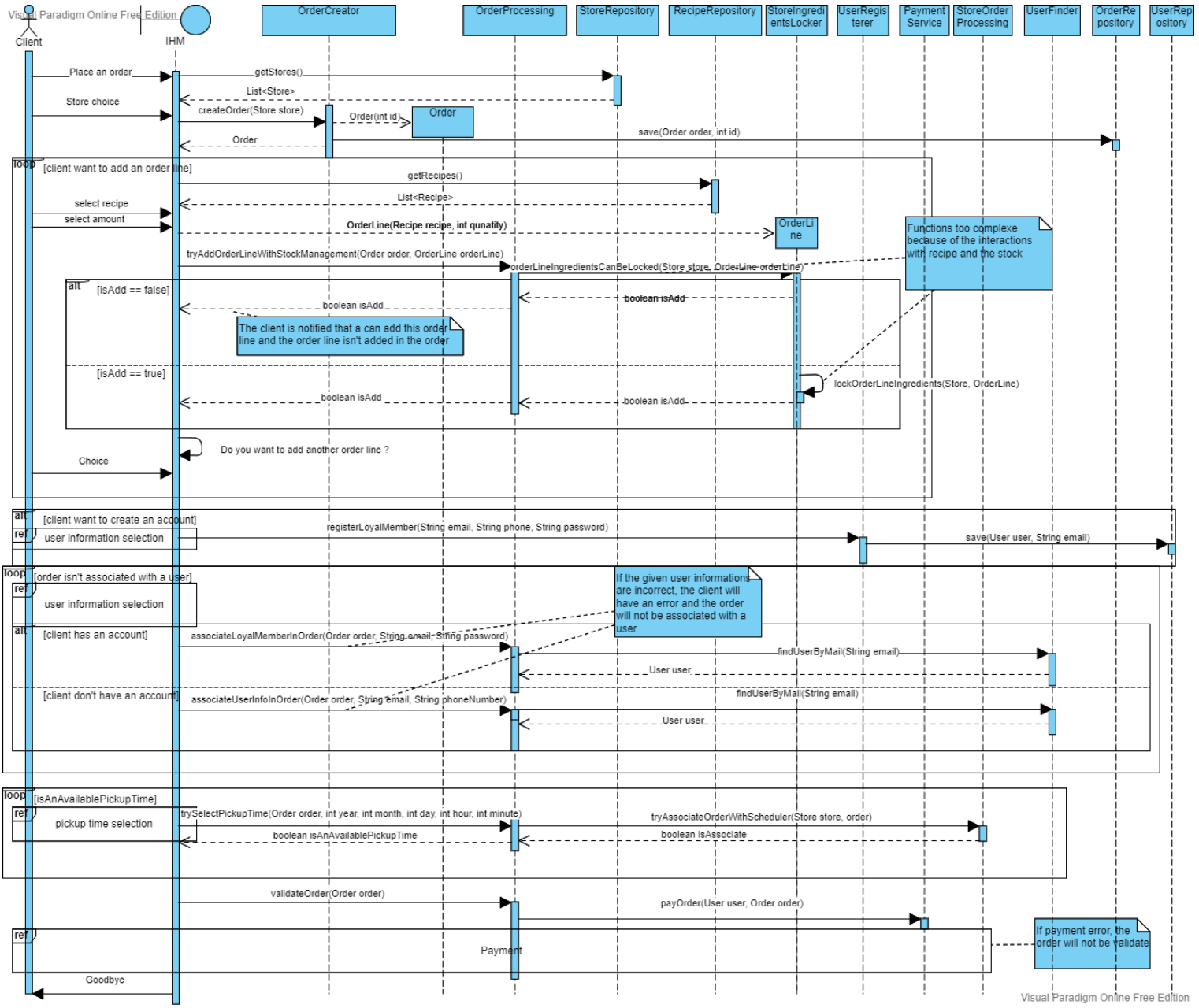


Diagramme de séquence (Passer une commande)



Patrons de conception appliqués

Builder

Le pattern Builder a été utilisé pour la création de recettes de cookie. Ainsi, nous considérons que la création d'un cookie est basée sur une recette par défaut et que le builder nous permet de modifier ou rajouter des ingrédients à cette recette.

Cela permet de faciliter la création de nouvelles recettes en ajoutant leurs spécificités. L'utilisation du builder permet aussi la modification de la recette car celle-ci peut être retransformée en builder pour y modifier des ingrédients.

Cette utilisation du pattern permet de facilement gérer les modifications des recettes et permet aussi aisément de prendre en compte les changements impliqués par l'ajout des parties cookies. Il a simplement fallu ajouter un attribut size qui est par défaut à Normal et que l'on peut redéfinir grâce au builder. La logique change ensuite suivant la taille des cookies.

Observer

Nous avons décidé d'utiliser le pattern Observer pour gérer les notifications dans le cadre de l'intégration de "ToGoodToGo". En effet, ce choix semblait pertinent pour plusieurs raisons, la raison évidente étant le besoin de tenir à jour des "abonnés". Il permet aussi de :

- ➢ Séparer les préoccupations : Le pattern Observer permet de séparer la logique de notification et les détails de l'envoi de l'e-mail. Cela signifie que la logique de notification peut être mise en œuvre sans avoir à se soucier de la façon dont l'e-mail sera effectivement envoyé.

- ➢ Anticiper l'évolution: Si vous souhaitez ajouter de nouvelles façons de notifier les utilisateurs (par exemple, en utilisant des notifications push ou en envoyant des SMS), le pattern Observer vous permet de le faire en ajoutant simplement de nouvelles classes "observatrices" sans avoir à modifier la logique de notification existante.

L'implémentation suit les grandes lignes suivantes:

- Une classe *EmailMsgListener* qui prend un Utilisateur en paramètre et qui implémente une interface "EventListener" (pour pouvoir éventuellement gérer plusieurs types "d'écouteurs".
- Un composant *TGTGController* qui implémente 2 interfaces, une pour gérer les "abonnés" et l'autre pour les notifier.

Proxy

Ce choix est venu après l'introduction au framework Spring en voyant l'exemple de la *SimpleCookieFactory*. Nous avons alors pu constater que l'utilisation de ce pattern constitue une bonne pratique lorsqu'une application Spring veut communiquer avec des services externes.

Son utilisation est intéressante car il permet de :

- Contrôler l'accès : Un proxy peut être utilisé pour contrôler l'accès à un composant, en implémentant des vérifications d'autorisation ou en masquant certaines fonctionnalités du composant original.

- Modifier un comportement : Un proxy peut être utilisé pour ajouter des comportements supplémentaires à un composant, tels que le suivi des performances ou la journalisation des appels de méthode.

- Gérer les ressources : Un proxy peut être utilisé pour gérer les ressources, telles que les connexions à une base de données ou les objets de synchronisation, de manière transparente pour l'utilisateur final.

L'implémentation se résume ainsi :
Création d'un composant qu'on a appelé *ToGoodProxy* qui implémente l'interface *TGTGSurpriseBox* qui contient toute les méthodes qui résument l'interaction avec ToGoodToGo. Le proxy possède également un attribut *SurpriseBoxHandler*
pour agir sur l'état des paniers surprises.

Patrons de conceptions non appliqués

Factory : Le pattern Factory est un design pattern qui permet de centraliser la logique de création d'objets dans une seule classe, appelée fabrique. L'objectif est de séparer la logique de création des objets de leur utilisation, ce qui permet de rendre le code plus flexible et plus facile à maintenir.
Le pattern Factory est utilisé lorsqu'on a besoin de créer des objets de différents types, mais sans avoir à connaître les détails de création de ces objets. Dans notre application, la partie qui aurait été la plus susceptible d'avoir besoin de ce pattern aurait été pour gérer les "Party cookies" avec leurs différents paramètres mais on a décidé d'adapter notre Builder pour cette nouvelle fonctionnalité car les changements à effectuer était assez minimes et ça permettait éviter d'alourdir notre architecture déjà bien chargé.

Decorator : Le pattern Décorateur est utilisé pour ajouter de nouvelles responsabilités à un objet de manière dynamique. Il est utile lorsqu'on a besoin de personnaliser un objet sans

avoir à créer une nouvelle sous-classe pour chaque combinaison de personnalisations.
Ce pattern aurait pu être intéressant pour gérer les différents types de notifications qu'un utilisateur pouvait recevoir par mail, téléphone, ou par l'application elle même pour le cas ToGoodToGo par exemple. Mais dans notre cas, l'utilisateur n'a jamais besoin de recevoir plusieurs types notifications au même instant. Donc nous n'avons pas eu ce besoin (à ce stade de la conception) qui est d'attacher de manière dynamique de nouvelles responsabilités à nos différents notificateurs.

Passage à Spring

Étapes pour mener à bien la migration:

Pour faire évoluer la conception de notre application vers une approche basée sur les composants avec Spring, nous avons suivi les étapes suivantes :

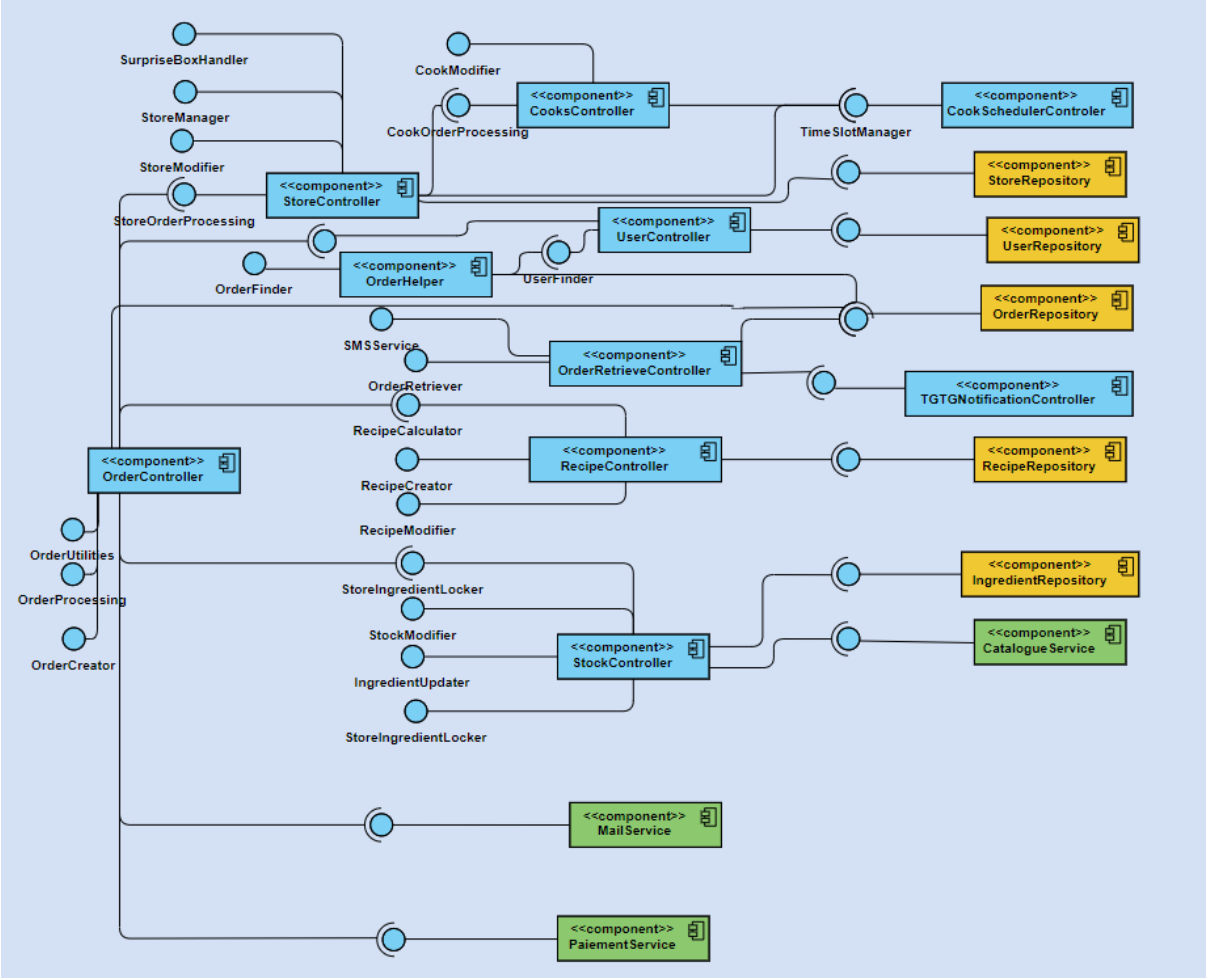
- Analyse de l'application : Nous avons analysé l'application existante pour comprendre le fonctionnement de l'application active et de ses différentes parties. Nous avons identifié les principaux blocs fonctionnels et les interactions entre ces différentes parties.

- Trouver et définir les interfaces : À l'aide des informations collectées lors de l'analyse de l'application, nous avons défini les interfaces ayant un impact du point de vue métier (des actions concrètes sur le système. Ex: ajouter des items à une commande)

- Définir nos composants : à l'aide des informations collectées lors de l'analyse de l'application, nous avons divisé l'application en différents composants en fonction de leurs responsabilités et de leurs rôles au sein de l'application. Ces composants implémentent donc nos interfaces précédemment définies afin de séparer la logique de chaque composant de son implémentation concrète.

- Implémentation des composants : Nous avons implémenté chaque composant selon les spécifications définies dans l'interface. Nous avons également utilisé Spring pour gérer l'injection de dépendances entre les composants, ce qui nous a permis de découpler davantage la logique de chaque composant.

Diagramme de composants



Focus sur nos composants

CookScheduleController : La responsabilité de ce composant est de gérer l'agenda d'un cuisinier dans un magasin. L'interface *TimeSlotCreator* permet de créer la représentation de la préparation d'une commande dans l'agenda d'un cuisinier. L'interface *TimeSlotManager* permet de gérer les ajouts, les suppressions et les récupérations des informations dans l'agenda d'un cuisinier.

CooksController : Ce composant à la responsabilité de gérer les actions liées à la prise en compte d'une commande par un cuisinier via l'interface *CookOrderProcessing*. L'interface *CookModifier* sert de endpoint afin de modifier les informations d'un cuisinier.

OrderController : La responsabilité de ce composant est de servir de endpoint pour les actions nécessaires à passer une commande avec les interfaces *OrderProcessing* et *OrderCreator.* L'interface *OrderUtilities* permet de d'apporter des modifications précises à une commande.

OrderRetrieverController : Ce composant vient implémenter les méthodes venant des interfaces *SmsService* et *OrderRetriever* qui représentent les logiques métiers des parties envoie de sms suite à l'oubli d'une commande et retraite d'une commande respectivement.

RecipeController : Possède la responsabilité de gérer tout ce qui est relatif aux recettes avec les interfaces suivantes: *RecipeCreator* pour la création, *RecipeModifier* pour les modifications et *RecipeCalculator* pour connaître le prix de la recette.

StockController : Implémente les logiques métiers associés au stock, comme la mise à jour des ingrédients via le catalogue d'un fournisseur et la gestion du stock par l'ajout d'ingrédients, respectivement les interfaces *IngredientsUpdater* et *StockModifier* .

Store Controller : Gère la logique des stores avec la possibilité de modifier les informations d'un store grâce à *StoreModifier*, gérer l'organisation des commandes du store grâce à *StoreOrderProcessing*, ajouter ou retirer des stores avec *StoreManager* et associe les boîtes surprises aux stores grâce à *SurpriseBoxHandler* **.**

TGTGNotificationController : Gère tout l'aspect "notification" (possibilité de spécifier un jour et une heure pour être notifié). Il implémente pour ça 2 interfaces *TGTGNotification* qui contient la méthode pour notifier les abonnés et *SuscriberManaging* qui gère la liste des abonnés.

UserController : Composant qui vient implémenter l'interface *UserRegisterer* correspondant à la création des utilisateurs et les membres. *UserFinder* permet la récupération de user.

ToGoodProxy : Il nous permet de simuler la connexion avec le service externe ToGoodToGo implémentant l'interface *TGTGSurpriseBox* qui contient les méthodes pour

gérer les paniers surprises et leurs états. Pour ce faire, le proxy utilise également un attribut du type *SurpriseBoxModifier* qui va permettre de faire le lien entre les magasins et leurs panier surprises associés.

OrderHelper : Permet d'obtenir toutes les anciennes commandes effectuées avec un mail grâce à l'interface *OrderFinder*.

Auto-Évaluation