

# The Cookie Factory - Team E

## Rendu 2

AL ACHKAR Badr, BEN AISSA Weel, CALAS Louis,  
EL KATEB Sami, HAITAM Ali

17 décembre 2022

# Table des matières

<b>1 UML</b>	<b>2</b>
1.1 Diagramme de Use-Case . . . . .	2
1.2 Diagramme de Classes . . . . .	3
1.3 Diagramme de Sequence . . . . .	6
<b>2 Choix des patrons de conception</b>	<b>7</b>
2.1 Patrons Retenus . . . . .	7
2.1.1 Builder et Decorator . . . . .	7
2.1.2 Observer . . . . .	8
2.1.3 Proxy . . . . .	8
2.1.4 Command dans le cas de la gestion du status de la commande . . . . .	9
2.2 Patrons Non-retenus . . . . .	9
2.2.1 Command dans le cas de la validation de commande . . . . .	9
2.2.2 State dans le cas de la gestion du status de la commande . . . . .	10
<b>3 Retrospective Spring</b>	<b>11</b>
<b>4 Auto-Evaluation</b>	<b>14</b>

# 1 | UML

## 1.1 Diagramme de Use-Case

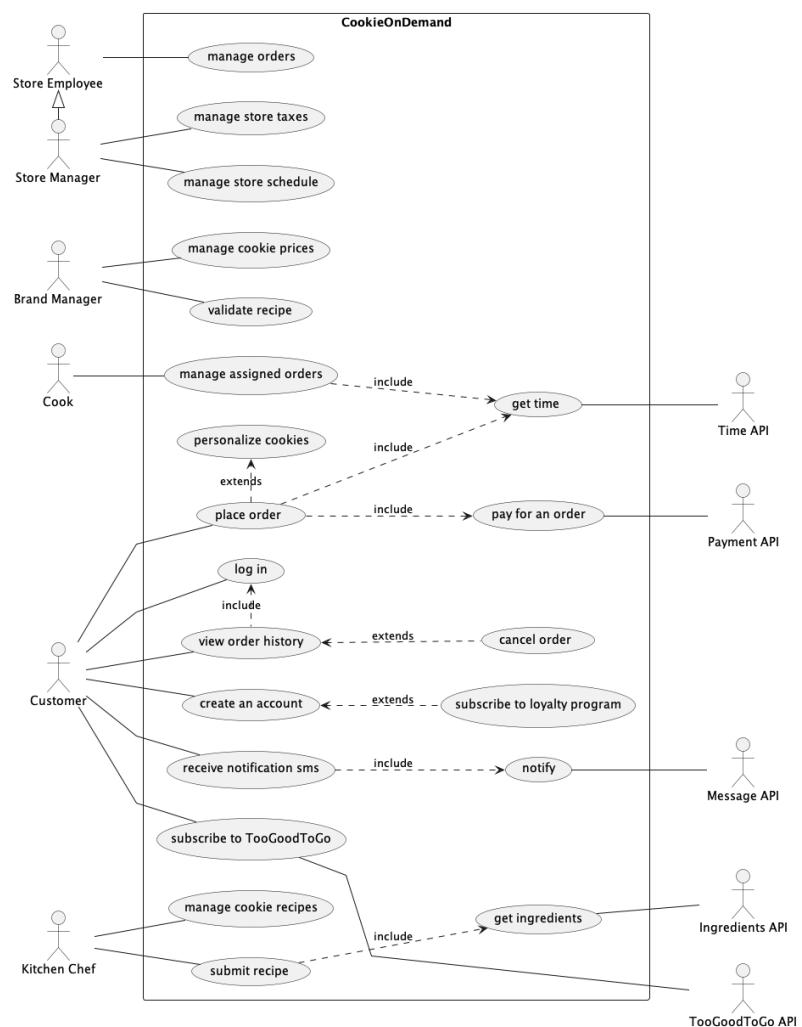


FIGURE 1.1 – Diagramme de Use-Case couvrant le périmètre final

Ce diagramme de Use case part du principe que tous les utilisateurs sont connectés et que leurs comptes bénéficient des droits d'accès pour effectuer leurs tâches respectives.

## 1.2 Diagramme de Classes

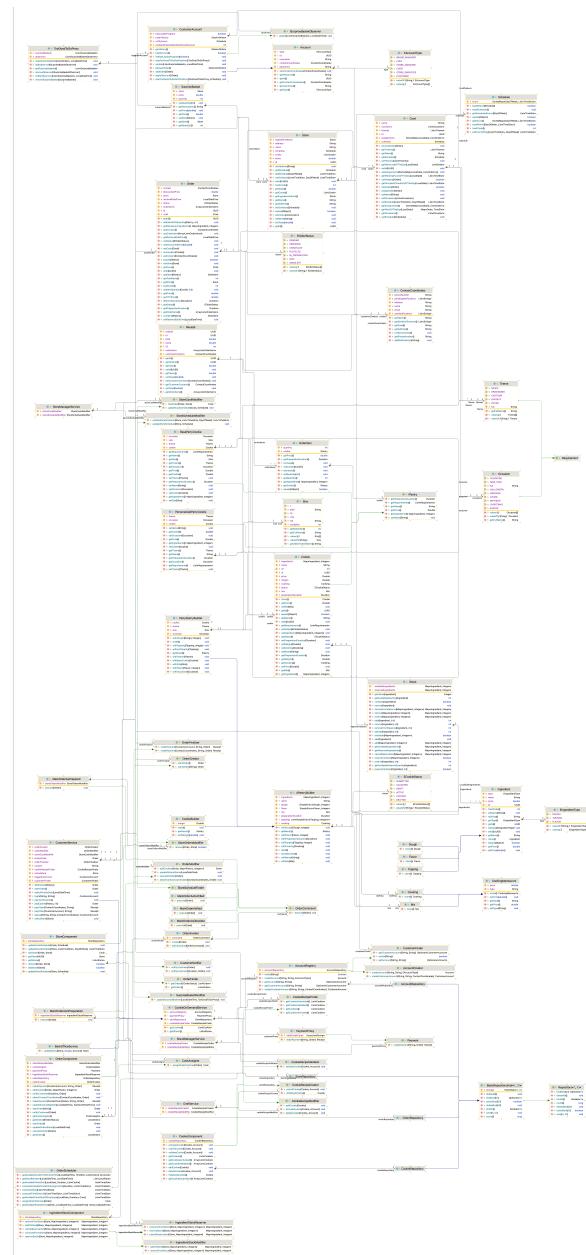


FIGURE 1.2 – Diagramme de Classes Complet

**Diagramme de classe :** le diagramme complet étant difficilement lisible, nous l'avons séparé en deux parties pour faciliter la lecture. Le diagramme complet est disponible sous format png dans doc/images/ClassesIntelliJ.png

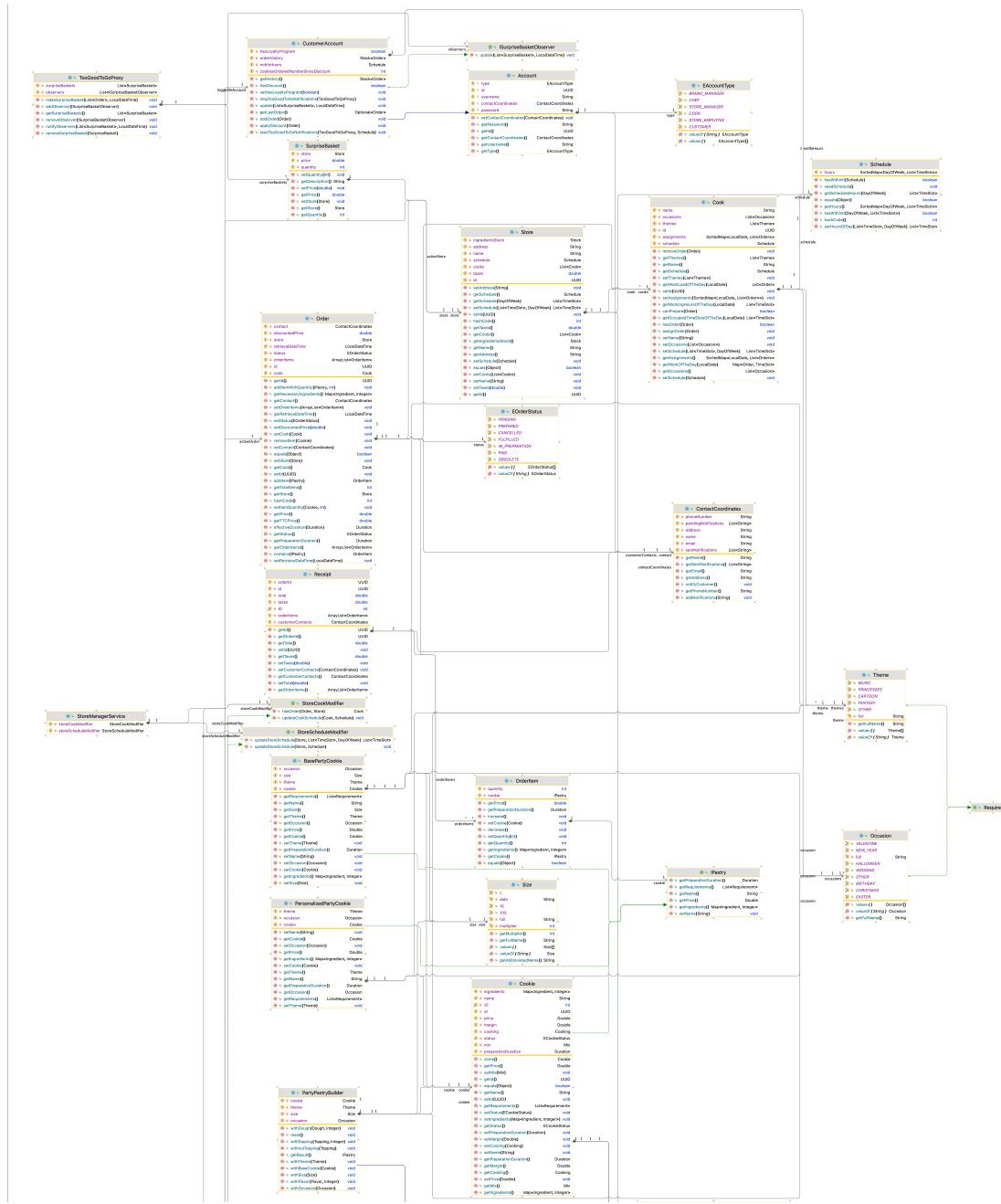


FIGURE 1.3 – Diagramme de Classes partie du haut

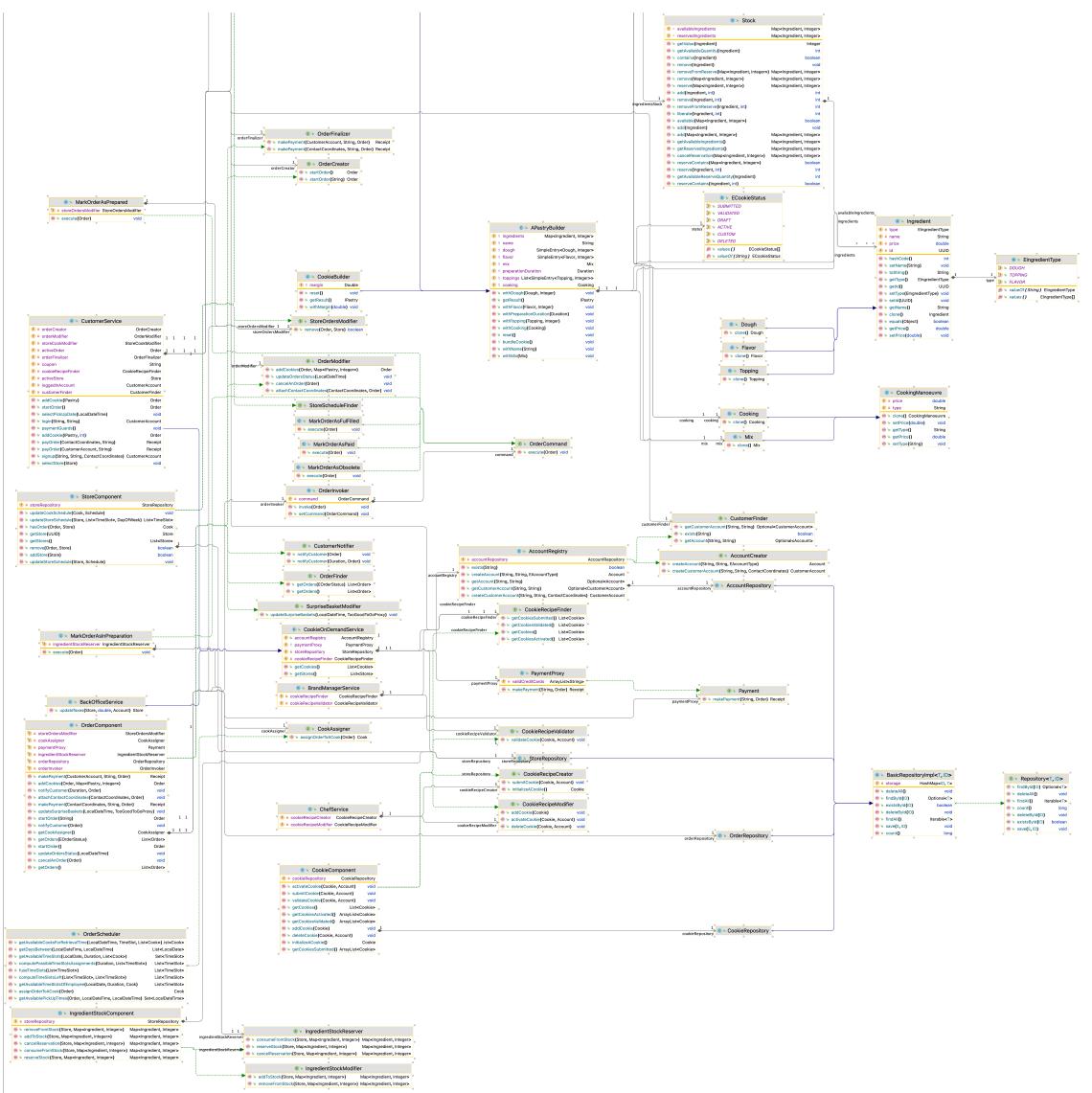


FIGURE 1.4 – Diagramme de Classes partie du bas

### 1.3 Diagramme de Séquence

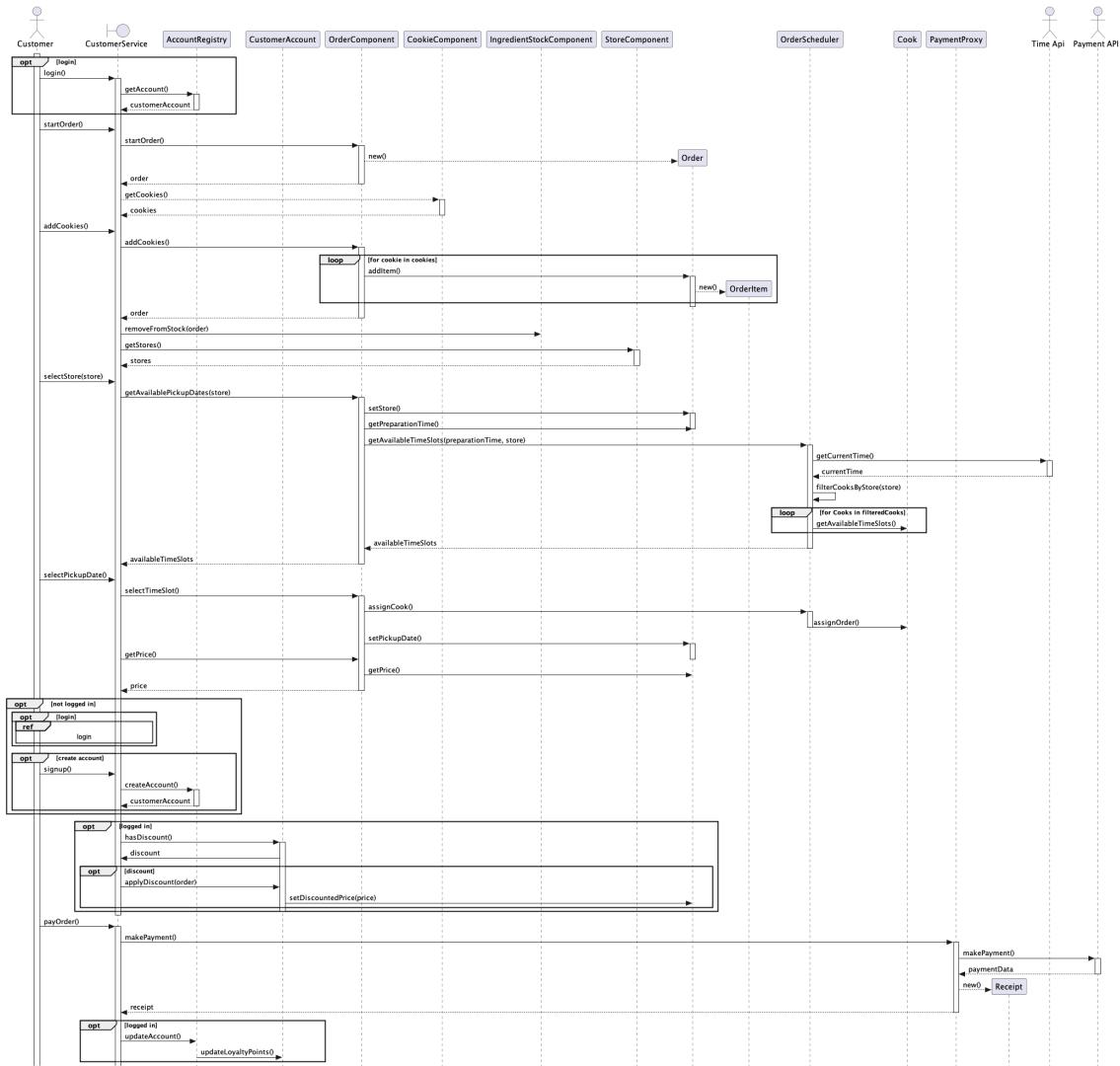


FIGURE 1.5 – Diagramme de Séquence

# 2 | Choix des patrons de conception

## 2.1 Patrons Retenus

### 2.1.1 Builder et Decorator

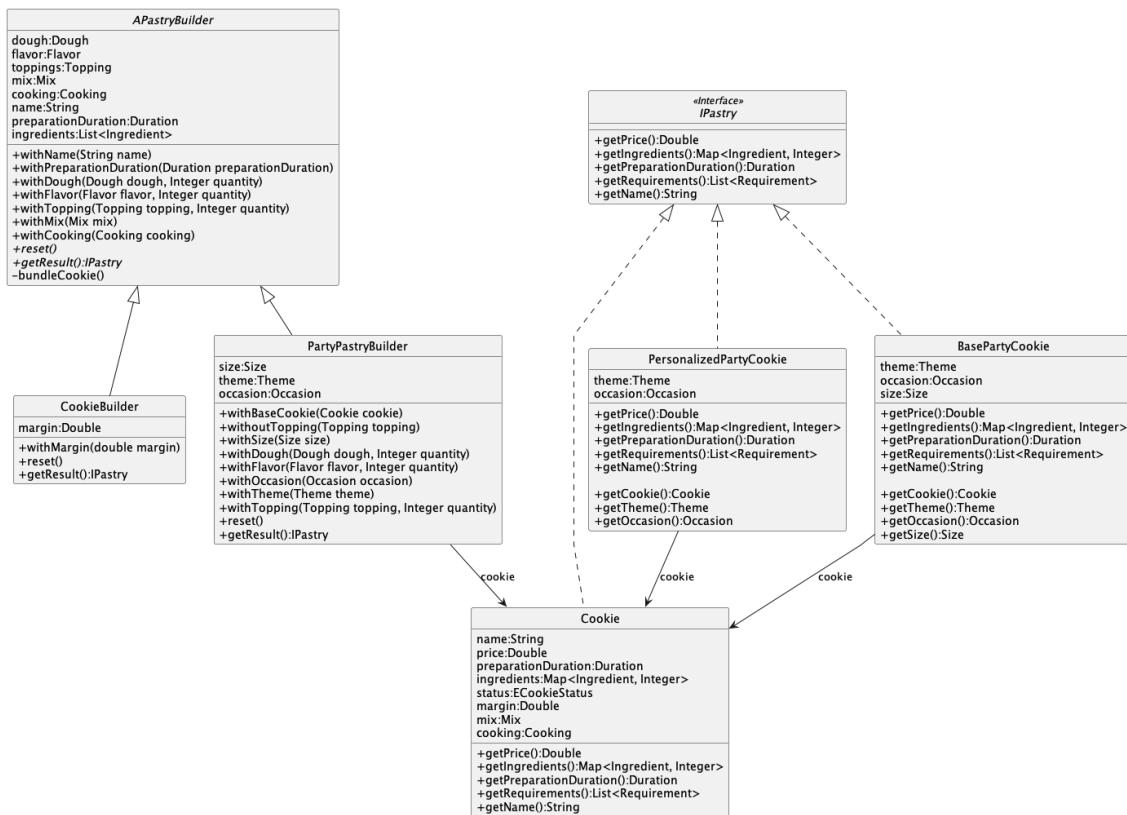


FIGURE 2.1 – Diagramme de Classe simplifié des Patrons Builder et Decorator

**Builder** : nous avons choisi d'utiliser ce patron de conception à deux reprises dans notre application. Dans un premier temps, nous l'avons utilisé pour la création de nouvelles recettes. En effet, lorsque nous souhaitons créer une recette de cookie, il est entre autre nécessaire de renseigner un

nom, une liste d'ingrédients, un prix, un type de cuisson, etc. Pour mettre en place ce patron de conception, nous avons créé la classe CookieBuilder qui expose des méthodes tel que `withName`, `withFlavor`, `getResult`. Son utilisation nous permet de faciliter création des recettes en choisissant les attribut que nous souhaitons initialiser et en gardant dans attribut par défauts pour ceux qui ne sont pas définis.

**Decorator :** dans un second temps, nous avons associé les patrons de conception Builder et Decorator pour la création de cookies festif (PartyCookies). Le patron de conception Decorator nous permet de modifier les caractéristiques d'un cookie de base (prix, taille, ingrédients, durée de préparation) ainsi que d'ajouter de nouveaux attributs (thème, occasion, ...), sans avoir à modifier la classe Cookie. Son association avec le Builder nous permet d'abstraire le choix du type de PartyCookie qui sera créé. Nous pouvons donc obtenir à la fin du processus de création un cookie de type PersonalizedPartyCookie, qui est un cookie entièrement personnalisé, ou un cookie de type BasePartyCookie qui est un cookie standard dont la taille a été augmentée. Le choix du type de cookie dépendra uniquement des éléments qui ont été ajouté lors de la création du cookie par le PartyCookieBuilder.

### 2.1.2 Observer

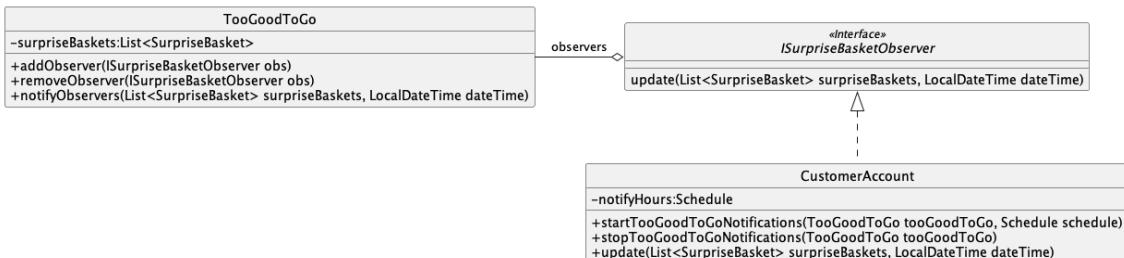


FIGURE 2.2 – Diagramme de Classe simplifié du Patron Observer

**Observer :** pour mettre en place le service TooGoodToGo dans notre application, nous avons choisi d'utiliser le patron de conception Observer. Les clients doivent pouvoir facilement s'abonner et se désabonner des notifications TooGoodToGo. Le patron de conception Observer semble bien répondre à ce cas d'utilisation. Chaque client peut s'inscrire aux notifications TooGoodToGo par l'intermédiaire de la méthode `startTooGoodToGoNotifications`. Il est alors ajouté aux observers des paniers surprise et lorsqu'un nouveau panier est ajouté, tous les observers sont notifiés. Les clients peuvent également à l'inscription au service choisir les horaires et les jours auxquels ils veulent être notifiés en signifiant un calendrier (schedule).

### 2.1.3 Proxy

**Proxy :** nous avons choisi d'implémenter le design pattern Proxy sur les classes ayant pour responsabilité le dialogue avec des services partenaires comme le catalogue d'ingrédient, le service de paiement ou l'application TooGooToGo. Ce patron de conception consiste à mettre en place un objet intermédiaire qui va représenter un objet distant. Il permet entre autre de contrôler l'accès à ces objets et permet de facilement remplacer l'objet réel par un objet simulé.



FIGURE 2.3 – Diagramme de Classe simplifié des Objets Proxy

#### 2.1.4 Command dans le cas de la gestion du status de la commande

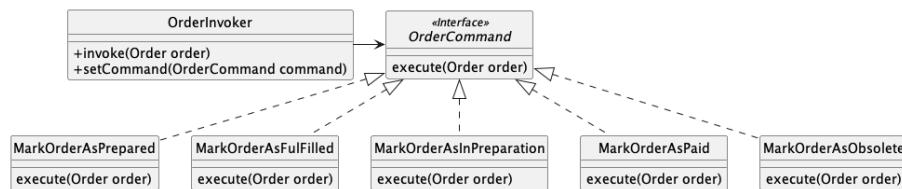


FIGURE 2.4 – Diagramme de Classe du design pattern command dans la gestion des status de commandes

**Command :** nous avons choisi d’implémenter le design pattern Command pour gérer le changement de status de la commande, en partant de l’idée que c’est bien fait par des invocations de la part des acteurs du système. Ainsi, la responsabilité de la classe ‘OrderInvoker’ est d’exécuter le bon algorithme de mise à jour (spécifier au démarrage dans le contexte du système) sur la bonne ressource réceptrice (le bon order). Ce design pattern permet d’extraire un algorithme avec ses conditions d’exécution dans une classe à part et donne la liberté de gérer les cas d’erreurs, et même d’implémenter des opérations d’undo/redo, chose qu’on a pas pu l’implémenter dans le temps permis.

## 2.2 Patrons Non-retenus

### 2.2.1 Command dans le cas de la validation de commande

**Command :** nous avons étudié l’implémentation du patron de conception command au niveau de la validation de commande par les clients. En effet, lors de la validation de commande, nous devons réserver incrémentalement les ressources pour la préparation de la commande, avant de savoir si la commande va aboutir. Ce patron de conception semblait donc judicieux, car il aurait permis de facilement annuler les réservations dans le cas où la commande n’aboutit pas. Nous aurions également pu créer des MacroCommand composites qui représenteraient les commandes de cookies, ce qui faciliterait leur annulation. Cependant, nous avons choisi de ne pas implémenter ce patron de conception, car il aurait impliqué de trop grandes modifications sur notre base de code sans entraîner de gains importants étant donné que les fonctionnalités en question étaient déjà implémentées.

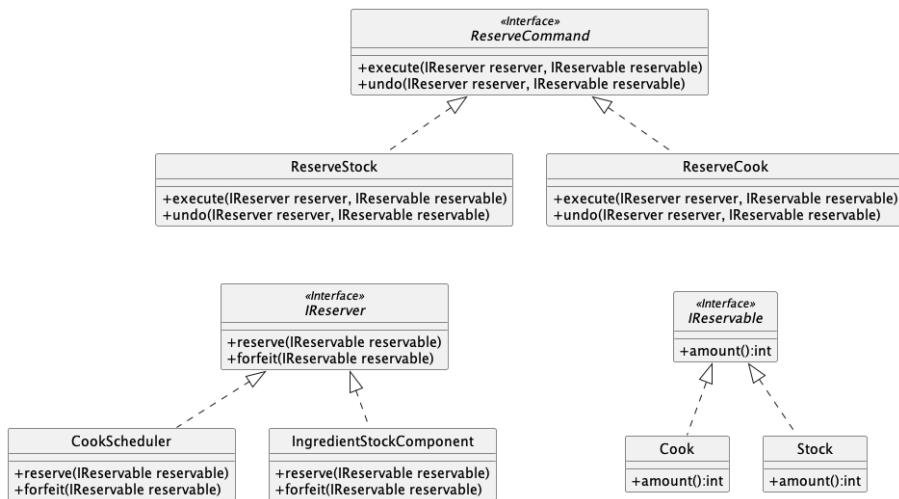


FIGURE 2.5 – Diagramme de Classe d'étude préliminaire d'implémentation du patron de conception Command

## 2.2.2 State dans le cas de la gestion du status de la commande

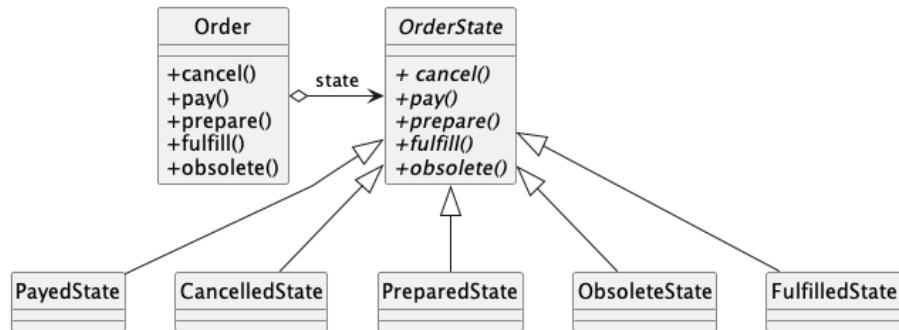


FIGURE 2.6 – Diagramme de Classe d'étude préliminaire d'implémentation du patron de conception State

**State :** pour la gestion du statut de commande, nous avons étudier l'implémentation du patron de conception State. Ce patron de conception semblait approprié car il aurait permis de modifier les actions des méthodes en fonction du statut de la commande. Par exemple nous aurions pu empêcher l'annulation de la commande à partir du moment où la commande est en cours de préparation. Cependant, nous avons choisi de ne pas implémenter ce patron de conception car nous avions déjà implémenté toutes les fonctionnalités sur la gestion des status des commandes.

### 3 | Retrospective Spring

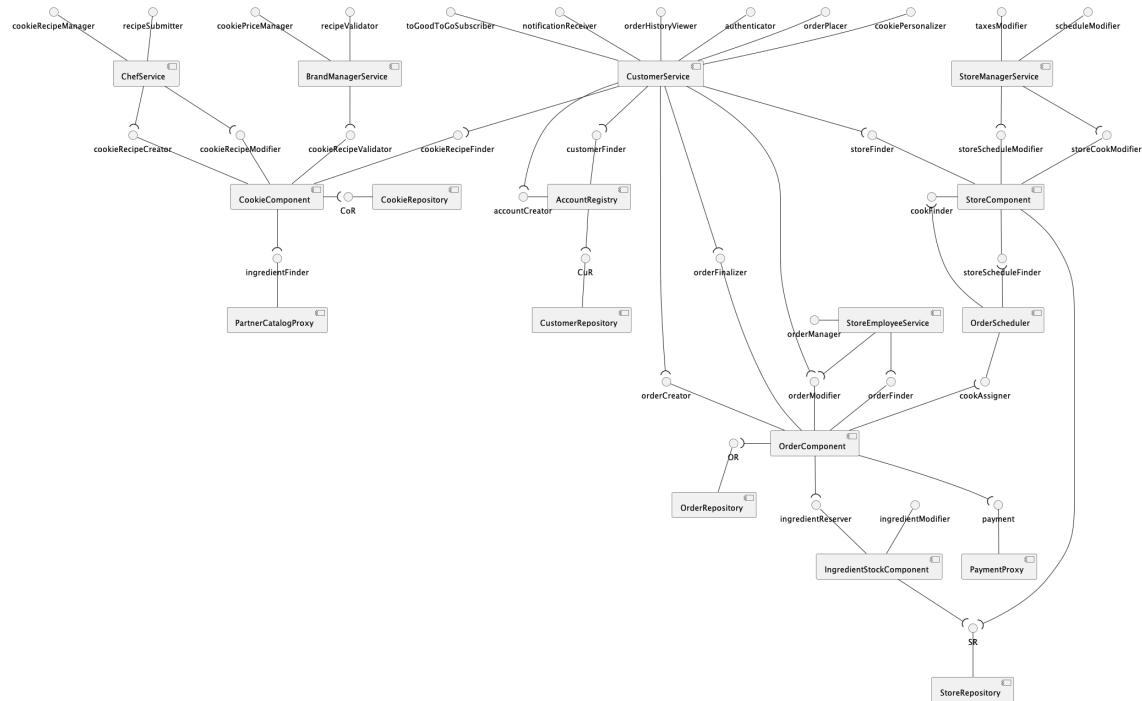


FIGURE 3.1 – Diagramme de Composants Spring

**Pour migrer** notre code d'une conception OOP vers une approche par composant, nous nous sommes appuyés sur notre conception initiale qui comportait déjà une décomposition des classes en fonction des fonctionnalités métier recherchées, ainsi que les services respectifs chargés d'offrir l'accès à ces fonctionnalités.

Comme certaines de nos classes de "service" initiales avaient les lignes brouillées entre leurs responsabilités, nous avons opté pour une approche de ségrégation d'interface pour visualiser quelle classe de service prenait en charge quelle responsabilité.

Sur cette base, les modifications suivantes ont été apportées : Nous avons séparé notre Store-Service (maintenant appelé StoreComponent), en deux composants distincts : StoreComponent et

`IngredientStockComponent`. Comme la classe intiale avait deux grandes responsabilités, la première étant de gérer les préoccupations des magasins et la seconde étant de gérer méticuleusement les stocks d'ingrédients, nous les avons séparés en deux composants.

Comme le montre le diagramme des composants, la classe `IngredientStockComponent` expose deux interfaces : `ingredientReserver` pour gérer la réservation et la libération des ingrédients en stock, et `ingredientModifier` pour gérer les quantités réelles en stock au fur et à mesure de l'arrivée de nouveaux lots.

**Le StoreComponent**, il a été restreint au périmètre de la gestion des plannings des magasins (pour leur interrogation et leur modification), et de leurs cuisiniers (pour leur interrogation et la modification de leur planning).

**OrderScheduler** a été définit comme un composant, car il se charge de gérer le planning des cuisiniers et l'affectation des commandes aux cuisiniers respectifs. Il n'expose donc qu'une seule interface qui est : `cookAssigner`.

**Le CookieService** initial a été renommé en un `CookieComponent` qui expose les interfaces nécessaires pour gérer la logique de soumission, d'ajout, de confirmation et d'interrogation des différentes recettes : `cookieRecipeFinder`, `cookieRecipeValidator`, `cookieRecipeCreator`, `cookieRecipeModifier`.

**AccountService** a été renommé `AccountRegistry`, et a été modifié de manière à ce qu'il implémente deux interfaces : un `accountCreator` et un `customerFinder` pour exposer les services derrière la logique d'inscription et de recherche d'informations d'identification.

**Les "services"** originaux qui étaient des proxies sont devenus des connecteurs, c'est-à-dire des composants qui fournissent une interface pour des services externes : les API. Par exemple : `PaymentProxy`, `TooGoodToGo`, et `PartnerCatalogProxy`.

**OrderComponent**, basé à l'origine sur `OrderService`, a été conçu de manière à exposer les services de création d'une commande, de sa recherche, de sa modification et de sa confirmation. Les interfaces respectives sont : `orderCreator`, `orderModifier`, `orderFinder` et `orderFinalizer`.

Comme ce composant fait beaucoup de travail de base, il dépend de plusieurs interfaces importantes pour le réaliser, notamment : `Payment`, `cookAssigner`, et `ingredientReserver` pour passer à la caisse.

Nous n'avons pas mentionné le rôle que jouent les composants "Repository" dans la décomposition ci-dessus, car ces composants sont simplement chargés de gérer l'état de la base de données "en direct", sans logique commerciale apparemment intéressante.

**Pour conclure,** toutes nos classes 'Façade' suffixées 'System' auparavant, ont été transformées en services dans l'approche par composant, car ces classes sont devenues responsables de l'exposition des abstractions les plus élevées et des points d'accès directs aux différentes caractéristiques et fonctionnalités du système :

**CustomerService :** offre des services d'authentification via Authenticator tels que la connexion et l'inscription, des services liés aux commandes tels que le passage d'une commande et la spécification de ses propriétés (heure et lieu d'enlèvement) et de ce qu'elle concerne : cookies génériques ou cookie de partie personnalisé, etc. via orderPlacer, orderHistoryViewer, tooGoodToGoSubscriber, cookiePersonalizer et notificationReceiver. A utiliser par un client régulier.

**BrandManagerService :** expose des fonctionnalités de validation des recettes et de gestion des prix à utiliser par le gestionnaire de la marque par le biais de recipeValidator et cookiePriceManager.

**ChefService :** expose des fonctionnalités liées aux recettes qui ont un impact sur le catalogue ; soumission et modification en cas d'acceptation, à utiliser par les chefs de la marque à travers recipeSubmitter et cookieRecipeManger.

**StoreManagerService :** expose les fonctionnalités liées au magasin et destinées à être utilisées par le gestionnaire du magasin, à savoir : les taxes et la modification du calendrier du magasin par le biais de scheduleModifier et taxesModifier.

## 4 | Auto-Evaluation

**AL ACHKAR Badr** : 100 points

**BEN AISSA Weel** : 100 points

**CALAS Louis** : 100 points

**EL KATEB Sami** : 100 points

**HAITAM Ali** : 100 points