Rapport Conception Logicielle

The Cookie Factory

Université Côte d'Azur

Polytech Nice Sophia

SI4-Conception Logicielle

Equipe H

Ben Aissa Nadim

Boubia Marouane

El Garmit Youssef

Zoubair Hamza

Saissi Omar

Année 2022-2023

Sommaire

1) Section UML

1.a) Diagramme de cas d'utilisation :

Nous avons émis les hypothèses suivantes :

- Un acteur responsable de la marque, **Brand Manager** qui va valider les recettes proposées par le chef de la marque **Brand Chef**.

- Un acteur **Guest** qui est un visiteur qui ne possède pas de compte client et qui a donc la possibilité de s'inscrire et de naviguer sur le système.

- Un acteur **Client** qui est un guest qui possède un compte dans l'application .

- Un acteur **Cook** responsable de préparer la commande et donc d'annoncer le début et la fin de la préparation de cette dernière.

- Un acteur **Cashier** responsable de la validation du retrait de commande.

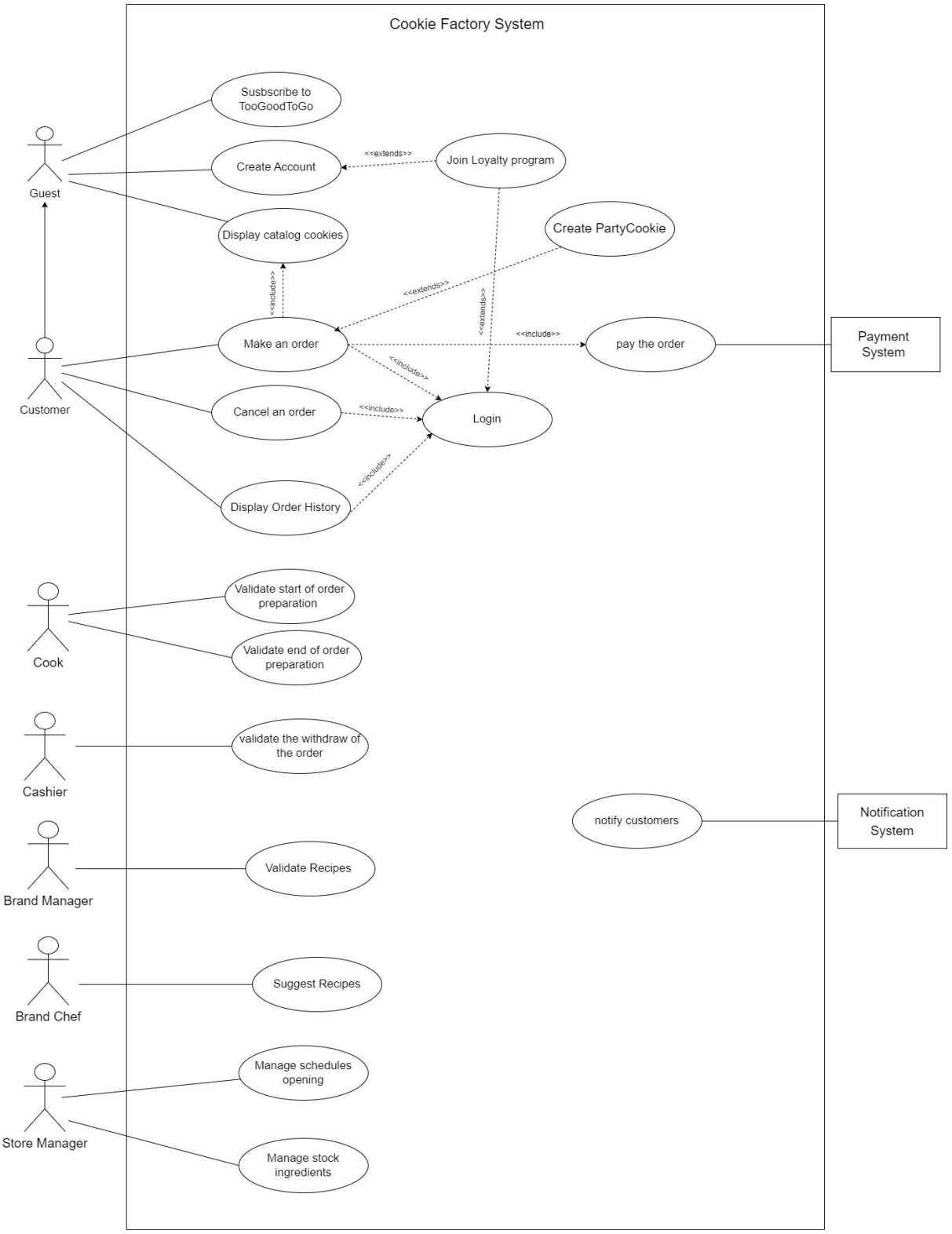
- Un acteur externe **SystemNotifier** qui notifie le client 5 minutes après que la commande soit prête afin qu'il puisse venir la récupérer.
  Si la commande n'est pas récupérée 2 heures après l'heure de l'enlèvement programmé, elle sera considérée comme obsolète.
  Enfin nous avons également mis en place les notifications pour notifier les utilisateurs ainsi que des paniers surprises TooGood ToGo sont disponibles.

- Un acteur externe **Payment** représente le système de paiement qui gère le règlement des commandes passé par les clients .

- Un acteur **Store Manage**r est responsable de la gestion des stocks et des horaires du magasin.

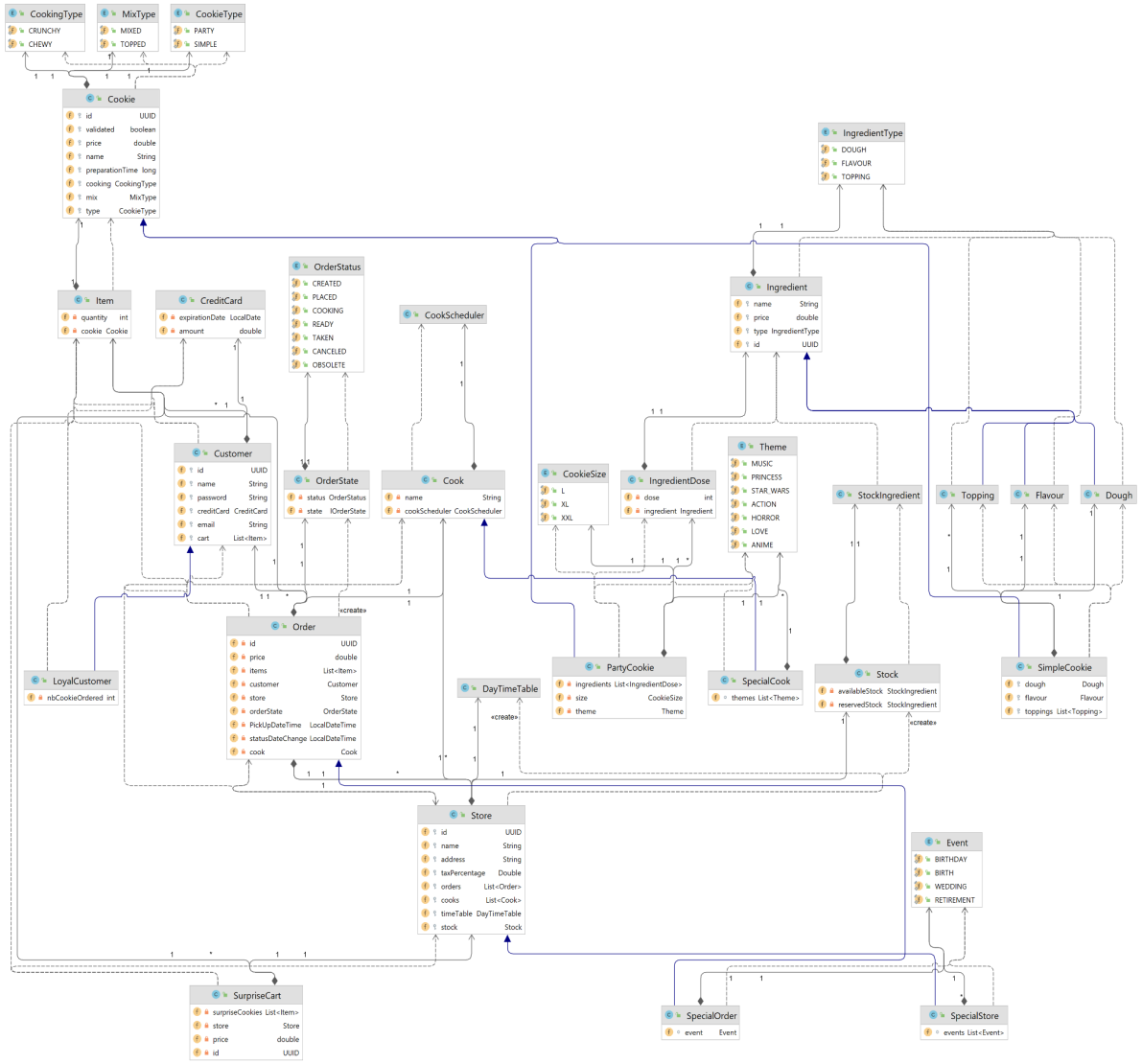
Pour la partie TooGood ToGo, on a supposé que n'importe quel utilisateur (Guest) peut s'abonner au programme seulement à l'aide de son email donc il peut recevoir des notifications à propos des paniers surprises .

Diagramme de cas d'utilisation : Cookie Factory System



1.b) Diagramme de classe :

- **Diagramme global :**



Lien vers le diagramme :

https://github.com/PNS-Conception/cookiefactory-22-23-h/blob/main/doc/diagram/class/global.png

- **Diagramme de classe par package :**

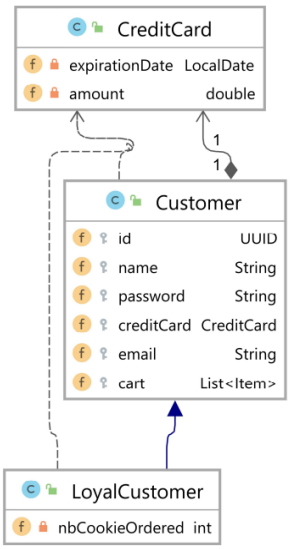


Diagramme de classe du package "Customer"

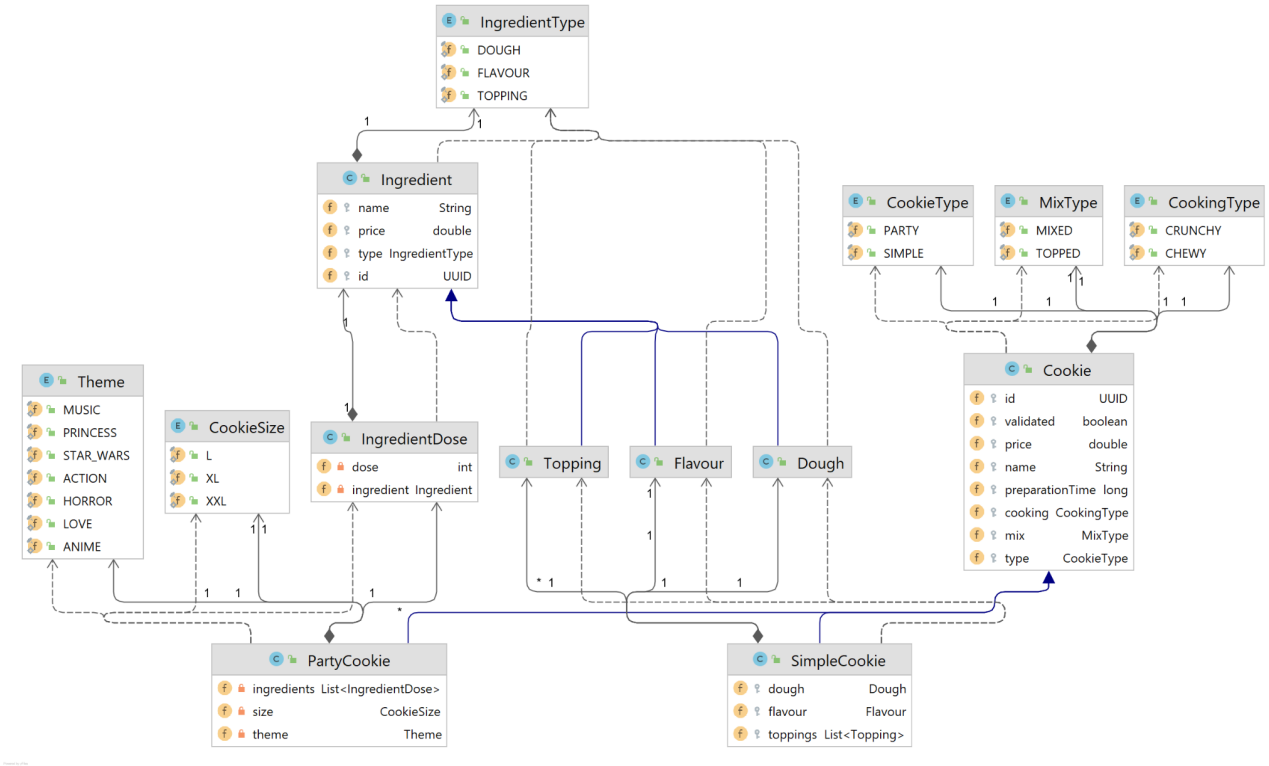


Diagramme de classe du package "Cookie"

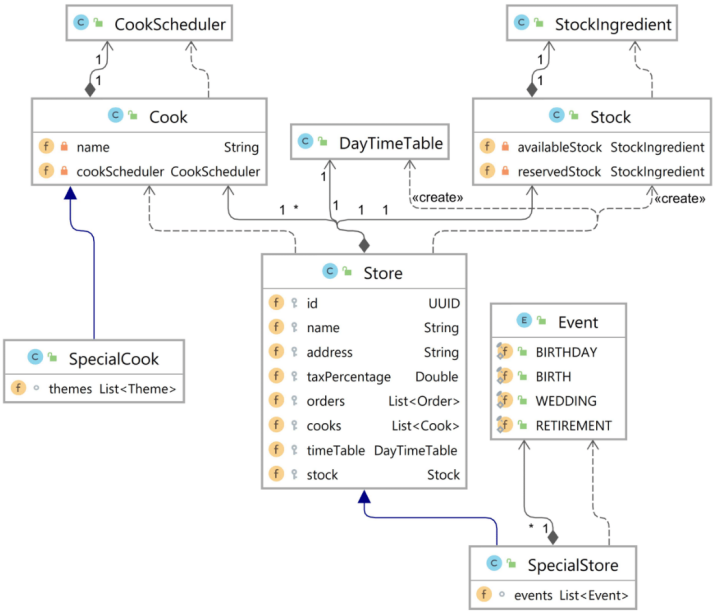


Diagramme de classe du package "Store"

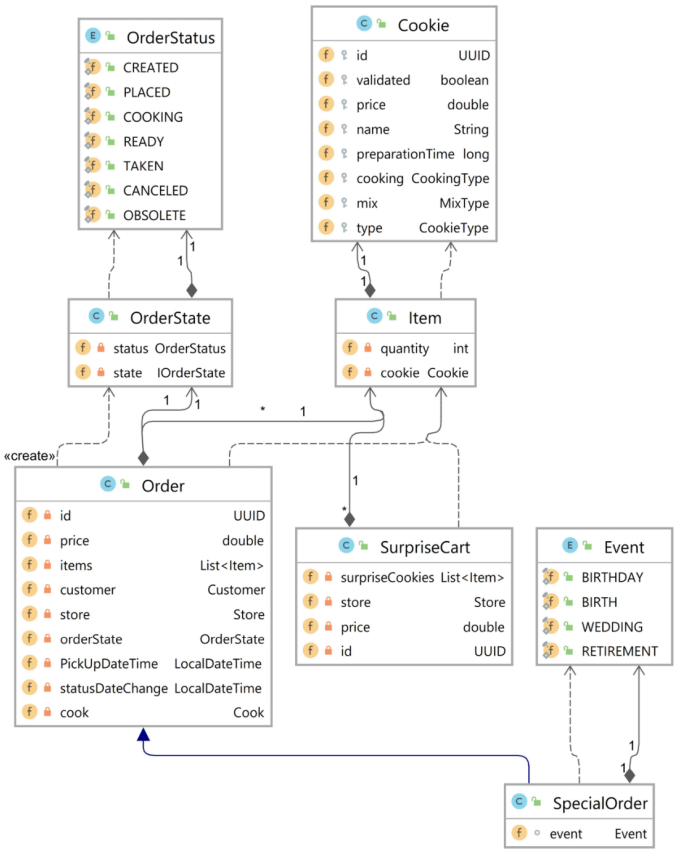


Diagramme de classe du package "Order"

1.c) Diagramme de séquence :

Voici ci-dessous les différentes étapes pour passer une commande :

1- Le client ajoute les cookies désirés avec la quantité souhaitée.
2- Le client valide son panier, ainsi une commande est créée .
3- Le client choisi le store :

- Si le store est introuvable, un message d'erreur est affiché au client

4- Le client choisit l'heure à laquelle il souhaite récupérer sa commande ;

- Si le store choisi n'est pas ouvert à cette heure, un message d'erreur est affiché

5- Le client valide sa commande en payant par carte.

- Le système *OrderCreatorService* vérifie ainsi :
  - la disponibilité du stock puis réserve le stock requis par la commande.
  - la disponibilité d'un cuisinier pour pouvoir lui affecter la commande.

- la validation du paiement.

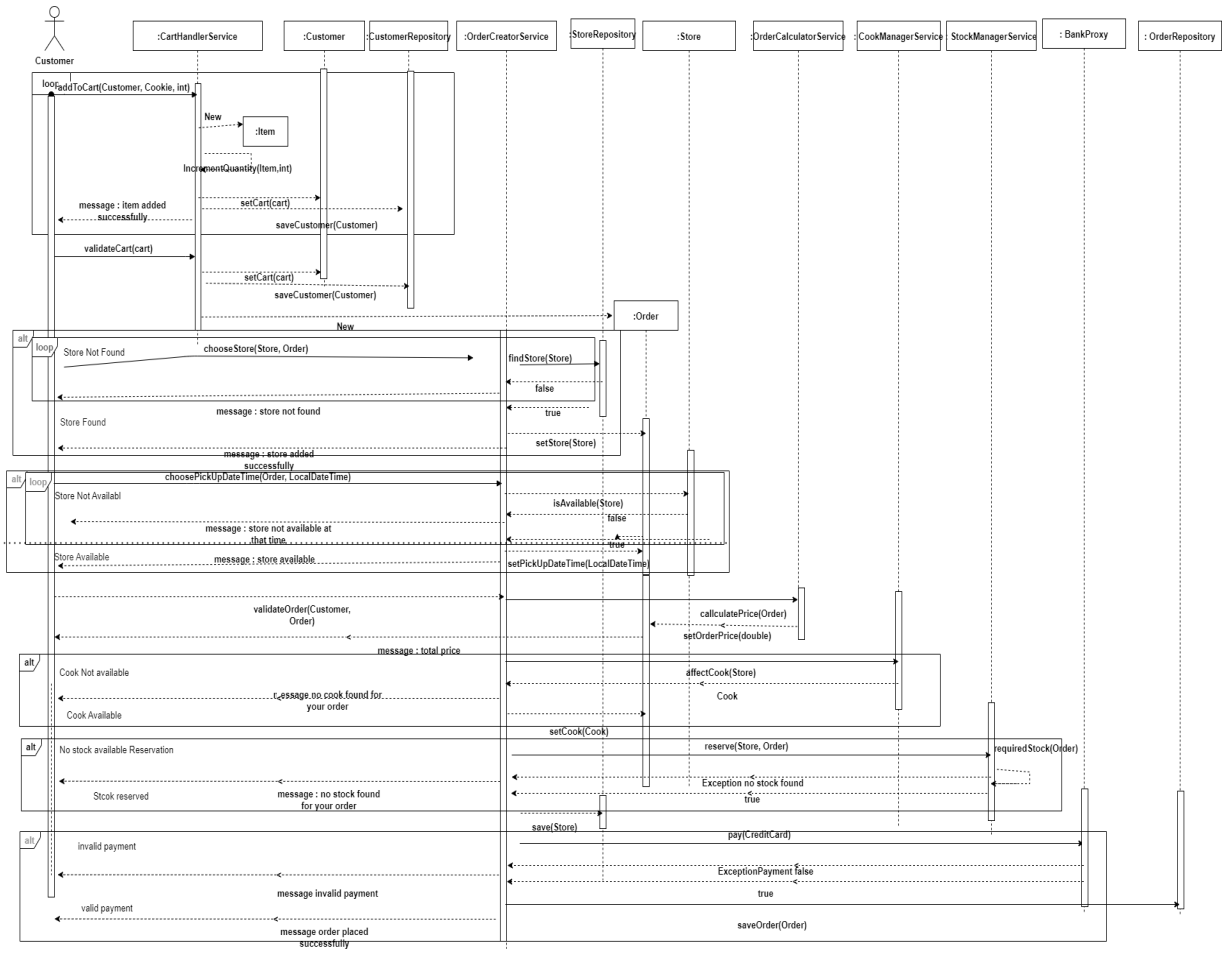


Diagramme de séquence : Passer une commande

lien du diagramme de séquence :

https://github.com/PNS-Conception/cookiefactory-22-23-h/blob/main/doc/diagram/sequence/sequence.png

2) Patrons de conception :

Ce projet de conception logicielle est d'une grande envergure, afin de structurer et de faciliter la maintenance du code nous avons décidé de mettre en place des design pattern. Ces design pattern nous permettent également d'améliorer la lisibilité et la compréhension du code ce pourquoi nous avons jugé opportun de mettre les six design pattern suivants :

- **Builder :**

L'utilisation du design pattern "Builder" nous a permis de construire des objets complexes étape par étape. Nous l'avons notamment utilisé dans l'objet Cookie qui est l'un des objets les plus complexes de ce programme puisque sa création nécessite l'ajout du type de pâte , de la saveur , le type de cuisson , les toppings, etc… ce qui rend la création un peu difficile , d'où vient l'intérêt des classes **CookieBuilder , PartyCookieBuilder** qui ont comme rôle la composition de notre objet seulement en appelant les méthodes withDough , withTopping , etc…
Enfin il nous fournit un objet cookie prêt. Ce pattern nous a permis de séparer la création de cet objet de sa représentation et par suite une bonne lisibilité de notre code .

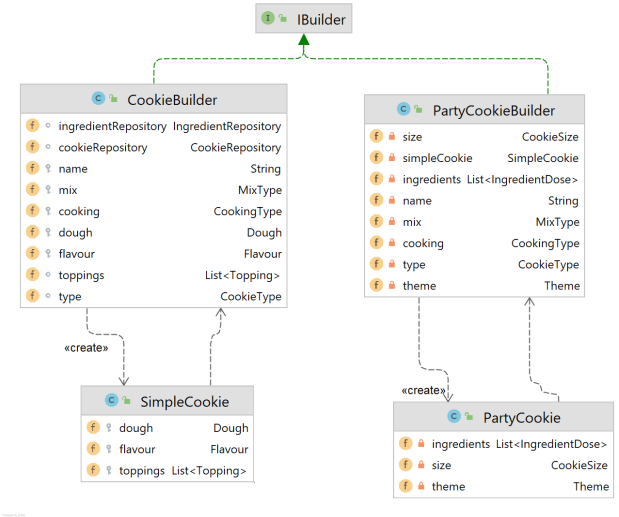


Diagramme de classe du design pattern "Builder"

- **Proxy:**

Le design pattern “Proxy” permet d’ajouter une couche d'abstraction au-dessus d'un objet existant afin de contrôler l'accès à celui-ci. Ce pattern a été utilisé pour notre classe Bank, avec la création d’une classe BankProxy. Cette dernière nous permet de vérifier en amont les données de paiement avant de procéder au paiement final avec la méthode pay() de la classe Bank. On garde ainsi le contrôle sur l'accès à l'objet original Bank.

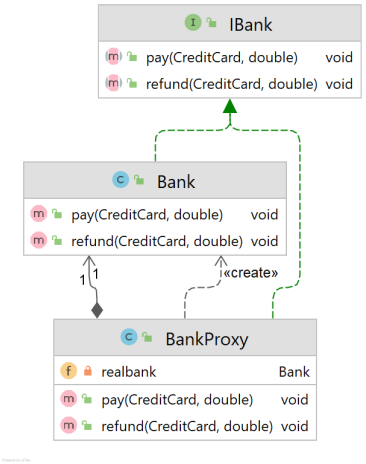


Diagramme de classe du design pattern “Proxy” :

- **Decorator :**

Le Decorator est un pattern très utile lorsqu’on veut combiner plusieurs comportements d’un objet sans apporter beaucoup de modifications à notre code . Ainsi nous avons utilisé ce pattern dans notre système de notification , plus spécifiquement lorsque nous souhaitons notifier un utilisateur avec deux méthodes en même temps . Les deux classes EmailDecorator et SMSDecorator nous permettent d’envelopper notre objet Notification qui est représenté par l’attribut “wrappee” dans la classe BaseDecorator et puis envoyer le message de deux manières différentes . Nous pourrions utiliser dans ce cas un héritage simple , mais cela poserait un problème s’il y avait beaucoup de méthodes de notification d'où vient l’efficacité du Decorator .

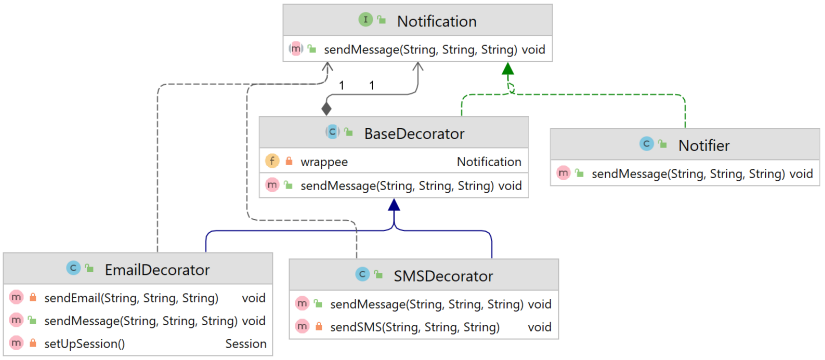


Diagramme de classe du design pattern "Decorator"

- **State :**

Nous avons utilisé dans ce projet le design pattern "State" pour permettre à notre objet OrderState de changer de comportement lorsque son état interne change, sans avoir à changer de classe. Cela nous permet notamment de switcher entre les différents états (Obsolete, Placed, Ready, Cooking Created Canceled et Taken) de la commande. Ainsi en faisant appel à la méthode next() de l'interface l'OrderState qui implémente les différents états de la commande, le cycle de vie de la commande suit son rythme (qui a été défini au préalable) et agit en fonction de son état automatiquement.

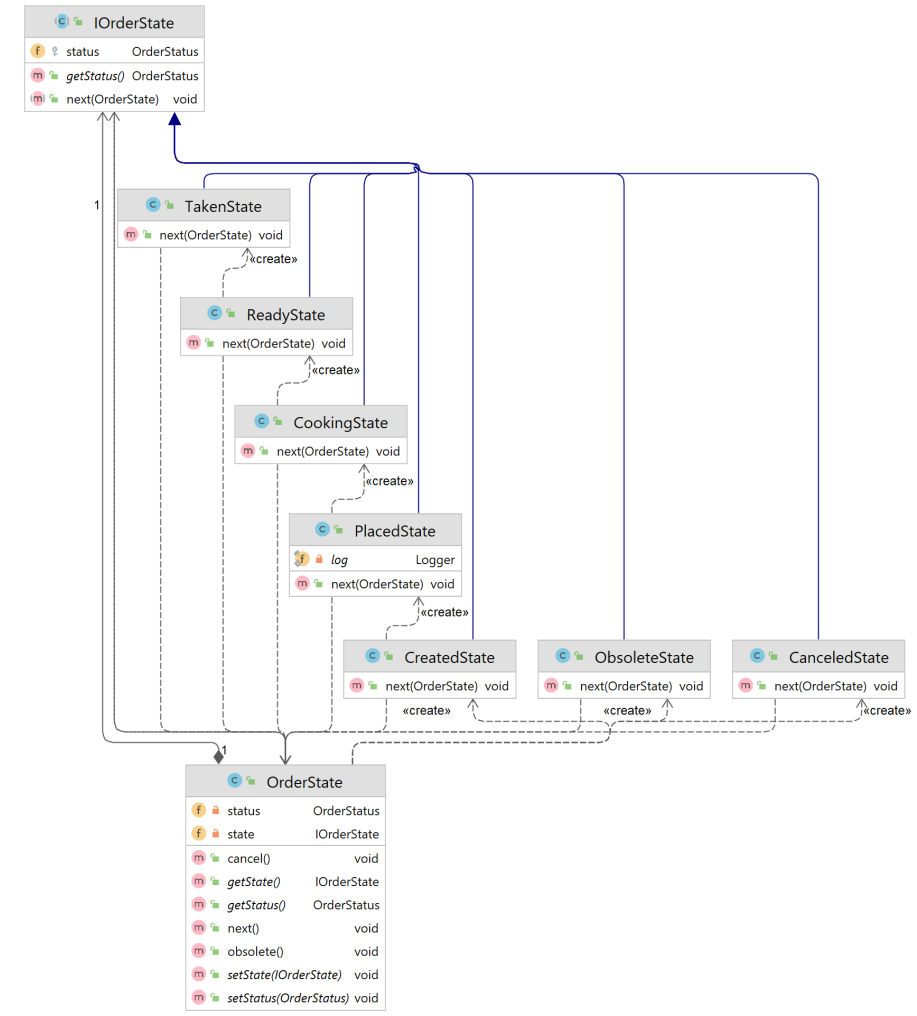


Diagramme de classe du design pattern "State"

- **Strategy :**

Le design pattern Strategy est utilisé pour permettre à un objet de changer de comportement en utilisant une stratégie différente qui convient le mieux à ses besoins en fonction de son état ou de ses préférences. Ce dernier nous à permis ici de changer la façon de calculer les temps de préparations et les prix de nos cookies en fonction de si il s'agit d'un cookie Simple ou bien d'un Party. Pour cela dans notre code nous utilisons l'interface commune CookieStartegyCalculator pour implémenter les classes PartyCookieStrategyCalculator et SimpleCookieStrategyCalculator.

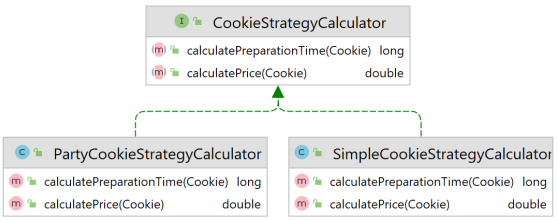


Diagramme de classe du design pattern "Strategy"

- **Factory :**

Le pattern factory est un pattern très efficace dans la création des objets d'une manière plus maintenable puisqu'il respecte les principes SOLID , en particulier le principe "ouvert / fermé" c'est-à-dire on peut créer de nouveaux objets sans changer le code de base. Nous avons choisi ce pattern pour modéliser la création des nouveaux ingrédients qui est une action dynamique puisqu'ils sont changés et ajoutés dans le catalogue des ingrédients par les managers à chaque fois . Donc , ce pattern nous garantit la stabilité de notre code à l'aide de la classe IngredientFactory qui est chargée de créer un nouvel ingrédient selon son type .

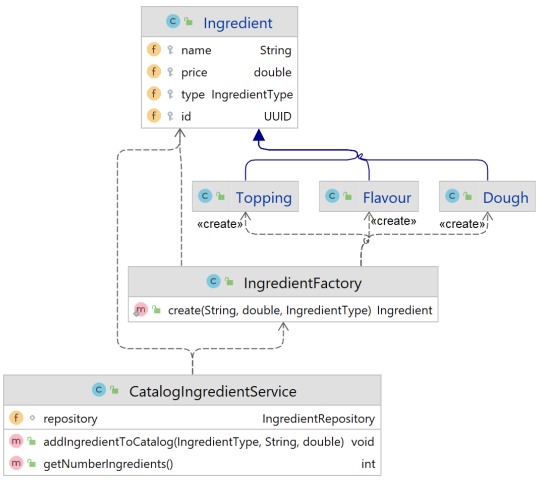


Diagramme de classe du design pattern "Factory"

En ce qui concerne les patrons de conception qui pouvaient être appliqués dans notre projet on peut citer :

- ***Command*** **:**

Ce design pattern était déjà utilisé avant la migration Spring pour modéliser le passage de la commande par plusieurs étapes avant d'être livrée au client, de la création, du paiement, de la préparation et enfin de la récupération. Nous avons implémenté plusieurs classes qui représentent l'état de la commande et définissent la méthode d'exécution afin de faire avancer l'état de la commande. Cependant, il était difficile d'appliquer ce design pattern dans la partie Spring avec les composants, nous avons donc opté de le remplacer par le pattern State.

- **Observer :**

Nous avons envisagé l'utilisation d'un Observer afin de notifier le stock pour les mettre à jour (par exemple diminution du stock à chaque commande en cours de préparation).
Cependant, nous avons supposé que lorsque chaque cuisinier veut valider le début de préparation de la commande, notre système gère la diminution du stock selon les ingrédients utilisés dans la commande .

3) Rétrospective :

En ce qui concerne la migration de notre application vers une approche à composants Spring nous avions décidé au début de réaliser un MVP constitué de la création de la commande , le passage au paiement, la validation du cuisinier du début de préparation et puis la fin de préparation . Tout d'abord, nous avons bien étudié l'architecture de notre code de base , puis nous avons essayé de transformer les classes qui jouent le rôle d'un *Manager* (*exemple: OrderProcess*, *CookManager*, …) en des composants Spring en ajoutant les interfaces associées.

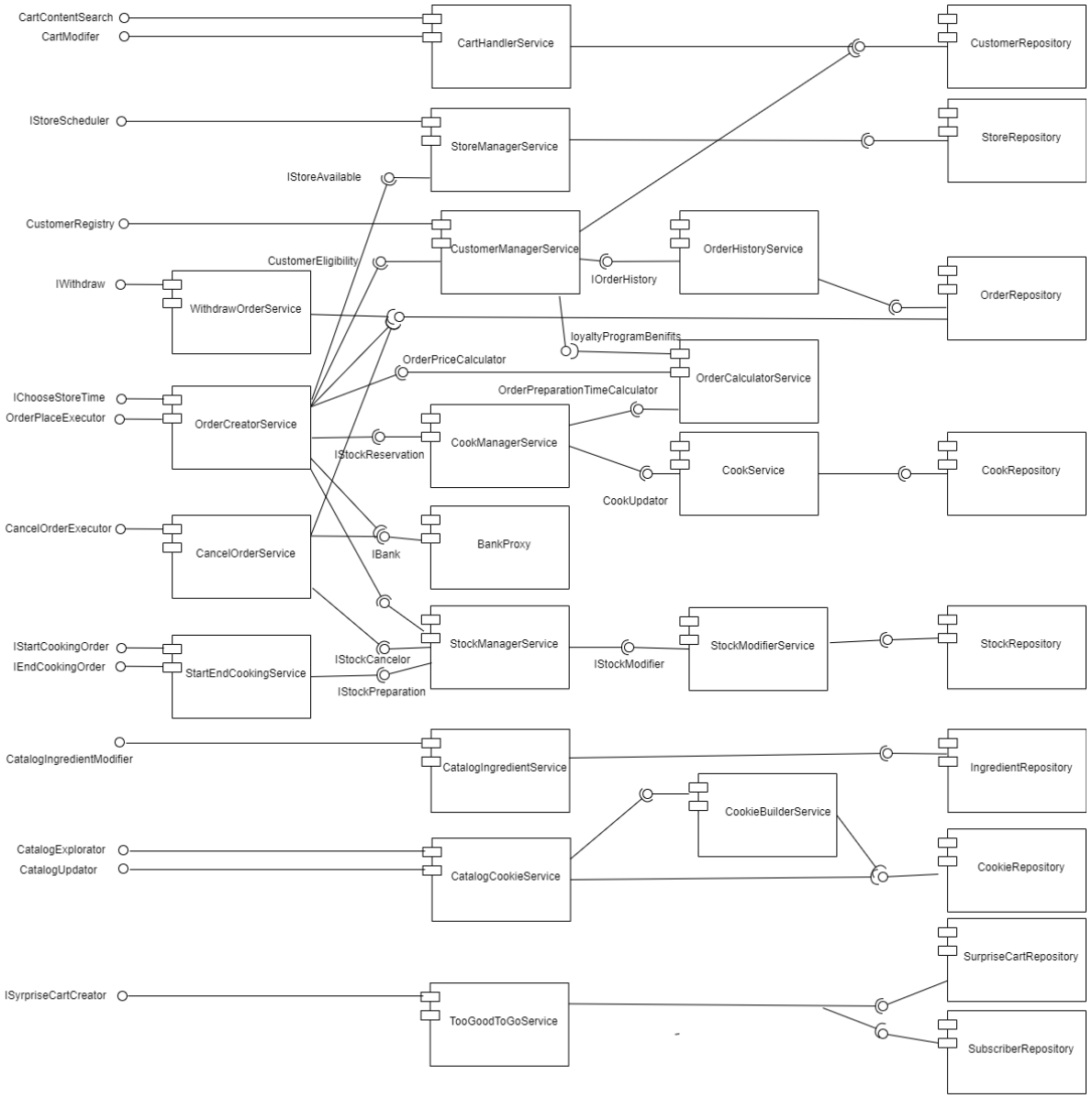
Pour chaque objet, nous avons créé au minimum un composant qui a une responsabilité spécifique (*exemple:* pour l'objet Order , nous avons créé les composants *OrderCalculatorService* et *OrderCreator*) . Ensuite , nous avons ajouté pour chaque objet un Repository qui nous sert à ajouter, modifier et supprimer les éléments de notre base de données locale .

A la fin nous avons obtenu une architecture 3-tiers basée sur des composants qui représentent l'interface utilisateur, des composants intermédiaires qui font les traitements nécessaires en se basant sur les données récupérées de la couche précédente , et les Repository pour accéder aux données.

De cette manière , nous avons réussi à réaliser le MVP , puis nous avons intégré les autres fonctionnalités en parallèle avec la migration des tests unitaires et les tests cucumbers en respectant notre architecture Spring .

Enfin, nous sommes vraiment satisfaits du résultat final du projet, nous avons bien conservé au cours de la migration en Spring une bonne couverture de test : ***90%*** (grâce aux tests unitaires et des tests cucumbers associés à nos User Story) .

- *Diagramme de composants :*



Composants	Rôle	Interfaces implémentées	Interfaces consommées
CartHandlerService	Modifier le panier en ajoutant ou supprimant des cookies et le valider	CartContentSearch CartModifier	-
StoreManagerService	Fixer les horaires de l'ouverture et fermeture du magasin, et vérifier la disponibilité du magasin	IStoreAvailable IStoreScheduler	-
CustomerManager Service	Permet de gérer la connexion et l'enregistrement des clients et gérer le LoyaltyProgram	CustomerEligibility CustomerRegistry LoyaltyProgramBenefits	IOrderHistory
OrderHistoryService	Joue le rôle d'intermédiaire avec le repository order afin de récupérer les commandes d'un client selon le statut..	IOrderHistory	-
OrderCalculator Service	Permet de calculer le temps de préparation de la commande et le prix d'une commande donnée.	OrderPriceCalculator OrderTimePreparation Calculator	-
WithdrawOrder Service	Permet de gérer le retrait de la commande.	IWithdrawOrder	
OrderCreatorService	Permet de choisir le magasin et la date de la prise de la commande ainsi que la valider en vérifiant le stock, les cuisiniers et le paiement utilisés grâce aux composants intermédiaires.	OrderPlaceExcecutor IChooseStoreTime	OrderPrice Calculator
CancelOrderService	Permet à un client d'annuler la commande.	CancelOrderExcecutor	Ibank
CookManagerService	Joue le rôle d'un composant intermédiaire permettant de gérer l'affectation des cuisiniers selon les disponibilités pour une commande.	ICookFinder	ICookUpdator
StockManagerService	Permet de réserver et préparer le stock, calculer le stock nécessaire pour une commande.	StockReservation StockPreparation StockCancelor	IStockModifier
StockModifierService	Joue le rôle d'intermédiaire entre stockManagerService et la repository, qui permet de modifier (ajouter et diminuer le stock) puis sauvegarder les modifications	IStockModifier	-
CatalogIngredient Service	Permet d'ajouter, supprimer des ingrédients	CatalogIngredient Modifier	-
CatalogCookieService	Permet de chercher, ajouter et valider des cookies.	CatalogExplorator CatalogUpdater	-
TooGoodToGoService	Permet la création de nouveaux paniers surprises et l'envoie des emails aux abonnés.	ISubscribtion ISupriseCartCreator	-

4) Auto-évaluation:

Zoubair Hamza	Ben Aissa Nadim	Saissi Omar	Boubia Marouane	El Garmit Youssef
100	100	100	100	100

Nous avons essayé de distribuer les tâches entre nous tout au long du projet en se basant sur des milestones que nous créons chaque semaine , chaque membre de groupe se charge de faire une User Story ou plus selon les fonctionnalités de chaque milestone . Pour la rédaction du rapport et la réalisation des diagrammes nous avons essayé de travailler en groupe à l'aide de l'outil diagrams.net pour avoir une meilleure vision sur le travail réalisé par chaque membre du groupe .

Pour le git nous avons fait une branche pour chaque User Story sur laquelle nous avons développé toutes fonctionnalités des user story avant de migrer les versions finales sur la branche principale "main". Pour le Spring nous avons surtout fait du "quintuple programming" contrairement au début ou une User Story était attribué à un membre de l'équipe. Car cette dernière partie était assez complexe et nécessitait plus de raisonnement et de réflexion que de code à proprement parler.

5) Conclusion :

Nous sommes dans l'ensemble satisfaits du travail accompli. Grâce à ce projet, nous avons pu découvrir de nombreux aspects de conception et ainsi approfondir nos compétences dans l'utilisation des patrons qui nous ont permis de définir un langage commun pour aider notre équipe à communiquer plus efficacement. Nous avons pu ainsi justifier les choix de conception d'une application, et ainsi connaître mieux l'architecture de son projet avant de coder grâce aux normes et diagrammes UML.
Nous avons également pu utiliser l'architecture Spring qui nous a permis d'avoir une grande flexibilité dans nos fonctionnalités afin de faciliter le développement et les tests.

De plus ce projet nous a permis de découvrir des nouvelles méthodes agiles tels que : *T-shirt-sizing* et *MoSCoW* qui nous a permis de pondérer nos user stories et ainsi bien répartir les tâches entre les différents membres de l'équipe, pour permettre de développer les fonctionnalités nécessaires au cours des sprints.

Avec du recul ce projet nous a bien fait mûrir, le fait de coder tous ensemble a été une nouvelle expérience dans l'organisation du travail. Les nouvelles fonctionnalités découvertes nous seront d'une grande aide pour nos futurs projets et nous saurons surtout mieux les utiliser pour développer de manière plus saine.