COOKIE FACTORY

Mercredi 2 Dec 2020

Shuangshuang CHEN

Yuqiao ZHAO

Fanling GU

Keyan FANG

Xin TANG

Yao Lu

Chargé de TD
Phillipe Collet

Sommaire

Introduction

Présentation du sujet

The Cookie Factory™ est leader dans la fabrication de cookies artisanaux, il est pour ses clients l'assurance d'une expérience gourmande à travers tout le pays, avec des gâteaux d'une extrême fraîcheur.
Il est possible pour n'importe qui de créer une commande, de choisir sa boutique de récupération et la date de rendez-vous puis de payer le montant dû par carte de crédit avec ou sans compte utilisateur. Et *The Cookie Factory*™ permet aux clients de faire leur sélection parmi la gamme d'ingrédients pour créer leurs cookies personnalisés!
En créant un compte utilisateur, il est possible d'adhérer au "Loyalty program", qui offre après 30 cookies commandés un discount de 10% sur la prochaine commande.
The Cookie Factory™ est aussi parvenu à un accord avec MarcelEat qui permet de livrer les commandes à toute vitesse avec juste une petite somme additionnelle.
Notre projet (cookieFactory) simule *The Cookie Factory*™ avec ses magasins et ses usines de biscuits. On vous donne des user story complet pour chaque situation dans les ventes des cookies.

Présentations des solutions

Diagramme de cas d'utilisation

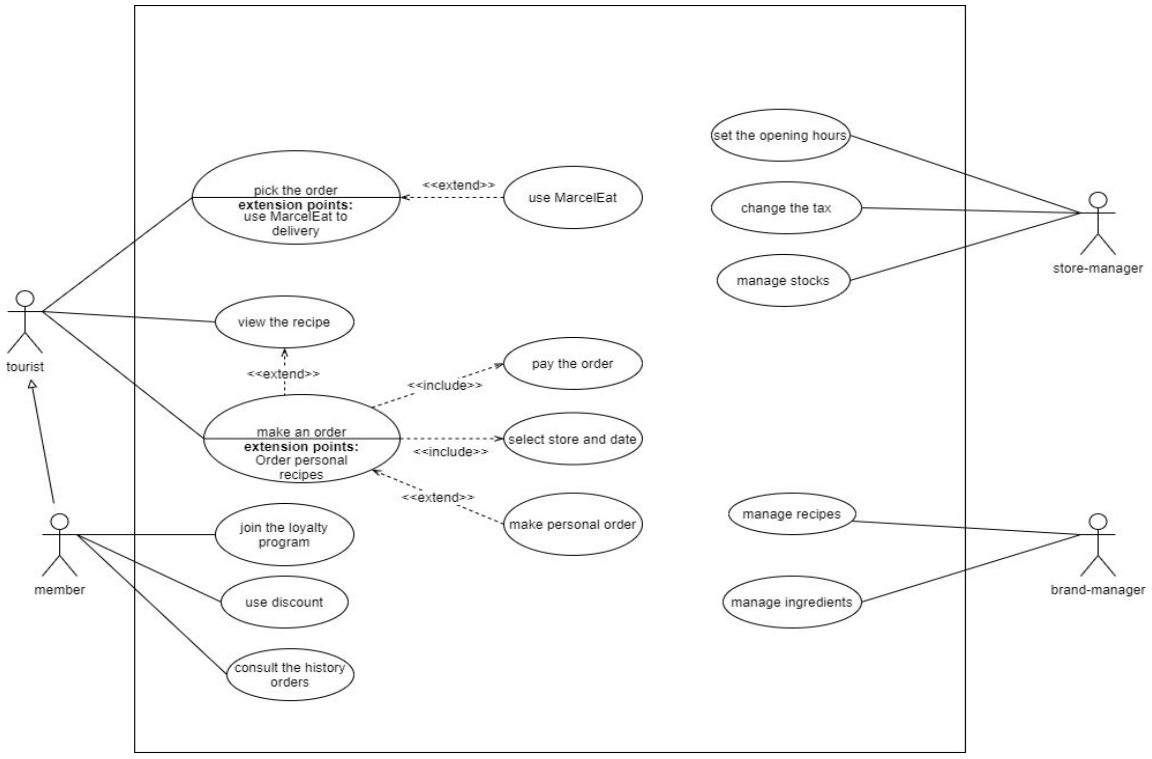
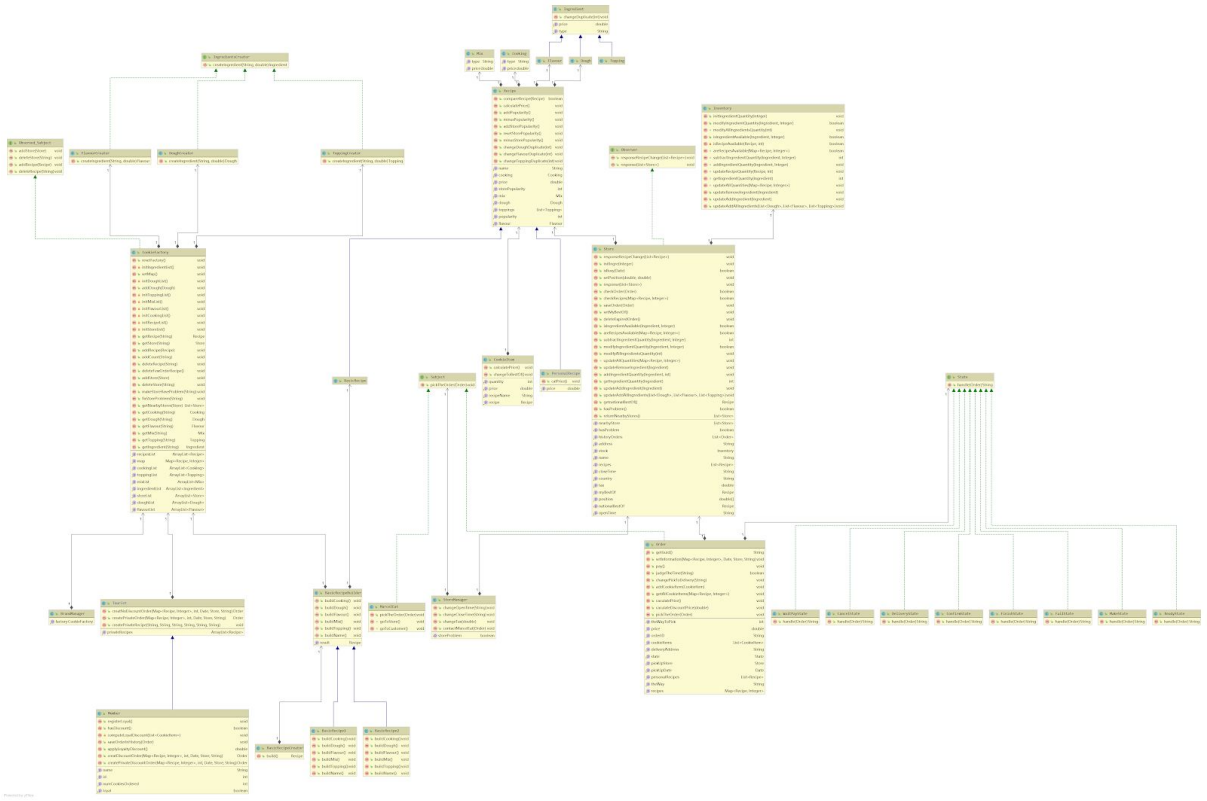


Diagramme de classe



La taille de l'image étant très grande, on a téléchargé l'image correspondante sur github, sous le répertoire:
diagram/class_diagram.png

Patrons de conception retenus

1. Builder (inclus) et Decorator (non inclus)

Pourquoi utiliser le patron?:

On a choisi d'utiliser builder pour créer le recipe de base. Parce que le recipe de base se compose de cinq parties, plus le nom du recipe, il faut six étapes pour créer, on construira le menu de base dans l'ordre de cooking, dough, flavour, mix, topping, nom. L'utilisateur n'a pas besoin de connaître les détails de la composition interne. Le patron sépare le code de construction et le code de présentation, quand on créer le nouveau recipe, on a juste besoin d'ajouter le constructeur correspondant.

Comment a été appliqué le patron?:

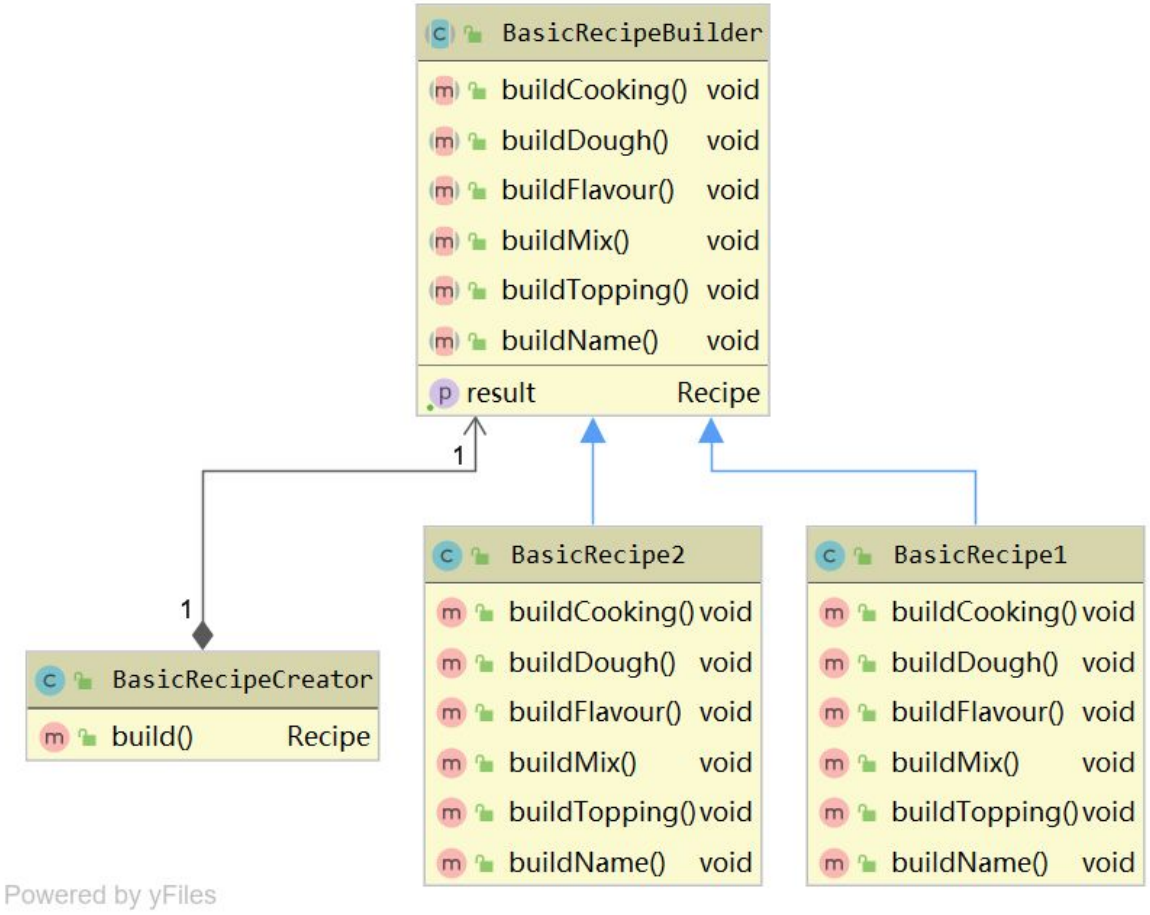
Pour appliquer ce patron, on crée une classe abstraite qui s'appelle BasicRecipeBuilder pour définir les méthodes nécessaires pour instancier un recipe de base, une classe qui s'appelle BasicRecipeCreator pour définir l'ordre de construction. La classe BasicRecipe1 et la classe BasicRecipe2 ont défini des implémentations spécifiques pour construire ces deux types de recipes.

Pourquoi on n’a pas utilisé Decorator?:

Le patron Decorator est une décoration extérieure d'une chose, tandis que le patron Decorator est un assemblage du cadre global d'une chose. Pour le recipe de base, il doit être composé de ces six parties, aucune d'elles ne peut manquer. Donc on a utilisé builder n’a pas utilisé decorator.

Diagramme de classe de ce patron:

diagram/builder.png



2. Factory pattern (inclus), Simple Factory (non inclus) et Abstract Factory (non inclus)

Pourquoi utiliser le patron?:

Pour créer les différents objets qui composent l’objet Recipe qui représente les recettes/cookies nous avons utilisé le Pattern Factory. En effet ces objets sont de différents types, on a la méthode de cuisson (cooking) la méthode de mixture (mix) et les différents ingrédients (Flavour,topping,dough) qui héritent tous de la classe (ingrédient).

Comment a été appliqué le patron?:

Pour la création de ces trois ingrédients nous devons passer par la classe principale (cookieFactory) et au moment de la création on donne le nom de l'ingrédient et son prix . Dans cette fonction on choisit ensuite le créateur concret de l’abstract factory

IngredientsCreator que l'on souhaite utiliser en fonction du type. On a donc des créateurs concrets dans la cookie factory (DoughCreator, FlavourCreator et ToppingCreator). Les créateurs vont ensuite créer l'objet avec le type correspondant ainsi que le nom et le prix en paramètre.
Nous définissons d'abord une classe de créateur d'ingrédient abstrait, puis nous définissons plusieurs classes de créateur concrètes pour implémenter les méthodes définies dans la classe de créateur d'ingrédient abstrait, qui sont utilisées pour générer trois types d'ingrédients. Le résultat de cette abstraction permet à cette structure d'introduire de nouveaux produits sans modifier la classe de créateur spécifique. Si un nouveau type d'ingrédient apparaît, il suffit de créer une classe de créateur(factory) spécifique pour ce nouveau type pour obtenir une instance de lui.

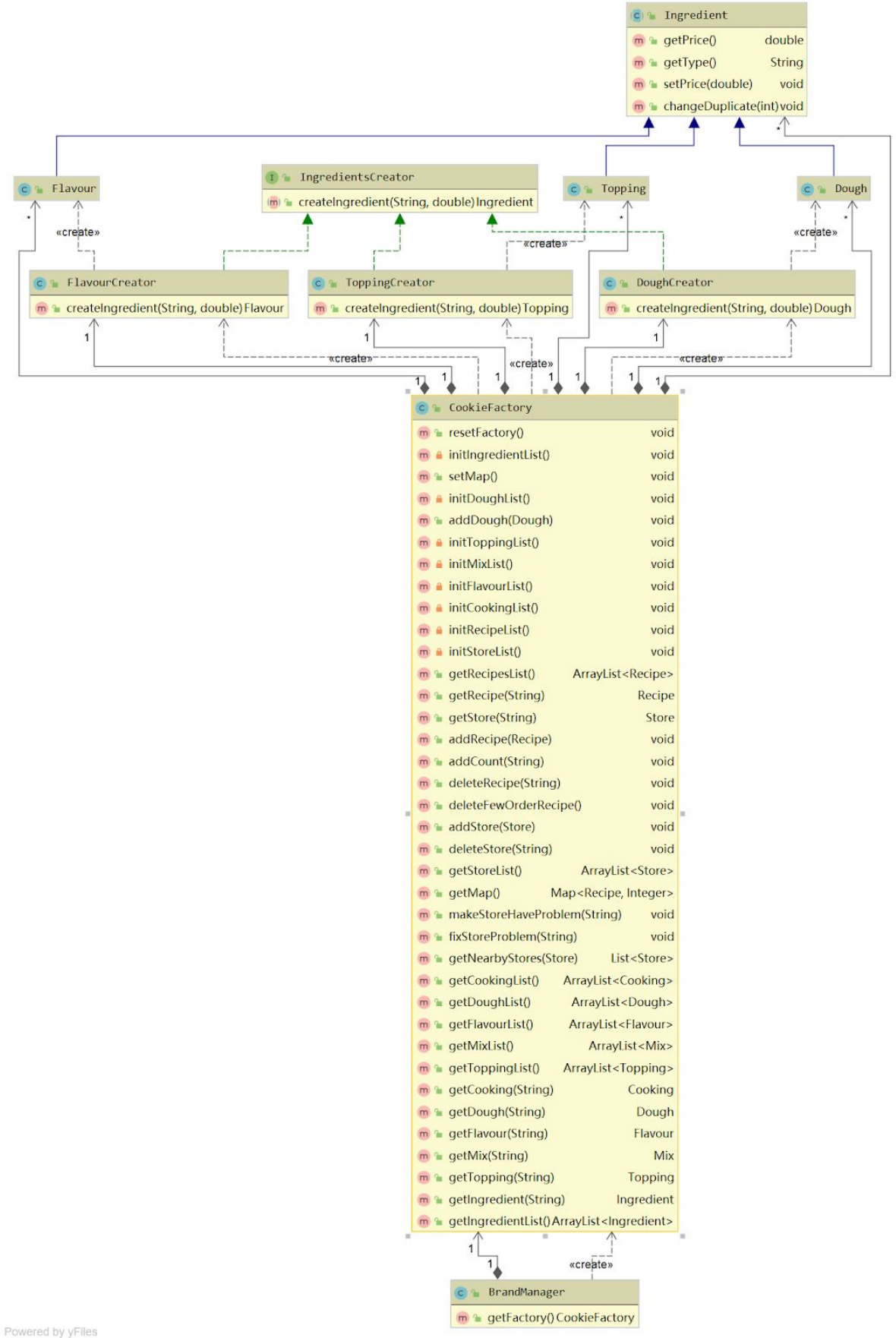
Pourquoi on n'a pas utilisé Simple Factory et Abstract Factory?:

Le patron Simple Factory et le patron Abstract Factory sont dérivés du patron Factory et appartiennent à la même catégorie; il existe trois solutions aux problèmes, et il est naturel d'utiliser une certaine solution pour résoudre un certain type de problème; mais en termes de complexité, la complexité de Simple Factory à Abstract Factory augmente progressivement..

Le patron Factory est une couche supplémentaire d'abstraction du patron d'Simple Factory, qui subdivise les objets instanciés d'un certain type aux usines correspondantes au lieu de passer des paramètres dépendants dans une usine. La fabrique simple encapsule toute la logique des objets instanciés dans une classe de fabrique. Chaque fois que la demande change, la classe de fabrique doit être modifiée séparément (violant le principe d'ouverture et de fermeture), et elle ne fonctionnera pas correctement si une exception se produit.

Le patron Abstract Factory est une abstraction du patron Factory. Il ne génère pas qu'un seul type de produit comme le patron Factory, mais une série de produits, qui peuvent être configurés de manière flexible pour différentes usines comme des pièces. L'usine abstraite augmente la complexité des nouveaux produits et doit modifier l'usine abstraite et toutes les classes Factory concrètes. Et notre seul produit est l'ingrédient. Le choix du patron Factory est donc le plus approprié.

Diagramme de classe de ce patron:
diagram/factory.png


Ingredient
getPrice() double
getType() String
setPrice(double) void
changeDuplicate(int) void
Flavour
IngredientsCreator
createIngredient(String, double) Ingredient
Topping
Dough
«create»
FlavourCreator
createIngredient(String, double) Flavour
ToppingCreator
createIngredient(String, double) Topping
DoughCreator
createIngredient(String, double) Dough
1
CookieFactory
resetFactory() void
initIngredientList() void
setMap() void
initDoughList() void
addDough(Dough) void
initToppingList() void
initMixList() void
initFlavourList() void
initCookingList() void
initRecipeList() void
initStoreList() void
getRecipesList() ArrayList<Recipe>
getRecipe(String) Recipe
getStore(String) Store
addRecipe(Recipe) void
addCount(String) void
deleteRecipe(String) void
deleteFewOrderRecipe() void
addStore(Store) void
deleteStore(String) void
getStoreList() ArrayList<Store>
getMap() Map<Recipe, Integer>
makeStoreHaveProblem(String) void
fixStoreProblem(String) void
getNearbyStores(Store) List<Store>
getCookingList() ArrayList<Cooking>
getDoughList() ArrayList<Dough>
getFlavourList() ArrayList<Flavour>
getMixList() ArrayList<Mix>
getToppingList() ArrayList<Topping>
getCooking(String) Cooking
getDough(String) Dough
getFlavour(String) Flavour
getMix(String) Mix
getTopping(String) Topping
getIngredient(String) Ingredient
getIngredientList() ArrayList<Ingredient>
BrandManager
getFactory() CookieFactory
Powered by yFiles

3. Observer (inclus) et Mediator (non inclus)

Pourquoi utiliser le patron?:

Lorsqu'il existe une relation un-à-plusieurs entre les objets, nous utilisons le patron Observer. Par exemple, lorsqu'un objet est modifié, ses objets subordonnés seront automatiquement notifiés. Le patron Observer est le patron de comportement. Par conséquent, dans nos projets, nous l'utilisons dans les magasins et les usines.

Définissez des dépendances un-à-plusieurs entre les objets. Lorsque l'état d'un objet change, tous les objets qui en dépendent seront notifiés et mis à jour automatiquement.

Comment a été appliqué le patron?:

Notre magasin est un observateur de l'usine, et chaque fois que l'usine modifie le menu, le magasin qui l'observe se met à jour automatiquement. De plus, lorsque l'usine ajoute ou supprime un magasin, le magasin d'observation met automatiquement à jour la liste des magasins à proximité.

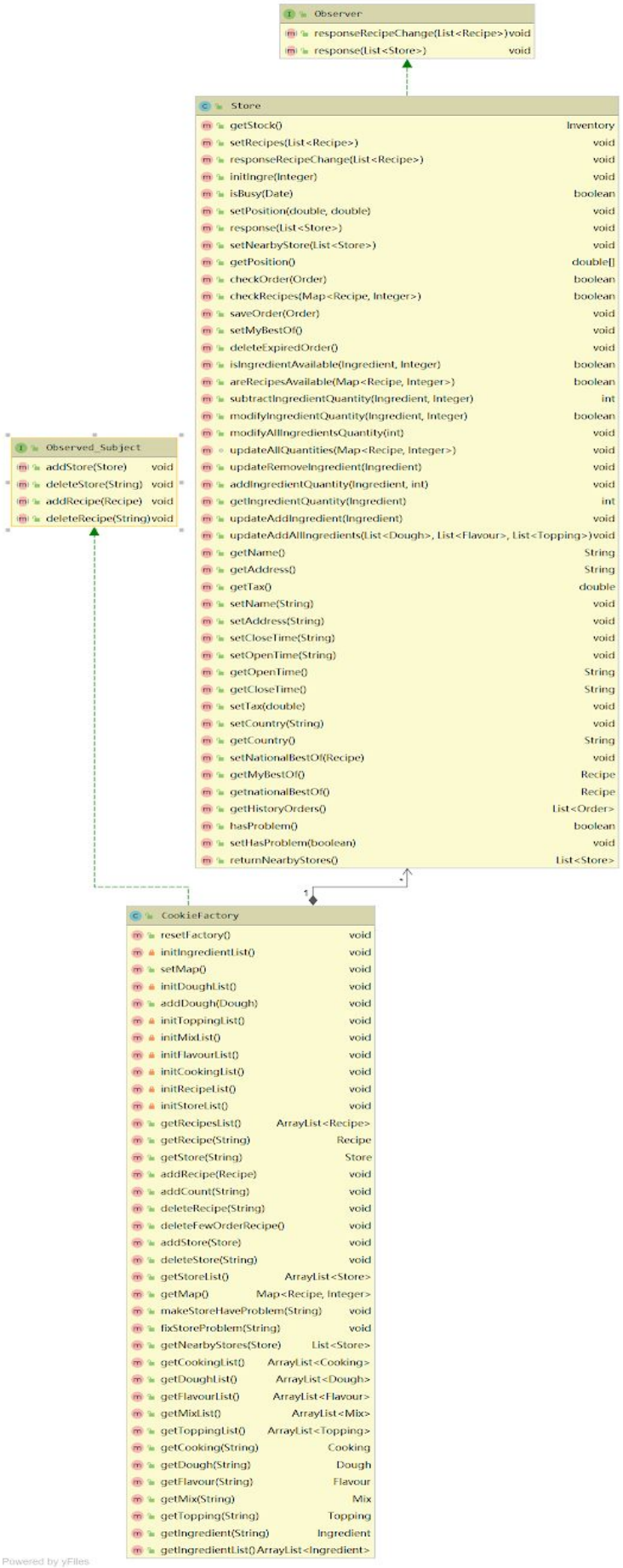
Dans la classe observée, il existe une ArrayList contenant des observateurs. L'observateur et l'observé sont couplés de manière abstraite. Établissez un mécanisme de déclenchement.

Pourquoi on n’a pas utilisé Mediator?:

Dans un premier temps, nous avons également testé le patron Mediator, mais comme notre magasin n'a besoin que de recevoir et de mettre à jour les informations de menu de l'usine, il n'y a pas besoin d'interaction complexe avec l'usine. Par conséquent, nous adoptons cette méthode d'observation passive, qui peut non seulement économiser la consommation de ressources de la recherche répétée des informations de menu, mais également atteindre une vitesse de retour plus élevée.

Diagramme de classe de ce patron:

diagram/oberver.png


Observer
responseRecipeChange(List<Recipe>)void
response(List<Store>) void
Store
getStock() Inventory
setRecipes(List<Recipe>) void
responseRecipeChange(List<Recipe>) void
initIngre(Integer) void
isBusy(Date) boolean
setPosition(double, double) void
response(List<Store>) void
setNearbyStore(List<Store>) void
getPosition() double[]
checkOrder(Order) boolean
checkRecipes(Map<Recipe, Integer>) boolean
saveOrder(Order) void
setMyBestOf() void
deleteExpiredOrder() void
isIngredientAvailable(Ingredient, Integer) boolean
areRecipesAvailable(Map<Recipe, Integer>) boolean
subtractIngredientQuantity(Ingredient, Integer) int
modifyIngredientQuantity(Ingredient, Integer) boolean
modifyAllIngredientsQuantity(int) void
updateAllQuantities(Map<Recipe, Integer>) void
updateRemoveIngredient(Ingredient) void
addIngredientQuantity(Ingredient, int) void
getIngredientQuantity(Ingredient) int
updateAddIngredient(Ingredient) void
updateAddAllIngredients(List<Dough>, List<Flavour>, List<Topping>) void
getName() String
getAddress() String
getTax() double
setName(String) void
setAddress(String) void
setCloseTime(String) void
setOpenTime(String) void
getOpenTime() String
getCloseTime() String
setTax(double) void
setCountry(String) void
getCountry() String
setNationalBestOf(Recipe) void
getMyBestOf() Recipe
getnationalBestOf() Recipe
getHistoryOrders() List<Order>
hasProblem() boolean
setHasProblem(boolean) void
returnNearbyStores() List<Store>
Observed_Subject
addStore(Store) void
deleteStore(String) void
addRecipe(Recipe) void
deleteRecipe(String) void
CookieFactory
resetFactory() void
initIngredientList() void
setMap() void
initDoughList() void
addDough(Dough) void
initToppingList() void
initMixList() void
initFlavourList() void
initCookingList() void
initRecipeList() void
initStoreList() void
getRecipesList() ArrayList<Recipe>
getRecipe(String) Recipe
getStore(String) Store
addRecipe(Recipe) void
addCount(String) void
deleteRecipe(String) void
deleteFewOrderRecipe() void
addStore(Store) void
deleteStore(String) void
getStoreList() ArrayList<Store>
getMap() Map<Recipe, Integer>
makeStoreHaveProblem(String) void
fixStoreProblem(String) void
getNearbyStores(Store) List<Store>
getCookingList() ArrayList<Cooking>
getDoughList() ArrayList<Dough>
getFlavourList() ArrayList<Flavour>
getMixList() ArrayList<Mix>
getToppingList() ArrayList<Topping>
getCooking(String) Cooking
getDough(String) Dough
getFlavour(String) Flavour
getMix(String) Mix
getTopping(String) Topping
getIngredient(String) Ingredient
getIngredientList() ArrayList<Ingredient>
Powered by yFiles

4. Proxy (inclus) et Adapter (non inclus)

Pourquoi utiliser le patron?:

Le patron Proxy offre un autre moyen d'accéder à l'objet cible. C'est-à-dire d'accéder à l'objet cible via l'objet proxy. L'avantage est que sur la base de la réalisation de l'objet cible, des opérations fonctionnelles supplémentaires peuvent être améliorées, c'est-à-dire que l'objet cible peut être développé Fonction.

Donc, on utilise ce patron pour simuler la fonction selon laquelle les commandes peuvent être livrées via MarcelEat. Lorsqu'un client choisit MarcelEat pour livrer une commande, le commerçant doit contacter MarcelEat pour effectuer la livraison. MarcelEat équivaut à un objet proxy et l'ordre est l'objet cible.

Un avantage du patron Proxy est de fournir une méthode d'interface unifiée vers l'extérieur, et la classe proxy implémente des comportements d'opération supplémentaires pour la classe réelle dans l'interface, afin que le système puisse être étendu sans affecter les appels externes.

Comment a été appliqué le patron?:

Chaque fois que l'utilisateur a besoin de livrer à la porte, la classe BrandManager utilise l'objet proxy MarcelEat. Dans la classe d'interface proxy, il existe une fonction proxy appelée pickTheOrder. L'utilisateur peut réaliser la fonction de livraison à domicile grâce à la fonction de la classe Order. Il n'est pas nécessaire de modifier la fonction pickTheOrder d'origine dans la classe Order. On peut modifier la fonction par proxy.

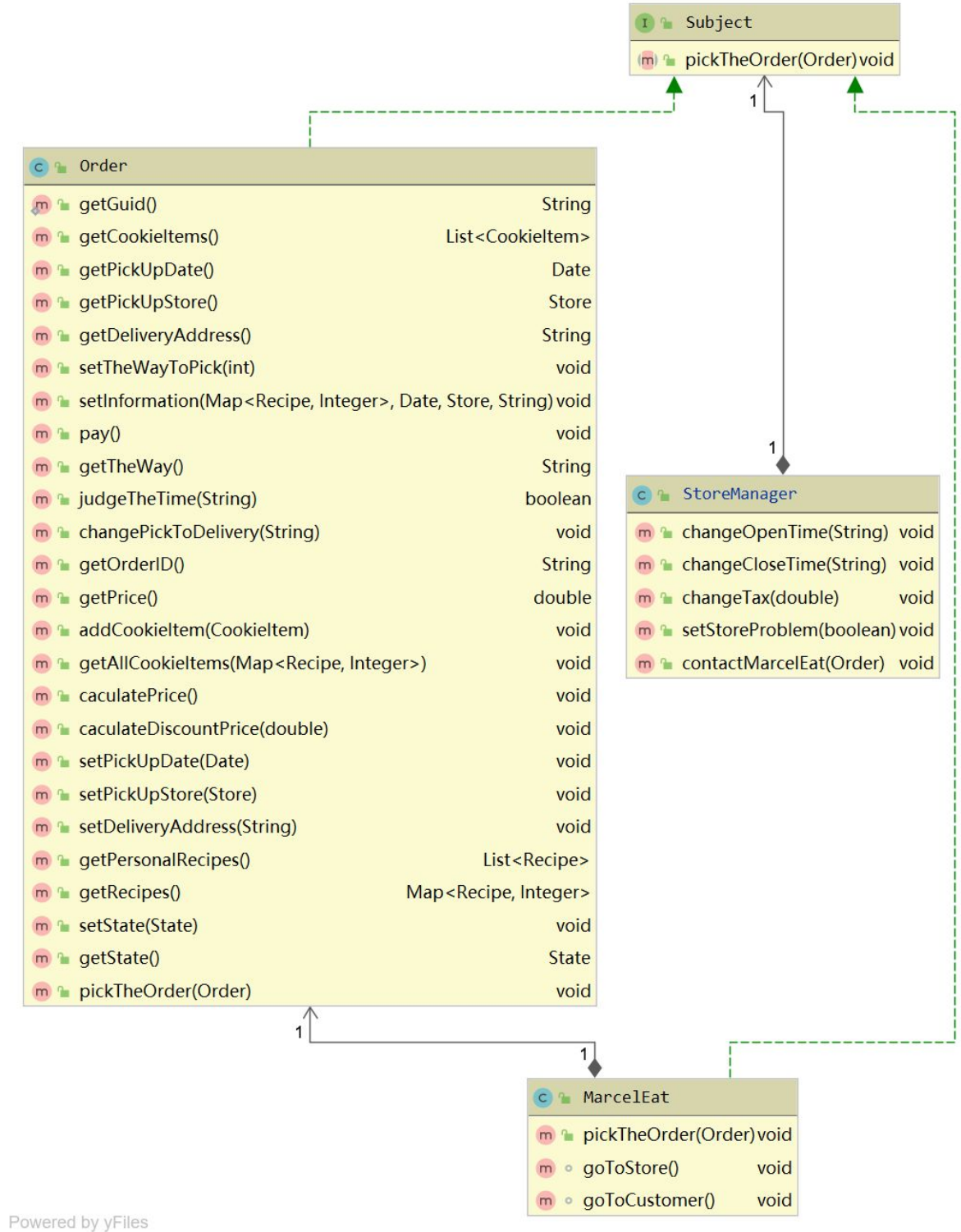
D'ailleurs, dans ce programme, utiliser le patron Proxy équivaut à simuler l'api, car MarcelEat est un service tiers, il ne fait pas partie du magasin.

Pourquoi on n'a pas utilisé Adapter?:

Au départ, nous avons également considéré le patron Adapter, qui convertit l'interface d'une classe en une autre interface attendue par le client. Mais dans la fonction de livraison, MarcelEat ressemble plus à un agent de magasin, chargé d'aider le magasin à livrer les marchandises. Il a non seulement implémenté la fonction pickTheOrder dans la classe Order, mais a également ajouté quelques actions, sans changer complètement cette fonction. Finalement, Proxy Pattern est mieux.

Diagramme de classe de ce patron:

diagram/proxy.png



5. State (inclus) et Strategy (non inclus)

Pourquoi utiliser le patron?:

En réalité, lorsqu'un utilisateur commande quelque chose en ligne, le magasin doit traiter une série de commandes. Les clients comprennent la progression du statut de la commande. Donc, dans cette conception de programme, nous utilisons le patron State pour

gérer les commandes. L'action de la commande changera en fonction du statut. En patron State, nous extrayons l'interface d'état, puis chaque état et implémentons l'interface. Chaque état contient la méthode correspondante du comportement pour réagir en conséquence; tandis que l'ordre contient tous les états et fait référence à l'état Interface pour faire fonctionner différents états.

Comment a été appliqué le patron?:

Dans ce programme, nous avons créé l'interface État, puis nous avons défini différents états et y avons défini les actions correspondantes. Ajoutez ensuite l'attribut state à la classe de commande, de sorte que la commande contienne tous les états.

Le client n'interagit pas avec le statut, et c'est le travail de la commande de bien comprendre le statut. Bien que l'utilisation du patron State augmente toujours le nombre de classes dans la conception, cela se fait au détriment de l'évolutivité et de la flexibilité du programme. Si le code n'est pas unique et que différents états peuvent être ajoutés en continu plus tard, alors la conception du patron State en vaut vraiment la peine.

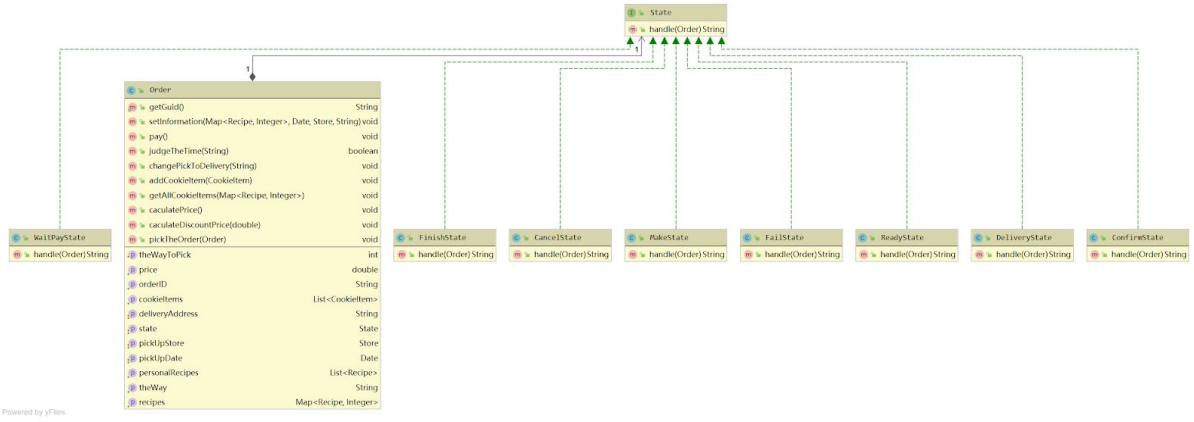
Pourquoi on n'a pas utilisé Strategy?:

Nous avons également considéré le patron Strategy. La structure du patron State et du patron Strategy est similaire: ils sont tous deux basés sur le principe O (principe d'ouverture et de fermeture) dans les principes de conception SOLID, mais leurs intentions sont complètement différentes. La différence la plus fondamentale est que le patron Strategy consiste à résoudre plusieurs solutions au même problème et que ces différentes solutions ne sont pas liées les unes aux autres; le patron State est différent et le patron State nécessite des corrélations entre les états afin de réaliser une transition d'état.

Dans notre projet, le comportement de la commande est déterminé en fonction de l'état, nous utilisons donc le patron State.

Diagramme de classe de ce patron:

diagram/state.png



Rétrospective

Évolution de la conception de notre application

Dans ce projet, on a utilisé plusieurs patterns pour réaliser différentes tâches de Cookie Factory. Avec l'aide de ces patterns on a bien structuré notre programme.

Au début, nous nous sommes concentrés sur la réalisation de la fonction et dans le processus de réalisation de la fonction, nous avons ajouté le modèle de conception correspondant en fonction de l'objectif de la fonction et du processus de réalisation.

Par exemple, dans la réalisation d'une série de processus depuis la passation d'une commande jusqu'à la fin de la commande, nous considérons que l'utilisation du patron State permet de mieux réaliser cette fonction, nous avons donc ajouté le patron State.

Cette façon nous permet de bien comprendre les patrons de conception que nous utilisons dans cette partie, de sorte que les patrons de conception que nous utilisons puissent correspondre à nos projets.

En même temps, nous avons bien réparti notre temps.

Termes de découpage

En termes de découpage, Nous avons principalement divisé le projet en deux parties, partie de base et partie d'extension. Pour la partie de base, nous avons passé environ deux semaines. Tout d'abord, nous avons déterminé le diagramme de cas d'utilisation et le diagramme de classe de base après discussion, et avons écrit quelques "use story" basées sur ces derniers, puis avons commencé à écrire le code, et finalement terminé la base de la "Cookie Factory". Ce programme peut fonctionner normalement et a réussi le test Junit associé et le test concombre lié à ces "use story". Pour la partie d'extension, nous avons passé trois semaines. Chaque semaine, nous étendons les fonctions du programme précédent et écrirons des nouvelles "use story". Cela garantit que notre programme fonctionne normalement chaque semaine.

Par exemple, au cours de la première semaine, nous avons ajouté que les utilisateurs peuvent modifier le magasin sélectionné au début en raison de problèmes techniques dans le magasin après avoir passé une commande, et ajouté des options de service de livraison tiers. La deuxième semaine, nous avons ajouté une nouvelle raison pour laquelle les utilisateurs peuvent mettre à jour le magasin: le magasin manque d'ingrédients. Dans la troisième semaine, nous avons amélioré la fonction de livraison afin que les utilisateurs puissent non seulement choisir la livraison au début, mais également passer du ramassage à la livraison avant l'heure spécifiée.

Amélioration de notre conception et notre manière

Pendant cette période, on a beaucoup appris sur les patterns mais on est sûr qu'il y a encore des améliorations qui puissent être effectuées. Et quelqu'un d'entre nous a une mauvaise habitude de partager les codes avant de les tester et vérifier qu'elles sont toutes compilables et sans erreur, cela a causé des petits problèmes pendant la programmation mais heureusement on a pu trouver des solutions sans trop d'effort. Nous devons encore

améliorer la structure hiérarchique du programme pour approfondir la structure, notre structure actuelle est plate.

Auto-Evaluation

Au cours de cette période, on a une connaissance et une compréhension plus complètes de l'UML orienté objet et nous avons non seulement une compréhension plus profonde d'UML, mais également une meilleure compréhension des Patterns comme Factory Pattern, Observer Pattern, State Pattern etc.

Prénom Nom	Travail réalisé	600 points
Shuangshuang CHEN	La diagramme usecase, les classes, par exemple CookieFactory, Store, Order, etc. le patron Builder et les classe associés, les Junit tests et les cucumber tests.	130
Yuqiao ZHAO	Les classes, par exemple CookieFactory, CookieItem, Store, Order, etc. Le patron Observer et les classes associées, les Junit tests et toutes les cucumber tests.	130
Fanling GU	Les classes, par exemple Customer CookieFactory, Store, Order, etc.Le patron state et proxy et les classes associé, les Junit tests et les cucumber tests	130
Keyan FANG	Les classes Store/storeManager et les test Junit et cucumber correspondants pour Store	70
Yao LU	Les classes “member” “inventory” et leurs Junit test. 2 Cucumber tests pour “member”. Factory pattern pour les ingrédients	70
Xin TANG	Les fonctions “ajouter/supprimer” les recettes. La fonction de créer commande privée et augmenter le price.	70