POLYTECH NICE-SOPHIA

Université Nice Sophia Antipolis

Rapport conception logicielle

Cookie Factory

SI4 : 2019 - 2020

GIUNTINI Romain - LEGRIFI Amine - PERES Richard - SABRI Anass

Sommaire

Diagramme des cas d’utilisation

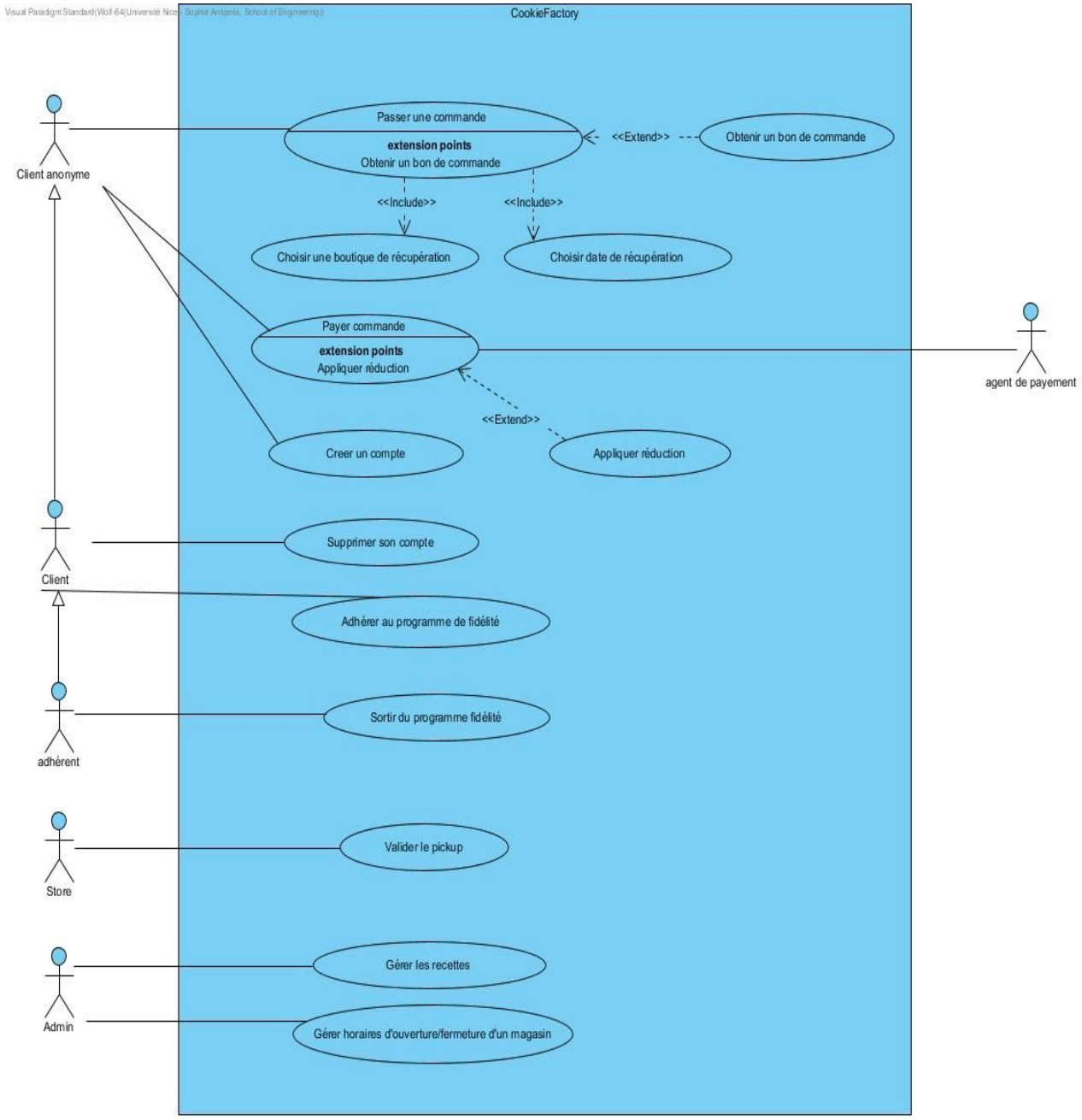

Diagramme des classes

(cf class_diagram.svg)

Avancée fonctionnelle et tests cucumber

- ## Gestion des magasins

Le manager d'un store peut définir les heures d'ouverture/fermeture de chaque jour. Pour cela nous avons fait une petite vérification des nouvelles chronos, par exemple si le manager essaye de mettre l'heure d'ouverture à 18h alors que le magasin ferme à 16h le système lance une exception "CannotChangeOpeningHourException", et dans le cas ou le changement concerne l'heure de fermeture le système lance l'exception "CannotChangeClosingHourException".

Les scénarios testés qui implémentent cette fonctionnalité sont dans le fichier ("StoreTime.feature").

- ## Gestion des stocks

Le système gère le stock d'ingrédients et de boisson. Chaque boutique a son propre stock pour pouvoir vérifier si elle peut ou non assurer une commande. Dans le cas où le stock n'a pas assez d'ingrédients pour assurer la commande, une exception "CannotOrderOutOfStockException" est lancée et la commande ne passe pas.

Cependant, une pré-vérification de stocks disponibles est effectué dans le Builder de la commande lors de la sélection du store de retrait.

Cette fonctionnalité est testée par les scénarios dans le fichier ("Stock.feature").
La boutique peut aussi ajouter des ingrédients qu'ils leur manquent(augmenter la quantité d'un ingrédient/ ajouter un nouvel ingrédient). Et pour vérifier cette fonctionnalité nous avons fait 4 scénarios de gestion de stock dans le fichier ("AddIngredient.feature")

- ## Gestion du calcul des prix

Tous nos produits ont une marge commune qui lui sert à calculer leur prix pour toutes les boutiques. Chaque produit peut individuellement avoir, s'il le faut, une marge supplémentaire par rapport à la marge commune à tous les produits. Ainsi les prix hors taxes sont tous les mêmes d'une boutique à une autre.
Ensuite pour calculer les prix des commandes, on a représenté notre commande avec une liste de ProductOrder qui représente la liste des items d'une commande. Chaque ProductOrder associe un Product avec une quantité pour pouvoir par exemple commander un même cookie cinq fois sans avoir une liste de produits contenant cinq fois le même cookie.

- ## Discounts

Le système gère les discounts. Il permet tout d'abord à tous les clients membres du Loyalty Program d'avoir un réduction immédiate de 10% sur leur prochaine commande après avoir commandé au moins 30 cookies.
Il permet aussi d'avoir une réduction de 10% sur une commande de plus de 100 cookies si le client a donné le code "EVENT" au préalable.
Il permet d'avoir une réduction de 5% sur une commande si le client fournit un code CE d'une compagnie agrémentée par le système.
Nous avons utilisé différents scénarios enregistrés dans le fichier ("Discount.feature") pour représenter et tester ces fonctionnalités.

- ## Gestion des comptes/ utilisateurs

Un utilisateur peut être anonyme(utilise que son nom et email pour récupérer sa commande), enregistré dans le système(avec un mot de passe) ou un client membre du système de fidélité de CookieFactory. Dans cette partie il y avait plusieurs cas qu'il faudrait prendre en compte comme un client anonyme ne peut pas s'abonner au programme de fidélité sans être enregistré, et un client non membre ne peut pas désabonner du coup nous avons utilisé les exceptions "CannotAdhereException", "CannotRegisterException" et "CannotUnadhereException".

Ici nous avons utilisé plusieurs scénarios (clients qui s'abonnent/ désabonnent au programme de fidélité, qui s'enregistrent dans le système…) qui sont tous définis dans le fichier ("ClientRegister.feature").

- ## Choisir l'heure pour récupérer une commande

Le client à la possibilité de choisir une heure pour récupérer sa commande lors de sa création. Et pour éviter de choisir un horaire où la boutique est fermée, une exception "CannotOrderStoreWillBeClosedException" sera lancée pour le prévenir.
Nous avons créé 2 scénarios pour tester cette fonctionnalité qui sont définis dans le fichier ("ChosePickUpTime.feature").

- ## Payer une commande

Une commande ne peut pas être ramassée sans être payée. Ici nous n'avons pas traité le cas où le paiement est invalide, une fois que le client est demandé de payer sa commande le statut de la commande passe à "waiting for pick up". De ce fait nous avons implémenté cette fonctionnalité dans un seul scénario("PayOrder.feature")

Thème	Fonctionnalité	Test correspondant
Boutique	Je peux changer les horaires d'ouverture de ma boutique	StoreTime.feature
Boutique	Je ne peux pas commander en dehors des horaires d'ouverture	ChosePickUpTime.feature
Boutique	Je peux retirer ma commande	OrderPickUp.feature
Stock	Je peux ajouter des nouveaux ingrédients dans le stock/ augmenter le nombre d'articles d'un ingrédient	AddIngredient.feature
Commande	Je peux commander sans compte sur la plateforme CoD	Order.feature(scénario 1)
Commande	Il n'est pas possible de passer une commande qu'on ne peut pas préparer	Stock.feature
Commande	Je peux payer ma commande	PayOrder.feature
Commande	Je peux passer une commande avec des cookies classiques et personnalisés(avec respect de contraintes de fabrication)	Order.feature (Cookies classiques : scénarios 1&2, cookies personnalisés : scénario 3)
Commande	Je peux recevoir des packs de cookies selon le nombre de cookies commandés et associer des boissons à ces packs	Pack.feature
Discount	Je peux bénéficier des 10% tous les 30 cookies	Discount.feature
Discount	Je peux bénéficier d'une réduction avec le bon EVENT à partir de 100 cookies commandés	Discount.feature
Discount	Je peux bénéficier d'une réduction avec le CE de mon entreprise	Discount.feature

Patrons de conception utilisés

- ## Factory

Nous avons utilisé plusieurs factory dans notre projet, car la création de plusieurs objets dépendait du contexte. On a une factory pour les cookies et les boissons qui valide les ingrédients et les cooktechnique passés en paramètre et nous crée soit un cookie personnalisé soit elle nous renvoie un cookie déjà existant si on passe un string en paramètre qui correspond au nom du cookie.

Nous avons aussi utilisé une factory de commandes, cette factory a pour objectifs de nous construire des orders simple ou des packs en fonction du nombre de cookies demandé, par exemple si un client veut 11 cookies et une boisson, la factory va créer un pack de 10 cookies qui correspond à la taille du grand pack et un simple order d'une seule cookie et un order de boisson qui sera dans le pack.

Les factory que nous avons, sont un peu sous exploitée, on aurait pu par exemple rajouter une hiérarchie (via l'héritage) afin de les rendre plus ouvertes à de futures améliorations.

- ## Singleton

Nous avons choisi d'utiliser le Singleton pour notre façade qu'on peut le considérer comme une mini façade de notre projet. Il permet à tout le monde d'accéder à la même instance de la base de donnée et garantit aussi qu'une seule instance de la BD sera créée.
Notre base de donnée aurait pu être une vrai façade afin de limiter le nombre d'acteurs qui modifient la base de donnée.

Nous avons utilisé le pattern Singleton également pour avoir database persistantes pour nos tests. Le fait d'utiliser un Singleton permet d'avoir une instance commune pour tout le système et ainsi d'avoir une liste de cookies dont tous les magasins doivent disposer.

Il est cependant plus compliqué de tester (en particulier de manière fonctionnelle) ce Singleton, d'autant qu'une erreur sur ce dernier peut avoir de lourdes répercussions sur le reste du programme.

- ## Adapter

Afin de cacher les actions du de l'OrderBuilder, on a utilisé un Adapter "OrderInterface" qui décompose en quelque sorte le processus du build et le cache.
Grâce à cet Adapter, on peut choisir de cacher ou non une fonctionnalité de l'OrderBuilder pour un Customer.

- State

Nous avons choisi de mettre en place des états de Customer pour qu'en fonction de son état "Anonymous", "Registered" ou "Member", ses méthodes aient un fonctionnement différent. Par exemple, un customer anonyme peut s'enregistrer mais pas adhérer au LoyaltyProgram, un customer enregistré ne peut pas se réenregistrer ou encore seul les membres du LoyaltyProgram peuvent en sortir.

- Builder

Nous avons choisi d'appliquer le pattern Builder au processus de création d'une commande qui s'avère complexe, pour mieux encadrer son instanciation. Il nous sert dans ce cas-ci à rendre une commande valide et complète car il est indispensable d'éviter d'avoir des commandes incomplètes. Si cela avait était le cas, on n'aurait eu aucun autre choix que d'implémenter des setter sur ces parties manquantes d'une commande et cela équivaudrait à pouvoir instancier une commande non valide. Avec ce pattern, on a choisi de déléguer la vérification des informations d'instanciation d'une commande à son Builder.
On a ainsi un builder se composant de 4 parties :

- ➔ Une partie qui actionne le parsage des produits désiré en une liste ProductOrder spécifique.
- ➔ Un autre partie qui s'occupe de lier une boutique officielle en vérifiant l'adresse du magasin donnée par le client. Il va également vérifier si le magasin voulut peut ou non assumer une commande de la liste des produits désiré spécifiée précédemment.
- ➔ Une partie optionnelle visant à ajouter des discounts si le client peut y avoir droit
- ➔ Et enfin le build de la commande qui fait les dernières vérifications et ajoute certains discounts si le client y a droit ou non, ajoute la commande au client et ajoute la commande au magasin voulut.

Patrons de conception retenus mais pas implémentés

- Observer

Pour les stocks nous aurions pu utiliser le pattern Observer, et notifier tous les Stores lorsqu'on ajoute des produits au stock. Cela permet aux Stores d'avoir toujours la dernière liste des produits qui sont disponible, sont passés par la base de donnée.

- Façade

L'utilisation du patron Façade aurait pu être intéressante et utile mais pas obligatoire, on pourrait l'utiliser comme une interface pour passer toutes les opérations qu'un Customer peut faire (makeOrder, register, adhérer) ou la modification de la Recipe et l'ajout d'un Customer par exemple. Cela aurait également eu pour effet de "cacher" la partie complexe algorithmique qui fait tout le travaille en arrière-plan.
La façade pourrait travailler avec le singleton pour assurer qu'un seul acteur est en train de faire une modification pour éviter la sur-écriture.

- Adapter/Proxy

L'implémentation d'un patron Adapter ou Proxy aurait aidé à ne pas utiliser les hashmaps pour passer une commande. Cela peut aussi aider le client à créer des semblants de product voir productOrder qui ne peuvent pas impacter "négativement" le système ce qui ne cause pas de problème de responsabilité dans la classe customer.

- Command pattern

Nous avons pensé à utiliser le pattern Command qui aurait pu être intéressant pour supprimer le state d'order et le remplacer par l'état de la commande qui pourrait être soit (en cours, en attente de paiement, payé, en attente de pick up…). Néanmoins, l'ajout de Command aurait à notre point de vue ajouté trop de complexité pour une fonctionnalité qui n'en nécessite pas autant, vu qu'il nécessite trop de modification et le gain apporté n'est pas trop intéressant.

Rétrospective et voies d’amélioration

- ## Customer

Dans notre modèle un Customer est représenté d’une manière très simple, on voudrait ajouter d’autre fonctionnalité qui permettent d’avoir plus d’information sur le Customer et aussi de pouvoir modifier ses informations.
On voulait aussi mettre en place de nouveaux bonus, par exemple des bonus qui s’appliquent selon le type d’ancienneté.
On peut aussi penser à ajouter de nouveaux états de Customer par exemple un Customer peut devenir un LoyalCustumer s’il effectue un certain nombre d’Order, et ainsi il peut bénéficier de quelques réduction.

- ## Store

On peut dans une future amélioration définir des heures d’ouverture/ fermeture pour les matins et les soirées à la place des heures pour toute la journée.
Une autre amélioration peut être de faire des taxes variables soit par rapport au jours ou par rapport au type de commandes ou par rapport aux ingrédients mais ce dernier cas nécessite un changement dans les ingrédients et chaque ingrédient aura sa propre tax pour évider de couples les ingrédients avec les stores.

- ## Système de commande

Nous avons essayé d’appliquer une factory pour construire les recettes et les commandes, mais cela n’était pas bien organisé, du coup il faudra revoir l’implémentation de factory dans ces deux fonctionnalités.

La gestion de pickup et payement est trop simple et elle manque de détails, car en passant la commande on passe directement au demande de paiement puis la demande de récupérer la commande, les seules vérifications que nous avons utilisées sont juste des vérifications de l’état actuel de la commande avant de passer à l’état suivante.

- ## Stocks

Nos stocks sont capables de stocker n’importe quel produit consommable (comme les ingrédients et les boissons), du coup il est possible dans le future de permettre aux magasins de stocker des cookies et des produits fabriqués à condition que leur type devient consommable.

- Statistiques

Parmi les nombreuses fonctionnalités manquantes, il y a celle visant à afficher les statistiques de vente des magasins. On s'est tout de même penché sur la question et nous avons pensé à un Singleton car il suffisait ici d'une simple instance unique qui calculerait les stats d'un magasin à un moment T. Dans ce cas-là le Singleton est approprié car on demande simplement de récupérer des données immédiatement. Il n'y aurait donc plus qu'à demander à un magasin de nous donner ces statistiques en utilisant le Singleton.
D'un autre côté, nous avons aussi regardé vers l'Observer. Ce dernier nous aurait permis d'avoir un objet Statistique qui regrouperait tous les statistiques de tous les magasins du système. Ainsi, à chaque nouvelle commande, un magasin pourrait notifier cette objet Statistique qui se mettrait a jour automatiquement via les donnée notifiées.

Résumé des avantages et des inconvénients de ces deux solutions :

Singleton	
Avantage	**Inconvénient**
Ne pas recalculer a chaque nouvelle commande les statistiques	Calcul des statistiques uniquement sur demande
On peut faire le calcul de statistique uniquement quand on en a besoins	Le calcul des statistiques prend de temps dans le sens ou il faut faire le calcul sur toutes les données à chaque fois

Observer	
Avantage	**Inconvénient**
Calcul automatique des statistiques	Plus difficile de faire des sauvegardes
Aucun problème de concurrence	Plus difficile de faire un historique
Facilité d'optimisation des calculs : les calculs sont plus rapides étant donné que les résultats précédent sont connus	

Conclusion

Nous sommes dans l'ensemble très satisfait de la manière dont s'est déroulé ce projet et du résultat auquel nous avons abouti. Nous avons essayé d'implémenter toutes les fonctionnalités demandées en commençant par une conception minimal. Ensuite, nous avons amélioré ce modèle en y ajoutant les fonctionnalités 1 par 1 au cours des TD.

En prenant du recul, nous nous sommes rendu compte qu'à chaque semaine notre projet devenait de plus en plus compliqué et moins extensible. Le fait d'ajouter une fonctionnalité par une était certe une assez bonne approche en termes de productivité cependant cela impliquait de changer la conception globale de notre projet à chaque changement quasiment.

Malheureusement tenter d'ouvrir ainsi au maximum les solutions a également eu pour effet de nous faire prendre un peu en retard. Cela explique en partie pourquoi nous n'avons pas pu finir de couvrir 100% des fonctionnalités des 2 TDs néanmoins nous sommes fier d'avoir réussi à faire un modèle qui nous semble bien conçu (à l'aide des différents patterns et choix d'implémentations) et assez ouvert à de futures implémentations de nouvelles fonctionnalités. Par exemple, nous avons tenu comptes des parties manquantes du projet dans la conception générale et la structure du projet afin de pouvoir les ajoutés sans trop d'efforts.

Auto-évaluation

GIUNTINI Romain	LEGRIFI Amine	PERES Richard	SABRI Anass
90	90	130	90

Concernant notre gestion de projet, tout le monde a pu apporter du contenu au projet. Que ce soit au niveau de l'aspect technique ou gestion de projet, nous nous sommes efforcés de rester le plus organisé afin de ne perdre que le minimum de temps nécessaire pour les différents pans du projet.
On a un peu eu de difficulté au début en ce qui concerne la répartition des tâches car nous n'avons pas passé beaucoup de temps au début de chaque séance pour faire répartition des tâches.

Avec du recul, nous pensons qu'il aurait mieux fallu passer plus de temps dès le début du projet pour s'occuper notamment de la création et répartition des tâches, en suivant une approche agile (avec Milestone et Kanban), entre les membres du groupe.