Rapport de conception logiciel
Cookie On Demand

Otake: David Bisegna, Jason Haenlin, Ruben Houri, Betsara Marcellin

(Libre de droit) https://p1.pxfuel.com/preview/501/231/776/cookies-smarties-cookie-home-made.jpg

Responsable : Philippe Collet

Intervenants : Philippe Collet, Johann Mortara, Anne-Marie Pinna

Sommaire

Contents

1. Fonctionnalités réalisées

Boutique
Je peux changer les horaires d'ouverture de ma boutique
Scénario cucumber : **ShopAdjustTheSchedule**

Boutique
Je ne peux pas commander en dehors des horaires d'ouverture
Scénario cucumber : **OrderOnCorrectTimePeriod**

Boutique
Je ne peux pas retirer ma commande avant l'heure spécifiée lors de la commande
Scénario cucumber : **OrderOnCorrectTimePeriod**

Boutique
Je peux mettre ma "recette du mois" à disposition
Scénario cucumber : **CheckCinemaTicket**
Fait dans le test checkTicketTest. La recette du mois et le cookie du jours sont identique.

Boutique
Je peux retirer ma commande
Scénario cucumber : **OrderOnCorrectTimePeriod**

Commande
Je sais que la commande a été payée
Scénario cucumber : **MakingABasicOrder**

Commande
Je peux commander sans compte sur la plateforme CoD
Scénario cucumber : **MakingABasicOrder**

Commande
Il n'est pas possible de commander quelque chose qu'on ne sait pas fabriquer
Scénario cucumber : Dans les tests Unitaire de Storage, mais gère mal les boissons

Commande
Je peux payer ma commande par carte bancaire
Scénario cucumber : Pas de scénarios

Commande
Je peux passer une commande avec des cookies classiques et personnalisés
Scénario cucumber : **DiscountOnClosingHour**

Commande
Je peux calculer un prix TTC à partir des ingrédients, des discounts et des taxes.
Scénario cucumber : **DiscountOnClosingHour**

Commande
Les cookies personnalisés respectent les contraintes de fabrication (mix, topping, etc.)
Scénario cucumber :

Commande
Je dois payer un surcoût en cas de commande de cookies personnalisés
Scénario cucumber : **DiscountOnClosingHour**

Commande
Je peux recevoir des packs de cookies selon le nombre de cookies commandés
Scénario cucumber :

Commande
Je peux associer des boissons à des packs de cookies
Scénario cucumber : **PackWithBeverageInOrder**

Discount
Je peux bénéficier des 10% tous les 30 cookies
Scénario cucumber :

Discount
Je peux bénéficier d'une réduction avec le bon EVENT à partir de 100 cookies commandés
Scénario cucumber : **DiscountEventInShop**

Discount
Je peux bénéficier d'une réduction avec le CE de mon entreprise
Scénario cucumber : **DiscoundForCompanies**

Discount
Je peux bénéficier d'une réduction grâce à mon ancienneté
Scénario cucumber : **DiscountWithSeniority**

Discount
Je peux bénéficier de 30% de réduction sur les recettes préexistantes dans la dernière heure d'ouverture
Scénario cucumber : **DiscountOnClosingHour**

Recette
La marque peut ajouter de nouveaux ingrédients
Scénario cucumber :

Recette
Je peux gérer mes marges sur les cookies (ingrédients, recettes personnalisés)
Scénario cucumber : **ManagingShopTaxes**

Statistiques
Je peux collecter des statistiques par boutique (p. ex. nombre de cookies vendu par jour, % de cookies personnalisés)

Scénario cucumber : **GettingStatisticOnAShop**

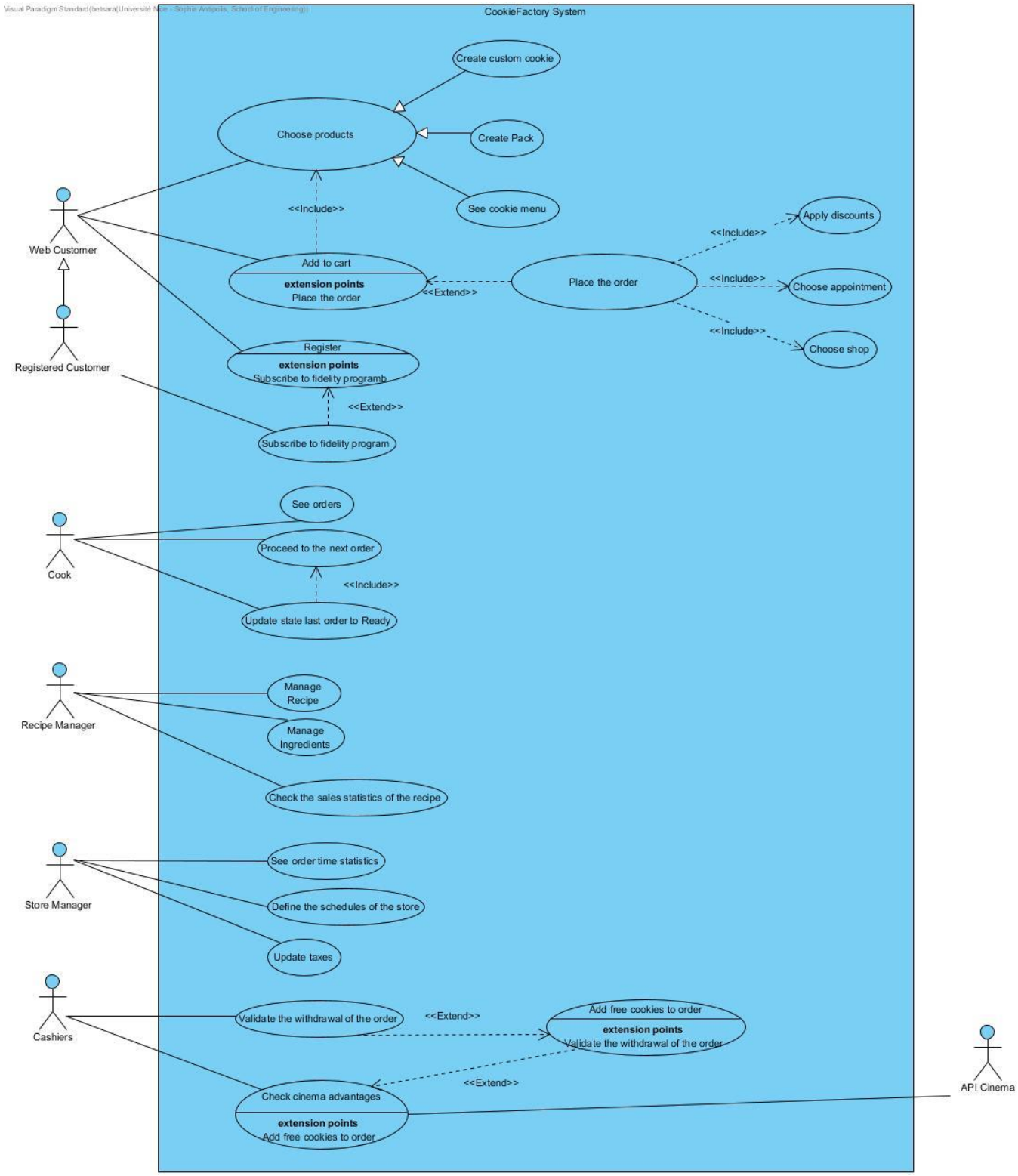
Statistiques
Je peux agréger des statistiques nationalement pour TCF
Scénario cucumber : **CookieUseofStatistic**

Cinéma
Je peux bénéficier de cookies gratuits avec mon ticket de cinéma du Paté-Poorsha
Scénario cucumber : **CheckCinemaTicket**

Cinéma
Je peux interagir avec le système du cinéma à partir d'un simulateur de son API
Scénario cucumber : **CheckCinemaTicket**

2. Diagrammes UML

2.1. Use case



Ce diagramme donne un aperçu des interactions des utilisateurs/acteurs avec notre système. Nous noterons les points ci-dessous :

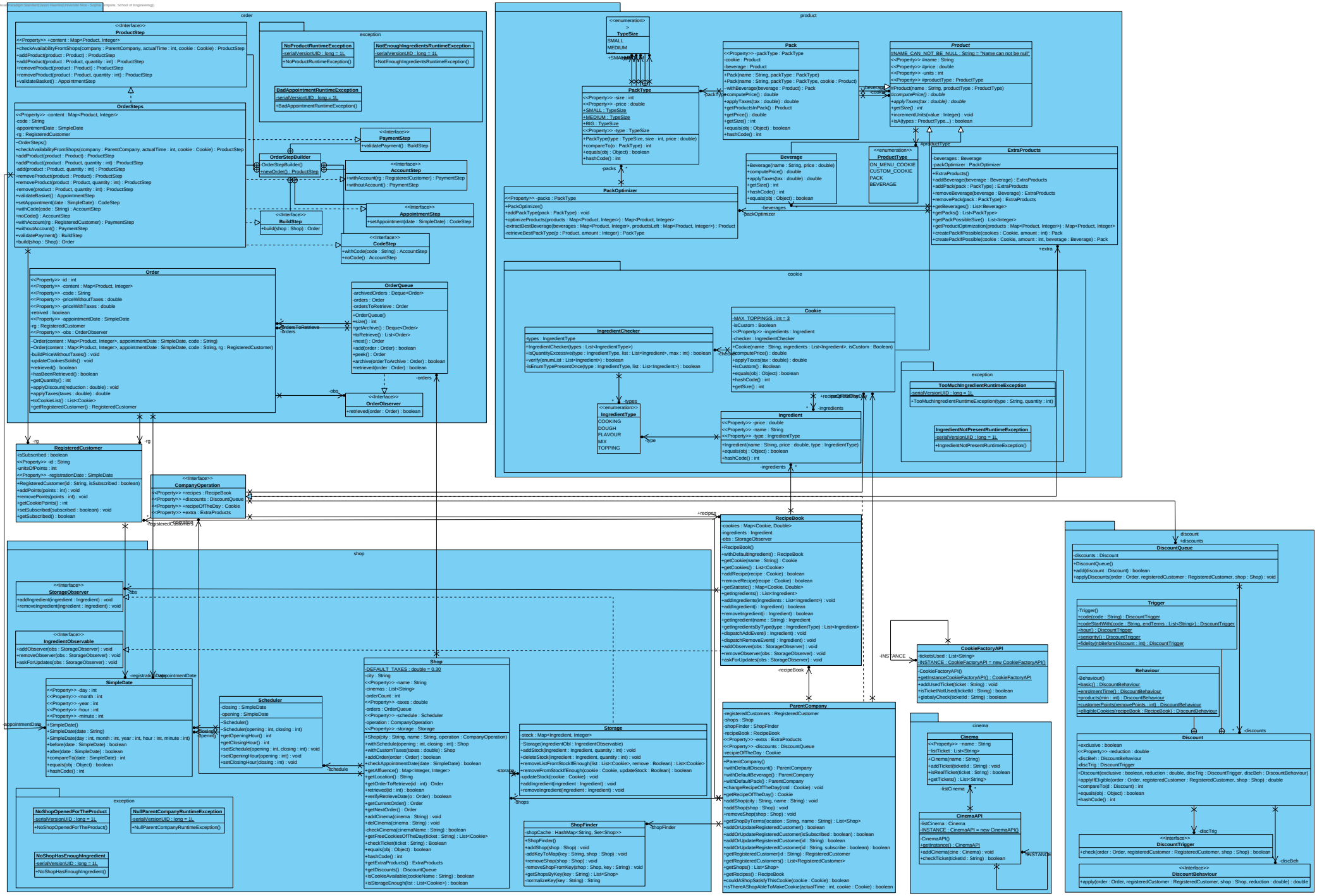
Les clients :

- Nous avons hiérarchisé les acteurs clients du système, en tenant compte du fait qu'un client enregistré peut effectuer les mêmes actions qu'un client non enregistré.
- Le choix des produits est représenté par un héritage des use cases car les use case "create custom cookie", "create pack" et "see cookie menu" héritent du comportement de "choose products".
- Un client peut ajouter un produit à son panier, cependant il doit au préalable avoir choisi ce dernier. Ensuite, une fois son panier remplis le client peut effectuer une commande dans laquelle on retrouve l'application des réductions, le choix de l'heure de récupération ainsi que le choix du magasin.
- Un client non enregistré peut créer un compte, à partir de ce moment ce dernier devient un client enregistré (il a donc accès à plus de discount qu'avant, ex : ancienneté). Il peut alors participer au programme de fidélité de cookie factory.

Les employés du magasin :

- Les préparateurs de commande sont aussi invités à utiliser notre système cookie factory. En effet quand une commande a été effectuée pour être récupéré dans le shop où il est, le préparateur de commande peut voir ces commandes, afin qu'il puisse les préparer, une fois prêt il peut le faire savoir ce qui nécessite qu'il consulte la prochaine commande à préparer.
- Un manager du magasin, lui, peut consulter les statistiques d'affluence horaires, définir les horaires du magasin et modifier les taxes.
- Le caissier va pouvoir, recevoir le ticket de commande et valider dans le système la validité du ticket et la récupération de la commande. Il peut toutefois faire valider un ticket de cinéma (appel à l'API externe), il peut ensuite rajouter des cookies du jour dans la commande initiale et valider le retrait de la commande. Nous avons un acteur externe à notre système, qui est actuellement "simulé" dans le projet (à la demande du client), mais ne devrait pas faire partie de notre système.

Le manager des recettes, il est le seul qui puisse consulter les statistiques nationales de vente des cookies. Il a la possibilité de gérer les recettes ainsi que les ingrédients


order
<<Interface>>
ProductStep
<<Property>> +content : Map<Product, Integer>
+checkAvailabilityFromShops(company : ParentCompany, actualTime : int, cookie : Cookie) : ProductStep
+addProduct(product : Product) : ProductStep
+addProduct(product : Product, quantity : int) : ProductStep
+removeProduct(product : Product) : ProductStep
+removeProduct(product : Product, quantity : int) : ProductStep
+validateBasket() : AppointmentStep
OrderSteps
<<Property>> -content : Map<Product, Integer>
-code : String
-appointmentDate : SimpleDate
-rg : RegisteredCustomer
-OrderSteps()
+checkAvailabilityFromShops(company : ParentCompany, actualTime : int, cookie : Cookie) : ProductStep
+addProduct(product : Product) : ProductStep
+addProduct(product : Product, quantity : int) : ProductStep
-add(product : Product, quantity : int) : ProductStep
+removeProduct(product : Product) : ProductStep
+removeProduct(product : Product, quantity : int) : ProductStep
-remove(product : Product, quantity : int) : ProductStep
+validateBasket() : AppointmentStep
+setAppointment(date : SimpleDate) : CodeStep
+withCode(code : String) : AccountStep
+noCode() : AccountStep
+withAccount(rg : RegisteredCustomer) : PaymentStep
+withoutAccount() : PaymentStep
+validatePayment() : BuildStep
+build(shop : Shop) : Order
exception
NoProductRuntimeException
-serialVersionUID : long = 1L
+NoProductRuntimeException()
NotEnoughIngredientsRuntimeException
-serialVersionUID : long = 1L
+NotEnoughIngredientsRuntimeException()
BadAppointmentRuntimeException
-serialVersionUID : long = 1L
+BadAppointmentRuntimeException()
<<Interface>>
PaymentStep
+validatePayment() : BuildStep
OrderStepBuilder
-OrderStepBuilder()
-newOrder() : ProductStep
<<Interface>>
AccountStep
+withAccount(rg : RegisteredCustomer) : PaymentStep
+withoutAccount() : PaymentStep
<<Interface>>
AppointmentStep
+setAppointment(date : SimpleDate) : CodeStep
<<Interface>>
BuildStep
+build(shop : Shop) : Order
<<Interface>>
CodeStep
+withCode(code : String) : AccountStep
+noCode() : AccountStep
Order
<<Property>> -id : int
<<Property>> -content : Map<Product, Integer>
<<Property>> -code : String
<<Property>> -priceWithoutTaxes : double
<<Property>> -priceWithTaxes : double
-retrived : boolean
<<Property>> -appointmentDate : SimpleDate
-rg : RegisteredCustomer
<<Property>> -obs : OrderObserver
-Order(content : Map<Product, Integer>, appointmentDate : SimpleDate, code : String)
-Order(content : Map<Product, Integer>, appointmentDate : SimpleDate, code : String, rg : RegisteredCustomer)
-buildPriceWithoutTaxes() : void
-updateCookiesSolds() : void
+retrieved() : boolean
+hasBeenRetrieved() : boolean
+getQuantity() : int
+applyDiscount(reduction : double) : void
+applyTaxes(taxes : double) : double
+toCookiesList() : List<Cookie>
+getRegisteredCustomer() : RegisteredCustomer
OrderQueue
-archivedOrders : Deque<Order>
-orders : Order
-ordersToRetrieve : Order
+OrderQueue()
+size() : int
+getArchive() : Deque<Order>
+toRetrieve() : List<Order>
+next() : Order
+add(order : Order) : boolean
+peek() : Order
+archive(orderToArchive : Order) : boolean
+retrieved(order : Order) : boolean
<<Interface>>
OrderObserver
+retrieved(order : Order) : boolean
RegisteredCustomer
-isSubscribed : boolean
<<Property>> -id : String
-unitsOfPoints : int
<<Property>> -registrationDate : SimpleDate
+RegisteredCustomer(id : String, isSubscribed : boolean)
+addPoints(points : int) : void
+removePoints(points : int) : void
+getCookiePoints() : int
+setSubscribed(subscribed : boolean) : void
+getSubscribed() : boolean
<<Interface>>
CompanyOperation
<<Property>> +recipes : RecipeBook
<<Property>> +discounts : DiscountQueue
<<Property>> +recipeOfTheDay : Cookie
<<Property>> +extra : ExtraProducts
product
<<enumeration>>
TypeSize
SMALL
MEDIUM
BIG
Pack
<<Property>> -packType : PackType
-cookie : Product
-beverage : Product
+Pack(name : String, packType : PackType)
+Pack(name : String, packType : PackType, cookie : Product)
+withBeverage(beverage : Product) : Pack
+computePrice() : double
+applyTaxes(tax : double) : double
+getProductsInPack() : Product
+getPrice() : double
+getSize() : int
+equals(obj : Object) : boolean
+hashCode() : int
Product
#NAME_CAN_NOT_BE_NULL : String = "Name can not be null"
<<Property>> #name : String
<<Property>> #price : double
<<Property>> -units : int
<<Property>> #productType : ProductType
#Product(name : String, productType : ProductType)
#computePrice() : double
#applyTaxes(tax : double) : double
+getSize() : int
+incrementUnits(value : Integer) : void
+isA(types : ProductType...) : boolean
PackType
<<Property>> -size : int
<<Property>> -price : double
+SMALL : TypeSize
+MEDIUM : TypeSize
+BIG : TypeSize
<<Property>> -type : TypeSize
+PackType(type : TypeSize, size : int, price : double)
+compareTo(o : PackType) : int
+equals(obj : Object) : boolean
+hashCode() : int
Beverage
+Beverage(name : String, price : double)
+computePrice() : double
+applyTaxes(tax : double) : double
+getSize() : int
+hashCode() : int
+equals(obj : Object) : boolean
<<enumeration>>
ProductType
ON_MENU_COOKIE
CUSTOM_COOKIE
PACK
BEVERAGE
ExtraProducts
-beverages : Beverage
-packOptimizer : PackOptimizer
+ExtraProducts()
+addBeverage(beverage : Beverage) : ExtraProducts
+addPack(pack : PackType) : ExtraProducts
+removeBeverage(beverage : Beverage) : ExtraProducts
+removePack(pack : PackType) : ExtraProducts
+getBeverages() : List<Beverage>
+getPacks() : List<PackType>
+getPackPossibleSize() : List<Integer>
+getProductOptimization(products : Map<Product, Integer>) : Map<Product, Integer>
+createPackIfPossible(cookies : Cookie, amount : int) : Pack
+createPackIfPossible(cookie : Cookie, amount : int, beverage : Beverage) : Pack
PackOptimizer
<<Property>> -packs : PackType
+PackOptimizer()
+addPackType(pack : PackType) : void
+optimizeProducts(products : Map<Product, Integer>) : Map<Product, Integer>
-extractBestBeverage(beverages : Map<Product, Integer>, productsLeft : Map<Product, Integer>) : Product
-retrieveBestPackType(p : Product, amount : Integer) : PackType
cookie
Cookie
-MAX_TOPPINGS : int = 3
-isCustom : Boolean
<<Property>> -ingredients : Ingredient
-checker : IngredientChecker
+Cookie(name : String, ingredients : List<Ingredient>, isCustom : Boolean)
+computePrice() : double
+applyTaxes(tax : double) : double
+isCustom() : Boolean
+equals(obj : Object) : boolean
+hashCode() : int
+getSize() : int
IngredientChecker
-types : IngredientType
+IngredientChecker(types : List<IngredientType>)
+isQuantityExcessive(type : IngredientType, list : List<Ingredient>, max : int) : boolean
+verify(enumList : List<Ingredient>) : boolean
-isEnumTypePresentOnce(type : IngredientType, list : List<Ingredient>) : boolean
<<enumeration>>
IngredientType
COOKING
DOUGH
FLAVOUR
MIX
TOPPING
Ingredient
<<Property>> -price : double
<<Property>> -name : String
<<Property>> -type : IngredientType
+Ingredient(name : String, price : double, type : IngredientType)
+equals(obj : Object) : boolean
+hashCode() : int
exception
TooMuchIngredientRuntimeException
-serialVersionUID : long = 1L
+TooMuchIngredientRuntimeException(type : String, quantity : int)
IngredientNotPresentRuntimeException
-serialVersionUID : long = 1L
+IngredientNotPresentRuntimeException()
shop
<<Interface>>
StorageObserver
+addIngredient(ingredient : Ingredient) : void
+removeIngredient(ingredient : Ingredient) : void
<<Interface>>
IngredientObservable
+addObserver(obs : StorageObserver) : void
+removeObserver(obs : StorageObserver) : void
+askForUpdates(obs : StorageObserver) : void
SimpleDate
<<Property>> -day : int
<<Property>> -month : int
<<Property>> -year : int
<<Property>> -hour : int
<<Property>> -minute : int
+SimpleDate()
+SimpleDate(date : String)
+SimpleDate(day : int, month : int, year : int, hour : int, minute : int)
+before(date : SimpleDate) : boolean
+after(date : SimpleDate) : boolean
+compareTo(date : SimpleDate) : int
+equals(obj : Object) : boolean
+hashCode() : int
Scheduler
-closing : SimpleDate
-opening : SimpleDate
-Scheduler()
-Scheduler(opening : int, closing : int)
+getOpeningHour() : int
+getClosingHour() : int
+setSchedule(opening : int, closing : int) : void
+setOpeningHour(opening : int) : void
+setClosingHour(closing : int) : void
Shop
-DEFAULT_TAXES : double = 0.30
-city : String
<<Property>> -name : String
-cinemas : List<String>
-orderCount : int
<<Property>> -taxes : double
-orders : OrderQueue
<<Property>> -schedule : Scheduler
-operation : CompanyOperation
<<Property>> -storage : Storage
+Shop(city : String, name : String, operation : CompanyOperation)
+withSchedule(opening : int, closing : int) : Shop
+withCustomTaxes(taxes : double) : Shop
+addOrder(order : Order) : boolean
+checkAppointmentDate(date : SimpleDate) : boolean
+getAffluence() : Map<Integer, Integer>
+getLocation() : String
+getOrderToRetrieve(id : int) : Order
+retrieved(id : int) : boolean
+verifyRetrieveDate(o : Order) : boolean
+getCurrentOrder() : Order
+getNextOrder() : Order
+addCinema(cinema : String) : void
+delCinema(cinema : String) : void
-checkCinema(cinemaName : String) : boolean
+getFreeCookiesOfTheDay(ticket : String) : List<Cookie>
+checkTicket(ticket : String) : Boolean
+equals(obj : Object) : boolean
+hashCode() : int
+getExtraProducts() : ExtraProducts
+getDiscounts() : DiscountQueue
+isCookieAvailable(cookieName : String) : boolean
+isStorageEnough(list : List<Cookie>) : boolean
Storage
-stock : Map<Ingredient, Integer>
-Storage(ingredientObl : IngredientObservable)
+addStock(ingredient : Ingredient, quantity : int) : void
+deleteStock(ingredient : Ingredient, quantity : int) : void
+removeLesFromStockIfEnough(list : List<Cookie>, remove : Boolean) : List<Cookie>
+removeFromStockIfEnough(cookie : Cookie, updateStock : Boolean) : boolean
-updateStock(cookie : Cookie) : void
+addIngredient(ingredient : Ingredient) : void
+removeIngredient(ingredient : Ingredient) : void
ShopFinder
-shopCache : HashMap<String, Set<Shop>>
+ShopFinder()
+addShop(shop : Shop) : void
-addKeyToMap(key : String, shop : Shop) : void
+removeShop(shop : Shop) : void
-removeShopFromKey(shop : Shop, key : String) : void
+getShopsByKey(key : String) : List<Shop>
-normalizeKey(key : String) : String
exception
NoShopOpenedForTheProduct
-serialVersionUID : long = 1L
+NoShopOpenedForTheProduct()
NullParentCompanyRuntimeException
-serialVersionUID : long = 1L
+NullParentCompanyRuntimeException()
NoShopHasEnoughIngredient
-serialVersionUID : long = 1L
+NoShopHasEnoughIngredient()
RecipeBook
-cookies : Map<Cookie, Double>
-ingredients : Ingredient
-obs : StorageObserver
+RecipeBook()
+withDefaultIngredient() : RecipeBook
+getCookie(name : String) : Cookie
+getCookies() : List<Cookie>
+addRecipe(recipe : Cookie) : boolean
+removeRecipe(recipe : Cookie) : boolean
+getStatistic() : Map<Cookie, Double>
+getIngredients() : List<Ingredient>
+addIngredients(ingredients : List<Ingredient>) : void
+addIngredient() : Ingredient) : boolean
+removeIngredient() : Ingredient) : boolean
+getIngredient(name : String) : Ingredient
+getIngredientsByType(type : IngredientType) : List<Ingredient>
+dispatchAddEvent() : Ingredient) : void
+dispatchRemoveEvent() : Ingredient) : void
+addObserver(obs : StorageObserver) : void
+removeObserver(obs : StorageObserver) : void
+askForUpdates(obs : StorageObserver) : void
ParentCompany
-registeredCustomers : RegisteredCustomer
-shops : Shop
-shopFinder : ShopFinder
-recipeBook : RecipeBook
<<Property>> -extra : ExtraProducts
<<Property>> -discounts : DiscountQueue
-recipieOfTheDay : Cookie
+ParentCompany()
+withDefaultDiscount() : ParentCompany
+withDefaultBeverage() : ParentCompany
+withDefaultPack() : ParentCompany
+changeRecipeOfTheDay(rotd : Cookie) : void
+getRecipeOfTheDay() : Cookie
+addShop(city : String, name : String) : void
+addShop(shop : Shop) : void
+removeShop(shop : Shop) : void
+getShopByTerms(location : String, name : String) : List<Shop>
+addOrUpdateRegisteredCustomer() : boolean
+addOrUpdateRegisteredCustomer(isSubscribed : boolean) : boolean
+addOrUpdateRegisteredCustomer(id : String) : boolean
+addOrUpdateRegisteredCustomer(id : String, subscribe : boolean) : boolean
+getRegisteredCustomer(id : String) : RegisteredCustomer
+getRegisteredCustomers() : List<RegisteredCustomer>
+getShops() : List<Shop>
+getRecipes() : RecipeBook
+couldAShopSatisfyThisCookie(cookie : Cookie) : boolean
+isThereAShopAbleToMakeCookie(actualTime : int, cookie : Cookie) : boolean
CookieFactoryAPI
-ticketsUsed : List<String>
-INSTANCE : CookieFactoryAPI = new CookieFactoryAPI()
-CookieFactoryAPI()
+getInstanceCookieFactoryAPI() : CookieFactoryAPI
+addUsedTicket(ticket : String) : void
+isTicketNotUsed(ticketId : String) : boolean
+globalyCheck(ticketId : String) : boolean
cinema
Cinema
<<Property>> -name : String
-listTicket : List<String>
+Cinema(name : String)
+addTicket(ticketId : String) : void
+isRealTicket(ticket : String) : boolean
+getTickets() : List<String>
CinemaAPI
-listCinema : Cinema
-INSTANCE : CinemaAPI = new CinemaAPI()
-CinemaAPI()
+getInstance() : CinemaAPI
+addCinema(cine : Cinema) : void
+checkTicket(ticketId : String) : boolean
DiscountQueue
-discounts : Discount
+DiscountQueue()
+add(discount : Discount) : boolean
+applyDiscounts(order : Order, registeredCustomer : RegisteredCustomer, shop : Shop) : void
Trigger
-Trigger()
+code(code : String) : DiscountTrigger
+codeStartWith(code : String, endTerms : List<String>) : DiscountTrigger
+hour() : DiscountTrigger
+seniority() : DiscountTrigger
+fidelity(nbBeforeDiscount : int) : DiscountTrigger
Behaviour
-Behaviour()
+basic() : DiscountBehaviour
+enrolmentTime() : DiscountBehaviour
+products(min : int) : DiscountBehaviour
+customerPoints(removePoints : int) : DiscountBehaviour
+eligibleCookies(recipeBook : RecipeBook) : DiscountBehaviour
Discount
+exclusive : boolean
<<Property>> -reduction : double
-discBeh : DiscountBehaviour
-discTrig : DiscountTrigger
+Discount(exclusive : boolean, reduction : double, discTrig : DiscountTrigger, discBeh : DiscountBehaviour)
+applyIfEligible(order : Order, registeredCustomer : RegisteredCustomer, shop : Shop) : double
+compareTo(d : Discount) : int
+equals(obj : Object) : boolean
+hashCode() : int
<<Interface>>
DiscountTrigger
+check(order : Order, registeredCustomer : RegisteredCustomer, shop : Shop) : boolean
<<Interface>>
DiscountBehaviour
+apply(order : Order, registeredCustomer : RegisteredCustomer, shop : Shop, reduction : double) : double

3 Patrons de conceptions

1.1. Les commandes

#State, Builder, Stepbuilder, Command

Analyse

Une commande consiste en une série d'étape avec les états de son avancement. Cet objet est particulièrement sensible car a besoin de beaucoup de vérification. Une commande contient, la liste des produits, le magasin où aller retirer la commande, mais aussi la date de récupération, le paiement, etc.

Problèmes

Il s'agit de concevoir au mieux l'architecture pour pallier aux mieux aux problèmes. Au début, nous avions fait un objet basique sans patron de conception, bien entendu, on s'est rendu compte qu'il était très compliqué de gérer une commande avec autant de vérification et de méthodes à appeler. Pour éviter de se perdre avec autant d'étape dans la construction, nous devions trouver le meilleur patron de conception à appliquer.

Options

Premièrement, trois pattern design nous ont inspirés, le premier et le command pattern (peut-être à cause du nom) qui aurait pu être pratique pour mettre et enlever des produits, néanmoins, il ne nous semblait pas forcément nécessaire d'avoir cette fonctionnalité, car on peut facilement retrouver quel produit on veut enlever.

Lorsque nous nous dit, construire un objet étape par étape, le Builder pattern design semblé sortir du lot, mais un point nous déranger, c'est qu'il est principalement (par défaut,) utilisé pour construire un objet dynamiquement lorsque l'on a des éléments optionnels. Dans notre cas, tout est important, c'était surtout pour avoir un assemblage plus propre de l'objet. Finalement, nous avons trouvé une alternative au Builder, le StepBuilder qui lui, agit comme un Builder mais avec une notion de Step très intéressante dans notre cas.
Et en dernier, il fallait aussi pouvoir gérer les états de la commande (En attente, Validée, Retiré, etc.). D'où la proposition d'un enseignant encadrant qui nous a suggéré que cela pourrait résoudre notre problème.

Solutions

Pour la partie de l'Order, nous avons finalement utilisé un StepBuilder avec le diagramme de classe illustré ci-dessous. Cela nous permet de voir clairement les méthodes à appeler à chaque étape du processus de création et la méthode build (en fin de cycle) d'appliquer toutes les vérifications, de ce fait, un Order n'est jamais créé s'il ne respecte pas certaines contraintes.

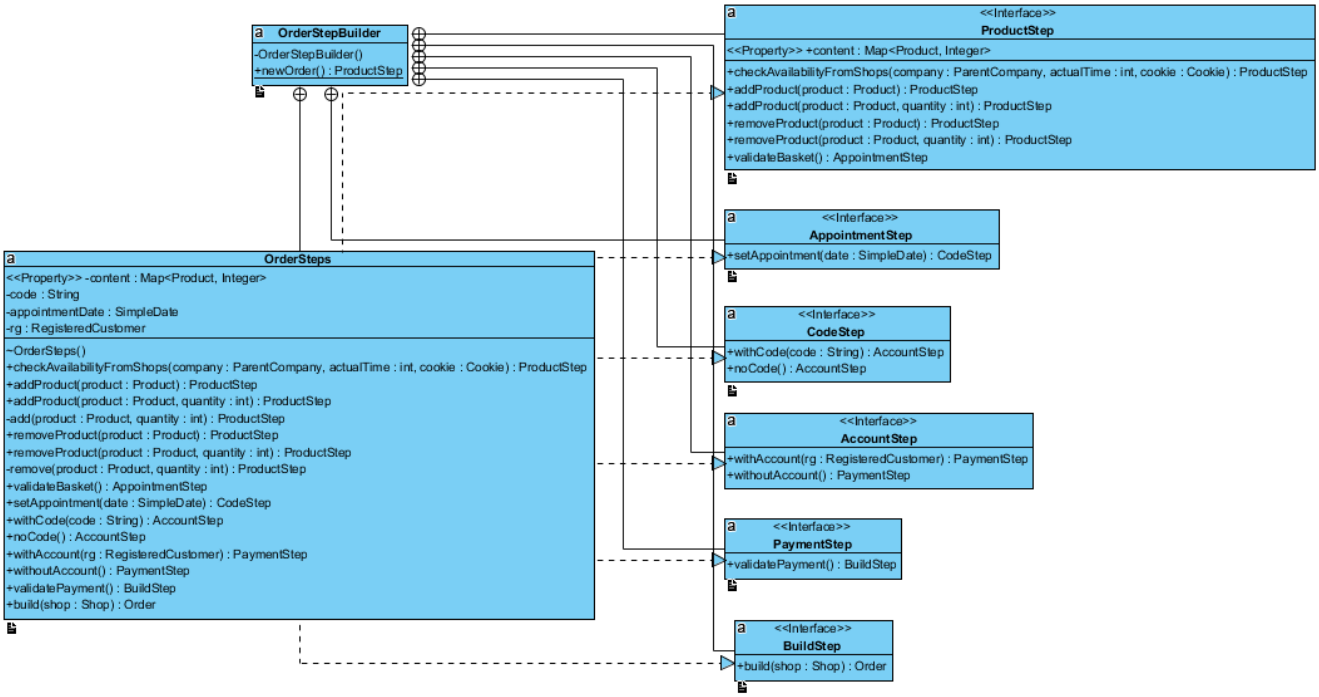


Figure : Diagramme de classe pour la partie de la construction de l'Order

Concernant l'état d'une commande, il sera géré autrement que par les états comme indiqués plus haut, car nous avons optés pour une autre implémentation.

1.2. La liste des orders

#Observer, Callback

Analyse

Pour gérer la liste des objets Order, nous aurions pu faire une liste et utiliser le State design pattern comme énoncé dans la partie 'a', néanmoins, cela nous semblait gourmand en ressources sur le long terme, de ce fait, une classe OrderQueue a été faite pour gérer tous les états d'une commande, état enregistré, prêt à être retiré et archivé.

Solutions

Pour gérer les commandes, la classe OrderQueue contient 2 Queue pour la liste des commandes en attente et la liste des commandes archivés (donc retirés). Celle des commandes en attente est optimisé avec l'utilisation d'une PriorityQueue pour trier les commandes par date de récupération, de ce fait, quand un cuisinier veut faire la commande suivante, il recevra la commande qui a la date de retrait la plus courte. Lorsque le cuisinier fait next() il reçoit automatiquement la prochaine commande et place l'autre en tant qui prête à être récupéré. Une commande ne peut être retiré que si le magasin approuve de la bonne date de récupération (si le client arrive avant, il ne peut pas) et la classe OrderQueue s'assure que la commande est bien prête à être dégusté. À ce moment, il faut pouvoir dire à l'OrderQueue que la commande en question a bien été récupéré. Pour réussir cette opération, nous avons utilisé la fonctionnalité de Callback du pattern design Observer, pour pouvoir à partir d'un Order, se placer en tant que 'récupéré' et l'archiver dans la Queue.

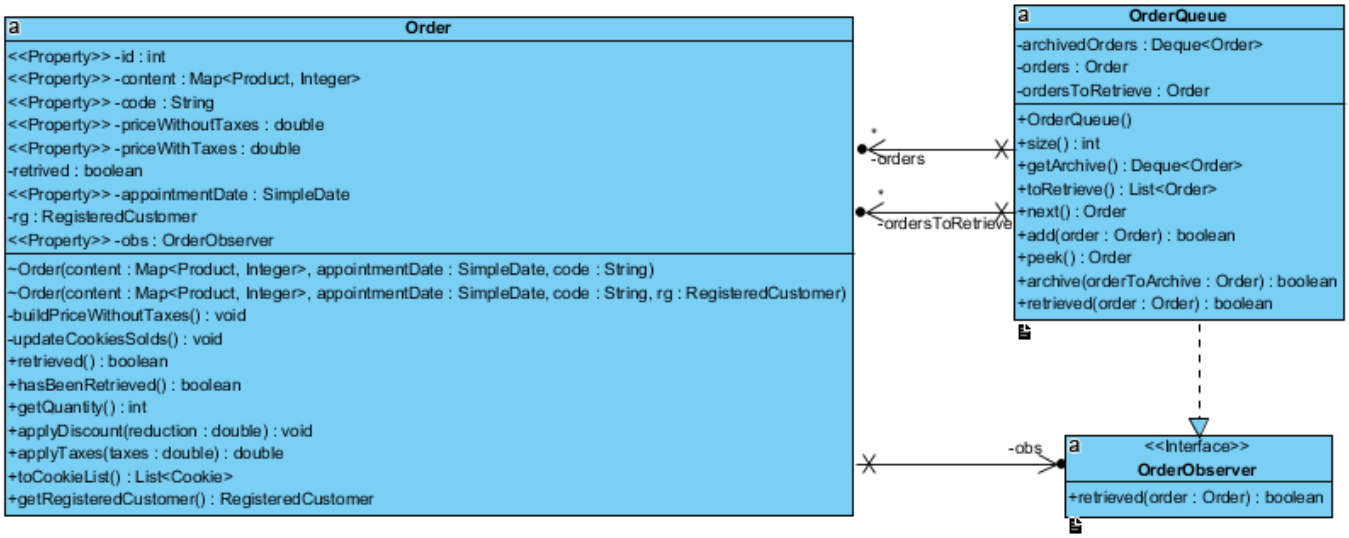


Figure : Diagramme de classe de la partie de la gestion de l'état d'un Order

1.3. Les discounts

#Strategy, Template, Chain of Responsability

Analyse

Les Discounts représentent une grosse partie du projet et apportent avec elles, son lot de complexité. Nous avons différent type de réduction s'appliquant sur plusieurs critères possibles, l'heure de fermeture d'une boutique, les points du programme de fidélité, les codes promos et bien d'autres encore.

Problèmes

Il faut pouvoir appliquer des réductions de la même manière mais qui pour la plupart, possèdent des comportements relativement différents les uns des autres. De plus en se basant sur le cahier des charges, on peut avoir les mêmes éléments d'activations sans pour autant avoir le même comportement sur la suite du processus.
Un code "EVENT" va vérifier le nombre de produit dans le panier alors qu'un code "CE_<entreprise>" ne va rien vérifier dans le système, du moment que le code est bon (donc que l'entreprise est partenaire) alors la réduction s'applique. Qui plus est, il faut voir plus loin et avoir un code qui offre une bonne flexibilité en évitant la duplication et la modification. Un dernier point, le fait que plusieurs réductions puissent s'appliquer si aucune réduction exclusive n'est déjà validée. Il faut donc cette notion de priorité dans les responsabilités.

Options

Première idée, l'utilisation du pattern design Strategy pour pouvoir appliquer la même structure à tous les Discount tout en pouvant changer le comportement (code, quantité, fidélité, etc.).
En deuxième lieu, comme énoncé, nous avons parfois les mêmes "triggers" sans pour autant avoir les mêmes "Behaviour" par la suite. Un State design pattern peut permettre de modéliser ce que l'on cherche.

Pour gérer les responsabilités et la priorité, il nous a été proposé de faire un patron Chain of Responsability. Néanmoins, en regardant de plus près, nous n'avions pas trouvé ce pattern mieux approprié

qu'une simple PriorityQueue. Ce design pattern pouvait nous permettre d'activer des discounts selon s'ils remplissent la condition (exclusive ou pas) et si c'est validé, alors on arrête de demander aux autres discounts s'ils sont en conditions de remplir la demande. Néanmoins, dans notre cas, nous préférions ranger directement les discounts par exclusivité, de ce fait, si une réduction exclusive passe, alors on cesse d'aller demander aux autres.

Solutions

D'après nos observations, nous avons opté pour l'application d'un Strategy design pattern lié avec un Template design pattern. Ainsi que l'application d'une PriorityQueue pour la gestion des priorités. Chaque Discount est composé d'un Trigger et d'un Behaviour s'appliquant quand le trigger passe.
Deux classes internes statiques, Behaviour et Trigger sont justes là pour avoir des éléments par défauts facilitant la construction des réductions en réduisant la duplication. Mais cela n'empêche pas la création d'un nouveau discount de manière totalement dynamique.

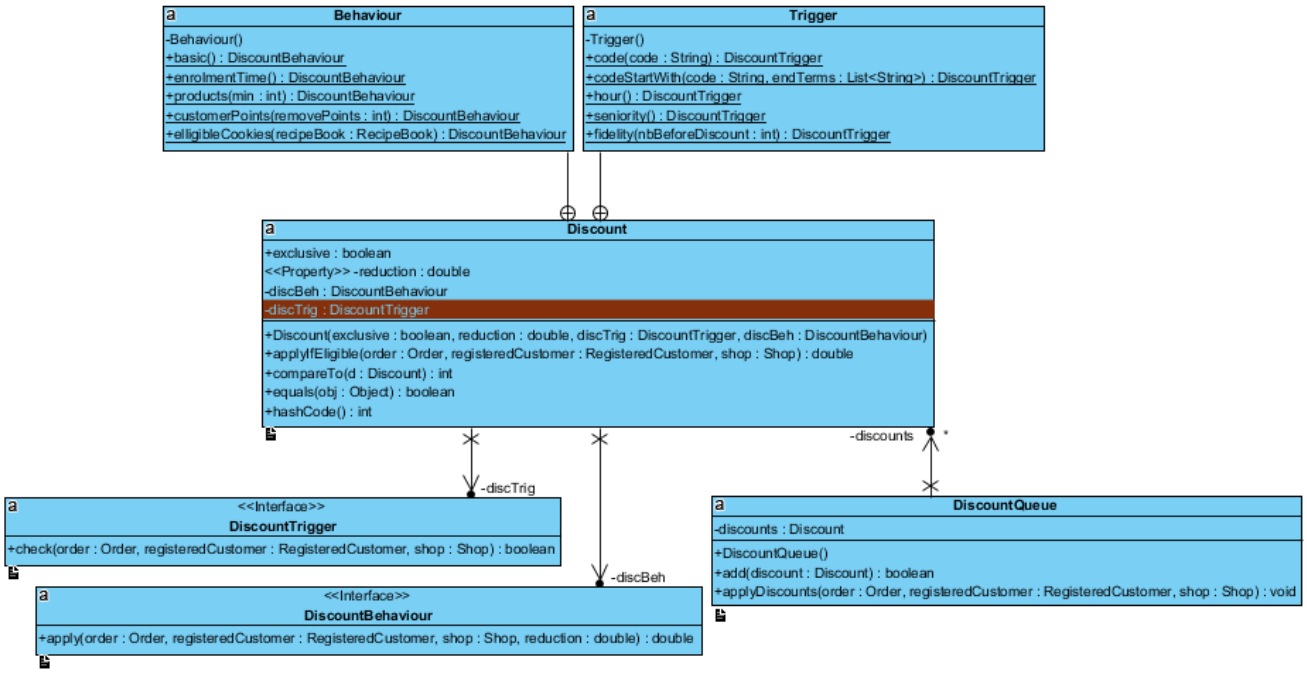


Figure : Diagramme de classe de la partie Discount

1.4. Le stockage des ingrédients

#Observer, callback

Analyse

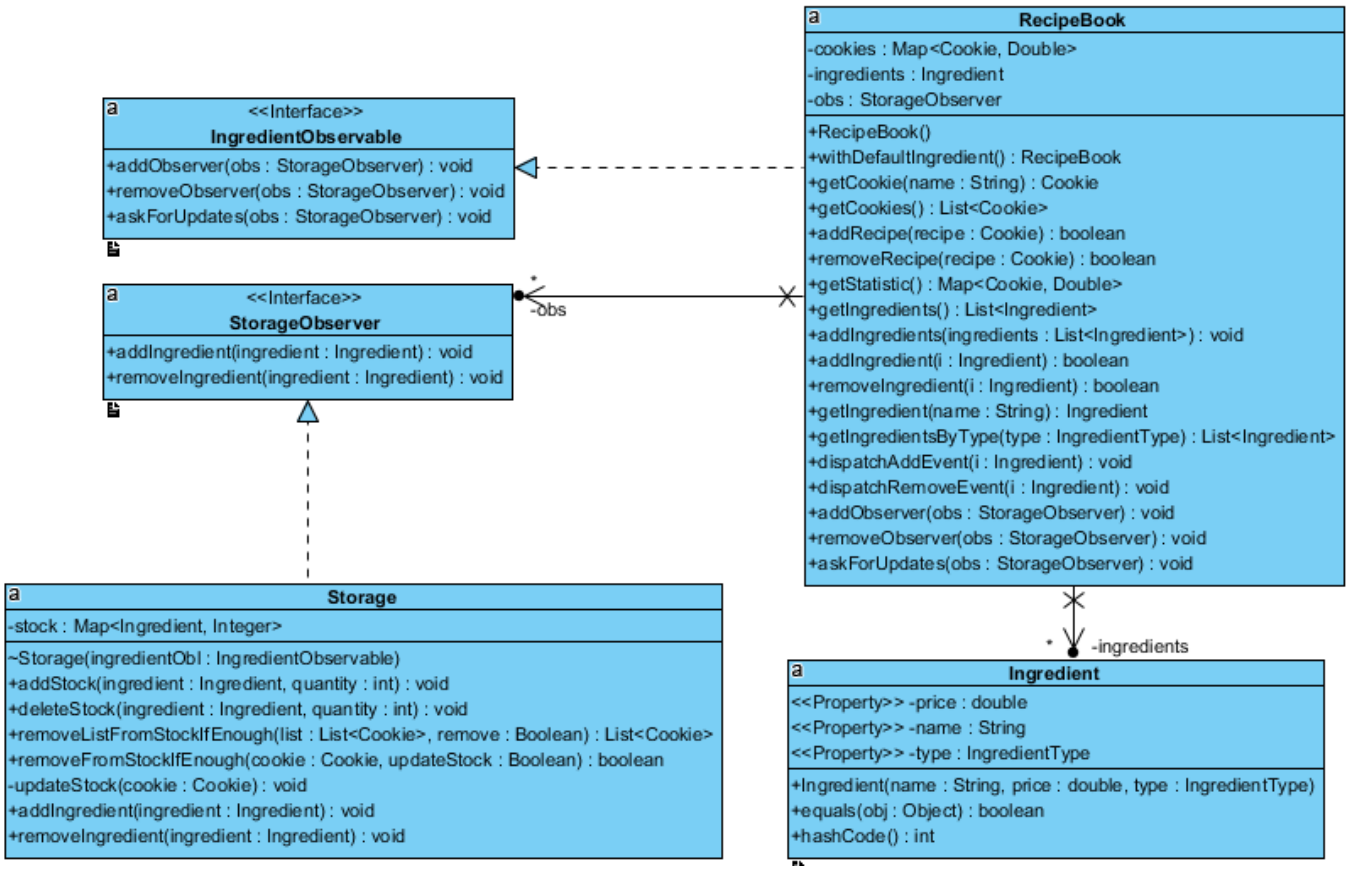
Stocker les ingrédients implique d'avoir un stockage individuel par magasin tout en sachant que la liste des ingrédients est gérée par l'objet RecipeBook, instancié dans ParentCompany car c'est elle qui a la responsabilité des ingrédients de toutes les boutiques.

Problèmes

Une liste d'ingrédient par Magasin, sachant que si un ingrédient est ajouté, il faut mettre à jour les informations de tous les stocks de chaque Shop.

Solutions

La meilleure solution est celle d'utiliser un pattern Observer pour utiliser sa fonctionnalité de callback en broadcast. Lorsque qu'un produit est créé, il faut passer un Observable au Storage pour lui demander de s'abonner au Stream du RecipeBook, de ce fait, quand un ingrédient et ajouté ou supprimé, cela va pour chaque Observer, faire une mise à jour de son stock.



1.5. Les classes servants d'API

#singleton

Analyse

Il nous a été demandé de simuler la communication avec les cinémas Paté-Poorsha. Il nous fallait pouvoir à partir des informations d'un ticket, savoir si le ticket était valide pour un partenaire donné.
Validée le cinéma partenaire est vérifié individuellement par chaque magasin, l'API centrale des cinémas, elle, vérifie pour un cinéma, si le ticket est valide. Pour finir, il faut pouvoir vérifier si le ticket n'a pas déjà utilisé quel que soit le magasin pour éviter de donner trop de cookie.

Choix adopté

Comme une API est par principe, une instance unique d'un programme, le pattern design le plus approprié était le Singleton. Une seule API existe pour les cinémas et il est impossible d'en avoir une autre instance. Pour les mêmes raisons, nous avons fait une API, toujours singleton pour la cookieFactory pour

stocker les tickets de cinéma utilisé et ne pas pouvoir les utiliser dans un autre magasin (si on imagine deux magasins proches d'un cinéma par exemple (, on voit bien certaine franchise à tous les coins de rue parfois dans certains pays).

1.6. Les cookies

#Strategy, Enums Polymorphiques, Telescoping Constructor Anti-Pattern

Problème

Faire un cookie repose sur beaucoup de types d'ingrédients et leurs constructions peuvent être fastidieuses. Avant d'avoir les ingrédients dynamiques, nous avions des Enum pour chaque type, nous construisons chaque cookie avec les ingrédients un par un. Nous avions appliqué un Strategy design pattern lié à une Énumération Polymorphiques (avec méthode abstraite) pour pouvoir construire rapidement un Cookie juste avec son nom en appelant "Recipe.CHOCOLALALA.create()" par exemple.

Fix

Après s'être rendu compte de ce problème, nous avons fait les ingrédients dynamiques et uniquement les types d'ingrédients en statiques, Finalement, nous passons uniquement une liste d'ingrédients et non plus un type d'ingrédient un par un, nous avions tout aussi penser à un Builder ou autre, mais sans succès.

1.7. Les produits

#Composite ou Decorator, Flyweight

Analyse

Un produit peut-être à la fois une boisson, un cookie voire même un pack composé de produits. Et ces objets peuvent être créés à de multiple reprise du fait des commandes qui se suivent.

Options

Vers la fin du projet, nous avons pensé à représenter la combinaison des produits par un Composite voire un Decorator du fait de leurs ressemblances. L'un respectivement permet de wrapper un objet ou groupe d'objet dans un autre objet de la même famille alors que l'autre va décorer un autre objet pour y ajouter un nouveau comportement ou une nouvelle fonctionnalité de manière dynamique.

"Sometimes the information displayed in a shopping cart is the product of a single item while other times it is an aggregation of multiple items. By implementing items as composites we can treat the aggregates and the items in the same way, allowing us to simply iterate over the tree and invoke functionality on each item. By calling the getCost() method on any given node we would get the cost of that item plus the cost of all child items, allowing items to be uniformly treated whether they were single items or groups of items."

source: Design pattern cheat sheet - Jason MCDONALD

Cette petite définition nous a influencé dans notre idée d'appliquer ce design pattern pour la gestion des packs et des boissons, dû au manque de temps, nous n'avons pas cherché plus loin et avons une architecture basique. Nous aurions aussi pu faire un Flyweight design pattern pour éviter de dupliquer le même objet à

plusieurs endroits. Si on utilise les cookies que nous récupérons du RecipeBook, alors tout va bien car c'est la même référence et donc le même espace en mémoire. Néanmoins, si on crée un cookie ou autre "à la volé", le nombre d'objet instancié en mémoire va rapidement augmenter et encombrer la mémoire, sachant que les commandes sont archivées et non supprimées, le Garbage collector ne passeras jamais et la mémoire va saturer.

1.8. Unification de l'interface Utilisateur

#Facade

Analyse

Dans une ultime analyse de notre code, nous aurions pu mettre des Façade pour pouvoir encadrer les différents utilisateurs et ce qu'ils ont le droit de faire, chaque Façade représentent une entrée du système pour différent type d'utilisateur.

2. Rétrospectives

2.1. Évolution de l'application

Des outils tel que sonar et Jenkins ont été utilisé pour surveiller la santé du projet à mesure que l'on avance. Notre CI nous permet de valider les push sur le dépôt et nous facilite le suivi de celui-ci de manière automatique. Sonarqube permet en autre d'avoir un aperçu de la qualité du code, bien évidemment, il ne faut pas forcément le suivre à la lettre, cependant, cela nous permet de repérer plus facilement des erreurs de code, d'optimisation simple ou autres auxquelles nous n'aurions pas pensé ou pas vu par inattention. Par ailleurs, GitHub nous interdit de merge une branche dans la branche master (si configuré ainsi) et permet de s'assurer que le code est toujours fonctionnel dans la branche principale. Des tests par mutations sont aussi exécutés pour vérifier la pertinence des tests (65% de tués). En termes de couverture, l'application a globalement bien été testé avec 83% de couverture. Notre application a évolué en essayant d'appliquer un aspect incrémental tout en veillant à ajouter de nouvelle fonctionnalité à chaque release. Par exemple, pour les réductions, nous avons commencé par implémenter les réductions les plus faciles à mettre en place et par incrémentation, on ajoute au fur à mesure des discounts plus compliqué à mettre en place. Néanmoins, nous avons rencontré des problèmes car, il est vrai que nous pouvons sortir avec une fonctionnalité plus rapidement, mais parfois, le fait de commencer par la plus simple, ne nous permet pas de nous représenter tout le périmètre de la Feature dans son ensemble. De ce fait, il nous a fallu dans certain cas, revoir notre implémentation car la version simple ne correspondait pas à l'architecture de la version plus compliqué alors que le contraire est valide. Dans l'ensemble, l'application a bien évoluée mais certain refactor nous ont coûté beaucoup de temps car nous avions mal pensé la première version de l'architecture de façon poussé. De ce fait, quand un upgrade arrive, nous ne sommes plus en mesure de les ajouter sans impliquer des modifications dans le code.

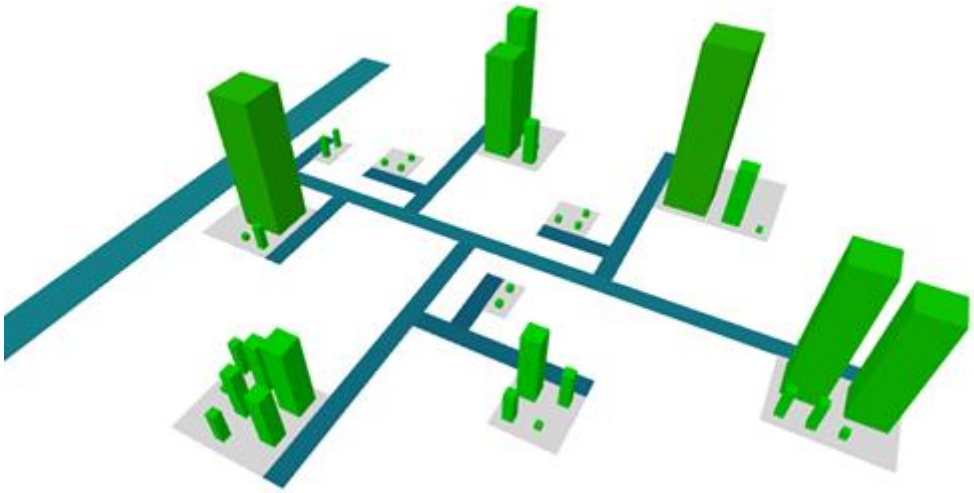


Figure : représentation code-city de l'état du projet

Ci-dessus, une vue de "code city" sur sonar. Nous n'avons pas de "god class", les classes sont plutôt équitablement réparti et la complexité de chaque classe est faible (vert : faible et rouge : forte). Nous pouvons voir un grand nombre de ligne de code pour les classes OrderStepBuilder, Order, Shop, ParentCompany ainsi que RecipeBook et Discount.

2.2. Amélioration possible

Concernant les possibles améliorations de notre application, il serait intéressant voire nécessaire d'avoir un emploi du temps non pas avec seulement l'heure d'ouverture et de fermeture, mais étalé sur au moins une semaine pour avoir une meilleure flexibilité comme les jours de fermeture ou s'il y a des horaires variables selon les jours, etc. Après une réflexion d'équipe, il s'est avéré que le I (Indescriptive Naming) de STUPID était parfois un peu présent dans les parties tests et rendaient le code moins lisible pour les autres membres de l'équipe. Nous avons aussi quelques doutes sur le T (Tight Coupling) dans certain cas, même si les tests restent relativement simples à implémenter, le fait de devoir construire trop d'objet peut être preuve d'une mauvaise conception.

Pour les statistiques, nous aurions pu utiliser des objets de type date pour obtenir des statistiques plus poussées sur les commandes de cookie, au lieu d'utiliser des entiers. En effet, cela permettrait d'avoir une trace définie dans le temps, des statistiques.

2.3. Conception à revoir

Il aurait été plus intéressant d'avoir un plus grand nombre de statistiques différentes, ainsi qu'une généralisation de celles-ci dans une classe servant d'API à toutes les différentes statistiques, plutôt que de demander à chaque classe différente de faire des statistiques sur elle-même. Cela est aussi problématique car une classe est amené à avoir une responsabilité qu'elle ne devrait pas avoir, à savoir, faire des stats sur elle-même (ne respecte pas le single responsibility principle).

2.4. Découpage des tâches

Nous avons initialement défini nos tâches pour la partie code du projet sur la période du 5 Novembre au 11 Novembre pour le TD4 et jusqu'au 21 Décembre pour inclure le TD5, ce qui nous donnait une première vision à long terme du projet. Nous avions alors 6 releases, avec quelques retards sur les releases impliquant un grand refactor et la montée en complexité du projet. Nous avons gardé une certaine maîtrise du projet en

avançant par “découpage vertical” et commencé par les parties qui nous semblaient les plus compliqué (qui aurait donc une forte influence sur le projet en cas de changement, ex : les discounts avant les packs). Actuellement le projet comporte une partie sur les statistiques qui répond aux demandes du TD 4 et 5 mais est à un stade initial de développement. Un retard que nous avons mal anticipé par rapport aux dernières semaines d’avant la pause pédagogique, qui fut assez chargé.

3. Auto-évaluations

Notre choix de répartition des 400 points pour cette auto-évaluation est le suivant :

- Betsara Marcellin 100 points
- Ruben Houri 100 points
- David Bisegna 100 points
- Jason Haenlin 100 points

Nous estimons avoir bien travaillé et ça en équipe sur l’ensemble du projet en prenant soin de profiter des heures de TD pour discuter et planifier les tâches à faire pour la semaine qui suit.