SI4

Rendu 2 - Conception Logicielle

Équipe L

MEULLE Nathan, ESTEVE Thibaut, DADIOUARI Asaréel,
DONTO Komlan Lemuel et DELMOTTE Vincent

2020-2021

POLYTECH NICE SOPHIA | UNIVERSITÉ CÔTE D'AZUR

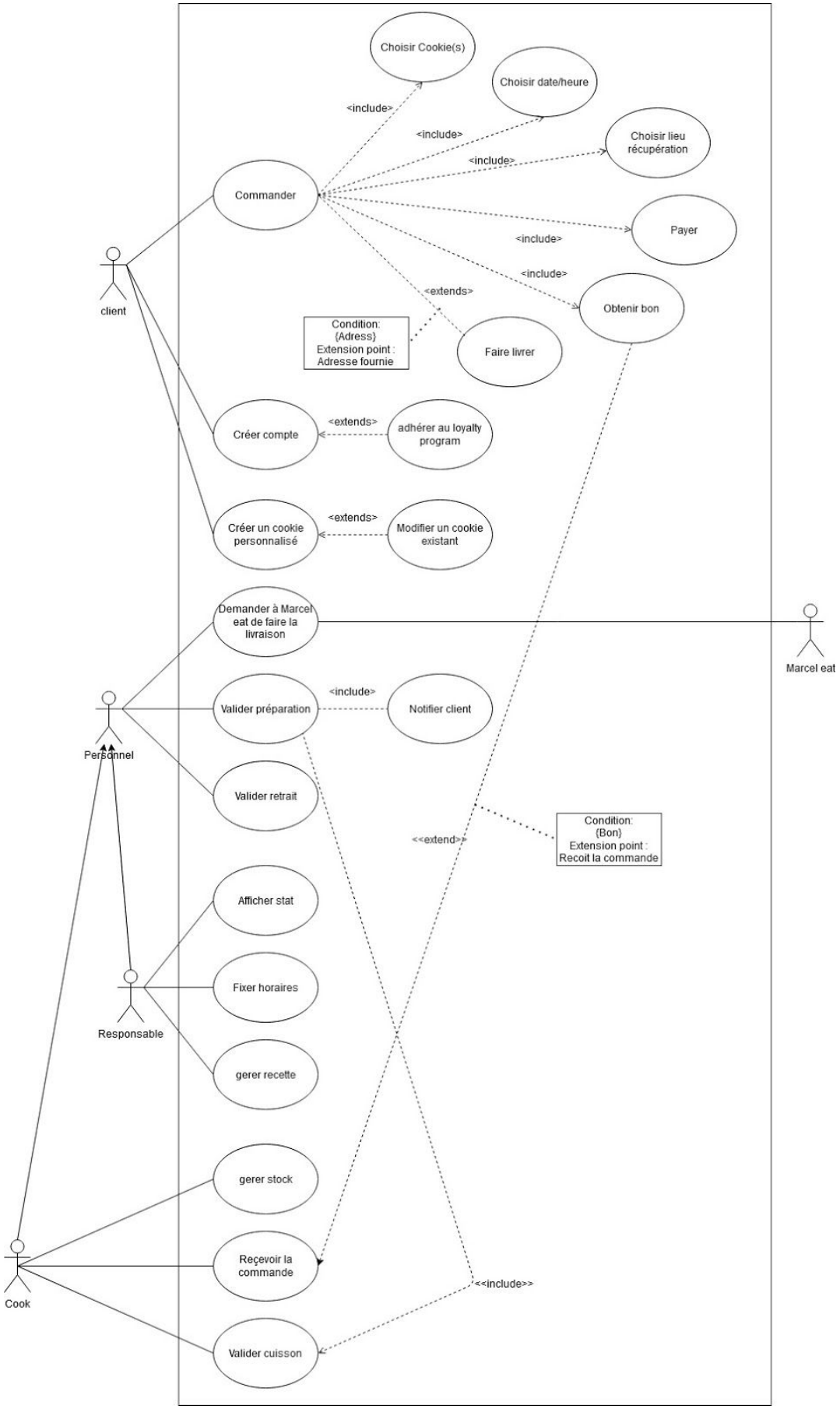
Sommaire

L’objectif de ce projet est de mettre en œuvre un service informatique de commande de cookie pour l’entreprise *The Cookie Factory™*, une entreprise leader aux Etat-unis dans la confection et la distribution de cookies artisanaux.
Ce service permet aux clients de faire leur sélection de cookies parmi la gamme proposée, d’en faire la commande et d'indiquer la date et l’heure de récupération. Ainsi le client pourra récupérer des cookies fraîchement préparés avec amour.

Les clients ont la possibilité de commander leur cookie parmi la gamme proposée ou bien de créer leur propre cookie personnalisé. Une commande peut bien sûr contenir plusieurs cookies et, une fois réalisée, est traitée par notre système informatique.

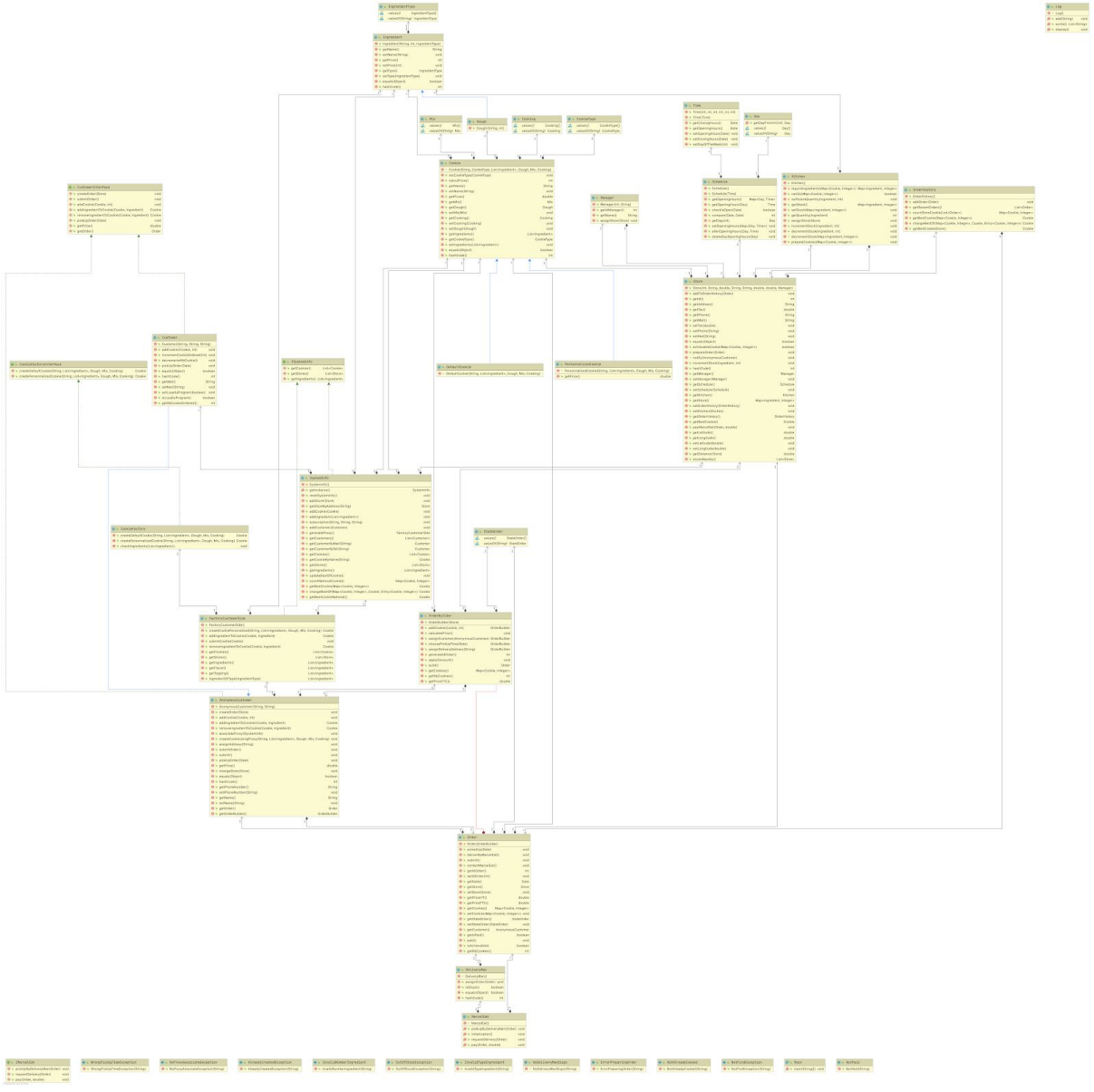
A travers ce rapport, l’équipe L vous dévoile ses recettes et choix de conception les plus secrets !

I. Diagramme des cas d'utilisation



II. Diagramme de classes

Voici notre diagramme de classes global :
(Téléchargeable ici)



Cf : détail en Annexe

III. Patrons de conception

A. Employés

- **Singleton**

Nous avons choisi d'appliquer le patron de conception **Singleton** sur notre classe *SystemInfo* qui fait office de base de données : elle contient tous les clients, recettes, magasins, et ingrédients au niveau national. Cela nous permet d'avoir une unique instance de cette classe.

Grâce à notre unique méthode *getInstance* nous pouvons accéder à l'instance de la classe. Cela nous permet de contrôler l'accès à notre base de données.

- **Factory**

Nous avons mis en place une **Factory** afin de créer nos différents cookies. Nous avons donc une classe (*CookieFactory*) implémentant une interface (*CookieFactoryInterface*). Cette classe permet de déléguer la création des cookies : *DefaultCookie* / *PersonnalizedCookie*, héritant de la classe abstraite Cookie.

Utiliser une factory nous a également permis d'encapsuler la logique de création des cookies et de simplifier la création de n'importe quel produit : il suffit d'appeler la méthode correspondante sur la factory pour créer l'objet souhaité. Par ailleurs, nous avions confié à la Factory la charge de vérifier les ingrédients fournis : ainsi, un cookie sans pâte ne peut être créé.

- **Builder**

Nous avons choisi d'appliquer le patron de conception **Builder** sur la classe *Order*. L'objectif de cette implémentation est de séparer la création d'un objet de sa représentation.

En effet *Order* est l'un des objets les plus complexes de ce programme. Avec un même mécanisme de création on peut arriver à différentes représentations. Nous avons donc retenu le patron Builder qui nous permet de construire la commande pas à pas : nous pouvons ajouter des cookies, sélectionner l'heure et le lieu de récupération… Enfin, la méthode *build* construit le véritable objet *Order* avec tous ses attributs.

- **Proxy**

Nous avons fait le choix de mettre un **Proxy** afin de rendre l'instance du *SystemInfo* (ie: notre base de données) accessible aux clients. Notre classe *FactoryCustomerSide* joue ce rôle de proxy et va retourner uniquement les informations nécessaires aux clients, c'est-à-dire renvoyer une copie de la liste des magasins, une copie des cookies existants et une copie des ingrédients répertoriés. En effet, nos clients n'ont pas besoin d'avoir accès à tout le *SystemInfo* mais seulement à certains de ces attributs (par exemple un client ne devrait pas avoir la possibilité de connaître la liste de tous les clients).

Afin de respecter le modèle de proxy vu en cours, nous avons créé une interface *ISystèmeInfo* que nous avons implémentée dans notre classe *SystèmeInfo* (notre real subject) et dans notre *FactoryCustomerSide* (le proxy utilisé par le client). Ainsi le client s'attend à utiliser l'interface *ISystèmeInfo* sans savoir s'il s'agit du vrai *SystèmeInfo* ou de son proxy *FactoryCustomerSide*. Ainsi notre client peut accéder aux ingrédients et aux cookies dont il a besoin pour créer ses propres cookies sans qu'il n'y ait de risque de modifier les ingrédients ou les cookies enregistrés dans le *SystèmeInfo*.

B. Envisagés

- **Etat**

Nous avons considéré l'utilisation du patron de conception **État** pour la gestion de l'état de nos commandes car celui-ci permettrait d'avoir une interface offrant la possibilité d'altérer l'état d'une commande. Dans le but de réduire la complexité et la verbosité dans notre code et ainsi éviter de faire l'*over engineering,* nous avons plutôt opté pour l'utilisation d'une énumération StateOrder pour définir l'état de nos commandes.

- **Observer**

Nous avons envisagé l'utilisation d'un **Observer** dans la *factoryCustomerSide* afin de mettre à jour les cookies proposés au client suivant s'il y a assez d'ingrédients pour les réaliser ou non.

Pour cela, il aurait fallu indiquer à la *factoryCustomerSide* les magasins (*Store*) où le client commande puisque le stock de la cuisine (*Kitchen*) serait observé. Une fois le stock réapprovisionné, le store notifie toutes les recettes et affiche celles disponibles.

Cependant, comme nous traitions déjà le cas où il n'y a pas assez d'ingrédients dans un magasin avec le Kitchen Balancing, nous avons décidé de ne pas nous concentrer sur cette implémentation d'Observer.

IV. Rétrospective

A. Conception

Chaque cookie est identifié par un nom et composé de 5 caractéristiques. Sa pâte (dough), le mélange (mix) et sa cuisson (cooking) sont les 3 attributs obligatoires pour créer un cookie, mais nous pouvons aussi choisir un parfum particulier (flavor) ainsi que jusqu'à 3 garnitures (topping) (qui peuvent être ajoutées au cookie sous forme d'une liste d'ingrédients passée en paramètre). La factory que nous avons créée s'assure que les cookies sont bien formés. De plus, celle-ci nous permet de facilement ajouter de futurs produits : donuts, crèpes, gauffres, seraient facile à intégrer au code existant.

Le client peut ajouter autant de cookies qu'il le souhaite à sa commande grâce au builder que nous avons mis en place. Lors de la soumission de la commande, on vérifie d'abord si la commande est réalisable (s'il y a assez d'ingrédients en stock dans le magasin affilié à la commande).

Quelques semaines après le début du développement, des **fonctionnalités supplémentaires** nous ont été demandées par le client : tout d'abord, la possibilité pour le client de **créer ses propres cookies** personnalisés (en partant de zéro ou en modifiant un existant).
Ensuite, si pour une quelconque raison la commande n'est pas réalisable, nous proposons au client de venir récupérer sa commande dans **un autre magasin à proximité**. Dans le cadre de notre découpage, nous avons implémenté cette fonctionnalité en cas d'ingrédient manquant (les autres incidents n'ont pas été pris en compte).

Et enfin la possibilité de **faire livrer ses cookies** en passant par un service tiers du nom de Marcel Eat. En effet, l'API *MarcelEat* propose la livraison à domicile des commandes réalisées à *The Cookie Factory*.

Nous avons implanté le système de **livraison planifiée** : le customer renseigne donc son adresse lorsqu'il passe sa commande. Nous allons donc *submit* la commande, si cette dernière est validée, le store contacte MarcelEat via l'API. Une fois préparé, un livreur vient chercher la commande et la livre à un client. *MarcelEat* est payé par le store qui a facturé au préalable le customer. Ce prix a été fixé arbitrairement à deux euros. Les informations (order) sont passées par les objets en eux même au sein de cette architecture. On pourrait très bien imaginer, dans un contexte plus professionnel, de faire transiter les données par des fichiers json.

L'autre service (que nous n'avons pas implanté) est la commande de dernier moment, celui-ci devait reposer sur le même mécanisme que le premier service. Le client soumet (via *submit*) sa commande mais sans renseigner son adresse. La commande part en préparation, si le client contacte le store pour demander une livraison de dernière minute, le

store va prélever une nouvelle fois un supplément pour cette livraison puis va contacter *MarcelEat* pour avoir un livreur.

En vue d'un prochain développement, nous améliorerons notre **gestion des principes de Tee shirt Sizing et MoSCoW**. En effet, nous avions dans un premier temps indiqué ces informations au sein d'un Google Document : l'user story était présente sur Github mais sa taille et son importance n'y était pas, ce qui nous contraignait à regarder à deux endroits différents. Nous avons ensuite corrigé le tir et **intégré directement ces informations au sein du projet Github** au travers des labels.

B. Qualité

Nous sommes satisfaits de la qualité de notre projet dans la mesure où nous avons réussi à conserver une bonne couverture de tests tout au long du projet.
Nous avons associé des tests unitaires permettant de tester spécifiquement nos méthodes et des tests BDD (*Behaviour Driven Development*) associés à nos *User Story*.
Cela nous permet d'avoir, en fin de projet, **une couverture de 87%** de nos lignes (80 tests unitaires et 56 scénarios).

L'utilisation de GithubAction (*build with maven*) nous a permis d'effectuer de l'**intégration continue** en automatisant la vérification. Cela nous assure que chaque commit ne remet pas en cause les précédents tests.

Nous avons également associé à notre projet à **SonarQube** : notre code est de qualité puisqu'il comporte peu de code smells et pas de bug : nous avons 27 code smells soit 5h de dette ce qui est peu. (La plupart liée à l'utilisation de méthodes *Deprecated* pour gérer les Dates).

Une qualité moindre aurait rendu notre code plus difficilement compréhensible, avec probablement plus d'erreurs : par exemple, Sonar nous a permis d'identifier rapidement les passages de code sensibles aux erreurs : possible *NullPointer*, variable non initialisée, complexité trop grande...

V. Auto-Évaluation

MEULLE Nathan	ESTEVE Thibaut	DADIOUARI Asaréel	DONTO Komlan Lemuel	DELMOTTE Vincent
105	100	100	95	100

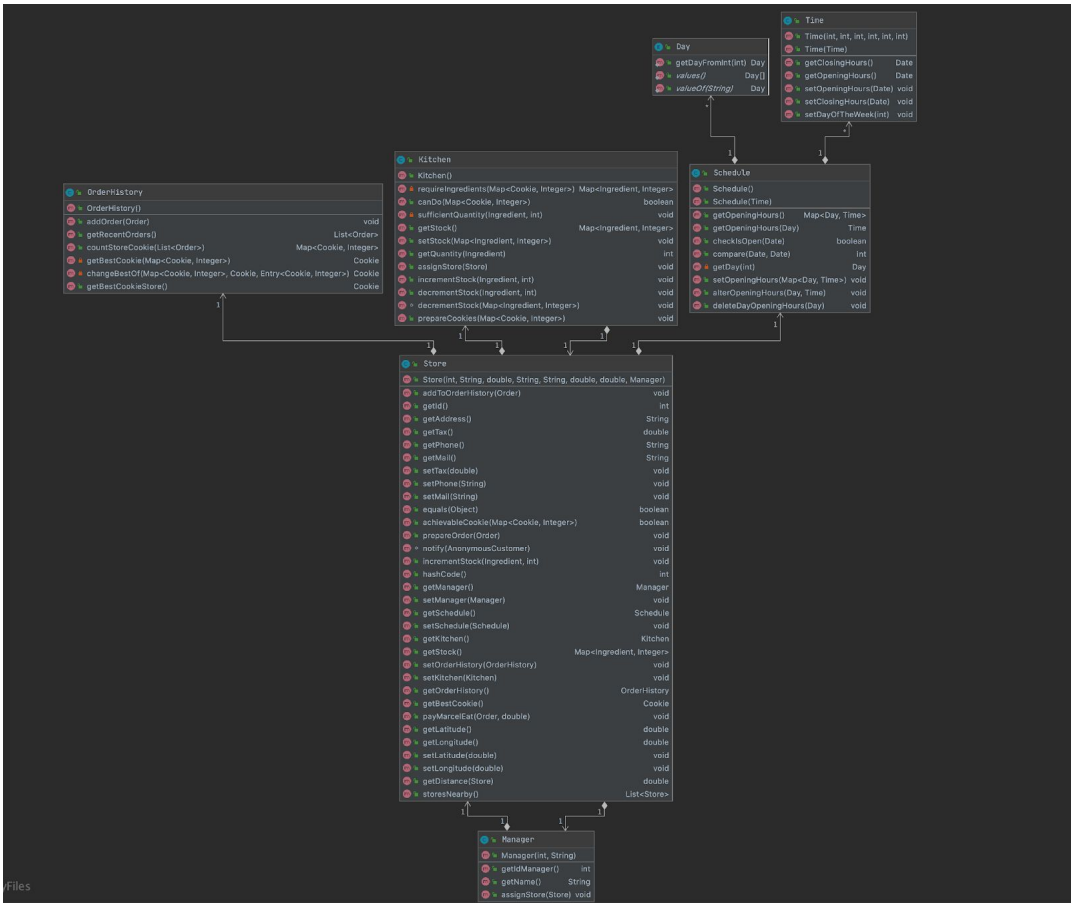
Concernant la gestion du projet, chaque membre a contribué à l'avancée de celui-ci.
Nous avons cependant éprouvé quelques difficultés de communication (du fait du passage à distance notamment) au sein de l'équipe.

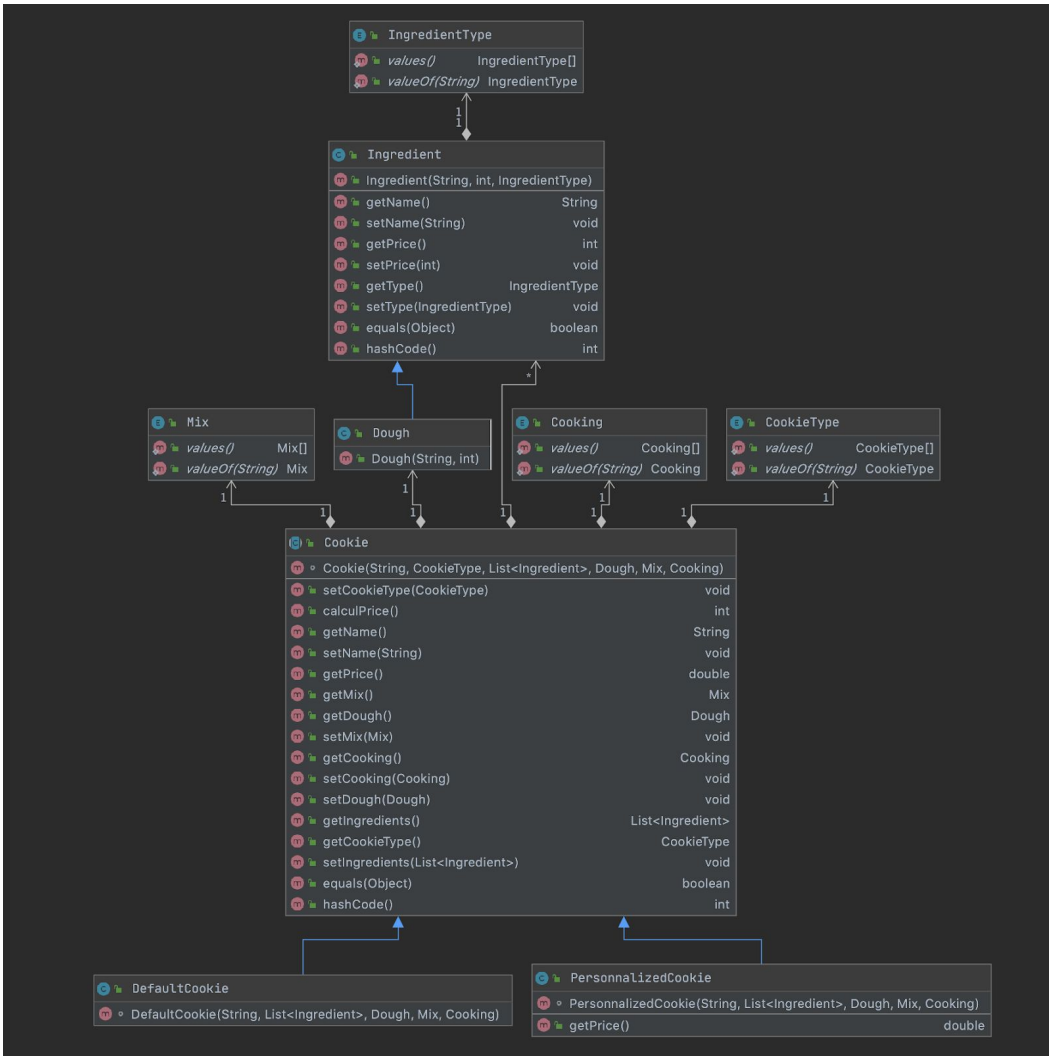
VI. Conclusion

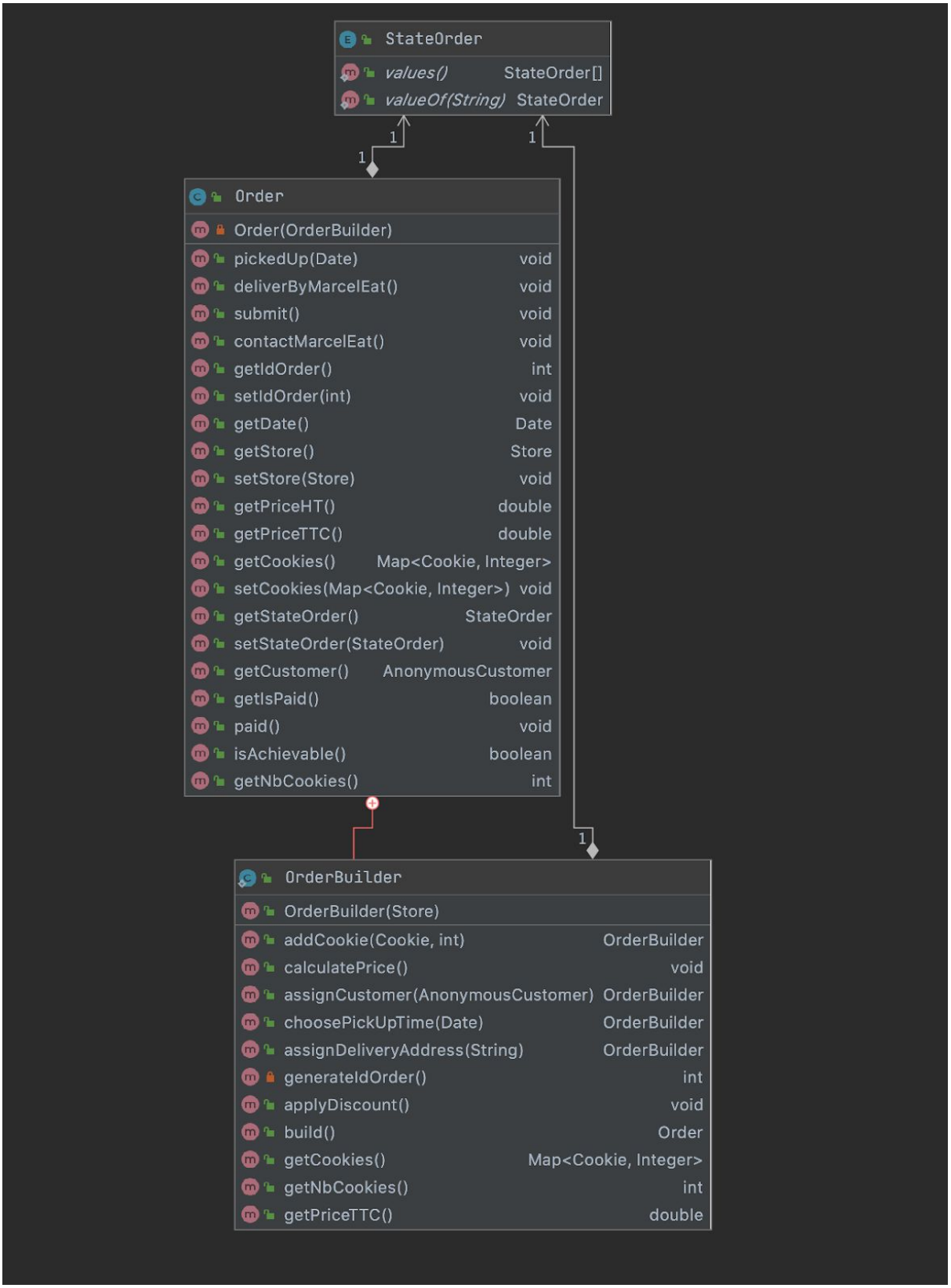
Nous sommes dans l'ensemble **satisfaits du travail accompli**. Bien que nous n'avons pas implémenté chaque fonctionnalité dans son entièreté, nous avons construit pour chacune d'entre elles **une version minimale**.

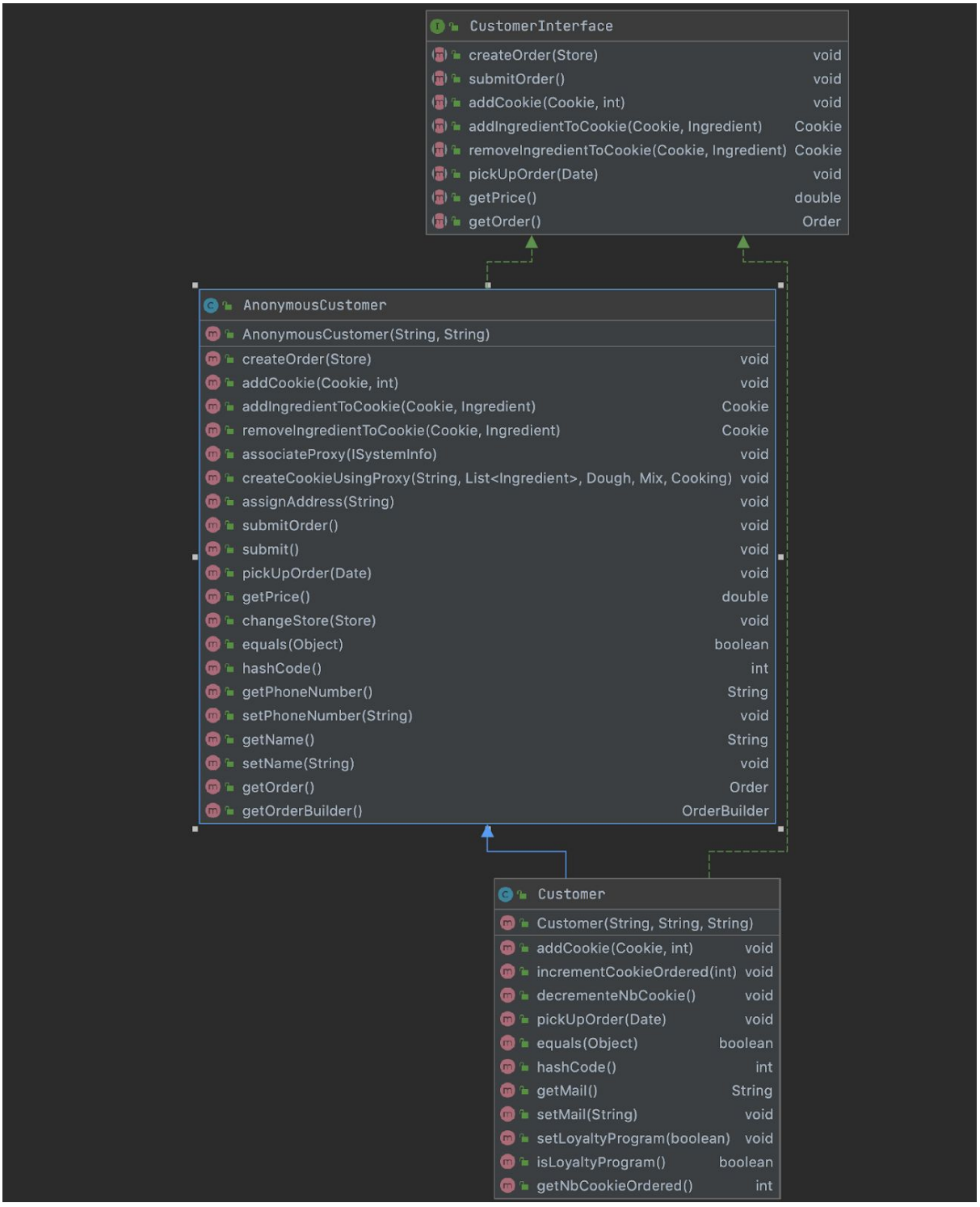
Grâce à ce projet, nous avons pu découvrir de nombreux **aspects de conception** et ainsi **appliquer certains patrons**. Nous sommes maintenant capables d'analyser l'architecture d'une application et de justifier les choix de conception. L'utilisation des patrons de conception ainsi que les outils de de modélisation de normes UML ont permis l'**enrichissement de notre bagage technique**.

Annexe


Day
getDayFromInt(int) Day
values() Day[]
valueOf(String) Day
Time
Time(int, int, int, int, int, int)
Time(Time)
getClosingHours() Date
getOpeningHours() Date
setOpeningHours(Date) void
setClosingHours(Date) void
setDayOfTheWeek(int) void
Kitchen
Kitchen()
requireIngredients(Map<Cookie, Integer>) Map<Ingredient, Integer>
canDo(Map<Cookie, Integer>) boolean
sufficientQuantity(Ingredient, int) void
getStock() Map<Ingredient, Integer>
setStock(Map<Ingredient, Integer>) void
getQuantity(Ingredient) int
assignStore(Store) void
incrementStock(Ingredient, int) void
decrementStock(Ingredient, int) void
decrementStock(Map<Ingredient, Integer>) void
prepareCookies(Map<Cookie, Integer>) void
Schedule
Schedule()
Schedule(Time)
getOpeningHours() Map<Day, Time>
getOpeningHours(Day) Time
checkIsOpen(Date) boolean
compare(Date, Date) int
getDay(int) Day
setOpeningHours(Map<Day, Time>) void
alterOpeningHours(Day, Time) void
deleteDayOpeningHours(Day) void
OrderHistory
OrderHistory()
addOrder(Order) void
getRecentOrders() List<Order>
countStoreCookie(List<Order>) Map<Cookie, Integer>
getBestCookie(Map<Cookie, Integer>) Cookie
changeBestOf(Map<Cookie, Integer>, Cookie, Entry<Cookie, Integer>) Cookie
getBestCookieStore() Cookie
Store
Store(int, String, double, String, String, double, double, Manager)
addToOrderHistory(Order) void
getId() int
getAddress() String
getTax() double
getPhone() String
getMail() String
setTax(double) void
setPhone(String) void
setMail(String) void
equals(Object) boolean
achievableCookie(Map<Cookie, Integer>) boolean
prepareOrder(Order) void
notify(AnonymousCustomer) void
incrementStock(Ingredient, int) void
hashCode() int
getManager() Manager
setManager(Manager) void
getSchedule() Schedule
setSchedule(Schedule) void
getKitchen() Kitchen
getStock() Map<Ingredient, Integer>
setOrderHistory(OrderHistory) void
setKitchen(Kitchen) void
getOrderHistory() OrderHistory
getBestCookie() Cookie
payMarcelEat(Order, double) void
getLatitude() double
getLongitude() double
setLatitude(double) void
setLongitude(double) void
getDistance(Store) double
storesNearby() List<Store>
Manager
Manager(int, String)
getIdManager() int
getName() String
assignStore(Store) void


IngredientType
values() IngredientType[]
valueOf(String) IngredientType
Ingredient
Ingredient(String, int, IngredientType)
getName() String
setName(String) void
getPrice() int
setPrice(int) void
getType() IngredientType
setType(IngredientType) void
equals(Object) boolean
hashCode() int
Mix
values() Mix[]
valueOf(String) Mix
Dough
Dough(String, int)
Cooking
values() Cooking[]
valueOf(String) Cooking
CookieType
values() CookieType[]
valueOf(String) CookieType
Cookie
Cookie(String, CookieType, List<Ingredient>, Dough, Mix, Cooking)
setCookieType(CookieType) void
calculPrice() int
getName() String
setName(String) void
getPrice() double
getMix() Mix
getDough() Dough
setMix(Mix) void
getCooking() Cooking
setCooking(Cooking) void
setDough(Dough) void
getIngredients() List<Ingredient>
getCookieType() CookieType
setIngredients(List<Ingredient>) void
equals(Object) boolean
hashCode() int
DefaultCookie
DefaultCookie(String, List<Ingredient>, Dough, Mix, Cooking)
PersonnalizedCookie
PersonnalizedCookie(String, List<Ingredient>, Dough, Mix, Cooking)
getPrice() double


StateOrder
values() StateOrder[]
valueOf(String) StateOrder
1
1
1
1
Order
Order(OrderBuilder)
pickedUp(Date) void
deliverByMarcelEat() void
submit() void
contactMarcelEat() void
getIdOrder() int
setIdOrder(int) void
getDate() Date
getStore() Store
setStore(Store) void
getPriceHT() double
getPriceTTC() double
getCookies() Map<Cookie, Integer>
setCookies(Map<Cookie, Integer>) void
getStateOrder() StateOrder
setStateOrder(StateOrder) void
getCustomer() AnonymousCustomer
getIsPaid() boolean
paid() void
isAchievable() boolean
getNbCookies() int
OrderBuilder
OrderBuilder(Store)
addCookie(Cookie, int) OrderBuilder
calculatePrice() void
assignCustomer(AnonymousCustomer) OrderBuilder
choosePickUpTime(Date) OrderBuilder
assignDeliveryAddress(String) OrderBuilder
generateIdOrder() int
applyDiscount() void
build() Order
getCookies() Map<Cookie, Integer>
getNbCookies() int
getPriceTTC() double


CustomerInterface
createOrder(Store) void
submitOrder() void
addCookie(Cookie, int) void
addIngredientToCookie(Cookie, Ingredient) Cookie
removeIngredientToCookie(Cookie, Ingredient) Cookie
pickUpOrder(Date) void
getPrice() double
getOrder() Order
AnonymousCustomer
AnonymousCustomer(String, String)
createOrder(Store) void
addCookie(Cookie, int) void
addIngredientToCookie(Cookie, Ingredient) Cookie
removeIngredientToCookie(Cookie, Ingredient) Cookie
associateProxy(ISystemInfo) void
createCookieUsingProxy(String, List<Ingredient>, Dough, Mix, Cooking) void
assignAddress(String) void
submitOrder() void
submit() void
pickUpOrder(Date) void
getPrice() double
changeStore(Store) void
equals(Object) boolean
hashCode() int
getPhoneNumber() String
setPhoneNumber(String) void
getName() String
setName(String) void
getOrder() Order
getOrderBuilder() OrderBuilder
Customer
Customer(String, String, String)
addCookie(Cookie, int) void
incrementCookieOrdered(int) void
decrementeNbCookie() void
pickUpOrder(Date) void
equals(Object) boolean
hashCode() int
getMail() String
setMail(String) void
setLoyaltyProgram(boolean) void
isLoyaltyProgram() boolean
getNbCookieOrdered() int