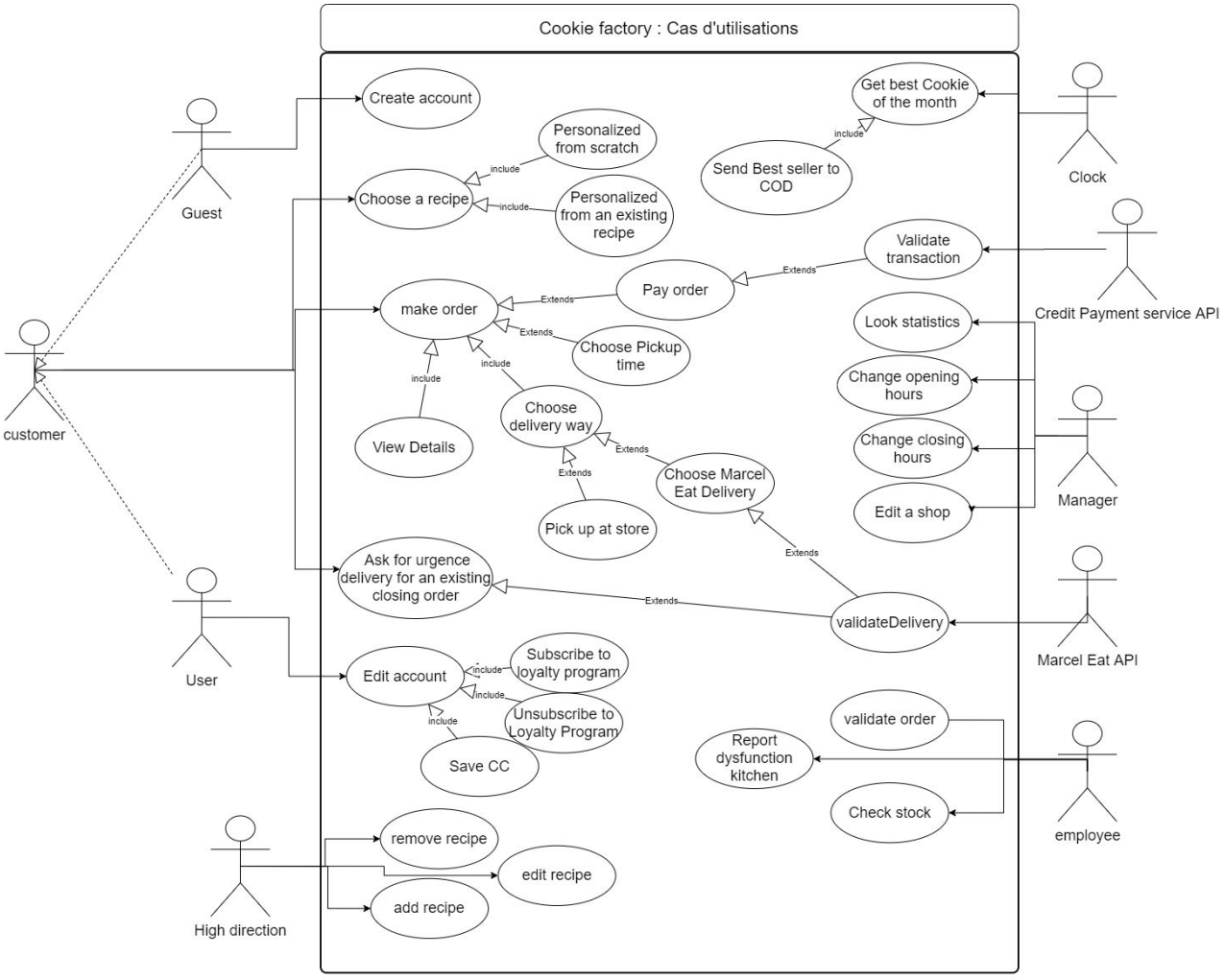
Rapport Conception logicielle : Équipe K

Membres:

El Adlani Rachid, Kaplan Julien, Fertala Mohamed, Belkhiri Abdelouhab, Fargeon Armand

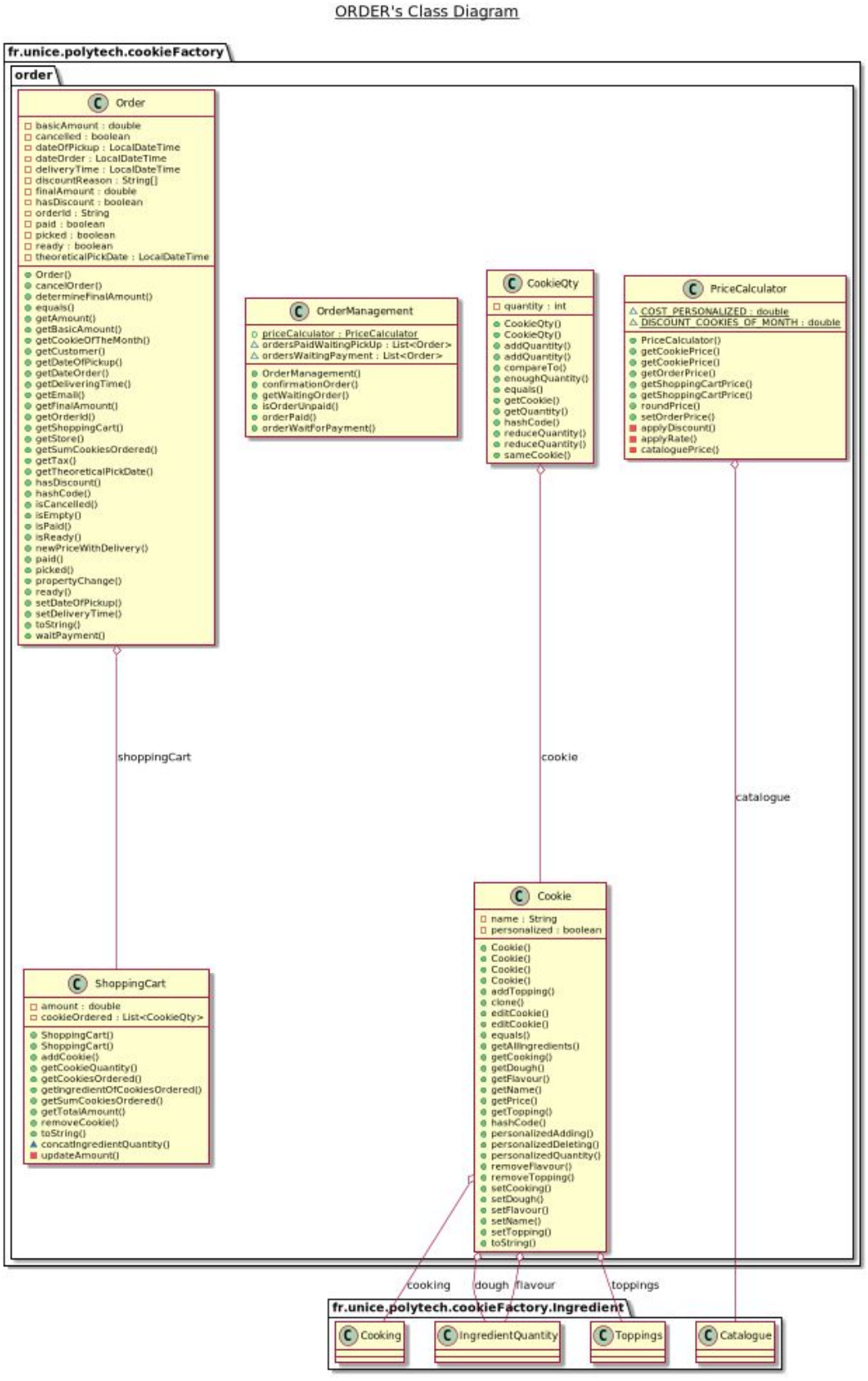
Sommaire

1) Diagramme cas d'utilisation



2) Diagramme de classe : UML

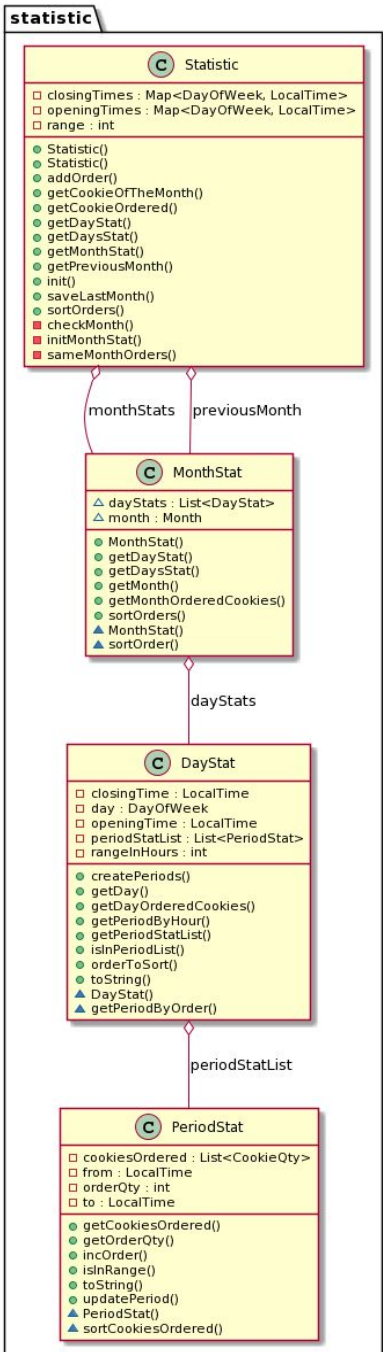
Partie Gestion des commandes :



La gestion des commandes se fait depuis *OrderManagement*. C'est cette classe qui aura la responsabilité de gérer une commande entre le moment où celle-ci est en attente de paiement et celui où, lorsqu'elle est payée, elle doit être transférée au magasin qui s'en occupe.

C'est donc avec la classe *PriceCalculator* que le prix final d'une commande sera déterminé. En effet, *OrderManagement* à connaissance des informations d'une commande et donc également des réductions et taxes applicables à celle-ci. De plus, *PriceCalculator* pourra à l'aide du catalogue, général à toutes les franchises cookies factory, vérifier que chaque cookie au sein d'une commande existe bien dans les cookies officiellement répertoriés, et dans le cas contraire, appliquer un coût supplémentaire. Ce catalogue pourra également donner le cookie élu *National Best Cookie Of The Month*.

Partie Statistiques :



Chacun des magasins possède une instance “Statistic” qui lui est propre. Cette classe est construite à partir des horaires du magasin ainsi qu’un entier représentant en combien d’heures les périodes d’une journée doivent être découpées.

Une commande finalisée sera ajoutée puis triée dans le mois en cours.

On fait ici la distinction par le “mois en cours” puisque la classe *Statistic* possède aussi une seconde instance *MonthStat*, qui est une sauvegarde des statistiques du mois précédent. C’est d’ailleurs cette classe qui sera importante pour toutes statistiques nécessaires à un instant t comme la récupération du *BestCookieOfTheMonth* ou les périodes d'affluence.

3) Choix de patrons de conception

3.1. Patrons de conceptions retenus

- **State**

Le store a un attribut du type *Kitchen*.

La cuisine a un attribut *kitchenState* peut être dans 2 états:
- l’état “broken” où le magasin ne pourra plus traiter des commandes.
- l’état “ok” qui permettra de traiter les commandes.

L’état sera ajouté dans la classe *Kitchen*. Cela permet au store de vérifier avant de prendre une commande, si la cuisine est en mesure de traiter une nouvelle commande ou non.

- **Observer**

Le pattern *Observer* va permettre de notifier, en cas de dysfonctionnement de la cuisine, le Store concerné qui va se mettre par la suite en état “closed”, pour ne plus accepter de nouvelles commandes.

Le store avertira ensuite le *StoreManagement* qui se chargera de rediriger les commandes ne pouvant être traitées par le store concerné puis les redirigera une à une au store le plus proche pouvant les recevoir.

StoreManagement observe tous les stores qui sont ajoutés dans la liste, par conséquent on aura cette architecture d’observer - Observable

- **Prototype**

Le patron de conception prototype a été utilisé pour créer les cookies sur demande en générant une copie du premier cookie de départ, et en lui modifiant certains ingrédients selon la demande du client, sans que ces derniers n'impactent le cookie de base.

- **Façade**

Le patron de conception façade nous permet de fournir à l'utilisateur une interface "simple" où derrière il y a un système complexe. Ici, notre façade, c'est la classe *COD* qui centralise les différents éléments tels que la création d'une commande, la redirection de magasins, le paiement…. Il permet aux différents utilisateurs (clients, managers) d'accéder aux différentes méthodes dont ils ont besoin, et ensuite de répartir les tâches aux autres classes.

On a donc un regroupement de classes en une seule, ce qui permet de limiter le couplage entre les différentes fonctionnalités. Grâce à l'implémentation de ce patron de conception, cela permet un découpage en sous-systèmes, on réduit donc la complexité du système en distribuant ses responsabilités entre des classes.

Ce modèle est également appliqué aux classes *StoreManagement*, *OrderManagement*

- **Template méthode (héritage)**

Ce patron de conception qui est le plus souvent utilisé, permet de créer une abstraction de classe. Cette abstraction de classe permet d'assurer l'évolutivité du code et aussi de permettre de mettre en commun plusieurs classes qui hériteront d'une seule classe.

Dans notre cas, on aura la classe Ingrédient qui est une classe mère pour *Dough, Flavour, Topping* car elle possède tous les mêmes attributs (name, price).

La classe *Customer* est une classe mère pour *User* et *Guest*, cela nous permet de mettre en commun ce que peuvent faire au minimum un visiteur (*Guest*) et un utilisateur inscrit (*User*).

3.2. Patrons de conceptions intéressants non retenus

- **Build**

Nous avons pensé au pattern *Build* vers la fin du projet en effectuant la rétrospective, pour la création des cookies. Ce patron aurait pu alléger le processus de création d'un cookie. Après avoir déjà implémenté une version que nous trouvions plutôt satisfaisante, nous n'avons pas continué à explorer les possibilités et c'est en phase finale que nous avons remarqué que ce patron aurait été en parfaite adéquation avec la création de cookies. Nous aurions pu par exemple créer des cookies sans Flavour, or actuellement, c'est un Flavour "nature" qui est ajouté.

- **Singleton**

Le pattern *Singleton* aurait pu être implémenté dans notre code concernant la partie COD, ou nous aurions pu le lier au pattern Façade afin de garantir une seule instance de COD durant toute l'exécution.

Cependant, nous avons jugé plus intéressant de diminuer les responsabilités sur le *COD* en les dispatchant sur les nouvelles classes *StoreManagement*, *OrderManagement*, *ApiManagement* et *CustomerManagement*. Ce choix permettait de garder le *COD* comme classe *Façade*, et éviter que toutes les classes se voient communiquer avec cette classe.

4) Rétrospective

4.1 Rétrospective concernant l'architecture

- **Gérer les stores**

On a pensé les choses selon le "Domain First". On a pu respecter le concept du Domain Driven Design qui permettait de penser métier en premier et de ne pas se focaliser sur aucun une autre chose que sur le métier, c'est pour cela que nous avons en premier lieu cherché à comprendre le fonctionnement d'un système tel que Cookie Factory.

De plus, concernant les user stories nous avons pris le temps de segmenter les actions que les différents acteurs de cookie factory pouvaient faire en produisant le diagramme d'utilisation de CookieFactory.

Nous avons créé un glossaire à partir des informations données dans les différents TP ce qui nous a aidé à créer les user stories.

C'est lorsque nous avons procédé au T-shirt sizing grâce au votes des membres de l'équipe, que nous nous sommes aperçus que certaines tâches étaient mal découpées car beaucoup trop importantes.

- **Gestion des quantités d'ingrédients**

Nous avions tout d'abord choisi pour la gestion des quantités l'utilisation de *hashmap*. Le traitement nécessaire sur celles-ci étaient au départ acceptables, mais les ajouts du sujet nous ont obligé à revoir cette partie. En effet, même si l'utilisation de *stream* sur les *hashmap* donnait du code assez compact au premier abord, mais ces méthodes auraient dû être dupliquées à différents endroits dans le code.

En effet, c'est lorsque le stock à été ajouté au sujet que le choix d'utilisation d'une *Hashmap* pour les quantités d'ingrédients a été abandonné et que l'objet *IngredientQty* a été implémenté répondant de façon précise au besoin.

Ainsi, cet objet pouvait aussi bien servir à la tenue des stocks d'un magasin qu'à la création d'un cookie.

- **Statistiques**

L'implémentation des statistiques s'est faite assez naturellement. La première implémentation des statistiques sur les jours de la semaine était assez bonne, ce qui a permis de s'adapter sans difficultés avec la seconde version des statistiques (l'ajout des mensuelles). C'est d'ailleurs une fois que les statistiques ont été finalisées que nous avons pu grâce aux méthodes existantes, récupérer le *National Best Cookie* grâce aux statistiques de tous les magasins.

- **Calcul du prix d'une commande**

Le calcul du prix d'une commande était au début du projet fait grâce au prix de chacun des cookies la constituant. Ce choix a dû être rapidement abandonné face aux diverses variables à traiter sur le prix d'une commande. C'est pourquoi nous avons choisi de donner la responsabilité à chaque objet entrant en jeu dans le prix d'une commande de calculer lui-même son prix afin d'obtenir récursivement le prix total d'une commande. C'est par contre à la classe *PriceCalculator* que revient la responsabilité d'appliquer les diverses taxes, les surcoûts et les réductions au prix final.

4.1 Rétrospection concernant le github

Nous avons décidé de partir sur une ***branching strategy*** de type git flow où chaque nouvelle fonctionnalitée était implémentée dans une branche à part, une fois la fonctionnalité implémentée et testée, la branche était merge sur la branche principale par l'intermédiaire de "pull request" qui permettent de **valider le code par les différents membres de l'équipe et ainsi tout centraliser.**

Dès le début du projet nous nous sommes organisés sur des documents communs et sur slack, mais nous n'avons pas rentré nos User Story ou nos tâches au fur et à mesure dans github, nous avons pris conscience très tard de cette erreur. Nous avons donc dû créer vers la fin du projet les **issues / labels** github et apporter de la cohérence entre nos **User Story** dans le code et sur le github.

5) Répartition du travail

Voici notre répartition du travail :

Nom	Points
El Adlani Rachid	100
Kaplan Julien	100
Fargeon Armand	100
Fertala Mohamed	100
Belkhiri Abdelouhab	100
Total (5 membres)	500