Rapport de TP

Antunes Benjamin
Nekkaa Yousra

Sujet du TP : Kata refactoring GildedRose .

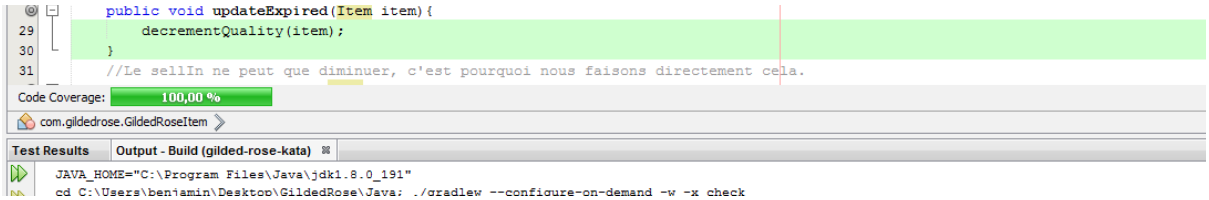
Démarche Suivie :

Dans un premier temps, nous avons créé notre dépôt git. Git est un gestionnaire de version, il nous a permis de sauvegarder chaque étape du refactoring.

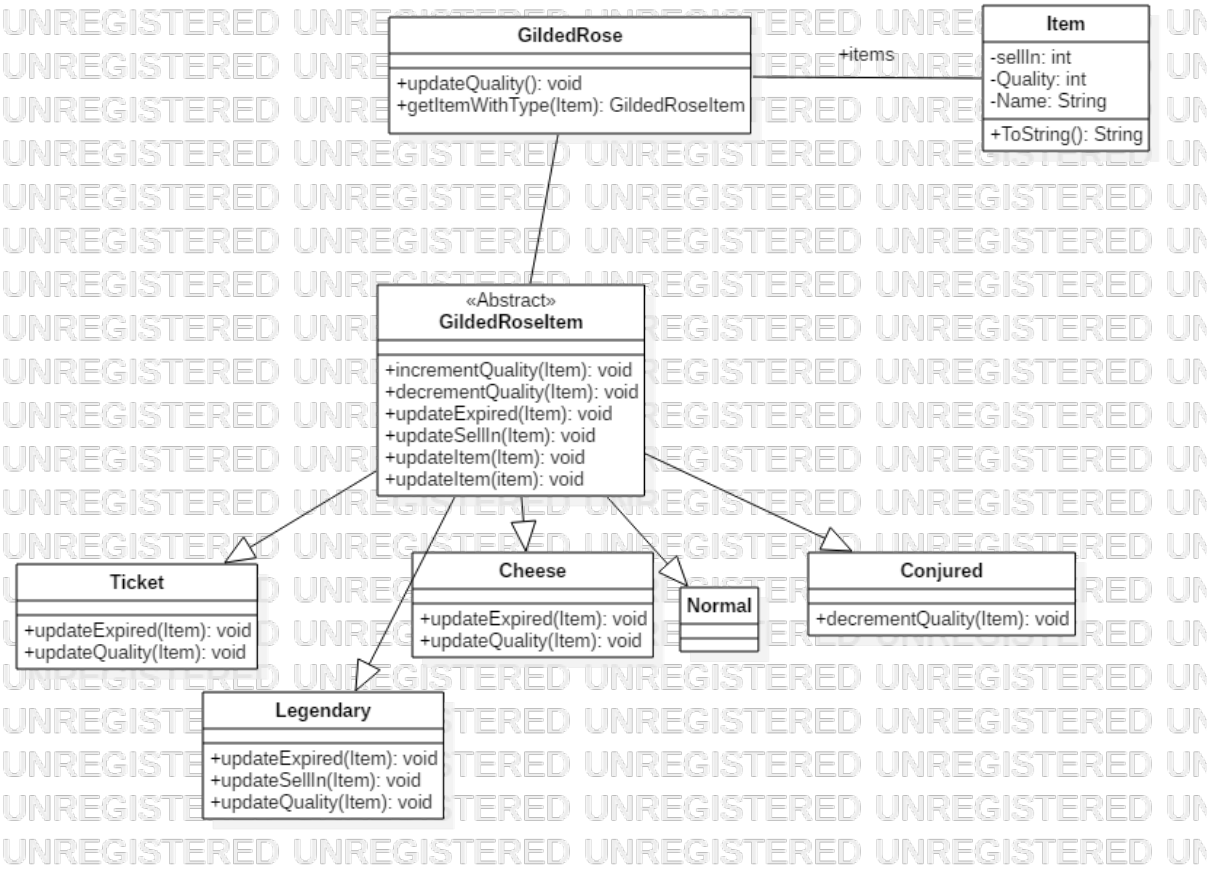
Ensuite, avant de procéder au refactoring, nous avons réalisé tous les tests unitaires, pour chaque fonction décrite dans le sujet de GildedRose. Ainsi, nous testons la bonne baisse des qualités, augmentation, caractères particuliers pour les Aged Brie, BackstagesPasses, etc.
Malgré la complexité initiale de la fonction de base, tous les tests passent ! (Ce qui est tout à fait normal).

Ensuite, nous avons commencé le refactoring.
Nous observons que l'objectif est de mettre à jour la Quality et le SellIn à chaque tour de boucle. Nous commençons donc par sortir des méthodes updateQuality() et updateSellIn().
La qualité parfois augmente, parfois diminue. Nous créons les méthodes incrementQuality() et decrementQuality(), pour clarifier le code. Une méthode pour une fonctionnalité. Nous observons également qu'un comportement est différent si l'item est périmé. Alors nous créons la méthode updateExpiredItem(). Nous avons dû remanier cette énorme méthode qu'était updateQuality(). A ce niveau, nous avions toujours des if avec les item.name.equals de chaque item. A la lecture du sujet, nous comprenons qu'il existe des types d'objets, et qu'il va nous falloir en rajouter un, qui est "Conjured". Chaque objet ayant un comportement spécifique, nous décidons de créer une classe pour chaque type d'objet, héritant d'une classe mère qui définirait les fonctionnalités de base que doivent contenir les objets, comme updateQuality(),SellIn,.... Mais en définissant pour eux même le comportement. Cela correspond au patron de conception méthode. Par exemple, un objet Normal ne modifie pas le comportement de base, un objet légendaire ne change pas en qualité, un objet Conjured perd en qualité deux fois plus rapidement.
Évidemment, le mieux à faire aurait été de modifier directement la classe Item en ajoutant directement un attribut qui définit le type de l'item (Legendary,Cheese,Conjured,...) mais cela nous été interdit .
Et donc, pour palier a ce souci, nous avons dû utiliser le nom de l'objet pour nous retourner le type de l'objet.
Nous avons utilisé le polymorphisme. Ainsi, nous travaillons dans la méthode de base updateQuality() sur des objets du type de la classe mère GildedRoseItem, mais avec des objets de type réel Legendary, Cheese, etc.., héritant de la classe mère.

Finalement, pour nous assurer du bon état de notre code, nous avons effectué un test de code coverage, qui est de 100%, donc les tests couvrent bien les fonctionnalités du code.



Architecture du système :



Nous avons utilisé le patron de conception Méthode pour définir les opérations que devront réaliser tous les types d'objets différents. Cela rend le projet extensible, puisque de cette façon, nous pouvons facilement ajouter de nouveaux types d'objets sans avoir à modifier le code existant. Respect de la contrainte de LISKOV, Open-Closed Principle : Ouvert à l'extension et fermé à la modification.

Nous avons réfléchi à l'utilisation d'une interface plutôt que d'une classe abstraite pour la classe GildedRoseItem. Cela aurait permis à la classe Normal d'avoir à implémenter des méthodes. Mais l'utilisation d'une classe abstraite nous permet de coder directement dans la classe mère des comportements commun potentiellement à toutes les classes filles, et donc d'éviter une redondance dans les sous classes.

Nous avons néanmoins choisi de garder la classe Normal, pour clarifier la sémantique des Items.

Dans le patron Méthode, la méthode socle est crée dans la classe GildedRoseItem. C'est la méthode updateItem(Item item) qui permet de définir le comportement de base. C'est pourquoi la méthode updateItem() est déclaré final.