RAPPORT DE TP

Le travail demandé consiste à faire le refactoring du **glidedRose-Kata**. Le refactoring consiste à produire un code purement meilleur. L'objectif est de pouvoir intégrer de nouveaux éléments sans pouvoir générer des erreurs. Dans notre contexte actuel, on devrait intégrer un produit de type **Conjured.**

La première tâche a été de mettre en place des tests pour apporter plus tard des changements au code afin de limiter les risques. Ces tests ont permis d'avoir une meilleure compréhension du code. Le principal objectif de nos tests est de garantir la non régression et en ayant une meilleure couverture du code. J'ai développé plus d'une dizaine de tests (18 en réalité)pour comprendre le code du glidedRose dans les moindres compartiments. Afin de créer ces tests, je me suis approprié le document d'expression des besoins disponible dans le fichier glidedRose.

Après avoir réussi à créer le maximum de tests, j'ai eu une couverture totale de 100 % grâce à l'outil **Jacoco.** Les 100% de couverture nous assurent que toutes les lignes du code de production ont été exécutées lors des tests, guère plus. De plus, en couverture de code, seul le parcours d'une ligne est vérifié, mais pas son contenu ou son comportement. Mais c'est suffisant pour passer à la phase de refactoring proprement dite.

Avant de procéder au refactoring, j'ai initialisé un dépôt git pour assurer le contrôle de version. Ceci me permet de faire des commit à chaque fois que mes petites modifications passent avec succès.

- La première chose que j'ai eu à faire pour rendre le code lisible était de créer une méthode pour l'incrémentation et la décrémentation de la qualité d'un produit: increaseQuality(Item item) et decreaseQuality(Item item).
- Ensuite, l'idée a été de rendre plus simple les conditions complexes du GlidedRose.java. Pour ce fait, j'ai supprimé les conditions AND, NOT dans l'optique d'avoir des conditions à terme unique. Ceci a permis d'avoir une meilleure lisibilité des conditions de branchement. Grâce à cette tâche, j'ai

pu éliminer la redondance dans le code c'est-à-dire les conditions qui étaient dupliquées dans le code.

- Pour éviter une boucle for avec un compteur , j'ai privilégié le choix de for each qui choisit chaque élément d'une liste et fait un traitement sur cela.
- En analysant tout le code, j'ai remarqué que les conditions sur les noms des produits étaient au cœur de la logique de ce code. Pour cela ,j'ai décidé de procéder à une extraction de méthodes.
- J'ai ainsi créé des méthodes pour assurer la mise à jour de la qualité de chaque type de produit. j'ai constaté que le type Sulfuras était particulier puisqu'il n'y a aucun traitement sur ce type de produit.
- A chaque petite modification dans mon code, je m'assure d'exécuter l'ensemble de tous mes tests pour être sûr de n'avoir rien cassé.
- Par la suite, j'ai créé la méthode de mise à jour du produit Aged Brie (upgradeQualityAgedBrie()) , en m'assurant que les conditions sont assez simples . J'ai pareil en créant upgradeQualityBackstage(), upgradeQualityNormal().
- Après s'être assuré qu'à chaque fois mes tests passent , je supprime le bloc d'instructions relatif à la méthode que je venais d'extraire manuellement afin d'éviter d'avoir du code mort.
- Je m'assure que le sellIn soit décrémenté à chaque fois également.
- A ce niveau de mon refactoring ,j'ai un ensemble de méthodes dédiées à la mise à jour de chaque produit .
- Pour éviter l'utilisation des if else imbriqués, j'ai utilisé la structure du switch pour referencer chacun de mes cas de produits.
- Afin de le rendre plus optimal, j'ai décidé de créer une classe pour la mise à jour d'un unique produit pour faciliter la lisibilité de mon code et le rendre meilleur.

- Dans ma classe principale GlidedRose.java, je fais appel à cette classe ItemUpdate pour instancier chacun de mes items et assurer leur traitement immédiatement grâce aux méthodes que j'ai ajouté dans ma classe ItemUpdate

- Eh bien! je relance mon code et tout se passe bien. Je supprime le code mort de ma classe principale GlidedRose qui n'est d'aucune utilité, puisque ItemUpdate reprend déjà ces méthodes.

- Place maintenant à exécuter la tâche principale de ce kata, l'intégration d'un nouveau type de produit : Conjured.
- J'ai créé ma méthode pour la mise à jour d'un produit Conjured et ajouter une autre référence à mon switch pour le cas de ce produit .Tout ça dans ma classe ItemUpdate sans toucher en aucun cas ma classe principale GlidedRose.
- J'ai à présent créé pour vérifier les fonctionnalités de ce nouveau produit. Et toujours tout se passe bien.
- Ainsi , mon code a été rendu meilleur en ajoutant une classe ItemUpdate.

De tout ce qui précède , il est important de souligner certains enseignements. Au fur et à mesure que l'on crée des tests unitaires pour le refactoring, on comprend mieux le code.Après aussi , il est important de lancer des tests à la moindre modification pour s'assurer que le code marche toujours bien.Il faut également souligner que on s'arrêtera dès que le code sera lisible et facilement compréhensible et à ce juste moment, l'intégration de nouvelles fonctionnalités sera aisée.