Rapport TP

Kata : Gilded Rose Génie Logiciel Avancé

Nom : HADID Prénom : Amine Groupe 3

Introduction

Le kata "Gilded Rose" est un exercice de programmation conçu pour perfectionner les compétences des développeurs en matière de génie logiciel, de tests unitaires, de refactoring et de méthodologie TDD (Test-Driven Development). Il s'agit d'un problème de gestion de stocks de bières, où les produits ont des règles de mise à jour de la qualité et de la date de péremption spécifiques.

Pendant le processus de traitement du kata "Gilded Rose", des concepts clés ont été abordés :

Tests Unitaires : créer des tests unitaires pour valider le comportement de fonctions spécifiques dans le code. Ces tests permettent d'assurer que les modifications ultérieures n'avaient pas d'impact négatif sur le fonctionnement du code existant.

Couverture de Code : L'utilisation d'outil **JaCoCo** pour mesurer la couverture de code des tests. Atteindre une couverture de code de 100% était un objectif important, car cela garantit que toutes les parties du code sont testées et documentées.

Mutation de Code (PIT) : L'explorationde PIT (Pitest) pour détecter des erreurs potentielles dans les tests en générant des mutations de code. Cela a aidé à améliorer la qualité des tests en assurant qu'ils étaient robustes face aux modifications du code.

Refactoring : L'application des techniques de refactoring pour simplifier le code "GildedRose.java". Cela a permis d'éliminer la complexité cognitive, de regrouper des cas similaires et d'améliorer la lisibilité du code.

Test-Driven Development (TDD) : L'utilisation de la méthodologie TDD pour ajouter une nouvelle fonctionnalité au code, en écrivant d'abord les tests, puis en implémentant le code pour répondre à ces tests. Cette approche permet de développer du code de manière incrémentielle tout en maintenant la qualité.

Gestion des versions avec Git : L'utilisation de Git pour suivre les modifications du code et faciliter la collaboration, ce qui est essentiel pour la gestion d'un projet logiciel.

Partie 1 Tests

i. Tests unitaires

La création d'ensemble de tests unitaires pour le kata "Gilded Rose" afin de valider le comportement du code existant. Les tests ont été élaborés de manière à couvrir tous les cas possibles du code, garantissant ainsi un niveau élevé de fiabilité.

Les tests couvrent touts les cas possibles en utilisant JaCoCo pour verifier la couverture du code.

Nombre de Tests : 26 tests unitaires pour le code (sans les tests de mutation et les tests de la nouvelle fonction (TOTAL : 44)).

On peut avoir une couverture de 100 % de code avec un nombre peu de test. (J'ai choisi d'écrire tout)

Cas Couverts : Les tests couvrent l'ensemble des situations possibles pour les différents types de produits gérés par le programme, y compris les cas où la qualité augmente, diminue ou reste inchangée en fonction de la date de péremption.

Couverture de Code : Grâce à l'outil JaCoCo, une couverture de code de 100% signifie que l'ensemble du code source du kata "Gilded Rose" a été testé et documenté par les tests unitaires. Les lignes de code sont toutes en vert dans le rapport JaCoCo, indiquant une couverture complète.

```
    public void updateQuality() {
        for (int i = 0; i < items.length; i++) {
            if (!items[i].name.equals("Aged Brie")
                    && !items[i].name.equals("Backstage passes to a TAFKAL80ETC concert")) {
                if (items[i].quality > 0) {
                    if (!items[i].name.equals("Sulfuras, Hand of Ragnaros")) {
                        items[i].quality = items[i].quality - 1;
                    }
                }
            } else {
                if (items[i].quality < 50) {
                    items[i].quality = items[i].quality + 1;

                    if (items[i].name.equals("Backstage passes to a TAFKAL80ETC concert")) {
                        if (items[i].sellIn < 11) {
                            if (items[i].quality < 50) {
                                items[i].quality = items[i].quality + 1;
                            }
                        }

                        if (items[i].sellIn < 6) {
                            if (items[i].quality < 50) {
                                items[i].quality = items[i].quality + 1;
                            }
                        }
                    }
                }
            }

            if (!items[i].name.equals("Sulfuras, Hand of Ragnaros")) {
                items[i].sellIn = items[i].sellIn - 1;
            }

            if (items[i].sellIn < 0) {
                if (!items[i].name.equals("Aged Brie")) {
                    if (!items[i].name.equals("Backstage passes to a TAFKAL80ETC concert")) {
                        if (items[i].quality > 0) {
                            if (!items[i].name.equals("Sulfuras, Hand of Ragnaros")) {
                                items[i].quality = items[i].quality - 1;
                            }
                        }
                    } else {
                        items[i].quality = items[i].quality - items[i].quality;
                    }
                } else {
                    if (items[i].quality < 50) {
                        items[i].quality = items[i].quality + 1;
                    }
                }
```

./gradlew test et **./gradlew jacocoTestReport** pour l'exécution des tests et la génération du rapport jacoco

ii. Mutation du code

PIT (Pitest) est un outil d'analyse de mutation qui permet de tester l'efficacité de nos tests unitaires en introduisant délibérément des mutations dans le code source. Ces mutations consistent à apporter de petites modifications au code, telles que des inversions de conditions, des suppressions de lignes, etc., pour évaluer si nos tests sont capables de détecter ces mutations et signaler les erreurs potentielles.

Utilisation de PIT : Pour tester nos tests unitaires contre des mutations de code, on utilise l'outil PIT de la manière suivante :

a) Configuration PIT dans le projet pour qu'il puisse générer des mutations dans le code source.
b) L'exécution PIT en utilisant la commande **./gradlew pitest.**

Résultats de PIT : Les résultats de l'analyse de mutation effectuée par PIT ont révélé les informations suivantes :

a) **Nombre de Mutations Générées** : PIT a généré un total de 37 mutations dans le code source. Ces mutations ont inclus des modifications telles que l'inversion de conditions, la modification des opérateurs de comparaison, etc.
b) **Mutations Tuées** : Mes tests unitaires ont réussi à "tuer" 32 mutations. Cela signifie que les tests ont détecté ces mutations et ont signalé des erreurs, confirmant ainsi l'efficacité des tests.
c) **Exemples de Mutations** : Voici quelques exemples de mutations que PIT a générées, avec les modifications apportées au code source :
   Inversion de Condition : Un if d'origine, tel que if (item.quality > 50), a été modifié en if (item.quality >= 50). Cela met en évidence l'importance d'avoir des tests pour détecter ces petites variations dans les conditions.
   Modification d'Opérateur : Une expression originale, par exemple, item.sellIn < 0, a été modifiée en item.sellIn > 0. Ces modifications montrent comment les tests doivent être capables de détecter des erreurs même avec des opérations modifiées.

   Atteindre un taux de 100% de mutations tuées peut être difficile, car cela signifie que vos tests détectent toutes les mutations possibles. Dans mon cas, j'ai dû ajouter des tests unitaires supplémentaires pour couvrir certaines mutations spécifiques et atteindre un taux de mutations tuées élevé.

Partie 2 Refactoring

Refactoring de "GildedRose.java"

Principes de Refactoring : Le refactoring consiste à réorganiser le code existant pour améliorer sa lisibilité, sa maintenabilité et sa performance, sans changer son comportement. Pour simplifier "GildedRose.java", plusieurs principes de refactoring ont été appliqués, notamment :

Élimination de la Complexité Cognitive : réduire la complexité cognitive en regroupant des cas similaires, en éliminant des imbrications excessives de conditions (if et else), et en réduisant le nombre de branches de décision.

Utilisation d'une structure plus adaptée (SWITCH CASE) que des if/else .

Mutualisation de Code : Identification des portions de code similaires et les regroupées pour éviter la duplication, ce qui améliore la maintenabilité.

Focus sur les Cas Généraux : La modification d'approche du code pour se concentrer sur les cas généraux plutôt que sur des cas spécifiques, ce qui simplifie le traitement des produits.

Commits Git du Refactoring : Pendant le processus de refactoring, j'ai effectué plusieurs commits Git pour montrer ma progression.

A chaque changement du code j'effectue un test pour voir si le changement n'a pas affecté le comportement du code.

Quelques étapes du refactoring

Création de constantes pour les noms des bières pour rendre le code plus lisible.

```
static final String SULFURAS = "Sulfuras, Hand of Ragnaros";
```

Creation des fonctions de calcul de « qualité » et « Sellin »

```
private void decreaseQuality(Item item) {
  if (item.quality > 0 && !item.name.equals("Sulfuras, Hand of Ragnaros")) {
            item.quality--;
        }
    }
 private void decreaseSellIn(Item item) {
        if (!item.name.equals(SULFURAS)) {
            item.sellIn--;
        }
    }
private void increaseQuality(Item item) {
        if (item.quality < 50) {
            item.quality++;
        }
    }
```

Switch case : par rapport au nom de l'objet

```
switch (item.name) {
                case AGED_BRIE:
                    updateAgedBrie();
                    break;
                case SULFURAS:
                   // updateSulfuras();
                    break;
                case BACKSTAGE_PASSES:
                    updateBackstagePasses();
                    break;
                case CONJURED:
                    updateConjured();
                    break;
                default:
                    updateDefault();
                    break;
            }
```

Partie 3 Ajout de Fonctionnalité updateConjured

Apres avoir refactorisé le code et qu'il est fonctionnelle, on ajoute une nouvelle marque de bière « Conjured » et pour cela on le fait avec la technique de TDD (Test-Driven Development) qui consiste à coder les tests en premier après le code de la fonction, elle sert à effectuer des modifications tout en s'assurant la qualité du code, et sa minimalité.

La rédaction des tests de Conjured

Y' en a deux tests a rajouter

- SellIn >= 0, la qualité de la bière Conjured diminue de 2
- SellIn <0, La qualité de la bière Conjured diminue de 4

L'ajout de Conjured au code

1. Ajouter une case de Conjured a la fonction updateQuality
2. Créer la fonction update Conjured qui fait diminuer la qualité sous les conditions

```
case CONJURED:
     updateConjured();
     break;
```

```
private void updateConjured() {

       for (int i = 0; i < items.length; i++) {

           if (items[i].sellIn < 0) {
               decreaseQuality(items[i]);
```

```
                decreaseQuality(items[i]);
                decreaseQuality(items[i]);
                decreaseQuality(items[i]);

            }
            else{
                decreaseQuality(items[i]);
                decreaseQuality(items[i]);

            }
             decreaseSellIn(items[i]);

        }
    }
```

Conclusion

Le kata "Gilded Rose" a renforcé la compréhension des bonnes pratiques de développement logiciel, de la qualité du code et de l'importance des tests unitaires. Il a également démontré l'efficacité de PIT pour détecter des erreurs potentielles et du refactoring pour simplifier un code complexe. Ces compétences sont essentielles pour tout développeur soucieux de produire un code de qualité et maintenable. Le refactoring a permis de simplifier considérablement "GildedRose.java" tout en maintenant son comportement d'origine. Ce kata a été une expérience enrichissante, et les connaissances acquises seront appliquées dans les futurs projets de développement.