CLERMONT AUVERGNE INP Isima

Kata : Gilded Rose

Génie Logiciel Avancé

Groupe 03

NOM	PRENOM
BAHOU	Yassine
HATMI	Ayoub

Tuteur: FORESTIER FLORIAN

Github : TP-Refactoring

Contents

1 | Introduction

Un projet Kata est un exercice de pratique, souvent sous la forme d'un défi ou d'un problème de programmation, conçu pour aider les développeurs à améliorer leurs compétences, à apprendre les meilleures pratiques et à renforcer les concepts fondamentaux dans un domaine spécifique, tel que le développement logiciel. Le but d'un projet Kata est de fournir une expérience pratique permettant aux développeurs d'appliquer et de perfectionner leurs connaissances dans un contexte concret.

Le Kata "Gilded Rose" est un projet Kata bien connu dans le développement logiciel. Il simule un système logiciel de gestion des stocks d'une bière fictive, mettant au défi les développeurs d'appliquer divers concepts d'ingénierie logicielle et les meilleures pratiques.

Dans ce rapport, nous mettrons l'accent sur le Kata "Gilded Rose" et son importance dans le développement logiciel. Ce Kata est un outil d'apprentissage précieux, car il englobe des concepts et des techniques essentiels à la réalisation de projets logiciels dans le monde réel. Le rapport couvrira les concepts clés suivants :

- **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.
- **Tests unitaires :** Rédaction et implémentation de tests unitaires pour vérifier la fonctionnalité correcte du logiciel et assurer sa fiabilité.
- **Analyse de la couverture du code avec JaCoCo :** Évaluation de la couverture des tests unitaires pour identifier les zones du code non testées et améliorer la qualité des tests.
- **Tests de mutation du code avec PIT :** La pratique d'introduire des mutations dans le code pour évaluer l'efficacité et la robustesse des tests unitaires.
- **Refactoring :**Le processus de restructuration et de simplification du code pour améliorer la lisibilité, la maintenabilité et les performances.
- **Développement piloté par les tests (TDD) :** La méthodologie consistant à rédiger des tests avant d'écrire du code pour guider le processus de développement et s'assurer que le logiciel répond aux exigences spécifiées.

Ces concepts représentent collectivement des aspects essentiels du développement logiciel et seront détaillés dans le rapport, mettant en évidence leur pertinence dans le contexte du Kata "Gilded Rose" et leur applicabilité plus large dans les projets logiciels du monde réel.

2 | Installation

Nous avons installé plusieurs outils essentiels pour notre projet, notamment Java, Gradle, JaCoCo, Git, et Pitest. Cependant, l'un des défis auxquels nous avons été confrontés a été le problème de compatibilité. Plus précisément, nous avons constaté que les versions de Gradle et JaCoCo que nous avions choisies n'étaient pas parfaitement compatibles avec la version de JDK que nous utilisions. Cela a nécessité des efforts supplémentaires pour harmoniser ces composants, mais une fois que nous avons résolu ce problème, notre environnement de développement a fonctionné de manière fluide, nous permettant de progresser dans notre projet avec succès.

3 | Tests

3.1 | Tests unitaires

Notre processus a débuté par la lecture attentive du document Markdown afin de bien comprendre le contexte du projet. À partir de là, nous avons déduit quelques tests unitaires, en cherchant à cerner les fonctionnalités clés du code. Cependant, le document s'est avéré insuffisamment spécifique, ce qui nous a contraints à nous appuyer sur le code préalablement écrit. Cette dépendance au code existant ne s'est pas avérée aisée, car sa complexité nous a demandé une attention particulière lors de l'écriture des tests unitaires.

Pour garantir la qualité et la fiabilité de notre code, nous avons poursuivi en créant méticuleusement les tests un par un, en nous assurant de couvrir toutes les fonctionnalités importantes. Dans ce processus, nous avons utilisé JaCoCo pour vérifier la couverture de code. L'objectif ultime était d'atteindre une couverture de 100%, une norme rigoureuse qui présente de nombreux avantages. Cette approche vise à assurer une fiabilité accrue de l'application, une facilité de maintenance, une réduction des erreurs potentielles et une meilleure compréhension du code. En somme, notre démarche visait à produire un logiciel de qualité, solide et durable, essentiel pour notre projet. Pour exécuter les tests et générer un rapport de couverture JaCoCo, nous avons utilisé les commandes Gradle suivantes :

- **Pour exécuter les tests :** ./gradlew test
- **Pour générer un rapport de couverture JaCoCo :** ./gradlew jacocoTestReport

Cette méthodologie a permis d'assurer un développement professionnel et une meilleure maîtrise de notre projet, tout en garantissant un code fiable et de haute qualité.

3.2 | Mutation de code

La mutation de code est une technique puissante qui vise à évaluer la robustesse de nos tests unitaires en introduisant délibérément de petites modifications, ou "mutations", dans le code source. L'objectif de cette technique est de détecter les failles potentielles dans nos tests en vérifiant s'ils sont capables de détecter ces mutations. Si un test parvient à repérer une mutation, cela signifie qu'il est solide et qu'il identifie correctement les erreurs. En revanche, si une mutation passe inaperçue, cela suggère que le test ne couvre pas correctement le code, ce qui révèle une faiblesse dans la suite de tests. Pour automatiser ce processus on a utilisé PIT (Mutation Testing Tool).PIT est un outil de test de mutation qui génère automatiquement une multitude de versions mutées de votre code source. Il exécute ensuite vos tests unitaires sur chacune de ces versions mutées pour évaluer la qualité de vos tests. L'objectif est de détecter tous les types de mutations possibles grâce à vos tests.Pour générer un rapport PIT On a utilisé la commande :

- **./gradlew pitest**

Le rapport PIT est généré dans le répertoire du projet. Grâce à l'analyse de cet outil, nous avons pu mettre en évidence plusieurs insuffisances au sein de nos tests. Ces constats nous ont motivés à entreprendre l'ajout de tests supplémentaires, afin d'atteindre une couverture complète de 100 %. Cette démarche vise à renforcer la qualité et la fiabilité de notre code en identifiant et en comblant les lacunes dans notre suite de tests existante.

4 | Refactoring

4.1 | Les étapes du Refactoring

Le code existant présente une complexité significative avec des structures de contrôle imbriquées et des conditions multiples. Cette complexité rend la maintenance, la compréhension et la modification du code difficiles. Les nombreuses vérifications et conditions imbriquées augmentent le risque d'erreurs et compliquent la garantie de la qualité du code. Il est impératif de simplifier et de clarifier ce code en le refactorant. Cette démarche permettra d'améliorer la lisibilité, de réduire la duplication de code et d'augmenter la facilité de maintenance, tout en réduisant les chances d'introduire de nouveaux bugs. Nous avons commencé notre processus de refactoring en effectuant de petites modifications et en testant la fonction pour vérifier son bon fonctionnement à l'aide des tests unitaires précédemment codés. Ensuite, nous les avons commités dans Git pour conserver une trace de notre code au cas où des problèmes surviendraient plus tard.

Pour simplifier le code, voici les étapes de refactoring que nous avons suivies :

- Création de constantes pour les noms des bières et les valeurs maximales et minimales de la qualité, pour rendre le code plus lisible.

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

- Traitement séparé du cas de "sulfuraas", car sa qualité ne change pas.

```
if (items[i].name.equals(SULFURAS)){
    continue;
  }
```

- Regroupement de certaines déclarations "if" pour réduire l'imbrication du code.
- Tentative de réécriture des conditions "if" en méthodes renvoyant des booléens. Cependant, nous avons rapidement réalisé que cela constituait une sur-optimisation, et nous sommes revenus à la version précédente dans Git.

```
    public boolean isAgedBrie() {
    return this.name == AGED_BRIE;
  }
```

Après une analyse approfondie, nous avons identifié des opportunités pour remplacer certaines des conditions "if" par des fonctions "max" ou "min" afin de réduire l'imbrication.

```
// Avant Refactoring
if (items[i].sellIn < 11) {
        if (items[i].quality < 50) {
                items[i].quality = items[i].quality + 1;
        }
}

// Apres Refactoring
if (this.sellIn >= 11) {
          this.quality = Math.min(this.quality + 1, MAXIMUM_QUALITY);
}
```

Nous avons également isolé les conditions en fonction du nom de la bière. À ce stade, nous avons dû prendre une décision :

- **Polymorphisme :** Définir le traitement en fonction de la classe de l'objet(Ce qui aurait nécessité la création d'une classe pour chaque type de bière.)
- **Switch case :** Définir le traitement en fonction du nom de l'objet.

4.2 | Choix à faire

Nous avons opté pour l'utilisation de la structure "switch case" car autrement, nous aurions dû modifier l'ensemble des tests et la structure du programme. Étant donné que nous avons un nombre limité de types de bières, cette approche s'est avérée plus optimale. De plus, maintenant que le code est refactoré, nous pouvons toujours revenir à l'utilisation du polymorphisme si notre code venait à s'étendre.

On peut remarquer qu'il est encore possible d'identifier des sections de code répétitif, notamment en ce qui concerne la diminution de la variable "sellin." Il serait envisageable de réduire la taille du code en supprimant ces répétitions, cependant, cette action pourrait affecter la lisibilité du code et Il sera essentiel de préciser le traitement préalable et postérieur de la variable "sellin."

Malgré cela, étant donné la taille actuelle de nos méthodes et l'état actuel de notre code, il semble qu'il s'agisse d'un point de refactoring satisfaisant. Si des changements significatifs surviennent ultérieurement, nous pourrons nous adapter, car, en fin de compte, l'objectif fondamental de la méthodologie agile n'est pas de prédire les problèmes, mais plutôt de disposer de la flexibilité nécessaire pour faire face à toute modification inattendue.

5 | Ajout de fonctionnalité

Maintenant que notre code est propre, refactorisé, et testé à 100%, nous avons été confrontés à un nouveau défi passionnant : l'intégration d'une nouvelle marque de bière, appelée "Conjured". Cette marque de bière a des caractéristiques spéciales, car la qualité de ses produits diminue deux fois plus rapidement que les autres bières. En d'autres termes, la qualité diminue de 2 points par jour lorsque sellIn est supérieur ou égal à zéro, et de 4 points par jour lorsque sellIn est inférieur à zéro.

Pour répondre à ce besoin, nous avons choisi d'appliquer la méthodologie TDD (Test-Driven Development). Cette approche est largement reconnue dans le développement logiciel pour garantir la qualité du code et sa minimalité. Elle repose sur le principe de créer d'abord les tests, puis de coder le comportement en conséquence.

Étape 1 - Création des Tests (Commit 1)

Dans cette première étape, nous avons commencé par coder les tests qui décrivent le comportement attendu pour la nouvelle marque de bière "Conjured". Conformément aux spécifications, deux scénarios sont à prendre en compte :

- Lorsque **sellIn** est supérieur ou égal à zéro, la qualité de la bière "Conjured" doit diminuer de 2 points par jour.
- Lorsque **sellIn** est inférieur à zéro, la qualité de la bière "Conjured" doit diminuer de 4 points par jour.

Étape 2 - Implémentation du Code (Commit 2)

Une fois les tests écrits, nous avons procédé à l'implémentation du code nécessaire pour que la classe gère correctement la nouvelle marque de bière "Conjured". Cette étape a impliqué la modification du comportement de la classe en fonction des spécifications précédemment définies.

Nous avons pris soin de maintenir le code propre et de respecter les principes de refactoring que nous avons appliqués précédemment. De plus, les tests existants ont continué à garantir que les autres marques de bière fonctionnaient comme prévu.

Le résultat final est un code maintenable, testé à 100%, qui inclut désormais le traitement spécifique de la marque "Conjured". Cette approche a permis d'ajouter cette nouvelle fonctionnalité tout en minimisant les risques d'introduire des bugs ou des problèmes dans le code existant.

```
    private void updateQualityForConjured() {
    if (this.sellIn > 0) {
        this.quality -= 2;
    } else {
        this.quality -= 4;
    }
}
```

6 | Conclusion

En conclusion, ce rapport de TP de refactoring portant sur la kata GildedRose a mis en lumière une réalité fondamentale du processus de refactoring : sa nature subjective et étroitement dépendante du contexte d'utilisation. Les différentes approches et choix de conception effectués au cours de ce projet ont clairement démontré que la meilleure manière de refactorer le code peut varier en fonction des spécificités du cas d'utilisation, des contraintes techniques, et des objectifs de performance.

Cette expérience a également souligné l'importance de l'analyse préalable et de la compréhension approfondie du code source avant d'entreprendre un refactoring. Les décisions prises doivent être guidées par une vision claire des besoins actuels et futurs du système, ainsi que par des considérations de maintenabilité, de lisibilité et de performance.

En fin de compte, le refactoring est un art autant qu'une science. Il nécessite une expertise technique, une réflexion critique et la capacité à faire des compromis judicieux pour aboutir à une solution qui répond au mieux aux exigences spécifiques de chaque projet. Il est essentiel de reconnaître que les choix de refactoring peuvent varier en fonction des circonstances, et que l'objectif ultime doit toujours être l'amélioration de la qualité du code et de la facilité de maintenance.