Refaktoryzacja Gilded Rose Refactoring Kata - zaawansowane języki programowania 2019/2020

Iwona Jaśtak, nr 246849 - Github

1. Opis problemu

Sprawozdanie dotyczy procesu refaktoryzacji kodu GildedRose Kata. Kod z początkowym programem znajduje się w repozytorium Emily Bache https://github.com/emilybache/GildedRose-Refactoring-Kata. Program musi spełniać pewne założenia (dokładnie opisane tutaj http://iamnotmyself.com/2011/02/14/refactor-this-the-gilded-rose-kata/):

- każdy przedmiot na wartość SellIn, czyli liczbę dni, w których trzeba sprzedać przedmiot,
- każdy przedmiot na wartość Quality, czyli jak wartościowy jest przedmiot,
- dla każdego przedmiotu obie te wartości zmniejszają się pod koniec każdego dnia.

Dodatkowo sformułowane są zasady dla niektórych przedmiotów:

- gdy SellIn osiągnie wartość 0, to Quality zmniejsza się dwa razy szybciej,
- Quality nigdy nie jest liczbą ujemną,
- Quality przedmiotu 'Aged Brie' zwiększa się z każdym dniem zamiast zmniejszać się,
- Quality nigdy nie jest większe niż 50,
- Quality i SellIn przedmiotu 'Sulfuras' nigdy się nie zmieniają,
- przedmiot 'Backstage passes' tak jak 'Aged Brie' zwiększa Quality, dodatkowo zwiększa je o dwa jeśli zostało mniej niż 10 dni do terminu sprzesaży, o trzy jeśli zostało mniej niż 5 dni, ale spada do zero jeśli minie termin,
- przedmiot 'Conjured' traci Quality dwa razy szybciej niż normalny przedmiot.

Moją refaktoryzajcę GildedRose-Refactoring-Kata wykonałam w obiektowym języku Java.

2. Testy

Pierwszym krokiem jaki wykonałam było napisanie testów do każdego rodzaju przedmiotu. Wykorzystałam do tego bibliotekę JUnit5. By uruchomić testy należy w folderze projektu wykonać komendę: `mvn test`.
Testy pomogły mi podczas refaktoryzacji sprawdzać, czy moje modyfikacje nie zmieniły działania początkowego programu. Dodatkowo połączyłam moje repozytorium z serwisem ciągłej integracji Travis CI, który uruchamia testy po każdym commicie przesłanym do zdalnego repozytorium.

Na poniższym zrzucie ekranu zaprezentowany jest rezultat testów. Jak widać wszystkie testy przeszły pozytywnie.

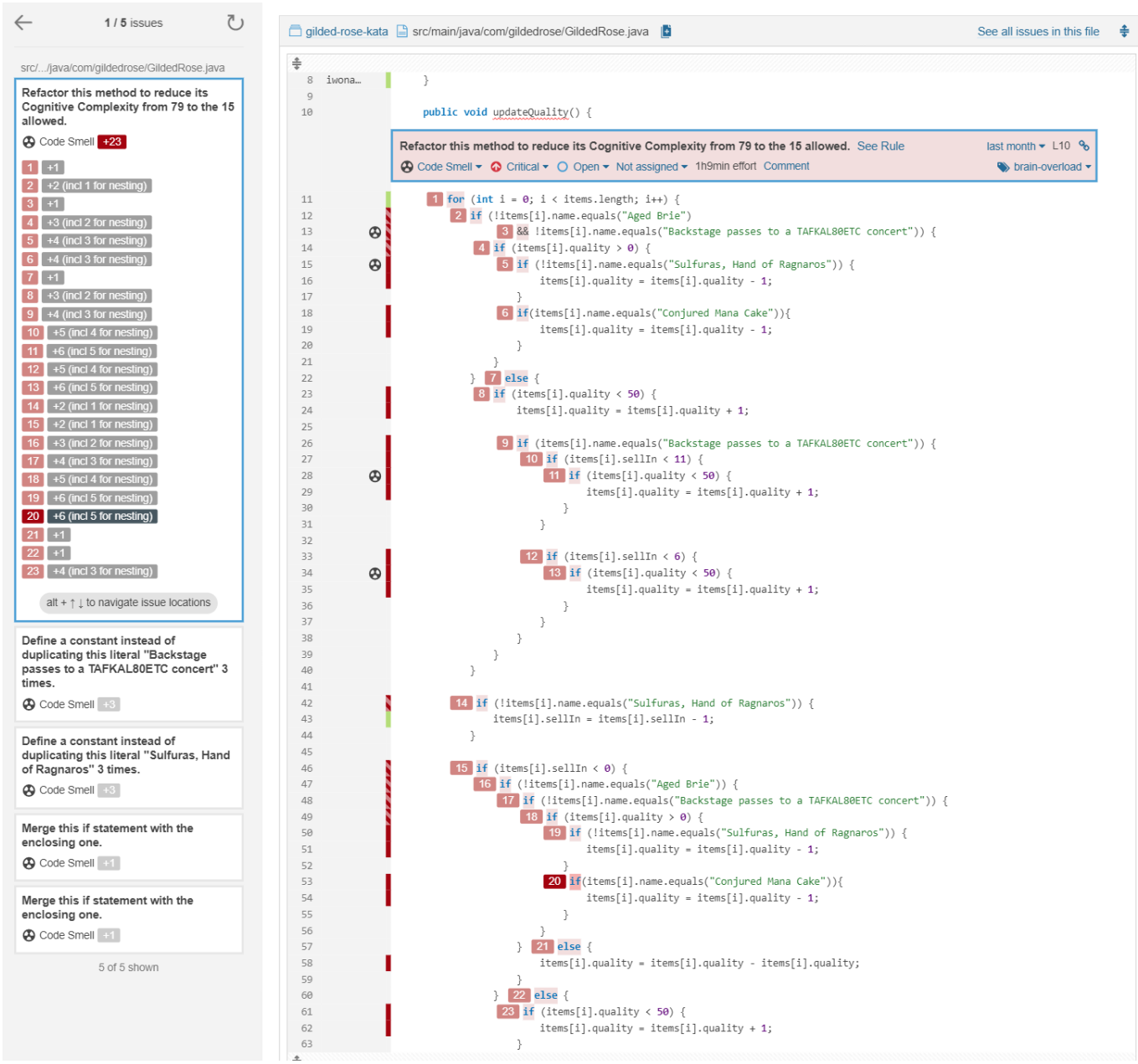
```
-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running com.gildedrose.AgedBrieTest
Tests run: 7, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.076 s - in com.gildedrose.AgedBrieTest
Running com.gildedrose.BackstagePassTest
Tests run: 9, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.003 s - in com.gildedrose.BackstagePassTest
Running com.gildedrose.ConjuredItemTest
Tests run: 7, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.003 s - in com.gildedrose.ConjuredItemTest
Running com.gildedrose.DefaultItemTest
Tests run: 7, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.008 s - in com.gildedrose.DefaultItemTest
Running com.gildedrose.SulfurasTest
Tests run: 3, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.004 s - in com.gildedrose.SulfurasTest

Results:

Tests run: 33, Failures: 0, Errors: 0, Skipped: 0
```

3. Metryki

Do analizy kodu wykorzystałam narzędzie Sonarcloud, które między innymi wylicza złożoność programu, dług technologiczny oraz wykrywa code smelle.

Wyniki analizy z Sonarcloud przed refaktoryzacją:

Pierwszym problemem, który najbardziej rzuca się w oczy, jest zbyt wysoki dług technologiczny jak na tak mały program składający się z dwóch klas. Następnym problemem są code smelle, czyli cechy, że kod jest napisany w zły sposób, pokazane poniżej:



Na powyższym zrzucie ekranu możemy zauważyć, że jednym z code smelli jest zbyt duża złożoność poznawcza dla metody `updateQuality()` w klasie `GildedRose`. Jest na nim również przedstawione w jaki sposób jest to wyliczane.

4. Refaktoryzacja

Celem mojej refaktoryzacji było pozbycie się zagnieżdżonych ifów oraz zastąpienie ich polimorfizmem - metoda refaktoryzacji **Replace Conditional with Polymorphism**. Wykonałam to w następujących krokach:

1. Utworzenie klasy `DefaultItem` dziedziczącej po `Item` oraz zdefiniowanie działania zwykłego przedmiotu. Skorzystałam tutaj również z metody refaktoryzacji **Replace Magic Number with Symbolic Constant** wydzielając zmienne statyczne `MAX_VALUE` oraz `MIN_VALUE`.

```
public class DefaultItem extends Item {
    private static final int MAX_VALUE = 50;
    private static final int MIN_VALUE = 0;

    public DefaultItem(String name, int sellIn, int quality) {
        super(name, sellIn, quality);
    }

    public void updateQuality() {
        decreaseSellIn();
        if(quality < MAX_VALUE && quality >= MIN_VALUE)
            changeQuality();
        if(sellIn < 0)
            changeQuality();

        checkQualityRange();
    }

    protected void checkQualityRange(){
        if(quality < MIN_VALUE)
            quality = MIN_VALUE;
        else if(quality >= MAX_VALUE)
            quality =  MAX_VALUE;
    }

    protected void decreaseSellIn(){
        sellIn--;
    }

    protected void changeQuality(){
        quality--;
    }
}
```

2. Usunięcie powtarzających się fragmentów

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

oraz wydzielenie ich do osobnej klasy `Sulfuras` dziedziczącej po `DefaultItem` z nadpisaną metodą `updateQuality()` - skorzystanie z metody refaktoryzacji **Extract Class**.

```
@Override
public void updateQuality(){
  //nothing changes
}
```

3. Usunięcie fragmentu

```
if(items[i].name.equals('Conjured Mana Cake')){
    items[i].quality--;
}
```

oraz wydzielenie go do klasy `ConjuredItem` dziedziczącej po `DefaultItem` z nadpisaną metodą `changeQuality()` - **Extract Class**.

```
@Override
public void changeQuality(){
    quality -= 2;
    checkQualityRange();
}
```

4. Usunięcie fragmentów

```
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;
        }
    }
}
```

oraz

```
if (!items[i].name.equals('Backstage passes to a TAFKAL80ETC concert')) {
} else {
    items[i].quality = items[i].quality - items[i].quality;
}
```

oraz wydzielenie go do klasy `BackstagePass` dziedziączącej po `DefaultItem` z nadpisaną metodą `changeQuality()` - **Extract Class**.

```
@Override
public void changeQuality(){
    if(sellIn <= 0)
        quality = 0;
    else if(sellIn <= 5)
        quality += 3;
    else if(sellIn <= 10)
        quality += 2;
```

```
    else
        quality++;
    checkQualityRange();
}
```

5. Usunięcie pozostałej części

```
if (!items[i].name.equals('Aged Brie')
       && !items[i].name.equals('Backstage passes to a TAFKAL80ETC concert')) {
    } else {
        if (items[i].quality < 50) {
            items[i].quality = items[i].quality + 1;
        }
    }

    if (items[i].sellIn < 0) {
        if (!items[i].name.equals('Aged Brie')) {
        } else {
            if (items[i].quality < 50) {
                items[i].quality = items[i].quality + 1;
            }
        }
    }
```

oraz wydzielenie jej do osobnej klasy `AgedBrie` dziedziczącej po `DefaultItem` z nadpisaną metodą `changeQuality()` - **Extract Class**.

```
@Override
public void changeQuality(){
    quality++;
    checkQualityRange();
}
```

6. Zmiana pętli w `updateQuality()` w klasie `GildedRose`.

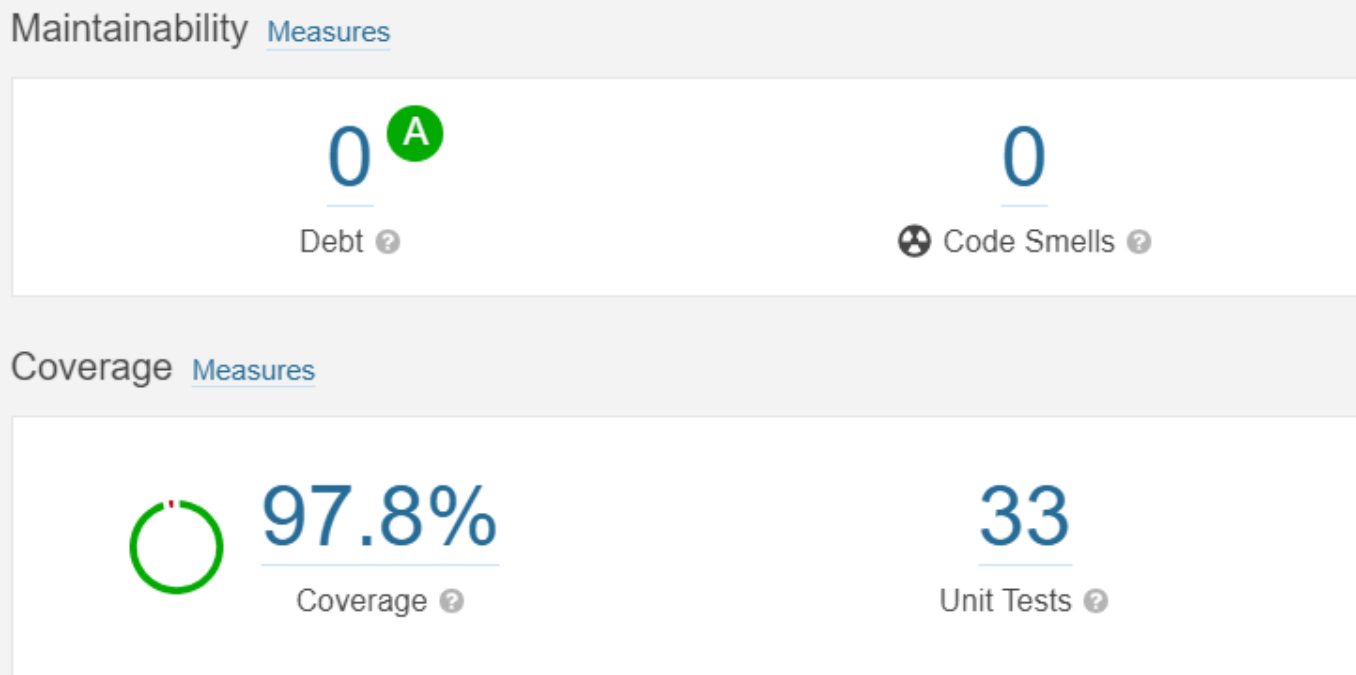
```
public void updateQuality() {
    for (DefaultItem item : items) {
         item.updateQuality();
    }
}
```

7. Skorzystanie z Factory method pattern oraz stworzenie fabryki produkującej przedmioty - metoda refaktoryzacji **Replace Constructor with Factory Method**:

```
public class ItemFactory {
    public DefaultItem createItem(String type, int sellIn, int quality){
        switch(type){
            case 'Sulfuras, Hand of Ragnaros':
                return new Sulfuras(type, sellIn, quality);
            case 'Conjured Mana Cake':
                return new ConjuredItem(type, sellIn, quality);
            case 'Aged Brie':
                return new AgedBrie(type, sellIn, quality);
            case 'Backstage passes to a TAFKAL80ETC concert':
                return new BackstagePass(type, sellIn, quality);
            default:
                return new DefaultItem(type, sellIn, quality);
        }
    }
}
```

5. Po refaktoryzacji

Wyniki z Sonarclouda po refaktoryzacji:



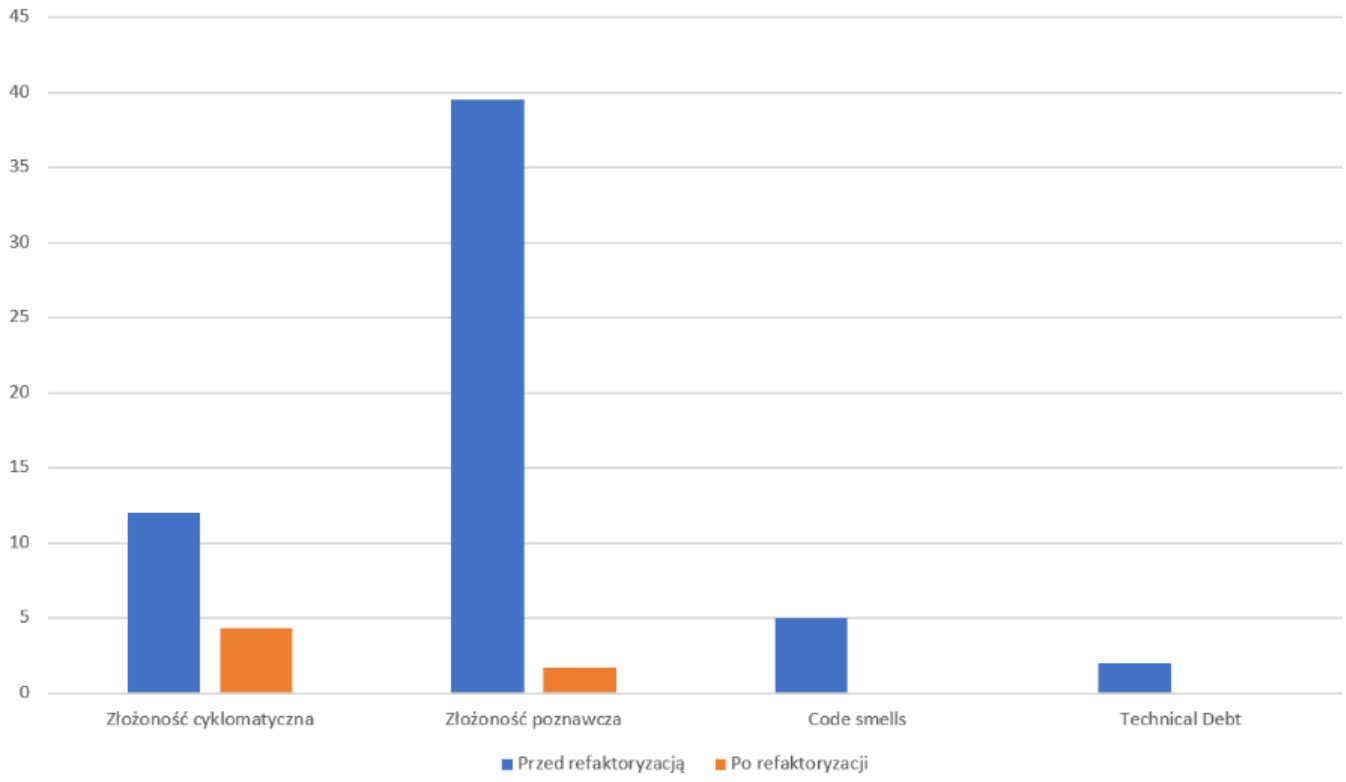
Jak widać na załączonym zrzucie ograniczyłam dług technologiczny do zera oraz pozbyłam się wszystkich code smelli, a moje testy pokryły prawie 98% kodu.

Złożoność poznawcza moich klas jest również zdecydowanie niższa:

Cognitive Complexity **12**

	New code: last 30 days
src/main/java/com/gildedrose/item/DefaultItem.java	5
src/main/java/com/gildedrose/item/BackstagePass.java	4
src/main/java/com/gildedrose/GildedRose.java	2
src/main/java/com/gildedrose/item/ItemFactory.java	1
src/main/java/com/gildedrose/item/AgedBrie.java	0
src/test/java/com/gildedrose/AgedBrieTest.java	0
src/test/java/com/gildedrose/BackstagePassTest.java	0
src/main/java/com/gildedrose/item/ConjuredItem.java	0
src/test/java/com/gildedrose/ConjuredItemTest.java	0
src/test/java/com/gildedrose/DefaultItemTest.java	0
src/main/java/com/gildedrose/Item.java	0
pom.xml	0
src/main/java/com/gildedrose/item/Sulfuras.java	0
src/test/java/com/gildedrose/SulfurasTest.java	0
src/test/java/com/gildedrose/TexttestFixture.java	0

Żeby rezultaty były bardziej czytelne przedstawiłam je również na poniższym wykresie słupkowym.



Niebieskim kolorem zaznaczone są wyniki przed refaktoryzacją, natomiast pomarańczowym po. Jako pierwsze porównuje złożoności cyklomatyczne (ang. Cyclomatic Complexity) - jest to wartość wyliczana na podstawie liczby niezależnych ścieżek w programie. Drugą porównywaną wartością jest złożoność poznawcza (ang. Cognitive Complexity) - jest to miara określająca jak kod jest trudny do czytania i zrozumienia. Obie wartości wyliczone są jako średnia na klasę.

Kolejno podane są wartości liczby code smelli oraz długu technologicznego w godzinach. Im mniejsze są wszystkie z wymienionych wartości, tym lepiej. Z wykresu można odczytać zdecydowaną ich poprawę. Po przeanalizowaniu wykresu można stwierdzić, że refaktoryzajca poprawiła znacząco jakość kodu oraz sprawiła, że dalsza praca nad nim będzie łatwiejsza oraz przyjemniejsza.