

The Gilded Rose Refactoring Kata

Po co refaktoryzacja?

Jak mówi Martin Fowler: Refaktoryzacja to zmiana wprowadzona w wewnętrznej strukturze oprogramowania, aby ułatwić jego zrozumienie i szybszą modyfikację bez zmiany istniejącego działania.

Platforma *Code Climate* wykazała, iż kod bazowy Gilded Rose zawierał liczne code smells, takie jak zbyt długie metody, szereg duplikacji oraz splątana logika warunkowa. Te elementy sprawiały, iż kod był trudny do zrozumienia i wymagał refaktoryzacji, której dokonałam w języku obiektowym Java.

<https://github.com/dkwitt/gilded-rose>

Kroki

Pierwszą rzeczą, którą zrobiłam przed podjęciem próby refaktoryzacji tego kodu było napisanie testów, by upewnić się, iż wprowadzane zmiany nie wpłyną negatywnie na działanie programu i wszystko będzie działać poprawnie. W tym celu posłużyłam się biblioteką *Approval Tests*. Testy pokryły 100% linii kodu, co utworzyło dobrą bazę do późniejszej refaktoryzacji. Dodatkowo poddałam swój zestaw testów manualnemu testowi mutacji – zmieniając drobny element kodu uruchamiałam testy ponownie by sprawdzić, czy test wyłapie błędy, które wprowadziłam.

Następnie dokonałam ekstrakcji metody `doUpdateQuality`, umieszczając w niej warunek pętli `for` dla metody `UpdateQuality()`. Usunęłam fragmenty martwego kodu, wykorzystałam również wbudowane funkcje refaktoryzacji w programie IntelliJ, takie jak odwracanie konstrukcji `if`, aby zmniejszyć zagnieżdżanie. Po dokonaniu szeregu zmian powstał całkiem dobrze skonstruowany, długi warunek, który pozwolił mi zgrupować wszystkie instrukcje:

```

private void doUpdateQuality(Item item) {
    switch (item.name) {
        case "Aged Brie":
            if (item.quality < 50) {
                item.quality = item.quality + 1;
            }

            item.sellIn = item.sellIn - 1;

            if (item.sellIn < 0) {
                if (item.quality < 50) {
                    item.quality = item.quality + 1;
                }
            }
            break;
        case "Backstage passes to a TAFKAL80ETC concert":
            if (item.quality < 50) {
                item.quality = item.quality + 1;

                if (item.sellIn < 11) {
                    if (item.quality < 50) {
                        item.quality = item.quality + 1;
                    }
                }

                if (item.sellIn < 6) {
                    if (item.quality < 50) {
                        item.quality = item.quality + 1;
                    }
                }
            }

            item.sellIn = item.sellIn - 1;

            if (item.sellIn < 0) {
                item.quality = 0;
            }
            break;
        case "Sulfuras, Hand of Ragnaros":
            break;
        default:
            if (item.quality > 0) {
                item.quality = item.quality - 1;
            }

            item.sellIn = item.sellIn - 1;

            if (item.sellIn < 0) {
                if (item.quality > 0) {
                    item.quality = item.quality - 1;
                }
            }
            break;
    }
}

```





```
}  
}
```

Utworzyłam również podklasy dla każdego przedmiotu, w których umieściłam części logiki warunkujące działanie poszczególnych przedmiotów. Zastąpiłam instrukcję warunkową polimorfizmem oraz dodałam poprawnie działający fragment Conjured Mana Cake, który wygląda następująco:

```
public class ConjuredManaCakeItem extends Item {  
    ConjuredManaCakeItem(int sellIn, int quality) {  
        super("Conjured Mana Cake", sellIn, quality);  
    }  
  
    @Override  
    protected void doUpdateQuality() {  
  
        sellIn = sellIn - 1;  
  
        if (sellIn > 0) {  
            if (quality >= 2) {  
                quality -= 2;  
            } else {  
                quality = 0;  
            }  
        } else {  
            if (quality > 4) {  
                quality -= 4;  
            } else {  
                quality = 0;  
            }  
        }  
    }  
}
```

Ocena kodu po refaktoryzacji

Kod stał się bardziej czytelny, metoda `doUpdateQuality` ma złożoność poznawczą równą 8 a łatwość utrzymania kodu znacznie się poprawiła, co obrazuje tabelka poniżej:

NAME ^	LINES OF CODE	MAINTAINABILITY
 src/main/java/com/gildedrose/AgedBriarItem.java	18	 1 hr
 src/main/java/com/gildedrose/BackstageItem.java	26	 1 hr
 src/main/java/com/gildedrose/ConjuredManaCakeItem.java	23	 45 mins
 src/main/java/com/gildedrose/GildedRose.java	12	 0 mins
 src/main/java/com/gildedrose/Item.java	40	 1 hr
 src/main/java/com/gildedrose/SulfurasItem.java	9	 0 mins