Gilded Rose

1. Refaktoryzacja kodu

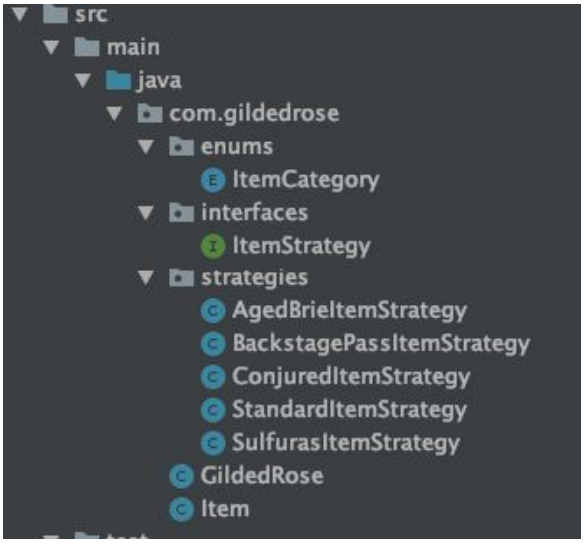
Przed rozpoczęciem pracy nad kodem należy przeprowadzić analizę potrzeb biznesowych aplikacji.
Sklep ma kilka produktów, każdy z nich ma atrybuty:

- SellIn - ilość dni pozostałych na sprzedaż produktu
- Quality - określa jak wartościowy jest produkt (min. 0, max. 50)

Produkty mają kategorie, każda z nich ma inny sposób zmiany atrybutów wraz z upływem czasu. Przedstawia się to w następujący sposób:

- Standard (SellIn - 1, Quality -1 lub -2 jeśli SellIn = 0 / dzień)
- Aged Brie (SellIn -1, Quality +1 / dzień)
- Sulfuras (SellIn i Quality nie zmienia się)
- Backstage Pass (SellIn -1, Quality +1 jeśli SellIn > 10, Quality +2 jeśli SellIn <= 10, Quality +3 jeśli SellIn <= 5, Quality = 0 jeśli SellIn = 0 / dzień)
- Conjured (SellIn -1, Quality -2 / dzień)

Jak więc widać każda kategoria ma swoją “strategię” jak powinny zmieniać się atrybuty produktów do niej przynależnych. Stworzono więc dla nich osobne klasy implementujące interfejs ItemStrategy, w których wymuszone jest zdefiniowanie metody updateItem zwracającej zmodyfikowany produkt. W zależności od złożoności obliczeń nowych wartości atrybutów, każda klasa strategii posiada swoje prywatne metody pomocnicze (oprócz Sulfurasa, gdyż jego wartości pozostają stałe). Do klasy Item dodano dodatkowy atrybut typu Enum, w który zapewnia nam wybór istniejącej kategorii). Struktura projektu wygląda następująco:



ItemCategory:

```java
package com.gildedrose.enums;

public enum ItemCategory {
    Standard,
    AgedBrie,
    Sulfuras,
    BackstagePass,
    Conjured
}
```

Przykładowa strategia dla kategorii Standard:

```java
package com.gildedrose.strategies;

import com.gildedrose.Item;
import com.gildedrose.interfaces.ItemStrategy;

public class StandardItemStrategy implements ItemStrategy {
    @Override
    public Item updateItem(Item item) {
        int newSellIn = this.isSellInPositive(item.getSellIn()) ? item.getSellIn() - 1 : 0;
        int newQuality = this.isSellInPositive(newSellIn) ? item.getQuality() - 1 : item.getQuality() - 2;
        newQuality = this.isQualityPositive(newQuality) ? newQuality : 0;

        item.setSellIn(newSellIn);
        item.setQuality(newQuality);

        return item;
    }

    private boolean isSellInPositive(int sellIn) {
        return sellIn > 0;
    }

    private boolean isQualityPositive(int quality) {
        return quality > 0;
    }
}
```

W klasie GildedRose, używając mapy, każdej kategorii przypisano odpowiednią strategię aktualizacji atrybutów. Dzięki wykorzystaniu interfejsu ItemStrategy, możliwe było skorzystanie z polimorfizmu:

```java
    private static Map<ItemCategory, ItemStrategy> itemStrategyMap = new HashMap<>();

    static {
        itemStrategyMap.put(ItemCategory.Standard, new StandardItemStrategy());
        itemStrategyMap.put(ItemCategory.Sulfuras, new SulfurasItemStrategy());
        itemStrategyMap.put(ItemCategory.AgedBrie, new AgedBrieItemStrategy());
        itemStrategyMap.put(ItemCategory.BackstagePass, new BackstagePassItemStrategy());
        itemStrategyMap.put(ItemCategory.Conjured, new ConjuredItemStrategy());
    }

    public GildedRose(Item[] items) {
        this.items = items;
    }

    public void updateQuality() {
        for (int i = 0; i < items.length; i++) {
            ItemStrategy itemStrategy = itemStrategyMap.get(items[i].getCategory());
            items[i] = itemStrategy.updateItem(items[i]);
        }
    }
```

2. Testy

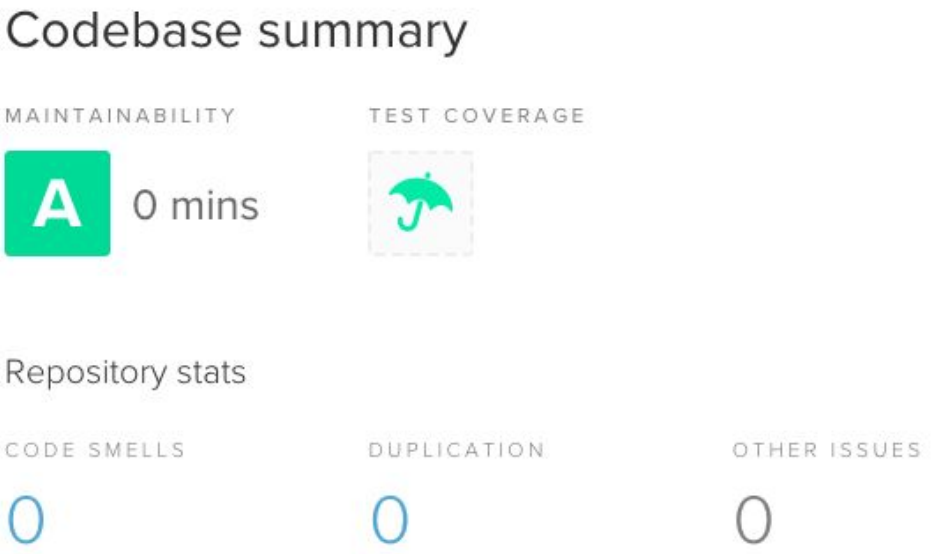
Każda strategia ma osobne testy sprawdzające standardowe oraz newralgiczne sytuacje (np. zmiana Quality o 3 jeśli 0 < SellIn <= 5 dla kategorii Backstage Pass). W trakcie realizacji projektu skorzystano z techniki TDD (Test Driven Development), czyli najpierw powstawały testy, a dopiero później logika biznesowa. Pozwalało to na szybsze ustalenie gdzie wyliczenia były błędne oraz uniknięcie błędów logicznych przy refaktoryzacji kodu. Przykład testów dla kategorii Standard:

```java
public class StandardItemStrategyTest {
    private static final String NAME = "Standard Item";
    private static final int SELL_IN = 10;
    private static final int QUALITY = 20;
    private static final ItemCategory CATEGORY = ItemCategory.Standard
    private ItemStrategy itemStrategy = new StandardItemStrategy();

    @Test
    public void decreaseSellInAndQuality() {
        Item item = new Item(NAME, SELL_IN, QUALITY, CATEGORY);

        item = this.itemStrategy.updateItem(item);

        assertEquals( expected: SELL_IN - 1, item.getSellIn());
        assertEquals( expected: QUALITY - 1, item.getQuality());
    }

    @Test
    public void decreaseQualityTwiceAsNormal() {
        Item item = new Item(NAME,  sellIn: 0, QUALITY, CATEGORY);

        item = this.itemStrategy.updateItem(item);

        assertEquals( expected: 0, item.getSellIn());
        assertEquals( expected: QUALITY - 2, item.getQuality());
    }

    @Test
    public void sellInAndQualityNotNegative() {
        Item item = new Item(NAME,  sellIn: 0,  quality: 0, CATEGORY);

        item = this.itemStrategy.updateItem(item);

        assertEquals( expected: 0, item.getSellIn());
        assertEquals( expected: 0, item.getQuality());
    }
```

Przetestowane zostały również klasy Item oraz GildedRose. W Items sprawdzone zostały gettery oraz settery, natomiast w GildedRose sprawdzono czy w metodzie updateQuality z mapy pobierane były odpowiednie instancje strategii.

3.Ocena kodu na Code Climate

Kod został sprawdzony pod kątem złożoności, jakości oraz duplikacji.



Szacowany czas na wprowadzenie zmian, poprawek, etc. oszacowany został na 0 minut.