

Autor: Maciej Boniaszczuk Informatyka II stopień – studia dzienne.

Link do githuba: <https://github.com/MaciejBoniaszczuk/GildedRose-Kata>

The Power of Refactoring

Gilded Rose Kata – Sprawozdanie

1. Wstęp

Zadanie zostało pobrane z GitHuba pani Emily Bache dostępnego pod adresem: <https://github.com/emilybache/GildedRose-Refactoring-Kata/tree/master/Java>. Praca została wykonana w języku Java. Celem projektu było zrefaktoryzowanie kodu sklepu. Należało tego dokonać wykorzystując metodę małych kroków, opierając się na zasadach SOLID, z głównym naciskiem na zasadę Open/Closed, która mówi o tym aby kod był „możliwy do rozszerzenia i zamknięty na modyfikacje”.

2. Przebieg pracy nad projektem

Napisanie testu:

Test napisałem z pomocą oraz sugestią wcześniej wspomnianej Emily Bache. Na swoim kanale YouTube udostępniła ona film (<https://www.youtube.com/watch?v=zyM2Ep28ED8&t=665s>), gdzie pokazuje jak napisać testy używając „Approval tests”. Na samym początku zmodyfikowałem istniejący już w klasie „GildedRoseTest” test, tak aby sprawdzał, czy zmiany wprowadzane podczas refaktoryzacji, nie powodują błędnych wyników. Do testowania skorzystano z metody „Approvals.verify”, która dokonuje zapisu wyników, a następnie porównuje go do kolejnego testu, sprawdzając czy wyniki nie uległy zmianie. Poniższy rysunek przedstawia kod testu „update Quality”.

```
@Test
public void updateQuality() {

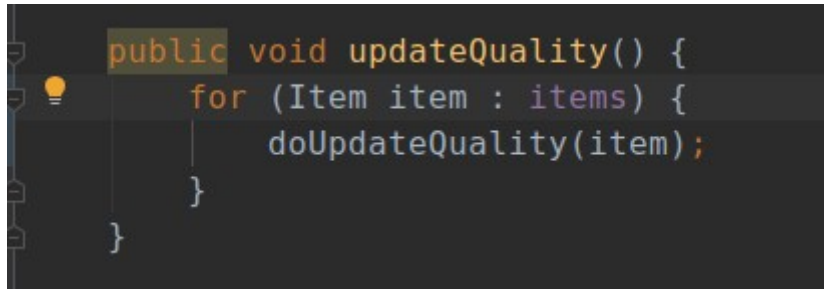
    CombinationApprovals.verifyAllCombinations(this::doUpdateQuality, new String[]{"foo", "Aged Brie",
        "Backstage passes to a TAFKAL80ETC concert", "Sulfuras, Hand of Ragnaros"},
        new Integer[]{-1, 0, 2, 6, 11},
        new Integer[]{0, 1, 49, 50});
}

private String doUpdateQuality(String name, int sellIn, int quality) {
    Item[] items = new Item[] { new Item(name, sellIn, quality) };
    GildedRose app = new GildedRose(items);
    app.updateQuality();
    return app.items[0].toString();
}
```

Rys1. Test: update Quality()

Refaktoryzacja:

Refaktoryzację zacząłem od „extract method”. W głównej metodzie programu „updateQuality” zostawiłem tylko pętlę for (zmieniłem ją na for each), która iteruje po każdym z istniejących itemów. Poniższy rysunek przedstawia wyżej wspomnianą metodę.

A screenshot of a code editor showing the implementation of the updateQuality method. The code is written in Java and consists of three lines: a public void declaration, a for loop iterating over an items collection, and a call to doUpdateQuality(item). The code is syntax-highlighted with colors: 'public' is orange, 'void' is light blue, 'updateQuality()' is orange, '{' is light blue, 'for' is orange, '(Item item : items)' is light blue, '{' is light blue, 'doUpdateQuality(item);' is orange, and '}' is light blue. A lightbulb icon is visible in the left margin next to the for loop line.

```
public void updateQuality() {  
    for (Item item : items) {  
        doUpdateQuality(item);  
    }  
}
```

Rys2. Metoda: updateQuality

Następnie metoda „updateQuality” przenosi nas do „wyjętej” metody doUpdateQuality, w której odbywa się główne działanie programu. Początkowo zawartość tej pętli była bardzo skomplikowana i trudna do zrozumienia ponieważ powtarzało się tam wiele if-ów oraz else-ów. Poniżej metoda doUpdateQuality przed refaktoryzacją.

```

private void doUpdateQuality(Item item) {
    if (!item.name.equals("Aged Brie")
        && !item.name.equals("Backstage passes to a TAFKAL80ETC concert")) {
        if (item.quality > 0) {
            if (!item.name.equals("Sulfuras, Hand of Ragnaros")) {
                item.quality = item.quality - 1;
            }
        }
    } else {
        if (item.quality < 50) {
            item.quality = item.quality + 1;

            if (item.name.equals("Backstage passes to a TAFKAL80ETC concert")) {
                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;
                    }
                }
            }
        }
    }
}

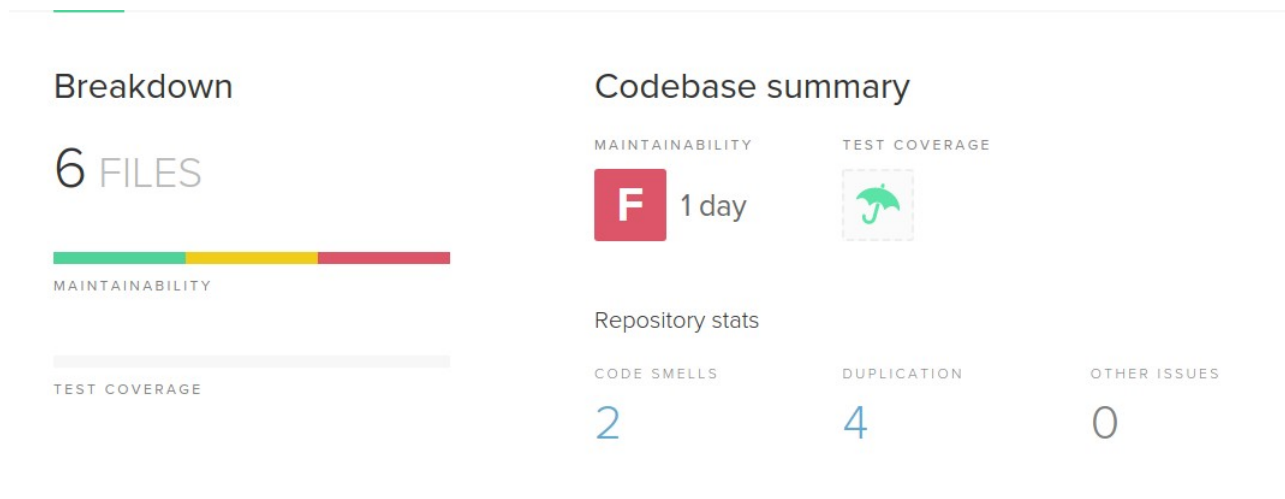
if (!item.name.equals("Sulfuras, Hand of Ragnaros")) {
    item.sellIn = item.sellIn - 1;
}

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

```

Rys3. DoUpdateQuality przed refaktoryzacją

Następnie kod został poddany testowi w CodeClimate. Otrzymany wynik maintainability to F.



Rys.4 CodeClimate po wstępnym refaktorze.

CodeClimate wykazał błędy takie jak:

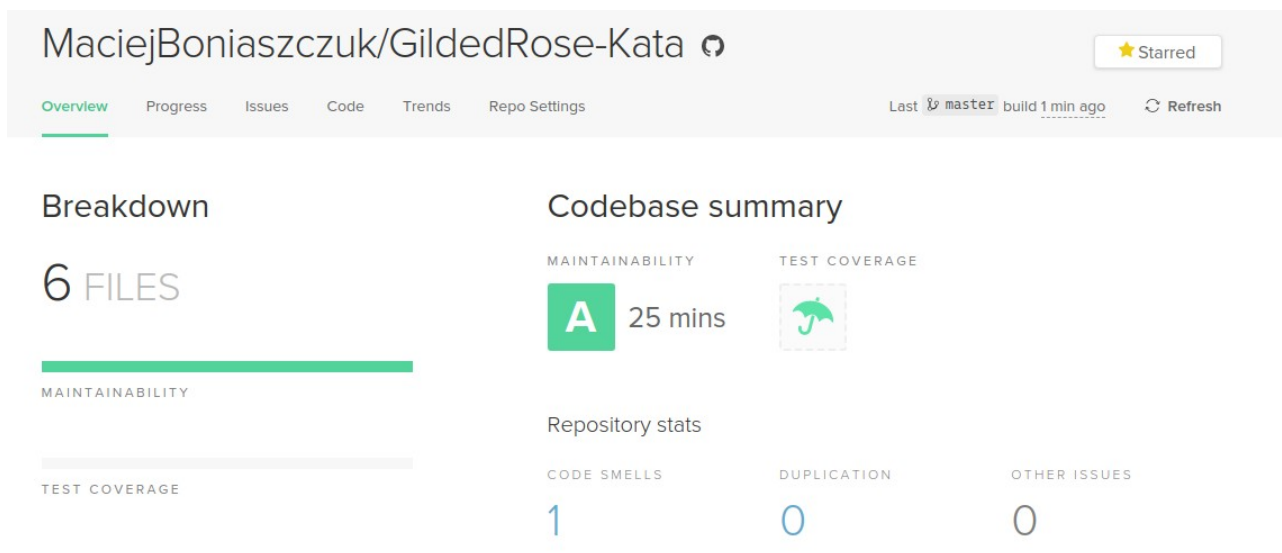
- powtarzające się – identyczne bloki kodu,
- powtarzające się – podobne bloki kodu,
- bardzo duża „complexity” metody doUpdateQuality,
- bardzo dużo linii kodu w metodzie doUpdateQuality,
- głęboko zagnieżdżone instrukcje kontroli.

Uwzględniając powyższe uwagi postanowiłem kontynuować refaktoryzację kodu. W metodzie doUpdateQuality zastosowałem instrukcję switch, która w zależności od nazwy itemu wykonuje określoną metodę. Poniżej zrefaktoryzowana metoda doUpdateQuality.

```
private void doUpdateQuality(Item item) {  
    switch (item.name) {  
        case "Aged Brie":  
            updateAgedBrie(item);  
            break;  
        case "Backstage passes to a TAFKAL80ETC concert":  
            updateBackstagePasses(item);  
            break;  
        case "Sulfuras, Hand of Ragnaros":  
            break;  
        case "Conjured":  
            updateConjured(item);  
            break;  
        default:  
            updateDefault(item);  
            break;  
    }  
}
```

Rys.5 Metoda doUpdateQuality po refaktoryzacji.

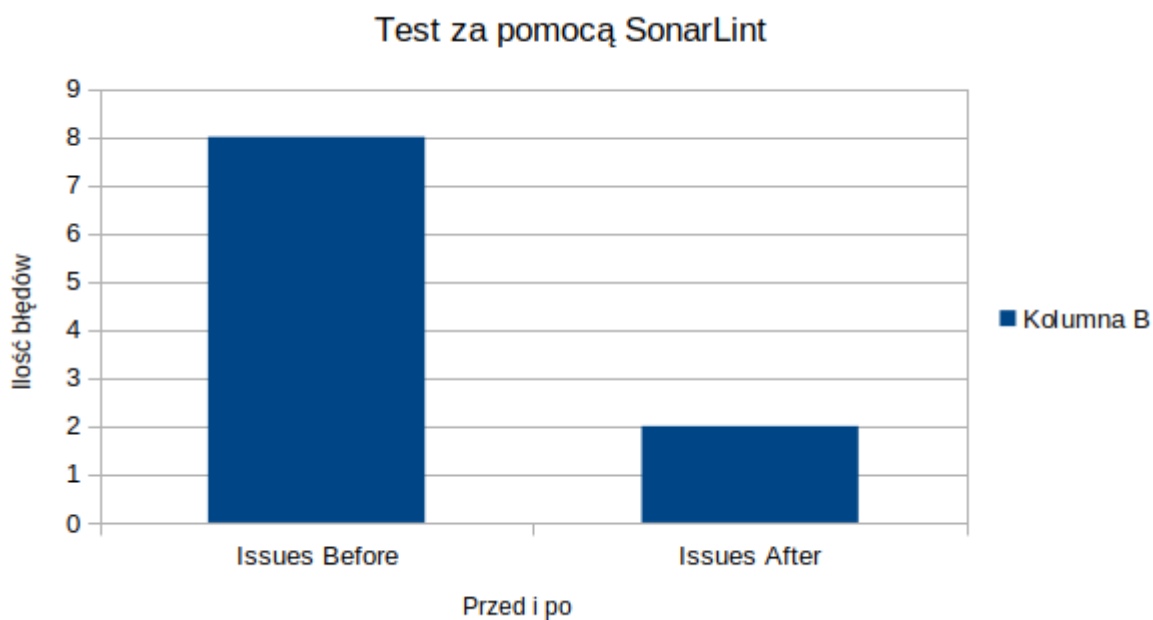
Jak widać na powyższym rysunku powstały 4 kolejne metody, w których znajdował się kod odpowiadający za zmienianie wartości „sellIn” oraz „quality” dla każdego z itemów. Ponownie przetestowano kod w CodeClimate, jednak wynik nadal nie był zadowalający. Powtarzające się fragmenty kodu ponownie „wyekstraktowano” na zewnątrz”. Finalnie CodeClimate ocenił kod na najlepszą ocenę – A.



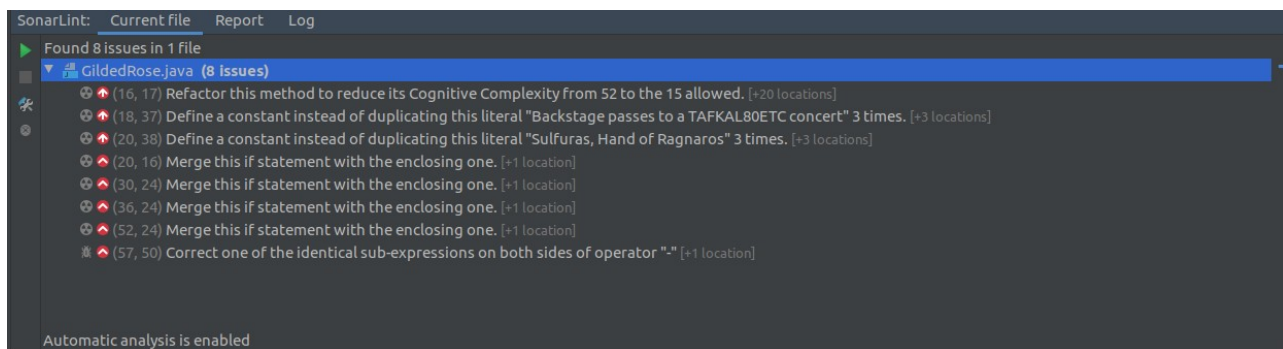
Rys.6 CodeClimate po refaktoryzacji.

Na końcu przetestowano działanie sklepu w okresie 120 dni. Test nie wykrył żadnych błędów.

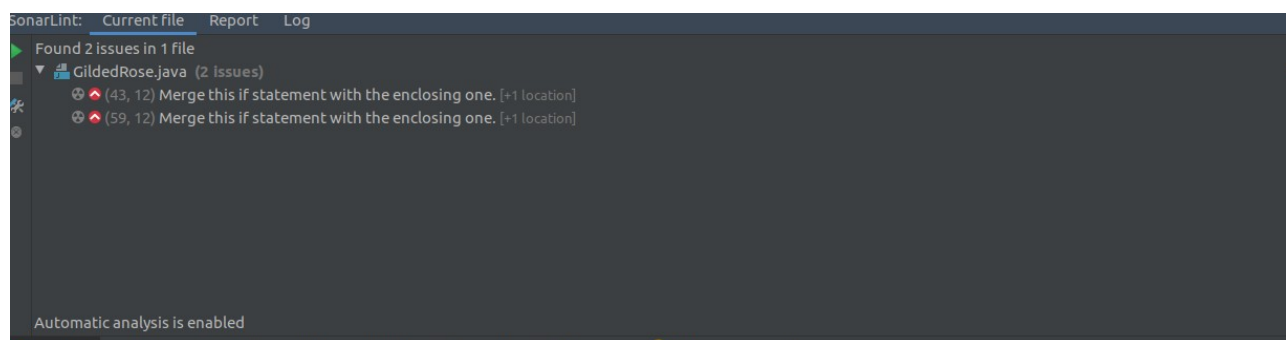
Kod przeskanowano również za pomocą jednego z CodeSmells – SonarLint. Poniżej wykres opisujący ilość „issues” przed oraz po refaktoryzacji.



Jak widać na powyższym wykresie słupkowym. Kod po zrefaktoryzowaniu zawiera 4 razy mniej błędów niż przed. Na kolejnej stronie zamieszczam zapis z konsoli SonarLint-u.

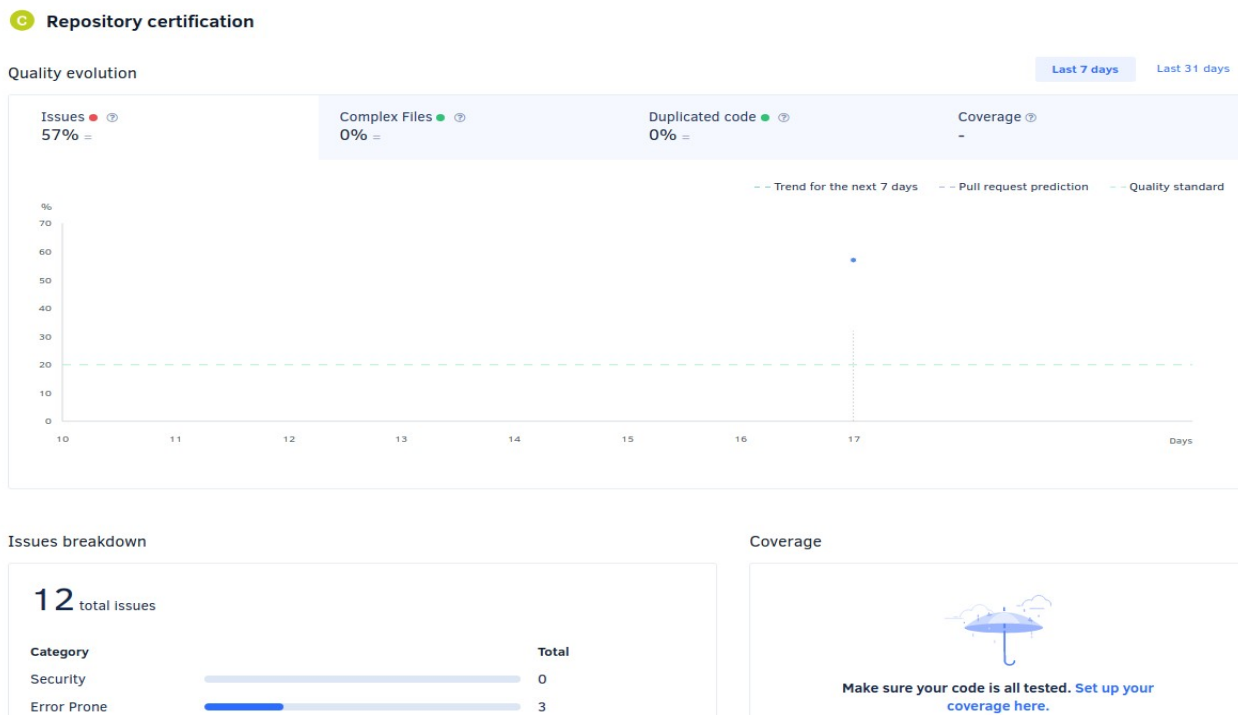


Rys.8 SonarLint przed refaktoryzacją.



Rys.8 SonarLint po refaktoryzacji.

Kod sprawdzono również na platformie „Codacy”. Kod przed refaktoryzacją został oceniony na ocenę C, wartość procentowa błędów została określona na 57%, a ich liczba na 12.



Rys.9 Wynik z platformy Codacy przed refaktoryzacją.

Następnie biorąc pod uwagę błędy jakie zostały wymienione wnosilem poprawki do kodu. Poniżej znajduje się zestawienie, jak wraz z kolejnymi commitami zmieniała się ilość błędów.

CREATED	ISSUES	
3 minutes ago	0 NEW	3 FIXED
6 minutes ago	0 NEW	4 FIXED
9 minutes ago	0 NEW	0 FIXED
4 days ago	0 NEW	0 FIXED
4 days ago	0 NEW	0 FIXED
4 days ago	- NEW	- FIXED
4 days ago	0 NEW	1 FIXED
4 days ago	0 NEW	0 FIXED
4 days ago	2 NEW	1 FIXED

Rys.10 Proces refaktoryzacji – zmiana ilości „issues”

Finalnie osiągnięto wynik B – wartość procentowa issues spadła o 16%, a wartość liczbowa o 9. Zmianę tej wartości możemy zaobserwować na poniższym wykresie.



Rys.11 Wynik z platformy Codacy po refaktoryzacji.