Zaawansowane Języki Programowania
Gilded Rose Refactoring Kata

Arkadiusz Dąbrowski
11.11.2019

1. Opis problemu

Zdanie oryginalnie opracowane przez Terry'ego Hughesa polega na dodaniu do istniejącego już systemu sklepu nowej funkcjonalności. Ogólnie system ten można opisać w trzech zdaniach:

- Wszystkie przedmioty mają wartość (*SellIn)*, która opisuje liczba dni, w której trzeba sprzedać przedmiot.
- Wszystkie przedmioty mają wartość *(Quality)*, która opisuje wartość przedmiotu.
- Na koniec każdego dnia system obniża obie wartości dla każdego przedmiotu.

Niektóre przedmioty mają specjalne własności, które opisują poniższe reguły:

- W momencie kiedy dzień sprzedaży minął *Quality* zmniejsza się dwukrotnie szybciej.
- *Quality* przedmiotu nigdy nie może być ujemna.
- *Quality* przedmiotu „Aged Brie" rośnie z każdym dniem zamiast maleć.
- *Quality* przedmiotu nigdy nie może być większa niż 50.
- Przedmiot „Sulfuras" jest wyjątkowy, nigdy nie będzie sprzedany ani nie traci na *Quality*, która zawsze wynosi 80 i nigdy się nie zmienia.
- Przedmiot „Backstage passes", podobnie jak „Aged Brie", zyskuje na *Quality* w miarę zbliżania się do dnia koncertu. *Quality* zwiększa się o 2 kiedy zostało 10 dni lub mniej, o 3 kiedy zostało 5 dni lub mniej, ale *Quality* spada do 0 po koncercie.

Naszym zadaniem jest dodanie obsługi kolejnego przedmiotu „Conjured", który spełnia poniższą regułę:

- Przedmiot „Conjured" traci na *Quality* dwa razy szybciej niż normalne przedmioty.

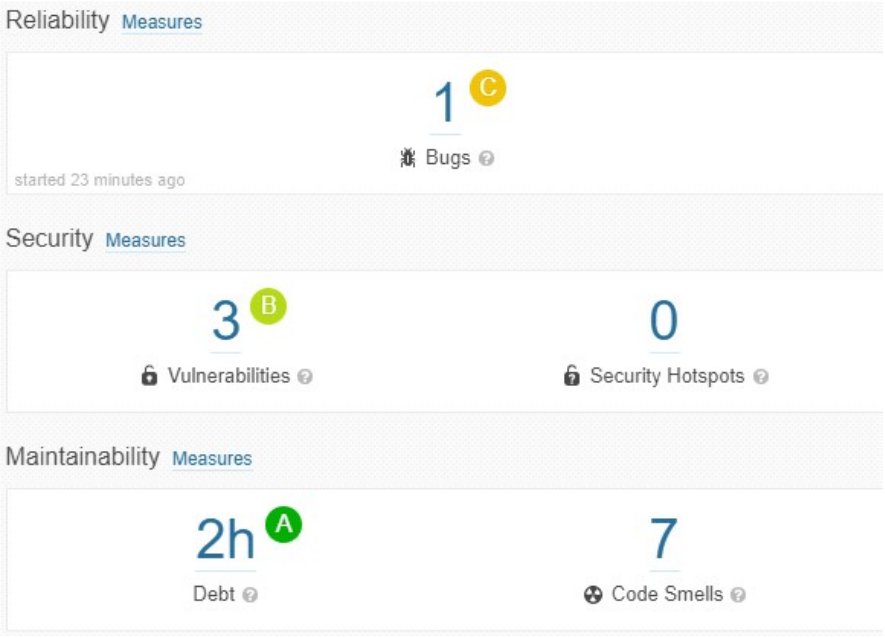
Przy dodawaniu tej funkcjonalności musimy trzymać się kilku narzuconych z góry dwóch zasad:

- Możemy zmieniać metodę *UpdateQuality* oraz dodawać jakikolwiek nowy kod, tak długo jak wszystko nadal działa poprawnie.
- Nie możemy zmieniać niczego w klasie *Item.*

2. Analiza projektu

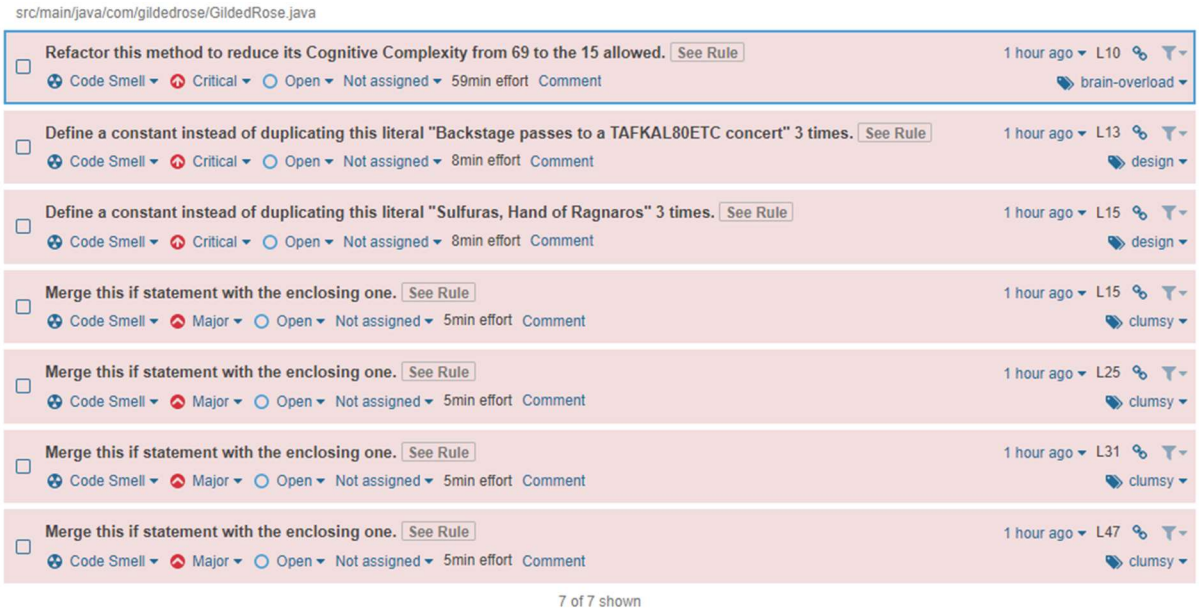
Do analizy projektu skorzystałem z narzędzia sonarqube (https://www.sonarqube.org/), które umożliwia wykonanie przejrzystej i wnikliwej analizy statycznej naszego kodu. Przy wykorzystaniu tego narzędzia wykonano wszystkie dalsze pomiary.

Ogólne dane dotyczące analizowanego kodu bazowego przedstawiały się następująco:



Z powyższego zrzutu ekranu możemy łatwo zauważyć, że kod zawiera 3 podatności (Vulnerabilities), które nie będą jednak przedmiotem zadania, gdyż dotyczą wyłącznie klasy *Item,* której z założenia nie mamy modyfikować. Zdecydowanie ważniejsze są pozostałe 3 informacje, czyli:

- 1 błąd (Bug) – istotny błąd w kodzie, najważniejsza dla nas informacja, jest to rzecz którą powinno się naprawić w pierwszej kolejności.
- 2 godziny długu technicznego (Debt) – jest to wszystko w kodzie co powoduje, że jesteśmy wolniejsi w swojej pracy, świadczy o jakości danego kodu.
- 7 zapachów kodu (Code Smells) – cechy kodu mówiące o złym sposobie implementacji i będące przesłanką do refaktoryzacji. W analizowanym kodzie prezentują się one następująco:



Ostatnim punktem naszej analizy jest sprawdzenie jak prezentują się dwie ważne metryki:

- Złożoność cyklomatyczna (Cyclomatic Complexity) – mówi o stopniu skomplikowania programu, gdzie podstawą do wyliczeń jest liczba punktów decyzyjnych w tym programie. Można przyjąć poniższe wartości:
  - od 1 do 10 – prosty kod stwarzający nieznaczne ryzyko
  - od 11 do 20 – złożony kod powodujący ryzyko na średnim poziomie
  - od 21 do 50 – bardzo złożony kod związany z wysokim ryzykiem
  - powyżej 50 – kod niestabilny grożący bardzo wysokim poziomem ryzyka.
- Złożoność poznawcza (Cognitive Complexity) – mówi ona o tym jak ciężki do zrozumienia jest dany kod przez czytacza. Sonarqube przyjmuje wartość 15 jako dopuszczalną.

Dla analizowanego projektu wyżej wymienione metryki prezentują się następująco:

Na podstawie powyższych metryk możemy wywnioskować, że kod jest bardzo złożony, związany z wysokim ryzykiem oraz niezwykle trudny do zrozumienia przez czytacza, wartość dopuszczalna jest przekroczona tutaj ponad czterokrotnie.

3. Wpływ dodania przedmiotu „Conjured" na bazowy kod

Kolejnym wykonanym przeze mnie krokiem była implementacja obsługi przedmiotu „Conjured" według założeń wynikających z treści zadania. Wykonanie sprowadziło się do dodania poniższego kodu w 2 miejscach.

```
if (items[i].name.equals("Conjured Mana Cake") && items[i].quality != 0) {
    items[i].quality = items[i].quality - 1;
}
```

Przyjrzyjmy się wpływowi tej prostej zmiany w kodzie na badane przez nas wcześniej metryki. Poniższy zrzut ekranu prezentuje złożoności dla zmodyfikowanego kodu. Jak widać złożoność cyklomatyczna wzrosła o 4 punkty, natomiast złożoność poznawcza aż o 14 punktów, co świadczy o dużym poziomie skomplikowania, spowodowanych dużą ilością zagnieżdżonych if'ów.

4. Testy jednostkowe

Istotnym etapem przed przystąpieniem do faktycznej refaktoryzacji kodu źródłowego jest napisanie odpowiednich testów jednostkowych. Mają one za zadanie zadbać o to, aby żadna wprowadzona zmiana nie spowodowała błędnego działania aplikacji. W tym celu napisałem 49 testów jednostkowych na podstawie specyfikacji zadania, zwracając przy tym szczególną uwagę na różne skrajne przypadki. Do sporządzenia testów wykorzystałem JUnit5 oraz AssertJ.

Na poniższym raporcie z przeprowadzonych testów można zauważyć jak rozkładała się Ilość testów, w zależności od testowanego typu przedmiotu.

```
-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running com.gildedrose.AgedBrieTest
Tests run: 11, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.11 s - in com.gildedrose.AgedBrieTest
Running com.gildedrose.BackstagePassesTest
Tests run: 13, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.016 s - in com.gildedrose.BackstagePassesTest
Running com.gildedrose.ConjuredTest
Tests run: 9, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.006 s - in com.gildedrose.ConjuredTest
Running com.gildedrose.DefaultItemTest
Tests run: 9, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.005 s - in com.gildedrose.DefaultItemTest
Running com.gildedrose.SulfurasTest
Tests run: 7, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0 s - in com.gildedrose.SulfurasTest

Results:

Tests run: 49, Failures: 0, Errors: 0, Skipped: 0
```

```java
public class AgedBrieTest {
    GildedRose app;
    Item[] items;
    String itemName = "Aged Brie";
    int sellIn = 5;
    int quality = 5;
    int numberOfUpdates = 3;

    @Test
    void
quality ShouldIncreaseProperly WhenPositiveSellInAndUpdatedMultipleTimes()
{
        items = new Item[] { new Item(itemName, sellIn, quality) };

        app = new GildedRose(items);
        for(int i = 0; i < numberOfUpdates; i++){
            app.updateQuality();
        }

        assertThat(app.items[0].quality).isEqualTo(quality +
numberOfUpdates);
    }

    @Test
    void quality_ShouldIncreaseProperly_WhenNegativeSellInAndUpdatedOnce()
{
        int sellIn = -2;
        items = new Item[] { new Item(itemName, sellIn, quality) };

        app = new GildedRose(items);
        app.updateQuality();

        assertThat(app.items[0].quality).isEqualTo(quality + 2);
    }

    @Test
    void quality_CannotExceed50_WhenPositiveSellInAndUpdatedOnce() {
        int quality = 49;
        items = new Item[] { new Item(itemName, sellIn, quality) };

        app = new GildedRose(items);
        app.updateQuality();

        assertThat(app.items[0].quality).isEqualTo(50);
    }
}
```

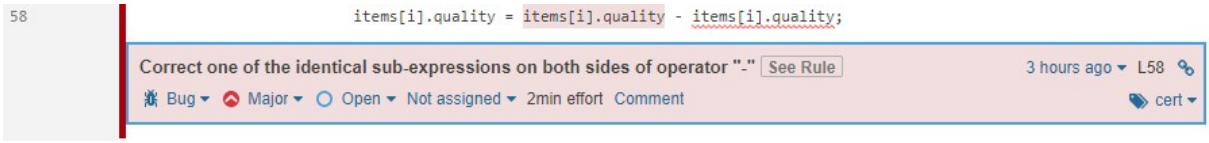
Na powyższym fragmencie kodu widać trzy przykładowe testy sprawdzające działanie przedmiotu „Aged Brie” dla odpowiednich przypadków, opisanych odpowiednio w nazwach metod.

Od tego momentu po każdej, nawet najmniejszej, wprowadzonej zmianie w kodzie uruchamiane były powyższe testy w celu kontroli poprawności działania programu.

5. Refaktoryzacja – „małe kroki”

Poniższe etapy zostały wykonane zostały metodą „małych kroków”. Modyfikowano możliwie małe fragmenty kodu i po każdej ze zmian przeprowadzano testy jednostkowe. Rozpoczynając proces refaktoryzacji niezwykle pomocne okazały się informacje otrzymane z analizy sonarqube’a (bugi, „code smells”). Sugerując się tymi danymi, własną intuicją oraz wskazówkami zawartymi na stronie Refactoring Guru (https://refactoring.guru/) wykonane zostały poniższe kroki we wskazanej kolejności. (Znak ↓ oznacza „kierunek” zmiany kodu):

- Usunięcie jedynego buga wskazanego przez analizę statyczną.




```
items[i].quality = items[i].quality - items[i].quality;
```

↓

```
items[i].quality = 0;
```

- Poprawienie ogólnej czytelności kodu.
  - Zamiana pętli for na for each.

```
for (int i = 0; i < items.length; i++) {
    if (!items[i].name.equals("Aged Brie")
```

↓

```
for(Item item : items){
    if (!item.name.equals("Aged Brie")
```

  - Zamiana inkrementacji i dekrementacji.

```
item.quality = item.quality + 1;
```

↓

```
item.quality += 1;
```

  - „Zwinięcie” if’ów pod jeden warunek.

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

↓

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

- Przeniesienie dużego fragmentu zduplikowanego kodu, występującego w dwóch miejscach, do osobnej metody (**Extract Method**).

```
if (!item.name.equals("Backstage passes to a TAFKAL80ETC concert")) {
        if (item.quality > 0 && !item.name.equals("Sulfuras, Hand of
Ragnaros")) {
        item.quality -= 1;
        if (item.name.equals("Conjured Mana Cake") && item.quality != 0) {
            item.quality -= 1;
        }
    }
} else {
    item.quality = 0;
}
```

↓

```
if (!item.name.equals("Backstage passes to a TAFKAL80ETC concert")) {
    decreaseQuality(item);
} else {
    item.quality = 0;
}
```

```
private void decreaseQuality(Item item) {
    if (item.quality > 0 && !item.name.equals("Sulfuras, Hand of
Ragnaros")) {
        item.quality -= 1;
        if (item.name.equals("Conjured Mana Cake") && item.quality != 0) {
            item.quality -= 1;
        }
    }
}
```

- Przeniesienie pozostałych dużych bloków if w metodzie *updateQuality()* do osobnych metod (**Extract Method**) i uproszczenie tej metody do poniższej postaci:

```
public void updateQuality() {
    for(Item item : items){
        modifyQualityWhenSellInAboveZero(item);
        decreaseSellIn(item);
        modifyQualityWhenSellInDropsToZero(item);
    }
}
```

- Wydzielenie nazw przedmiotów jako stałe (**Replace Magic Number with Symbolic Constant**).

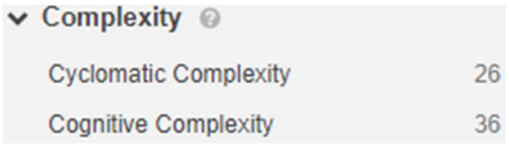
```
if (item.name.equals("Backstage passes to a TAFKAL80ETC concert"
)) {
```

↓

```
private static final String BACKSTAGE_PASSES = "Backstage passes to a
TAFKAL80ETC concert";
```

```
if (item.name.equals(BACKSTAGE_PASSES)) {
```

Na obecnym etapie refaktoryzacji wykonałem po raz kolejny analizę statyczną obecnego kodu, aby zbadać czy obrany przeze mnie kierunek jest dobry. Na poniższym zrzucie ekranu widać efekt przeprowadzonych przeze mnie zmian. Można łatwo zauważyć, że złożoność poznawcza uległa znacznej poprawie, z wyniku 83 zmalała do 36. Choć nie jest to jeszcze idealna wartość, to pokazuje to wprost, że kierunek refaktoryzacji jest odpowiedni i efektywny.



Ostatnim „małym krokiem” będzie uproszczenie złożonych warunków (**Decompose Conditional**).

```
if (item.sellIn < 11 && item.quality < 50) {
```

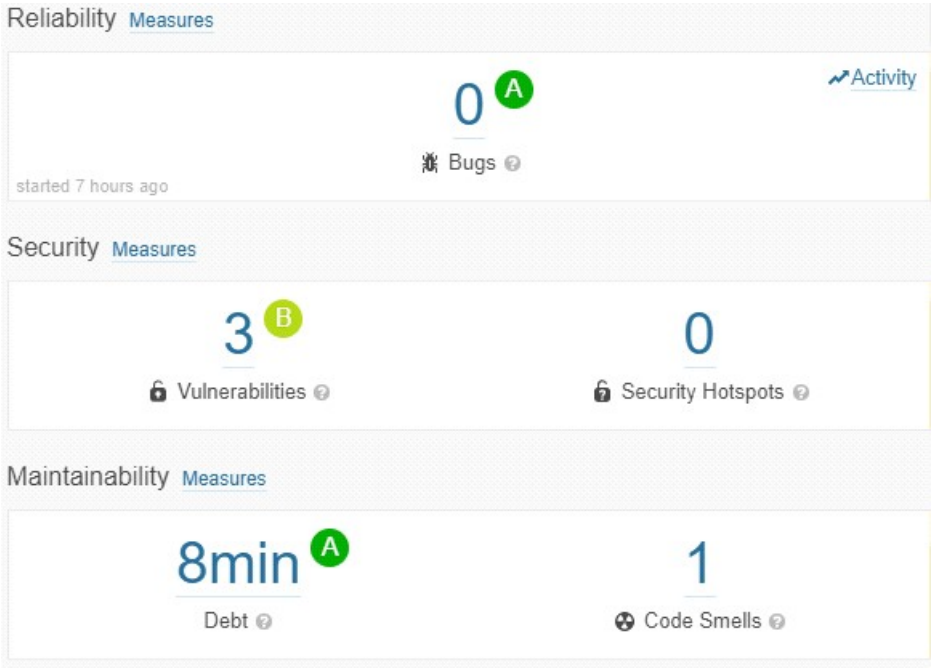
↓

```
if (isTenDaysBeforeConcert(item)) {
```

```
private boolean isTenDaysBeforeConcert(Item item) {
    return item.sellIn < 11 && item.quality < 50;
}
```

W podobny sposób wydzieliłem w sumie 14 nowych metod, znacznie upraszczając warunki w kodzie i ułatwiając ich ogólne zrozumienie.

Wszystkie powyższe zmiany doprowadziły mnie do momentu, w którym analiza statyczna wykazała dane jak na poniższym zrzucie ekranu. Błąd został wyeliminowany, dług techniczny został zredukowany z 2 godzin do 8 min oraz 6 z 7 „code smells” zostało wyeliminowanych. Potwierdza to poprzednie stwierdzenie, że kierunek refaktoryzacji jest odpowiedni i efektywny.



6. Refaktoryzacja – zastosowanie polimorfizmu

Dotychczas w kodzie wydzielone zostały cztery stałe:

```
private static final String BACKSTAGE_PASSES = "Backstage passes to a
TAFKAL80ETC concert";
private static final String AGED_BRIE = "Aged Brie";
private static final String SULFURAS = "Sulfuras, Hand of Ragnaros";
private static final String CONJURED = "Conjured Mana Cake";
```

Z powyższego fragmentu kodu wynika wprost, że mamy do czynienia z czterema typami różnych przedmiotów, gdzie dla każdego z nich wykonywane są operacje uaktualnienia dla nich dwóch wartości, *Quality* oraz *SellIn*. Można z tego wywnioskować, że najlepszą metodą dalszej refaktoryzacji będzie zastosowanie polimorfizmu (**Replace Conditional with Polymorphism**).

Pierwszym krokiem było utworzenie klasy abstrakcyjnej wyspecjalizowanego przedmiotu, który jest niejako odzwierciedleniem klasy *Item* i posłuży nam ona do jej „opakowania” (wrapper). Dodatkowo ta klasa posiada abstrakcyjną metodę *update()*, którą każda z dziedziczonych klas musi zaimplementować. Każda klasa przedmiotu dziedziczy po tej klasie abstrakcyjnej. Zgodnie z tym zamysłem zaimplementowałem tą klasę wraz z różnymi metodami pomocniczymi.

```
public abstract class SpecializedItem {
    private static final int MAX_VALUE = 50;
    private static final int MIN_VALUE = 0;
    protected Item item;

    public SpecializedItem(Item item){
        this.item = item;
    }

    public abstract void update();

    private boolean isQualityNotMax(){
        return item.quality < MAX_VALUE;
    }

    private boolean isQualityNotMin(){
        return item.quality > MIN_VALUE;
    }

    protected void increaseQuality(){
        if(isQualityNotMax()) item.quality++;
    }

    protected void increaseQualityBy(int n){
        for(int i = 0; i < n; i++) if(isQualityNotMax()) item.quality++;
    }

    protected void decreaseQuality(){
        if(isQualityNotMin()) item.quality--;
        if(isSellInBelowZero()) item.quality--;
    }

    protected void decreaseSellIn(){
        item.sellIn--;
    }

    protected boolean isSellInBelowZero(){
        return item.sellIn < MIN_VALUE;
    }
}
```

Kolejny krok to stworzenie klas odpowiadających rozważanym przez nam przedmiotom. Sposób implementacji takiej klasy przedstawię na przykładzie przedmiotu „Aged Brie”. Wszystkie pozostałe klasy zostały zaimplementowane w sposób niemal identyczny, z odpowiednim uwzględnieniem wymagań zadania.

```
public class AgedBrie extends SpecializedItem {
    public AgedBrie(Item item){
        super(item);
    }

    @Override
    public void update() {
        increaseQuality();
        decreaseSellIn();
        if(isSellInBelowZero()) increaseQuality();
    }
}
```

Jak można zauważyć na powyższym fragmencie kodu implementacja kolejnych klas przedmiotów sprowadza się do przeciążenia metody *update()* co znacząco ułatwia potencjalne rozszerzanie aplikacji o kolejne typu przedmiotów.

Następny krok to stworzenie mechanizmu odpowiedniego tworzenia przedmiotu w zależności od jego typu. W tym przypadku dobrym rozwiązaniem jest zastosowanie wzorca projektowego fabryki. W moim przypadku zdecydowałem się na utworzenie fabryki ze statyczną metodą zwracającą nam „wyprodukowany" obiekt. Prezentuje się ona w następujący sposób:

```
public class SpecializedItemFactory {
    private SpecializedItemFactory(){}

    public static SpecializedItem getSpecializedItem(Item item){
        switch(item.name){
            case "Aged Brie":
                return new AgedBrie(item);
            case "Sulfuras, Hand of Ragnaros":
                return new Sulfuras(item);
            case "Backstage passes to a TAFKAL80ETC concert":
                return new BackstagePass(item);
            case "Conjured Mana Cake":
                return new ConjuredItem(item);
            default:
                return new DefaultItem(item);
        }
    }
}
```

Mając w ten sposób przygotowaną strukturę mogłem rozpocząć odpowiednią implementację w klasie *GildedRose*. Na początku stworzyłem listę moich wyspecjalizowanych przedmiotów oraz odpowiednio ją wypełniłem opakowując przedmioty klasy *Item* w konstruktorze wykorzystując przy tym utworzoną wcześniej fabrykę w sposób przedstawiony na poniższym fragmencie kodu.

```
Item[] items;
private ArrayList<SpecializedItem> specializedItems;

public GildedRose(Item[] items) {
    specializedItems = new ArrayList<>();
    this.items = items;
    for(Item item : items){
        specializedItems.add(SpecializedItemFactory
                .getSpecializedItem(item));
    }
}
```

Od tego momentu zamiast iterować po tablicy przedmiotów klasy *Item* mogę robić to wykorzystując listę wyspecjalizowanych przedmiotów i wywołując dla nich metodę *update()*. Klasa *GildedRose* po wszystkich przeprowadzonych refaktoryzacjach prezentuje się w następujący sposób:

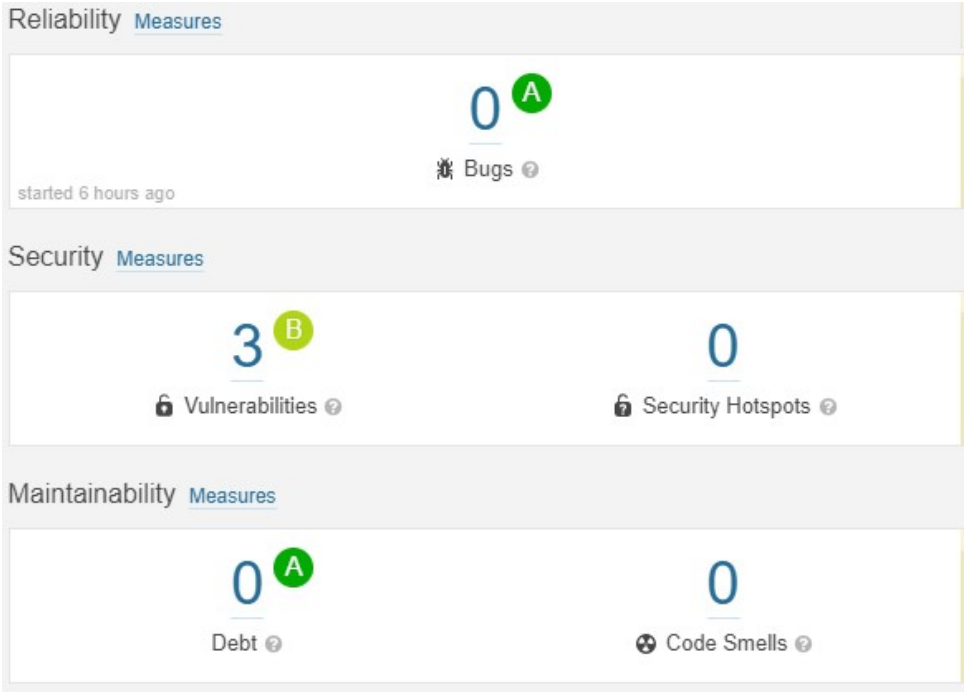
```
class GildedRose {
    Item[] items;
    private ArrayList<SpecializedItem> specializedItems;

    public GildedRose(Item[] items) {
        specializedItems = new ArrayList<>();
        this.items = items;
        for(Item item : items){
            specializedItems.add(SpecializedItemFactory
                    .getSpecializedItem(item));
        }
    }

    public void updateQuality(){
        for(SpecializedItem item : specializedItems){
            item.update();
        }
    }
}
```

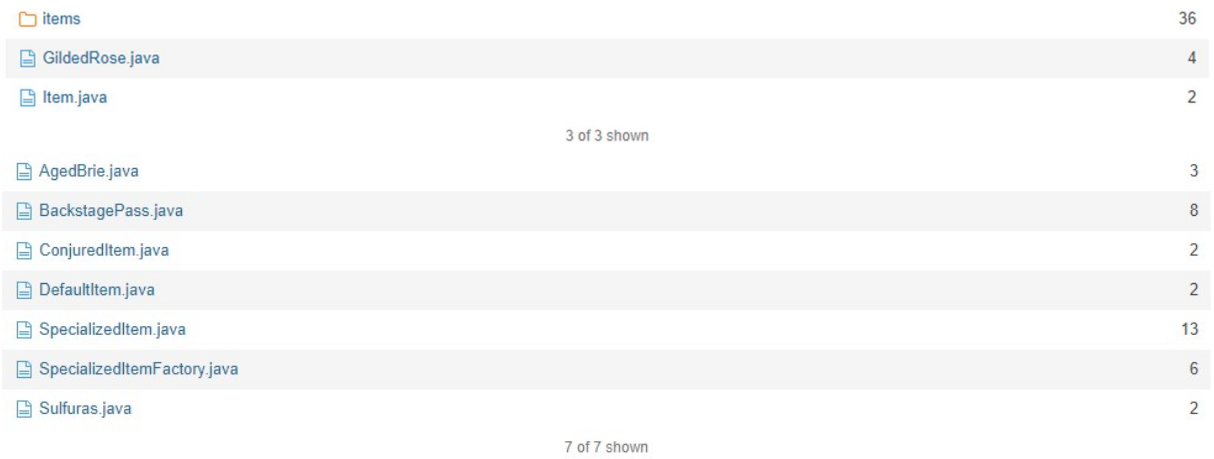
7. Końcowa analiza projektu

Po zakończeniu procesu refaktoryzacji nadszedł czas na wykonanie końcowej analizy statycznej kodu. Jak widać na poniższym zrzucie ekranu całkowicie wyeliminowane zostały wszystkie błędy, „code smells" oraz dług techniczny, co świadczy o dobrym wpływie refaktoryzacji na jakość kodu.

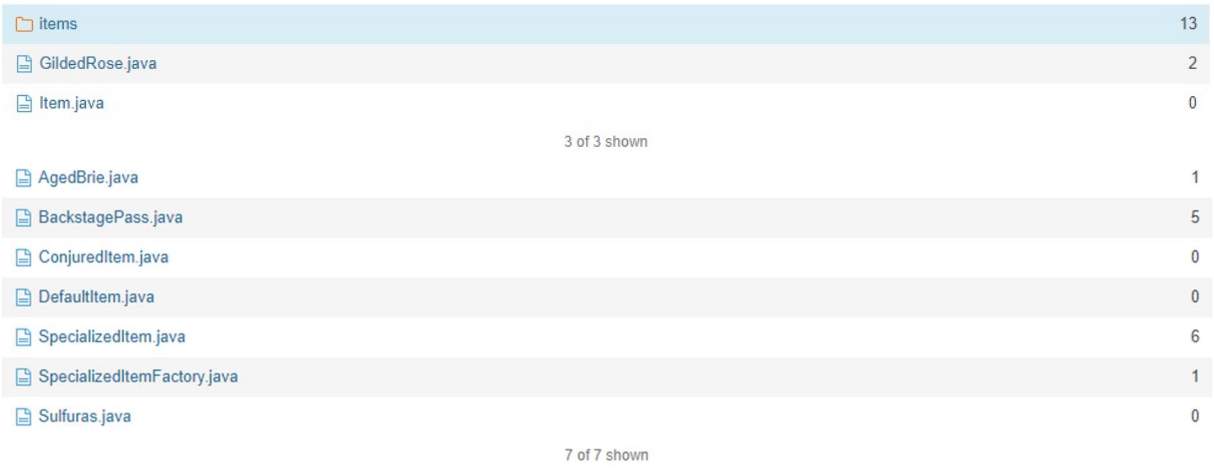


Przyjrzyjmy się teraz jak wyglądają złożoności dla poszczególnych klas.

Złożoność cyklomatyczna przedstawiona na poniższym zrzucie ekranu pokazuje, że osiągnąłem zamierzony cel i znacząco obniżyłem wartości tej metryki. Aby lepiej zobrazować wyniki skupimy się na średniej wartości dla wszystkich klas. Jest ona na poziomie 4.7 punktów, co oznacza że w ogólności kod jest prosty i stwarza nieznaczne ryzyko.



Poniżej przedstawiona została złożoność poznawcza zrefaktoryzowanego kodu. Podobnie jak w przypadku złożoności cyklomatycznej, w celu lepszego zobrazowania wyników zwrócimy uwagę na średnią wartość. W tym przypadku wynosi ona 1.7 punktu. Porównując do założonej dopuszczalnej wartości na poziomie 15 punktów, można wywnioskować, że kod jest bardzo prosty do zrozumienia przez czytacza.



8. Podsumowanie

Udało mi się pomyślnie zaimplementować przedmiot „Conjured” przy zachowaniu założeń wyszczególnionych w specyfikacji zadania oraz wykonać niezbędne refaktoryzacje metodą „małych kroków”.

Napisanie odpowiednich testów jednostkowych okazało się kluczowym etapem przeprowadzania procesu refaktoryzacji. Pozwoliły one na kontrolowane zmiany w kodzie bez obawy o zepsucie działającego systemu.

Dodatkowo wykonanie tego zadania pokazało istotę efektywnego korzystania z narzędzi do statycznej analizy kodu źródłowego. W oparciu o nie o wykonane refaktoryzacje znacząco zredukowałem dwie istotne metryki: złożoność cyklomatyczną z 26 do 4.7 punktów oraz złożoność poznawczą z 83 do 1.7 punktu.