

Zaawansowane języki programowania

Gilded Rose Kata- „Widzę i rozumiem”

1. Naprawienie/ Stworzenie działającego testu programu

Podstawowy kod Gilded Rose Kata (Snapshot 0) pobrano z repozytorium:

<https://github.com/emilybache/GildedRose-Refactoring-Kata>

Pracę wykonano w języku Java, wykorzystano środowisko IntelliJ. Pracę zamieszczono na:

<https://github.com/tronsez/Gilded-Rose-Java>

Aby ułatwić sobie pracę w IntelliJ, warto załadować pliki pobrane z repozytorium jako Maven Project.

Poniższy test jest propozycją od Emily Bache, przedstawiono go tutaj:

<https://www.pragma.com/stories/advanced-testing-refactoring-techniques/>

Przed rozpoczęciem pracy nad kodem GildedRose, należy najpierw pomyśleć o stworzeniu testu, którego zadaniem będzie sprawdzanie, czy zmiany wprowadzone poprzez refaktoryzację, nie zmieniają działania całego programu. Test o nazwie „GildedRoseTest” dodany do całego programu na początku nie działa.

```
1 package com.gildedrose;
2
3 import static org.junit.Assert.*;
4
5 import org.junit.Test;
6
7 public class GildedRoseTest {
8
9     @Test
10    public void foo() {
11        Item[] items = new Item[] { new Item( name: "foo", sellin: 0, quality: 0 ) };
12        GildedRose app = new GildedRose(items);
13        app.updateQuality();
14        assertEquals( expected: "fixme", app.items[0].name);
15    }
16
17 }
```

Rys. 1. Plik *GildedRoseTest* przed modyfikacją

Jak można zauważyć test spodziewa się wartości „fixme”, podczas gdy zadaną testowaną zmienną jest „foo”. Aby test zadziałał należy zmienić w 14 linijce kodu „fixme” na „foo”, program się uruchomi, ale nie jest to pożądany test. Na początku warto zmienić nazwę całej metody na np „updateQuality”. Skorzystano tutaj z innej metody testowania „Approvals.verify”, zamiast „assertEquals”. Aby skorzystać z tej metody należy do pliku pom.xml, zaimportować ApprovalsTests, jak również do samego pliku testu. Zasada działania jest bardzo prosta. Approval tests wykonuje zapis wyników testu, a następnie porównuje go do nowego testu, sprawdzając czy wyniki nie uległy zmianie.

Następnie po wprowadzeniu nowej metody testowania uruchomiono test. Podczas pierwszej próby wykonania testu, ten wyrzuci błąd oraz podpowiedź jego naprawy. Należy przenieść odpowiedni plik za pomocą komendy „move” w terminalu. Chodzi o to, aby to co zwraca program zgadzało się z tym czego Approval test się spodziewa. Po zmianie kodu *GildedRoseTest*, nastąpiły zmiany, których Approval test się nie spodziewał, dlatego za pomocą „move” należy przenieść plik ze spodziewanym wynikiem, po każdej zmianie kodu *GildedRoseTest*. Następnie poddano modyfikacji wartość poddawaną weryfikacji na „app.items[0].toString”.

Aby ponownie ułatwić sobie pracę ze środowiskiem programowania, warto zmienić ustawienia Code Coverage, po to aby, przy uruchomieniu testu za pomocą „Run with coverage”, środowisko wskazało jak dobrze części kodu *GildedRose (Snapshot 0)*, są pokrywane przez test (kolor zielony, żółty i czerwony). Przy pierwszym uruchomieniu, można zauważyć, że duża część kodu nie jest pokrywana przez test (kolor czerwony). Wynika to z tego, że poszczególne linie kodu szukają konkretnych wartości „quality”, „sellIn”, czy chociażby konkretnych nazw przedmiotów. Należy więc test rozszerzyć o te wartości. Przy żółtych liniach IntelliJ podpowiada co dzieje się w danej linii. Dzięki czemu można lepiej zrozumieć, jak poszerzyć obsługę kodu.

```
6 public GildedRose(Item[] items) { this.items = items; }
9
10 public void updateQuality() {
11     for (int i = 0; i < items.length; i++) {
12         if (!items[i].name.equals("Aged Brie"))
13             && !items[i].name.equals("Backstage passes to a TAFKAL80ETC concert")) {
14             if (items[i].quality > 0) {
15                 if (!items[i].name.equals("Sulfuras, Hand of Ragnaros")) {
16                     items[i].quality = items[i].quality - 1;
17                 }
18             }
19         } else {
20             if (items[i].quality < 50) {
21                 items[i].quality = items[i].quality + 1;
22             }
23             if (items[i].name.equals("Backstage passes to a TAFKAL80ETC concert")) {
24                 if (items[i].sellIn < 11) {
25                     if (items[i].quality < 50) {
26                         items[i].quality = items[i].quality + 1;
```

Rys. 2. Program *GildedRose* z zaznaczonymi liniami kodu które pokrywa test

Następnie zmodyfikowano kod testu wprowadzając zmienne znajdujące się w tablicy, jako zmienne przed samym jej wywołaniem. Dzięki czemu można było stworzyć metodę upraszczając program, oraz dodać funkcję „CombinationApprovals.verifyAllCombinations”. Później, rozwinięto test o wspomniane wyżej wartości i nazwy przedmiotów, tak aby test miał jak największe pokrycie. Oczywiście każde zmiany w teście należy zatwierdzić poprzez komendę „move”.

Następnie przetestowano plik *GildedRose (Snapshot 0)* poprzez stworzony test, który nie wykazał żadnych błędów. Kolejnym krokiem było sprawdzenie czy modyfikacja kodu *GildedRose (Snapshot 0)* wpływa na wynik testu. Chwilowa modyfikacja wartości „quality” i „sellIn”, wyrzucało błąd testu, oprócz dwóch miejsc. Dzięki temu można było jeszcze bardziej poszerzyć działanie testu. Gotowy test pokrywa cały kod *GildedRose (Snapshot 0)*, a CombinationApprovals tworzy plik gdzie znajdują się wszystkie możliwe kombinacje nazw i wartości. Wszystkich kombinacji jest ponad 80, a test wykonuje się w około 120 milisekund, co jest dobrą podstawą do rozpoczęcia refaktoryzacji.

```
1 package com.gildedrose;
2
3 import static org.junit.Assert.*;
4
5 import org.approvaltests.Approvals;
6 import org.approvaltests.combinations.CombinationApprovals;
7 import org.junit.Test;
8
9 public class GildedRoseTest {
10
11     @Test
12     public void updateQuality() {
13
14         CombinationApprovals.verifyAllCombinations(
15             this::doUpdateQuality,
16             new String[]{"foo", "Aged Brie", "Backstage passes to a TAFKAL80ETC concert", "Sulfuras, Hand of Ragnaros"},
17             new Integer[]{-1, 0, 2, 6, 11},
18             new Integer[]{0, 1, 49, 50});
19     }
20
21     private String doUpdateQuality(String name, int sellIn, int quality) {
22         Item[] items = new Item[] { new Item(name, sellIn, quality) };
23         GildedRose app = new GildedRose(items);
24         app.updateQuality();
25         return app.items[0].toString();
26     }
27
28 }
```

Rys. 3. Gotowy test *GildedRoseTest*

2. Refaktoryzacja kodu *Gilded Rose*

Celem refaktoryzacji kodu *GildedRose* jest to, aby był bardziej przejrzysty i czytelny dla przyszłych programistów z nim pracujących. Kod należy zmodyfikować tak, aby dodanie kolejnych przedmiotów w sklepie nie sprawiało większych problemów, tak jak w wersji którą dostaje się w spadku od Leerooy'a. Problemem w jego kodzie jest mnóstwo zagnieżdżonych w sobie instrukcji warunkowych if-else. W takiej strukturze kodu, jeden przedmiot jest zależny od drugiego, co sprawia że dodanie kolejnego jest trudne, a na samą myśl o dodaniu jeszcze innych, przychodzi rezygnacja z podjęcia się próby ich dodania. Dlatego należy przedmioty uniezależnić od siebie, tak aby każdy przedmiot był obsługiwany osobno. Dzięki temu dodanie kolejnego przedmiotu i jego obsługi będzie banalnie proste. Stąd tytuł dokumentacji, spoglądając na kod od razu go rozumiem i wiem co mogę z nim zrobić.

Wykorzystano tutaj metodę „lift up conditional”, przedstawioną przez Emily Bache, tutaj:

<https://www.pragma.com/stories/advanced-testing-refactoring-techniques-2/>.

Jest to metoda, która pozwala zgrupować wszystkie instrukcje związane z jednym konkretnym warunkiem. W końcowym efekcie dostaje się instrukcję warunkową if-else if-...- else, której warunkami są nazwy przedmiotów, a instrukcjami - obsługa przedmiotów.

Refaktoryzację rozpoczęto od wprowadzenia zmiennej lokalnej „Item item= items[i]”, po to aby warunki instrukcji if, były bardziej czytelne (zamiast „items[i]”, po prostu „item”). Następnie całą instrukcję warunkową if, znajdującą się pod pętlą for, wyciągnięto z pętli i podpięto pod nową metodę „doUpdateQuality(item)”, którą tylko wywołuje się w pętli for. Kolejne uproszczenie - teraz metoda „updateQuality()” jest pętlą for wywołującą inną metodę.

Głównym zadaniem jest zgrupowanie wszystkich instrukcji związanych z jednym konkretnym warunkiem. Należy skopiować warunek który chcemy „podnieść” (lift up), w tym przypadku jest to pierwszy warunek traktujący o nazwie przedmiotu „Aged Brie”. Następnie zaznaczyć całą instrukcję związaną z tym warunkiem- tutaj jest to prawie cały kod- i wyciągnąć ją do nowej metody o dowolnej nazwie (nie jest to istotne, ponieważ metoda zaraz zniknie). Następnym krokiem jest otoczenie wywoływanej metody instrukcją if- else o warunku, który wcześniej skopiowano. (rys. 4). Następnie za pomocą narzędzia inline ciało nowej metody jest kopiowane w miejsce jej wywoływania.

```
private void doUpdateQuality(Item item) {
    if (item.name.equals("Aged Brie")) {
        foo(item);
    } else {
        foo(item);
    }
}

private void foo(Item item) {
    if (!item.name.equals("Aged Brie")
        && !item.name.equals("Backstage passes to a TAFKAL80ETC concert")) {
        if (item.quality > 0) {
```

Rys. 4. Otoczenie metody „foo(item)” instrukcją if- else o warunku który chce się „podnieść”

Następnie uruchomiono test. Z fragmentu kodu z rys. 5. wynika że część związana z „Sulfuras, Hand of Ragnaros” jest w ogóle nie używana, więc można się jej pozbyć.

```
15 @ private void doUpdateQuality(Item item) {
16     if(item.name.equals("Aged Brie")){
17         if (!item.name.equals("Aged Brie")
18             && !item.name.equals("Backstage passes to a TAFKAL80ETC concert")) {
19             if (item.quality > 0) {
20                 if (!item.name.equals("Sulfuras, Hand of Ragnaros")) {
21                     item.quality = item.quality - 1;
22                 }
23             }
24         } else {
```

Rys. 5. Przeprowadzenie testu po „podniesieniu” instrukcji

Przeglądając odpowiedź do linii kodu zaznaczonej na żółto można zobaczyć, że wartość boolowska tego warunku zawsze będzie „false”. Nic nie stoi na przeszkodzie aby cały długi warunek zastąpić tym wyrażeniem. Korzystając z narzędzia simplify, można pozbyć się również martwego kodu, występującego po warunku „false”, co umożliwia jego odwrócenie na „true” (ponieważ zostaje wyrażenie `if(false)else{}`), a to z kolei umożliwia jeszcze dalsze uproszczenie. Następnie kolejno sprawdzano uwagi przy liniach zaznaczonych kolorem żółtym, związanym ze zwracaniem wartości boolowskich i upraszczano je analogicznie do wcześniej opisanej sytuacji. Po zastosowaniu tych uproszczeń otrzymano jeden warunek traktujący o przedmiocie „Aged Brie” grupujący wszystkie instrukcje z nim związane. Dzięki temu przedmiot „Aged Brie” uniezależnił się w kodzie od pozostałych przedmiotów.

Proces „podnoszenia” warunków zastosowano analogicznie do przedmiotów „Backstage Passes” i „Sulfuras”, powtarzając kolejno kroki wykonane dla przedmiotu „Aged Brie”. Następnie przy kolejnych przedmiotach można uprościć strukturę `else{ if(){} }` (jak jest to pokazane na rys. 6), na formę `else if(){}.`

```

29
30 } else {
31     if (item.name.equals("Backstage passes to a TAFKAL80ETC concert")) {
32         if (item.quality < 50) {
33             item.quality = item.quality + 1;
34
35             if (item.sellIn < 11) {
36                 if (item.quality < 50) {
37                     item.quality = item.quality + 1;
38                 }
39             }
40
41             if (item.sellIn < 6) {
42                 if (item.quality < 50) {
43                     item.quality = item.quality + 1;
44                 }
45             }
46         }
47     }

```

Rys. 6. Obsługa przedmiotu „Backstage Passes”

Dzięki temu krokowi całą instrukcję `if -else if -...-else`, można przekształcić na `switch- case`, gdzie każdym `case`’m będzie nazwa przedmiotu. (rys. 7)

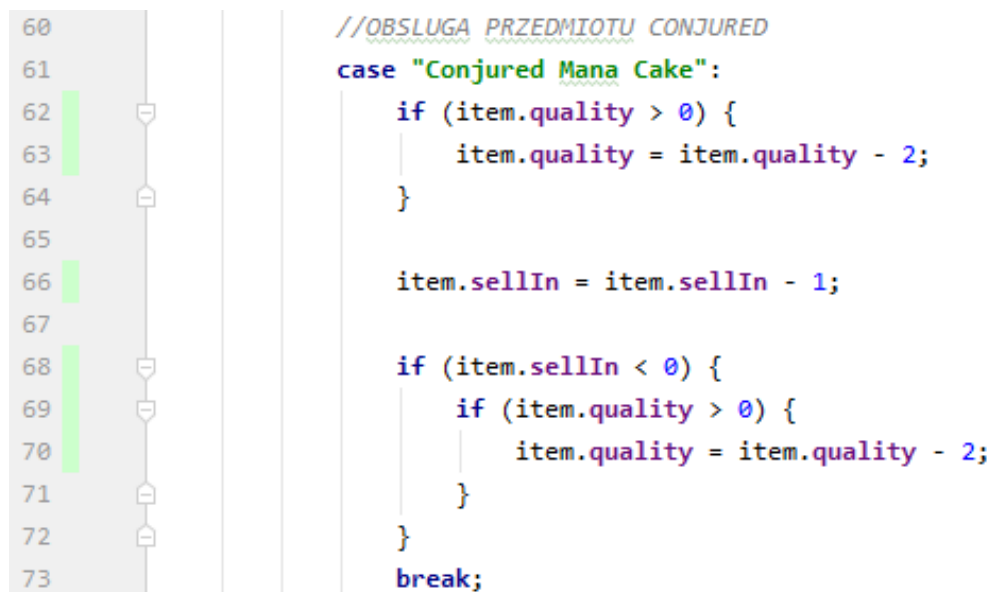
```

16 @ private void doUpdateQuality(Item item) {
17     switch (item.name) {
18         //OBSŁUGA PRZEDMIOTU AGED BRIE
19         case "Aged Brie":
20             if (item.quality < 50) {
21                 item.quality = item.quality + 1;
22             }
23
24             item.sellIn = item.sellIn - 1;
25
26             if (item.sellIn < 0) {
27                 if (item.quality < 50) {
28                     item.quality = item.quality + 1;
29                 }
30             }
31             break;

```

Rys. 7. Zamiana instrukcji `if`, na `switch case`

Dzięki tak przeprowadzonej refaktoryzacji, kod nie jest już tak odpychający jak to miało miejsce przed jego modyfikacją. Każdy przedmiot jest uniezależniony od pozostałych, dzięki czemu jest obsługiwany osobno, a to było głównym celem/ założeniem refaktoryzacji. Oczywiście po każdej zmianie kodu przeprowadzano test *GildedRoseTest*, celem sprawdzenia, czy działanie kodu się nie zmieniło. Teraz gdy kod jest jasny do zrozumienia, można z łatwością dodać obsługę przedmiotu „Conjured” tak jak przedstawiono to na rys. 8.



Rys. 8. Obsługa przedmiotu „Conjured”

Przedmiot ten uwzględniono również w teście *GildedRoseTest*.

Kod można jeszcze bardziej uprościć stosując w niektórych miejscach metodę „decompose conditional”. Polega ona na wyciągnięciu powtarzających się i mniej czytelnych części instrukcji warunkowej, do osobnej metody. Na przykład instrukcję `item.quality = item.quality - 1`, zastąpiono metodą `decreaseQuality()`. Tak oto powstała wersja *GildedRose* (Snapshot 1), a punkt 2 opisuje proces przejścia ze Snapshota 0 do Snapshota 1. Kod jest otwarty na rozszerzenie (jak widać dodanie kolejnego przedmiotu było bardzo proste), oraz jest otwarty na małe zmiany (kolejne instrukcje warunkowe można wyodrębnić jako kolejne metody).

Następnie przeprowadzono test *TexttestFixture*, który symuluje działanie sklepu przez zadaną ilość dni. Aby sprawdzić poprawne działanie zrefaktoryzowanego kodu, zadano test na 100 dni, oraz zwiększono wartość „quality” dla „Conjured Mana Cake” (tylko po to aby sprawdzić czy po upływie „sellIn”, wartość przedmiotu zacznie spadać dwukrotnie).

Jak widać na rysunkach 9, 10, 11, wszystkie przedmioty są prawidłowo obsługiwane. Na rysunku 12, przedstawiającym setny dzień symulacji sklepu widać, że błędy nie wystąpiły, wszystko jest zgodne z założeniami sklepu (przedmioty nigdy nie mają ujemnej wartości, maksymalna wartość przedmiotu to 50, chyba że jest to „Sulfuras” o z góry założonej wartości 80).

Welcome to the shop stranger! ----- day 0 ----- name, sellIn, quality +5 Dexterity Vest, 10, 20 Aged Brie, 2, 0 Elixir of the Mongoose, 5, 7 Sulfuras, Hand of Ragnaros, 0, 80 Sulfuras, Hand of Ragnaros, -1, 80 Backstage passes to a TAFKAL80ETC concert, 15, 20 Backstage passes to a TAFKAL80ETC concert, 10, 49 Backstage passes to a TAFKAL80ETC concert, 5, 49 Conjured Mana Cake, 3, 20	----- day 1 ----- name, sellIn, quality +5 Dexterity Vest, 9, 19 Aged Brie, 1, 1 Elixir of the Mongoose, 4, 6 Sulfuras, Hand of Ragnaros, 0, 80 Sulfuras, Hand of Ragnaros, -1, 80 Backstage passes to a TAFKAL80ETC concert, 14, 21 Backstage passes to a TAFKAL80ETC concert, 9, 50 Backstage passes to a TAFKAL80ETC concert, 4, 50 Conjured Mana Cake, 2, 18	----- day 2 ----- name, sellIn, quality +5 Dexterity Vest, 8, 18 Aged Brie, 0, 2 Elixir of the Mongoose, 3, 5 Sulfuras, Hand of Ragnaros, 0, 80 Sulfuras, Hand of Ragnaros, -1, 80 Backstage passes to a TAFKAL80ETC concert, 13, 22 Backstage passes to a TAFKAL80ETC concert, 8, 50 Backstage passes to a TAFKAL80ETC concert, 3, 50 Conjured Mana Cake, 1, 16
---	--	--

Rys. 9. Poprawne działanie sklepu

----- day 5 -----

name, sellIn, quality

+5 Dexterity Vest, 5, 15

Aged Brie, -3, 8

Elixir of the Mongoose, 0, 2

Sulfuras, Hand of Ragnaros, 0, 80

Sulfuras, Hand of Ragnaros, -1, 80

Backstage passes to a TAFKAL80ETC concert, 10, 25

Backstage passes to a TAFKAL80ETC concert, 5, 50

Backstage passes to a TAFKAL80ETC concert, 0, 50

Conjured Mana Cake, -2, 6

----- day 6 -----

name, sellIn, quality

+5 Dexterity Vest, 4, 14

Aged Brie, -4, 10

Elixir of the Mongoose, -1, 0

Sulfuras, Hand of Ragnaros, 0, 80

Sulfuras, Hand of Ragnaros, -1, 80

Backstage passes to a TAFKAL80ETC concert, 9, 27

Backstage passes to a TAFKAL80ETC concert, 4, 50

Backstage passes to a TAFKAL80ETC concert, -1, 0

Conjured Mana Cake, -3, 2

Rys. 10. Działanie sklepu z zaznaczeniem dodatkowych warunków dla specjalnych przedmiotów

----- day 10 -----

name, sellIn, quality

+5 Dexterity Vest, 0, 10

Aged Brie, -8, 18

Elixir of the Mongoose, -5, 0

Sulfuras, Hand of Ragnaros, 0, 80

Sulfuras, Hand of Ragnaros, -1, 80

Backstage passes to a TAFKAL80ETC concert, 5, 35

Backstage passes to a TAFKAL80ETC concert, 0, 50

Backstage passes to a TAFKAL80ETC concert, -5, 0

Conjured Mana Cake, -7, 0

----- day 11 -----

name, sellIn, quality

+5 Dexterity Vest, -1, 8

Aged Brie, -9, 20

Elixir of the Mongoose, -6, 0

Sulfuras, Hand of Ragnaros, 0, 80

Sulfuras, Hand of Ragnaros, -1, 80

Backstage passes to a TAFKAL80ETC concert, 4, 38

Backstage passes to a TAFKAL80ETC concert, -1, 0

Backstage passes to a TAFKAL80ETC concert, -6, 0

Conjured Mana Cake, -8, 0

Rys. 11. Działanie sklepu z zaznaczeniem dodatkowych warunków dla specjalnych przedmiotów

----- day 99 -----

name, sellIn, quality

+5 Dexterity Vest, -89, 0

Aged Brie, -97, 50

Elixir of the Mongoose, -94, 0

Sulfuras, Hand of Ragnaros, 0, 80

Sulfuras, Hand of Ragnaros, -1, 80

Backstage passes to a TAFKAL80ETC concert, -84, 0

Backstage passes to a TAFKAL80ETC concert, -89, 0

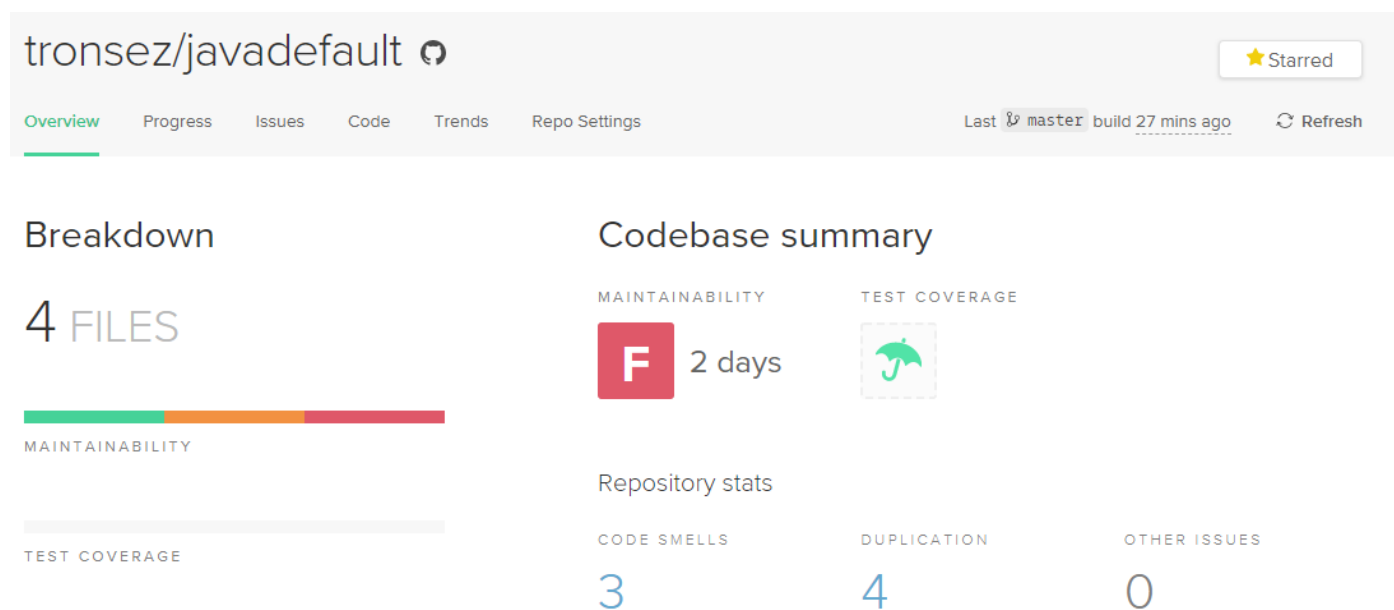
Backstage passes to a TAFKAL80ETC concert, -94, 0

Conjured Mana Cake, -96, 0

Rys. 12. Poprawne działanie sklepu dla 100 dnia symulacji

4. Sprawdzenie „jakości” kodu za pomocą „CodeClimate”

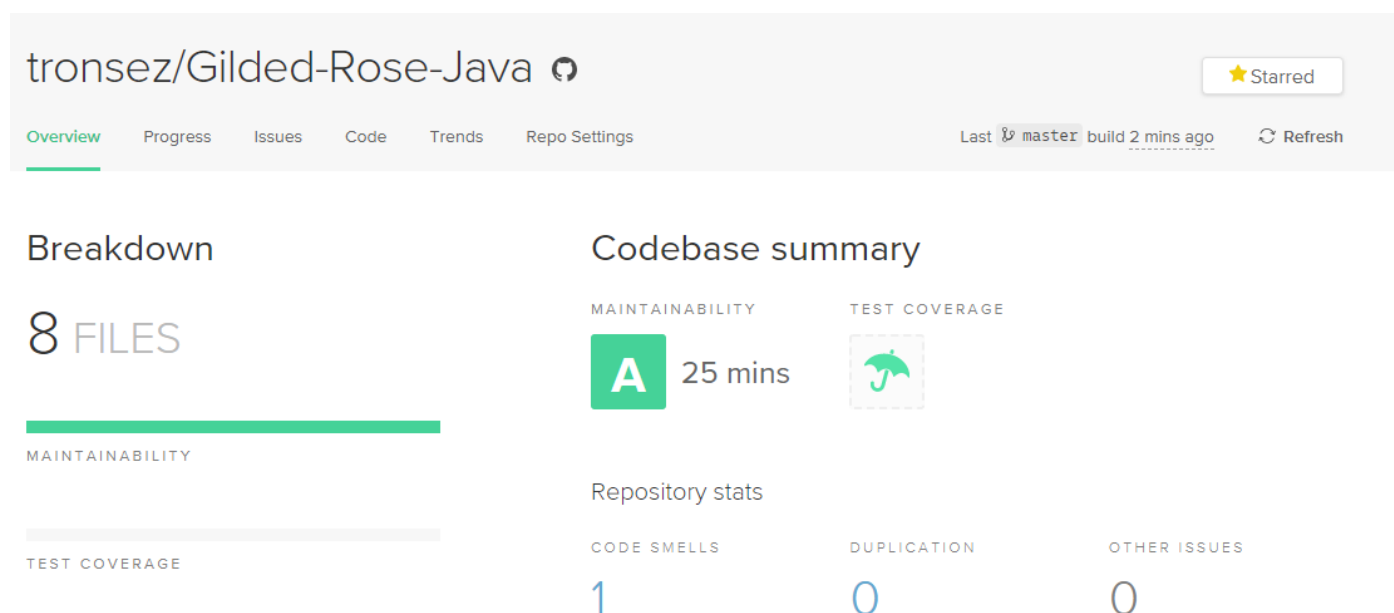
Po przesłaniu plików do repozytorium, zostało ono przeskanowane przez CodeClimate. Wyniki przedstawiono na rysunku 13.



Rys. 13. Pierwszy test CodeClimate (Snapshot 0)

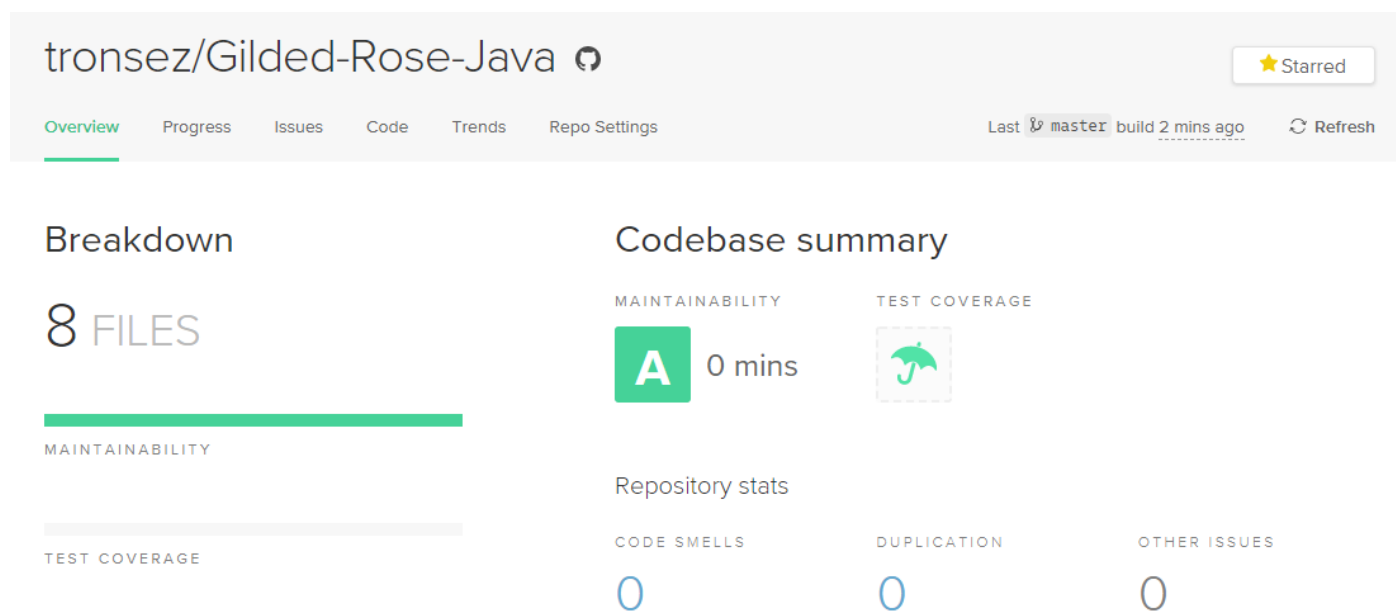
Jak widać program przed poprawkami otrzymał bardzo niską notę. Występują duplikacje kodu oraz Code Smells, czyli sygnały do refaktoryzacji. Ocena obecnego programu, to Cognitive Complexity na poziomie 60, oraz uwaga ostrzegająca o zbyt długiej metodzie `doUpdateQuality()` (około 60 linijek).

Następnie przystąpiono do poprawy tego problemu. Poczyniono refaktoryzację opisaną w punkcie 2. Dzięki tym zabiegom po przeprowadzeniu kolejnego testu, otrzymano wyniki przedstawione na rys. 14.



Rys. 14. Drugi test CodeClimate (Snapshot 1)

Jak widać wyniki znacząco się poprawiły i kod osiągnął najwyższą ocenę. Jednak został jeszcze jeden Code Smell. Instrukcje warunkowe z każdego case'ów wyciągnięto jako osobne metody, dzięki czemu pozbyto się ostrzeżenia o zbyt długiej metodzie doUpdateQuality(). Metoda odpowiedzialna za przedmiot „Backstage Passes”, me Cognitive Complexity na poziomie 7. Można to poprawić wyodrębniając metodę stay0(), która zeruje wartość przedmiotu, po upływie sellIn. Tak oto powstał kod GildedRose (Snapshot 2) Po wykonaniu tego kroku wyniki są następujące (rys. 15.).



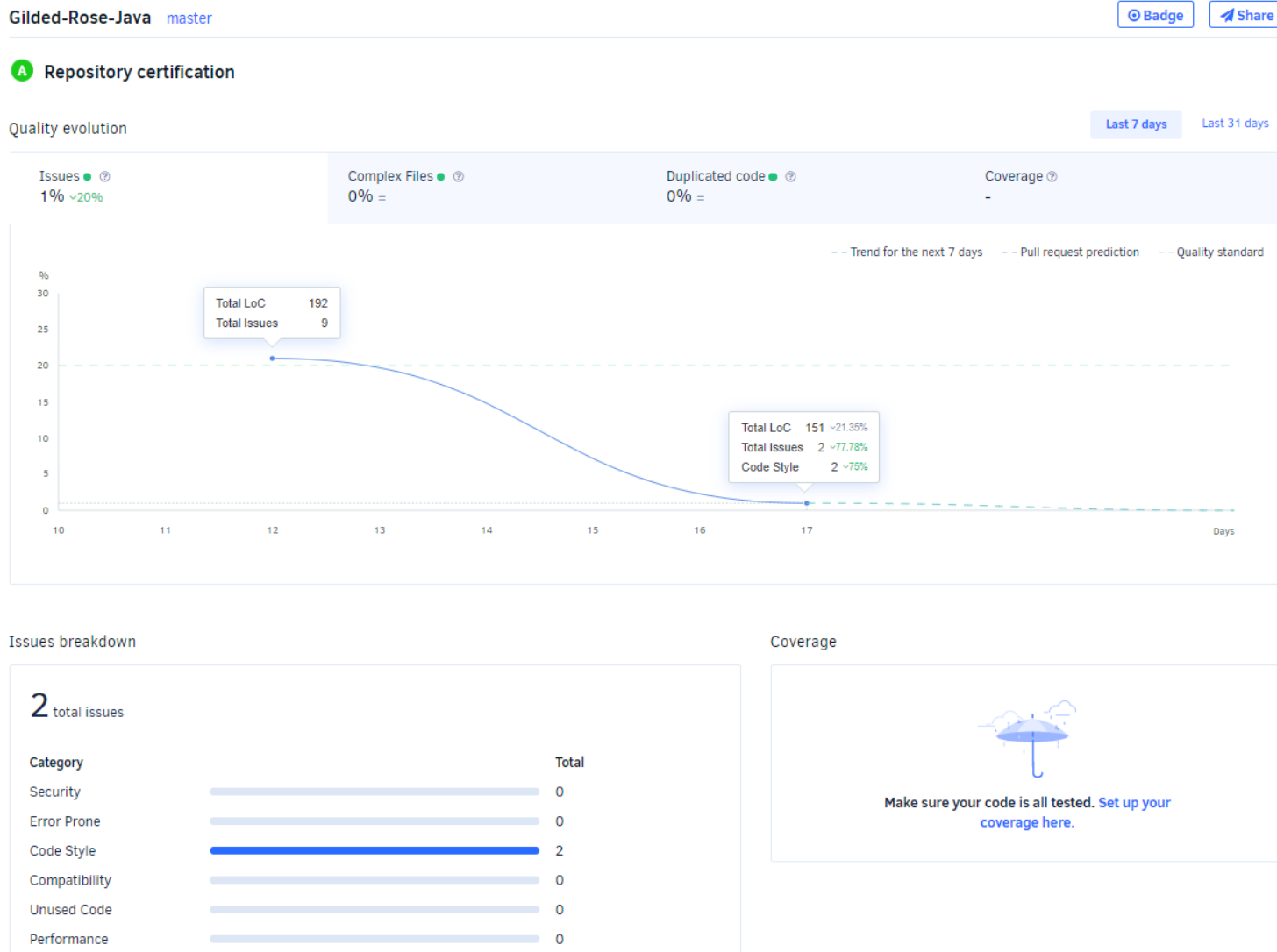
Rys. 15. Ostatnia ocena kodu przez CodeClimate (Snapshot 2)

Jak widać ocena kodu jest najwyższa, oraz nie trzeba wprowadzać już żadnych poprawek.

Jest to jednak ocena CodeClimate. Jeśli chciałoby się poruszyć temat oceny subiektywnej, to moim zdaniem, kod jest bardziej przejrzysty dla kolejnego użytkownika, w Snapshocie 1. Każdy przedmiot jest obsługiwany osobno i jest to dobrze widoczne. Po wprowadzeniu zmian, należy znać nowe metody z których korzysta się do obsługi przedmiotów. Dodanie nowego nie będzie trudne, ale w porównaniu do kodu sprzed zmian może zająć to trochę dłużej.

5. Sprawdzanie „jakości” kodu za pomocą „codacy”.

Przeprowadzono kolejny test za pomocą innego narzędzia. Poniższy wykres przedstawia różnicę pomiędzy Snapshotem 0 a Snapshotem 1. Jest to wykres ilustrujący poziom problemów (w procentach) kodu, na przestrzeni kolejnych poprawek.



Rys. 16. Wykres zależności procentowej wartości problemów w kodzie na przestrzeni kolejnych snapshotów

Pierwszym punktem wykresu jest Snapshot 0, gdzie całkowity poziom błędów jest na poziomie 21%, a samo narzędzie ocenia program na „C”. Drugi punkt to Snapshot 1 czyli kod zrefaktoryzowany za pomocą metody „lift up conditional”. Jak widać całkowity poziom problemów spadł do 1%, czyli nastąpiła różnica aż 20%. Poprawił się styl kodu a także zmniejszono liczbę problemów. Program oceniony jest na „A”, czyli najwyższą ocenę. Wykres przedstawia tylko ocenę Snapshotów 0 i 1. Nie zamieszczono tutaj oceny Snapshotu 2, ponieważ według subiektywnej oceny, ta wersja programu jest mniej czytelna dla użytkownika niż Snapshot 1.