Gilded Rose

Projekt sklepu z przedmiotami - refactoring na przykładzie języka Scala

Dawid Bińkuś, 246793

29 listopada 2019

1 Wstęp

1.1 Źródła

Przykład refactoryzacji w języku Scala został wykonany na podstawie projektu GildedRose.

- Oryginalna treść projektu: https://github.com/emilybache/GildedRose-Refactoring-Kata
- Rozwiązanie zadania zaimplementowane w języku Scala https://github.com/inql/zjp-assignment

1.2 Narzędzia wykorzystane do dokonania refactoryzacji kodu

W celu rozwiązania zadania zostały wykorzystane następujące narzędzia:

- Scalatest, wykorzystywany do implementacji testów jednostkowych: http://www.scalatest.org/
- Scoverage, dodatek wykorzystywany do określenia pokrycia kodu przez testy: https://github.com/scoverage/sbt-scoverage
- Scapegoat, dodatek służący do określenia jakości kodu: https://github.com/sksamuel/scapegoat
- SonarQube, narzędzie używane do analizy projektu wraz z dodatkiem *sbt-sonar* wykorzystywanym do integracji w/w narzędzia z projektem: https://github.com/mwz/sbt-sonar

1.3 Omówienie kodu początkowego

1.3.1 Struktura klas

Początkowa implementacja GildedRose składa się z dwóch klas:

- **Item** - Klasa określająca strukturę przedmiotów w systemie - posiadaja ona informacje o nazwie przedmiotu, jakości oraz terminie w którym dany przedmiot ma być sprzedany.
- **GildedRose** - Klasa przechowująca logikę systemu GildedRose - zawiera tablicę przedmiotów obecnych w systemie. Dodatkowo, zawiera metodę umożliwiającą zaktualizować stan wszystkich przedmiotów.

1.3.2 Omówienie logiki systemu GildedRose

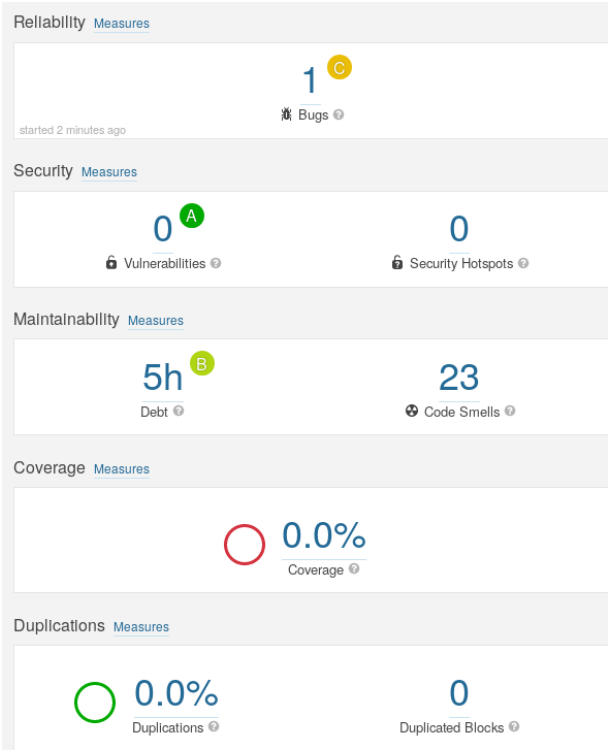
Zamysł logiki zaimplementowanej do systemu prezentuje się w sposób następujący:

1. Sprawdzenie typu przedmiotu bazując na jego nazwie.
2. Zmienienie jakości przedmiotu bazując na jego typie.
3. Obniżenie terminu sprzedaży na przedmiocie.

W zależności od typu przedmiotu mogą wystąpić dodatkowe efekty podczas aktualizacji danych, np. dla przedmiotu typu *Aged Brie* jakość rośnie im dłużej jest magazynowany (tj. wartość terminu sprzedaży jest mniejsza niż zero), a właściwości przedmiotu typu *Legendary* są stałe niezależnie od czasu spędzonego w systemie.

1.3.3 Omówienie początkowej jakości kodu

Początkowa logika systemu została napisany w sposób bardzo prosty a zarazem skomplikowany do zrozumienia. Cała logika opiera się na iteracji po poszczególnych przedmiotach w systemie a następnie zastosowaniu zmian zdefiniowanych w dokumentacji. Realnym problemem okazuje się sposób w jaki wykonywana jest ta czynność. Cała logika opiera się na skomplikowanej strukturze wyrażeń warunkowych, które rozgałęziają się i zagnieżdżają ze sobą co znacznie zwiększa złożoność implementacji.
Po początkowym skanowanie za pomocą narzędzia SonarQube kwestia jakości kodu została wyjaśniona:



Bazując na rezultatach skanowania, znaczącym błędem mogącym prowadzić do nieprawidłowości było zerowanie wartości *quality* produktu poprzez wykorzystanie danego wyrażenia:

```
items(i).quality = items(i).quality - items(i).quality
```

Następnym obszarem wymagającym poprawy jest ogólnopojęta jakość kodu - SonarQube wykrył nieprawidłowości w postaci 23 *Code Smells*. Dotyczyły one głównie braku odpowiedniego udokumentowania dla implementacji, zagnieżdżonych wyrażeń warunkowych oraz złożoności, utrudniającej zrozumienie instrukcji wykonywanych w programie.

2 Omówienie procesu refactoryzacji

W celu rozpoczęcia procesu refactoryzacji niezbędne było określenie jakie kolejne kroki należy podjąć by osiągnąć postawiony cel. Rozwój projektu podzieliłem na następujące etapy:

1. Implementacja testów jednostkowych - zgodnie z zasadą **TDD**, chcemy mieć ciągłą kontrolę na poprawnością implementacji - w początkowej wersji projektu, testy nie były zaimplementowane.

2. Uproszczenie głównej funkcji odpowiedzialnej za logikę projektu - by uczynić kod bardziej czytelnym i mniej skomplikowanym, należy całkowicie zmienić jego strukturę działania. Etap jest ten również przygotowaniem do kolejnej fazy.

3. Implementacja nowej funkcjonalności - treść zadania wymaga dodania obsługi przedmiotów typu *conjured* - po uproszczeniu logiki aplikacji zadanie to powinno być znacznie prostsze.

2.1 Implementacja testów

Do implementacji testów wykorzystana została biblioteka *Scalatest* - umożliwia ona pisanie testów w prosty sposób a wykonywane instrukcje powinny być zrozumiane również przez osobę nietechniczną. Dostarczona implementacja testów sprawdzała nieistotną w kontekście aplikacji funkcjonalność - możliwość zmiany przedmiotu po zaktualizowaniu wartości. Początkowy kod prezentował się w sposób następujący:

```
class GildedRoseTest  extends FlatSpec with Matchers {
      it should "foo" in {
        var items = Array[Item](new Item("foo", 0, 0))
        val app = new GildedRose(items)
        app.updateQuality()
        (app.items(0).name) should equal ("fixme")
      }
}
```

W celu napisania nowej implementacji testów, pozbyto się całkowicie starej implementacji i zastosowano nową, wykorzystującą moduły *FunSpec* oraz *BeforeAndAfterEach*. Pierwszy z nich posłużył nam do ustruktoryzowania procedur testowych (testy napisane w ten sposób można podzielić według testowanego obszaru i konkretnego zachowania w myśl zasday **BDD**). Drugi zaś umożliwił zaimplementowanie metody *beforeEach*, która przygotowywała stan systemu przed wykonaniem operacji określonych w teście.
Początkowa definicja testów prezentuje się w sposób następujący:

```
class GildedRoseTest extends FunSpec with BeforeAndAfterEach {

  var regular, agedBrie, sulfuras, backstage: Item = _
  var gildedRose: GildedRose = _

  override def beforeEach(): Unit = {
    regular = new Item("regular", 7, 7)
    agedBrie = new Item("Aged Brie", 15, 15)
    sulfuras = new Item("Sulfuras, Hand of Ragnaros", 30,80)
    backstage = new Item("Backstage passes to a TAFKAL80ETC concert", 11, 9)

    gildedRose = new GildedRose(Array(regular,agedBrie,sulfuras,backstage))
  }
```

Po przygotowaniu systemu do wykonywania testów poprzez wstrzyknięcie mu niezbednych danych, należało przejść do implementacji zachowań do sprawdzenia. Za pomocą instrukcji *describe* określony został obszar testów (system GildedRose), następnie za pomocą instrukcji *it* zdefiniowane zostały testy bazujące na określonych oczekiwanych zachowaniach systemu. Przykładowe testy prezentują się w sposób następujący:

```
  describe("GildedRose"){

    it("should degrade item's quality by one when the day ends"){
      gildedRose.updateQuality()
      assertResult(6)(regular.quality)
    }

    it("should reduce item's sellIn by one then the day ends"){
      gildedRose.updateQuality()
      assertResult(6)(regular.sellIn)
    }
```

```
it("should degrade item's quality twice when sellIn reached zero"){
  regular.sellIn = 0
  gildedRose.updateQuality()
  assertResult(5)(regular.quality)
}

it("should item's quality never reach zero"){
  regular.quality = 0
  gildedRose.updateQuality()
  assertResult(0)(regular.quality)
}
```

Po przygotowaniu wszystkich testów, należało przystąpić do wykonania testowego uruchomienia. Wszystkie testy były uruchamiane zarówno lokalnie jak i z użyciem kontenera tworzonego na witrynie github.com - repozytorium projektu zostało skonfigurowane w taki sposób by każda aktualizacja wiązała się z wykonaniem całej sekwencji testowej.

```
[info] GildedRoseTest:
[info] GildedRose
[info] - should degrade item's quality by one when the day ends
[info] - should reduce item's sellIn by one then the day ends
[info] - should degrade item's quality twice when sellIn reached zero
[info] - should item's quality never reach zero
[info] - should increase aged brie quality when the day ends
[info] - should increase aged brie quality twice when sellIn value is lower than zero
[info] - should not increase item quality then it's 50
[info] - should not decrease sulfuras quality or sellIn value
[info] - should increase backstage quality when sellIn value is (10,inf]
[info] - should increase backstage quality by 2 when sellIn value is greater than (5,10]
[info] - should increase backstage quality by 3 when sellIn value is greater than (0,5]
[info] - should not increase backstage quality when sellIn value is zero
[info] Run completed in 812 milliseconds.
[info] Total number of tests run: 12
[info] Suites: completed 1, aborted 0
[info] Tests: succeeded 12, failed 0, canceled 0, ignored 0, pending 0
[info] All tests passed.
[success] Total time: 19 s, completed Nov 24, 2019 8:13:14 PM
```

Dzięki zaimplementowaniu testów jednostkowych, upewniliśmy się, że implementacja systemu mimo wątpliwej jakości, działa tak jak zakładaliśmy. Umożliwia to przejście do kolejnego etapu refactoryzacji. Dodatkowo, osiągnięte zostało pokrycie kodu w postaci 100% instrukcji zawartych w systemie GildedRose

2.2 Upraszczanie kodu

Implementacja systemu składająca się z szeregu zagnieżdżonych zapytań *if* znacząco utrudniała zrozumienie kodu i znacznie ograniczała jego modularność oraz możliwość rozwinięcia jego zadań. By dokonać prawidłowego dokonania refactoryzacji kodu, należało dokładnie go przeanalizować a następnie podzielić go na mniejsze części by całość rozwiązania była bardziej spójna. W przypadku projektu GildedRose, pierwszym krokiem było wydzielenie funkcji odpowiedzialnej za aktualizację jakości produktu, ponieważ to właśnie ta operacja była najbardziej narażona na nieprawidłowe zmiany. Treść nowo dodanej funkcji prezentowała się następująco:

```
def setMaxQuality(item: Item, updateValue: Int, maxQuality: Int): Item = {
  item.quality += updateValue
  item.quality = item.quality min maxQuality
  item
}
```

Udało nam się już wydzielić część logiki z systemu GildedRose. Następnym krokiem było użycie go we właściwej już implementacji. Jest to również idealny moment by zmienić całą strukturę decydowania o aktualizacji stanu przedmiotu. Najrozsądniejszym rozwiązaniem okazało się reimpementacja klasy *Item* na tzw. *klasę przypadku* - specjalny rodzaj klasy, stworzony do kreowania takich obiektów jak np. przedmioty w bazie danych. Umożliwia on również zastosowanie tzw. zastosowania wzorca (ang. pattern matching) by stworzyć implementację nieopartą o zagnieżdżone zapytania if - co za tym idzie - ze znacznie zmniejszoną złożonością.

```
case class Item(val name: String, var sellIn: Int, var quality: Int)
```

Po aktualizacji pliku GildedRose.scala, implementacja prezentuje się następująco:

```
def updateQuality(): Unit = {
  items.foreach {
    case Item("Sulfuras, Hand of Ragnaros", _, _) =>
    case item@Item("Aged Brie", sellIn, quality) if quality <= 50 & sellIn >= 0 =
      item.sellIn -= 1
      setMaxQuality(item, 1, 50)
    case item@Item("Aged Brie", sellIn, quality) if quality <= 50 & sellIn < 0 =>
      item.sellIn -= 1
      setMaxQuality(item, 2, 50)
    case item@Item("Backstage passes to a TAFKAL80ETC concert", sellIn, quality)
      item.sellIn -= 1
      setMaxQuality(item, 1, 50)
    case item@Item("Backstage passes to a TAFKAL80ETC concert", sellIn, quality)
      item.sellIn -= 1
      setMaxQuality(item, 2, 50)
    case item@Item("Backstage passes to a TAFKAL80ETC concert", sellIn, quality)
      item.sellIn -= 1
      setMaxQuality(item, 3, 50)
    case item@Item("Backstage passes to a TAFKAL80ETC concert", sellIn, _) if sel
      item.sellIn -= 1
      item.quality = 0
    case item@Item(_, sellIn, quality) if sellIn > 0 && quality > 0 =>
      item.sellIn -= 1
      item.quality -= 1
    case item@Item(_, sellIn, quality) if sellIn <= 0 && quality > 0 =>
      item.sellIn -= 1
      item.quality -= 2
    case _ =>
  }
}
```

Dzięki zastosowaniu pattern matching, jakość i struktura kodu została znacznie usprawniona, jednocześnie pozostawiając implementację niezmienną. Ostatnim krokiem było wydzielenie zmiennych stałych dla nazw przedmiotu, by ułatwić przyszłą pracę z kodem źródłowym systemu.

2.3 Implementacja nowej funkcjonalności

Następnym etapem refactoryzacji kodu było dodanie nowej funkcjonalności zdefiniowanej w treści zadania. Zakładała ona dodanie nowego typu przedmiotu: *conjured*, którego jakość spada każdego dnia o 2. W przypadku gdy termin sprzedaży minął, wartość ta spada dwukrotnie. Pracę nad dodaniem implementacji rozpoczęto od zdefiniowania podstawowych testów jednostkowych - w myśl zasady **TDD** oraz **BDD**.

3 Implementacja paradygmatu funkcyjnego do istniejącego kodu obiektowego oraz dalsze usprawnienia

3.1 Wydzielenie logiki kodu

3.2 Funkcja jako wyrażenie