Gilded Rose

Projekt sklepu z przedmiotami - refactoring na przykładzie języka Scala

Dawid Bińkuś, 246793

30 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
- Materiał na temat refactoryzacji kodu https://www.youtube.com/watch?v=8bZh5LMaSmE
- Materiały dotyczące clean code specyficzne dla Scali https://github.com/alexandru/scala-best-practices https://medium.com/bigpanda-engineering/refactoring-to-functional-patterns-in-scala-bf81e7abfe77

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

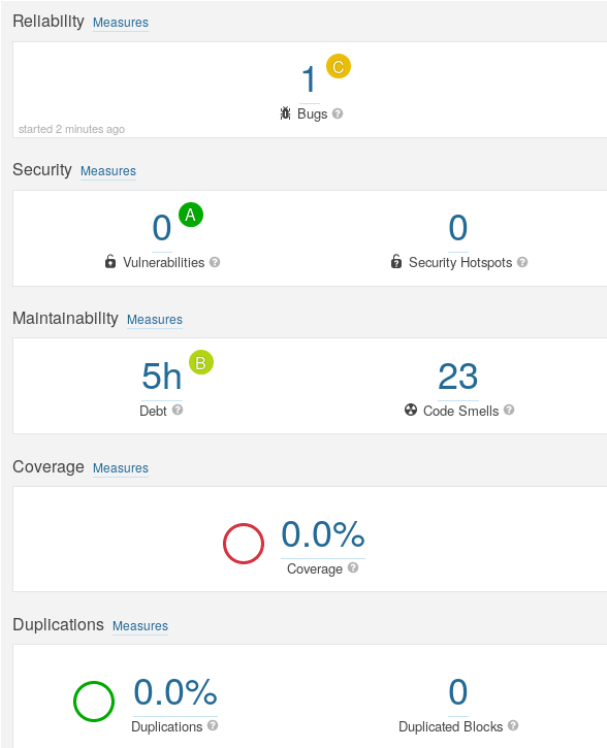
Zamysł logiki zaimplementowanej do systemu prezentuje się w sposób następujący:

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**.

```
it("should degrade conjured quality by two when the day ends"){
  gildedRose.updateQuality()
  assertResult(10)(conjured.quality)
}

it("should degrade conjured quality twice as normal if sellIn <=0"){
  conjured.sellIn = 0
  gildedRose.updateQuality()
  assertResult(8)(conjured.quality)
}
```

Dzięki refactoryzacji kodu, dodanie nowej funkcjonalności okazało się bardzo proste. Wystarczyło dodać kolejne instrukcje do głównej funkcji *UpdateQuality*.

```
...
case item@Item(Conjured, sellIn, quality) if quality > 0 && sellIn > 0 =>
    item.sellIn -= 1
    item.quality -=2
case item@Item(Conjured, sellIn, quality) if quality > 0 && sellIn <= 0 =>
    item.sellIn -= 1
    item.quality -= 4
...
```

Po dokonaniu implementacji, pozostało nam dodać kilka komentarzy w plikach źródłowych projektu by był on bardziej zrozumiały dla innych osób z nim pracujących, np.

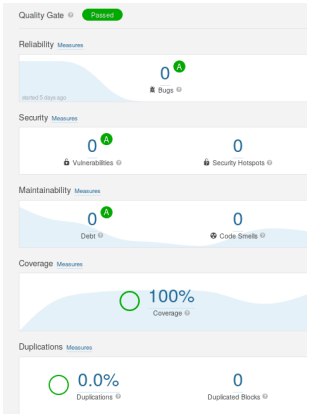
```
  /**
  * A function to avoid exceeding maximum quality of given item.
  * @param item Item to be checked
  * @param updateValue Value to be added to the item's quality.
  * @param maxQuality Maximum quality for given item.
  * @return
  */
 def setMaxQuality(item: Item, updateValue: Int, maxQuality: Int): Item = {
   item.quality += updateValue
   item.quality = item.quality min maxQuality
   item
 }
```

2.4 Podsumowanie

Teraz, gdy dodatkowa funkcjonalność została zaimplementowana, należy podsumować to co udało nam się osiągnąć podczas procesu refactoryzacji projektu GildedRose.
Jak można zauważyć wszystkie znaczące błędy oraz *code smells* zostały naprawione. Dodatkowo pokrycie kodu przez testy jednostkowe pozostało niezmienione - znajduje się na poziomie 100%.

3 Dalsze usprawnienia

Kod otrzymany w wyniku wykonanych zmian jest poprawny, aczkolwiek daleko mu do ideału. Brakuje mu skalowalności - każda nowa funkcjonalność wymaga zmiany głównej funkcji programu, co może powodować do błędów w przyszłości. Dodatkowo nie spełnia on warunków paradygmatów obiektowych oraz funkcjonalnych - Scala jest językiem który potrafi pogodzić te dwa paradygmaty,



dlatego też w tym rozdziale zajmiemy się dalszymi usprawnieniami projektu GildedRose. Składają się one na:

1. Wydzielenie logiki kodu za pomocą ekstrakcję zachowań poszczególnych przedmiotów.
2. Umożliwienie sprawne tworzenie nowych zachowań z zachowaniem domyślnych właściwości nowo utworzonych przedmiotów.
3. Reimplementacja głównej logiki by stała się niezależna i odporna na potencjalne zmiany.

3.1 Wydzielenie logiki kodu

Podczas refactoryzacji kodu zauważalne stały się zależności między zachowaniami poszczególnych przedmiotów. Podczas operacji aktualizacji danych nadpisywane są dwa z trzech pól - termin sprzedaży oraz jakość obiektu. Dodatkowo większość z nich współdzieli implementację niektórych zachowań.
Te obserwacje umożliwiły podzielenie zachowań według następującego wzorca:

1. Zachowanie bazowe, które definiuje sposób aktualizacji pól obiektów.
2. Zachowania dziedziczące po bazowym, które mogą być całkowicie zależne od bazowego, zachowywać się w sposób niezależny bądź cześciowo implementować zachowanie bazowe.

W późniejszych podrozdziałach podjęty zostanie temat dalszej pracy nad systemem GildedRose

3.1.1 Potencjał dziedziczenia zachowań przedmiotów

Zachowanie bazowe zostało zaimplementowane za pomocą tzw. *klasy abstrakcyjnej* w której zdefiniowane zostały operacje współdzielone przez poszczególne przedmioty.

```
/**
* Abstract class which create behaviour based on the item name.
* @param item Item to create a behaviour for.
*/
abstract class Behaviour(item: Item) {

  val maxQuality = 50
  val minQuality = 0

  def update(): Unit = {
    updateQualityValue()
    updateSellInValue()

    item.quality = if (item.quality > maxQuality)  maxQuality
      else if (item.quality < minQuality) minQuality
```

```
        else item.quality
  }


  protected def updateQualityValue(): Unit

  protected def updateSellInValue(): Unit = item.sellIn -= 1

}
```

Następnie, utworzone zostały klasy poszczególnych przedmiotów, które dziedziczyły po klasie abstrakcyjnej *Behaviour*. W zależności od potrzeb, dane podklasy nadpisywały zachowania swojego rodzica bądź czerpały implementację bezpośrednio od niego.

```
 class DefaultBehaviour(item: Item) extends Behaviour(item){
  override protected def updateQualityValue(): Unit = {
    item.quality = if (item.sellIn <= 0) item.quality - 2 else item.quality - 1
  }
}

class BackstageBehaviour(item: Item) extends Behaviour(item){
  override protected def updateQualityValue(): Unit = {
    item.quality = if (item.sellIn < 6) item.quality + 3
      else if (item.sellIn < 11) item.quality + 2
        else item.quality + 1
  }

  override protected def updateSellInValue(): Unit = {
    super.updateSellInValue()
    item.quality = if (item.sellIn < 0) 0 else item.quality
  }
}
```

Warto nadmienić tutaj fakt, że same klasy zachowań zostały napisane w taki sposób by był swoistym opakowaniem dla poszczególnych przedmiotów. Same w sobie nie zwracają żadnych wartości, jedyne co wykonują to pobieranie przedmiotu w konstruktorze oraz dokonywanie na nim niezbędnych operacji. Dodatkowo, wykorzystana została tutaj jedna z cech języka programowania Scala - wyrażenia traktowane w językach obiektowych jako imperatywne (takie jak np. **for, if**, tutaj są funkcjami i zwracają wartość. Dzięki tej właściwości wyrażenie

```
item.quality = if (item.sellIn < 0) 0 else item.quality
```

jest akceptowane przez maszynę wirtualną Javy - w efekcie przypisuje do pola żądaną wartość w zależności od danego warunku.

3.1.2 Wzorzec projektowy fabryka

Kiedy poszczególne zachowania zostały już zdefiniowane pozostaje kolejny problem do rozwiązania - przypisywanie przedmiotów do poszczególnych wzorców. W tym celu utworzony został obiekt *ItemFactory* wraz ze zdefiniowaną funkcją *update*. Jej zadaniem jest zdefiniowanie zachowania na podstawie jego nazwy. Implementacja prezentuje się w sposób następujący:

```
 object ItemFactory {
  def create(item: Item): Behaviour = item.name match {
    case "Sulfuras, Hand of Ragnaros" => new LegendaryItemBehaviour(item)
    case "Aged Brie" => new AgedBrieBehaviour(item)
    case "Backstage passes to a TAFKAL80ETC concert" => new BackstageBehaviour(item
    case "conjured" => new ConjuredBehaviour(item)
```

```
      case _ => new DefaultBehaviour(item)
    }
}
```

Do rozpatrzenia odpowiedniego zachowania dla danego przedmiotu zastosowane zostało dopasowanie wzorca.

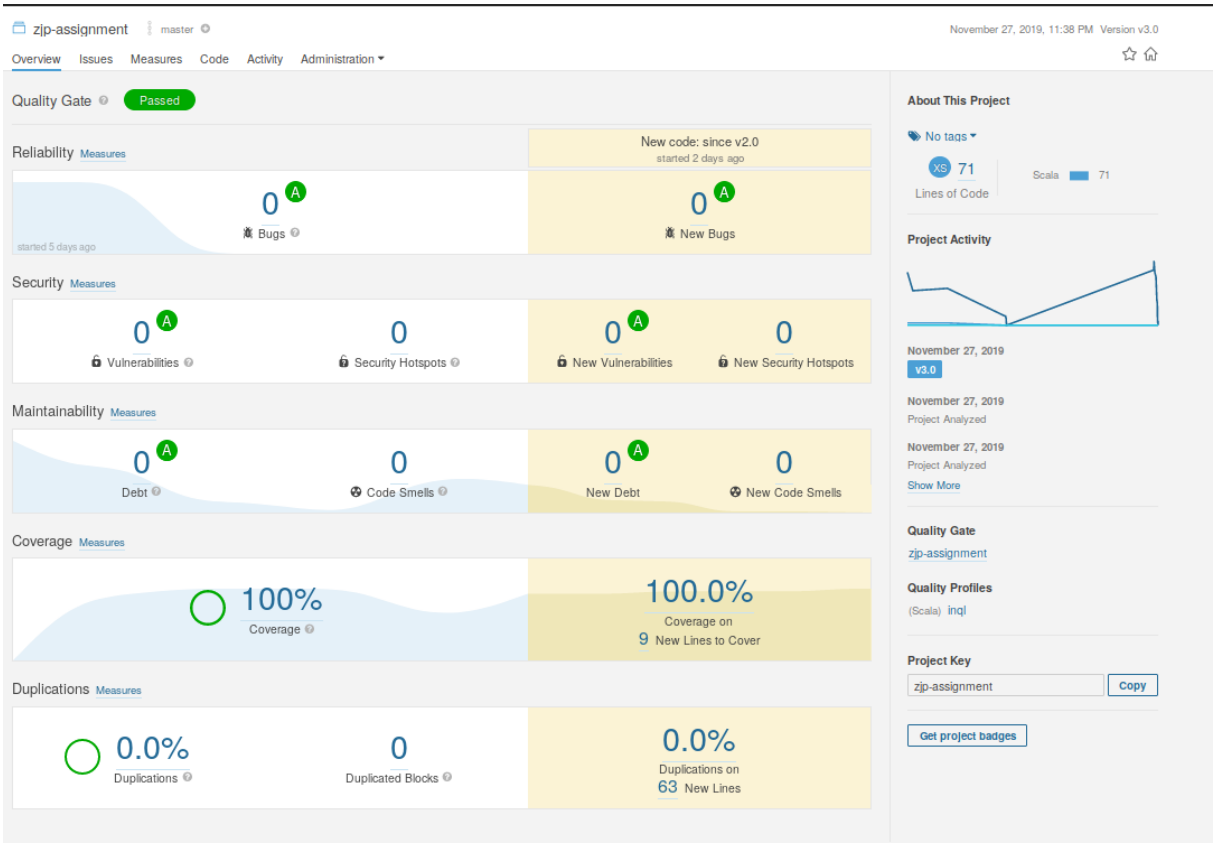
3.1.3 Finalizacja

W celu wykorzystania nowo utworzonej implementacji zachowań, należało zmienić główną funkcję obsługującą logikę GildedRose. Dotychczasowa iteracja oraz dopasowanie wzorca nie jest już potrzebne, dlatego cała obsługa przedmiotów w systemie została skrócona do czysto funkcyjnej postaci:

```
class GildedRose(val items: Array[Item]) {
  /**
   * A function which maps all given items into Behaviour wrapper and updates its
   */
  def updateQuality(): Unit = {
    items.map(ItemFactory.create).foreach(_.update())
  }
}
```

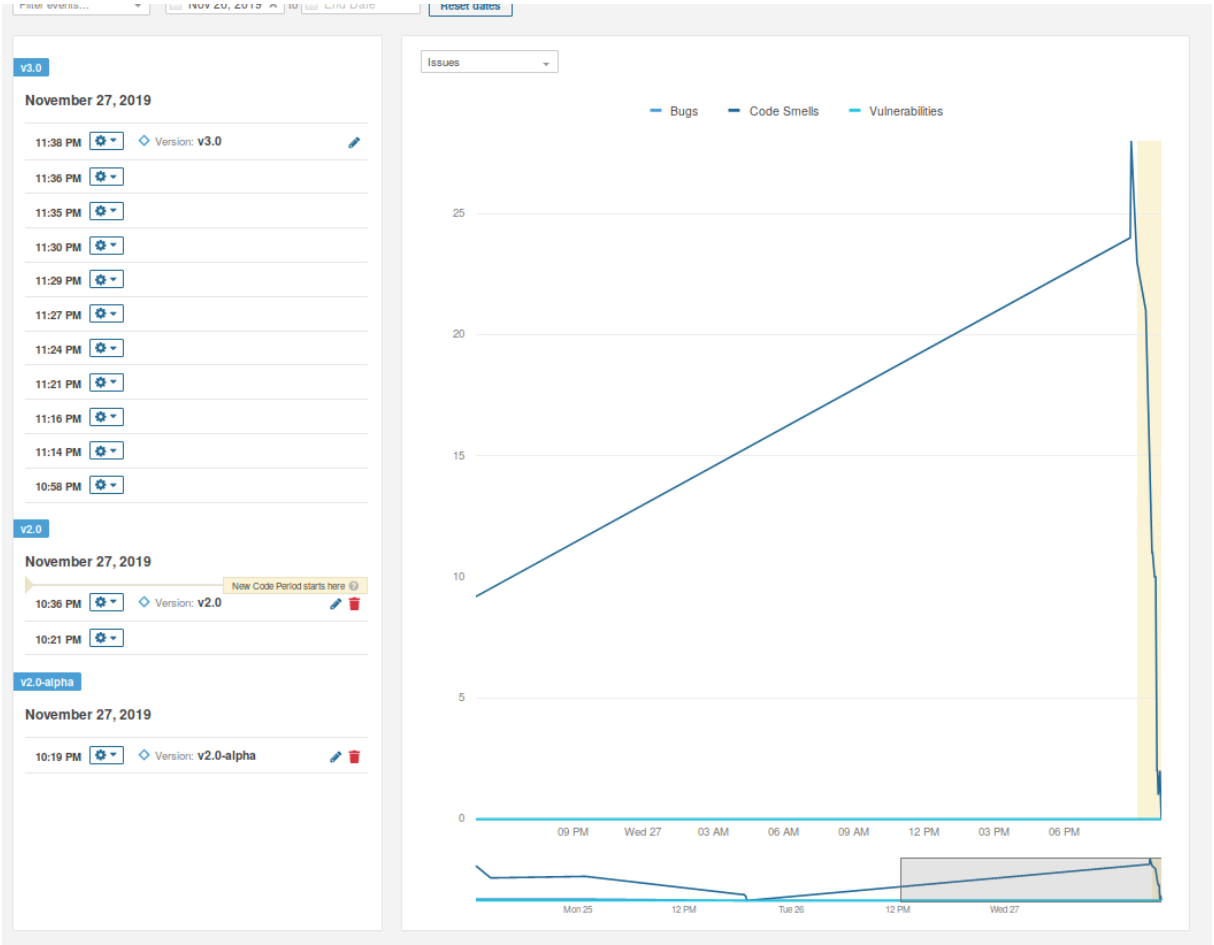
4 Podsumowanie

Finalna analiza projektu za pomocą narzędzia SonarQube prezentuje się w sposób następujący:

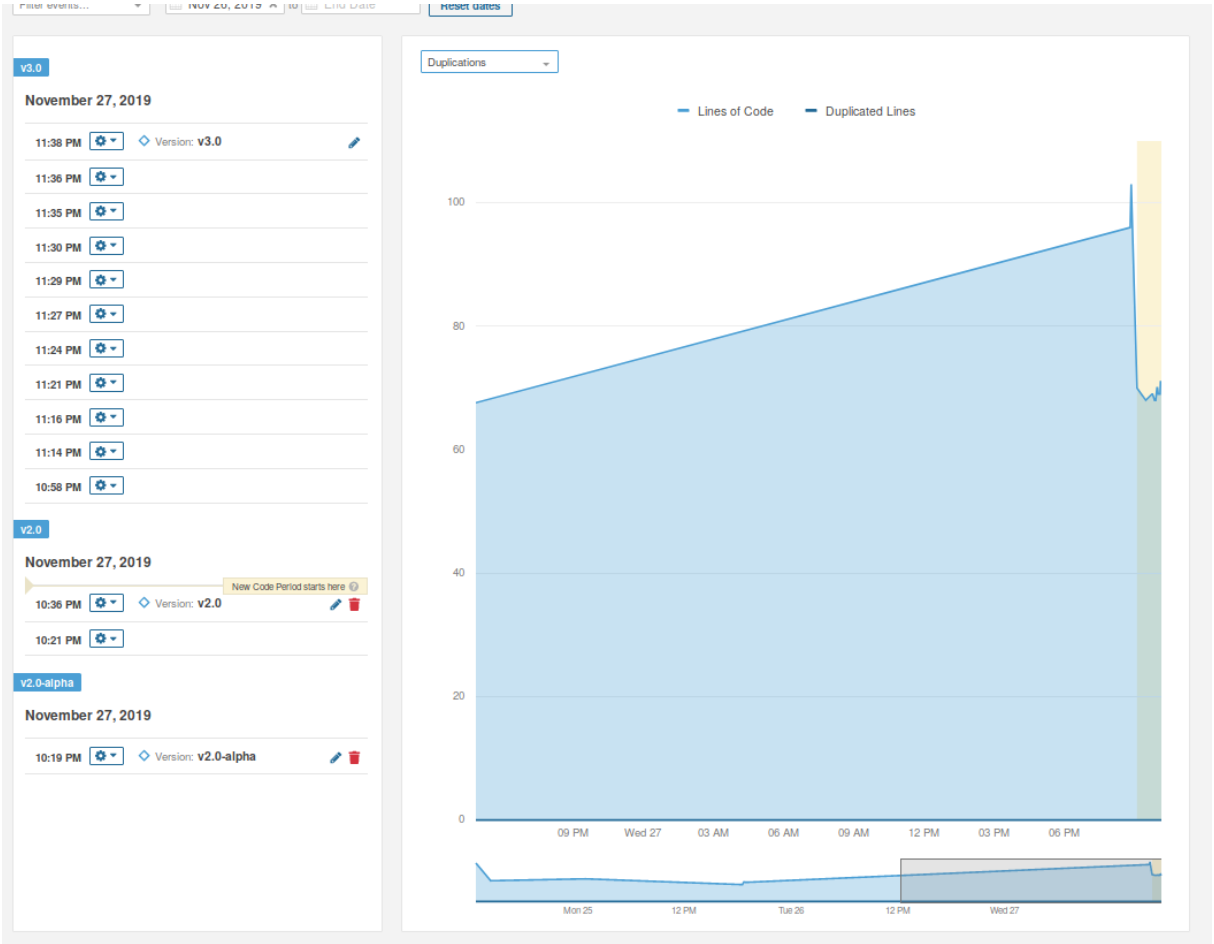


Jak widać na załączonym raporcie, wszystkie początkowe problemy związane zarówno z poprawnością implementacji jak i jej jakością zostały naprawione. Testy zgodnie z założeniem pokrywają 100% przypadków oraz w kodzie nie znajdują się żadne duplikaty. DOdatkowo udało się implementację znacznie uprościć i skrócić, dzięki zastosowaniu podeścia modularnego w postaci obiektów reprezentujacych zachowanie jak i uzyciu paradygmantów funkcyjnych. Przystąpmy teraz do analizy rozwoju projektu na

przestrzeni poszczególnych wydań.



Na załączonych wykresach widać spadek problemów z kodem źródłowym projektem na przestrzeni posczególnych wersji. Duża część z nich była generowana w trakcie procesu refactoryzacji, lecz równie sprawne były one eliminowane.
Zmniejszyła się również ilość linii co udowadnia dużą złożoność kodu wyjściowego oraz poprawność zastosowanego rozwiązania.


Reset dates
v3.0
November 27, 2019
11:38 PM
Version: v3.0
11:36 PM
11:35 PM
11:30 PM
11:29 PM
11:27 PM
11:24 PM
11:21 PM
11:16 PM
11:14 PM
10:58 PM
v2.0
November 27, 2019
New Code Period starts here
10:36 PM
Version: v2.0
10:21 PM
v2.0-alpha
November 27, 2019
10:19 PM
Version: v2.0-alpha
Duplications
Lines of Code
Duplicated Lines
100
80
60
40
20
0
09 PM
Wed 27
03 AM
06 AM
09 AM
12 PM
03 PM
06 PM
Mon 25
12 PM
Tue 26
12 PM
Wed 27