Damian Gryciuk
indeks: 269116
repozytorium:https://github.com/dragranis/zjp

Gilded Rose kata - refaktoryzacja

Omawiany przykład na wykładach i laboratoriach, czyli Gilded Rose, jest dobrym materiałem do analizy jakości kodu oraz jego czytelności.

Omawiane kata to prosty sklep z najróżniejszymi produktami. Jego podstawowa specyfikacja zakłada, że każdy przedmiot ma nazwę, jakość oraz ilość dni na jego sprzedanie. Z każdym dniem ma spadać zarówno ilość dni jak i jakość produktów. Jeśli ilość dni jest poniżej 0, jakość spada z podwójną prędkością.

Wydawać by się mogło, że sprawa jest prosta. Specyfikacja programu wymaga jednak obsługi specjalnych produktów. Pojawiają się takie elementy jak przedmioty legendarne, którym żadna wartość nie spada. Przedmioty zaklęte, które tracą jakość dwa razy szybciej. Przedmioty, których jakość rośnie, bądź spada do zera w przypadku przekroczenia terminu sprzedaży. Dodatkowo są ograniczenia, które mówią że jakość ma się mieścić w przedziale od 0 do 50.

Tworzy to nowe ścieżki działania programu i zależności. Jeżeli kod był tworzony na szybko, bez wizji tego że może być

potrzebna jego rozbudowa, powoduje to spore problemy i kod staje się zagmatwany. Tak też jest w przypadku przykładowego kodu dla omawianego kata. Ja opieram się w tym przypadku na kodzie js-jasmine, napisanym w języku javascript.

Wszystko jest w zasadzie napisane w jednej metodzie. Jest klasa na przedmioty, oraz jest klasa sklepu, która poprzez konstruktor przyjmuje tablicę produktów i jej jedyną metodą jest updateQuality, które ma przeprowadzić wszystkie operacje zapisane w specyfikacji programu. Kod został jedynie nieznacznie zmodyfikowany, aby dodać obsługę przedmiotów conjured oraz metodę, która wypisze produkty i zmiany na ekranie.

Skoro wszystko jest jedną metodą, nietrudno zauważyć co tu się stało. Przy takiej ilości różnych opcji i zależności powstało tu całe drzewo instrukcji warunkowych. Ich ilość i stopień zagnieżdżenia utrudnia zrozumienie kodu. Przykładowo, w niektórych miejscach jest to nawet 5 zagnieżdżonych instrukcji warunkowych. Dodatkowo przy analizie tego kodu można zauważyć wielokrotne powtórzenia, co dodatkowo utrudnia pracę z nim. Z tych powodów należy wykonać refaktoryzację, czyli przerobienie kodu na bardziej optymalną i przejrzystą formę przy zachowaniu tej samej funkcjonalności.

Niestety nie dałem rady wymyślić sposobu na proste i przystępne modyfikowanie kodu krok po kroku. Jest on bardzo nieczytelny. Postanowiłem dokonać większych zmian i na końcu porównać oba programy.

Podstawowe zmiany to:

1. Rozbudowa klasy item o metody decreaseDays, correctQuality, passedSellIn, write

2. Napisanie klas dziedziczących po Item: Backstage, Cheese, Legendary, Conjured
3. Każda z tych klas ma własny odpowiednik metody changeQuality
4. Napisanie testów

Należałoby jednak kody te porównać, aby móc wykazać który okazał się lepszy. Poniższa tabela powstała w oparciu o dane z analizy kodu poprzez jshint.com

Parametr	Oryginaly kod	Refaktoryzacja
Ilość funkcji	4	18
Najwięcej argumentów w funkcji	3	3
Mediana argumentów w funkcji:	1	0
Najwięcej instrukcji warunkowych w jednej funkcji	30	7
Mediana instrukcji warunkowych w funkcji	7.5	1
Cyclomatic complexity najbardziej złożonej funkcji	20	4
mediana	2	1

cyclomatic complexity		
Linie kodu	94	98

Powyższa tabela jasno wskazuje, że po refaktoryzacji jest lepiej, mimo tego że funkcji jest czterokrotnie więcej. Kod ma mniejszą złożoność, mniej instrukcji warunkowych, mniej linii kodu i praktycznie wszystkie funkcje są bezargumentowe.

Kod po refaktoryzacji przechodzi 25 przygotowanych testów, które mają na celu zweryfikować działanie kodu.

Podsumowując, uważam że powyżej opisany kod po refaktoryzacji działa lepiej, jest łatwiejszy w obsłudze i modyfikacji. Zapewne da się to napisać lepiej, ale teraz można bez problemu wprowadzać obsługę nowych rodzajów produktów.