Patryk Pobłocki
https://github.com/ppoblocki/adv-prog-2019

The Gilded Rose Refactoring Kata

W tym dokumencie opiszę moją próbę zrefaktorowania kodu aplikacji Glided Rose.
The Glided Rose to refactoring Kata, który ma na celu zwiększenie umiejętności refaktoryzacji. Kata dostępna jest w wielu językach programowania. Zdecydowałem się dokonać refaktoryzacji używając języka Python 3.8.

Pierwsze spojrzenie na aplikacje

The Glided Rose polega na tym, że mamy listę rzeczy, które mają swoją ważność oraz jakość. Każdego dnia odejmuje się 1 punkt od ważności oraz jakości. Jeżeli ważność przedmiotu będzie liczbą ujemną, to jakość maleje podwójnie. W programie występuje kilka rodzajów przedmiotów wyjątkowych, które muszą zostać obsłużone inaczej niż przedmioty standardowe.

Aplikacja składa się z dwóch klas: `GildedRose` oraz `Item`. Klasa `GildedRose` posiada metodę `updateQuality()`. Jest to najdłuższa funkcja, występująca w programie. Widzimy, że kod tej funkcji nie jest przyjazny dla programisty, posiada liczne powtórzenia kodu oraz wielokrotnie zagnieżdżone funkcje `if`-statement.

Klasa `Item` posiada tylko 3 pola: `name`, `sellIn` oraz `quality`. Posiada także metodę `__repr__()`, którą możemy rozumieć jako metodę `toString()` w innych językach. Jej zadaniem jest reprezentacja tych 3 pól na ekranie.

Wymagania / ograniczenia

Nawiązując do Kata możemy zmienić metodę `updateQuality()` w klasie `GildedRose` albo dodać własne funkcje. Wyróżnić możemy 2 główne ograniczenia:

1. nie należy zmieniać klasy `Item`
2. nie należy zmieniać właściwości klasy `Item`

Code smell programu startowego

Do analizy code smell użyłem platformy Code Climate. Jest to analizator kodu źródłowego. Wykrywa on duplikaty, zbyt długie funkcje, a także ocenia Code Smell.

Analiza Code Climate w startowym programie wykazała szereg duplikacji w kodzie, a także zbyt dużą złożoność funkcji `updateQuality()`.

Krok pierwszy: zrozumienie jak działa aplikacja

Na początku należy zrozumieć, jak działa funkcja `updateQuality()`, która jest najdłuższą i najistotniejszą funkcją w całej aplikacji. Zasada działania jest taka, że najpierw obniżamy jakość przedmiotu, a następnie jego ważność.

Krok drugi: naprawa kodu

Okazało się, że kod nie działał zgodnie z wcześniej opisanymi wymaganiami. Naprawa programu polegała na drobnej zmianie kodu w funkcji `updateQuality()`. Przedmiot o nazwie „Conjured Mana Cake” powinno obniżać wartość 2 razy szybciej, niż zwykłe przedmioty, a tego nie robiło. Należało wprowadzić obsługę tego przedmiotu w programie.

Krok trzeci: refaktoryzacja

Postanowiłem poszczególne fragmenty kodu „popakować” w krótkie, atomowe funkcje. Dzięki temu duża funkcja `update_quality()` stała się funkcją składającą się z kilkudziesięciu mniejszych funkcji, prostych do zrozumienia. W programie nie występują zagnieżdżone funkcje `if`-statment, a atomowe funkcje zostały sparametryzowane w celu uniknięcia duplikacji kodu. Uważam, że takie rozwiązanie bardzo poprawiło czytelność kodu. Zgodnie z zasadą – nie zmieniałem działania już istniejący funkcji tylko napisałem własne.

Wynik końcowy

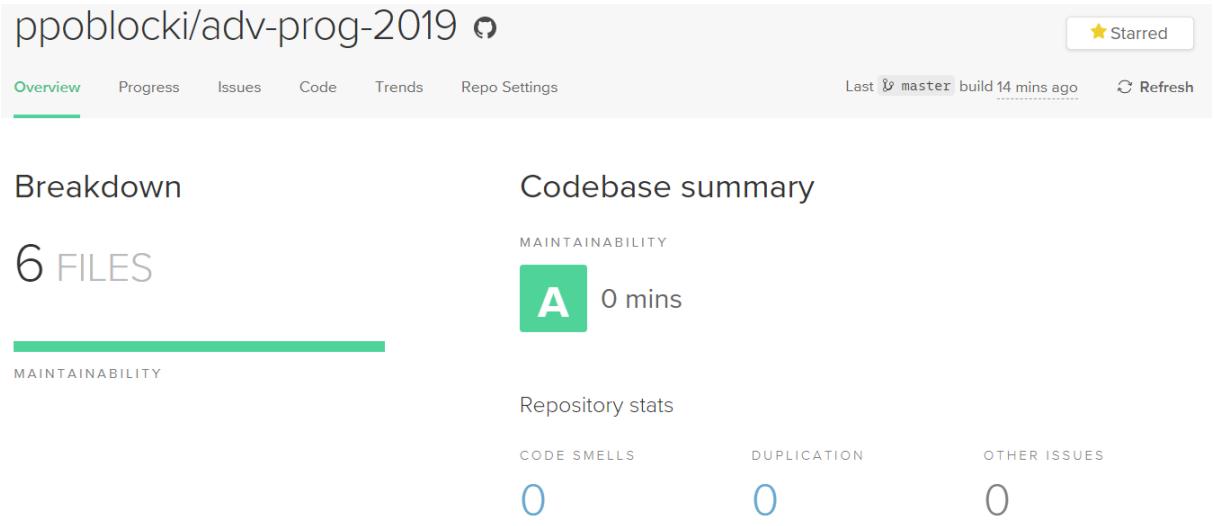
Po dokonaniu refaktoryzacji, tak wygląda moja funkcja `updateQuality()`:

```
11      def update_quality(self):
12          for item in self.items:
13              # Quality update
14              if item.isStandardItem():
15                  item.updateQualityForStandardItem()
16              else:
17                  item.updateQualityForNonStandardItem()
18              # Sell in update
19              item.updateSellIn()
```

Widzimy, że funkcja jest znacznie krótsza. Funkcja `updateQualityForStandardItem()` obniża wartość przedmiotu o 1, a funkcja `updateQualityForNonStandardItem()` ma w sobie szereg warunków, aby określić z jakim wyjątkowym produktem ma do czynienia.

Code Smell programu końcowego

Program końcowy zyskał na czytelności. Zlikwidowane zostały duplikacje, a złożoność wszystkich funkcji nie przekracza dozwolonej wartości. Tak wygląda ocena mojej refaktoryzacji według *Code Climate:*



Przeprowadziłem dodatkową weryfikację kodu programem *Pylint*. Ocenia on kod w skali 0-10 (tragicznie napisany kod może uzyskać ujemną ocenę!). Tak wyglądają wyniki porównując je do startowej wersji programu (maksymalny wynik to 10):

	Przed	Po
gilded_rose.py	6.67	9.81
test_gilded_rose.py	6.00	10
texttest_fixture.py	0	7.5

Podsumowanie

Uważam, że refaktoryzacja przeprowadzona przeze mnie poprawiła ogólny wygląd kodu. Pozbyłem się powtórzeń w kodzie, a także rozbiłem poszczególne fragmenty kodu na bardziej atomowe. Programowanie, refaktoryzacja to sztuka. Nie ma algorytmu, który jednoznacznie powiedziałby, że to zadanie należy napisać tak, a nie inaczej. Czasami po refaktoryzacji kod jest bardziej czytelny, a czasami - przeciwnie.