

Gilded Rose

RO 2019

*Opis przebiegu procesu refaktoryzacji programu Gilded Rose
napisanego w języku Python.*

Mikołaj Błaszczyk

github.com/mblaszczyk97

Spis treści

Porównanie mojej wersji z pierwotnym programem	1
Porównanie poprawności kodu według obowiązujących zasad w języku python	3
Wygląd kodu przed i po zmianach	4
Proces refaktoryzacji	6
Testy i przechodzenie	14
Wnioski	15

Porównanie mojej wersji z pierwotnym programem

Porównanie cykliczności obu wersji

```
gilded_roseold.py
M 8:4 GildedRose.update_quality - D
C 3:0 GildedRose - C
C 43:0 Item - A
M 5:4 GildedRose.__init__ - A
M 44:4 Item.__init__ - A
M 49:4 Item.__repr__ - A

6 blocks (classes, functions, methods) analyzed.
Average complexity: B (6.333333333333333)
```

Złożoność cykliczności

metryka
oprogramowania
opracowana przez
Thomasa J.
McCabe'a w 1976,
używana do pomiaru
stopnia
skomplikowania
programu. Podstawą
do wyliczeń jest
liczba dróg w
schemacie
blokowym danego
programu, co
oznacza wprost
liczbę punktów
decyzyjnych w tym
programie.

Rysunek 1 - złożoność cykliczności wersji pierwotnej obliczona w programie Radon

```
gilded_rose.py
M 82:4 GildedRose.update_quality - B
M 34:4 Check.is_normal - B
C 77:0 GildedRose - B
M 62:4 Check.add_backstage - A
C 1:0 Check - A
M 6:4 Check.updateQuality - A
M 48:4 Check.is_backstage_sellin_less_11 - A
M 55:4 Check.is_backstage_sellin_less_6 - A
M 72:4 Check.sub_conjured - A
M 16:4 Check.updateSellin - A
M 22:4 Check.is_sulfuras - A
M 28:4 Check.is_backstage - A
M 42:4 Check.is_aged_brie - A
C 101:0 Item - A
M 3:4 Check.__init__ - A
M 79:4 GildedRose.__init__ - A
M 103:4 Item.__init__ - A
M 108:4 Item.__repr__ - A

18 blocks (classes, functions, methods) analyzed.
Average complexity: A (3.0555555555555554)
```

Rysunek 2 - złożoność cykliczności mojej wersji programu także obliczona w Radonie

Wnioski

Jak widać pierwsza wersja programu otrzymała 6.3 punkty wartości cyklomatycznej natomiast moja wersja ponad 2 razy mniej czyli tylko 3 punkty. Obie te wartości są mniejsze od 10, gdzie pierwotny program otrzymał ocenę **B**, a moja implementacja ocenę **A**, obie oznaczają, że kod jest prosty i stanowi nieznaczne ryzyko. Nie ma w tym nic dziwnego, ponieważ program jest mało rozbudowany. Prawdopodobnie gdybyśmy zaczęli dodawać coraz to nowe funkcjonalności, punktacja pierwszej wersji mogłaby znacznie wzrosnąć, a kod stać się coraz to bardziej zawiły. Widać to przede wszystkim na wprowadzonej przeze mnie dodatkowo klasie i metodach w niej zawartych. Według *Radona* wszystkie otrzymały ocenę **A** – najlepszą możliwą. W ten sposób każda nowa metoda może być prosta do zrozumienia, ale co najważniejsze może być w łatwy sposób dodana do głównej klasy, czego nie można powiedzieć o starej wersji programu. Udowadniają to dwie oceny wystawione dla klasy **GildedRose** oraz metody **update_quality()** czyli odpowiednio **C** i **D** gdzie przy mojej implementacji klasa **GildedRose** ma ocenę **B**, tak samo jak i **update_quality()**.

Dowodzi to jednoznacznie, że mój program jest lepiej przystosowany na rozwój, a stworzone w nim metody są atomowe, więc nie są podatne na częste zmiany.

Jak liczona jest złożoność cyklomatyczna?

Złożoność cyklomatyczna w najprostszym tłumaczeniu jest liczona poprzez niniejszy wzór:

Liczba bloków – Liczba dróg (strzałek) + 2*ilość dróg posiadających wyjście.

Metrykę taką wylicza się ze stworzonego wcześniej diagramu blokowego naszego programu. W ten sposób jesteśmy w stanie sprawdzić złożoność każdego fragmentu kodu i oznacza to, że im więcej pętli i twierdzeń warunkowych w programie tym bardziej jest on skomplikowany i niejasny do zrozumienia.

Większość powszechnych języków programowania zawiera specjalnie opracowane technologie, które służą temu by taką złożoność policzyć na podstawie napisanego kodu. Dla **pythona** jest to np. **Radon**, a dla **Javy** np. **SonarQube**,

Porównanie poprawności kodu według obowiązujących zasad w języku python

```
***** Module gilded_roseold
gilded_roseold.py:10:0: C0301: Line too long (103/100) (line-too-long)
gilded_roseold.py:1:0: C0114: Missing module docstring (missing-module-docstring)
gilded_roseold.py:3:0: C0115: Missing class docstring (missing-class-docstring)
gilded_roseold.py:3:0: R0205: Class 'GildedRose' inherits from object, can be safely removed
gilded_roseold.py:8:4: C0116: Missing function or method docstring (missing-function-docstring)
gilded_roseold.py:9:8: R1702: Too many nested blocks (6/5) (too-many-nested-blocks)
gilded_roseold.py:9:8: R1702: Too many nested blocks (6/5) (too-many-nested-blocks)
gilded_roseold.py:9:8: R1702: Too many nested blocks (6/5) (too-many-nested-blocks)
gilded_roseold.py:8:4: R0912: Too many branches (20/12) (too-many-branches)
gilded_roseold.py:3:0: R0903: Too few public methods (1/2) (too-few-public-methods)
gilded_roseold.py:43:0: C0115: Missing class docstring (missing-class-docstring)
gilded_roseold.py:43:0: R0903: Too few public methods (1/2) (too-few-public-methods)

-----
Your code has been rated at 7.00/10 (previous run: 7.00/10, +0.00)
```

Rysunek 3 – rating starego programu Gilded Rose według pylint

```
-----
Your code has been rated at 10.00/10
```

Rysunek 4 - rating mojego rozwiązania według pylint

Wnioski

Jeżeli chodzi o zasady poprawnej pisowni oba programy utrzymują wysoki poziom. Warto jednak zauważyć, że pierwotna wersja Gilded Rose zawiera kilka mniejszych problemów np.:

- Braku dokumentacji stworzonych metod
- Zbyt długich linii kodu w niektórych miejscach
- Zbędnego dziedziczenia w jednym miejscu

Nie są to krytyczne błędy, natomiast wpływają one na jakość prezentowanego kodu.

Moje rozwiązanie nie posiada żadnych tego typu błędów i zostało ocenione najwyższą notą czyli **10/10** w porównaniu do starego rozwiązania, które ma ocenę **7/10**.

Wygląd kodu przed i po zmianach

Wersja Pierwotna:

[illegible]

Moje rozwiązanie:

```
1. def update_item(self):
2.     for item in self.items:
3.         Check(item).updateSellin(-1)
4.         if Check(item).is_aged_brie():
5.             Check(item).updateQuality(1)
6.         elif Check(item).is_backstage():
7.             Check(item).add_backstage()
8.         elif Check(item).is_sulfuras():
9.             item.quality = 80
10.        else:
11.            if Check(item).is_normal():
12.                Check(item).updateQuality(-1)
13.            else:
14.                Check(item).updateQuality(-2)
15.        if item.quality >= 50 and not Check(item).is_sulfuras():
16.            item.quality = 50
```

Jak widać zmieniona przeze mnie metoda `update_quality()` jest przede wszystkim łatwa do zrozumienia. Po nazwie wywoływanych metod wiemy, w którym fragmencie kodu co jest sprawdzane oraz jaka akcja jest wykonywana czego nie można powiedzieć o pierwszej wersji programu, w której mamy ogromną liczbę zagnieżdżonych **if-ów** i tak naprawdę ciężko jest zrozumieć gdzie sprawdzamy czy do jakiej kategorii należy przedmiot – **item**.

Proces refaktoryzacji

Krok 1

Wyciągnięcie linii zmieniającej wartość quality oraz sell_in przedmiotu w sklepie – zamiana:

```
1. item.quality = item.quality - wartość
2. item.sell_in = item.sell_in - wartość
```

Na:

```
1. Check(item).updateQuality(wartość)
2. Check(item).updateSellin(wartość)
```

Za pomocą dodanych metod:

```
1. def updateQuality(self, by):
2.     """Updating Quality of one item"""
3.     if self.item.sell_in >= 0:
4.         self.item.quality = self.item.quality + by
5.     else:
6.         self.item.quality = self.item.quality + 2*by
7.     if self.item.quality < 0:
8.         self.item.quality = 0
9.     return self.item
10.
11. def updateSellin(self, by):
12.     """Updating Sellin of one item"""
13.     if not Check(self.item).is_sulfuras():
14.         self.item.sell_in = self.item.sell_in + by
15.     return self.item
```

Oraz dodanie testu - **test_items_quality_decreases_as_name_suggest** sprawdzającego czy wartość **quality** jest w odpowiedni sposób zmieniana w czasie.

Krok 2

Stworzenie metod boolowskich sprawdzających warunki w **if-ach** – zmiana:

```
1. if item.name == "Sulfuras, Hand of Ragnaros":
2. if item.name == "Aged Brie":
3. if item.name == "Backstage passes to a TAFKAL80ETC concert":
```

Na:

```
1. Check(item).is_aged_brie():
2. Check(item).is_backstage():
3. Check(item).is_sulfuras():
```

Za pomocą metod:

```
1. def is_sulfuras(self):
2.     """Bool check for Sulfuras"""
3.     if self.item.name == "Sulfuras, Hand of Ragnaros":
4.         return True
5.     return False
6.
7. def is_backstage(self):
8.     """Bool check for Backstage"""
9.     if self.item.name == "Backstage passes to a TAFKAL80ETC concert":
10.        return True
11.    return False
12.
13. def is_aged_brie(self):
14.     """Bool check for Aged Brie"""
15.     if self.item.name == "Aged Brie":
16.         return True
17.    return False
```

oraz dodanie testu - **test_items_sellin_decreases_as_name_suggest** sprawdzającego czy wartość **sell_in** jest w odpowiedni sposób zmieniana w czasie.

Krok 3

Stworzenie metod które będą sprawdzać o ile zwiększać **quality** przedmiotu o nazwie **concert** zamiana:

```
1. if item.name == "Backstage passes to a TAFKAL80ETC concert":
2.     if item.sell_in < 11:
3.         if item.quality < 50:
4.             item.quality = item.quality + 1
5.     if item.sell_in < 6:
6.         if item.quality < 50:
7.             item.quality = item.quality + 1
```

Na:

```
1. check.add_backstage(item)
```

Za pomocą metod:

```
1. Za pomocą nowych metod: def is_backstage_sellin_less_11(self):
2.     """Bool check for Concert less than 11 sellin"""
3.     if self.item.sell_in < 11:
4.         if self.item.quality < 50:
5.             return True
6.     return False
7.
8. def is_backstage_sellin_less_6(self):
9.     """Bool check for Concert less than 6 sellin"""
10.    if self.item.sell_in < 6:
11.        if self.item.quality < 50:
12.            return True
13.    return False
14.
15. def add_backstage(self):
16.     """Changing backstage according to instructions"""
17.     if Check(self.item).is_backstage():
18.         if Check(self.item).is_backstage_sellin_less_6():
19.             Check(self.item).updateQuality(3)
20.         elif Check(self.item).is_backstage_sellin_less_11():
21.             Check(self.item).updateQuality(2)
22.         if self.item.sell_in < 0:
23.             self.item.quality = 0
```

Oraz dodanie testu - **test_add_backstage** sprawdzającego wartość zmiany w **concercie**.

Krok 4

Sprawdzanie czy przedmiot jest *normalny* - zmiana:

```
1. if item.name != "Aged Brie" and item.name != "Backstage passes to a TAFKAL80ETC concert" and item.name.lower().find("conjured") == -1:
2.     if item.quality > 0:
3.         if item.name != "Sulfuras, Hand of Ragnaros":
```

Na:

```
1. if Check(item).is_normal()
```

Za pomocą nowych metod:

```
1. def is_normal(self):
2.     """Bool check for Normal"""
3.     if self.item.name != "Aged Brie" and self.item.name != "Backstage passes to a TAFKAL80ETC concert" and self.item.name.lower().find("conjured") == -1:
4.         if self.item.quality > 0:
5.             if not Check(self.item).is_sulfuras():
6.                 return True
7.     return False
```

Oraz dodanie testu - **test_add_normal** sprawdzającego czy wartość zmiany w *normalnym* przedmiocie uległa poprawnie zmianie.

Krok 5

Zmiana warunku, który obsługuje przedmioty, które nie są *normalne* - zmiana:

```
1. else:
2.     if item.quality < 50:
3.         check.updateQuality(item, 1)
4.         if item.quality > 2 and item.name.lower().find("conjured") != -1:
5.             check.updateQuality(item, -3)
6.         check.add_backstage(item)
```

Na:

```
1. if not check.is_normal(item) and item.quality < 50:
2.     Check(item).updateQuality(1)
3.     Check(item).sub_conjured(-3, 2)
4.     Check(item).add_backstage()
```

Za pomocą metod:

```
1. def sub_conjured(self, subber, quality):
2.     """Method for subbing conjured items"""
3.     if self.item.quality > quality and self.item.name.lower().find("conjure
d") != -1:
4.         Check(self.item).updateQuality(subber)
```

oraz dodanie testu - **test_add_aged_brie** sprawdzającego wartość zmiany w przedmiocie, który nie jest zaliczany do przedmiotów *normalnych*.

Krok 6

Zmiana sposobu odejmowania wartości **sell_in** tak aby była ona odejmowana dla wszystkich po obrocie pętli. W tym celu usuwam wszystkie warunki po **if item.sell_in < 0:** oraz dla przedmiotu, który jest *conjured* muszę dodać warunek sprawdzający jak zmniejszać jego **quality**. Zmiana w liniach:

```
1. def update_quality(self):
2.     for item in self.items:
3.         if Check(item).is_normal():
4.             Check(item).updateQuality(-1)
5.         if not check.is_normal(item) and item.quality < 50:
6.             Check(item).updateQuality(1)
7.             Check(item).sub_conjured(-3, 2)
8.             Check(item).add_backstage()
9.         if not check.is_sulfuras(item):
10.            Check(item).updateSellin(-1)
11.        if item.sell_in < 0:
12.            if not Check(item).is_aged_brie():
13.                if not Check(item).is_backstage():
14.                    if item.name.lower().find("conjured") != -
15.                        1 and item.quality > 1:
16.                            Check(item).updateQuality(-1)
17.                            if item.quality > 0:
18.                                if not Check(item).is_sulfuras():
19.                                    Check(item).updateQuality(-1)
20.                                else:
21.                                    Check(item).updateQuality(-item.quality)
22.                            else:
23.                                if item.quality < 50:
24.                                    Check(item).updateQuality(1)
```

Na:

```
1. def update_quality(self):
2.     for item in self.items:
3.         if Check(item).is_normal():
4.             Check(item).updateQuality(-1)
5.         if not Check(item).is_normal() and item.quality < 50:
6.             if item.name.lower().find("conjured") == -
7.                 1 or item.name.lower().find("conjured") != -1 and item.quality > 0:
8.                 Check(item).updateQuality(1)
9.                 Check(item).sub_conjured(-3, 2)
10.                Check(item).add_backstage()
11.            if not Check(item).is_sulfuras():
12.                Check(item).updateSellin(-1)
```

Oraz dodanie testu - **test_add_conjured** sprawdzającego czy wartość przedmiotu *conjured* jest odpowiednio zmniejszana po osiągnięciu stanu **sell_in** mniejszego od 0.

Krok 6

Rozłożenie warunków sprawdzających z jakim przedmiotem mamy do czynienia tak aby były one bardziej czytelne. W metodzie **updateSellin()** dodałem sprawdzenie czy przedmiot nie jest **Sulfurasem**, ponieważ zostało ono usunięte z metody głównej **update_quality()**.

Jednocześnie w samej metodzie rozłożyłem warunki w ten sposób, że najpierw program sprawdza przypadki wyjątkowe, a dopiero potem zwykłe przedmioty. Dodatkowo przenieśliśmy zmniejszanie wartości **sell_in** na początek pętli dla wygody. Metoda **sub_conjured()** stała się teraz niepotrzebna ponieważ za pomocą sprawdzenia wiem czy przedmiot jest **conjured** i możemy odjąć od niego wartość **quality** stosując już metodę **updateQuality()**. Jeszcze jedna metoda została zmieniona, a jest to **add_backstage(item)** gdzie zmieniłem logikę warunków, aby wszystko było bardziej czytelne. Zmiana w liniach:

```
1. def update_quality(self):
2.     for item in self.items:
3.         if Check(item).is_normal():
4.             Check(item).updateQuality(-1)
5.         if not Check(item).is_normal() and item.quality < 50:
6.             if item.name.lower().find("conjured") == -
7. 1 or item.name.lower().find("conjured") != -1 and item.quality > 0:
8.                 Check(item).updateQuality(1)
9.                 Check(item).sub_conjured(-3, 2)
10.                Check(item).add_backstage()
11.            if not Check(item).is_sulfuras():
12.                Check(item).updateSellin(-1)
```

Na:

```
1. def update_quality(self):
2.     """Method to update every item"""
3.     for item in self.items:
4.         Check(item).updateSellin(-1)
5.         if Check(item).is_aged_brie():
6.             Check(item).updateQuality(1)
7.         elif Check(item).is_backstage():
8.             Check(item).add_backstage()
9.         elif Check(item).is_sulfuras():
10.            item.quality = 80
11.        else:
12.            if Check(item).is_normal():
13.                Check(item).updateQuality(-1)
14.            else:
15.                Check(item).updateQuality(-2)
16.        if item.quality >= 50 and not Check(item).is_sulfuras():
17.            item.quality = 50
```

Oraz usunięcie niepotrzebnego już warunku w metodzie `add_backstage()`:

Usuujemy linie

```
1. if Check(item).is_backstage():
```

Ponieważ już sprawdzamy ten warunek w metodzie `update_quality()`

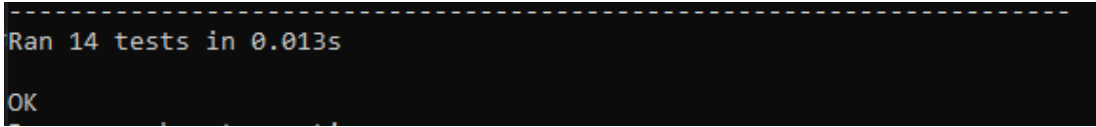
Oraz dodanie paru testów sprawdzających pozostałe przypadki i sprawność metod.

Krok 7

Zmiana nazewnictwa metody `update_quality()` aby nie myliła się z metodą zmieniającą tylko `quality` przedmiotu w klasie `Check()`.

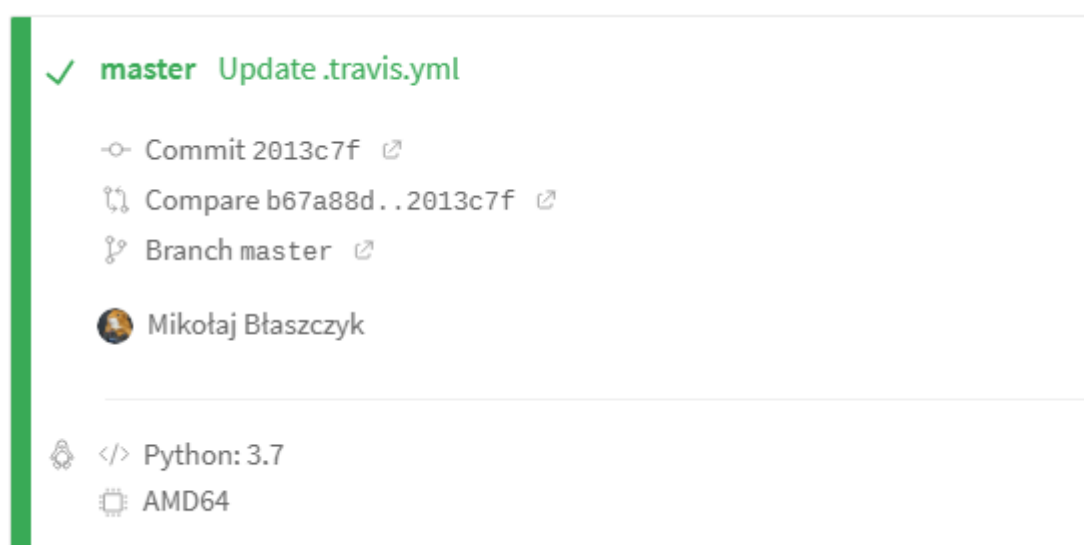
W ten sposób zmieniłem nazwę na `update_item()`.

Testy i przechodzenie



```
-----  
Ran 14 tests in 0.013s  
OK
```

Rysunek 5 - testy jednostkowe przeprowadzone na moim rozwiązaniu



Rysunek 6 - raport wygenerowany przez zewnętrzną maszynę przez TravisCI

Wszystkie 14 testy przeszły i potwierdziły zgodność obu wersji oprogramowania zarówno pierwotnego Gilded Rose jak i mojej implementacji.

Wnioski

Sądzę, że powyższe rozdziały rozwiały wszelkie wątpliwości do tego, która wersja programu jest lepsza. Zarówno wartość **cyklomatyczna** jak i **ocena poprawności stylistycznej kodu** przechylają wagę na korzyść mojej autorskiej implementacji. Oczywistym jest fakt, że nie jest ona idealna i można by jeszcze wiele w niej zmienić by była:

- lepiej zoptymalizowana
- jeszcze lepsza pod względem **cyklomatycznym**

Jednak to nie było celem tego zadania.

W tej pracy zadaniem było zamienienie klasy **GildedRose** tak aby była ona przede wszystkim:

- zrozumiała
- otwarta na rozszerzenia
- zamknięta na modyfikacje kodu

Jednocześnie nie modyfikując klasy **Item** i nie zmieniając struktury projektu. Przy czym pierwsze dwa punkty udało się całkowicie natomiast ostatni podpunkt mógłby zostać moim zdaniem lepiej zaimplementowany w napisanym przeze mnie kodzie.

Natomiast istotne jest to, że napisany przeze mnie program pod względem cyklomatycznym jest ponad 2 razy lepszy, a jeszcze ważniejszym jest fakt, że jest on po prostu łatwy do zrozumienia dla każdego kto go przeczyta, czego nie mogę powiedzieć o pierwotnej wersji *Gilded Rose*.