Projekt: Gilded Rose Małgorzata Szałajda

1) *Streszczenie*

Celem przedstawianego projektu była refaktoryzacja kodu obsługującego sklep Gilded Rose, jak również dodanie nowej funkcjonalności - obsługa kolejnej kategorii produktów tzw. "Conjured". Aby móc w bezpieczny sposób refaktoryzować kod najpierw napisałam dodatkowe testy. Otrzymany plik testowy *gilded_rose_test.py* zmodyfikowałam tak, aby ilość testowanych dni wynosiła 30. Dzięki temu mogłam w napisanych testach jednostkowych dostępnych w pliku *gilded_rose_unittest.py* wykorzystać otrzymane wartości i porównywać je z wartościami ze sprawdzanego kodu. Testy obejmują wszystkie kategorie produktów: "Normal", "Aged Brie", "Sulfuras", "Backstage passes" jak również "Conjured". Starałam się w testach uwzględnić również przypadki graniczne, kiedy produkty stają się przeterminowane lub data koncertu mija. Do refaktoryzacji starałam się zastosować metodę "małych kroków" Flocking Rules oraz Extract Method. Poniżej zaprezentowałam jak doszłam z 1. podstawowej wersji do 2. wersji - mojej po refaktoryzacji. Zamieściłam również porównanie złożoności obu wersji kodu otrzymane po zastosowaniu do nich narzędzia radon. Wszystkie kody zamieszczone są w moim repozytorium: https://github.com/malgorzatasz/zjp

2) *Nowa funkcjonalność: "Conjured item"*

Aby dodać obsługę nowej kategorii starałam się zrozumieć początkowy kod, znaleźć kategorię produktów, których obsługa jest zbliżona. Dlatego po zmniejszeniu jakości "Normalnych" produktów jeszcze raz zmniejszam jakość dla produktów, które w nazwie mają "Conjured", a ich wartość jest większa od zera:

```
if "Conjured " in item.name and item.quality>0:
    item.quality=item.quality-1
```

Ten sam kod wykorzystuję przy obniżaniu wartości produktów kiedy minie data sprzedaży.

3) *Kod po refaktoryzacji*

Na następnej stronie zaprezentowałam mój kod, który otrzymałam po refaktoryzacji. Stosując Flocking Rules chciałam żeby mój kod spełniał zasadę Open/Closed. Myślę, że udało mi się uprościć kod, poprawić jego czytelność. Jeżeli chodzi o jego otwartość na rozszerzenia, a zamkniętość na modyfikacje nie do końca się udało. Wydaje mi się, że robiłam tzw. "krok do przodu" porównując do pierwotnej wersji, lecz wymaga on kolejnych zmian, żeby spełniać tę zasadę.

Kod po refaktoryzacji:

```
class GildedRose(object):

    def __init__(self, items):
        self.items = items

    def update_quality(self):
        for item in self.items:
            self.decrease_quality(item)
            self.update_sell_in(item)
            if item.sell_in < 0:
                self.decrease_quality(item)

    def decrease_quality_value(self, item):
        if item.quality > 0:
            item.quality -= 1

    def increase_quality_value(self, item):
        if item.quality < 50:
            item.quality += 1

    def item_quality_decrease_twice(self, item):
        if "Conjured " in item.name:
            self.decrease_quality_value(item)

    def item_quality_decrease(self,item):
        return item.name != "Aged Brie" and item.name != "Backstage passes to a TAFKAL80ETC concert"

    def update_backstage_passes(self, item):
        if item.name == "Backstage passes to a TAFKAL80ETC concert":
            if item.sell_in < 11:
                self.increase_quality_value(item)
            if item.sell_in < 6:
                self.increase_quality_value(item)
            if item.sell_in<0:
                item.quality = item.quality - item.quality
        return item.quality

    def decrease_quality(self,item):
        if self.item_quality_decrease(item):
            if item.name != "Sulfuras, Hand of Ragnaros":
                self.decrease_quality_value(item)
                self.item_quality_decrease_twice(item)
        else:
            self.increase_quality_value(item)
            self.update_backstage_passes(item)

    def update_sell_in(self,item):
        if item.name != "Sulfuras, Hand of Ragnaros":
            item.sell_in = item.sell_in - 1

class Item:
    def __init__(self, name, sell_in, quality):
        self.name = name
        self.sell_in = sell_in
        self.quality = quality

    def __repr__(self):
        return "%s, %s, %s" % (self.name, self.sell_in, self.quality)
```

4) *Refaktoryzacja - Małe Kroki*

1) Metoda `decrease_quality_value(self,item)`

Najpierw stworzyłam nową metodę, która obniża wartość jakości produktu, jeżeli jest ona większa od zera. Odejmowanie 1 od jakości produktu występowało kilkukrotnie w kodzie, dlatego zamieniłam działanie: `item.quality = item.quality - 1` na wywołanie następującej metody:

```
def decrease_quality_value(self,item):
        if item.quality > 0:
                item.quality -= 1
```

2) Metoda `increase_quality_value(self,item)`

Ta metoda jest bardzo podobna do powyżej, z tą różnicą, że zamiast odejmowania występuje dodawanie do jakości produktu, jeżeli jest ona zmniejsza od górnej granicy 50. Dlatego wszystkie operacje `item.quality = item.quality + 1` zostały zastąpione wywołaniem metody:

```
def increase_quality_value(self,item):
        if item.quality < 50:
                item.quality += 1
```

3) Metoda `item_quality_decrease_twice(self,item)`

Opisywana metoda została utworzona dla produktów z kategorii "Conjured", których jakość następnie obniżamy wykorzystując wcześniej stworzoną metodę `decrease_quality_value`. A zatem zamieniamy warunek:
`if "Conjured " in item.name and item.quality>0` na wywołanie metody:

```
def item_quality_decrease_twice(self, item):
        if "Conjured " in item.name:
        self.decrease_quality_value(item)
```

Powyższe kroki miały na celu poprawienie czytelności kodu, natomiast kolejne kroki skupiają się na uzyskaniu zasady Open/Closed w kodzie.

4) Metoda `item_quality_decrease(self,item)`

Najmniejsza różnica w kodzie od której zaczęłam stosowanie Flocking Rules dotyczyła dwóch pierwszych warunków if w poniższych framgentach:

```
if item.name != "Aged Brie" and item.name != "Backstage passes to a TAFKAL80ETC concert":
        if item.quality > 0:
            if item.name != "Sulfuras, Hand of Ragnaros":
                self.decrease_quality_value(item)
                self.item_quality_decrease_twice(item)

        if item.name != "Aged Brie":
                if item.name != "Backstage passes to a TAFKAL80ETC concert":
                    if item.quality > 0:
                        if item.name != "Sulfuras, Hand of Ragnaros":
                            self.decrease_quality_value(item)
                            self.item_quality_decrease_twice(item)
```

Aby usunąć tę różnicę utworzyłam następującą metodę zastępując w pierwszym fragmencie pierwszy warunek, a w drugim dwa pierwsze warunki tą metodą:

```
def item_quality_decrease(self,item):
    return item.name != "Aged Brie" and item.name != "Backstage passes to a TAFKAL80ETC concert"
```

5) Metoda `update_backstage_passes(self,item)`

Powyższa zmiana w kodzie spowodowała zepsucie testów, dlatego potrzebna jest metoda wyszczególniająca kategorię produktów "Backstage passes":

```
def update_backstage_passes(self,item):
        return item.name == "Backstage passes to a TAFKAL80ETC concert"
```

Następnie zmieniamy następujący warunek else:

```
    else:
        item.quality = item.quality - item.quality
```

na warunek elif z wykorzystaniem utworzonej metody, co spowoduje naprawę testów:

```
if item.sell_in < 0:
        if item_quality_decrease(item):
                if item.quality > 0:
                        if item.name != "Sulfuras, Hand of Ragnaros":
                                self.decrease_quality_value(item)
                                self.item_quality_decrease_twice(item)
        elif self.update_backstage_passes(item):
                item.quality = item.quality - item.quality
        else:
                self.increase_quality_value(item)
```

6) Poprawa metody `update_backstage_passes(self,item)`

Kolejną różnicą do usunięcia będzie ta, która występuje w dwóch else po ujednoliconych powyżej fragmentach kodu. W tym momencie wyglądają one tak:

```
    else:
        if item.quality < 50:
            self.increase_quality_value(item)
            if item.name == "Backstage passes to a TAFKAL80ETC concert":
                if item.sell_in < 11:
                    if item.quality < 50:
                        self.increase_quality_value(item)
                if item.sell_in < 6:
                    if item.quality < 50:
                        self.increase_quality_value(item)

    elif self.update_backstage_passes(item):
            item.quality = item.quality - item.quality
        else:
            self.increase_quality_value(item)
```

Powyższe fragmenty kodu różnią się wywołaniem przy elif dla produktów oznaczonych jako "Backstage passes" w drugim fragmencie, oraz w pierwszym zmniejszaniem jakości jeżeli produkt to "Backstage passes". Dlatego poprawiłam metodę wcześniej utworzoną metodę `update_backstage_passes(self,item)` aby obsługiwała ona również te przypadki. Poprawiona metoda wygląda następująco:

```
    def update_backstage_passes(self, item):
        if item.name == "Backstage passes to a TAFKAL80ETC concert":
            if item.sell_in < 11:
                self.increase_quality_value(item)
            if item.sell_in < 6:
                self.increase_quality_value(item)
            if item.sell_in< 0:
                item.quality = item.quality - item.quality
        return item.quality
```

Po wykonaniu tej metody oba fragmenty są następujące:

```
    else:
        if item.quality < 50:
            self.increase_quality_value(item)
            self.update_backstage_passes(item)

    else:
        self.increase_quality_value(item)
        self.update_backstage_passes(item)
```

Różnią się one sprawdzaniem czy jakość jest mniejsza od 50. Skoro w metodzie `increase_quality_value(self,item)` taki warunek również jest sprawdzany przed wykonaniem operacji dodawania, to możemy usunąć go z pierwszego else i otrzymamy takie same funkcje wywoływane po else.

7) Metoda `decrease_quality(self,item)`

Po wykonaniu powyższych kroków metoda `update_quality(self)` wygląda następująco:

```
def update_quality(self):
    for item in self.items:
        if self.item_quality_decrease(item):
            if item.quality > 0:
                if item.name != "Sulfuras, Hand of Ragnaros":
                    self.decrease_quality_value(item)
                    self.item_quality_decrease_twice(item)
        else:
            self.increase_quality_value(item)
            self.update_backstage_passes(item)
        if item.name != "Sulfuras, Hand of Ragnaros":
            item.sell_in = item.sell_in - 1
        if item.sell_in < 0:
            if self.item_quality_decrease(item):
                if item.quality > 0:
                    if item.name != "Sulfuras, Hand of Ragnaros":
                        self.decrease_quality_value(item)
                        self.item_quality_decrease_twice(item)
            else:
                self.increase_quality_value(item)
                self.update_backstage_passes(item)
```

W ramce zaznaczyłam powtarzający się fragment kodu, który udało się uzyskać stosując Flocking Rules. Zatem ten kod wykorzystujemy do stworzenia kolejnej metody:

```
def decrease_quality(self,item):
        if self.item_quality_decrease(item):
                if item.quality > 0:
                        if item.name != "Sulfuras, Hand of Ragnaros":
                                self.decrease_quality_value(item)
                                self.item_quality_decrease_twice(item)
                else:
                        self.increase_quality_value(item)
                        self.update_backstage_passes(item)
```

Metodę możemy dodatkowo uprościć, ponieważ warunek `item.quality > 0` jest sprawdzamy przed wykonaniem metod `decrease_quality_value(item)` oraz `item_quality_decrease_twice(item)` to możemy go usunąć z metody bez regresji testów.

8) Metoda `update_sell_in(self,item)`

Kolejna metoda została stworzona, żeby poprawić czytelność kodu. Obecna postać metody `update_quality(self)`to:

```
def update_quality(self):
    for item in self.items:
        self.decrease_quality(item)
        if item.name != "Sulfuras, Hand of Ragnaros":
            item.sell_in = item.sell_in - 1
        if item.sell_in < 0:
            self.decrease_quality(item)
```

Zaznaczony kod odpowiada za zaktualizowanie dnia, dlatego tworzymy nową metodę:

```
def update_sell_in(self,item):
        if item.name != "Sulfuras, Hand of Ragnaros":
            item.sell_in = item.sell_in - 1
```

Dzięki temu ostatecznie metoda `update_quality(self)` uprościła się do takiej postaci:

```
def update_quality(self):
        for item in self.items:
            self.decrease_quality(item)
            self.update_sell_in(item)
            if item.sell_in < 0:
                self.decrease_quality(item)
```

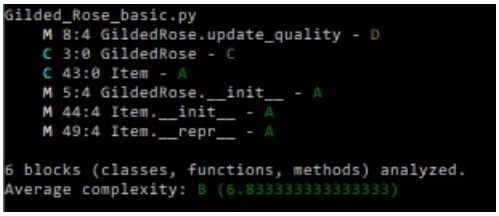
5) Wyniki złożoności

Do porównania złożoności użyłam narzędzia radon, które ocenia każdy z elementów w pliku stosując skalę od A-F gdzie poszczególne litery zgodnie z dokumentacją oznaczają:

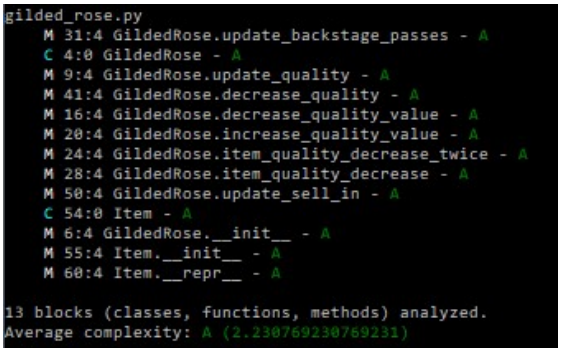
1 - 5	A (low risk - simple block)
6 - 10	B (low risk - well structured and stable block)
11 - 20	C (moderate risk - slightly complex block)
21 - 30	D (more than moderate risk - more complex block)
31 - 40	E (high risk - complex block, alarming)
41+	F (very high risk - error-prone, unstable block)

Here *block* is used in place of function, method or class.

Wyniki jakie uzyskałam dla początkowego kodu:



Wyniki dla kodu po refaktoryzacji:



Jak widać na załączonych rezultatach w wyniku refaktoryzacji uzyskałam dwa razy więcej bloków, lecz każdy z nich o najmniejszej możliwej złożoności A. W poprawionym kodzie metoda update_quality ma złożoność A, natomiast w wyjściowym kodzie było to D. Widać również jak poprawiła się średnia złożoność z klasy B na A. Zgodnie z zamieszczoną powyżej tabelą, każda z liter oznacza pewien zakres a nie dokładną wartość. Dokładniejsze wyniki uzyskałam z narzędzia lizard.ws:

Początkowy kod:

Code analyzed successfully.

File Type .py | Token Count 326 | NLOC 43

Function Name	NLOC	Complexity	Token #	Parameter #
__init__	2	1	12	
update_quality	33	23	254	
__init__	4	1	26	
__repr__	2	1	21	

W powyższym raporcie kolumna Complexity jest równoważna klasom z narzędzia radon. Dla początkowego kodu suma Complexity wynosi 26, a najbardziej złożona metoda to `update_quality` z wynikiem 23. Na kolejnej stornie zamieściłam wyniki dla mojego kodu.

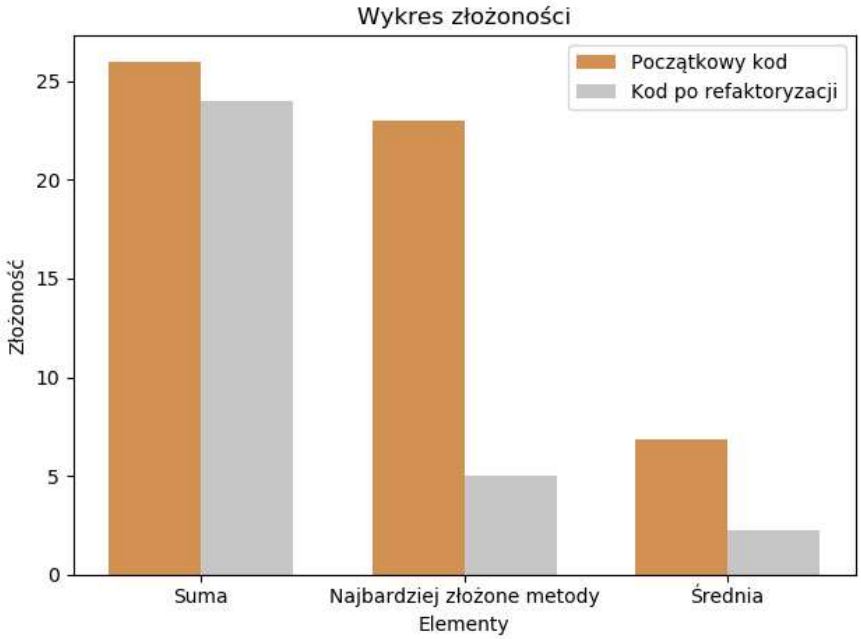
Kod po refaktoryzacji:

Code analyzed successfully.

File Type .py Token Count 326 NLOC 47

Function Name	NLOC	Complexity	Token #	Parameter #
__init__	2	1	12	
update_quality	6	3	37	
decrease_quality_value	3	2	19	
increase_quality_value	3	2	19	
item_quality_decrease_twice	3	2	20	
item_quality_decrease	2	2	19	
update_backstage_passes	9	5	62	
decrease_quality	8	3	48	
update_sell_in	3	2	23	
__init__	4	1	26	
__repr__	2	1	21	

Z raportu wynika, że sumaryczna złożoność wynosi 24, natomiast najbardziej złożoną metodą jest `update_backstage_passes` z wynikiem 5. Zobaczmy, jak uzyskane dane przedstawiają się na wykresie



Uzyskanie dużej ilości mniej złożonych metod spowodowało, że suma złożoności wyszła podobna. Najbardziej znacząca jest różnica w złożoności największych metod oraz w średniej, które wskazują, że poczynione kroki poprawiły kod. Dzięki temu najbardziej złożona metoda będzie prostsza w zrozumieniu.