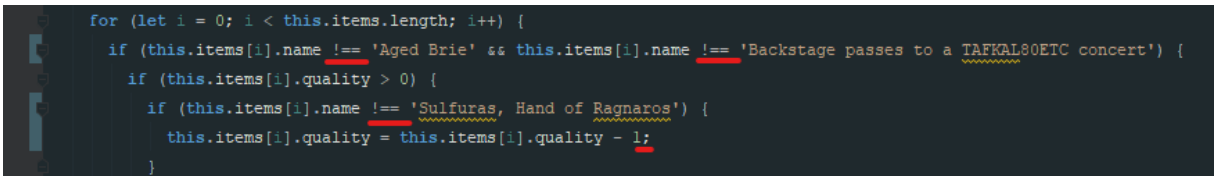
Tytuł: Refaktoryzacja kodów Gilded-Rose w języku TypeScript

Autor: Paweł Łakomiec

Kontekst: Udoskonalenie oraz naprawa błędów, jakimi są Code Smells, oraz Issues w projekcie Gilded-Rose, z użyciem frameworka Angular 8.

Link do repozytorium: https://github.com/s14094/CodeRefactor

Zdecydowałem się na refaktoryzację kodu z projektu Gilded-Rose (https://github.com/emilybache/GildedRose-Refactoring-Kata). Jako język wybrałem TypeScript, ponieważ aktualnie doszkalam się w nim, jak i wykorzystuję go w projektach komercyjnych. Do analizowania kodu użyłem narzędzia ze strony codeclimate.com. Wynik analizy wykazał sumę 9 issues gdzie 3 z nich oznaczono jako duplikaty fragmentów kodu a resztę jako code smells.



W środowisku korzystam z wtyczki SonarLint, która automatycznie pokazuje potencjalne zagrożenia w kodzie. Pierwsze na czym się skupiłem to wyeliminowanie dwóch błędów, które wskazała wtyczka.

Pierwszym z nich jest problem z Automatic Semicolon Insertion (ASI) – czyli automatycznym wstawianiu średników. TypeScript automatycznie dopisuje średniki jednak nie zawsze działa to w sposób, jaki wyobraził sobie programista. Jednym z przykładów może być ten o to fragment kodu:

```
function fn() {
    return
    {
        a: 1
    }
}
```

ASI doda średnik za return przez co funkcja zwróci undefined. Dlatego mimo możliwości języka dobrą praktyką jest dodawanie średników na końcu linii, w której normalnie by się znalazł.

Drugą rzeczą jest porównywanie zmiennych. W TypeScripcie wyróżnia się dwa typy porównań: podwójne oraz potrójne. Podwójne, standardowo porównują wartości po obu stronach, a potrójne nie tylko wartości, ale również typy zmiennych. Z uwagi na to, nie tylko dobrą praktyką, ale i lepiej zabezpieczone są porównania potrójne.

```
export namespace Globals {
  export class Dictionary {
    public static AGED = 'Aged Brie';
    public static BACKSTAGE = 'Backstage passes to a TAFKAL80ETC concert';
    public static SULFURAS = 'Sulfuras, Hand of Ragnaros';
  }
}
```

Następną rzeczą było wydzielenie wszystkich nazw do słownika oraz do osobnej klasy, dostępnej w łatwy sposób w całym projekcie. To rozwiązanie pozwala na bezproblemową zmianę wartości stringów we wszystkich miejscach. Dodatkowo dzięki temu wszystkie stałe nazwy są w jednym miejscu.

```
if (this.items[i].name !== Globals.Dictionary.AGED && this.items[i].name !== Globals.Dictionary.BACKSTAGE) {
```

a += b jest skróceniem a = a + b, zastosowanie tej reguły wyrażenia skraca kod, nie zmieniając jego funkcjonalności, co czyni go czytelniejszym:

```
this.items[i].quality += 1;
```

```
this.items[i].quality -= this.items[i].quality;
```

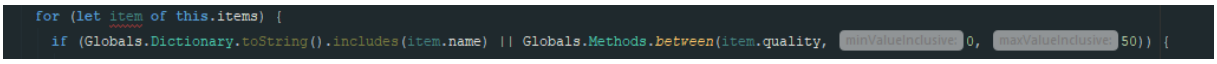
Kolejną rzeczą było zastosowanie konstrukcji pętli, jaką stosuje się w języku TypeScript:

```
for (const item of this.items) {
  if (item.name !== Globals.Dictionary.AGED && item
    item.quality -= 1;
  } else {
    if (item.quality < 50) {
```

Pętla ta pozwala w każdej iteracji działać na konkretnym obiekcie, eliminuje to potrzebę odnoszenia się do konkretnej pozycji z listy „this.items[n]”

```
item.quality -= item.quality;
```
jest równoznaczne z przypisaniem do zmiennej zera. Jest to świetny przykład, pokazujący jak można utrudnić innym programistą czytanie kodu

Pierwszą funkcję warunkową ‘if’ skróciłem do postaci:



Pierwsza część warunku działa na zasadzie sklejania wszystkich wartości w tablicy w jeden string, przez co jesteśmy w stanie użyć metody includes, która sprawdza, czy dany string zawiera się w tablicy. Jest to przydatne rozwiązanie, ponieważ niezależnie od ilości danych, jakie chcemy sprawdzać, długość tego warunku będzie niezmienna. Natomiast w drugiej części warunku napisałem metodę pomocniczą:

```
public static between(i: number, minValueInclusive: number, maxValueInclusive: number): boolean {
  return (i > minValueInclusive && i < maxValueInclusive);
}
```

Która sprawdza, czy dana wartość znajduje się w zakresie. Jest to prosta, ale niezwykle przydatna metoda, która znajdzie zastosowanie w większości projektów.

W momencie, gdy wszystkie podstawowe issues zostały naprawione, przyszedł czas na skrócenie metody poprzez rozbicie metody na pomniejsze. Rozbicie nie tylko zmniejsza ilość kodu w danej metodzie, gdzie dobrą praktyką jest maksymalnie 25 linijek na metodę, ale również pozwala w łatwy sposób pozbyć się duplikatów w kodzie.

Wyodrębniłem trzy metody:

```
changeQualityIfLowSell(sellIn: number, quality: number): number {
  if (quality === 49) {
    return 1;
  } else {
    return (sellIn < 11 ? ((sellIn < 6 ? 3 : 2)) : 1);
  }
}

calculateSellIn(item: Item): number {
  if (item.name !== Globals.Dictionary.SULFURAS) {
    return item.sellIn -= 1;
  }
  return item.sellIn;
}

changeQuality(item: Item): number {
  if (item.name === Globals.Dictionary.AGED && item.quality < 50) {
    return 1;
  } else if (item.name === Globals.Dictionary.BACKSTAGE) {
    return 0;
  } else if (item.name !== Globals.Dictionary.SULFURAS && item.quality > 0) {
    return -1;
  }
  return 0;
}
```

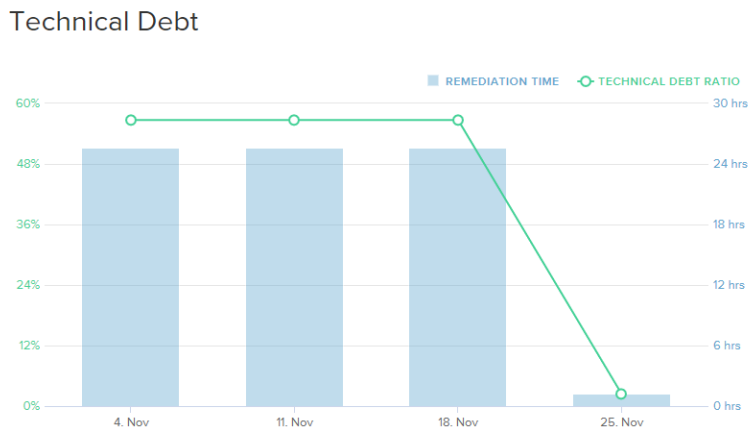
Przez co metoda główna wygląda finalnie w takiej postaci:

```
updateQuality() {
  for (let item of this.items) {
    if (Globals.Dictionary.toString().includes(item.name) || Globals.Methods.between(item.quality, minValueInclusive: 0, maxValueInclusive: 50)) {
      item.quality += this.changeQualityIfLowSell(item.sellIn, item.quality);
    } else {
      item.quality -= 1;
    }
    item.sellIn = this.calculateSellIn(item);
    if (item.sellIn < 0) {
      item.quality += this.changeQuality(item);
    }
  }
  return this.items;
}
```

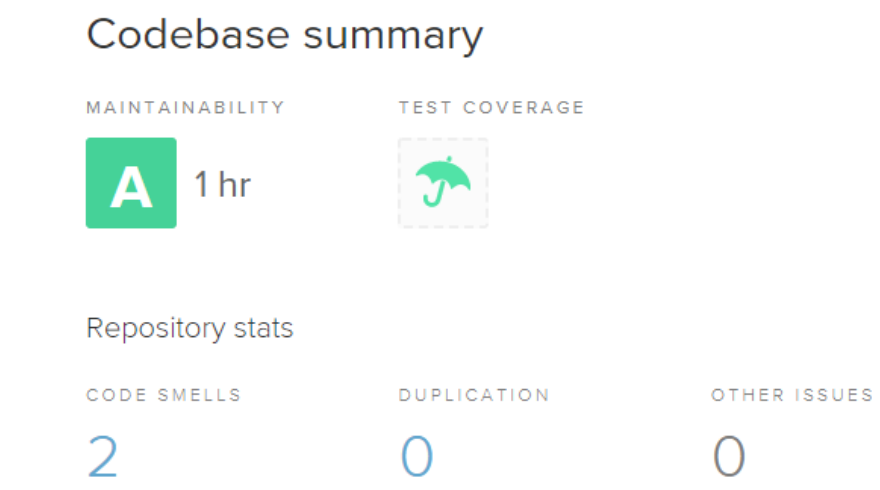
Warto dodać, że przy każdej rozbitej metodzie deklaruje na stałe jej zwracany typ. Typescript nie potrzebuje takiej informacji, sam dostosuje odpowiedni typ. Jednak jest to dobrą praktyką z uwagi na innych programistów, którzy będą również pracować na tym kodzie. Dzięki temu rozwiązaniu, w przypadku przypiania do metody do innej wartości niż zadeklarowana, dostaną błąd już w kompilacji, a nie w momencie wykonania metody.

Podsumowanie

Po wprowadzeniu zmian, Technical Debt (Dług techniczny) spadł z 56,7% do 2,4%, przez co projekt dostał najwyższa możliwą ocenę. Code Climate obliczając go, analizuje 10 kluczowych czynników powstawania różnych błędów, takich jak np. liczbę argumentów, złożoność logiki, długość pliku i metod, liczbę metod, zagnieżdżenia, duplikacje itd.



Dzięki zastosowaniu większej ilości metod, program nie ma już żadnych duplikatów, a liczba code smells zmalała z 6 do 2. Pozostałe w programie code smells to naruszona złożoność logiki, która dostała ocenę 7 z założonej maksymalnej dopuszczonej liczby 5. Podjąłem jednak decyzję, żeby zostawić to w taki sposób napisane, ponieważ aktualnie dzięki tym metodom, kod jest podatny na zmiany. Metody wpasowują się z akronim SOLID, a szczególnie w wytyczną, jaką jest otwartość na rozszerzenia ale zamknięcie na modyfikację.



Podsumowując, jakość kodu poprawiła się, jak i jego czystość oraz struktura plików, która przystosowana jest pod dalszą rozbudowę projektu.