Gilded Rose

Artur Pyśk (246832)

Refaktoryzację rozpoczniemy od napisania wartościowych testów do aktualnego (początkowego) rozwiązania Gilded Rose. Następnie zastanowimy się nad generycznym rozwiązaniem.

I Testy

Podchodząc do testów przyjrzymy się wymaganiom stawianym w opisie funkcjonowania sklepu Gilded Rose i dla każdego z nich przygotujemy osobny test, aby być pewnym, iż wszystkie istotne punkty z perspektywy sklepu zostały przez nas zauważone. Zaczynamy od stworzenia drugiego projektu w środowisku .NET - GildedRoseTest, będącym projektem do pisania testów dla Gilded Rose.

- **At the end of each day our system lowers both values for every item**

  W tym teście przygotujemy każdy przedmiot w domyślnym formacie i porównamy parametry z następnego dnia. Podczas pisania pierwszego testu okazuje się, że nie mamy przygotowanej metody Equals dla klasy Item. Chcemy być pewni, że zarówno SellIn jak i Quality są obsłużone prawidłowo przez nasz kod, więc przygotujemy metodę do porównania obiektów klasy Item.

```
public override bool Equals(object obj)
{
    if (obj == null)
        return false;

    Item comparableItem = (Item) obj;
    return Name.Equals(comparableItem.Name, System.StringComparison.CurrentCultureIgnoreCase) &&
           SellIn == comparableItem.SellIn && Quality == comparableItem.Quality;
}
```

- **The Quality of an item is never negative**

  W tym teście wybraliśmy kilka przedmiotów ustawiając Quality każdego z nich na 0 i oczekujemy, iż w następnym dniu Quality nie będzie mniejsze od zera.

- **“Aged Brie” actually increases in Quality the older it gets**

  Sprawdzimy na przełomie kilku dni czy Quality przedmiotu Aged Brie będzie się zwiększało z biegiem czasu. Ustawimy Quality na 5 oraz SellIn na 5 i wykonamy 3 razy funkcję UpdateQuality(). Później sprawdzimy czy Quality faktycznie zwiększyło swoją wartość.

- **Once the sell by date has passed, Quality degrades twice as fast.**

  W tym teście wybierzemy zwykły item “+5 Dexterity Vest” ustawiając mu SellIn na 0 i po wykonaniu kilku razy UpdateQuality() będziemy oczekiwać, że Quality będzie zmniejszać się dwa razy szybciej.

- **The Quality of an item is never more than 50.**

  Dla wszystkich przedmiotów testowych ustawiamy Quality na 50 i oczekujemy, iż przy kolejnych wykonaniach metody UpdateQuality() wartość pozostanie ta sama.

- **“Sulfuras”, being a legendary item, never has to be sold or decreases in Quality**

  Po dodaniu Sulfurasu do naszego sklepu będziemy badać czy jego Quality oraz SellIn się nigdy nie zmienia.

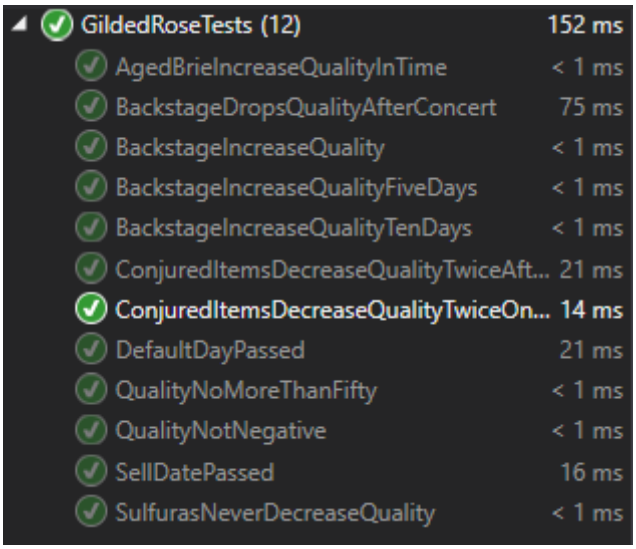
- **"Backstage passes", like aged brie, increases in Quality as it's SellIn value approaches; Quality increases by 2 when there are 10 days or less and by 3 when there are 5 days or less but Quality drops to 0 after the concert.**

  To zagadnienie rozbijemy na cztery osobne testy. Pierwszy z nich będzie dotyczył zwiększania Quality podczas upływu dni, jednakże przy SellIn > 10. Kolejnym testem będzie badanie zwiększania Quality o 2 gdy SellIn < 10 i SellIn > 5. Następny będzie dotyczył zwiększania Quality o 3 podczas gdy do koncertu zostało 5 lub mniej dni. Ostatni test dotyczy spadku Quality do 0, gdy koncert się odbędzie.

- **"Conjured" items degrade in Quality twice as fast as normal items.**

  Dla tego wymagania przygotujemy dwa testy – jeden, który sprawdza czy Quality dla Conjured zmniejsza się dwukrotnie przy SellIn > 0 oraz czy zmniejsza się czterokrotnie podczas gdy SellIn < 0.

Podsumowując, przygotowaliśmy 12 testów dotyczących wymagań postawionych dla sklepu Gilded Rose. Wszystkie testy wykonują się pomyślnie i jesteśmy gotowi do przystąpienia do refaktoryzacji kodu.



II Refaktoryzacja

Refaktoryzację rozpoczniemy od określenia koncepcji. Będziemy chcieli dążyć do stworzenia jednej klasy, która będzie podstawowym wyrażeniem Itemu w sklepie Gilded Rose. Wszystkie przedmioty, które będą posiadały jakieś inne własności niż te podstawowe, będą mogły korzystać z tego wzorca wraz z własną implementacją tych dodatkowych cech.

Zmienimy zatem klasę Item.cs na klasę abstrakcyjną opisującą Item wraz z deklaracją wszystkich niezbędnych atrybutów (Name, SellIn, Quality) oraz deklaracją metody UpdateQuality() dla każdego przedmiotu. Wszystkie typy przedmiotów, które stworzymy w obrębie sklepu będą dziedziczyły po tej klasie i nadpisywały jej UpdateQuality wg. własnych wymagań.

Zaczynamy od zmiany dostępu do atrybutów klasy na protected – aby dostęp do nich miały tylko klasy dziedziczące, czyli nasze "niezwykłe" przedmioty oprócz Quality, które będzie publiczne. Następnie dodajemy konstruktor podstawowy oraz konstruktor rozszerzony o parametry Name, SellIn oraz Quality, a także deklarujemy naszą metodę do aktualizowania Quality. Zostawiamy wcześniej zaimplementowane metody Equals() oraz ToString().

```
public abstract class Item
{
    protected string Name { get; set; }
    protected int SellIn { get; set; }
    public int Quality { get; set; }

    protected Item(string name, int sellIn, int quality)
    {
        Name = name;
        SellIn = sellIn;
        Quality = quality;
    }

    public abstract Item UpdateQuality();

    public override bool Equals(object obj)
    {
        if (obj == null)
            return false;

        Item comparableItem = (Item) obj;
        return Name.Equals(comparableItem.Name, System.StringComparison.CurrentCultureIgnoreCase) &&
               SellIn == comparableItem.SellIn && Quality == comparableItem.Quality;
    }

    public override int GetHashCode()
    {
        return Name.GetHashCode();
    }

    public override string ToString()
    {
        return Name + ", " + SellIn + ", " + Quality;
    }
}
```

Następnym krokiem jest stworzenie klas dla poszczególnych typów przedmiotów w sklepie Gilded Rose, czyli: **RegularItem** (zwykły przedmiot bez dodatkowych właściwości), **Aged Brie**, **Sulfuras**, **Conjured** oraz **BackstagePass**. Każda z tych klas dziedziczy konstruktor po klasie abstrakcyjnej oraz nadpisuje metodę UpdateQuality() realizując określone wymagania dotyczące aktualizacji Quality.

```
public class AgedBrie : Item
{
    public AgedBrie(string name, int sellIn, int quality) : base(name, sellIn, quality) { }

    public override Item UpdateQuality()
    {
        return Quality < 50 ? new AgedBrie(Name, SellIn - 1, Quality + 1)
            : new AgedBrie(Name, SellIn - 1, Quality);
    }
}
```

Dzięki takiej formie jeżeli w sklepie Gilded Rose pojawi się nowy przedmiot o niestandardowych własnościach, bez problemu możemy utworzyć klasę i zaimplementować UpdateQuality().

Sama klasa GildedRose.cs została zmodyfikowana, aby przyjmowała List<Item>, a następnie po wywołaniu metody UpdateQuality() iterowała po wszystkich przedmiotach i aktualizowała ich stan.

```
public class GildedRose
{
    public List<Item> Items { get; set; }

    public GildedRose(List<Item> items)
    {
        Items = items;
    }

    public void UpdateQuality()
    {
        for (var i = 0; i < Items.Count; i++)
        {
            Items[i] = Items[i].UpdateQuality();
        }
    }
}
```

III Poprawienie testów

Ostatni etap dotyczy wprowadzenia poprawek w naszych testach, ponieważ zmieniła się struktura samego Itemu, jak i tworzenie kolejnych przedmiotów w sklepie Gilded Rose. Po naprawieniu testów i ich uruchomieniu wszystkie testy zostały wykonane pomyślnie, zatem mamy pewność prawidłowości implementacji naszej refaktoryzacji.

IV Podsumowanie

Po wykonanej refaktoryzacji zdecydowanie prostsze stało się czytanie kodu. Wprowadzenie modyfikacji do któregoś z przedmiotów również nie powinno stanowić już wyzwania. Dodawanie kolejnych przedmiotów jest bardzo proste i równa się ze stworzeniem nowej klasy oraz zdefiniowaniem dla niej UpdateQuality(). Na koniec porównamy metrykę kodu za pomocą narzędzia **Code Metrics** wbudowanego w Visual Studio. Wykorzystamy do pomiaru jakości atrybut **Maintainability** - który mierzy łatwość utrzymania kodu, a w jego skład wchodzi między innymi skomplikowanie kodu, powiązania pomiędzy klasami, liczba linii kodu.

Wartość Maintainability dla oryginalnego Gilded Rose

Wartość Maintainability po refaktoryzacji

Do stwierdzenia jakości rozwiązania użyty został również **Code Climate**.