Kata refactoring

REFACTORING IN C#

Crivici Alexandre | .Net Freelancer | 07-11-2019

Analyse

After reading the requirements and starting application, we can see that this one respects the specifications needed except the new item « Conjured » newly created.

Indeed, by executing the application, we can point these things :

- Each day, the Quality and Sellin values are well lowered/increased depending the item.
- Once the SellIn value is lower than 0, the quality degrades twice as fast.
- Except for the « Aged Brie » for which the Quality is increased the older it gets.
- The Quality of an item is never negative and never more than 50.
- Except for « Sulfuras » items that have a Quality of 80 and never decrease because of their rarity.
- The « BackStage passes » items increase also in Quality each day but the difference is that :
  - The Quality increases twice if there is 10 days left or less before the end of the SellIn value.
  - The Quality increases three times more if there is 5 days left or less before the end of the Sellin value.
  - Quality is set to 0 after SellIn value is passed.
- **The « Conjured » item should decrease twice faster than normal items but it's not the case.**

By checking the code quickly, these are my first constatations on a first sight :

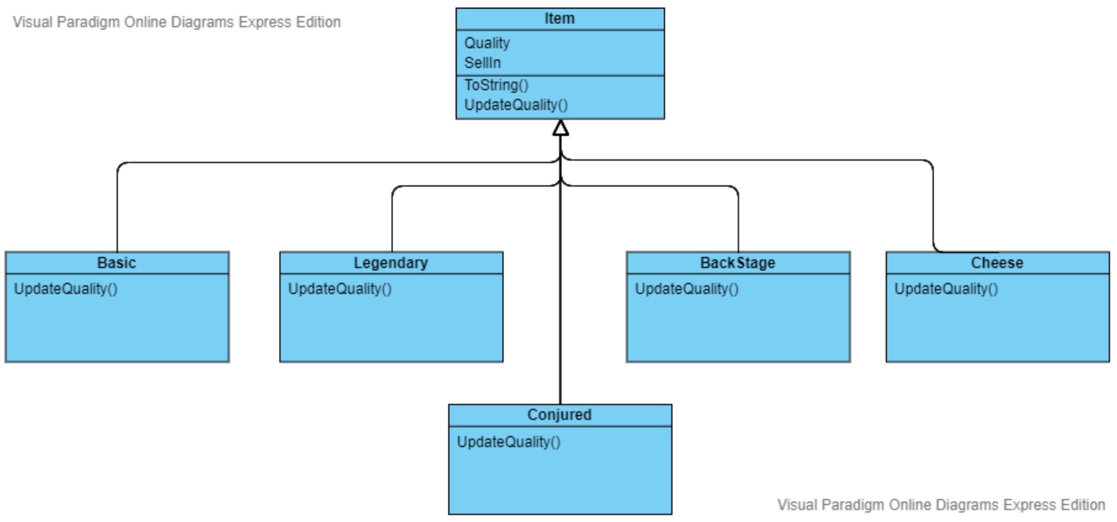
- The classes are not well structured in the Project folder. Even the unit tests are stored in the sale folder.
- The methods are too long and not well structured also. If I ask to my wife (who is not developer 😉) to just read the code and tell me what it does, she can't. Genrally, the name of the methods, variables, etc… must be very explicit and methids must be as readable as we can, to ensure the good transition between developers.
- The code is not very extensible. What if we add other items with other specifications ?? the update method will be huge and more and more unreadable. At the end, it will surely cause bugs.
- The unit tests do not cover every cases and are failing.
- The update of the quality is based on the name of the product. But what if, one day, we change the names of the products, then we have to review all the update method. If we have hundreds items, it will be crazy.

FIRST APPROACH

The first idea that comes to me instinctively is to use the principle of inheritance as described in the diagram below.

I created Item as the superclass and all types of item we could have to classes that inherit this superclass.

The UpdateQuality method is an abstract method into the superclass (So item is also abstract).
All subclasses that inherit this superclass redefine this method for their own purpose.
I let the ToString() method in the superclass as the behaviour is the same for all childs.



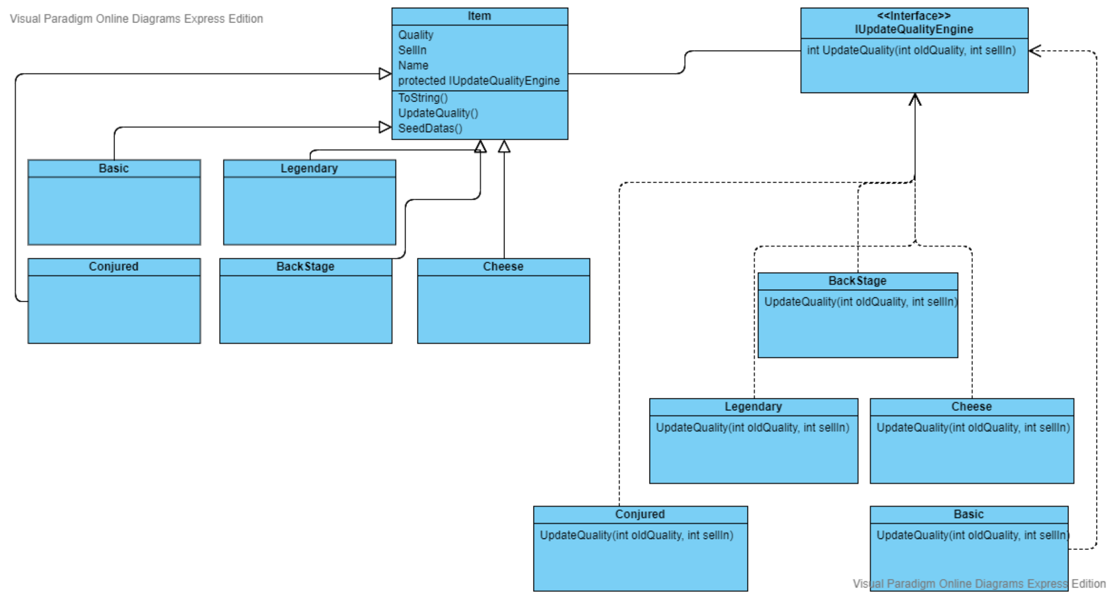
It’s a little bit better but not perfect. Indeed, the code is less complicated, we separate the logic into different type of items but what if the customer reviews the update quality system ? Then, we have to adapt the update method in ALL subclasses. Here, it’s not a problem because we have not so much items, but if we have hundred subclasses, we will have a lot of work and could cause bugs.

SECOND APPROACH

Let’s try now to improve the code. The inconvenient is that we have the Quality that is modified depending the item.

Why don’t we put then the update logic to a separate concern.
If there is only the quality which changes, then, it would be a good idea to add this behaviour to separate classes and implemented by an interface.

Moreover, it has this advantage to be more évolutive. Indeed, if our client needs one day, to add new algorithm to calculate his quality, then we only have to add a new class into the engine, which inherit from the interface, implement the method and add the interface to the right Item, that’s it !


Visual Paradigm Online Diagrams Express Edition
Item
Quality
SellIn
Name
protected IUpdateQualityEngine
ToString()
UpdateQuality()
SeedDatas()
<<Interface>>
IUpdateQualityEngine
int UpdateQuality(int oldQuality, int sellIn)
Basic
Legendary
Conjured
BackStage
Cheese
BackStage
UpdateQuality(int oldQuality, int sellIn)
Legendary
UpdateQuality(int oldQuality, int sellIn)
Cheese
UpdateQuality(int oldQuality, int sellIn)
Conjured
UpdateQuality(int oldQuality, int sellIn)
Basic
UpdateQuality(int oldQuality, int sellIn)
Visual Paradigm Online Diagrams Express Edition

It has also the advantage to be easily modified during the exécution of the program. If I implement a method 'SetUpdateQuality' to the Item class with the Delegation as parameter, then, with only few lines of code, I can switch the method of calculation between each item.

By simply adding this line into Item class:

```
  public void SetQualityCalculation(IUpdateQualityEngine updateQualityEngine)
        {
            this._updateQualityEngine = updateQualityEngine;
        }
```

```
I can now set easily the algorithm between items :

Item backStageItem = new BackStageItem();
backStageItem.SetQualityCalculation(new NormalQualityIncrement());
backStageItem.UpdateQuality();
```

IMPROVEMENTS

Wen could improve maybe the application again by putting the increment variable to the appsettings or a global variable, just in case one day, if the customer tells that SellIn value is depending of another thing that time. In that casen maybe it would be usefull at that moment to put this attribute into an Engine interface like Quality.

But let's try to keep the application as simplest as it must be.