

Lab Report 05

Assignment 05 - Legacy Code - Refactoring to Patterns

Authors: Dennis Loska, Tony Dorfmeister, Ai Dong 28.12.2017

Write Characterization Tests

From the given specification we've created test scenarios for each specification. This has been done by creating various test objects that will test the given specifications. As a tool for code coverage we have used the built-in tool of IntelliJ.

After this we have increased the overall code coverage for class `GildedRose` from 33% to 83% as seen in the images below. (Unfortunately we didn't know how to get rid of the extra packages in the images)

Coverage GildedRoseTest				
85% classes, 72% lines covered in 'all classes in scope'				
	Element	Class, %	Method, %	Line, %
	apple			
	com			
	java			
	javafx			
	javax			
	jdk			
	META-INF			
	netscape			
	oracle			
	org			
	resources			
	sun			
	toolbarBu...			
	GildedRose	83% (5/6)	81% (18/22)	66% (44/66)
	Item	100% (1/1)	100% (7/7)	100% (14/14)

Coverage GildedRoseTest				
85% classes, 85% lines covered in 'all classes in scope'				
Element	Class, %	Method, %	Line, %	
apple				
com				
java				
javafx				
javax				
jdk				
META-INF				
netscape				
oracle				
org				
resources				
sun				
toolbarBu...				
GildedRose	83% (5/6)	95% (21/22)	82% (55/67)	
Item	100% (1/1)	100% (7/7)	100% (14/14)	

By adding some test objects that have already been expired in sell date we were able to bump up the method percentage to 95%.

Coverage GildedRoseTest				
42% classes, 48% lines covered in 'all classes in scope'				
Element	Class, %	Method, %	Line, %	
apple				
com				
java				
javafx				
javax				
jdk				
META-INF				
netscape				
oracle				
org				
resources				
sun				
toolbarBu...				
GildedRose	33% (2/6)	36% (9/25)	37% (25/66)	
Item	100% (1/1)	100% (7/7)	100% (14/14)	

Our guess why the class coverage is only at 83% is that the main method of `GildedRose` is not tested due to the fact that we are using our own items and not the core items the program has implemented.

Refactoring

1. changing for-loop to foreach

- Just so the variables in the methods look more pretty

2. extracting method for better readability

```
public static void updateQuality()
{
    for (Item item : items) {
        updateOneItem(item);
    }
}
```

3. Three big if-else blocks extracted in 3 separate methods for now

- Makes the code **much** easier to understand
- 3 separate methods instead of a huge mess

```
public void updateQuality() {
    Item temp;

    for (Item item : items) {

        if(!(item instanceof ItemCategory)) {
            temp = categorize(item);
            items.set(items.indexOf(item),temp);
            ((ItemCategory)temp).updateSellin();
            ((ItemCategory)temp).updateQuality();
        }

        else {

            ((ItemCategory)item).updateSellin();
            ((ItemCategory)item).updateQuality();
        }
    }
}
```

4. More extracting and changing if-conditions for better understanding (no logical changes)

- More detailed cleaning in the separate methods
- Switching conditions and inverting if-statements
- If-statements are sorted by the name-conditions for now

After more thinking:

- Changing `.equals` for better readability for example
- More extracting like *increaseQuality(item);* instead of *item.setQuality(item.getQuality() - 1);*

```
java
    private static void updateQuality(Item item) {
        if ((item.getName().equals("Aged Brie"))) {
            increaseQuality(item);
        } else if (item.getName().equals("Backstage passes to a TAFKAL80ETC
concert")) {
            if (item.getSellIn() < 11) {
                increaseQuality(item);
            }
            if (item.getSellIn() < 6) {
                increaseQuality(item);
            }
        } else if (item.getName().equals("Sulfuras, Hand of Ragnaros"))
        { //do nothing
        } else {
            decreaseQuality(item);
        }
    }
}
```

5. Running the test-class to ensure, that all the refactoring is working

- That step has been done many times every now and then
- Tests were green most of the time until implementing the strategy pattern

6. Implementing the Strategy Design Pattern

- Adding a interface `ItemCategory`, which will include most of the methods so the `GuidedRose` class is nice and clean.
- Adding children of `Item` for different Items, which will use the methods differently meaning overriding them with a more specific implementation for the item
- This means different operations are executed depending on the item-category
- A more specific item is handled in it's respective category class
- E.g. a **otherItem** item aka a normal item
- The necessary methods are overridden

```
@Override
public void updateQuality() {
```

```
this.quality = this.sellIn<=0? this.quality -= 2: (this.quality - 1); if (quality >50)this.setQuality(50);
```

```
    if (quality<0) this.setQuality(0);  
}
```

- The category is determined by checking the item's name
- If there is no match the default *ItemCategory* will be created

```
public static Item categorize(Item i){  
    Item item = i.getName().contains("Aged Brie"? new  
Cheese(i.getName(), i.getSellIn(), i.getQuality()):  
        i.getName().contains("Sulfuras, Hand of Ragnaros"?  
new Legendary(i.getName(), i.getSellIn(), i.getQuality()):  
            i.getName().contains("Backstage passes to a  
TAFKAL80ETC concert"? new BackstagePass(i.getName(), i.getSellIn(),  
i.getQuality()):  
                i.getName().contains("Conjured"? new  
Conjured(i.getName(), i.getSellIn(), i.getQuality()):  
                    new OtherItem(i.getName(), i.getSellIn(),  
i.getQuality());  
            return item;  
}
```

7. Conclusion

- It is very easy now to implement very specific methods
- Items can be very custom now without changing the Item class
- Many if-statements are unnecessary now and got wiped out during the refactoring process
- The class **ItemCategory** acts as a wrapper for the Item classes
- It would be totally fine to put some, if not all of the methods from it into the Item-class, if that would be allowed

Add the new functionality

- to add a new functionality, the itemCategory needed to be implemented as Interface while every Category is an extension of item.
- with ItemCategory as interface, @override enables every category to implement their own updateQuality() and updateSellIn()
- even though item is public, as we assume a development with persistence frameworks, item would work best as a "base class"
- That way, the class Conjured could have its own implementation
- Additional checks in constructor and updateQuality() guarantees the values are valid.

- Additional checks in UpdateQuality() are made to check whether the item already exist in the list.
- For readability ternary operation are used as right hand sided arguments.

public class Conjured extends Item implements ItemCategory {

```
public Conjured(String name, int sellIn, int quality) {
    super(name,sellIn,quality);

    if (quality >50)this.setQuality(50);
    if (quality<0) this.setQuality(0);
}

@Override
public void updateSellin() {
    this.sellIn -=1;
}

@Override
```

```
public void updateQuality() {

    this.quality = this.sellIn <= 0? this.quality -=4 : this.quality - 2;

    if (quality >50)this.setQuality(50);

    if (quality<0) this.setQuality(0);

}
```

- As the sell-in value changes first before the quality value, these additional checks are necessary.