Refactoring the praxis-gildeddrose project

Item

The Item class is a model, this class is in charge of giving the structure of an Item instance, and it does so, it must not have business logic inside, complying with the SOLID single responsibility principle. On the other hand, this complies with the open/close principle, since, if I want to add a new attribute I only have to do this, which is very comfortable.

ItemService

At the beginning, **ItemService** was the one that handled the business logic, but this service is very long, too long. It had one hundred and forty lines of code and fifty of them corresponded to a single method called **UpdateQuality**. This method was cleaned up by a switch whose responsibility is to invoke the corresponding method to update the quality of each attribute type, in fifteen lines of code.

Then, 4 methods, **UpdateQualityTypeItem***,* are created for each of them. They use 2 additional methods, **decreaseSellIn()** and **IncreaseQuality()** or **decreaseQuality()**. Why do we do it?, because in this way each method applies the single responsibility principle, in addition, the code tracking is easier because they are named appropriately and it is not necessary to see other conditionals.However, they do exist and are named appropriately using terms like **minQuality** or **maxQuality** to verify that an Item does not exceed the accepted quality range, for this instance.

Now, it is important to note that in **ItemService** two services can be separated: the first one is **ItemServices** without the methods related to **updateQuality**, and the second one only the methods related to **updateQuality()**. So now we have a new service (*QualityService*) with **updateQuality()** and other methods like **reduceQuality()**, **increaseQuality()**, **reduceSellIn()** and the updates of each type using the previous ones.

What is the reason for doing this? It is just for the order, complying with the clean code so that it does not spread vertically and applies high cohesion and low coupling, also, if I want to add a new type of Item, I just have to add it in the Item model, in the switch and create the method for its behavior.

In **ItemService** class the **createItem** method was modified to check if the item to create already exists in the database before saving the item. The saved item is assigned to the variable **createdItem** to let clear what returns **itemRepository.save().** When the item to create already exists, is thrown a **DuplicatedFoundItemException** with its respective message to explain what is happening.

The method **checkDuplicatedItems** had the function of checking if a given item has the same attributes as another item in the database, this validation was disaggregated in the method **isDuplicatedItem.** This was done because the operation of comparing all attributes

is called in two different methods and also because it is an independent operation. The name of the booleans to verify if an item already exists was changed and now answer the question: Is this attribute duplicated?. The method that verifies that all the attributes of an item are duplicated has also been segregated in **isAllDuplicated** due to be an extensive conditional**.**

The method **updateItem** has been changed to verify if the item to update exists in the database, otherwise an exception is thrown indicating that the item doesn't exist. This was made in order to take into consideration the most possible exception in this method.

The **listItems** method hasn't been changed because the name itself is well-declared and clear in what the function does and returns.

In the **findById** method the name of the exception is clear but doesn't specify anything when the exception is thrown, so, was added a text indicating the error and the id that doesn't match any item, this to give clarity of what is happening when this error is thrown.

ItemServiceTest

The methods of this class were reorganized so that the tests are put first when a method is executed successfully and then those that do not.

The tests **testGetItemByIdSuccess, testGetItemByIdWhenItemWasNotFound** and **testUpdateQualityOfNormalTypeItem** have not been changed because is considered that the structure, naming and declaration of variables and comparisons are understandable. The **testUpdateQualityOfNormalTypeItem** was moved into the **QualityServiceTest** since there are many tests related only to update the quality of an item. This was made to avoid large classes and have a clearer separation of the tests.

Into the **ItemServiceTest** class has been added all tests related to the **ItemService** methods with the following structure:

- Name and create the necessary items to prove the method
- Then, inside the when, declare what the different methods used inside the method to test should return
- After that, call the method to test into a variable whose name refers to what the method returns
- And finally, validate whether the method returns the expected result through asserts.

This structure gives clarity on what receives and returns the method to be tested.