The Coding Challenge – Checkout

Thank you for performing this exercise to demonstrate your skills. This exercise is for us to see how you approach a problem and structure a solution.

Please invest no more than 4 hours for this exercise. It might be that not everything can be solved in 4 hours and that's okay. This exercise's purpose is to demonstrate your skills and will serve as a discussion starter for the next interview round.

The Problem Statement

Implement the code for a supermarket's checkout system with a customizable pricing schema.

Even though the exercise is not about knowledge of a particular framework, to make reviewing easier for us, please use:

- Java
- Maven

The business rules are following:

- The items in a supermarket are identified by using Stock Keeping Units (SKUs). We will use individual letters of the alphabet (A, B, C, ...).
- Every item has a unit price.
- Some items are multi-priced: if you buy N of them, their subtotal will be Y.
- Items may be a subject of a discount on a particular date: if you buy on DD.MM.YYYY, the item will cost X % less.
- We call the process of reading the list of purchased items "scanning". The items can be scanned in any order and the algorithm should still correctly detect multi-priced items.
- Because the pricing can change frequently, we need to be able to pass in a set of pricing rules each time we start handling a checkout transaction. The format of the rules is not defined.

Example

Prices of the items are as given in the following table:

SKU	Unit Price	Special Price
A	40	3 for 100
B	50	2 for 80
C	25	
D	20	
E	100	10 % off on Black Friday (26. 11. 2021)

If we scan items [B, A, B], the algorithm will recognize 2 Bs with subtotal 0,80 EUR and 1 A with a unit price 0,40 EUR, resulting in the total price of 1,20 EUR.

If we add another B (the whole input would be [B, A, B, B]), the algorithm will apply the special price for 2 Bs, a regular unit price for the 3rd B and a regular unit price for the single A, resulting in the total price of 1,70 EUR.

The Interface & Test Case

The interface to the algorithm should look like this:

```
var checkout = new CheckOut(pricing_rules);

checkout.scan(item);
checkout.scan(item);

price = checkout.total();
```

Here is a Unit test in Java. The helper method `calculatePrice()` lets you specify a sequence of items (using a string), calling the checkout's scan method for each item before returning the total price:

```java
public class TestPrice {
    public int calculatePrice(String goods) {
        CheckOut checkout = new CheckOut(rule);

        for (int i=0; i<goods.length(); i++) {
            checkout.scan(String.valueOf(goods.charAt(i)));
        }

        return checkout.total();
    }

    @Test
    public void totals() {
        assertEquals(0, calculatePrice(""));
        assertEquals(40, calculatePrice("A"));
        assertEquals(90, calculatePrice("AB"));
        assertEquals(135, calculatePrice("CDBA"));

        assertEquals(80, calculatePrice("AA"));
        assertEquals(100, calculatePrice("AAA"));
        assertEquals(140, calculatePrice("AAAA"));
        assertEquals(180, calculatePrice("AAAAA"));
        assertEquals(200, calculatePrice("AAAAAA"));

        assertEquals(150, calculatePrice("AAAB"));
        assertEquals(180, calculatePrice("AAABB"));
        assertEquals(200, calculatePrice("AAABBD"));
        assertEquals(200, calculatePrice("DABABA"));
    }

    @Test
    public void incremental() {
        CheckOut checkout = new CheckOut(rule);

        assertEquals(0, checkout.total);
        checkout.scan("A"); assertEquals(40, checkout.total);
        checkout.scan("B"); assertEquals(90, checkout.total);
        checkout.scan("A"); assertEquals(130, checkout.total);
        checkout.scan("A"); assertEquals(150, checkout.total);
        checkout.scan("B"); assertEquals(180, checkout.total);
    }
}
```

Assessment Criteria

- We are interested in well-structured and clean SOLID code. Your solution should demonstrate skills to write production code.
- For the test cases above, your algorithm should produce correct values. Test cases for item E are missing. Please add your own tests.
- We are especially interested in a foresighted approach that will be easy to customize, extend and maintain in the future. Your solution should demonstrate considerations regarding future extensibility.
- Decisions for your open questions not specified by the problem statement should be explained in accompanying README file. Unfinished work due to time limitations should be explained in accompanying README file.
- We are looking for a simple solution. Keep it reasonably simple, don't add unnecessary complexity that's not requested. More specifically, we do not expect REST API or any kind of persistence. If you do it, we won't penalize you, but it won't give you extra points.

Additional Notes

The exercise doesn't mention the format of the pricing rules:

- How can these be specified in such a way that the checkout doesn't know about items and their pricing strategies?
- How can we make the design flexible enough so that we can add new styles of pricing rules in the future?

Next steps

Send your source code as a .zip file to checkout-dev@idealo.de. **Please do not share the assignment or your solution in public.** After we have reviewed it, you will be informed if or when we invite you to the next interview round.

In this next round, we are going to talk about your solution in the format of a code review. Please, have your IDE ready for the next round and be prepared to share your screen!