Object Orientation – Spring 2018 **Assignment 10**

Patterns

Learning goals

This assignment consists of two parts. These parts were questions on patterns of the 2014 resit exam in a slightly modified form. You should be able to apply the *strategy pattern* and the *decorator pattern* to other problems as well. To be clear: The other patterns from last week's lectures are exam material as well.

1 Time for some ice cream!

In this exercise we will ask you to make an object oriented representation of various sorts of ice cream which can be ordered with or without several types of *toppings*. A topping is an ingredient to finish or garnish your ice cream, such as whipped cream or a dip. For the chocolate dip, the ice cream is briefly dipped into a liquid chocolate sauce. When hardened, the sauce should form a thin layer of chocolate around the ice. In principle, a customer can order as many toppings as he wants - even different layers of the same type of topping. However, you should not dip the ice cream into the chocolate sauce anymore after you added whipped cream. There certainly are plenty other unfavourable combinations, but you do not need to take these into account for this exercise.

We give you the following facts on the ice cream:

1. There are no objects of the basic type `Ice`. There *are* objects of derived types `VanillaIce` and `YoghurtIce`.

2. Each `Ice` has a *description* and a *price*. For the former, we will use a `protected` attribute of type `String` which we will call `description`, whereas we will model the latter using a parameter-free (abstract) method called `price` that returns an `int` (the price in cents). We will introduce a method called `giveDescription` to show the description to the customer. For the class `Ice` itself, this method should return the string *"unknown ice"*. The classes `VanillaIce` and `YoghurtIce` should assign a new, suitable value to `description` and define the `price` of the item. The price for a `VanillaIce` is 150 cents whereas a `YoghurtIce` costs 200 cents.

3. Every portion can be garnished with a topping if the customer wishes one. There are three kinds of toppings: `WhippedCream`, `ChocolateDip` and `Sprinkles`. The price for whipped cream is 50 cents, the price for a chocolate dip is 30 cents and the sprinkles are free.

To model the ice creams, you naturally want (and, in our case, are going) to use the so called *decorator* pattern. This means you should represent the basic type `Ice` and the concrete variants the usual way. For the toppings, you will introduce a suitable abstract class (called `Topping`, for example) that extends `Ice`, and which takes an attribute of the type `Ice`. In fact, this is how the decorator pattern works! The attribute is used to store the ice that is garnished with a certain type of topping. Other than that, all toppings are extensions of the class `Topping`.

1.a) Implement the Java classes and -if necessary- some additional interfaces that you need for this task.

1.b) Give a few examples of actual, garnished portions of ice cream by printing the description and the price.

2 Online Store

In this exercise you are going to design an object oriented version of an online store. The customer of the store gets assigned an empty *shopping cart* when he enters the website. During shopping, the shopping cart is filled with *items* until the customer decides to pay at the online store's cash register, so the store can send the items to him/her. In practice, the shipping method depends on the size and weight of the products. In our application, we are only interested in the shipping cost that is added to the total price in the end. Each `Item` comprises a `description` (represented as a String) and a `price` (for which we will use a `double`). On top of that, you can query the `shippingCost` for each item. The `description` and the `price` are defined as attributes. The `shippingCost` will be represented through an abstract method that returns a `double`. The concrete items get an implementation of this method.

The items are collected in a `ShoppingCart`. This shopping cart contains methods to add and remove items, to compute the total price (including shipping) and to complete the paying process. The type of payment is determined with the help of a `strategy pattern`, where the payment method is defined through the abstract method `boolean pay(double amount)`. The `boolean` return value indicates whether the payment was successful. There are different methods of payment: `IDeal`, `CreditCard` and `PayPal`. For `IDeal` you need a bank, an account number and a pin number. For the `CreditCard` you need a card number and an expiration date. For a `PayPal` payment you need an e-mail address and a password. The default payment method is `IDeal`. It can be changed with the method `changePaymentMethod`.

Naturally, not every item is dispatched on its own. When determining the sipping costs, you can assume that all items with equal shipping costs are dispatched together and the costs only incur once. To make it a bit more realistic, we presume you to implement these three concrete items: water melon (price €4,50 and shipping €6,75), wine glasses (price €8,50 and shipping €6,75) and washing machines (price €499 and shipping €30,00).

2.a) Implement the classes described above. Add a suitable constructor to each class. You only have to implement *one* of the concrete items and payment methods. Design the methods as simple as possible. It is for instance sufficient for the method `pay` to print out the payment method and the total price.

Hand in

Before Sunday May 6th, 23:59 on Blackboard.