Decorator Pattern

1. You are tasked with modeling the checkout procedure for an online store. In particular, you need to design a program that computes the total costs for customers' purchases. To get started, you should have a class `Checkout` that has a method `getTotalCost()`. You should have another class `SubtotalCost` that has a method `getCost()` returning the cost of the purchase before accounting for any additional costs such as those from taxes or shipping. To account for those additional costs, your boss tells you to follow the SOLID design principles and create two separate classes, one to handle the costs from taxes, and one for the costs from shipping.

   (a) (2pts) Which SOLID design principle is your boss referring to?

   (b) (4pts) Draw a class diagram involving the classes described above, and any others that you want, that could be implemented to compute the total cost for a purchase. Your model does not need to handle the computation of the subtotal cost. Instead, just focus on how to relate the `SubtotalCost` class with the others that handle costs from taxes and shipping. Include pseudocode for the implementation of `Checkout::getTotalCost()`.

   (c) (4pts) Now you need to extend your design to allow for coupons. In particular, you need to implement a $1 OFF coupon and a 10% OFF coupon. Customers are allowed to apply these coupons in any order they choose, and it is possible to apply multiple coupons of the same type. For instance if the subtotal cost is $21 then the customer could first apply a $1 OFF to bring the cost to $20, then a 10% OFF to get to $18, and finally another $1 OFF to get to $17. Of course, the smart customer with those three coupons would first apply the 10% OFF followed by the two $1 OFF's to get the pre-tax, pre-shipping cost from $21 to $17.90. Update your class diagram to include classes for these two types of coupons.

(d) (4pts) Draw a class diagram for the program in the CoCalc folder `Decorator/Coupons`.

(e) (3pts) Which design (yours or the one in CoCalc) is better in terms of the Single Responsibility Principle? Explain.

(f) (3pts) Extend the class diagram above by adding classes `DollarOffDecorator` and a `TenPercentOffDecorator` appropriately.

(g) (3pts) Which design (yours or the one in CoCalc) is better in terms of the Open-Closed Principle? Explain.

(h) (10pts) Implement the classes `DollarOffDecorator` and `TenPercentOffDecorator` into the program in CoCalc. As usual, you can check that your program is setup correctly by playing with `Main.cpp`.

2. (a) (2pts) Copy down the *intent* of the Decorator pattern according to the GoF:

(b) (4pts) Draw the class diagram for the Decorator pattern discussed in class.

(c) (2pts) What is the (single) responsibility of the `Decorator` class?

(d) (2pts) How can a concrete decorator add behavior to its inner-component?

(e) (2pts) Can a concrete decorator replace its inner-component's behavior with some entirely new behavior? Explain.

3. Consider the Mario program from our discussion of the State Pattern. We can use the Decorator pattern to allow Mario to have extra abilities. For example, Mario can power up with a flower to get into a `FireMarioState` which gives Mario the ability to shoot fire; or Mario can power up with a ★ to get into an `InvincibleMarioState` where taking a hit does no damage to Mario.

   (a) (5pts) Draw the class diagram for the Mario program with the addition of `MarioState` decorators. Include (at least) the two decorators described above.

   (b) (5pts) Explain why the Decorator pattern is appropriate here. In particular, why is it better to say `FireMarioState` (or `InvincibleMarioState`) is a `MarioStateDecorator` as opposed simply saying it is a `MarioState`?

4. Consider the following program that makes use of the Decorator pattern:

   - There is a class `Message` with an abstract method `print()`.
   - There is a concrete subclass of `Message` called `SimpleMessage` which has a data member that holds a message as a string.
   - There are two concrete message decorators called `Subject` and `Signature`. Both of these classes have string data members that hold the subject/signature for a message.

   (a) (5pts) Draw a class diagram for the program described above.

   (b) (10pts) Implement this program in the CoCalc folder `Decorator/Message` by completing all of the TODO items in that folder. You should only have to modify the following files:

   `MessageDecorator.h` `MessageDecorator.cpp` `Subject.cpp` `Signature.cpp`

   When you're done, running the program should produce something like the following in the terminal:

```
---------------------------
Subject: (none)
---------------------------

Hello World!

From:
Anonymous

---------------------------
Subject: wrappers
---------------------------

A decorator is a wrapper, but not all wrappers are decorators.

From:
Dr. Comes
MAPS Department
The College of Idaho
```