Decorator Pattern

Purpose

Wrap objects with decorators to modify functionality. Modifications can be done dynamically in runtime, independent of other instances, by wrapping an instance with another instance.

Motivation

If specific instances of an object need modification/augmentation, subclassing repeatedly becomes unscalable.

Intended use case

When many instances of an object are created at runtime, and each instance has its own specific set of properties slightly different from one another, the decorator pattern is useful, by using multiple decorators.

Decorators modify the behaviour of the base component under specified conditions during runtime, usually in non-overriding ways.

Example

In for a sandwich object, decorators could apply different toppings of variable cost to the sandwich, modifying its final cost. One could also make subclasses for each different combination of topping, but this would take 24 subclasses just for 4 toppings.
Note: The toppings decorator would be subclass of the sandwich class, which is slightly confusing, as semantically, a topping is not a sandwich.

Vocabulary

Component: Defines the interface for object that is to be decorated
Concrete Component: Implements the Component
Decorator: Provides an interface that conforms to the Component
Concrete Decorator: Implements the Decorator

Specific structure

Typically, an interface for the component, some number of concrete undecorated “base” class, and an abstract decorator class, all of the same type as the component.

Consequences

1. **Positive**
    - Avoids class explosion problem (avoids very heavy subclassing)
    - More flexible than inheritance
2. **Negative**
    - Decorators make instantiating somewhat more complex, as you may need to wrap the base component in many decorators
    - Many decorators are more difficult to maintain, as each decorator tends to be similar to other decorators, as opposed to distinct subclasses
    - Decorators can cause issues if the client relies heavily on the components concrete type

NFPs improved

1. **Increase scalability**
   The decorator pattern makes code more extensible. Changing the base class does not require changing multiple subclasses.
2. **Increase readability**
   The decorator pattern makes code cleaner and more cohesive, as unrelated code doesn't need to worry about the specific details of the component instance.
3. **Increase adaptability**
   It supports runtime changes on individual objects.

NFPS degraded

1. **Increases complexity**
   It adds many small decorator classes that are easy to overlook during maintenance due to how similar they may be with one another.
2. **Decreases Reusability**
   Leads to heavy copy-pasting by the nature of having many small decorator classes

Sample code

```
interface Component {
     public void getDescription();
}

public class ConcreteComponent implements Component {
     @Override
     public void getDescription() {
          System.out.println("Base");
     }
}

abstract class ComponentDecorator implements Component {
     protected Component component;

     public ComponentDecorator(Component component) {
          this.component = component;
     }
     @Override
     public void getDescription() {
          component.getDescription();
     }
}

public class DecoratorType1 extends ComponentDecorator {
     public DecoratorType1(Component component) {
          super(component);
     }
     @Override
     public void getDescription() {
          System.out.print("Decorator1, ");
          super.getDescription();
     }
     private void modifyComponent() {
     // modifies component in a way that the base behaviour of
component does not change
     }
}
```

Presentation:

Scenario: Coffee Shop
How to implement the ordering system?

1) Subclass
   1 seperate beverage listing on the menu per different combination of mixins
   a) List Coffee, Tea, Coffee1Sugar, Coffee1Milk, Coffee1Cream, Coffee2Milk1Sugar, Tea1Milk1Sugar
   b) Note that just a few mixins make this extremely tedious to work with
   c) Suppose a customer orders an unusual order, like a chai green tea latte with soy milk, non-caloric sweetener and a chocolate drizzle
2) Flag variables
   Flag each possible modification on the cup itself through checkboxes
   a) isCoffee, isTea, howMuchMilk, howMuchSugar etc
   b) Somewhat tedious to work with, have to check flags individually
   c) Suppose a new seasonal mixin PumpkinSpice is added
      i) Now have to reprint the cups to add checkboxes to support new mixin (that is, refactor each base class to support)
3) Decorator pattern
   a) Write 1s2c2m cf on cup to represent coffee with 1 sugar, 2 cream, 2 milk
   b) Beverage interface, Coffee, Tea, Latte base classes, Sugar, Cream, Milk mixins
   c) I.e. Double double coffee = Sugar(Sugar(Cream(Cream(Coffee)))) = 2s2c cf
   d) To add pumpkin spice
      i) PumpkinSpice(Latte) = ps lt