The Decorator Pattern

Updated: 3/6/2014

Based on Head-First Design Patterns, Chapter 3
Overview

- Starbuzz case study
- Drawbacks to using inheritance
- Open-Closed Principle
- Decorator Pattern
- Java Code for the Decorator Pattern
Starbuzz Coffee Case Study

Originally, their classes looked like this:

```java
class Beverage {
    String getDescription();
    abstract double cost();
}

class HouseBlend extends Beverage {
    double cost();
}

class DarkRoast extends Beverage {
    double cost();
}

class Decaf extends Beverage {
    double cost();
}

class Espresso extends Beverage {
    double cost();
}
```

Each subclass defines its own cost and getDescription methods.
Adding Condiments

• But what happens when they begin to offer all sorts of condiments, like whipped cream, milk, soy, and mocha?

• Should they create a separate subclass for each coffee with specific condiments?
  – each has its own price
  – each has a description
Idea: Adding methods

• What if we add a separate method to the Beverage class for each condiment?
  – hasMilk()
  – hasSoy()
  – hasMocha()
  – setMilk()
  – setSoy()
  – setMocha()
  – …and so on….

Maybe. But each drink subclass will have to call all of these methods in order to compute the price.

If new types of drinks and condiments are added in the future, will our classes have to be modified?
Open-Closed Principle

• Classes should be open for extension, but closed for modification.

Why do you think this is a good idea?

Should one's entire program be open to extension, without modifying existing classes?
Meet the Decorator Pattern

1. Start with a DarkRoast object.
2. Decorate it with a Mocha object.
3. Decorate it with a Whip object.
4. Call the cost() method and rely on delegation to add on the condiment costs.

Each decorator object will ask for the cost of the object it decorates.
Useful Observations

- Decorators often have the same supertype as the objects they decorate.
- You can use one or more decorators to wrap an object.
- Decorated objects can be passed to methods using their base type.
- You can decorate objects at runtime.
The Decorator Pattern

- The Decorator Pattern attaches additional responsibilities to an object dynamically.
- Decorators provide a flexible alternative to subclassing when you just want to extend the functionality of an object.
Each of the CondimentDecorator-derived objects holds a reference to the Beverage object it decorates.
public abstract class Beverage {
    String description;

    public String getDescription() {
        return "Unknown";
    }

    public abstract double cost();
}

public abstract class CondimentDecorator extends Beverage {
    public abstract String getDescription();
}

This could be an abstract method

Actual CondimentDecorator objects will override both cost() and getDescription()
public class Espresso extends Beverage {
    public Espresso() {
        description = "Espresso";
    }
    public double cost() {
        return 2.00;
    }
}

public class HouseBlend extends Beverage {
    public Espresso() {
        description = "House Blend";
    }
    public double cost() {
        return 1.00;
    }
}
class Mocha extends CondimentDecorator {

    Beverage beverage;

    public Mocha(Beverage bev) {
        this.beverage = bev;
    }

    public String getDescription() {
        return beverage.getDescription() + " , Mocha";
    }

    public double cost() {
        return .20 + beverage.cost();
    }
}

The decorated object is passed to this constructor.

Example of delegation.
public static void main(String args[]) {

    Beverage bev = new Espresso();
    System.out.println(bev.getDescription() + bev.cost());
    Beverage bev2 = new DarkRoast();
    bev2 = new Mocha(bev2);          // first decoration
    bev2 = new Mocha(bev2);          // wrap with a second Mocha
    bev2 = new Whip(bev2);           // give it a whip
    System.out.println(bev2.getDescription() + bev2.cost());
}
For Practice

• The coffee shop wants to introduce three sizes (Tall, Grande, Venti). They have added the getSize and setSize methods to the Beverage class. They want condiments to be charged according to size, so for instance, Soy costs 10 cents, 15 cents, and 20 cents, respectively. How would you alter the decorator classes to handle this change?
Final Thoughts

• This application is extendable because:
  – we can add new drink types
  – we can add new condiments
  – we can add new ways of enhancing the drinks
• No existing classes need to be modified when we create these extensions.