The Decorator Pattern

The Decorator Pattern attaches additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

Toni Sellarès
Universitat de Girona

Welcome to Starbuzz Coffee!

Because Starbuzz Coffee have grown so quickly, they are scrambling to update their ordering system to match their beverage offerings....

Beverage is an abstract class, subclassed by all beverages offered in the coffee shop.

The cost () method is abstract; subclasses need to define their own implementations.

The description instance variable is set in each subclass and holds a description of the beverage like: "Most Excellent Dark Roast".

Each subclass implements cost () to return the cost of the beverage.
Adding on....

In addition to your coffee you can also ask for several condiments like steamed milk, soy, mocha etc. Starbuzz charges a bit for each of these so they really need to get them built into the order system. First attempt.....

Can you say “Class Explosion” ?!!!

Alternatives to the Design?

It is pretty obvious that Starbuzz has created a maintenance nightmare for themselves: what happens when the price of milk goes up? or when they add a new caramel topping?

Can’t we just use instance variables and inheritance in the superclass to keep track of the condiments?
Alternatives to the Design?

What requirements or other factors might change that will impact this design?

1) Price changes for condiments will force us to alter the existing code.
2) New condiments will force us to add new methods and alter the cost method in the superclass.
3) We may have new beverages. For some of these beverages the condiments may not be appropriate.
4) What if a customer wants a double mocha?

Meet the Decorator Pattern

We start with a beverage and “decorate” it with the condiments at runtime. If a customer wants a Dark Roast with Mocha and Whip we do the following:

1. Take 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.

How do you “decorate” and how does delegation come into this?
Constructing a drink order with Decorators

1. Start with the DarkRoast object

2. Customer wants Mocha, so we create a Mocha object and wrap it around the DarkRoast.

3. The customer also wants Whip, so we create a Whip decorator and wrap Mocha with it.

4. Now it's time to compute the cost for the customer. Do this by calling `cost()` on the outermost decorator, Whip, and Whip is going to delegate computing cost to the objects it decorates. Once it gets a cost, it will add on the cost of the Whip.
So what do we know so far?

- Decorators have the same *supertype* as the objects that they decorate.
- You can use one or more decorators to wrap an object.
- Given that the decorator has the same *supertype* as the object it decorates, we can pass around a decorated object in place of the original object.
- The decorator adds its own behavior either before and/or after delegating to the object its decorates to do the job.
- Objects can be decorated at any time, so we can decorate objects at runtime with as many decorators as we like.

Key point!

Decorator Pattern

The **Decorator Pattern** attaches additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.
Decorator Pattern: Participants

- **Component**
  
  Define the interface for objects that can have responsibilities added to them dynamically.

- **ConcreteComponent**
  
  Defines an object to which additional responsibilities can be attached.

- **Decorator**
  
  Maintains a reference to a Component object and defines an interface that conforms to Component’s interface.

- **ConcreteDecorator**
  
  Adds responsibilities to the component.

Decorator Pattern: Structural Code

```java
public class DecoratorTest {
    public static void main(String[] args) {
        System.out.println("Decorator : Test");
        Component decorated1 = new ConcreteDecoratorA();
        decorated1.action();
        Component decorated2 = new ConcreteDecoratorB();
        decorated2.action();
    }
}

public class Decorator implements Component {
    Component component = new ConcreteComponent();
    public void action() {
        component.action();
    }
}
```

![Decorator Pattern Diagram](image-url)
public class ConcreteDecoratorA extends Decorator {
    String addedVariable;
    public void action() {
        super.action();
        System.out.println("ConcreteDecoratorA.action() called.");
        addedVariable = "extra";
        System.out.println("ConcreteDecoratorA.addedVariable="
        + addedVariable);
    }
}

public class ConcreteDecoratorB extends Decorator {
    public void action() {
        super.action();
        System.out.println("ConcreteDecoratorB.action() called.");
        addedBehavior();
    }
    private void addedBehavior() {
        System.out.println("ConcreteDecoratorB.addedBehavior ()
        called.");
    }
}

public class ConcreteComponent implements Component {
    public void action() {
        System.out.println("ConcreteComponent.action() called.");
    }
}

public interface Component {
    public void action();
}
Summary

- The Decorator Pattern is based on the open-closed principle!
  - We should allow behavior to be extended without the need to modify existing code.

- The Decorator Pattern
  - Provides an alternative to subclassing for extending behavior.
  - Involves a set of decorator classes that are used to wrap concrete components
  - Decorator classes mirror the types of the components they decorate.
  - Decorators change the behavior of their components by adding new functionality before and/or after method calls to the component.
  - You can wrap a component with any number of decorators.
  - Decorators are typically transparent to the client of the component -- unless the client is relying on the component’s concrete type.
  - Decorators can result in many small objects in our design, and overuse can be complex!

Decorate the Beverages!

Beverage acts as our abstract Component class

The four concrete components, one per coffee type.

The condiment decorators. Notice they need to implement the cost ( ) as well as the getDescription ( ).
Some Code

public abstract class Beverage {
    String description = "Unknown Beverage";
    public String getDescription () {
        return description;
    }
    public abstract double cost ();
}

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

Coding Beverages

public class Expresso extends Beverage {
    public Expresso () {
        description = "Expresso";
    }
    public double cost () {
        return 1.99;
    }
}

public class HouseBlend extends Beverage {
    public HouseBlend () {
        description = "HouseBlend";
    }
    public double cost () {
        return .89;
    }
}
### Coding Condiments

```java
public class Mocha extends CondimentDecorator {
    Beverage beverage;
    public Mocha(Beverage beverage) {
        this.beverage = beverage;
    }
    public String getDescription() {
        return beverage.getDescription() + " Mocha";
    }
    public double cost() {
        return .20 + beverage.cost();
    }
}
```

We want our description to say not only Dark Roast -- but to also include the item decorating each beverage for instance: Dark Roast, Mocha. So we first delegate to the object we are decorating to get its description, then append " Mocha" to that description.

### Ordering Coffee

```java
public class StarbuzzCoffee {
    public static void main(String args[]) {
        Beverage beverage = new Espresso();
        System.out.println(beverage.getDescription() + " $ " + beverage.cost());
        Beverage beverage2 = new DarkRoast();
        beverage2 = new Mocha(beverage2);
        beverage2 = new Mocha(beverage2);
        beverage2 = new Whip(beverage2);
        System.out.println(beverage2.getDescription() + " $ " + beverage2.cost());
    }
}
```