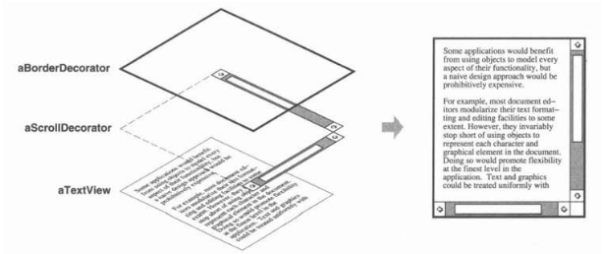
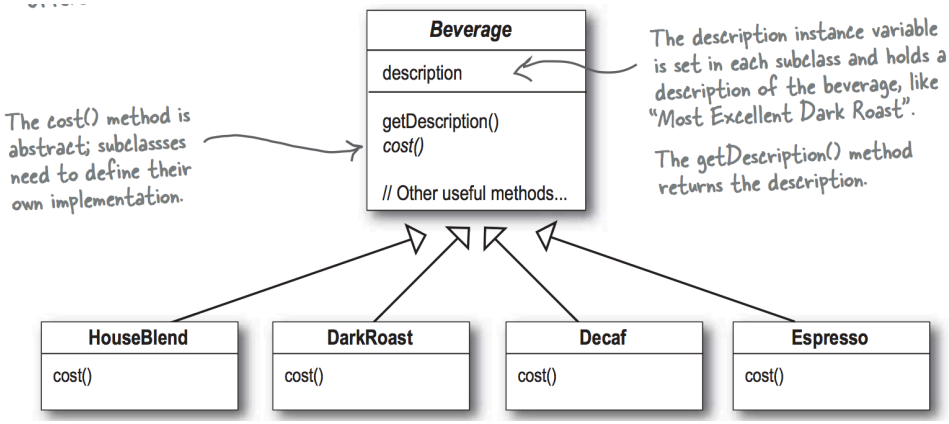
Decorator

Decorator

- Intent
  - Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

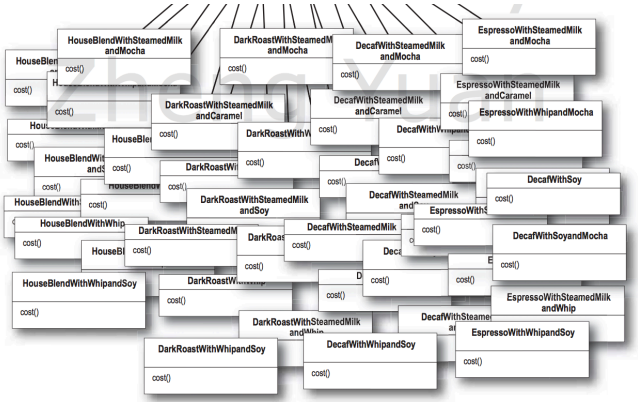


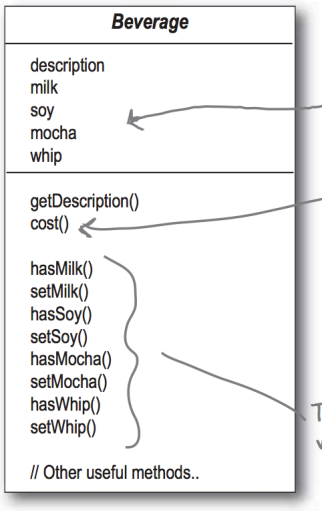
Starbuzz Coffee



Condiments

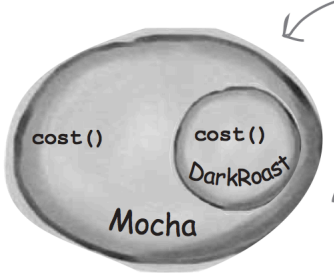
- Mocha, milk, sugar, cream, etc.





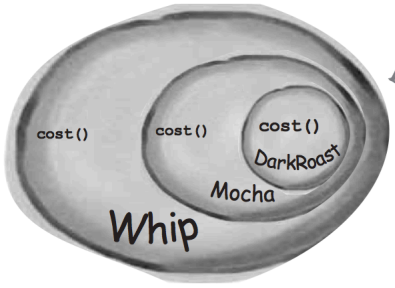
Problems

- New condiments will force us add new methods and change the cost() method
- New beverages will inherit all the condiment method although some may not apply to them (iced tea and cream?)
- What if somebody wants double mocha?



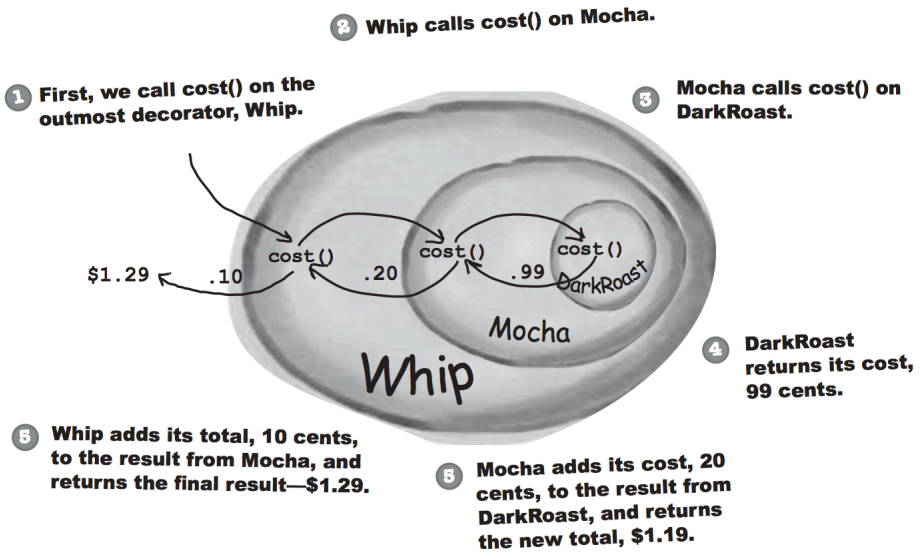
The Mocha object is a decorator. Its type mirrors the object it is decorating, in this case, a Beverage. (By "mirror", we mean it is the same type...)

So, Mocha has a cost() method too, and through polymorphism we can treat any Beverage wrapped in Mocha as a Beverage, too (because Mocha is a subtype of Beverage).



Whip is a decorator, so it also mirrors DarkRoast's type and includes a cost() method.

So, a DarkRoast wrapped in Mocha and Whip is still a Beverage and we can do anything with it we can do with a DarkRoast, including call its cost() method.



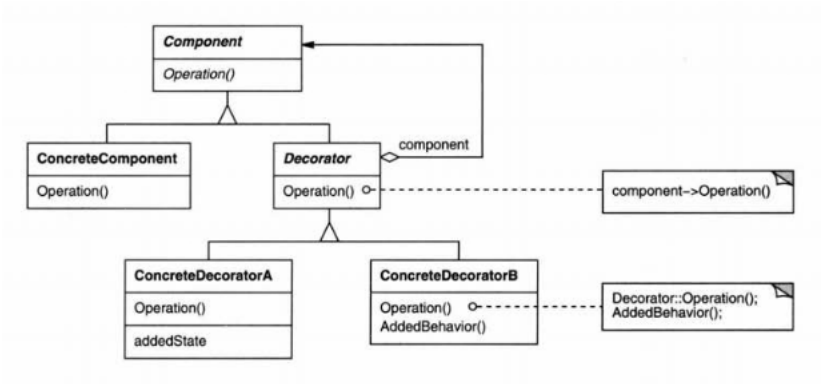
Decorator

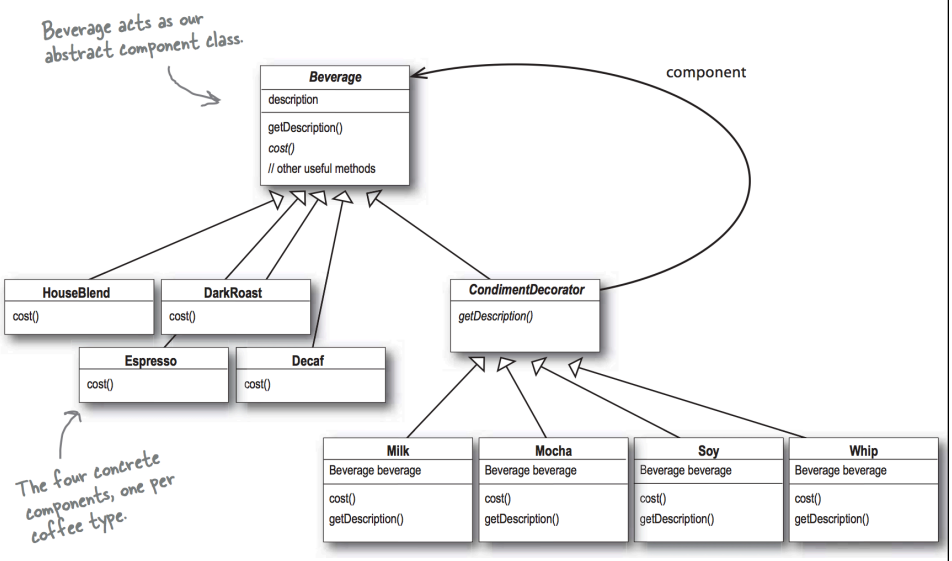
- Add responsibilities to individual objects, not to an entire class.
- Inheritance is inflexible, because the choice is made statically. A client can't control how and when to decorate the component.

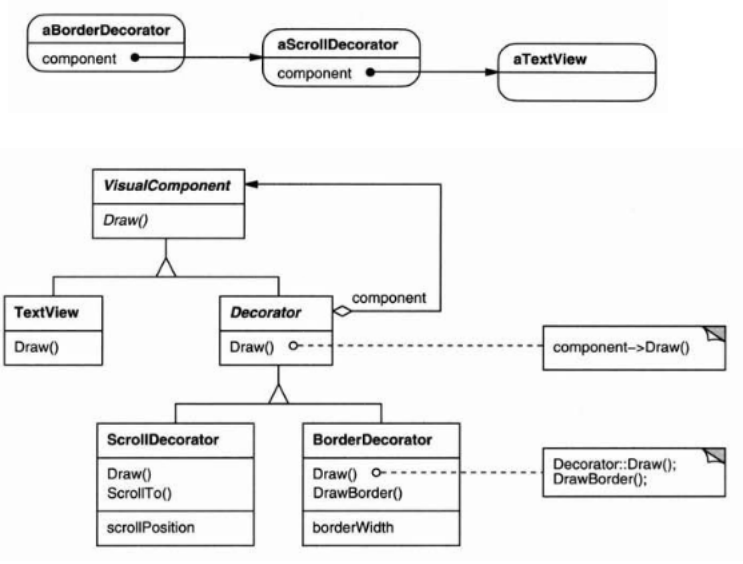
Decorator

- The decorator conforms to the interface of the component it decorates so that its presence is **transparent** to the component's clients.
- The decorator forwards requests to the component and may perform additional actions (such as drawing a border) before or after forwarding.
- Transparency lets you nest decorators recursively, thereby allowing an unlimited number of added responsibilities.

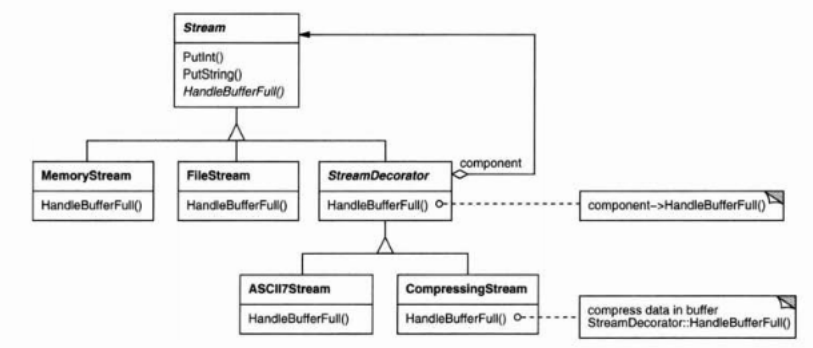
Structure




Beverage acts as our abstract component class.
Beverage
description
getDescription()
cost()
// other useful methods
component
HouseBlend
cost()
DarkRoast
cost()
Espresso
cost()
Decaf
cost()
CondimentDecorator
getDescription()
Milk
Beverage beverage
cost()
getDescription()
Mocha
Beverage beverage
cost()
getDescription()
Soy
Beverage beverage
cost()
getDescription()
Whip
Beverage beverage
cost()
getDescription()
The four concrete components, one per coffee type.


aBorderDecorator
component
aScrollDecorator
component
aTextView
VisualComponent
Draw()
TextView
Draw()
Decorator
Draw()
component
component->Draw()
ScrollDecorator
Draw()
ScrollTo()
scrollPosition
BorderDecorator
Draw()
DrawBorder()
borderWidth
Decorator::Draw();
DrawBorder();

Another Example




```
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();
    }
}
```

```
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());

        Beverage beverage3 = new HouseBlend();
        beverage3 = new Soy(beverage3);
        beverage3 = new Mocha(beverage3);
        beverage3 = new Whip(beverage3);
        System.out.println(beverage3.getDescription()
                + " $" + beverage3.cost());
    }
}
```

Order up an espresso, no condiments and print its description and cost.

Make a DarkRoast object.

Wrap it with a Mocha.

Wrap it in a second Mocha.

Wrap it in a Whip.

Finally, give us a HouseBlend with Soy, Mocha, and Whip.

Applicability

- Use Decorator
  - to add responsibilities to individual objects dynamically and transparently, that is, without affecting other objects.
  - for responsibilities that can be withdrawn.
  - when extension by subclassing is impractical. (e.g: an explosion of subclasses to support every combination.)

Liabilities

- A decorator and its component aren't identical. You shouldn't rely on object identity when you use decorators.
- Lots of little objects that differ only in the way they are interconnected, not in their class or in the value of their variables. Can be hard to learn and debug.