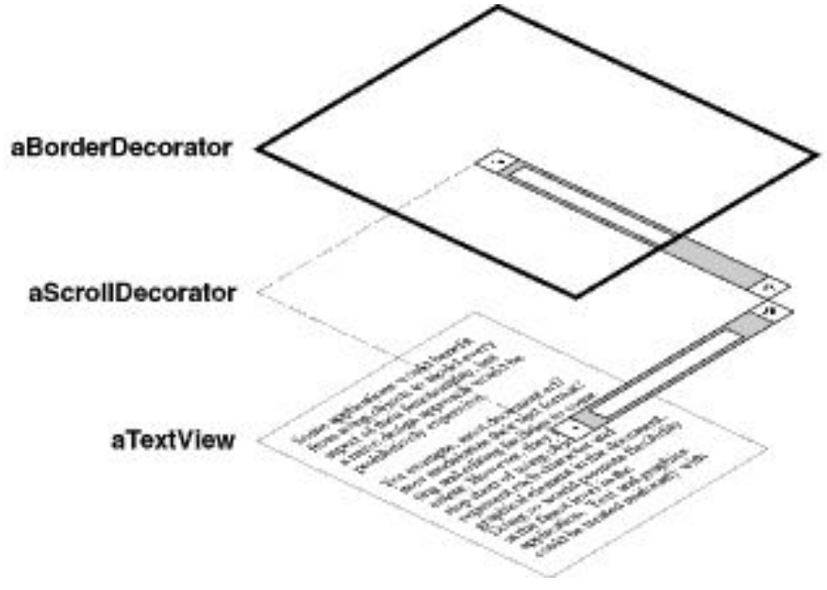
CMPE 202

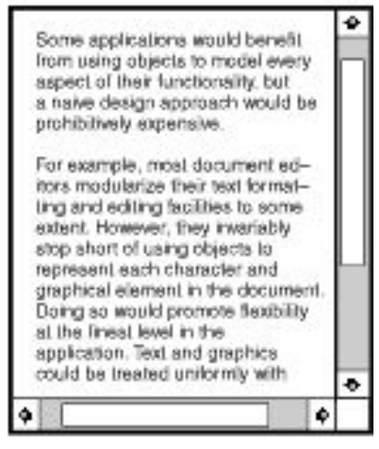
Gang of Four Design Patterns

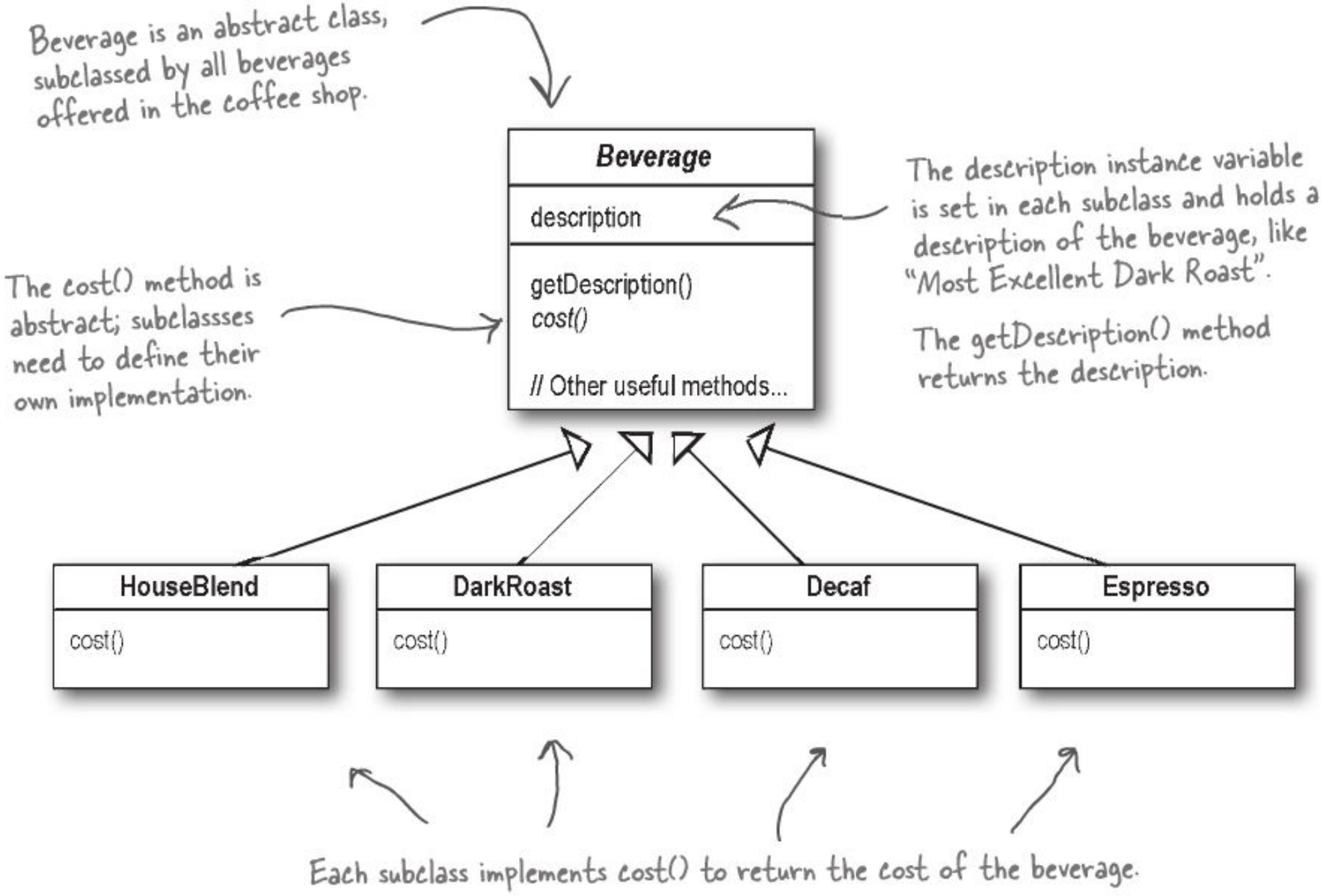
Decorator

Motivation

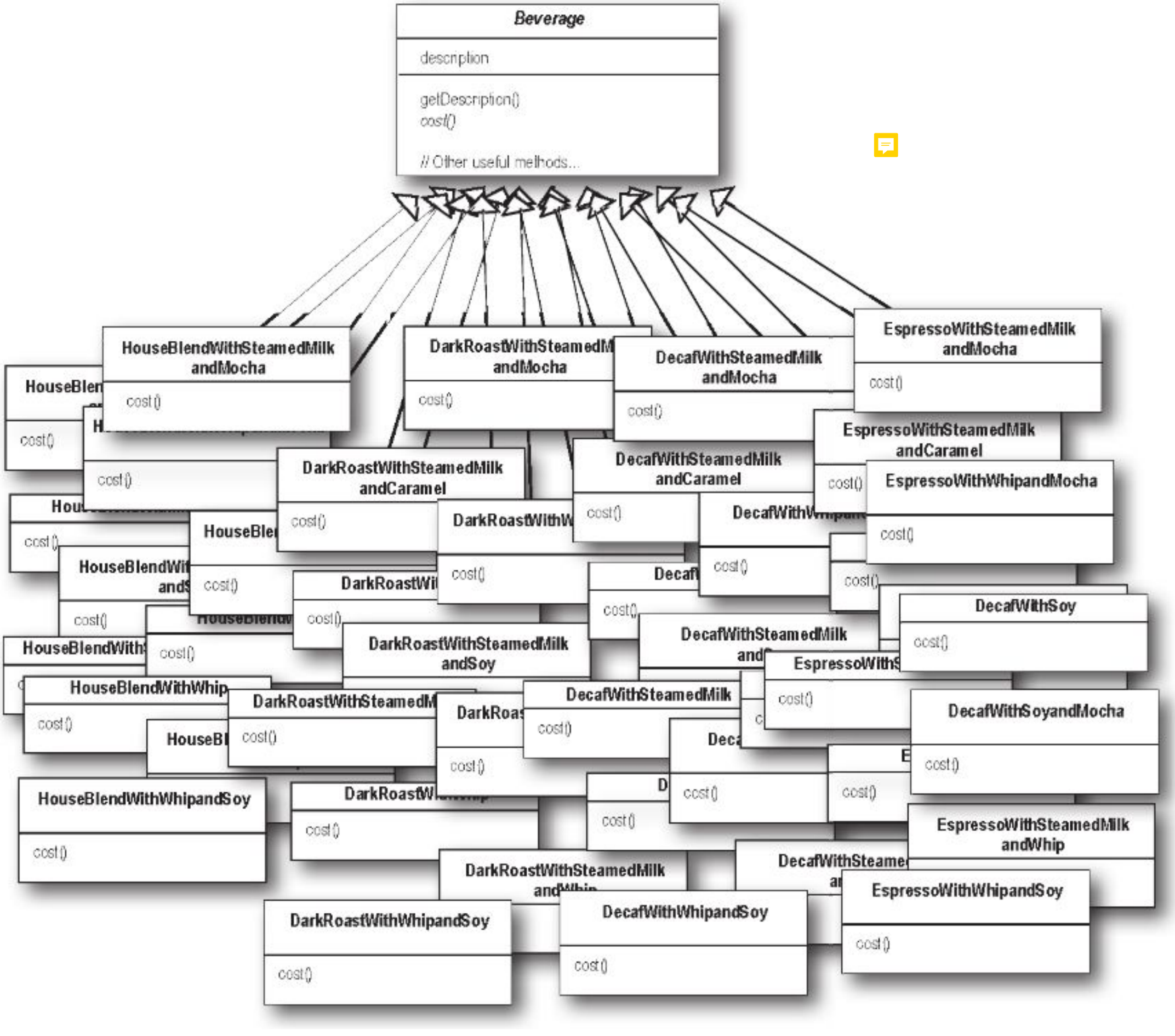
- Want to be able to add responsibilities to individual objects and not to all objects (i.e. the entire class)







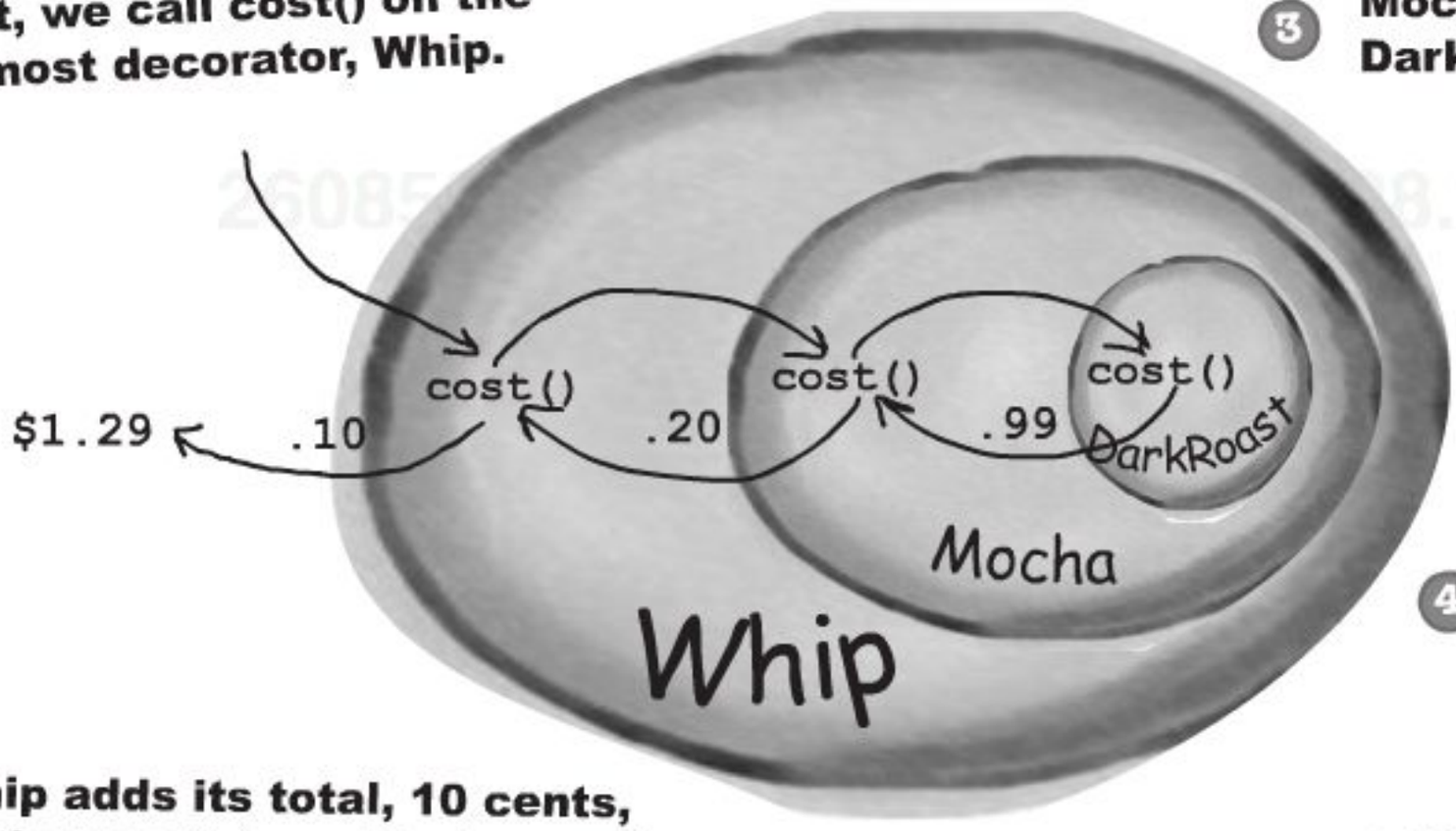
In addition to your coffee, you can also ask for several condiments like steamed milk, soy, and mocha (otherwise known as chocolate), and have it all topped off with whipped milk. Starbuzz charges a bit for each of these, so they really need to get them built into their order system.


Beverage
description
getDescription()
cost()
// Other useful methods...
HouseBlendWithSteamedMilk andMocha
cost()
DarkRoastWithSteamedMilk andMocha
cost()
DecafWithSteamedMilk andMocha
cost()
EspressoWithSteamedMilk andMocha
cost()
DarkRoastWithSteamedMilk andCaramel
cost()
DecafWithSteamedMilk andCaramel
cost()
EspressoWithSteamedMilk andCaramel
cost()
EspressoWithWhipandMocha
cost()
DecafWithSoy
cost()
DecafWithSoyandMocha
cost()
DarkRoastWithSteamedMilk andSoy
cost()
HouseBlendWithWhipandSoy
cost()
DarkRoastWithSteamedMilk andWhip
cost()
DarkRoastWithWhipandSoy
cost()
DecafWithWhipandSoy
cost()
EspressoWithSteamedMilk andWhip
cost()
EspressoWithWhipandSoy
cost()

2 Whip calls cost() on Mocha.

1 First, we call cost() on the outmost decorator, Whip.

3 Mocha calls cost() on DarkRoast.



4 DarkRoast returns its cost, 99 cents.

5 Whip adds its total, 10 cents, to the result from Mocha, and returns the final result—$1.29.

5 Mocha adds its cost, 20 cents, to the result from DarkRoast, and returns the new total, $1.19.

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. Sometimes a large number of independent extensions are possible and would produce an explosion of subclasses to support every combination. Or a class definition may be hidden or otherwise unavailable for subclassing.

Participants

- **Component** (Interface)
  - defines the interface for objects that can have responsibilities added to them dynamically.

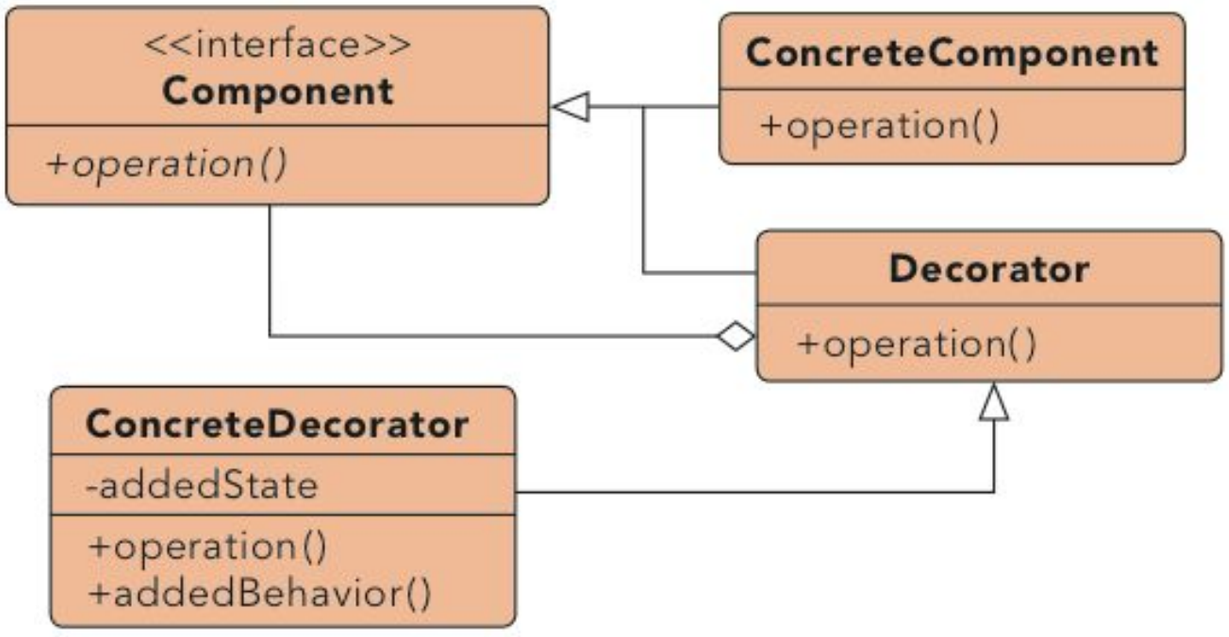
- **ConcreteComponent**
  - defines an object to which additional responsibilities can be attached.

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

- **ConcreteDecorator**

DECORATOR

Object Structural

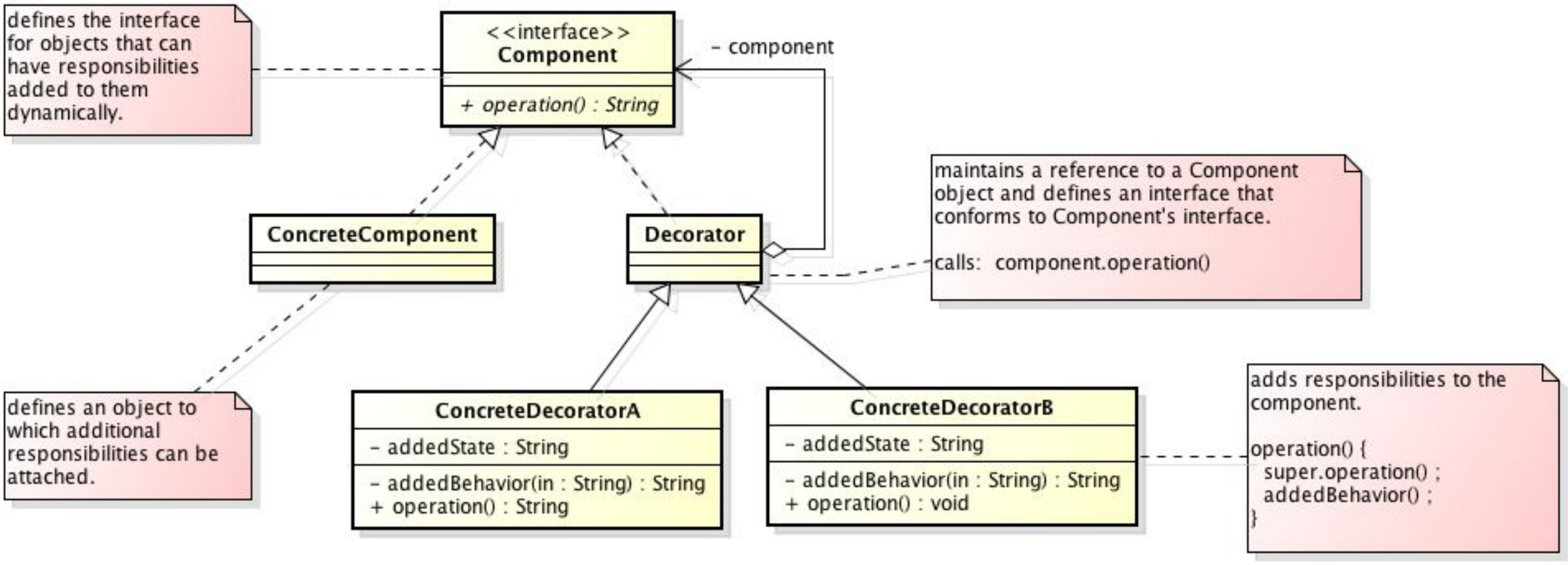


Purpose

Allows for the dynamic wrapping of objects in order to modify their existing responsibilities and behaviors.

Use When

- Object responsibilities and behaviors should be dynamically modifiable.
- Concrete implementations should be decoupled from responsibilities and behaviors.
- Subclassing to achieve modification is impractical or impossible.
- Specific functionality should not reside high in the object hierarchy.
- A lot of little objects surrounding a concrete implementation is acceptable.


defines the interface for objects that can have responsibilities added to them dynamically.
<<interface>>
Component
+ operation() : String
- component
ConcreteComponent
Decorator
maintains a reference to a Component object and defines an interface that conforms to Component's interface.
calls: component.operation()
defines an object to which additional responsibilities can be attached.
ConcreteDecoratorA
- addedState : String
- addedBehavior(in : String) : String
+ operation() : String
ConcreteDecoratorB
- addedState : String
- addedBehavior(in : String) : String
+ operation() : void
adds responsibilities to the component.
operation() {
super.operation() ;
addedBehavior() ;
}

```java
public class Decorator implements Component {

    private Component component;

    public Decorator( Component c )
    {
        component = c ;
    }

    public String operation()
    {
        return component.operation() ;
    }

}
```

```java
public class ConcreteComponent implements Component {

    public String operation() {
        return "Hello World!";
    }

}
```

```java
public class ConcreteDecoratorA extends Decorator {

    private String addedState;

    public ConcreteDecoratorA( Component c)
    {
        super( c ) ;
    }

    public String operation()
    {
        addedState = super.operation() ;
        return addedBehavior( addedState ) ;
    }

    private String addedBehavior(String in) {
        return "<em>" + addedState + "</em>" ;
    }

}
```

```java
public class ConcreteDecoratorB extends Decorator {

    private String addedState;

    public ConcreteDecoratorB( Component c)
    {
        super( c ) ;
    }

    public String operation()
    {
        addedState = super.operation() ;
        return addedBehavior( addedState ) ;
    }

    private String addedBehavior(String in) {
        return "<h1>" + addedState + "</h1>" ;
    }

}
```

```java
public class Tester {

    public static void runTest()
    {
        Component obj = new ConcreteDecoratorB( new ConcreteDecoratorA( new ConcreteComponent() ) ) ;
        String result = obj.operation() ;
        System.out.println( result );
    }
}
```