

TC3003: Diseño y Arquitectura de Software

Dr. Juan Manuel González Calleros

Email: jmgonzale@itesm.mx

Twitter: [@Juan__Gonzalez](https://twitter.com/Juan__Gonzalez)

Facebook: **Juan Glez Calleros**

Reuniones pedir cita



¿Dónde vemos más observadores?

- JButton sujeto
- Lleno de observadores
 - Listeners

```

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;

public class SwingObserverExample {
    JFrame frame;

    public static void main(String[] args) {
        SwingObserverExample example = new SwingObserverExample();
        example.go();
    }

    public void go() {
        frame = new JFrame();

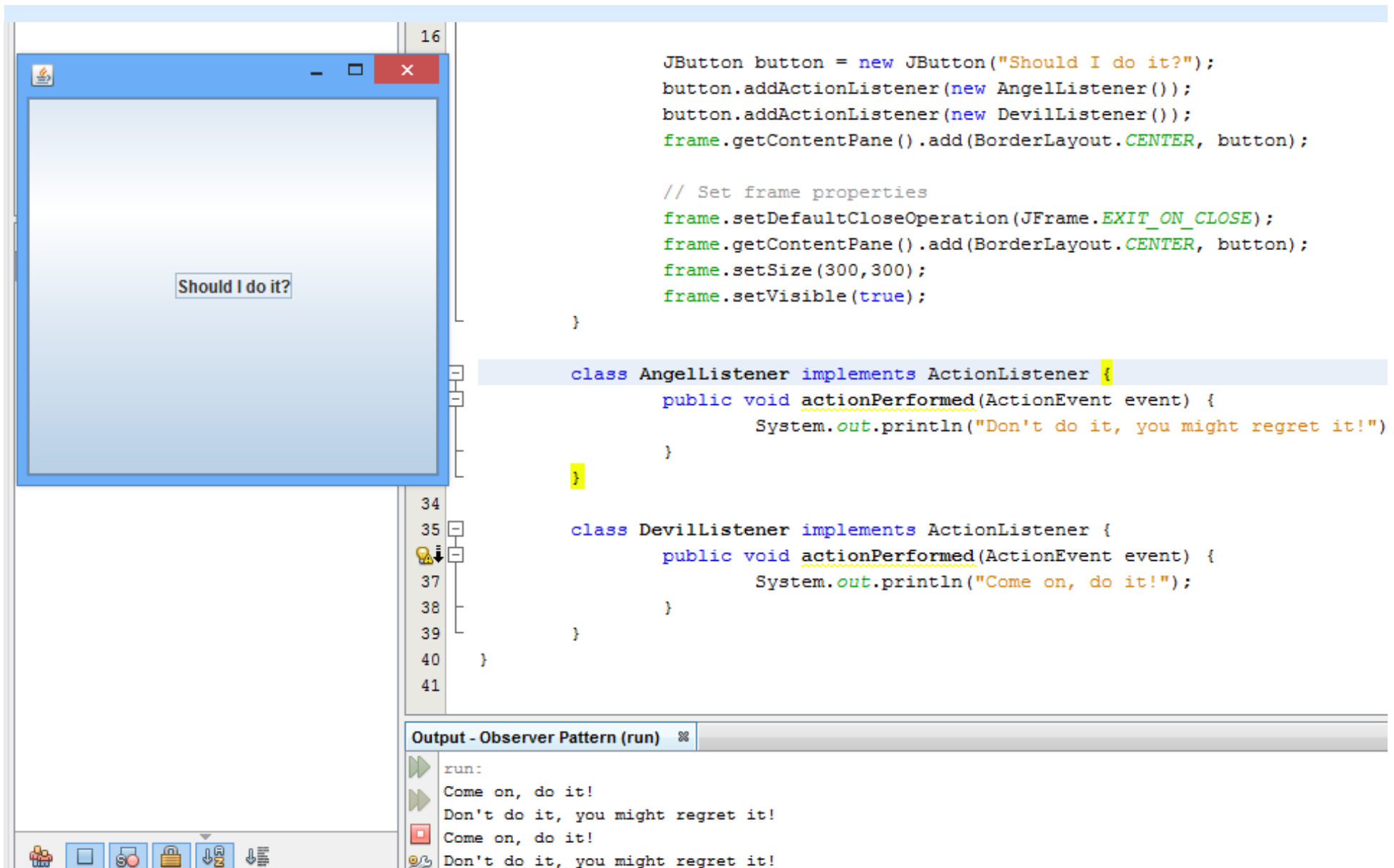
        JButton button = new JButton("Should I do it?");
        button.addActionListener(new AngelListener());
        button.addActionListener(new DevilListener());
        frame.getContentPane().add(BorderLayout.CENTER, button);

        // Set frame properties
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.getContentPane().add(BorderLayout.CENTER, button);
        frame.setSize(300,300);
        frame.setVisible(true);
    }

    class AngelListener implements ActionListener {
        public void actionPerformed(ActionEvent event) {
            System.out.println("Don't do it, you might regret it!");
        }
    }

    class DevilListener implements ActionListener {
        public void actionPerformed(ActionEvent event) {
            System.out.println("Come on, do it!");
        }
    }
}

```



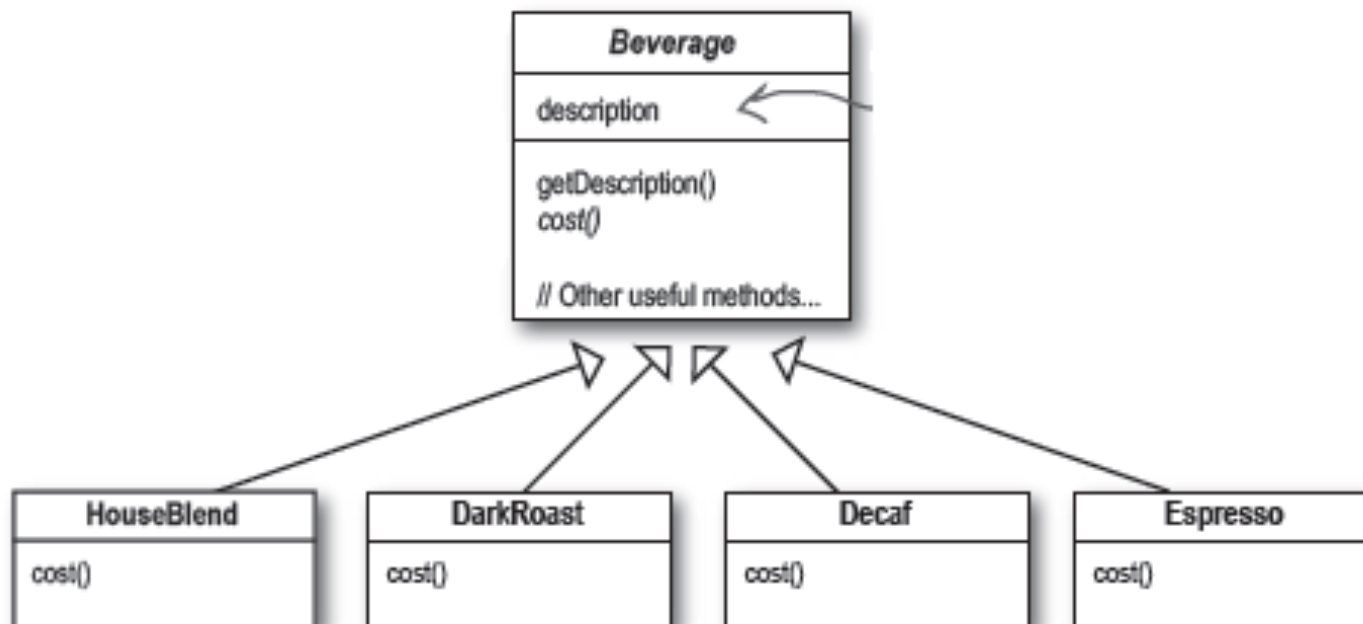
```
16  
  
JButton button = new JButton("Should I do it?");  
button.addActionListener(new AngellListener());  
button.addActionListener(new DevillListener());  
frame.getContentPane().add(BorderLayout.CENTER, button);  
  
// Set frame properties  
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
frame.getContentPane().add(BorderLayout.CENTER, button);  
frame.setSize(300,300);  
frame.setVisible(true);  
}  
  
class AngellListener implements ActionListener {  
    public void actionPerformed(ActionEvent event) {  
        System.out.println("Don't do it, you might regret it!")  
    }  
}  
  
class DevillListener implements ActionListener {  
    public void actionPerformed(ActionEvent event) {  
        System.out.println("Come on, do it!");  
    }  
}  
}  
  
34  
35  
36  
37  
38  
39  
40  
41  
  
Output - Observer Pattern (run) %  
run:  
Come on, do it!  
Don't do it, you might regret it!  
Come on, do it!  
Don't do it, you might regret it!
```

Patrones Estructurales

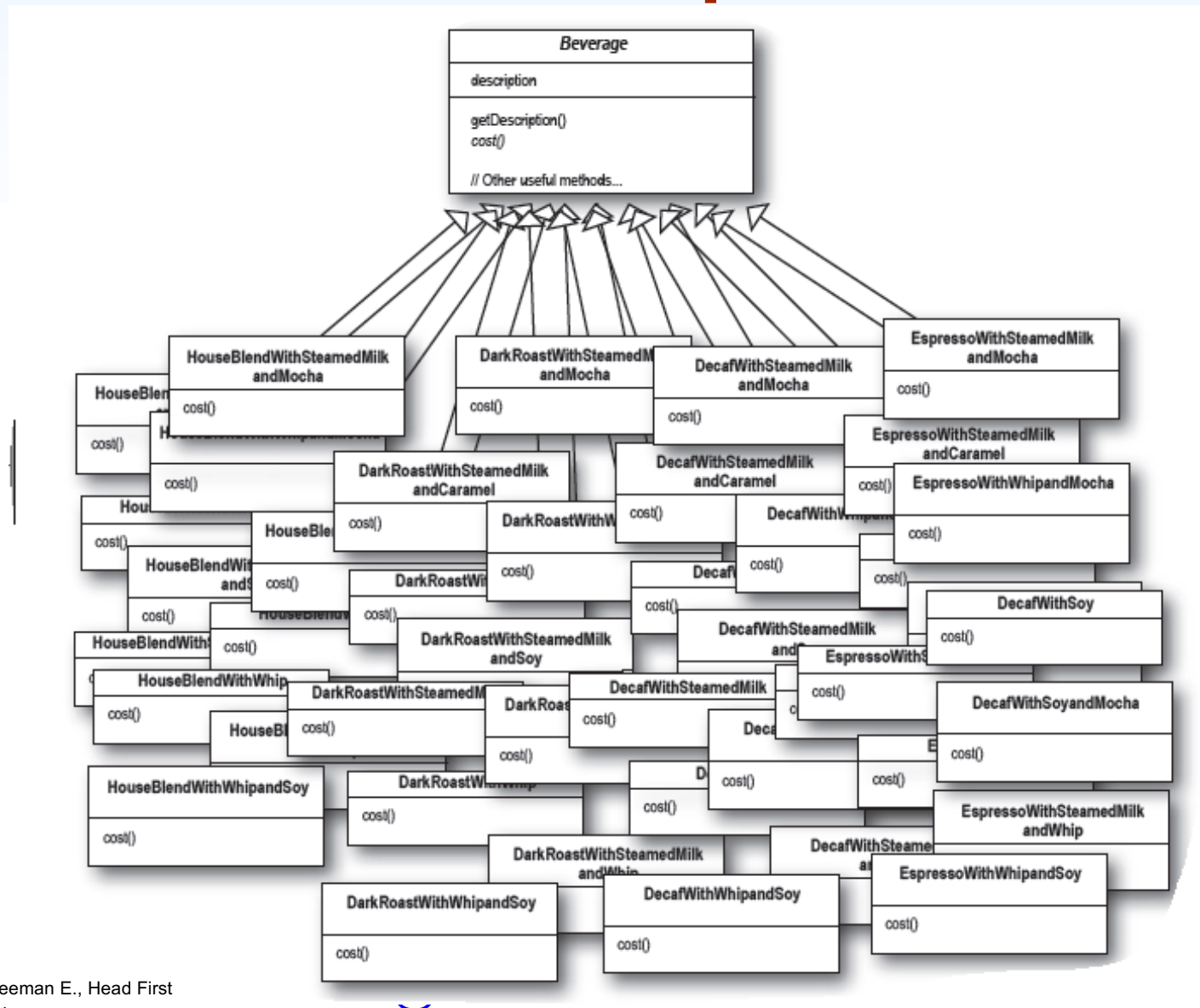
PATRÓN DECORADOR

Decorator Pattern

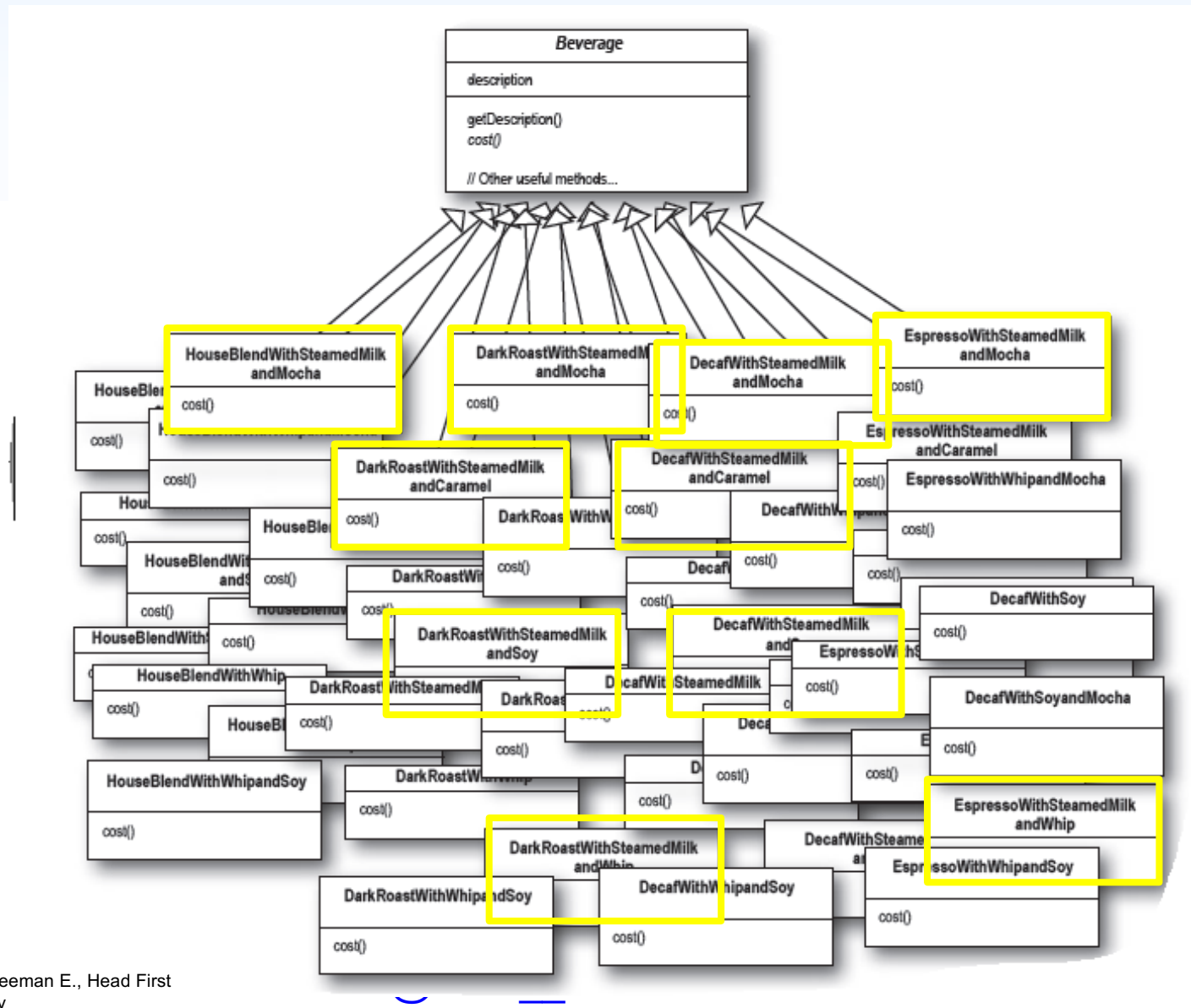
- iCoffee, below the current model



iCoffee has grown and its sales system has been surpassed



What if the Milk cost changes?

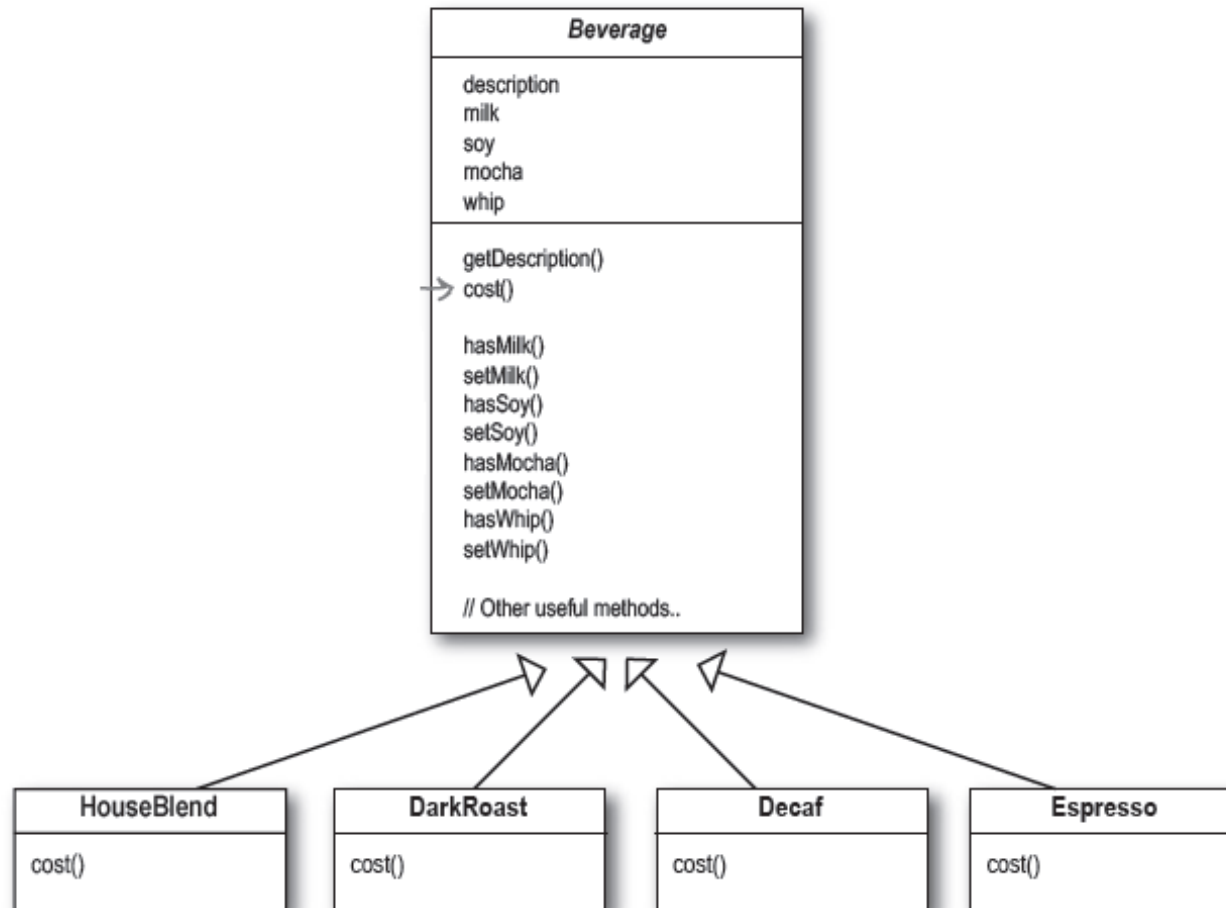


Class Explosion

- Rethink the Beverage class
 - Let's put in its attributes the possible ingredients (bool)
 - The cost is calculated based to the variables.
 - The children will implement the cost but just adding extra prices

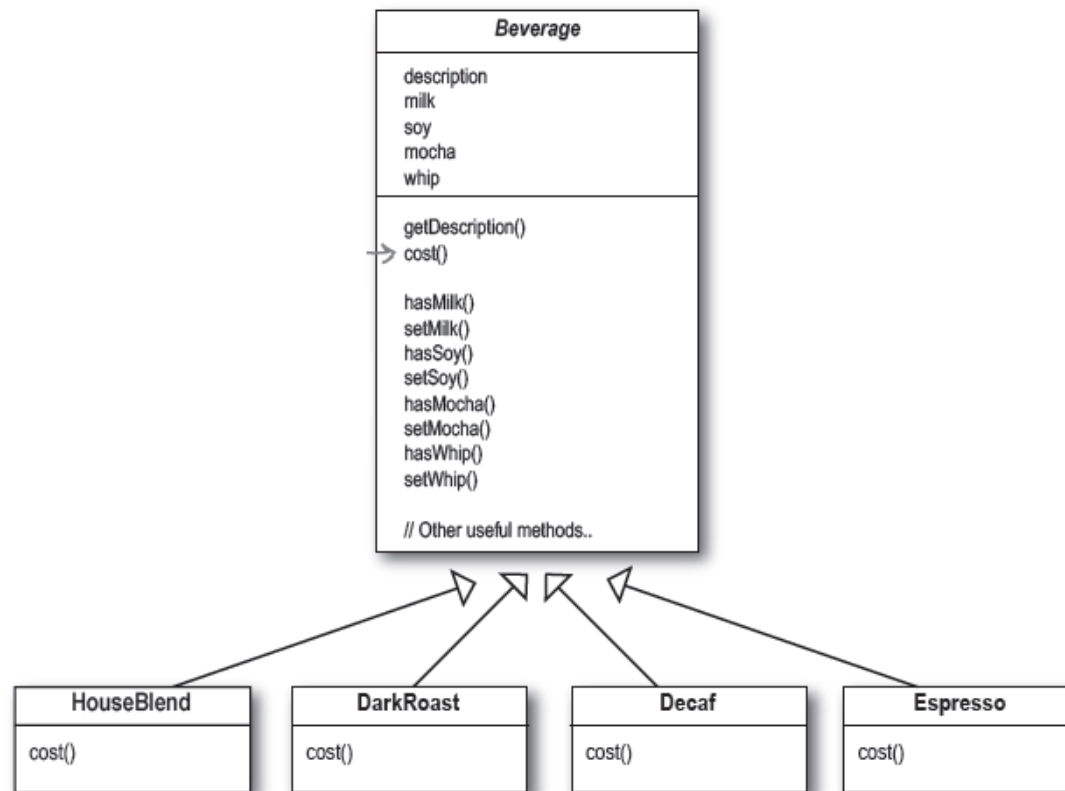
<i>Beverage</i>
description milk soy mocha whip
getDescription() cost() hasMilk() setMilk() hasSoy() setSoy() hasMocha() setMocha() hasWhip() setWhip() // Other useful methods..

Class Explosion



Class Explosion

- How would the code of the cost () method of the super class and that of the DarkRoast subclass which is 1.99 more expensive?



Class Explosion

- How would the code of the cost () method of the super class and that of the DarkRoast subclass which is 1.99 more expensive?

```
public class Beverage {  
    // declare instance variables for milkCost,  
    // soyCost, mochaCost, and whipCost, and  
    // getters and setters for milk, soy, mocha  
    // and whip.  
  
    public float cost() {  
        float condimentCost = 0.0;  
        if (hasMilk()) {  
            condimentCost += milkCost;  
        }  
        if (hasSoy()) {  
            condimentCost += soyCost;  
        }  
        if (hasMocha()) {  
            condimentCost += mochaCost;  
        }  
        if (hasWhip()) {  
            condimentCost += whipCost;  
        }  
        return condimentCost;  
    }  
}  
  
public class DarkRoast extends Beverage {  
    public DarkRoast() {  
        description = "Most Excellent Dark Roast";  
    }  
  
    public float cost() {  
        return 1.99 + super.cost();  
    }  
}
```

What can change in the future?

@Juan__Gonzalez

What can change in the future?

- The price of the ingredients may vary → we have to modify the code

What can change in the future?

- The price of the ingredients may vary → we have to modify the code
- New ingredients → add new methods, modification of the cost () method in the super and sub classes

What can change in the future?

- The price of the ingredients may vary → we have to modify the code
- New ingredients → add new methods, modification of the cost () method in the super and sub classes
- New drinks (tea) → with absurd ingredients

What can change in the future?

- The price of the ingredients may vary → we have to modify the code
- New ingredients → add new methods, modification of the cost () method in the super and sub classes
- New drinks (tea) → with absurd ingredients
- Special orders → double

Design Principle

- Classes must be open for extensions but closed for modifications.
 - Modify the classes and make your own version.
 - However, you can't manipulate the code we already did,
- A class must be prepared to have **more behaviors**.
 - **Adding more behaviors to the current behavior**
→ extended behavior

Decorator Pattern

- The iCoffee solution so far is insufficient
 - Class Explosion, rigid design, functionality to the base class that we know is inappropriate for subclasses,
- Let's decorate the Beverage class with its condiments at runtime not design!

Decorator Pattern

- Remember the Wrappers (wrappers).
Classes that involve primitive types to be able to use them in:
 - Data structures
 - Convert to String and vice versa

- Byte para byte.
- Short para short.
- Integer para int.
- Long para long.
- Boolean para boolean
- Float para float.
- Double para double y
- Character para char.

Decorator Pattern

- We are going to decorate the Beverage class with its condiments at runtime! Not design For example to:
 - For the DarkRoast object
 - Decorate it with the Mocha object
 - Decorate it with the Whip object
 - Call the cost () method and use delegation to add seasonings to the cost

Decorator Pattern

- For the DarkRoast object



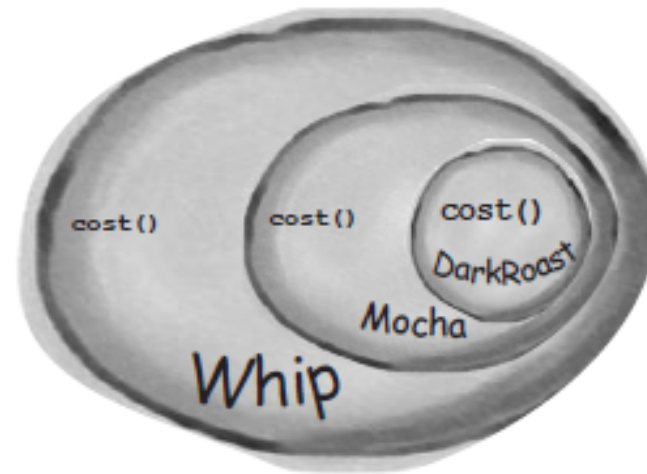
Decorator Pattern

- For the DarkRoast object
- **Decorate it with the Mocha object**



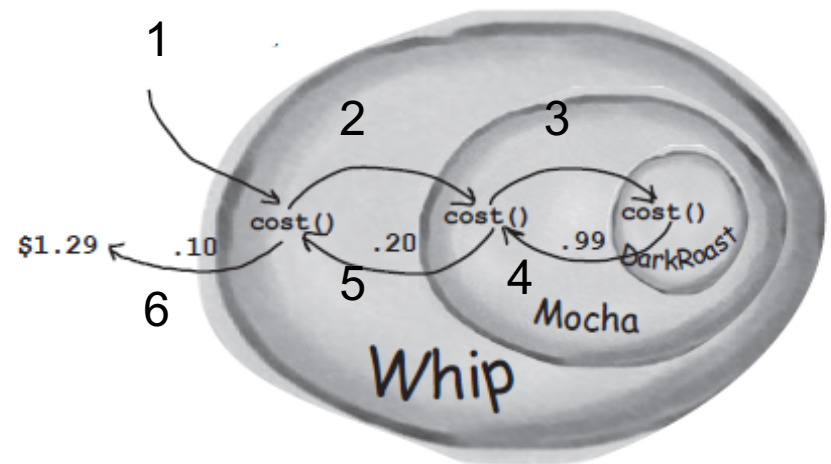
Decorator Pattern

- For the DarkRoast object
- Decorate it with the Mocha object
- **Decorate it with the Whip object**



Decorator Pattern

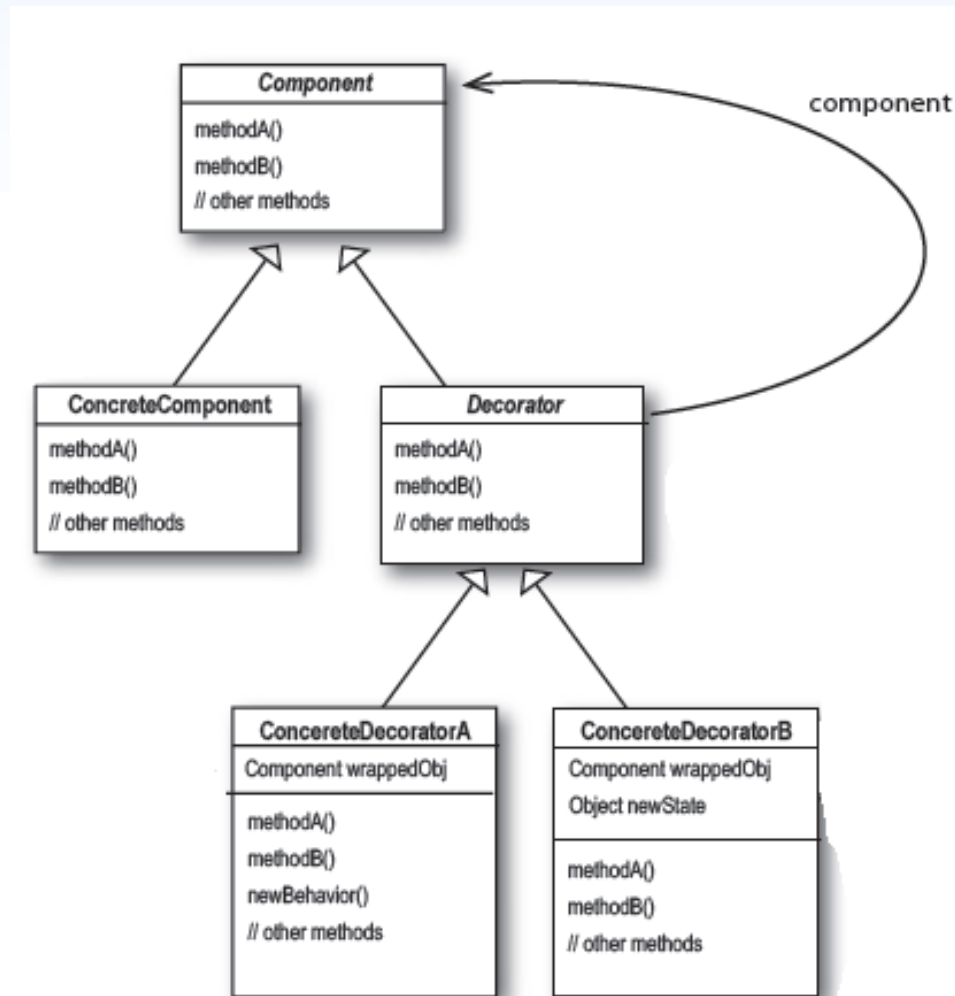
- For the DarkRoast object
- Decorate it with the Mocha object
- Decorate it with the Whip object
- **Call the cost () method and use delegation to add seasonings to the cost**



Decorator Pattern

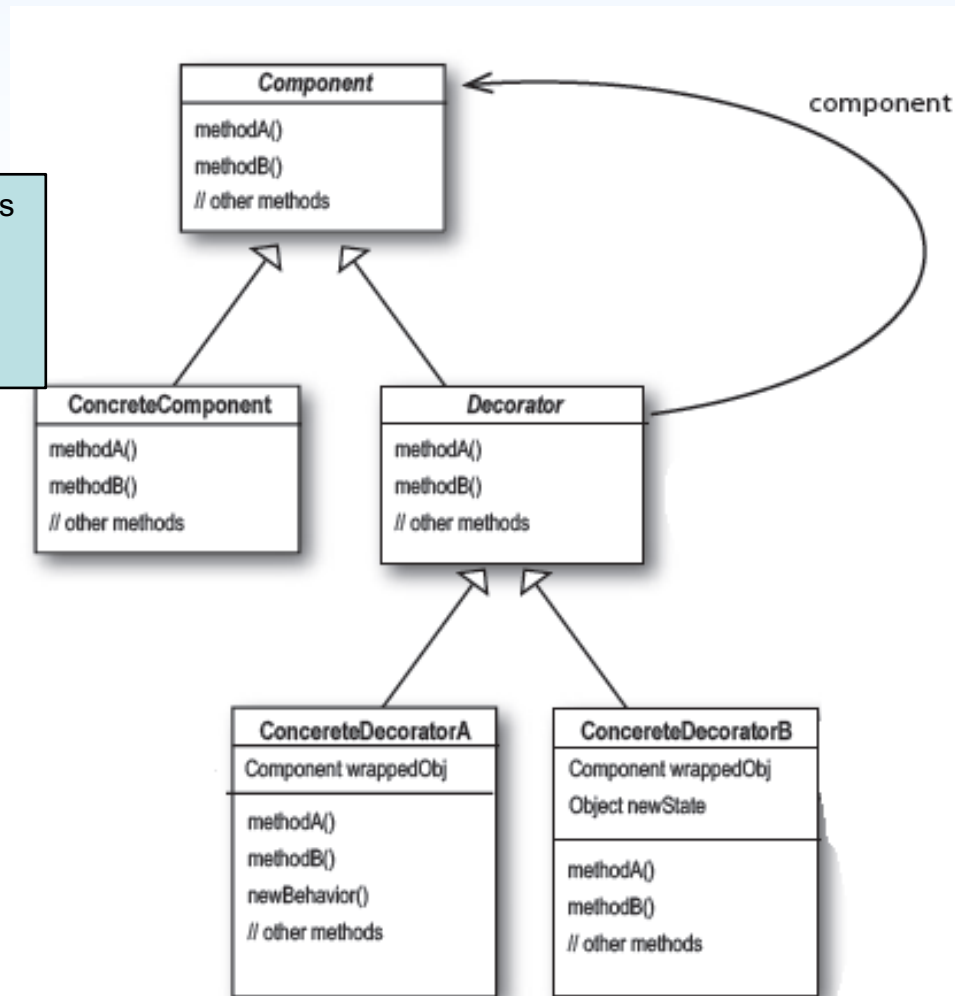
- The decorator pattern adds additional responsibilities to an object dynamically.
- Decorators offer a flexible alternative for subclasses and the extension of their functionality

Decorator Pattern

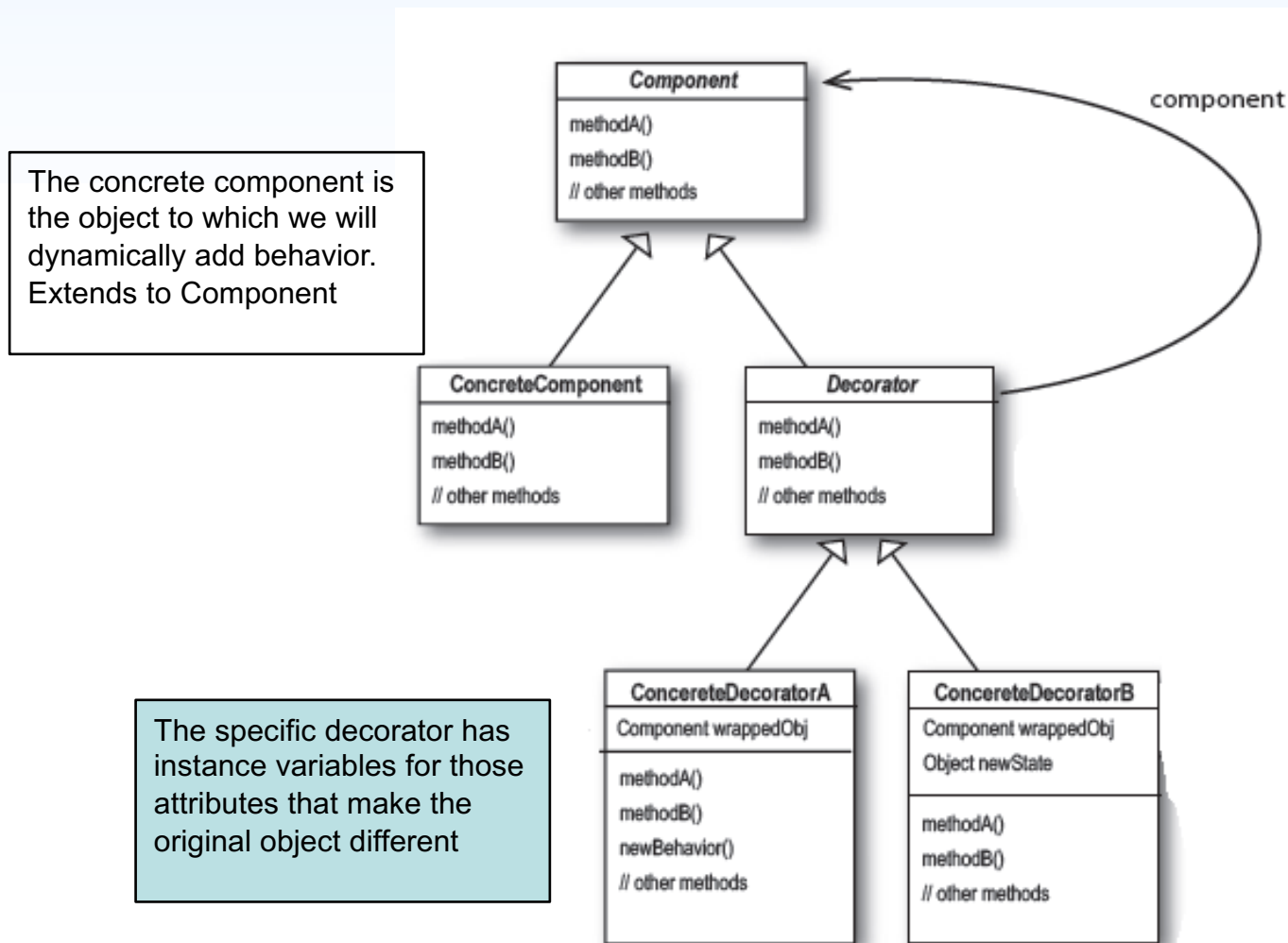


Decorator Pattern

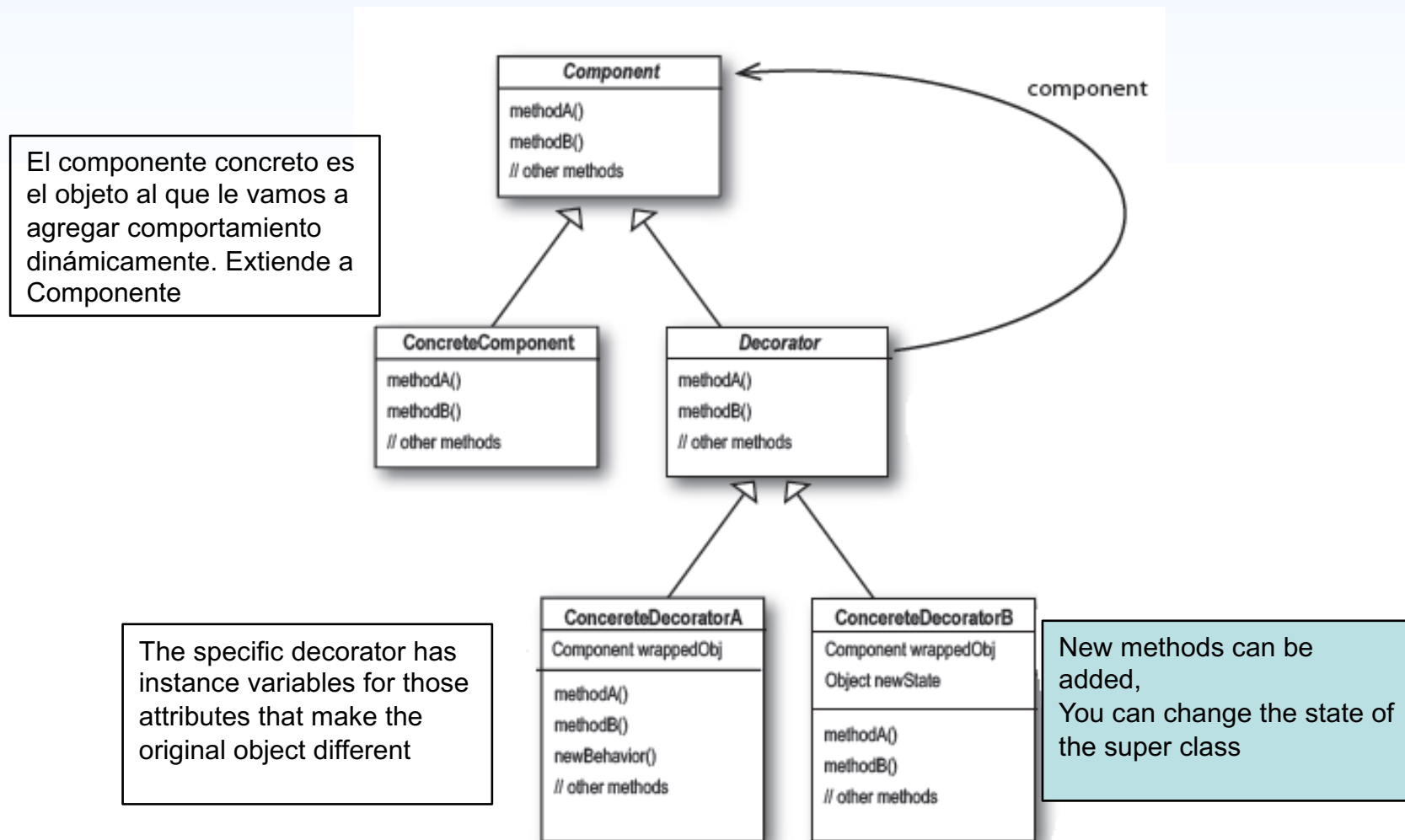
The concrete component is the object to which we will dynamically add behavior. Extends to Component



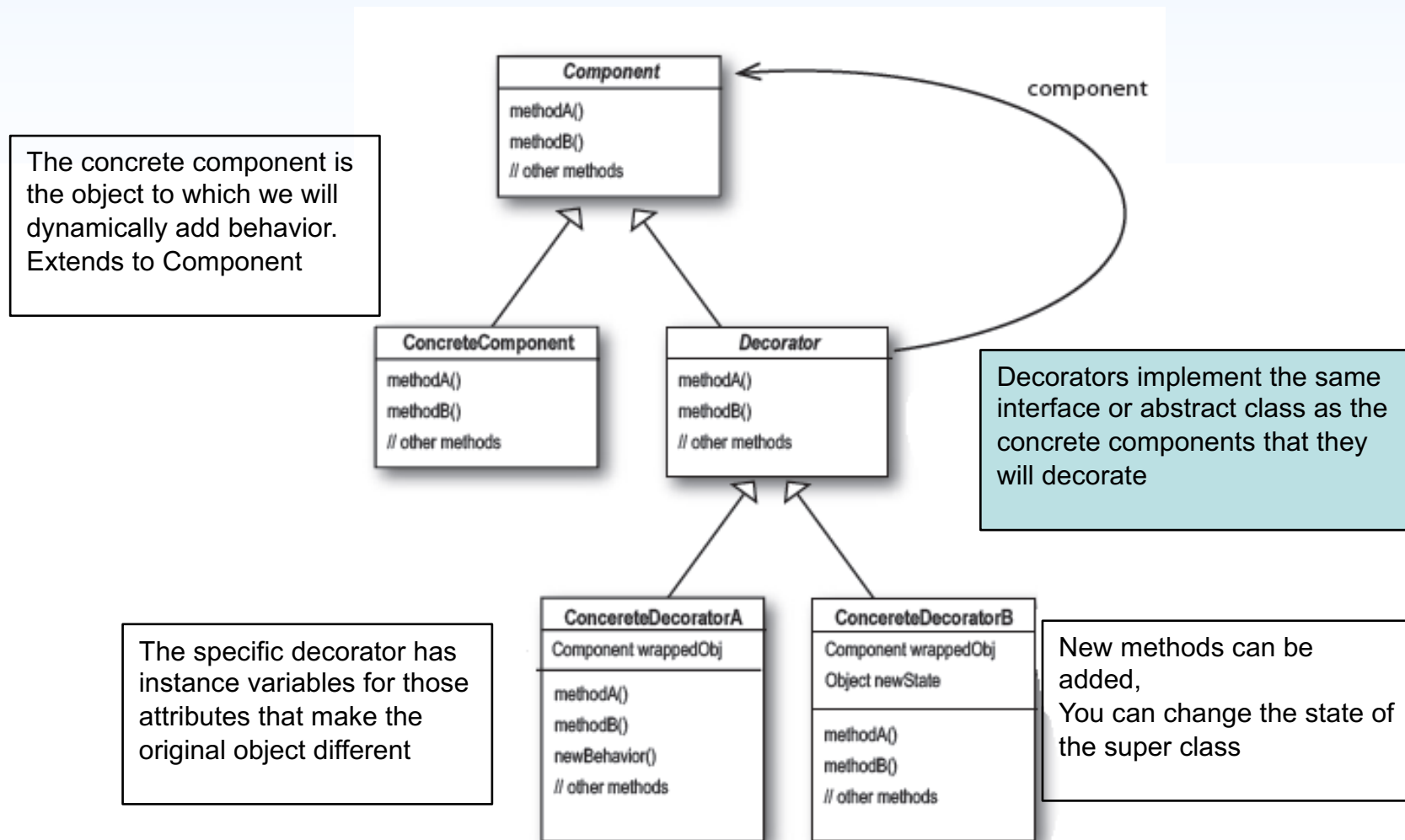
Decorator Pattern



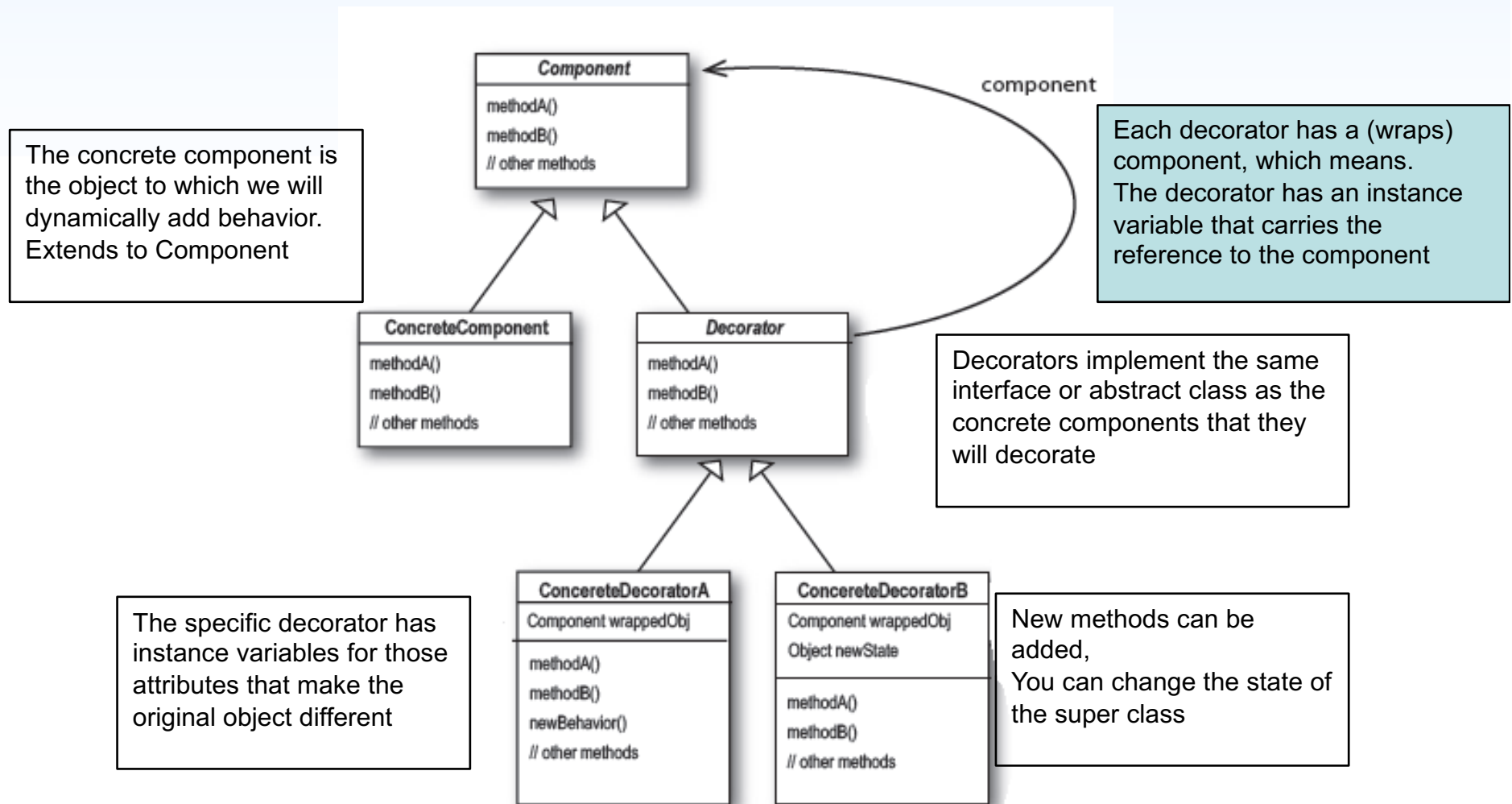
Decorator Pattern



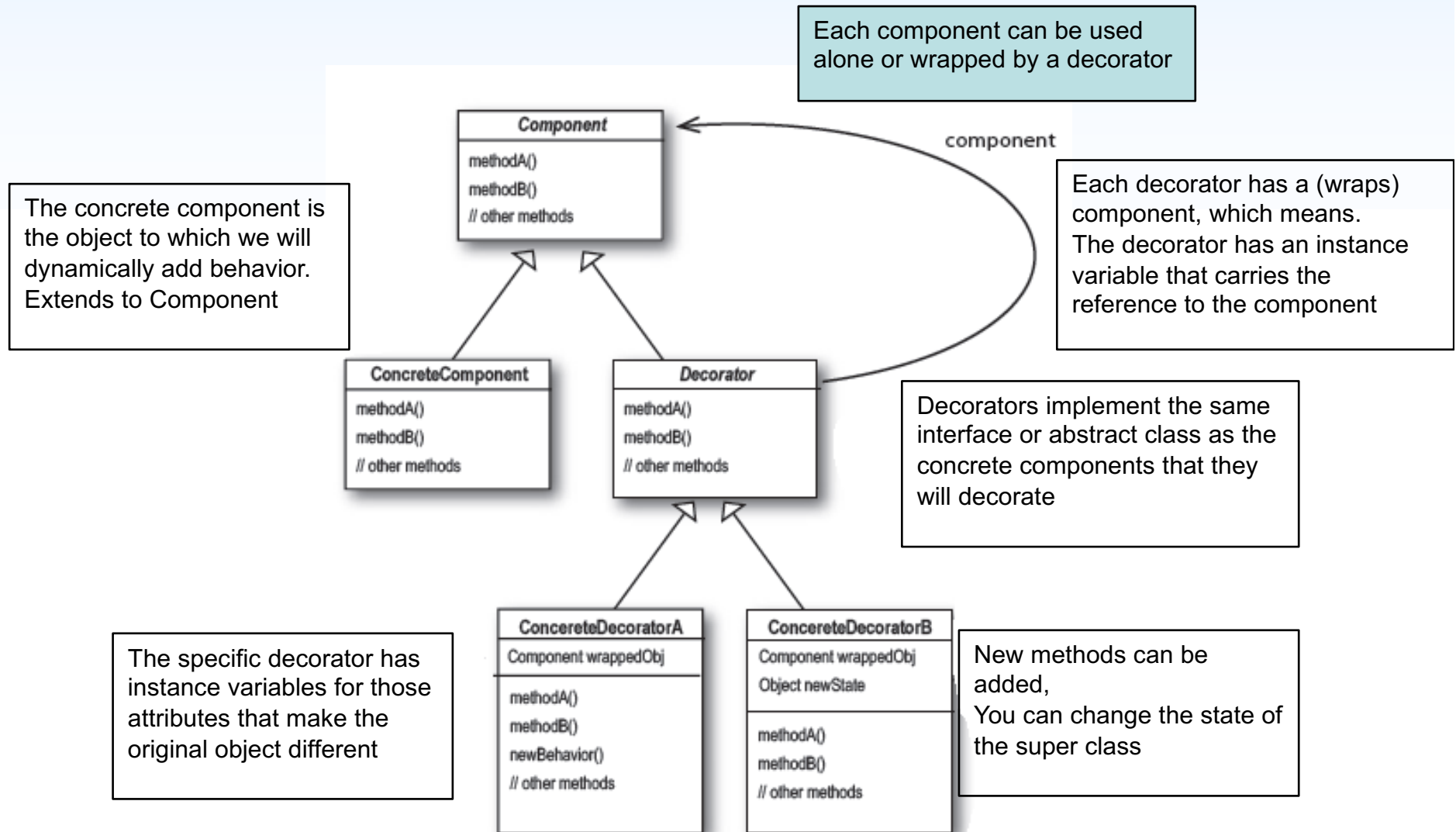
Decorator Pattern



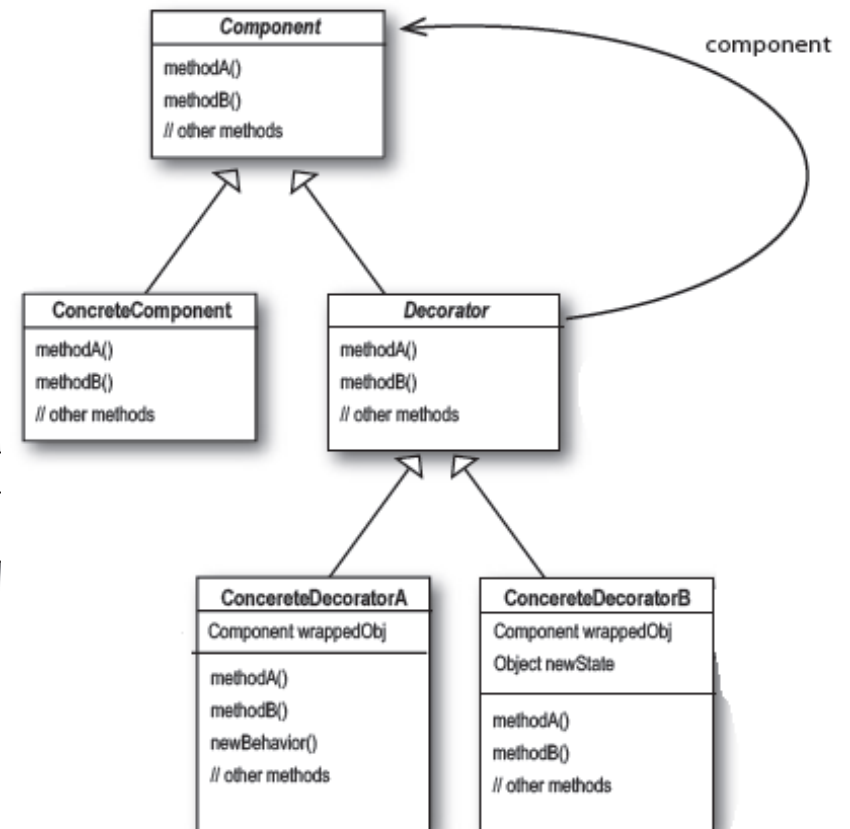
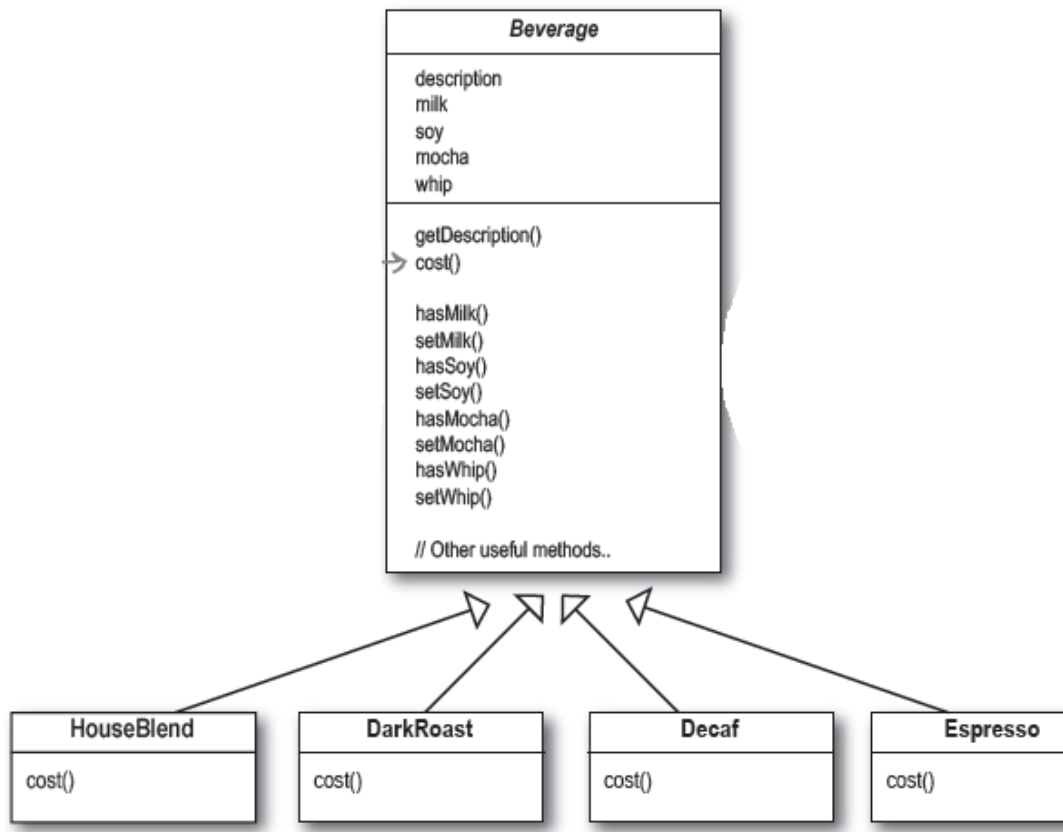
Decorator Pattern



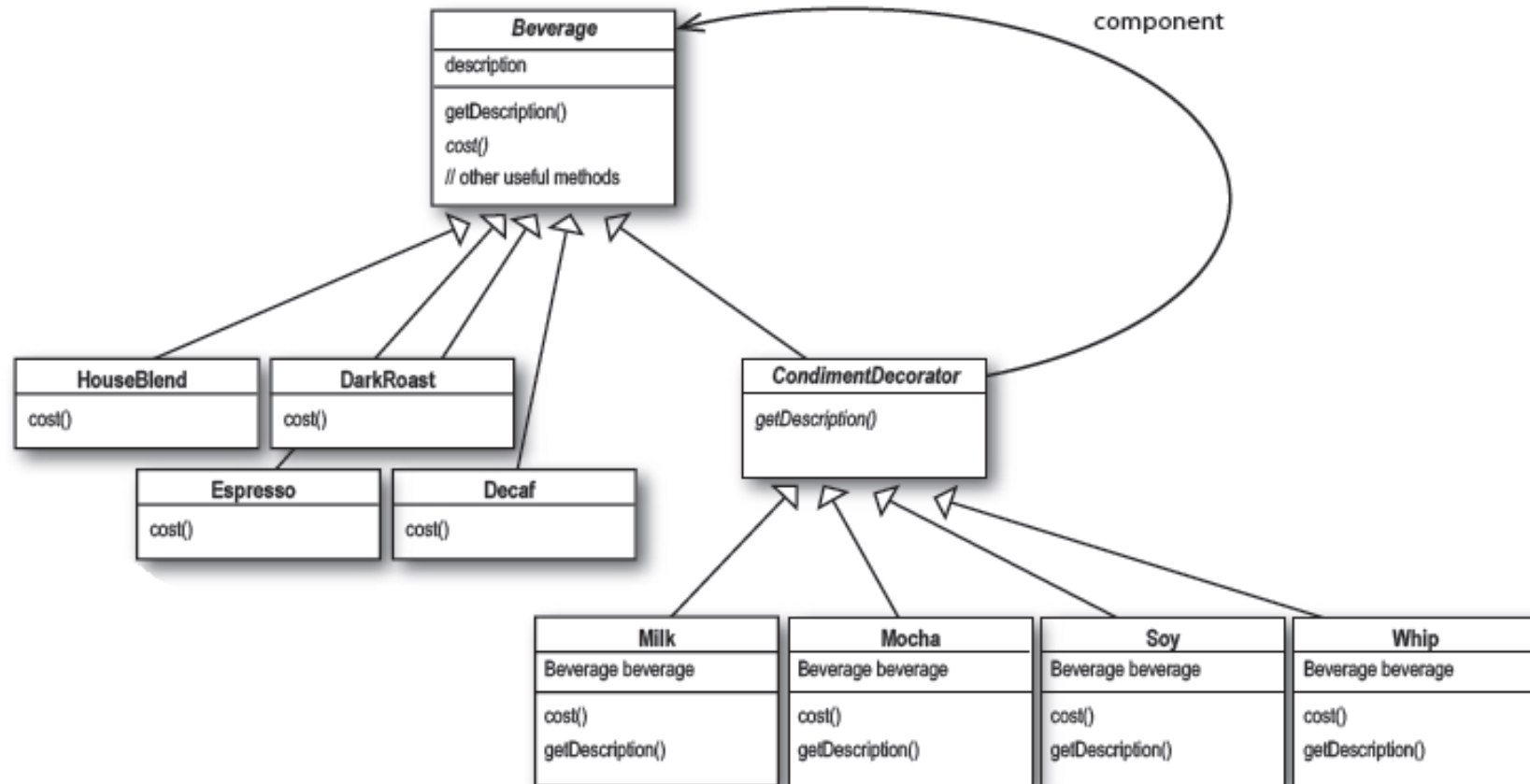
Decorator Pattern



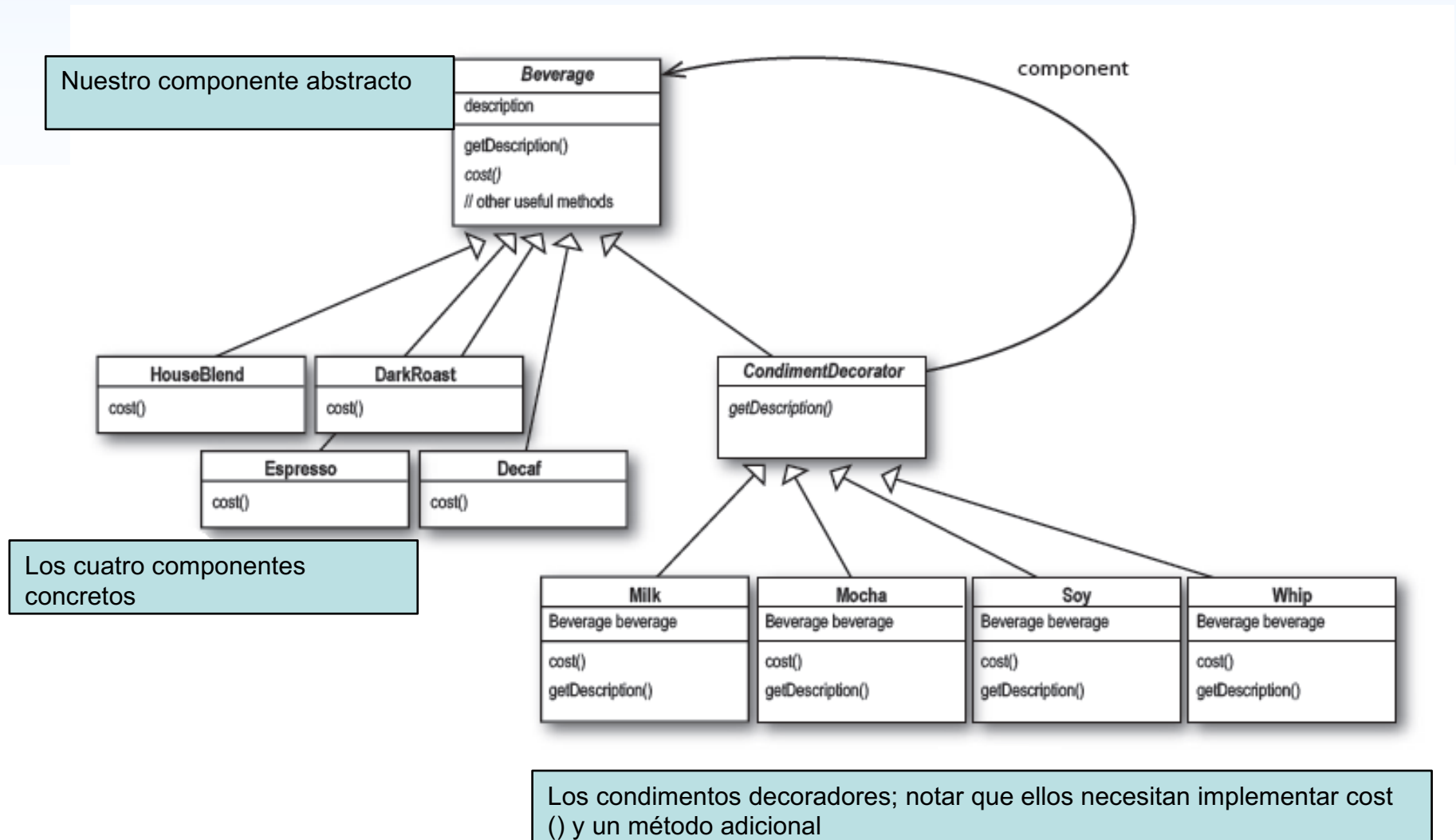
Based on this how would you modify the diagram?



iCoffee Decorator Pattern

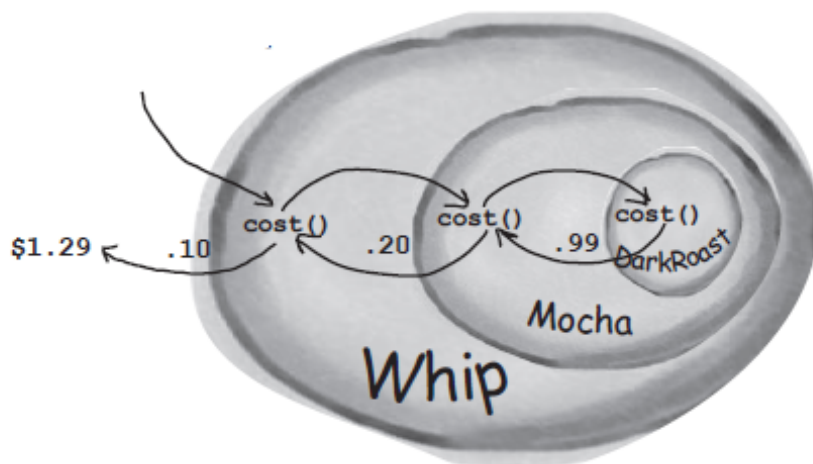


iCoffee Decorator Pattern



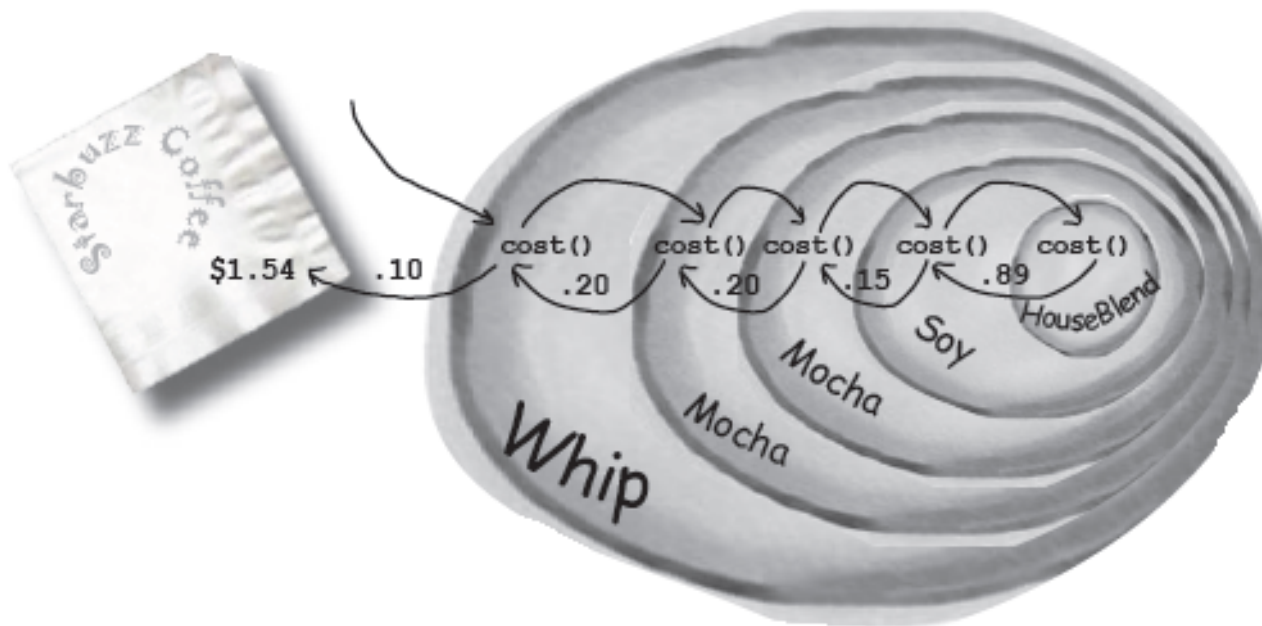
How are the objects wrapped for the next order and how much does it cost?

- Double Mocha Soy House Blend with Whip



Starbuzz Coffee	
<u>Coffees</u>	
House Blend	.89
Dark Roast	.99
Decaf	1.05
Espresso	1.99
<u>Condiments</u>	
Steamed Milk	.10
Mocha	.20
Soy	.15
Whip	.10

How to implement cost and getDescription() methods?





LET'S CODE IN JAVA