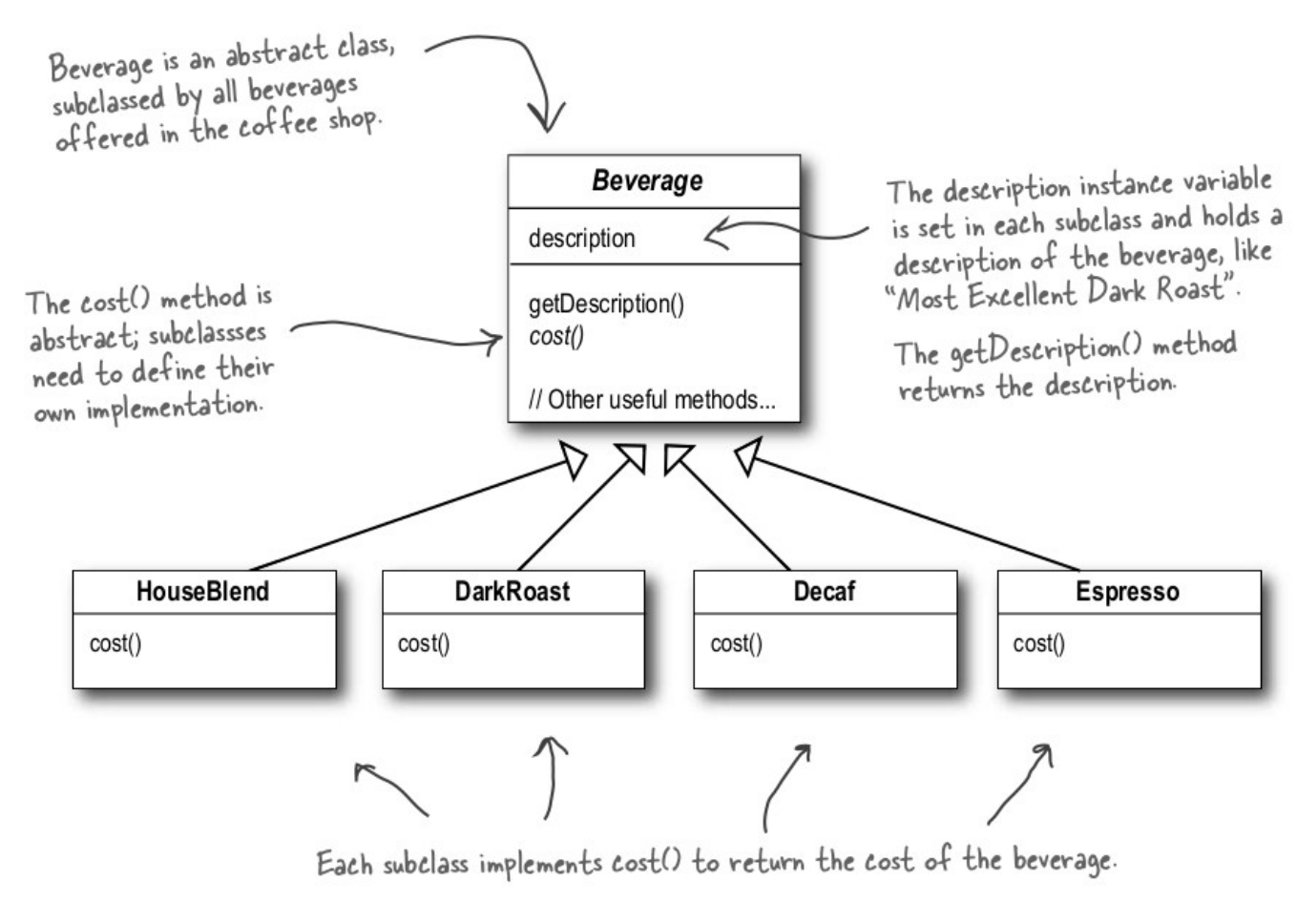
Patrón de Diseño "Decorator"

Cáfe Starbuzz

- Ha tenido un gran crecimiento dentro de las tiendas de café, ya que ha aumentado el número de locales Starbuzz.

- Esto ha provocado que necesiten hacer coincidir su sistema de pedidos con las bebidas que ofrecen.

El primer diseño fue:



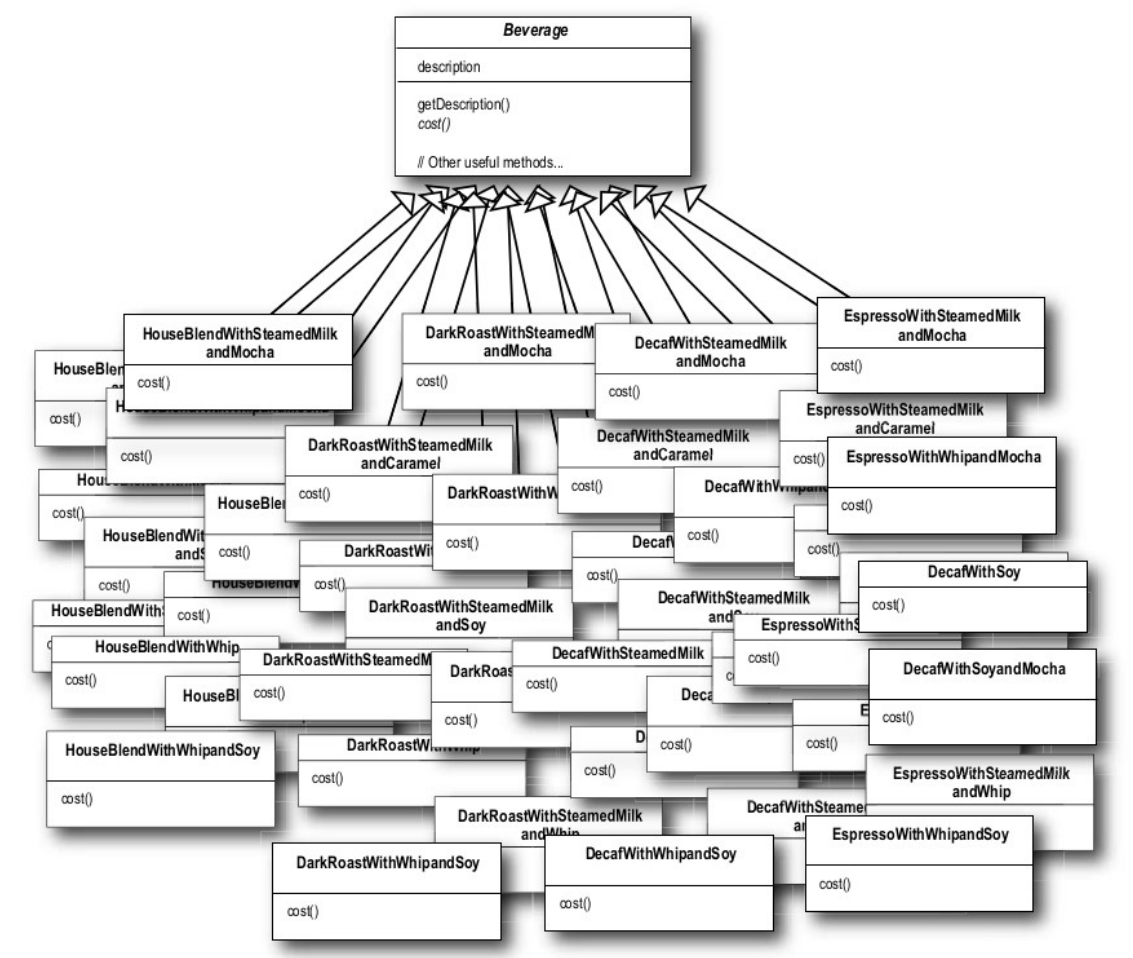
- Además de su bebida, los clientes pueden pedir varios ingredientes, como vapor de leche o mocha, pero siempre debe incluir al final batido de leche.

- Cobra un montó más por cada uno de los ingredientes extra que pidan los clientes, por lo cual debe tener cada uno de estos casos en su sistema de pedidos.

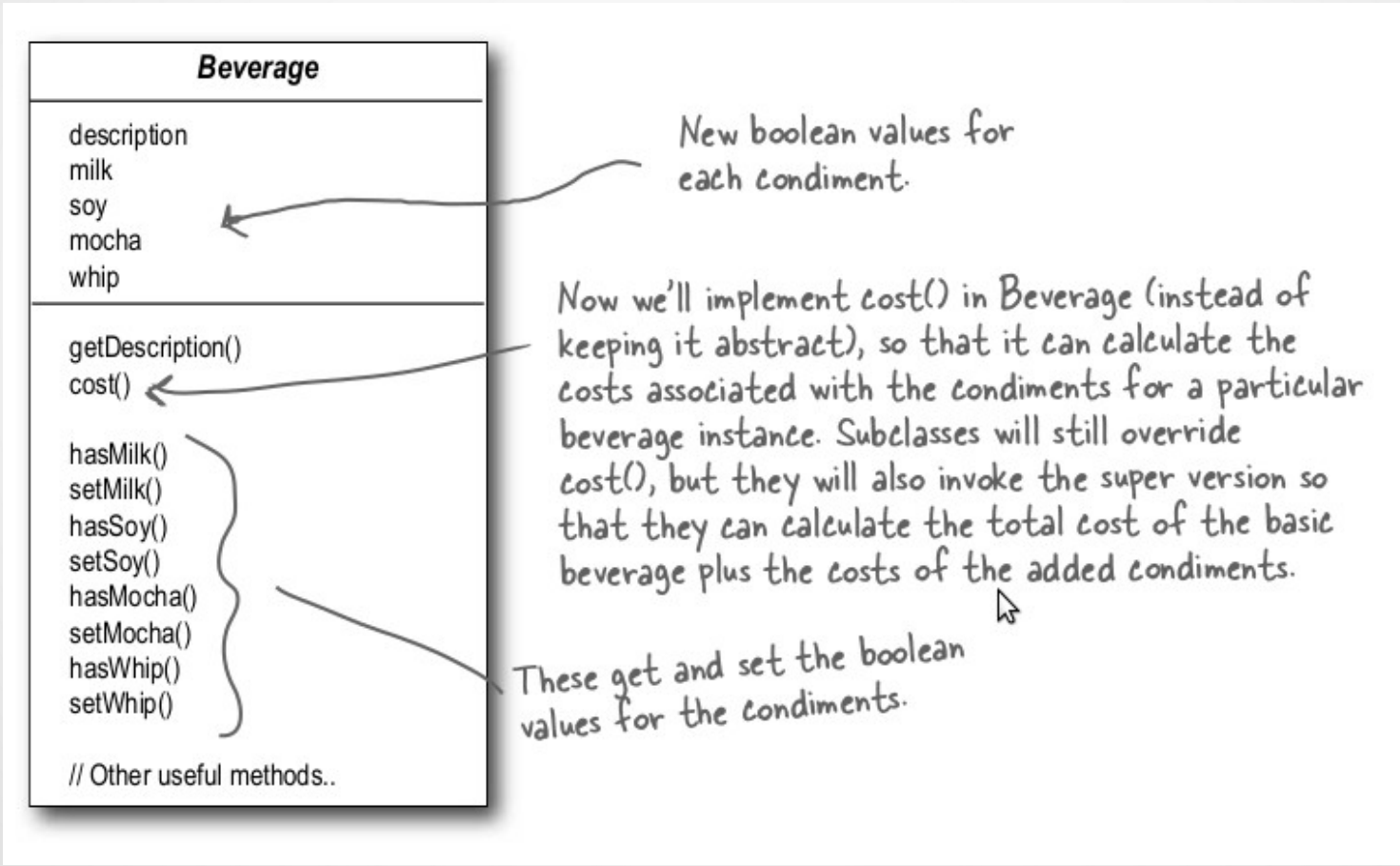
Entonces, ¿Qué tendríamos con el diseño anterior?

¡Advertencia! La siguiente diapositiva puede ser perturbadora.

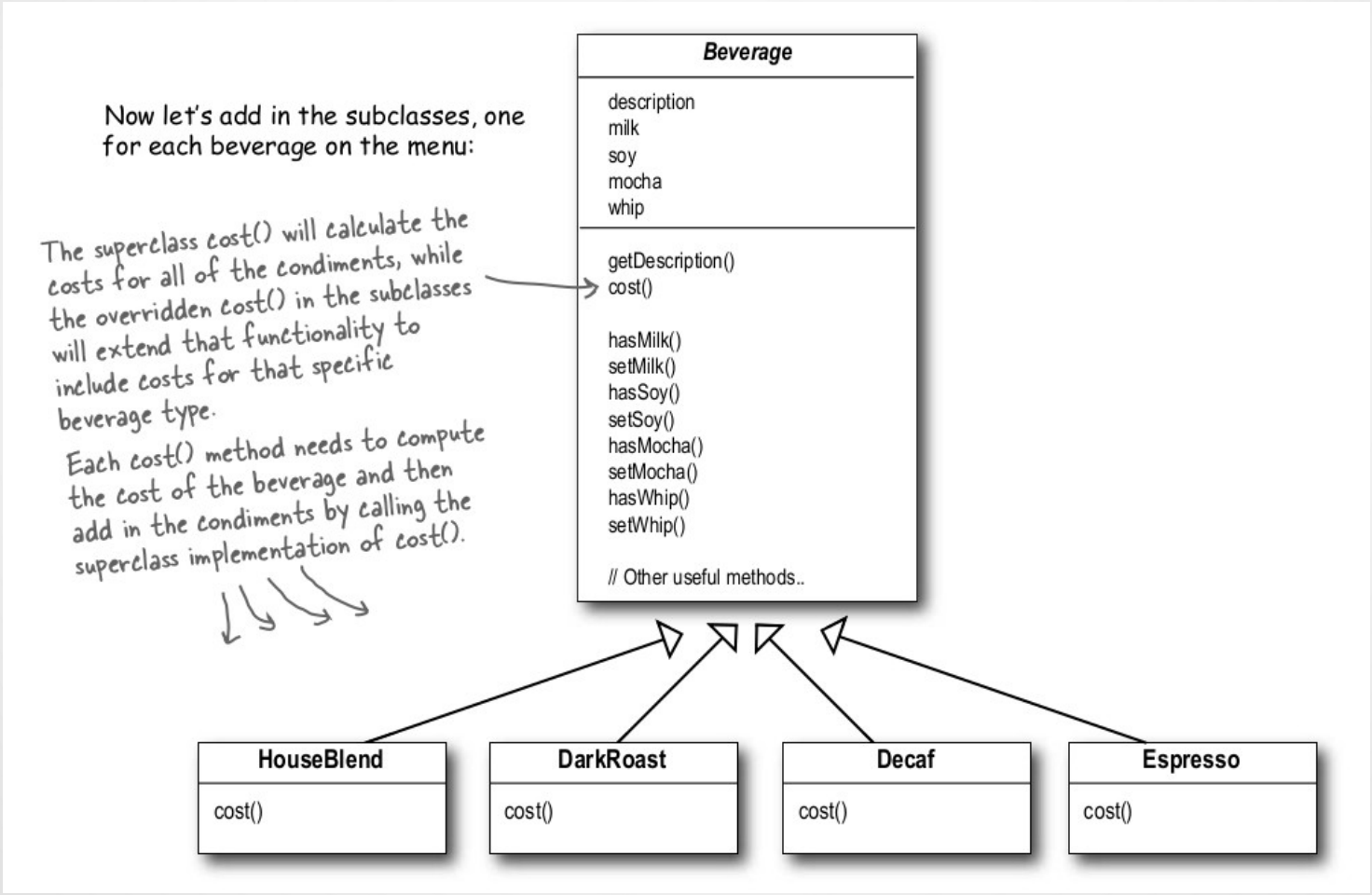
Obtenemos algo como esto:



¿Qué pasa si lo arreglamos así?



Ahora agregamos las subclases para cada tipo de bebida:



```
public class Beverage {

  // declare instance variables for milkCost,
  //  soyCost, mochaCost, and whipCost, and
  // getters and setters for milk, soy, mocha
  // and whip.

public double cost() {

      double 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 double cost() {

      return 1.99 + super.cost();


   }
}
```

- En la superclase, se calculara el costo de cada ingrediente con el método *cost()*.

- En método *cost()* sobrescrito en las subclases, necesita calcular el precio de la bebida y agregar el costo de los ingredientes llamando al método *cost()* de la superclase

¿Qué factores podrían causar un cambio o impacto en este diseño?

- Si cambia el precio de los ingredientes, sería forzoso alterar el código.

- Si es necesario agregar diferentes complementos, sería necesario agregar nuevos métodos y alterar el método costo en la superclase.

- Si agregamos nuevos tipos de bebidas, podría ser que algunos ingredientes o métodos no sean adecuados.

Principio de Diseño:

- Las clases debe estar abiertas a la extensión, pero cerradas a la modificación.

Nuestro objetivo es:

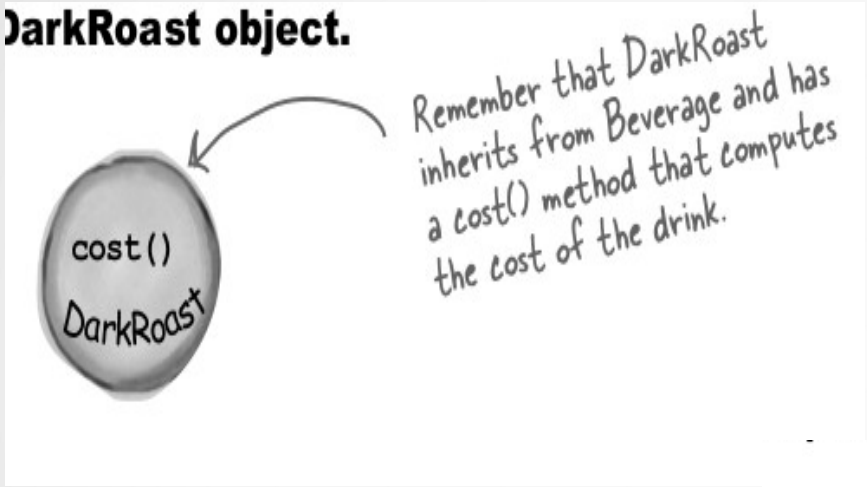
- Permitir que las clases sean fácilmente extendidas incorporando el nuevo comportamiento sin necesidad de modificar el código existente.

Para obtener lo anterior:

- Si tenemos una Bebida, la “decoramos” con los ingredientes en tiempo de ejecución.

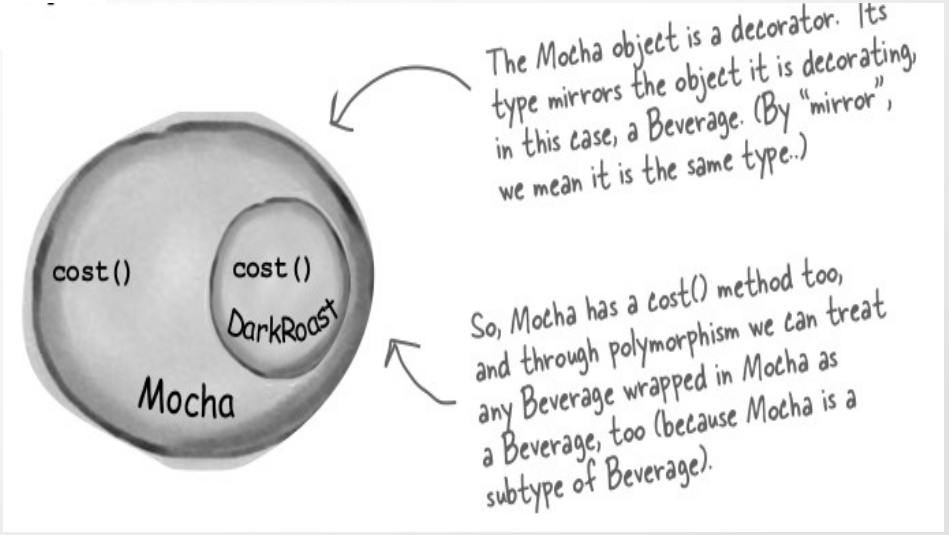
Por ejemplo:

El cliente pide un café oscuro con mocha y batido, entonces:

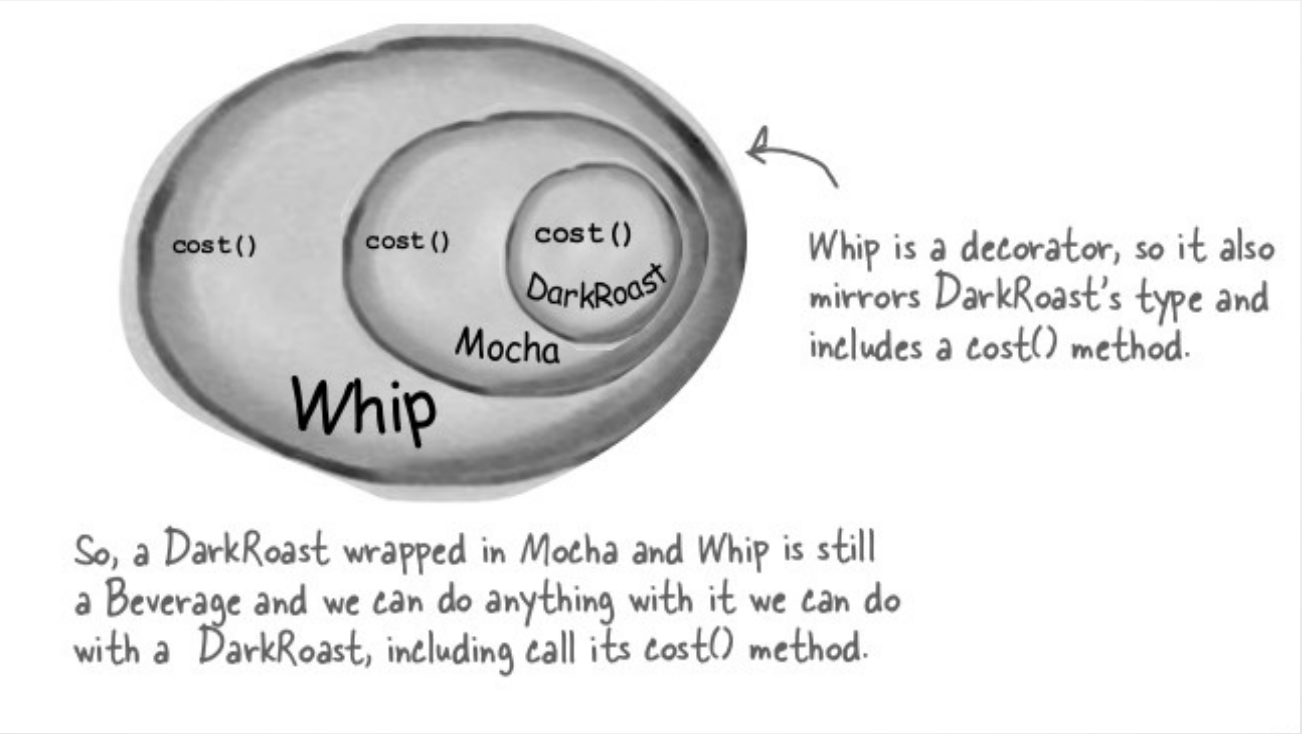


1) Comenzamos con nuestro objeto Café Oscuro

2) El cliente quiere Mocha, entonces creamos un objeto Mocha y envuélvelo alrededor al objeto Café Oscuro.



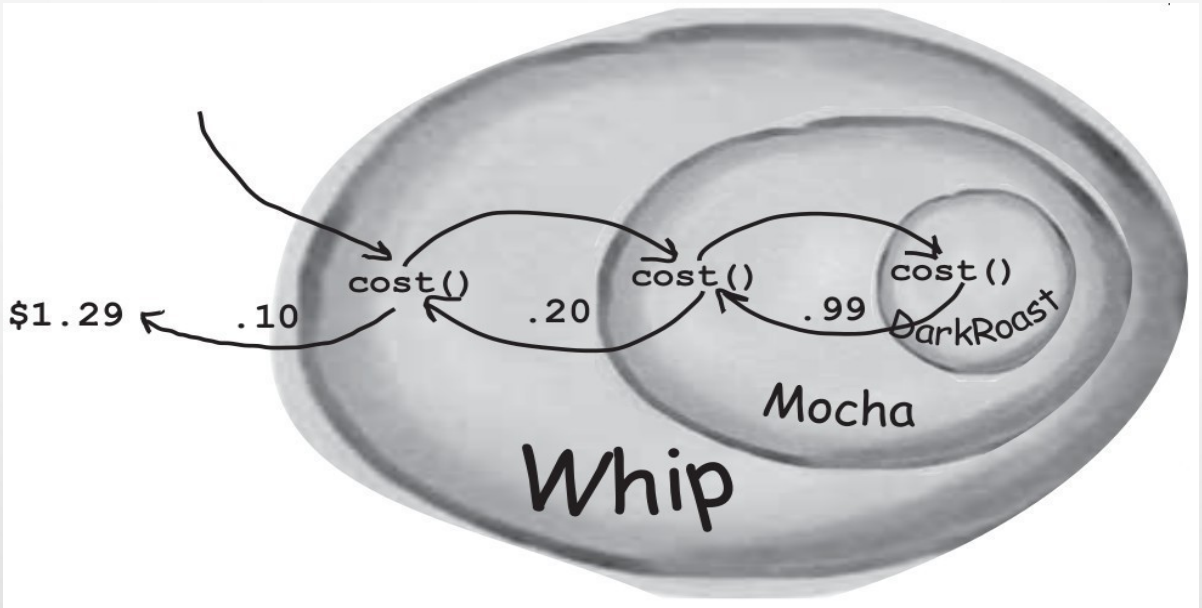
3) El cliente también quiere batido, así que creamos un decorador Batido y envolvemos Mocha con él.



Algo así

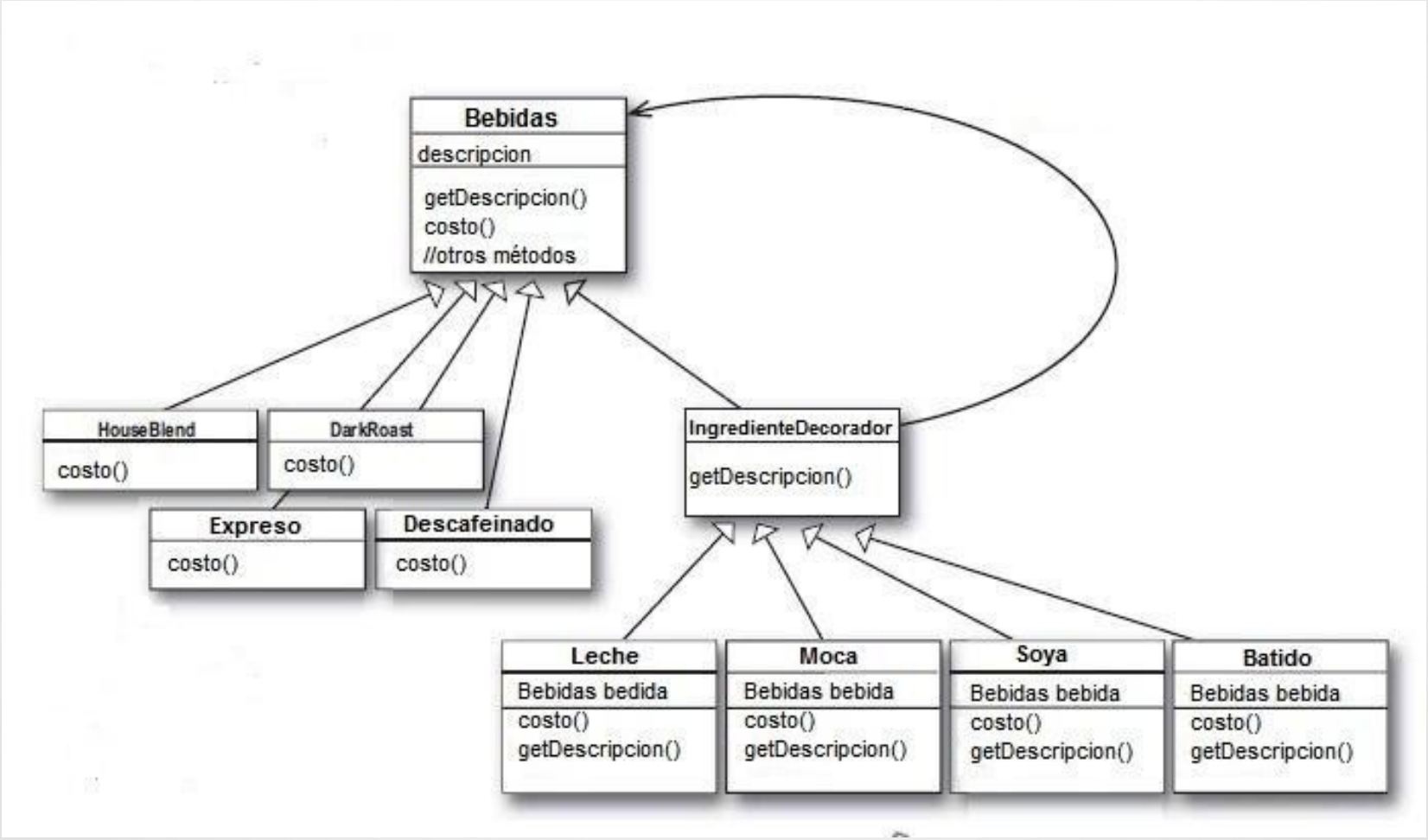
4) Ahora es el momento de calcular el costo para el cliente. Nosotros hacemos esto llamando al método cost() en el decorador externo, Batido, y Batido va a delegar el calcular el costo de los objetos que decora.

- Una vez que tenga un costo, agregará el costo del Batido.



- El patrón decorador atribuye responsabilidades adicionales a un objeto dinámicamente.
- Los decoradores proporcionan una alternativa flexible a las subclases para ampliar la funcionalidad.

El diseño final:



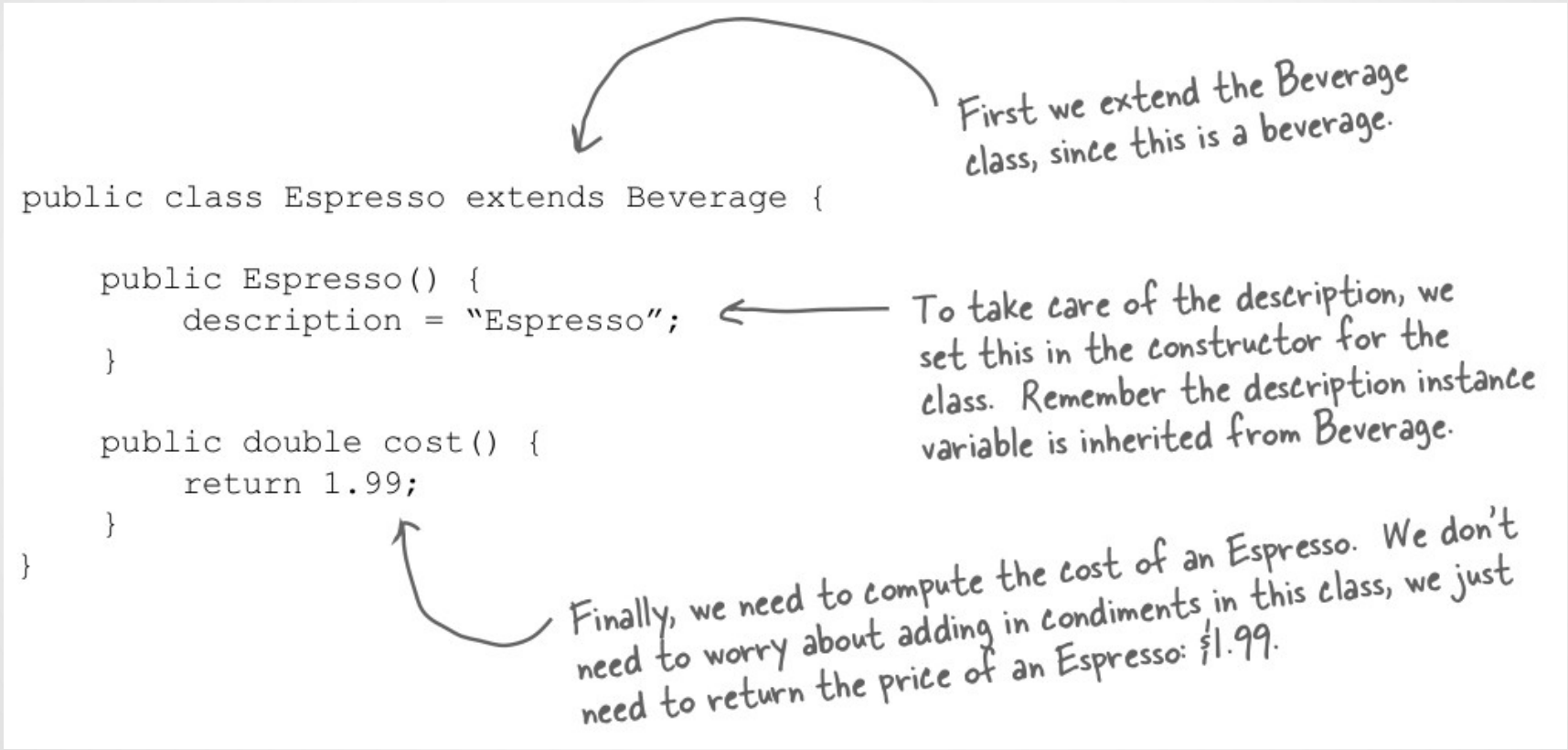
Clase Bebida

```
public abstract class Beverage {
    String description = "Unknown Beverage";

    public String getDescription() {
        return description;
    }

    public abstract double cost();
}
```

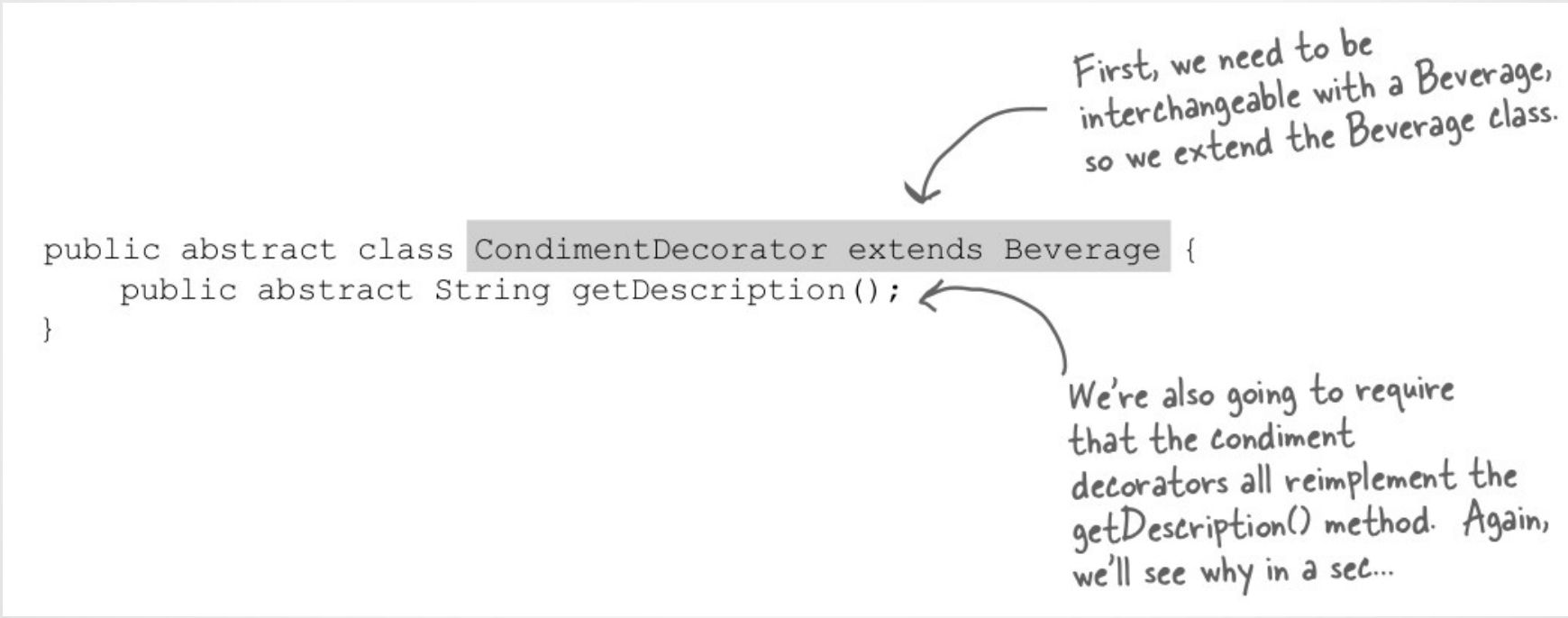
Beverage is an abstract class with the two methods getDescription() and cost().

getDescription is already implemented for us, but we need to implement cost() in the subclasses.

Clase Cafe Expresso



Clase Decoradora



Clase Moca

Mocha is a decorator, so we extend CondimentDecorator.

Remember, CondimentDecorator extends Beverage.

```
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're going to instantiate Mocha with a reference to a Beverage using:

(1) An instance variable to hold the beverage we are wrapping.

(2) A way to set this instance variable to the object we are wrapping. Here, we're going to to pass the beverage we're wrapping to the decorator's constructor.

We want our description to not only include the beverage – say "Dark Roast" – but also to include each item decorating the 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.

Now we need to compute the cost of our beverage with Mocha. First, we delegate the call to the object we're decorating, so that it can compute the cost; then, we add the cost of Mocha to the result.

Ahora tenemos:

Un diseño resistente al cambio y lo suficientemente flexible para soportar cambios en los requerimientos.

Cada componente se puede usar solo o envuelto por un decorador.

Cada decorador HAS-A (envuelve) un componente, lo que significa que el decorador tiene una variable de instancia que contiene una referencia a un componente

Clase Main

```
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.

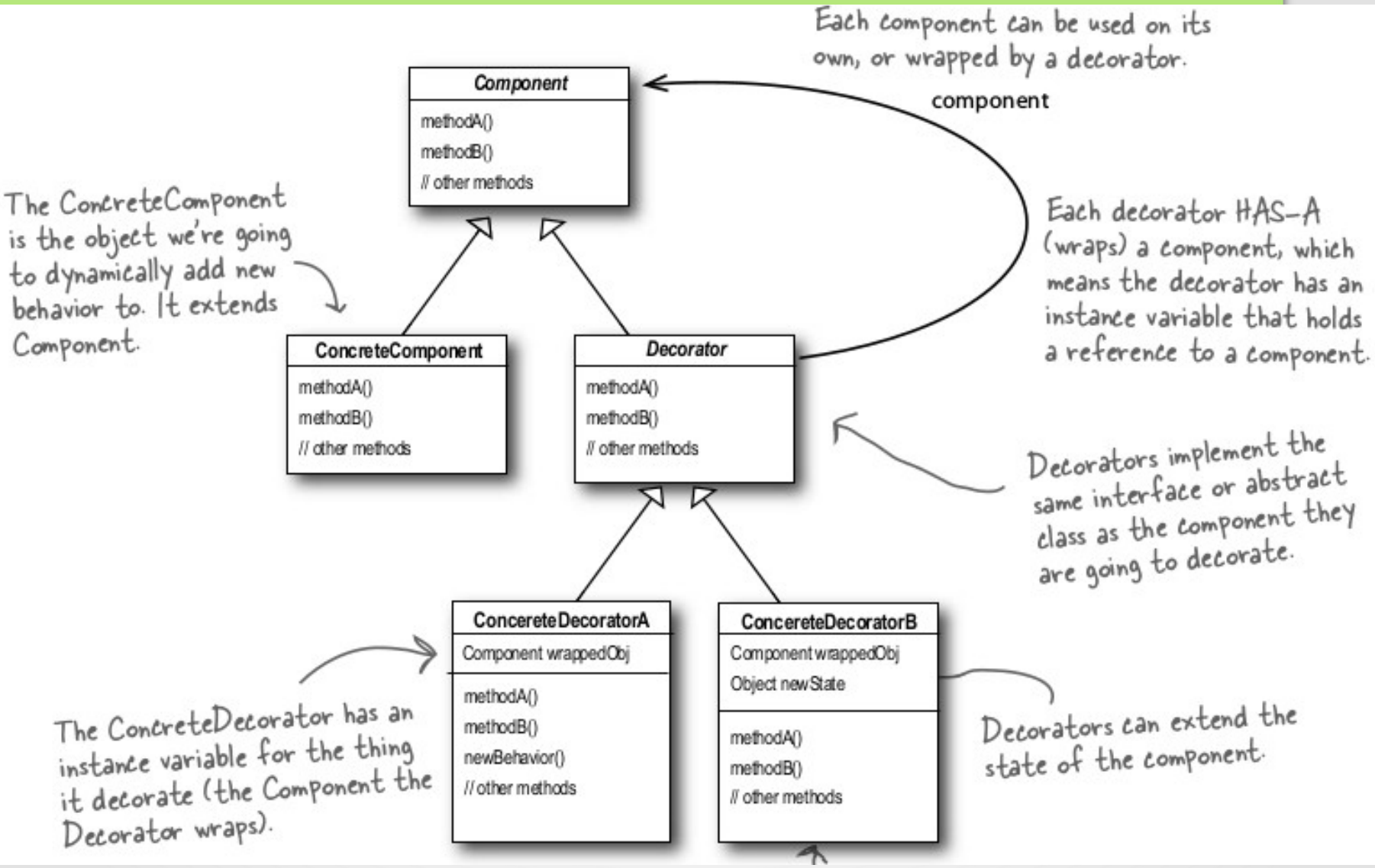
Wrap it with a Mocha.

Wrap it in a second Mocha.

Wrap it in a Whip.

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


```
File  Edit  Window  Help  CloudsInMyCoffee
% java StarbuzzCoffee
Espresso $1.99
Dark Roast Coffee, Mocha, Mocha, Whip $1.49
House Blend Coffee, Soy, Mocha, Whip $1.34
%
```


Each component can be used on its own, or wrapped by a decorator.
Component
methodA()
methodB()
// other methods
component
The ConcreteComponent is the object we're going to dynamically add new behavior to. It extends Component.
Each decorator HAS-A (wraps) a component, which means the decorator has an instance variable that holds a reference to a component.
ConcreteComponent
methodA()
methodB()
// other methods
Decorator
methodA()
methodB()
// other methods
Decorators implement the same interface or abstract class as the component they are going to decorate.
ConcereteDecoratorA
Component wrappedObj
methodA()
methodB()
newBehavior()
// other methods
ConcereteDecoratorB
Component wrappedObj
Object newState
methodA()
methodB()
// other methods
The ConcreteDecorator has an instance variable for the thing it decorate (the Component the Decorator wraps).
Decorators can extend the state of the component.

Recuerda

- Adjuntar responsabilidades adicionales a un objeto de forma dinámica.
- Decorator proporciona una alternativa flexible a la subclasificación para ampliar la funcionalidad.