Decorator

CSCI 4448/5448: Object-Oriented Analysis & Design

Lecture 13

Acknowledgement & Materials Copyright

- I'd like to start by acknowledging Dr. Ken Anderson
- Ken is a Professor and the Chair of the Department of Computer Science
- Ken taught OOAD on several occasions, and has graciously allowed me to use his copyrighted material for this instance of the class
- Although I will modify the materials to update and personalize this class, the original materials this class is based on are all copyrighted © Kenneth M. Anderson; the materials are used with his consent; and this use in no way challenges his copyright

What are these things in Java?

- @BeanParam()? or @Inject?

- These are annotations in Java
  - @BeanParam() is for a framework called JAX-RS for RESTful web services
  - @Inject is for javax.inject to identify injectible constructors, mentods, and fields when using dependency injection
- Annotations allow you to add metadata (extra information) to a program
- Often used by frameworks to enable additional functionality for compilation or runtime behavior

- Built in common annotations in Java are:
  - @Override – says the following method overrides an inherited one
  - @SuppressWarnings/@SafeVarargs – suppresses certain warnings
  - @Depreciated – marks part of an interface as depreciated
  - @FunctionalInterface – marks that an Interface has a single function and can be used with lambda notation
  - @Native – tags constants used by native code

Decorator Pattern

- The Decorator Pattern provides a powerful mechanism for adding new behaviors to an object at run-time
  - The mechanism is based on the notion of “wrapping” which is just a fancy way of saying “delegation” **but with the added twist that the delegator and the delegate both implement the same interface**
  - You start with object A that implements interface C
  - You then create object B that also implements interface C
  - You pass A into B’s constructor and then pass B to A’s client
  - The client thinks its talking to A (via C’s interface) but its actually talking to B
  - B’s methods augment A’s methods to provide new behavior

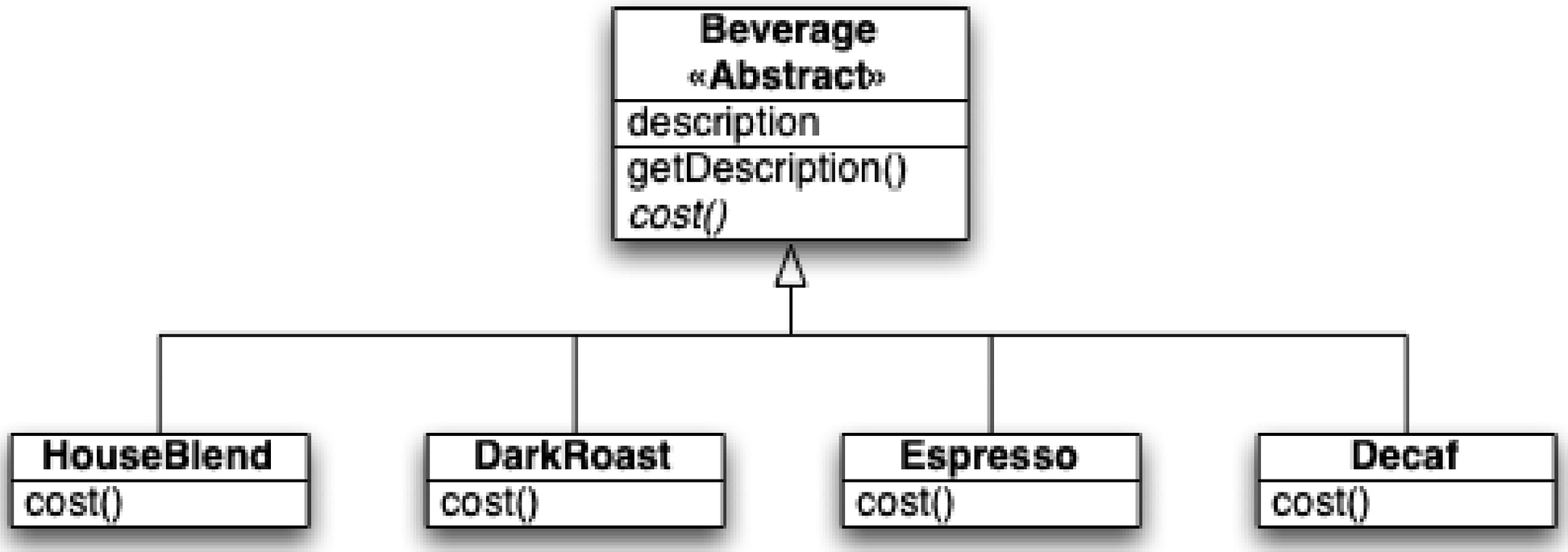
Why? Open-Closed Principle

- The decorator pattern provides yet another way in which a class's runtime behavior can be extended without requiring modification to the class
- This supports the goal of the open-closed principle:
  - Classes should be open for extension but closed to modification
  - Inheritance is one way to do this, but composition and delegation are more flexible (and Decorator takes advantage of delegation)
  - As the Gang of Four put it: "Decorator lets you attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality."
- Our "Starbuzz Coffee" example, taken from Head First Design Patterns, clearly demonstrates why inheritance can get you into trouble and why delegation/composition provides greater run-time flexibility

Starbuzz Coffee

- Under pressure to update their “point of sale” system to keep up with their expanding set of beverage products
  - Started with a Beverage abstract base class and four implementations: HouseBlend, DarkRoast, Decaf, and Espresso
    - Each beverage can provide a description and compute its cost
  - But they also offer a range of condiments including: steamed milk, soy, and mocha
- These condiments **alter** a beverage’s description and cost
  - The use of the word “Alter” here is key since it provides a hint that we might be able to use the Decorator pattern

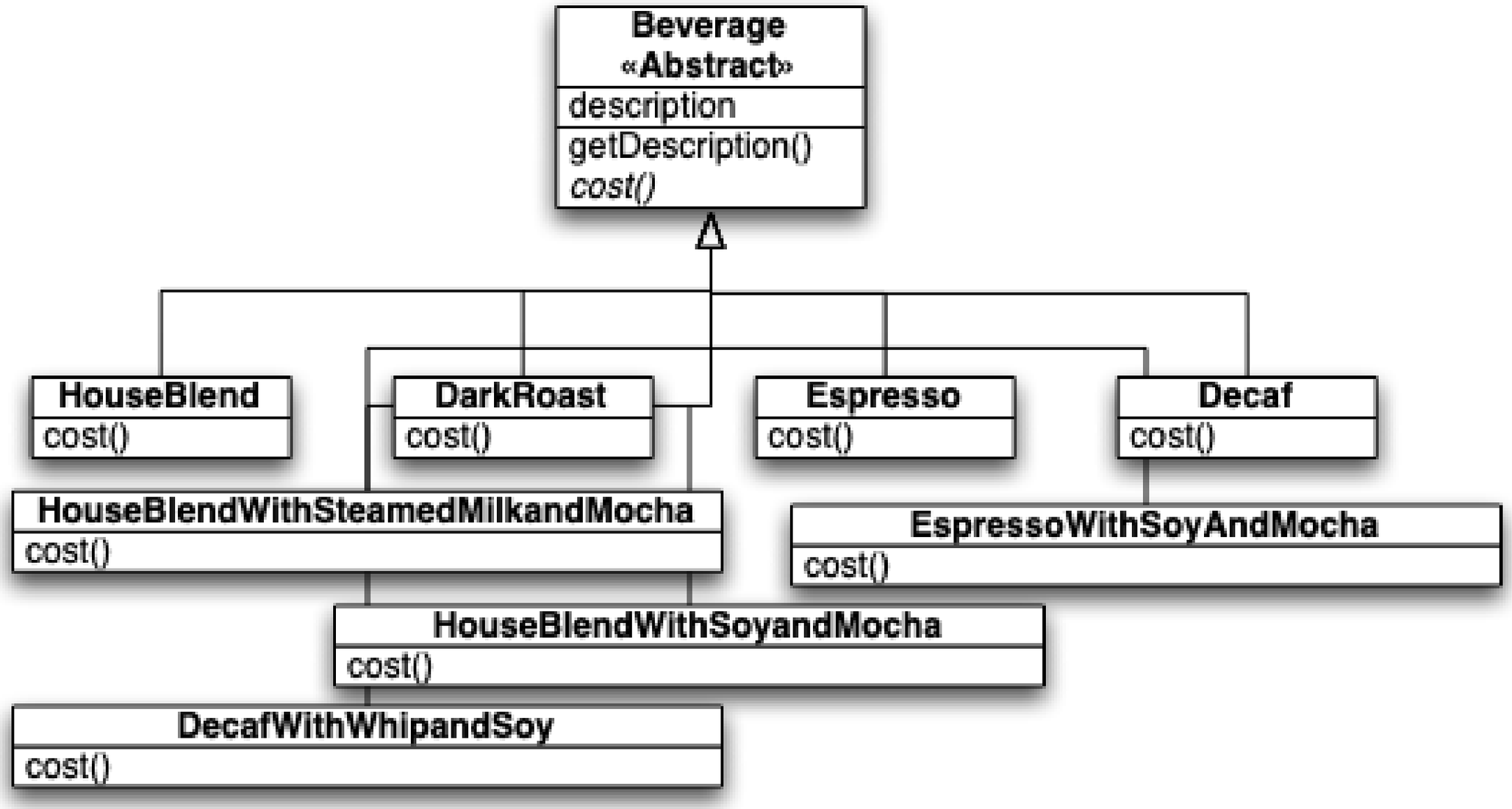
Initial Starbuzz System



With inheritance on your brain, you may add condiments to this design in one of two ways

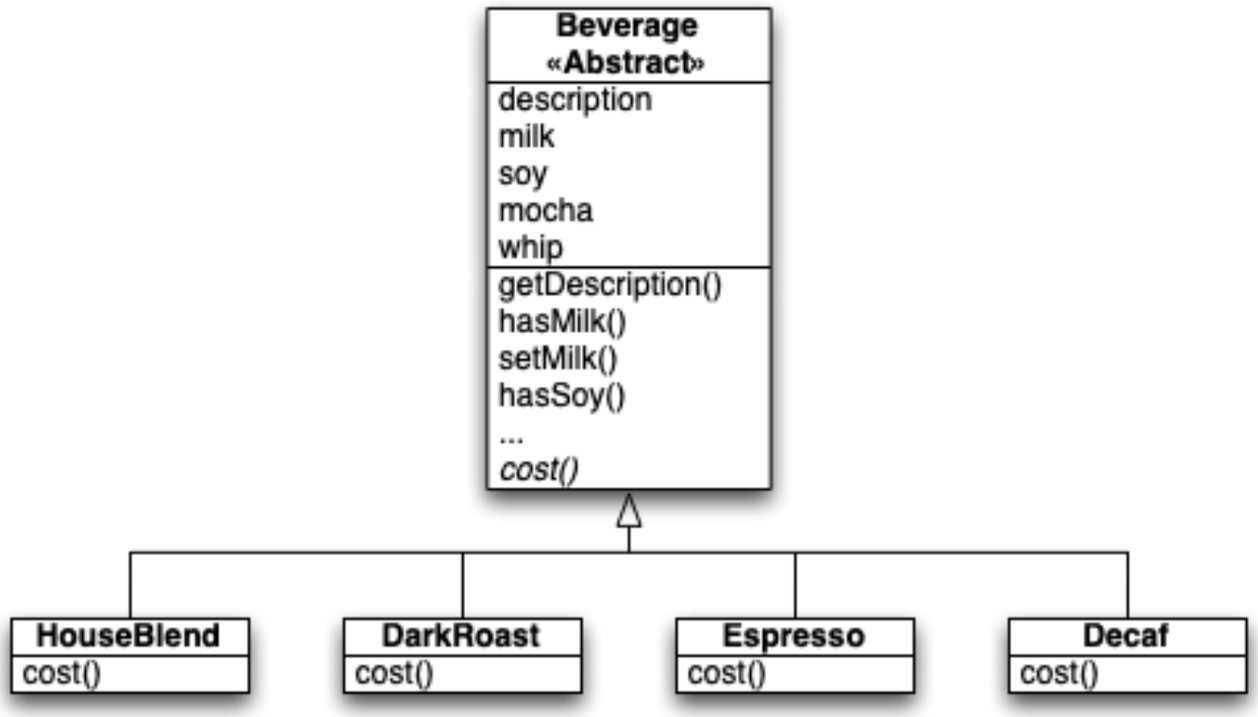
1. One subclass per combination of condiment (won't work in general)
2. Add condiment handling to the Beverage superclass

Approach One: One Subclass per Combination



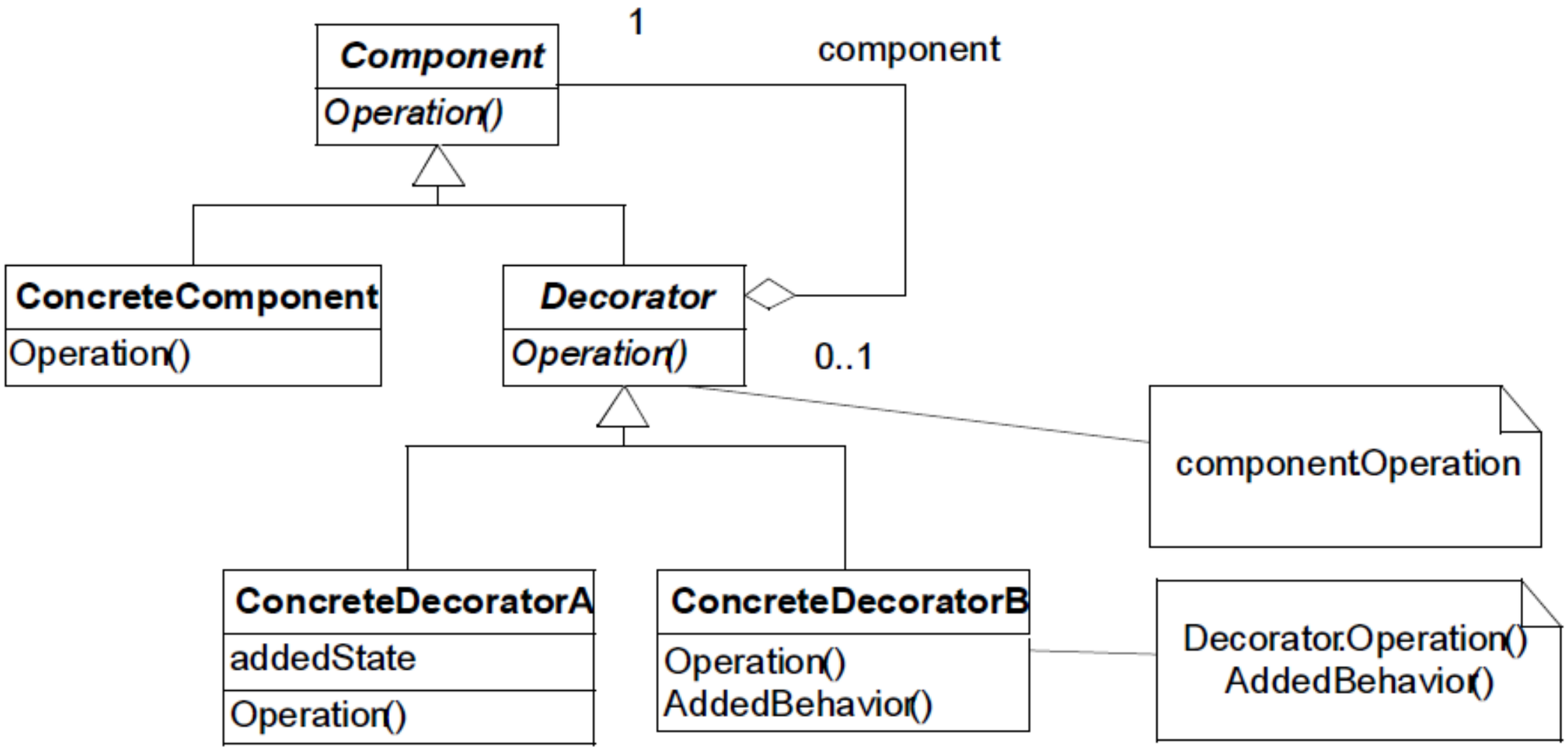
This is incomplete, but you can see the problem…

Approach Two: Let Beverage Handle Condiments



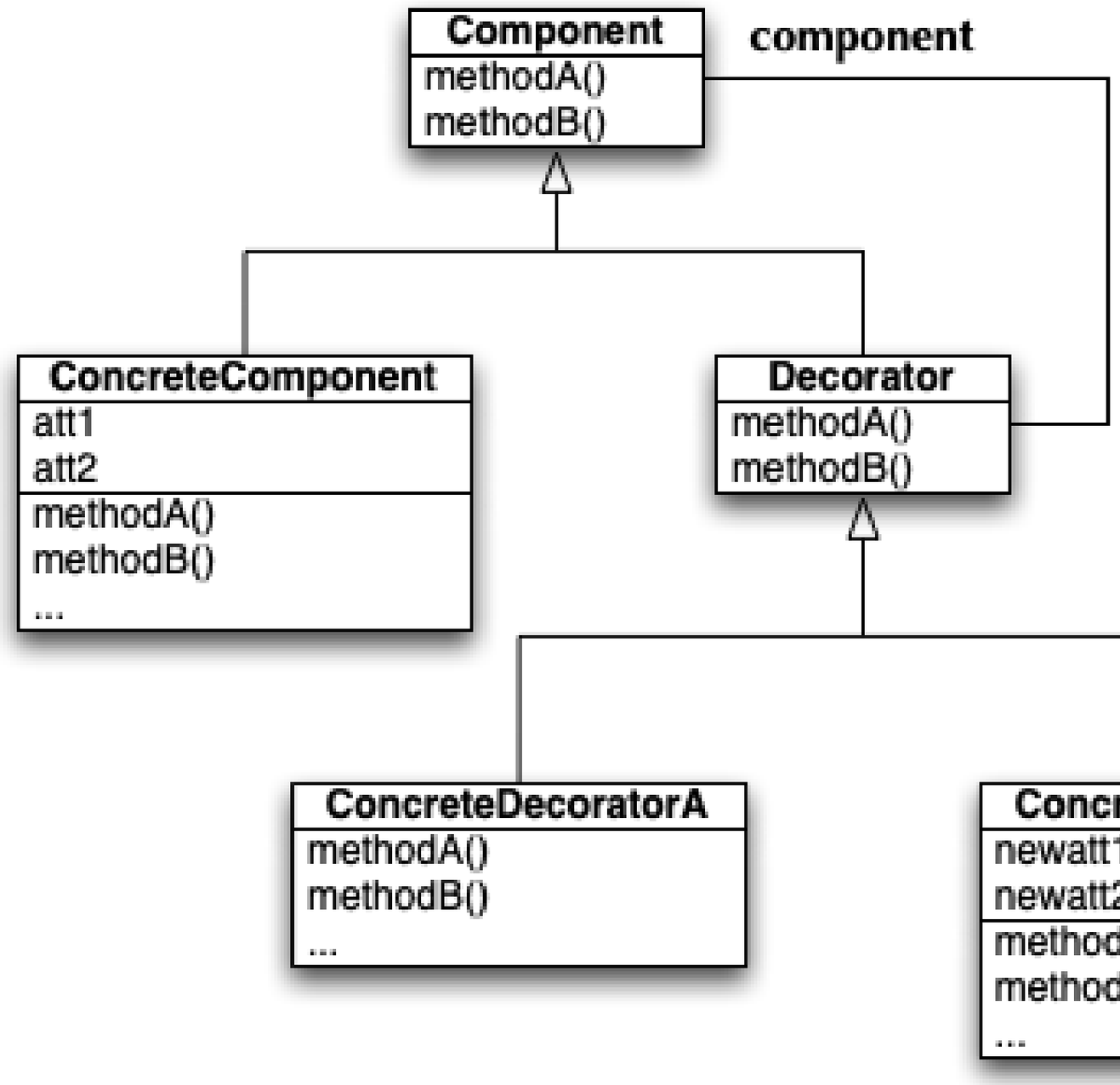
1. This assumes that all concrete Beverage classes need these condiments
2. Condiments may vary (old ones go, new ones are added, price changes occur, etc.), shouldn't Beverage be encapsulated from this some how?
3. How do you handle "double soy" drinks with boolean variables?

Decorator Pattern: Definition and Structure



The intent of the pattern is to attach additional responsibilities to an object dynamically. Decorators create chains of objects that start with the Decorators and end with the Concrete Component.

Decorator Pattern: Definition and Structure



Inheritance is to make sure that components and decorators share the same interface: namely used the public interface of Component which is either an abstract class or an interface.

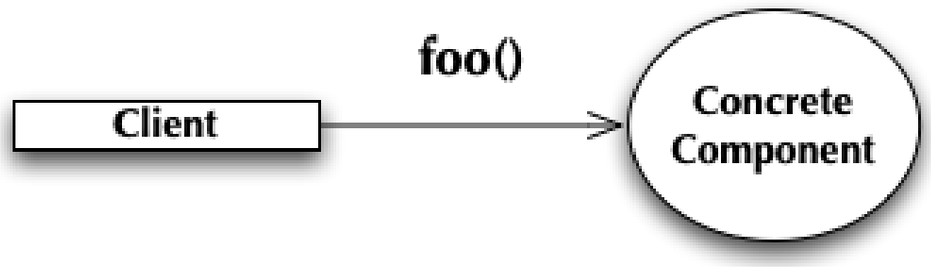
Each decorator HAS-A reference to a component.

At run-time, concrete decorators wrap concrete components and/or other concrete decorators

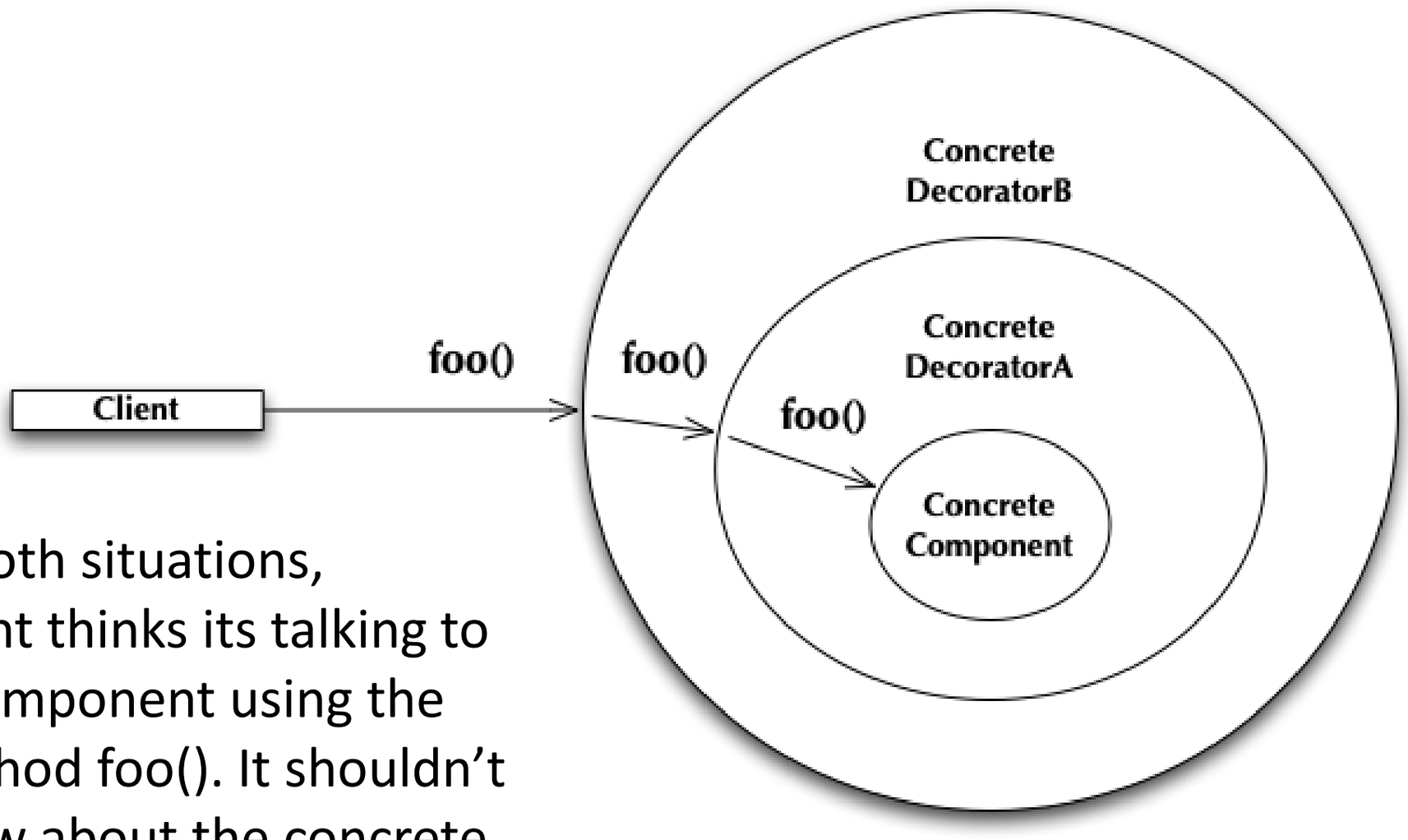
The object to be wrapped is typically passed in via the constructor

Each decorator is cohesive, focusing just on its added functionality

Client Perspective



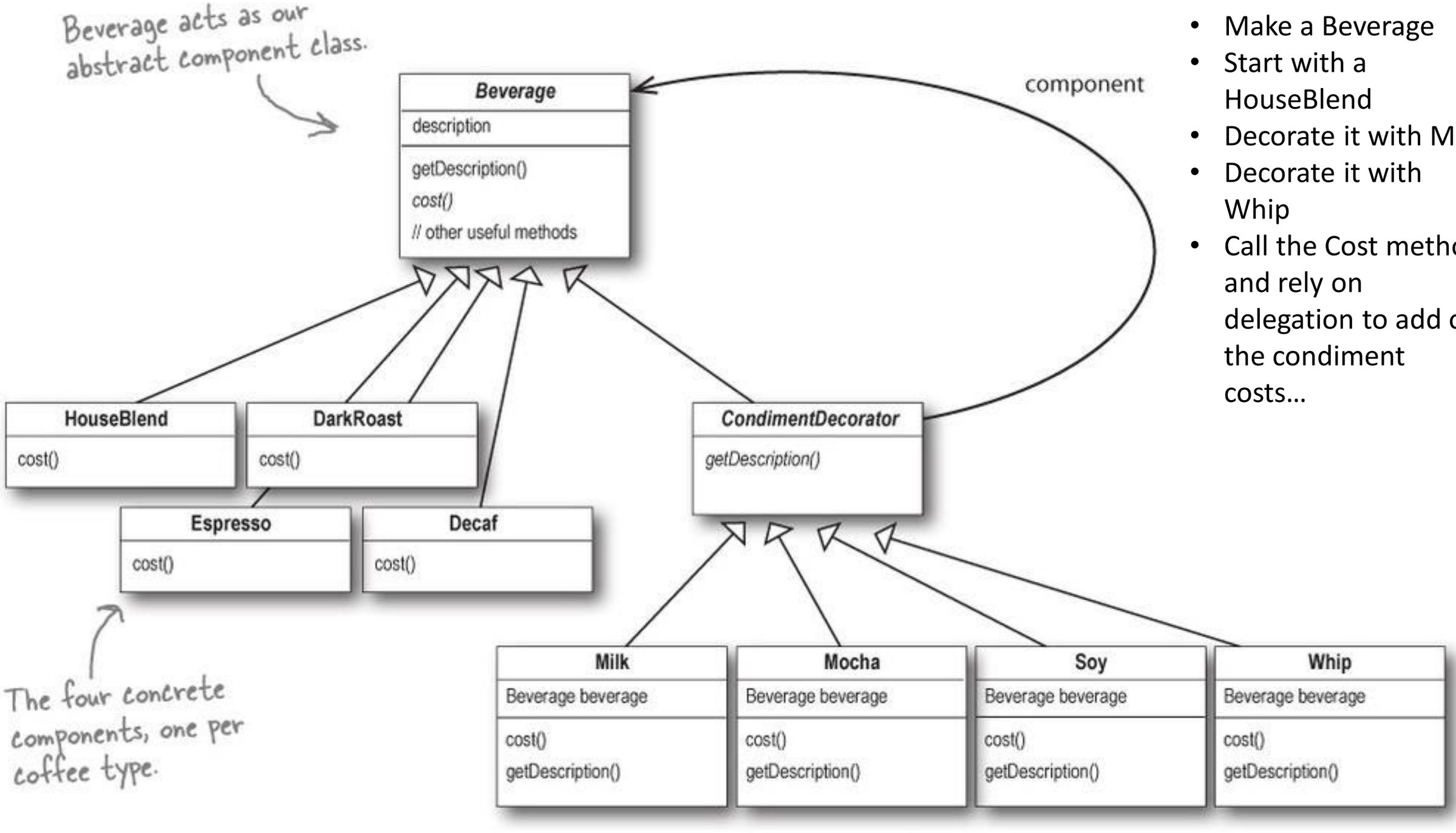
BEFORE



AFTER

In both situations,
Client thinks its talking to a Component using the method foo(). It shouldn't know about the concrete subclasses.

StarBuzz Using Decorators



- Make a Beverage
- Start with a HouseBlend
- Decorate it with Milk
- Decorate it with Whip
- Call the Cost method and rely on delegation to add on the condiment costs…

Create the base and decorator classes

```
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.

First, we need to be interchangeable with a Beverage, so we extend the Beverage class.

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

We're also going to require that the condiment decorators all reimplement the getDescription() method. Again, we'll see why in a sec...

Create instances of beverages

First we extend the Beverage class, since this is a beverage.

```
public class Espresso extends Beverage {

    public Espresso() {
        description = "Espresso";
    }

    public double cost() {
        return 1.99;
    }
}
```

To take care of the description, we set this in the constructor for the class. Remember, the description instance variable is inherited from Beverage.

Finally, we need to compute the cost of an Espresso. We don't need to worry about adding in condiments in this class, we just need to return the price of an Espresso: $1.99.

Create instances of decorators

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 beverage.cost() + .20;
    }
}
```

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 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.

Using the Decorator (Java example)

```
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 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.

What if I added sizes…

- …and the sizes change the cost of beverages and condiments? How do the decorators change?

```
public abstract class Beverage {
        public enum Size { TALL, GRANDE, VENTI };
        Size size = Size.TALL;
        String description = "Unknown Beverage";
        public String getDescription() {
                return description;
        }
        public void setSize(Size size) {
                this.size = size;
        }
        public Size getSize() {
                return this.size;
        }
        public abstract double cost();
}
```

Size…

- I'll end up making changes to concrete beverage and condiment decorator classes anywhere that the size might change the cost…

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

    public Size getSize() {
        return beverage.getSize();
    }
}

public class Soy extends CondimentDecorator {
    public Soy(Beverage beverage) {
        this.beverage = beverage;
    }

    public String getDescription() {
        return beverage.getDescription() + ", Soy";
    }

    public double cost() {
        double cost = beverage.cost();
        if (beverage.getSize() == Size.TALL) {
            cost += .10;
        } else if (beverage.getSize() == Size.GRANDE) {
            cost += .15;
        } else if (beverage.getSize() == Size.VENTI) {
            cost += .20;
        }
        return cost;
    }
}
```

We moved the Beverage instance variable into CondimentDecorator, and added a method, getSize() for the decorators that simply returns the size of the beverage.

Here we get the size (which propagates all the way to the concrete beverage) and then add the appropriate cost.

Python - Decorator

In this example of using Decorator, we will make our beverage handler as in the Java example, with a core beverage class and a decorator class for condiments.

Note that the Decorator is inheriting from Beverage…

Again, using classes instead of abstract classes is a bit weak, but the NotImplementedError throws will remind us when we've missed providing appropriate overrides in implementations.

```
class Beverage:
    def __init__(self):
        self.description = "Unknown Beverage"

    def get_description(self):
        return self.description

    def cost(self):
        raise NotImplementedError
```

```
class CondimentDecorator(Beverage):

    def get_description(self):
        raise NotImplementedError

    def cost(self):
        raise NotImplementedError
```

Example from:
https://github.com/jtortorelli/head-first-design-patterns-python/blob/master/src/python/chapter_3/starbuzz.py

Python - Decorator

Now we need to use the Beverage and Condiment Decorator classes to build out our concrete implementations…

When I instantiate a Condiment, I have to provide a reference to the beverage to maintain the building of cost…

```
class Espresso(Beverage):
    def __init__(self):
        super().__init__()
        self.description = "Espresso"

    def cost(self):
        return 1.99

# we'll do these the same way…
#class HouseBlend(Beverage):
#class Decaf(Beverage):
#class DarkRoast(Beverage):

class Mocha(CondimentDecorator):
    def __init__(self, beverage):
        super().__init__()
        self.beverage = beverage

    def get_description(self):
        return self.beverage.get_description() + ", Mocha"

    def cost(self):
        return 0.20 + self.beverage.cost()

# we'll add these other decorators the same way
#class Soy(CondimentDecorator):
#class Whip(CondimentDecorator):
#class SteamedMilk(CondimentDecorator):
```

Example from:
https://github.com/jtortorelli/head-first-design-patterns-python/blob/master/src/python/chapter_3/starbuzz.py

Python - Decorator

Just as in the Java example, the decorators will add their text and cost to each beverage made…

```
# main – make some drinks
b = Espresso()
print(f"{b.get_description()} ${b.cost()}")

b2 = DarkRoast()
b2 = Mocha(b2)
b2 = Mocha(b2)
b2 = Whip(b2)
print(f"{b2.get_description()} ${b2.cost()}")

b3 = HouseBlend()
b3 = Soy(b3)
b3 = Mocha(b3)
b3 = Whip(b3)
print(f"{b3.get_description()} ${b3.cost()}")
```

Example from:
https://github.com/jtortorelli/head-first-design-patterns-python/blob/master/src/python/chapter_3/starbuzz.py

Other Decorator thoughts

- Decorator lets you assign extra behaviors to objects at runtime without breaking the code that uses these objects – “Wrapper”
- Decorator lets you structure business logic into layers, create a decorator for each layer and compose objects with various combinations of this logic at runtime; client code can treat all these objects in the same way, since they all follow a common interface
  - Likewise, that added logic can easily be removed if not needed or appropriate
- Decorator can be used when it’s awkward or not possible to extend an object’s behavior using inheritance
  - Decorator can extend behavior(s) without making a new subclass
  - Many programming languages have the final keyword that can be used to prevent further extension of a class
  - For a final class, the only way to reuse the existing behavior would be to wrap the class with your own wrapper, using the Decorator pattern

- https://refactoring.guru/design-patterns/decorator

Other Decorator Issues

- Decorator is more flexible than inheritance for adding responsibilities
- Decorator avoids having feature packed classes higher in an inheritance structure – you don’t have to know everything you’ll be adding from the beginning
- Decorators and components are not identical – be careful of object identity dependence
- Decorators have to maintain the decorated component’s interface
- Tends toward lots of small objects – harder to understand or debug?
- Generally, the Decorator is abstract with concrete decorator instances – this is only needed if you have more than one decorator type
- Compared with Strategy – Decorator is making changes external to a class's internal implementation, Strategy makes those changes within the class
- http://www.cs.unc.edu/~stotts/GOF/hires/pat4dfso.htm

Summary

- The Decorator pattern (aka “Wrapper”) comes into play when there are a variety of optional functions that can precede or follow another function that is always executed
- This is a very powerful idea that can be implemented in a variety of ways
- The fact that all of the classes in the decorator pattern hide behind the abstraction of Component enables all of the good benefits of OO design discussed previously
- Supports open-closed principle – avoids changing working code

Next Steps

All staff now providing office hours:
Office hours are posted in Canvas Announcements and Piazza posts

- Latest
  - Stop by and meet Sudarshan Sridhar (susr2424@colorado.edu)
  - A lot in Project 3, don't wait too long…
- Assignments
  - New Piazza participation topic this week, keep up to get those 100 points!
  - Project 3 is up on Canvas – for your two-person team!
    - Part 1 (UML and OO questions) is due Wed 9/28
    - Part 2 (code submission) is due Wed 10/5
  - New Quiz 3 opened on Sat 9/17, due Thur 9/22; new Quiz this Sat 9/24
  - Graduate Research Project team topics are still under review, next up is the Outline due Fri 9/30
- Make sure you're getting Piazza and Canvas notifications
- Coming up
  - Next up: OO patterns and principles – Problem/Solution example, Factory patterns
  - Head First Design Patterns Textbook: Chap. 1 is Pattern/Strategy intro, Chap. 2 is Observer, Chap. 3 is Decorator, Chap. 4 is Factory – review as needed for different perspectives and descriptions – plus full(er) code examples
  - All textbook code examples are at https://github.com/bethrobson/Head-First-Design-Patterns
- Please come find us for any help you need or questions you have!