# CSE 213: Software Engineering

# Structural Design Patterns

Nafis Tahmid

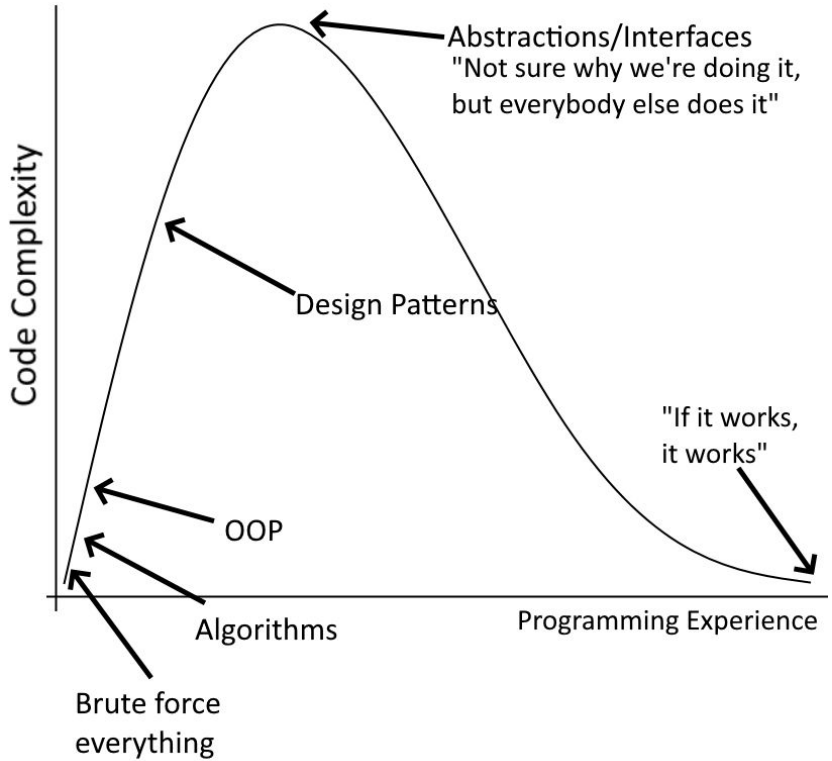
Lecturer, CSE, BUET

# Structural Design Pattern Defined

Structural pattern explain how to assemble objects and classes into *larger structures*, while keeping this structure *flexible and efficient*.

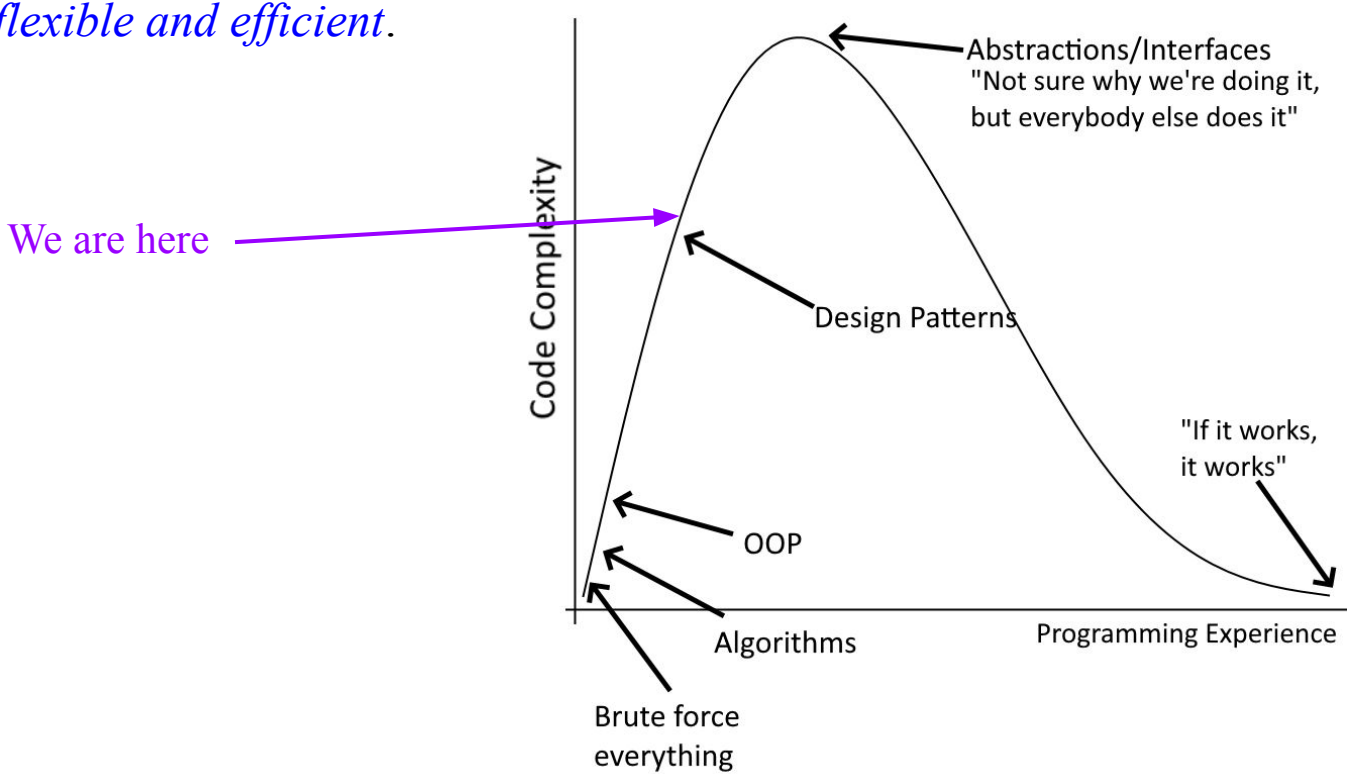
# Structural Design Pattern Defined

Structural pattern explain how to assemble objects and classes into *larger structures*, while keeping this structure *flexible and efficient*.



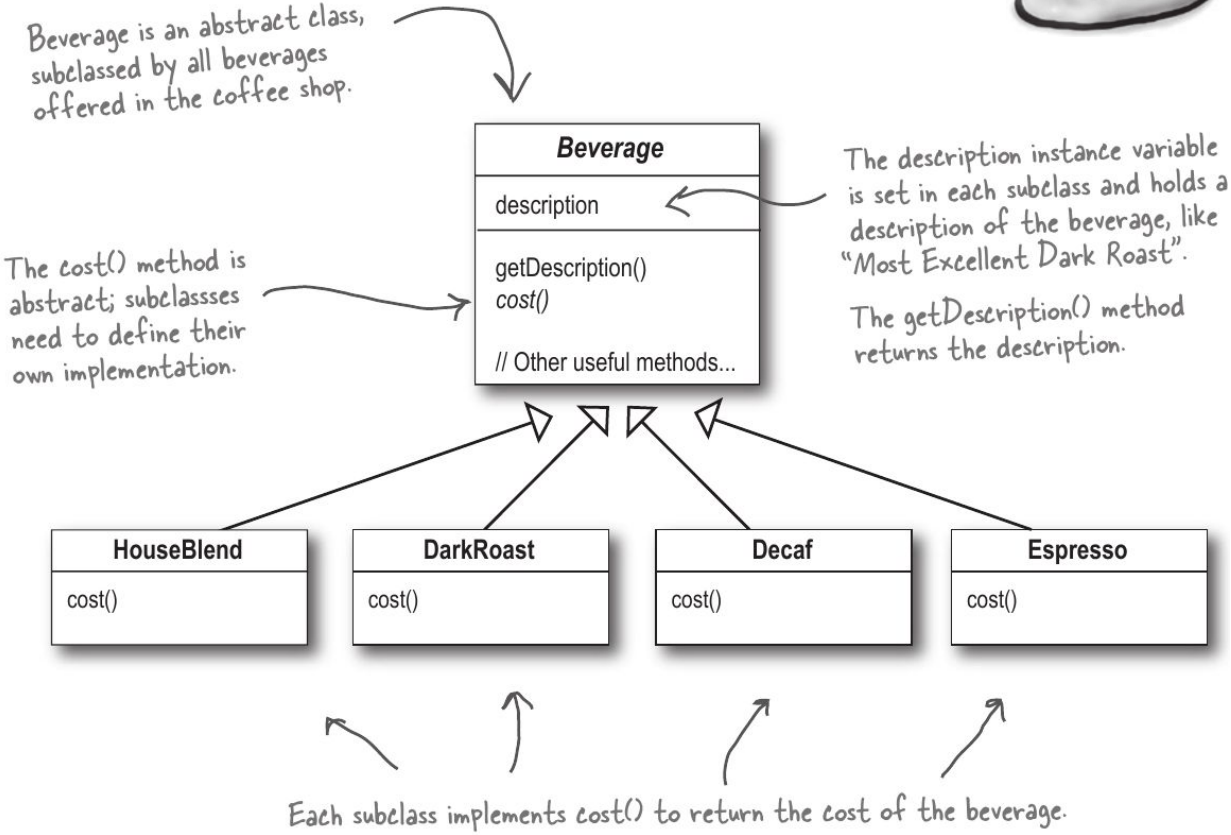
# Structural Design Pattern Defined

Structural pattern explain how to assemble objects and classes into *larger structures*, while keeping this structure *flexible and efficient*.



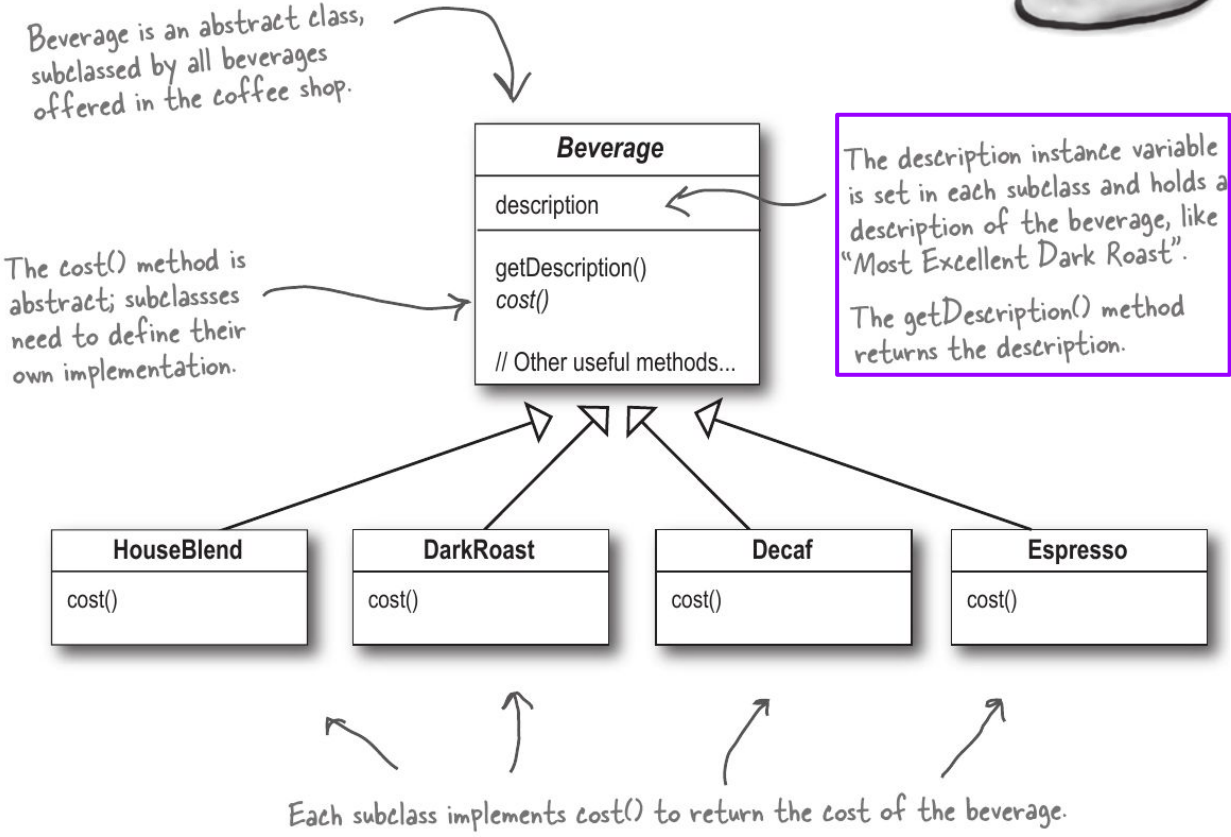
# Starbuzz Coffee

At first, they had simple menu.



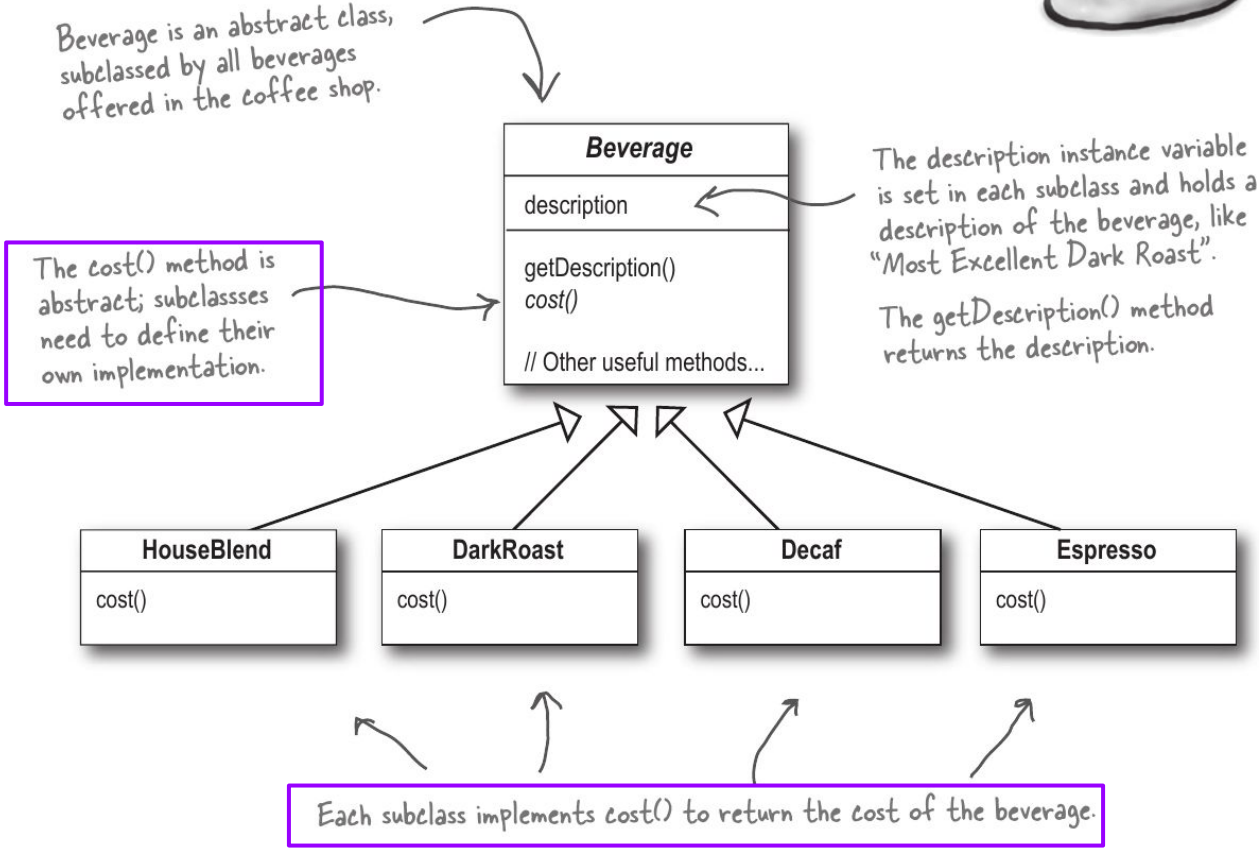
# Starbuzz Coffee

At first, they had simple menu.



# Starbuzz Coffee

At first, they had simple menu.

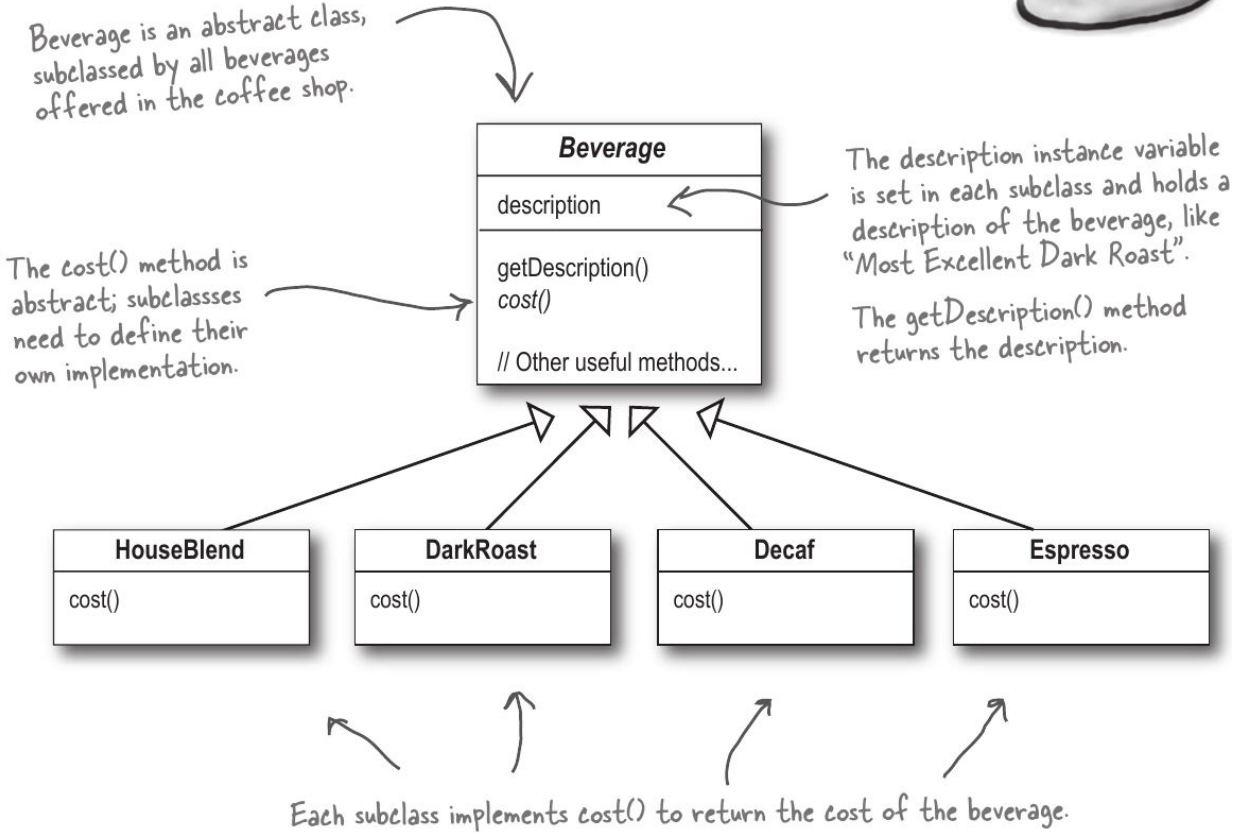


# Starbuzz Coffee

At first, they had simple menu.

Now you need to add condiments

- Steamed milk
- Soy
- Mocha
- Whipped milk



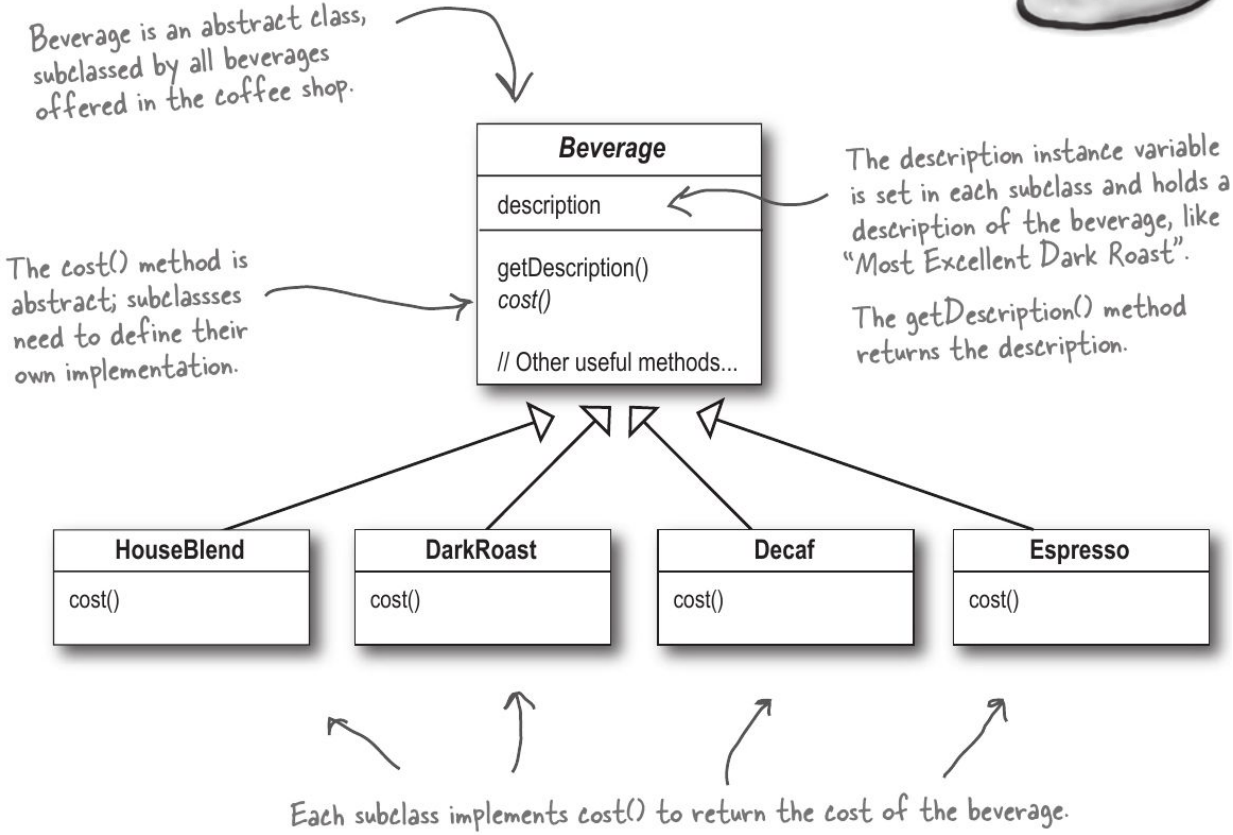
# Starbuzz Coffee

At first, they had simple menu.

Now you need to add condiments

- Steamed milk
- Soy
- Mocha
- Whipped milk

*Clients may take conds. or not*



# Starbuzz Coffee

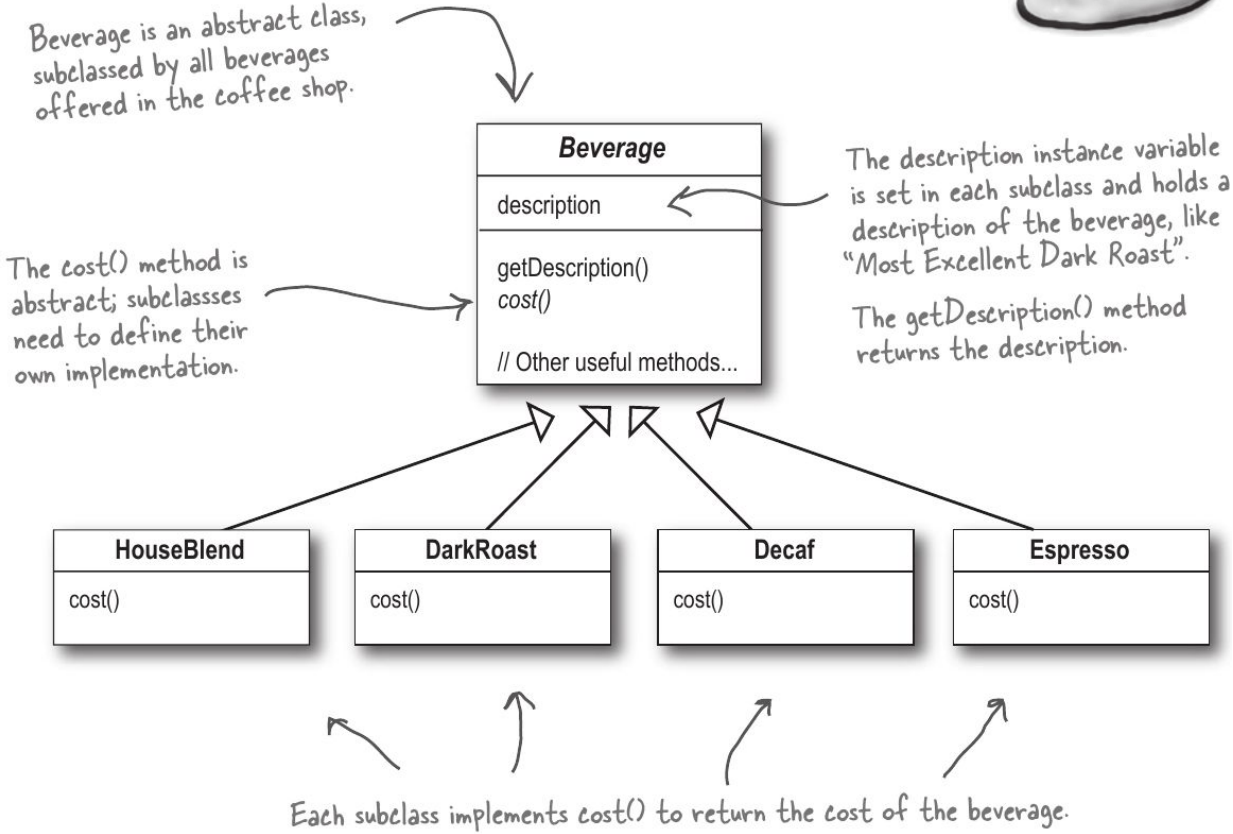
At first, they had simple menu.

Now you need to add condiments

- Steamed milk
- Soy
- Mocha
- Whipped milk

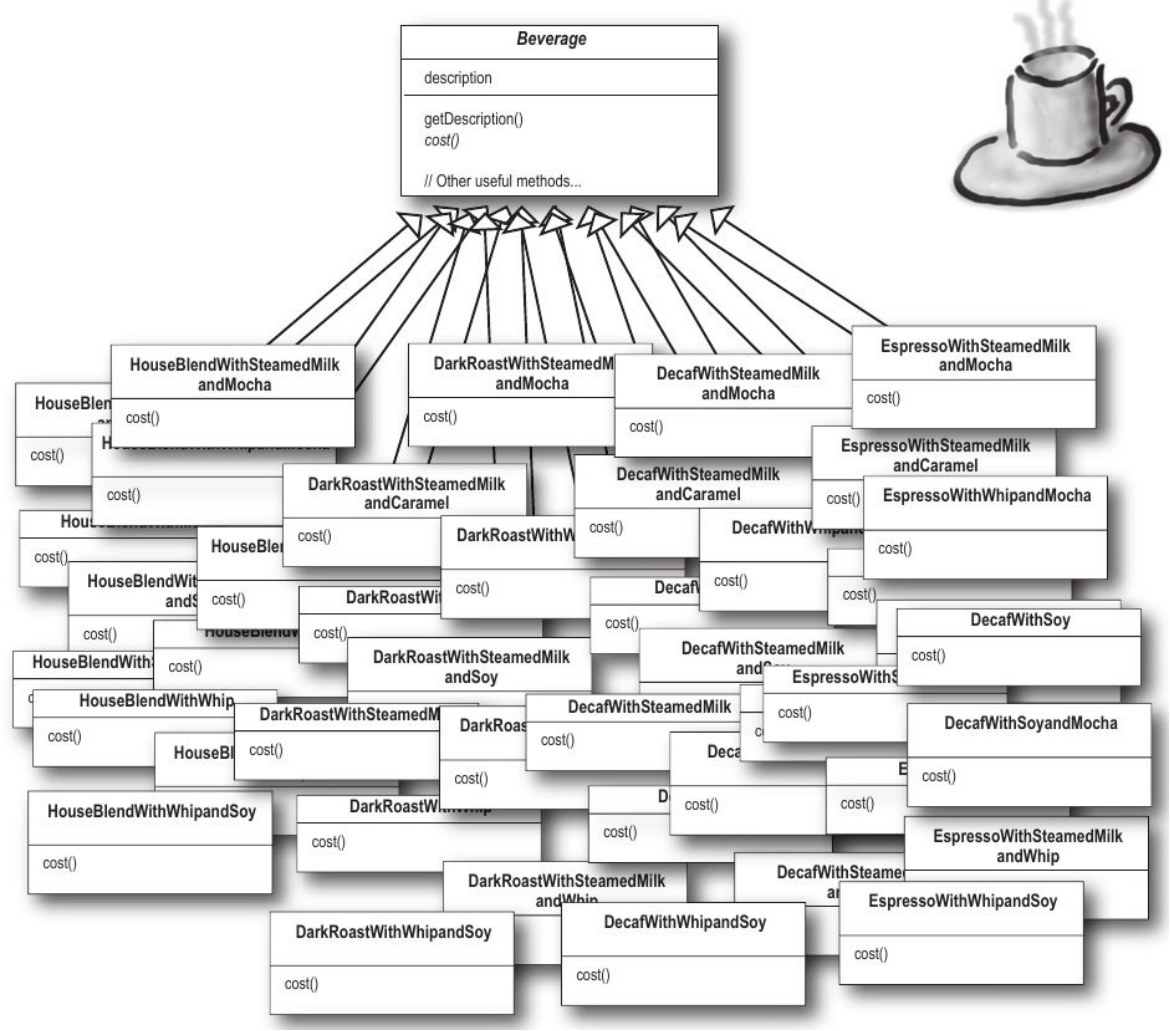
*Clients may take conds. or not*

How to handle this?



# Subclass explosion?

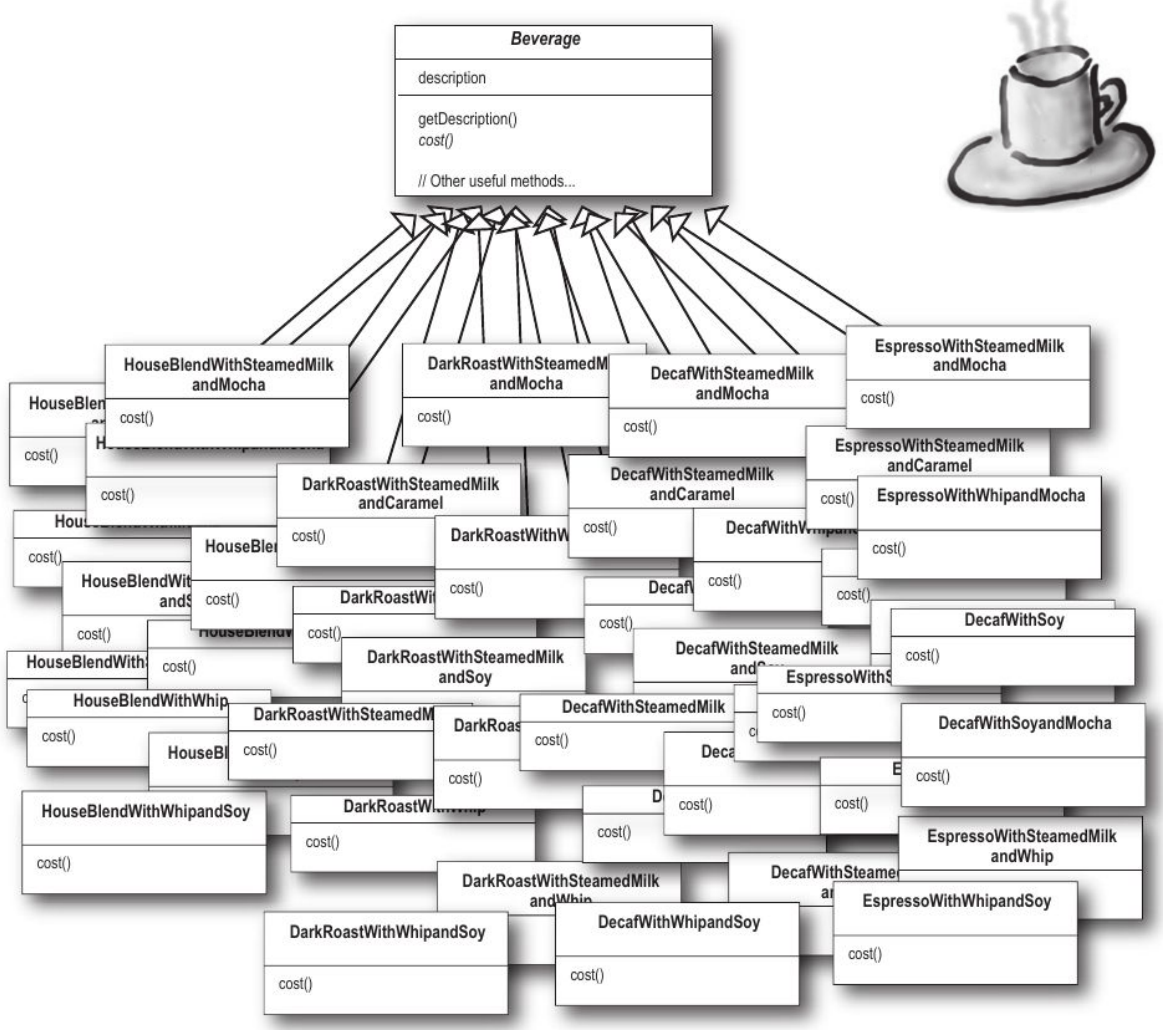
Each cost() computes the cost of *coffee + the condiments*



# Subclass explosion?

Each cost() computes the cost of *coffee + the condiments*

Definitely not an option…

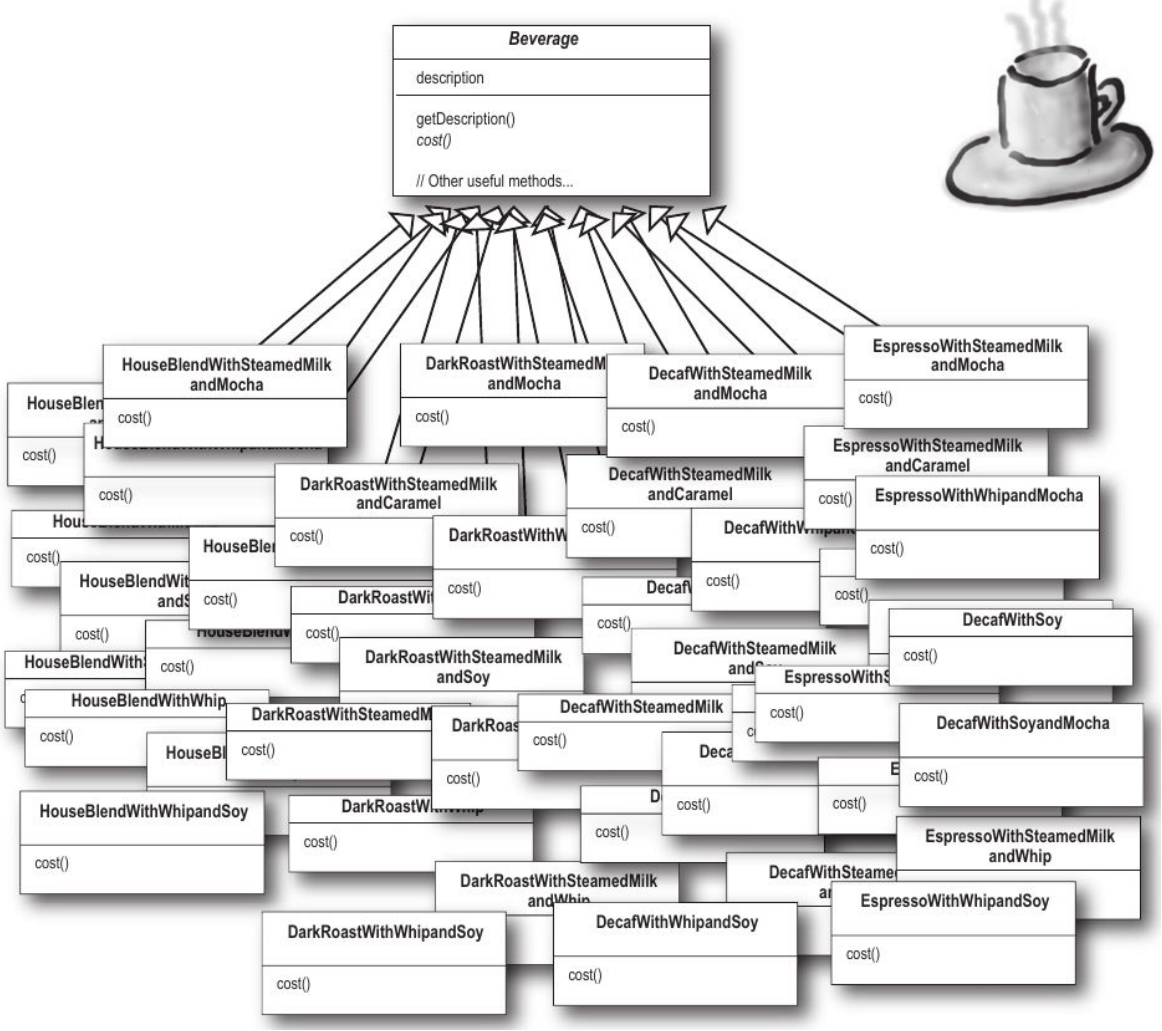


# Subclass explosion?

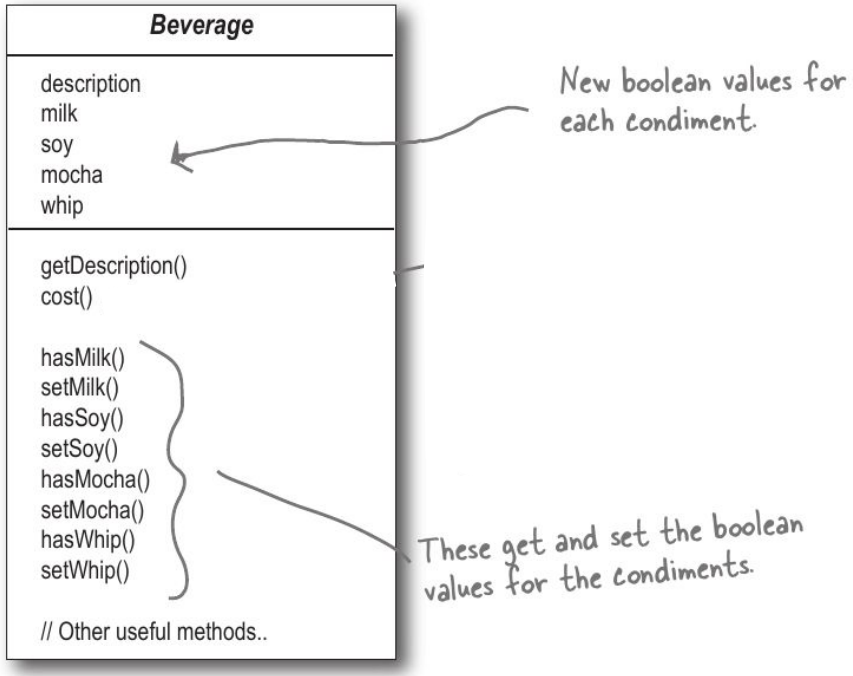
Each cost() computes the cost of *coffee + the condiments*

Definitely not an option…

This is stupid; why do we need all these classes? Can't we just use instance variables and inheritance in the superclass to keep track of the condiments?

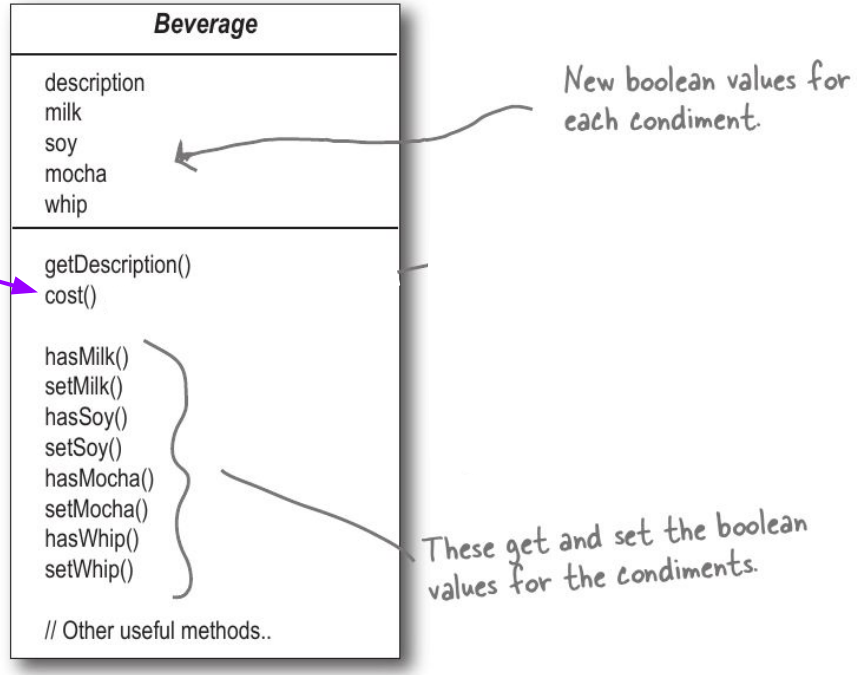


# Boolean Values



# Boolean Values

Beverage class's cost method calculates the cost of condiments.



# The design using Booleans

The superclass cost() will calculate the costs for all of the condiments, while the overridden cost() in the subclasses will extend that functionality to include costs for that specific beverage type.

Each cost() method needs to compute the cost of the beverage and then add in the condiments by calling the superclass implementation of cost().

**Beverage**

description
milk
soy
mocha
whip

getDescription()
cost()

hasMilk()
setMilk()
hasSoy()
setSoy()
hasMocha()
setMocha()
hasWhip()
setWhip()

// Other useful methods..

```
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;
}
```

**HouseBlend**
cost()

**DarkRoast**
cost()

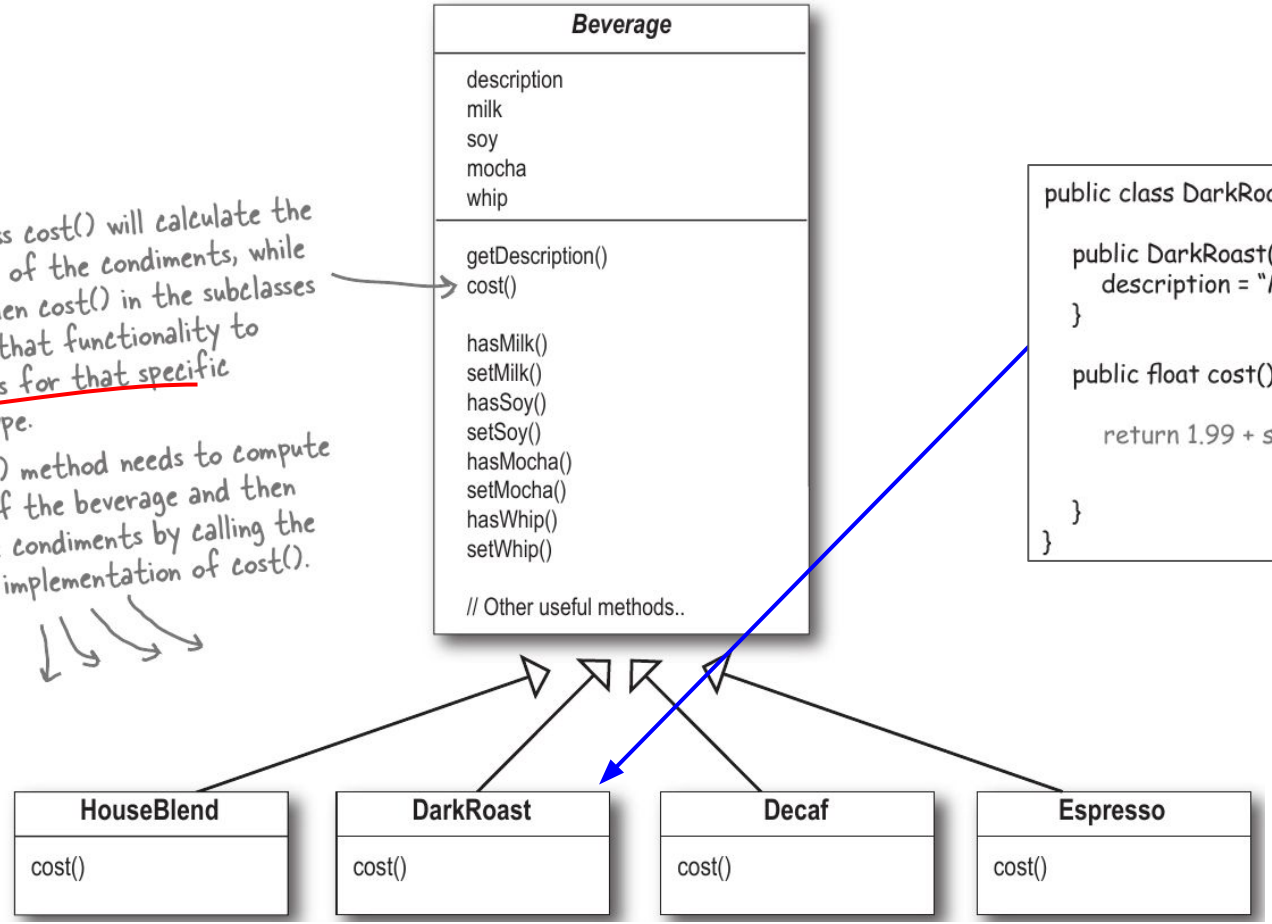
**Decaf**
cost()

**Espresso**
cost()

# The design using Booleans

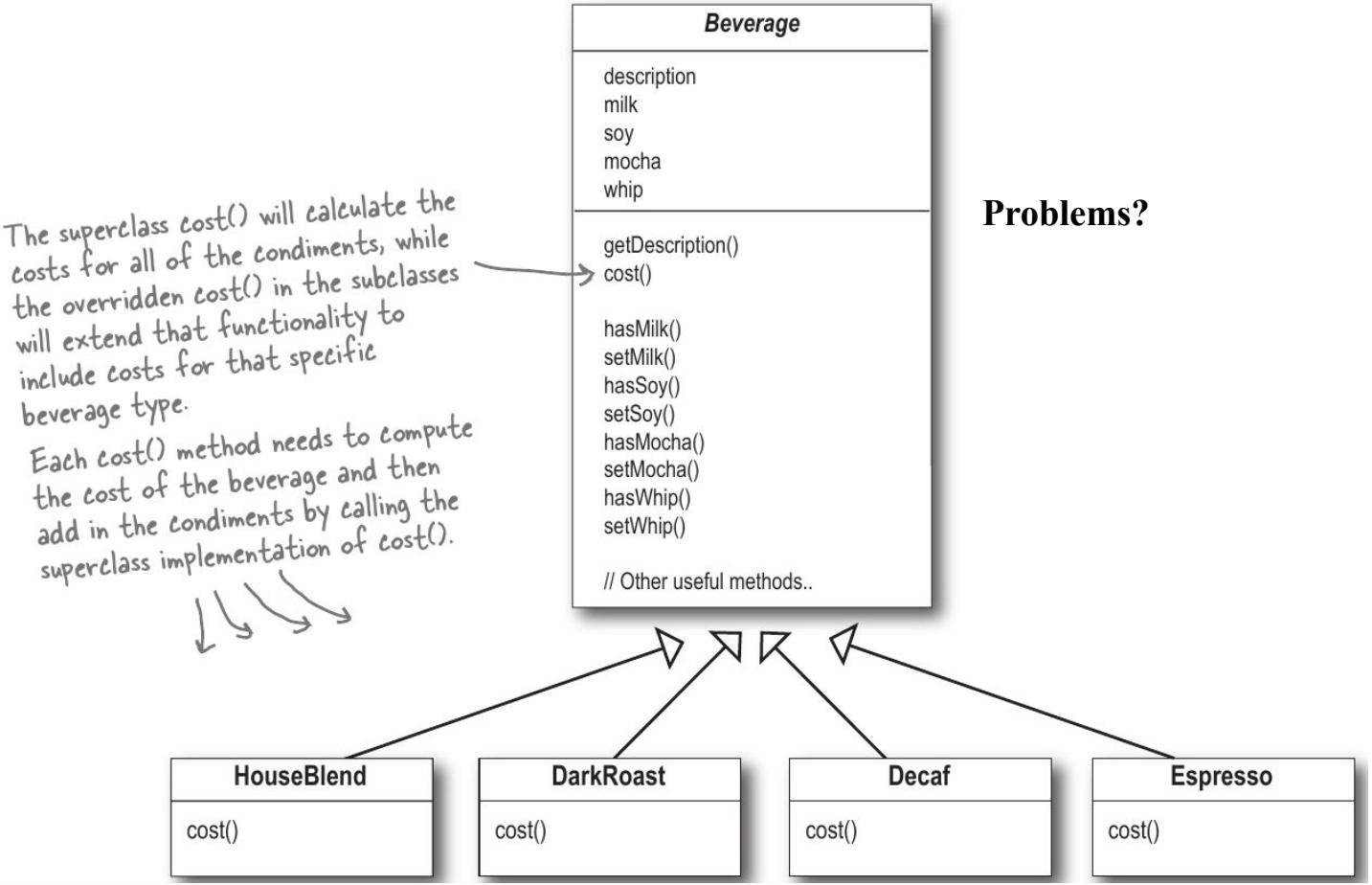
The superclass cost() will calculate the costs for all of the condiments, while the overridden cost() in the subclasses will extend that functionality to include costs for that specific beverage type.

Each cost() method needs to compute the cost of the beverage and then add in the condiments by calling the superclass implementation of cost().



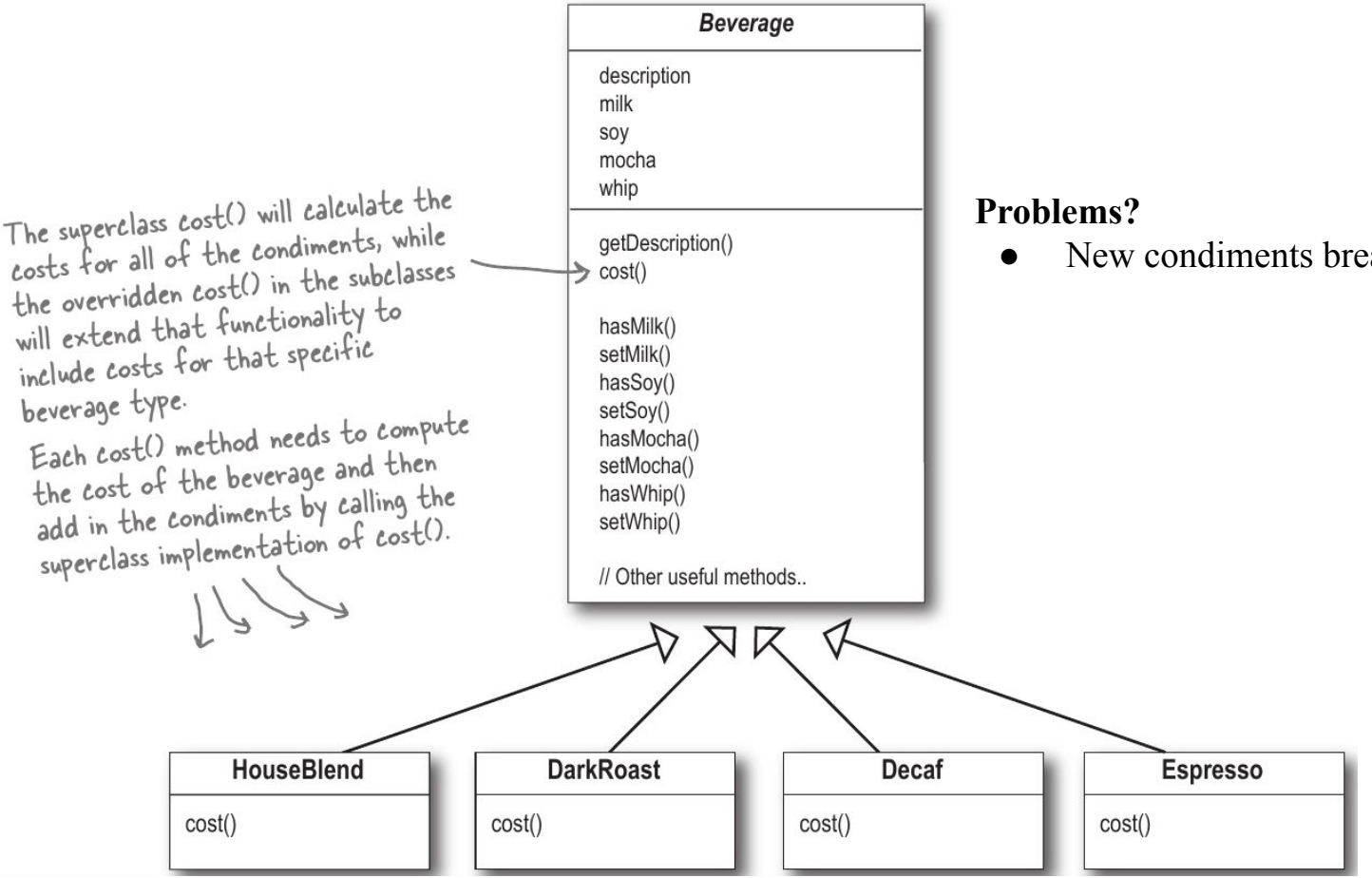

```
public class DarkRoast extends Beverage {

    public DarkRoast() {
        description = "Most Excellent Dark Roast";
    }

    public float cost() {

        return 1.99 + super.cost();


    }
}
```

# The design using Booleans




**Problems?**

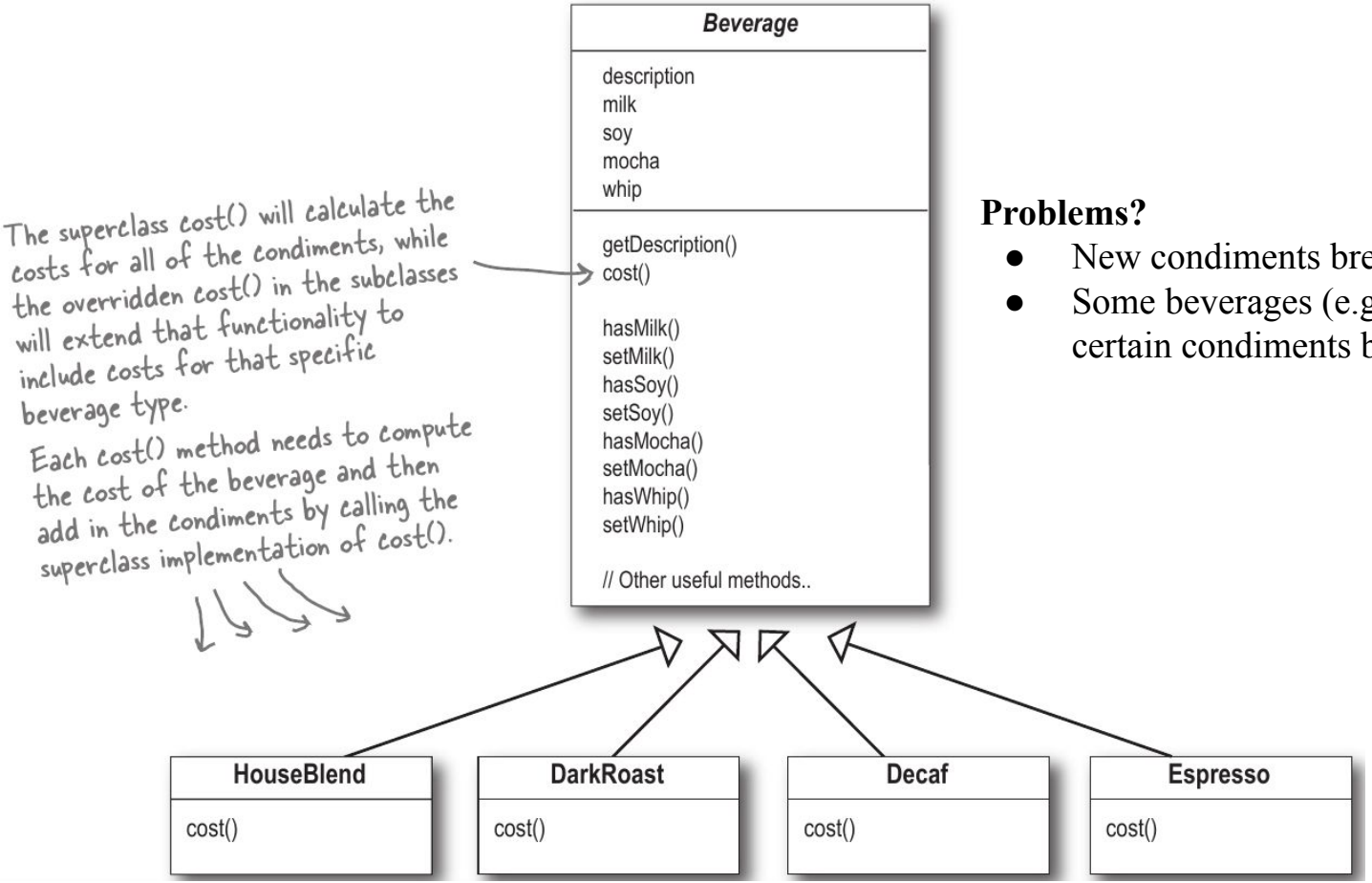
# The design using Booleans




**Problems?**

- New condiments break the *superclass design*.

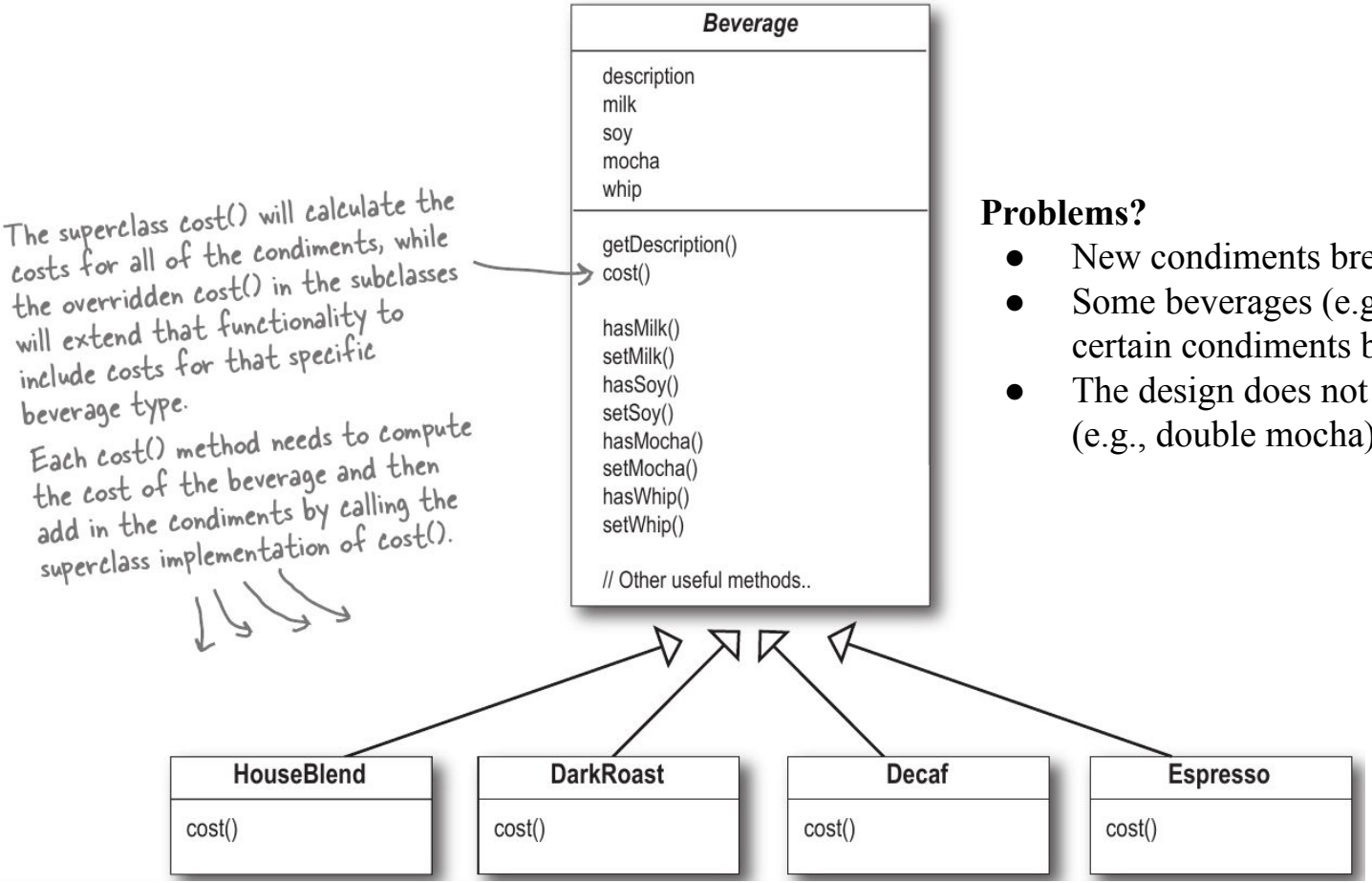
# The design using Booleans




**Problems?**

- New condiments break the *superclass design*.
- Some beverages (e.g.iced tea) may not support certain condiments but still inherit them. (BAD)

# The design using Booleans




**Problems?**

- New condiments break the *superclass design*.
- Some beverages (e.g.iced tea) may not support certain condiments but still inherit them. (BAD)
- The design does not handle multiple quantities (e.g., double mocha).

# Let's do it like "Decoration"

- Inheritance didn't work well.
- Instead we start with a beverage and **"decorate"** it with condiments

# Let's do it like "Decoration"

- Inheritance didn't work well.
- Instead we start with a beverage and **"decorate"** it with condiments

For DarkRoast with Mocha and Whip

1. **Take a DarkRoast object**
2. **Decorate it with a Mocha object**
3. **Decorate it with a Whip object**
4. **Call the cost() method and rely on delegation to add on the condiment costs**

# DarkRoast with Mocha and Whip

**1 We start with our DarkRoast object.**

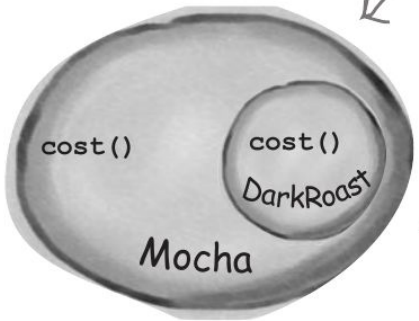

Remember that DarkRoast inherits from Beverage and has a cost() method that computes the cost of the drink.

# DarkRoast with Mocha and Whip

**1 We start with our DarkRoast object.**


Remember that DarkRoast inherits from Beverage and has a cost() method that computes the cost of the drink.

**2 The customer wants Mocha, so we create a Mocha object and wrap it around the DarkRoast.**

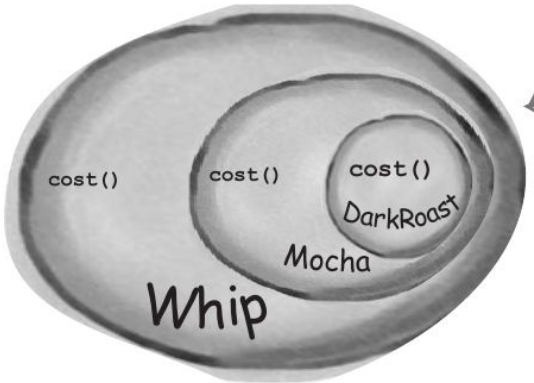



The Mocha object is a decorator. Its type mirrors the object it is decorating, in this case, a Beverage. (By "mirror", we mean it is the same type..)

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).

# DarkRoast with Mocha and Whip

**❸ The customer also wants Whip, so we create a Whip decorator and wrap Mocha with it.**

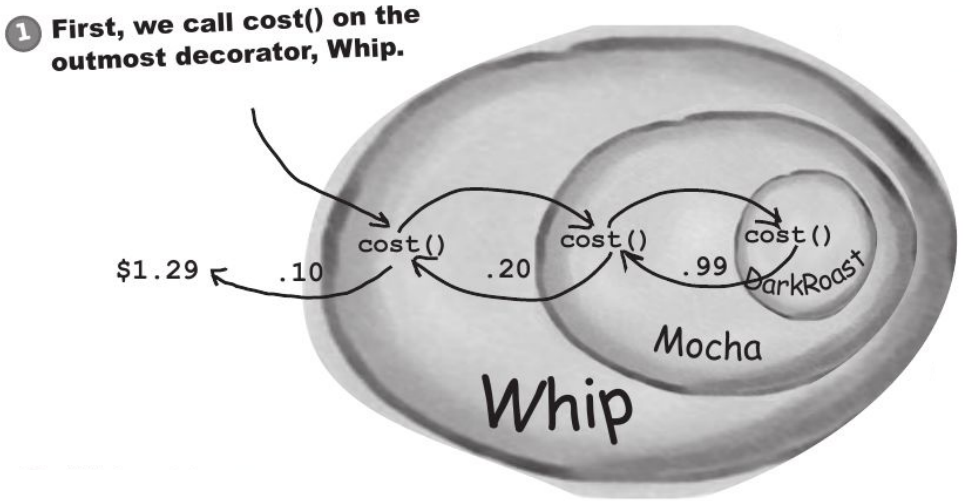



Whip is a decorator, so it also mirrors DarkRoast's type and includes a cost() method.

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.

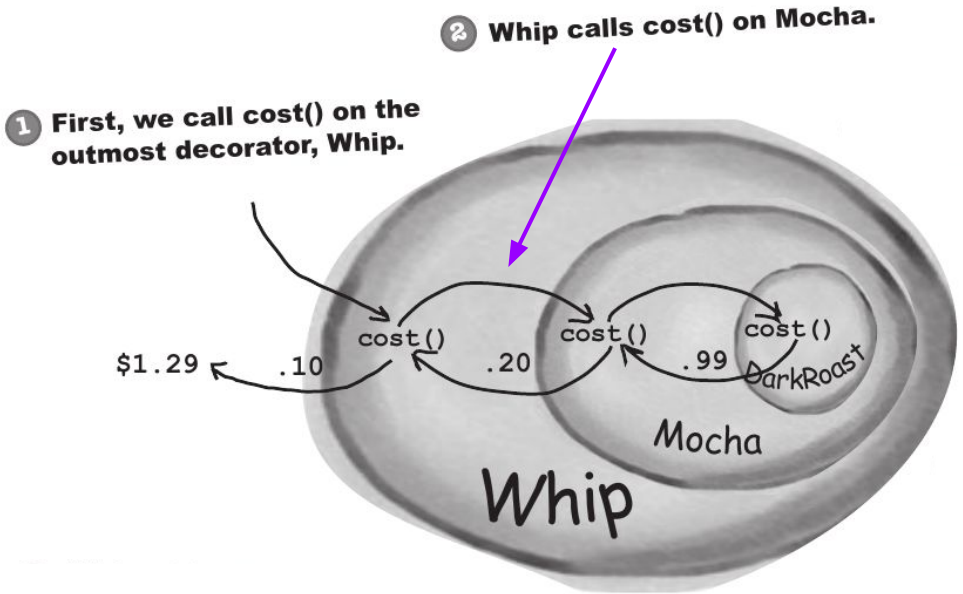
# DarkRoast with Mocha and Whip

4. **Now it's time to compute the cost for the customer. We do this by calling cost() on the outermost decorator, Whip, and Whip is going to delegate computing the cost to the objects it decorates. Once it gets a cost, it will add on the cost of the Whip.**



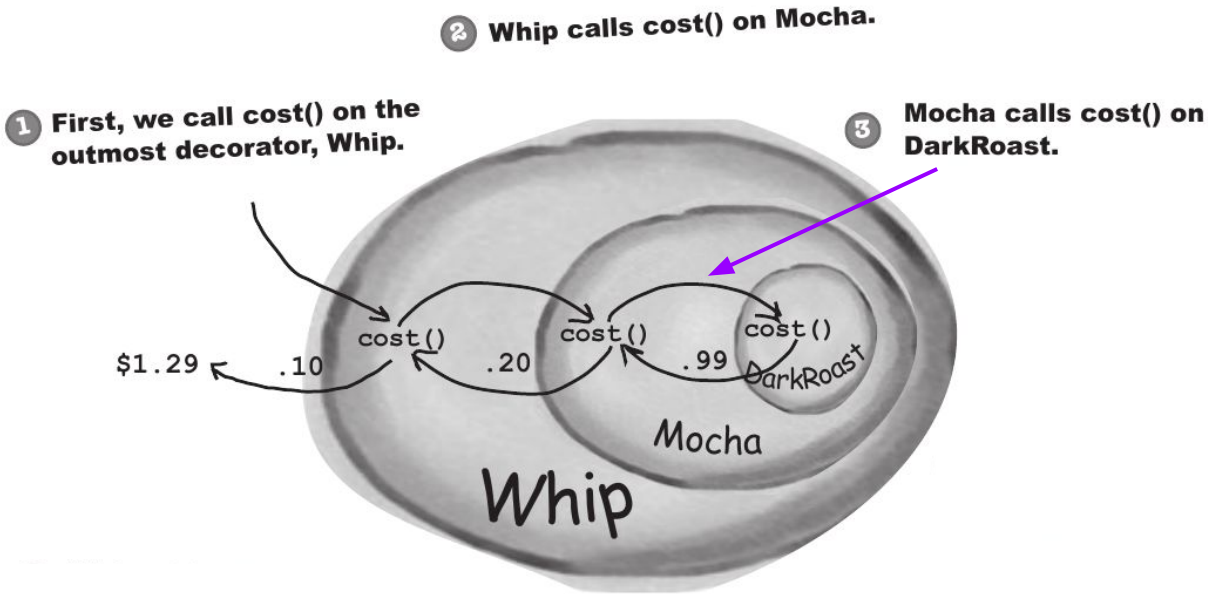
# DarkRoast with Mocha and Whip

4. **Now it's time to compute the cost for the customer. We do this by calling cost() on the outermost decorator, Whip, and Whip is going to delegate computing the cost to the objects it decorates. Once it gets a cost, it will add on the cost of the Whip.**



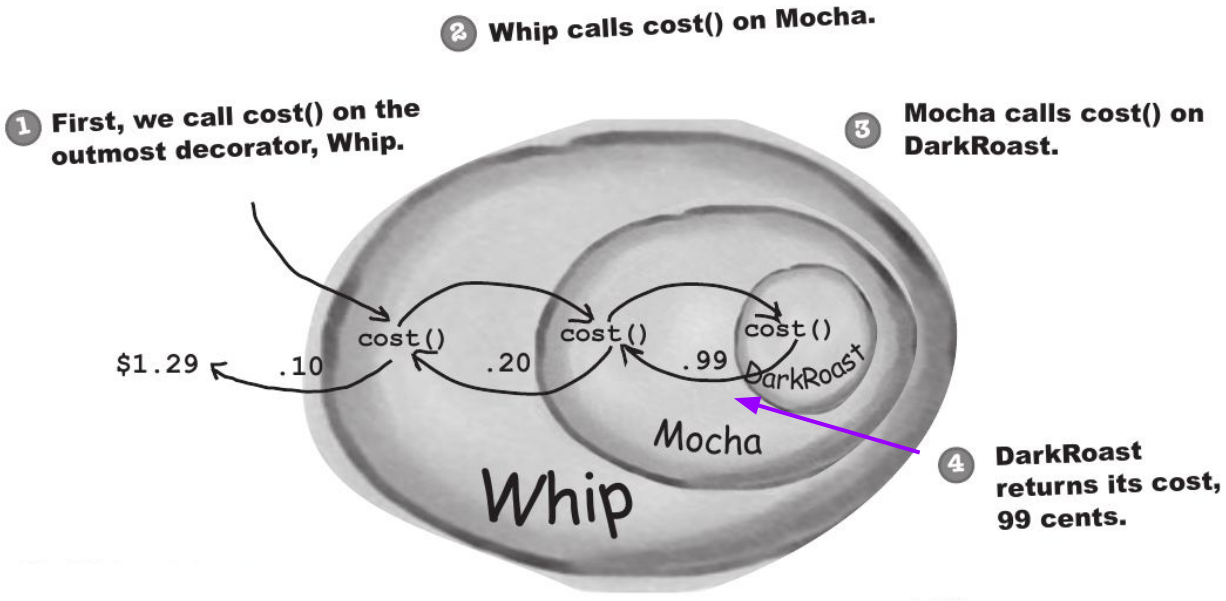
# DarkRoast with Mocha and Whip

4. **Now it's time to compute the cost for the customer. We do this by calling cost() on the outermost decorator, Whip, and Whip is going to delegate computing the cost to the objects it decorates. Once it gets a cost, it will add on the cost of the Whip.**



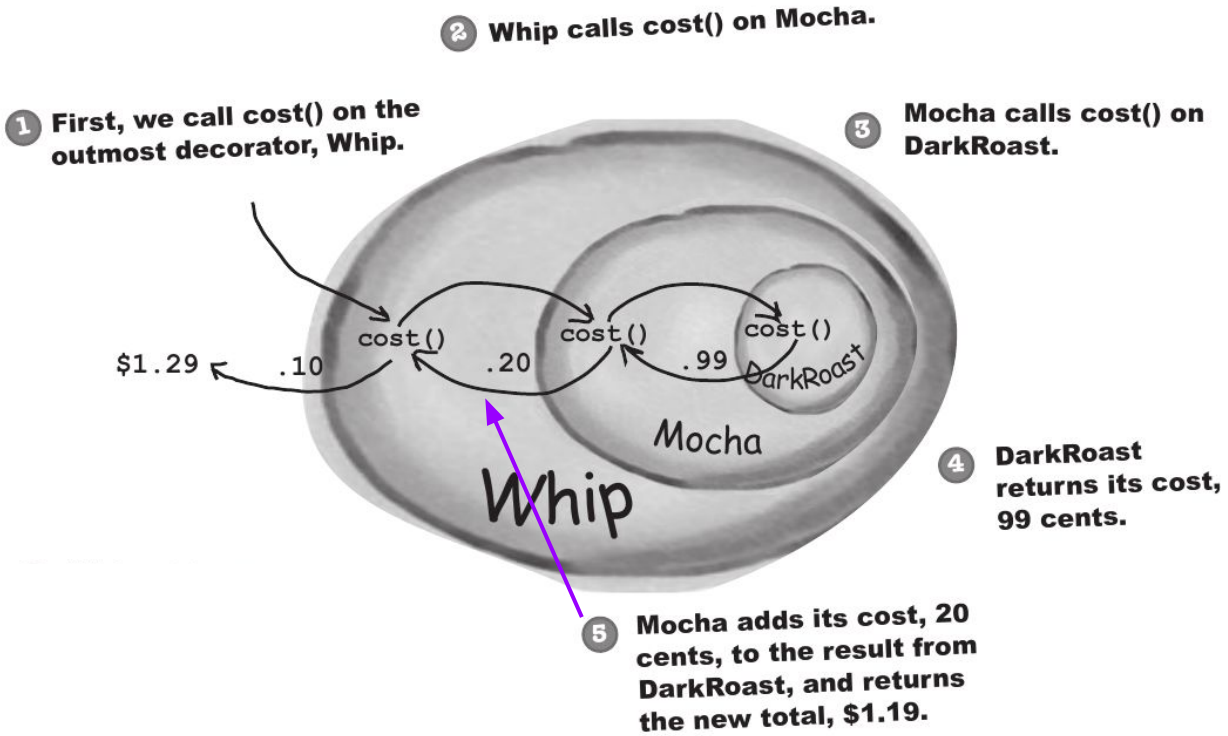
# DarkRoast with Mocha and Whip

4. **Now it's time to compute the cost for the customer. We do this by calling cost() on the outermost decorator, Whip, and Whip is going to delegate computing the cost to the objects it decorates. Once it gets a cost, it will add on the cost of the Whip.**



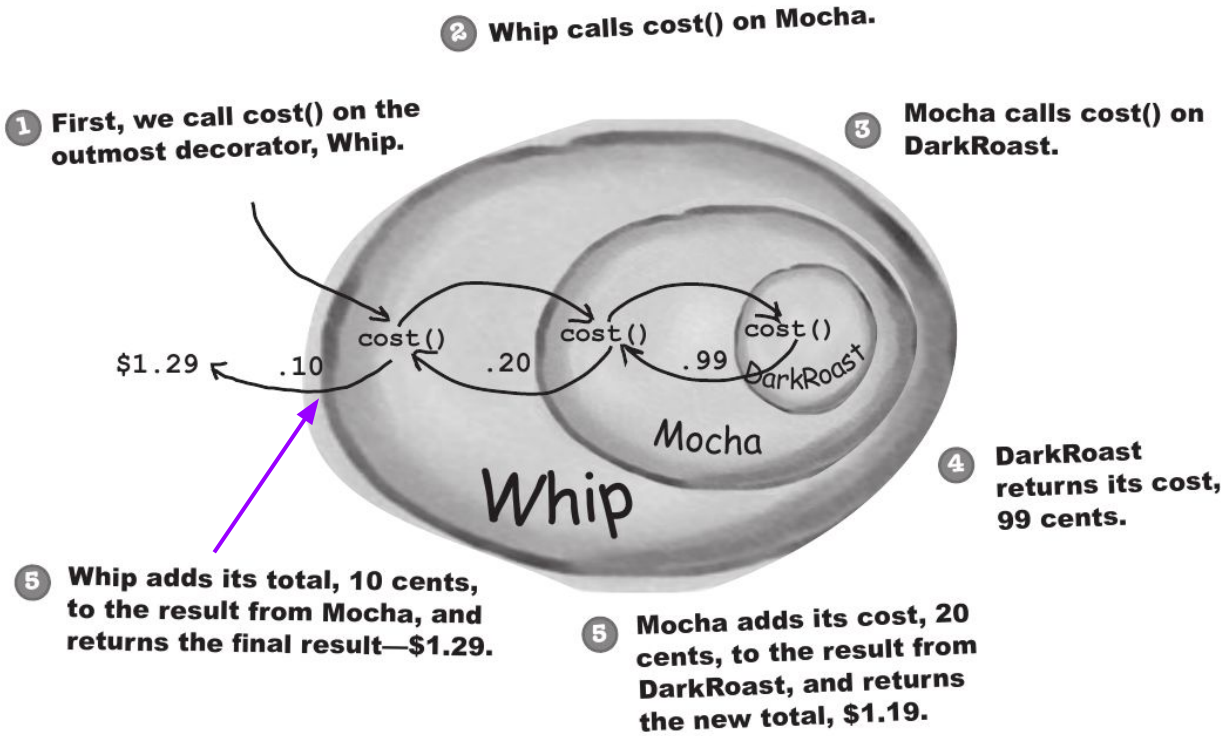
# DarkRoast with Mocha and Whip

4. **Now it's time to compute the cost for the customer. We do this by calling cost() on the outermost decorator, Whip, and Whip is going to delegate computing the cost to the objects it decorates. Once it gets a cost, it will add on the cost of the Whip.**

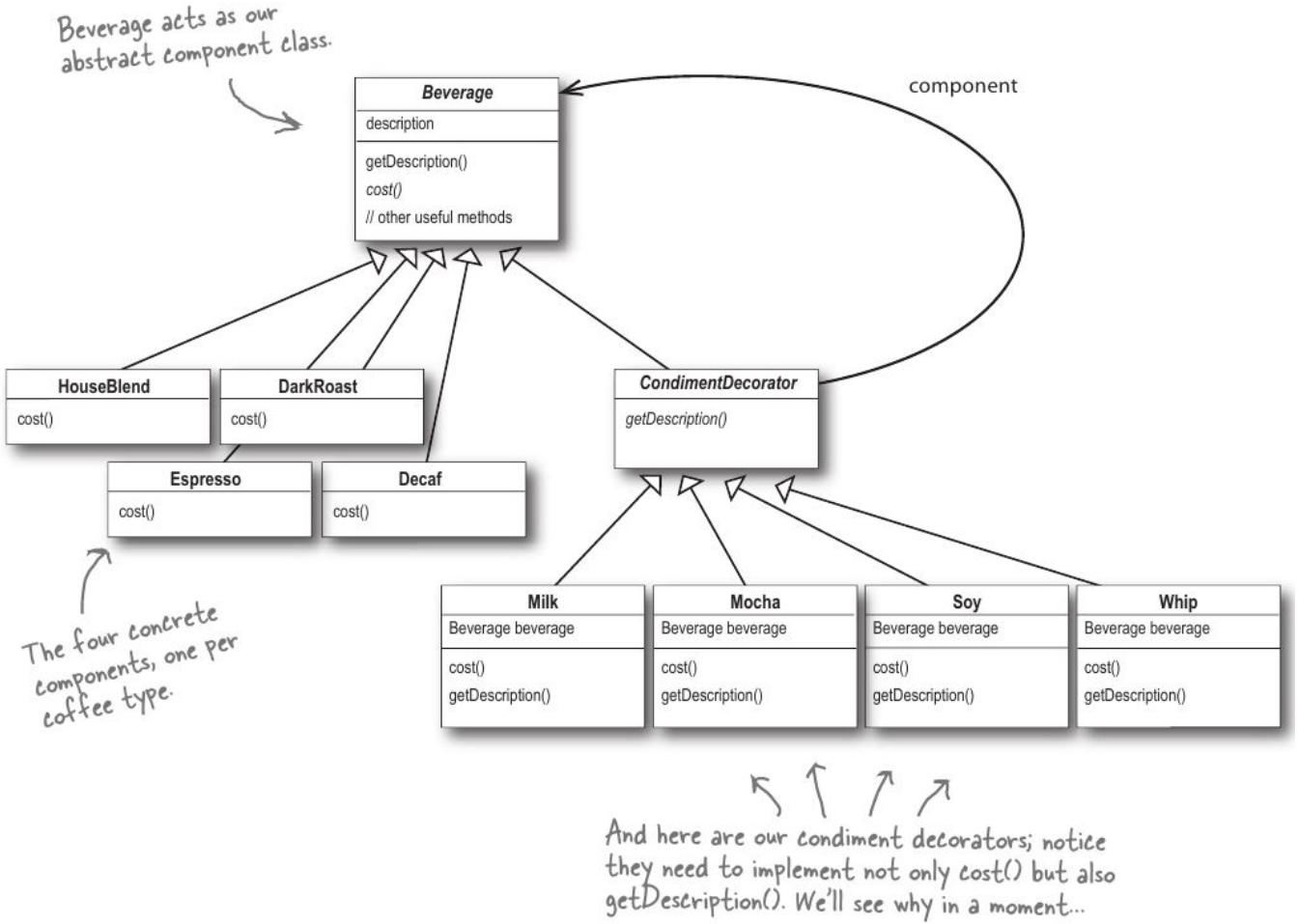


# DarkRoast with Mocha and Whip

4. **Now it's time to compute the cost for the customer. We do this by calling cost() on the outermost decorator, Whip, and Whip is going to delegate computing the cost to the objects it decorates. Once it gets a cost, it will add on the cost of the Whip.**

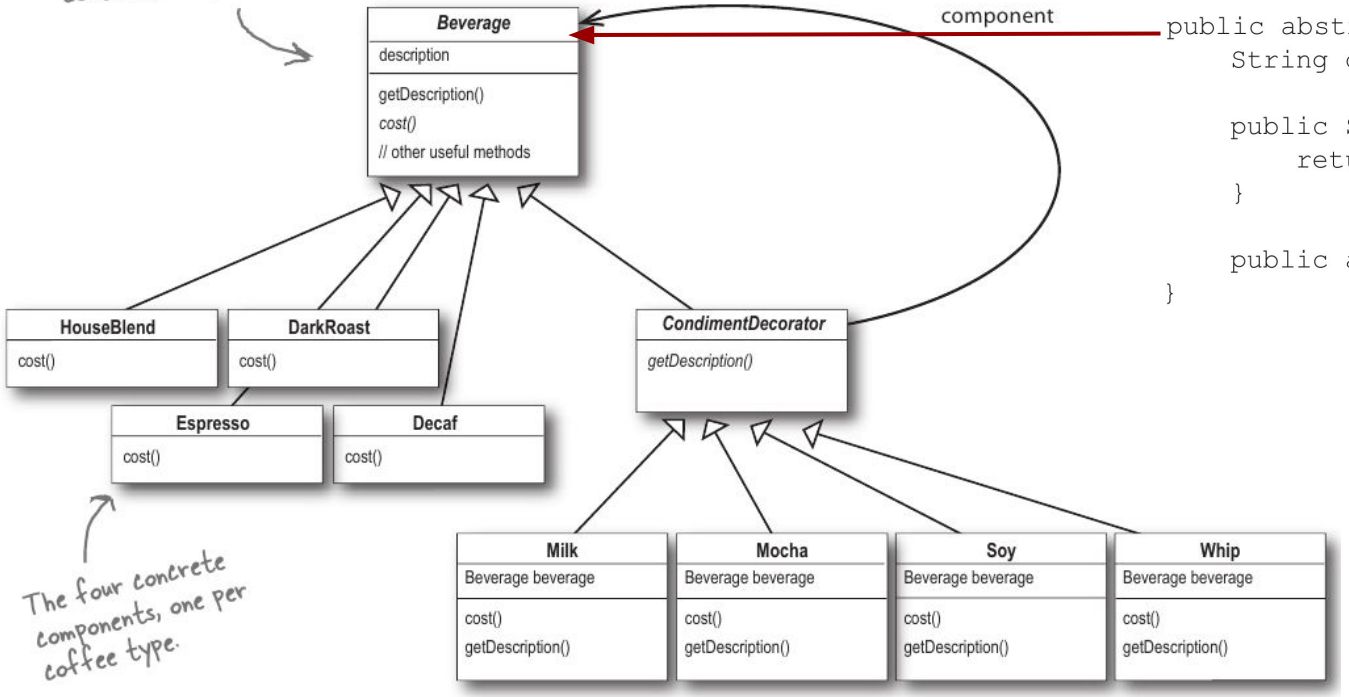


# The Design


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()
The four concrete components, one per coffee type.
Milk
Beverage beverage
cost()
getDescription()
Mocha
Beverage beverage
cost()
getDescription()
Soy
Beverage beverage
cost()
getDescription()
Whip
Beverage beverage
cost()
getDescription()
And here are our condiment decorators; notice they need to implement not only cost() but also getDescription(). We'll see why in a moment...

# The Design

Beverage acts as our abstract component class.



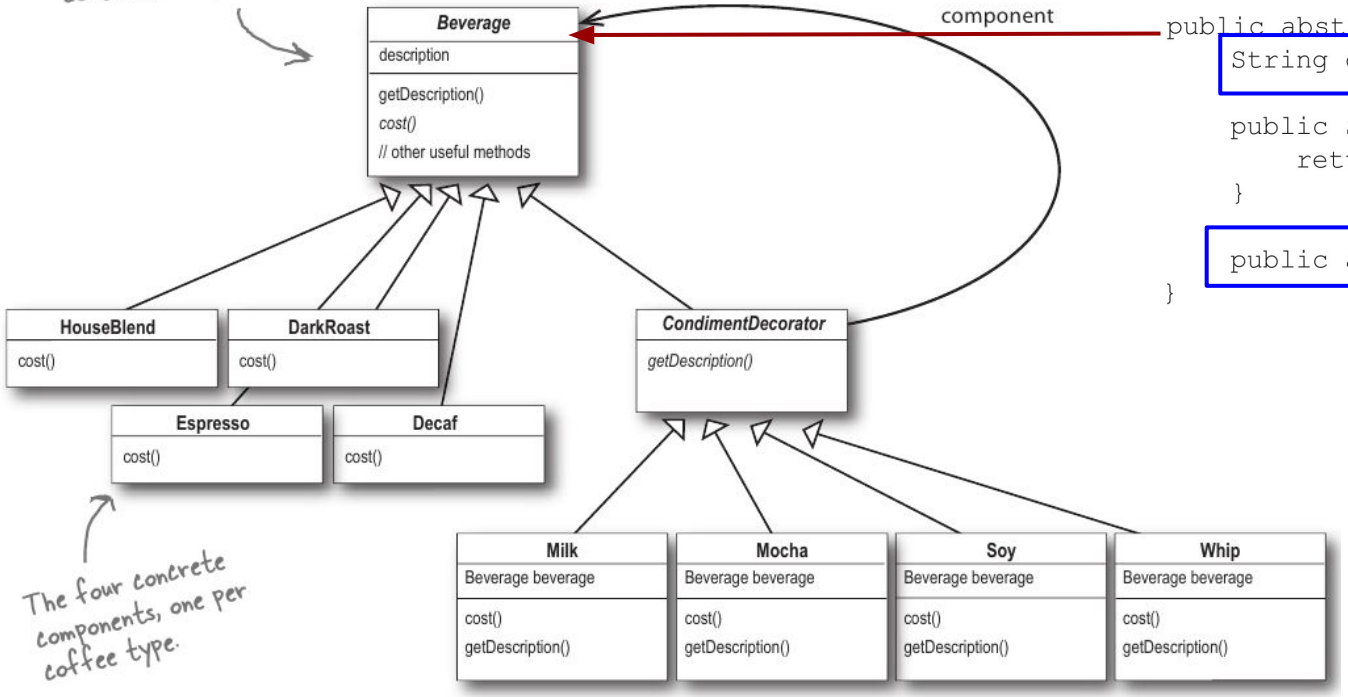

```
public abstract class Beverage {
    String description = "Unknown Beverage";

    public String getDescription() {
        return description;
    }

    public abstract double cost();
}
```

The four concrete components, one per coffee type.

And here are our condiment decorators; notice they need to implement not only cost() but also getDescription(). We'll see why in a moment...

# The Design

Beverage acts as our abstract component class.




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

    public String getDescription() {
        return description;
    }

    public abstract double cost();
}
```

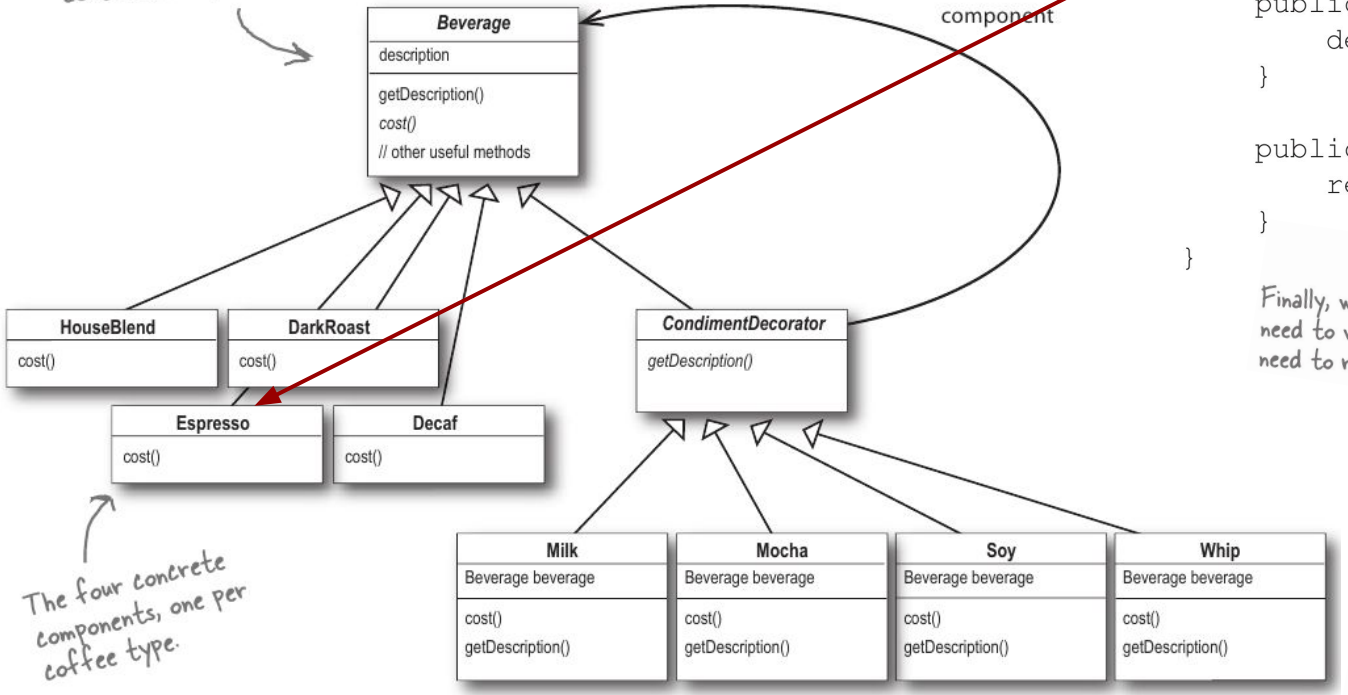
subclasses handle

The four concrete components, one per coffee type.

And here are our condiment decorators; notice they need to implement not only cost() but also getDescription(). We'll see why in a moment...

# The Design

Beverage acts as our abstract component class.




The four concrete components, one per coffee type.

And here are our condiment decorators; notice they need to implement not only cost() but also getDescription(). We'll see why in a moment...

```
public class Espresso extends Beverage {

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

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

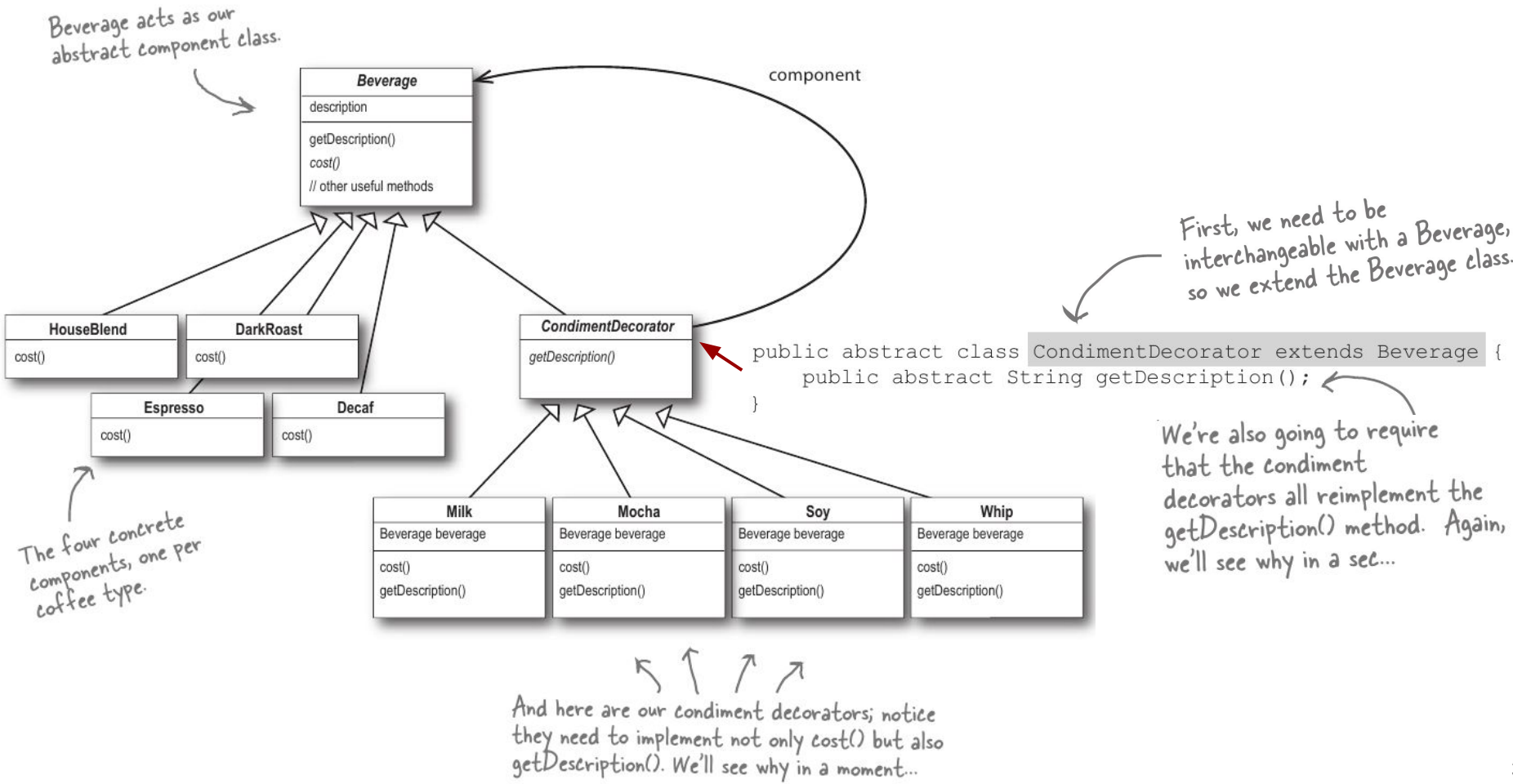
set description

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.

**Starbuzz Coffee**

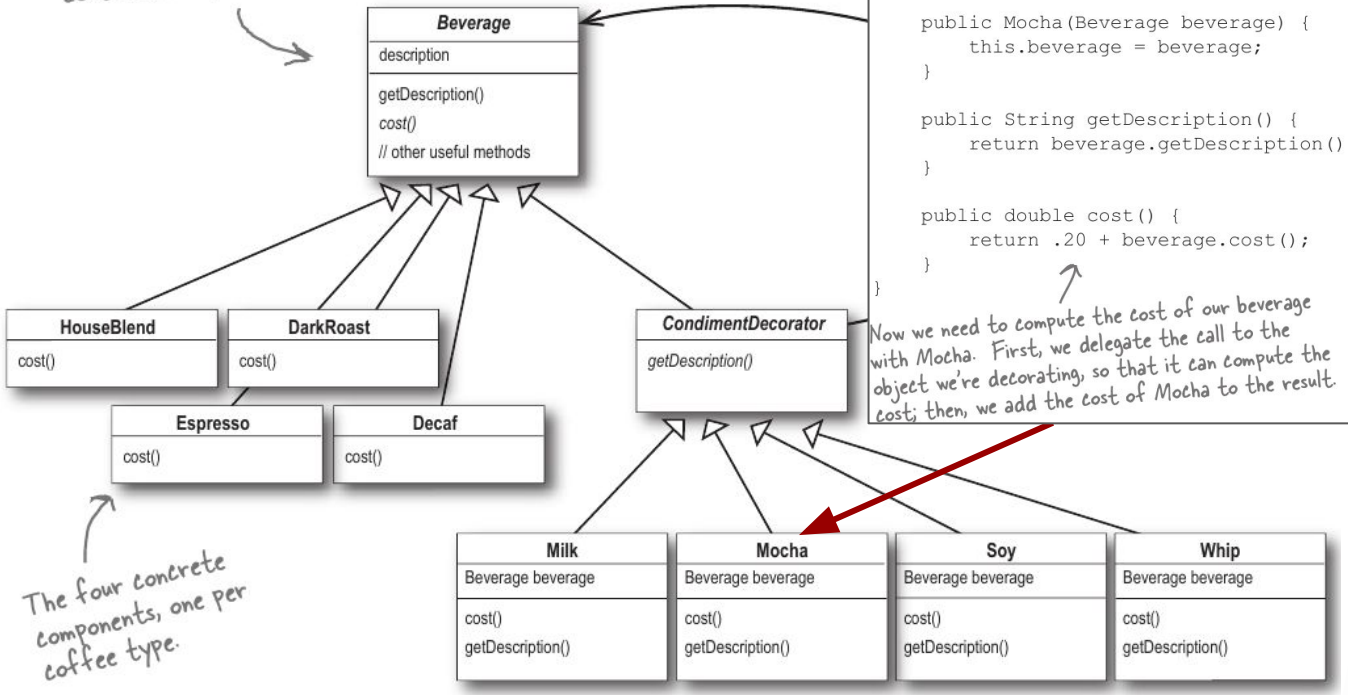
| Coffees | |
|---|---|
| House Blend | .89 |
| Dark Roast | .99 |
| Decaf | 1.05 |
| Espresso | 1.99 |
| **Condiments** | |
| Steamed Milk | .10 |
| Mocha | .20 |
| Soy | .15 |
| Whip | .10 |

# The Design


Beverage acts as our abstract component class.
Beverage
description
getDescription()
cost()
// other useful methods
component
HouseBlend
cost()
DarkRoast
cost()
CondimentDecorator
getDescription()
Espresso
cost()
Decaf
cost()
The four concrete components, one per coffee type.
Milk
Beverage beverage
cost()
getDescription()
Mocha
Beverage beverage
cost()
getDescription()
Soy
Beverage beverage
cost()
getDescription()
Whip
Beverage beverage
cost()
getDescription()
And here are our condiment decorators; notice they need to implement not only cost() but also getDescription(). We'll see why in a moment...
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...

# The Design

Beverage acts as our abstract component class.




The four concrete components, one per coffee type.

```
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 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.

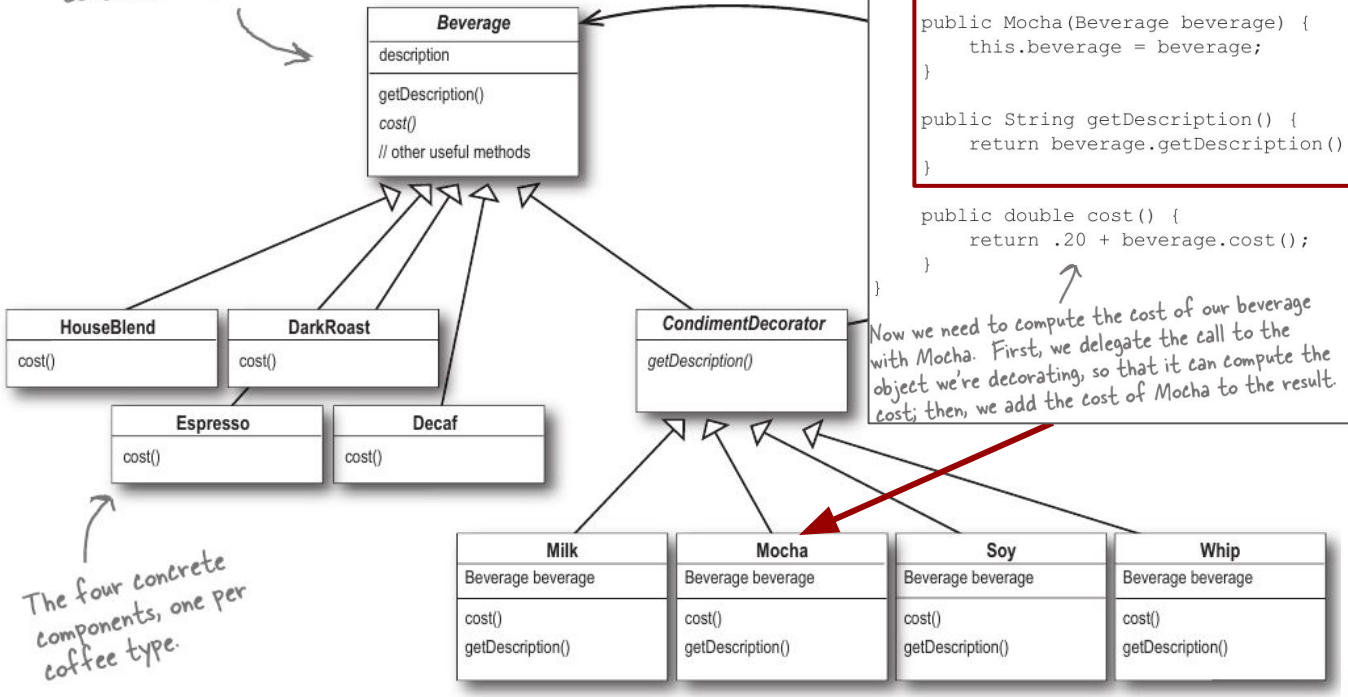
And here are our condiment decorators; notice they need to implement not only cost() but also getDescription(). We'll see why in a moment...

**Starbuzz Coffee**

| Coffees | |
|---|---|
| House Blend | .89 |
| Dark Roast | .99 |
| Decaf | 1.05 |
| Espresso | 1.99 |
| **Condiments** | |
| Steamed Milk | .10 |
| Mocha | .20 |
| Soy | .15 |
| Whip | .10 |

# The Design

Beverage acts as our abstract component class.




The four concrete components, one per coffee type.

```
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 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.

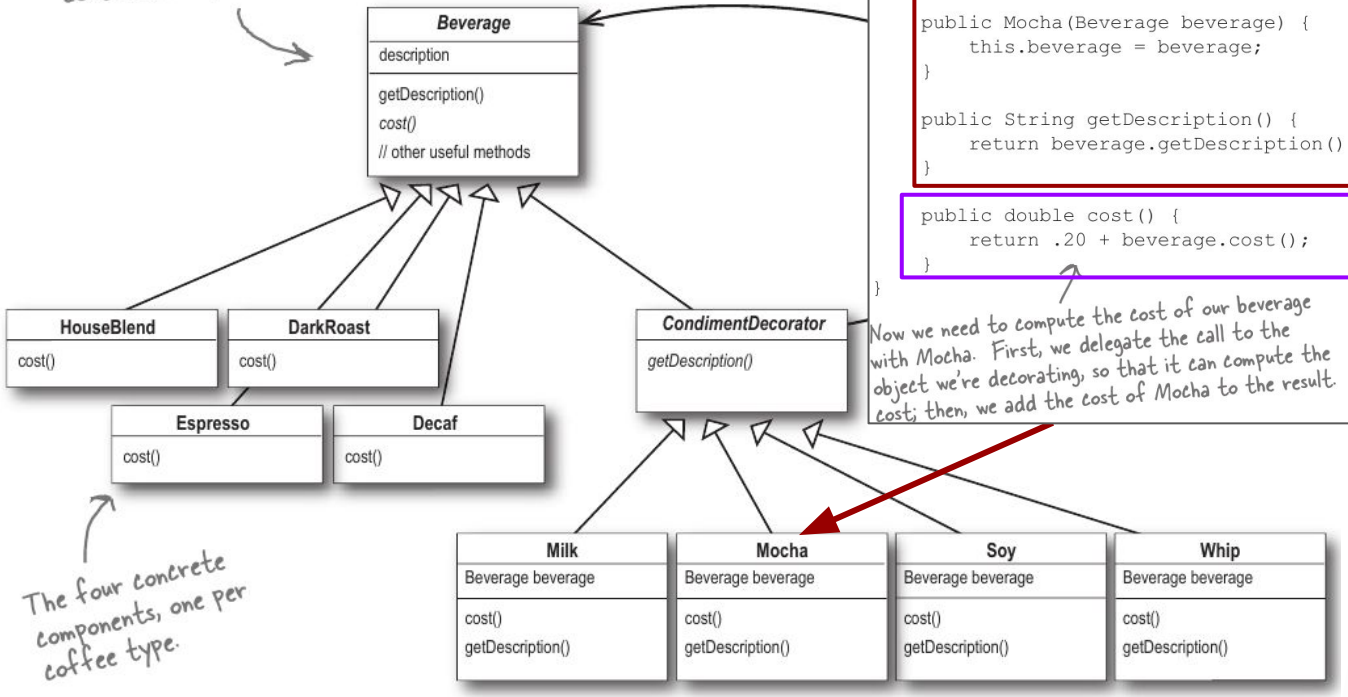
And here are our condiment decorators; notice they need to implement not only cost() but also getDescription(). We'll see why in a moment...

**Starbuzz Coffee**

| Coffees | |
|---|---|
| House Blend | .89 |
| Dark Roast | .99 |
| Decaf | 1.05 |
| Espresso | 1.99 |
| **Condiments** | |
| Steamed Milk | .10 |
| Mocha | .20 |
| Soy | .15 |
| Whip | .10 |

# The Design

Beverage acts as our abstract component class.




The four concrete components, one per coffee type.

```
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 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.

And here are our condiment decorators; notice they need to implement not only cost() but also getDescription(). We'll see why in a moment...

**Starbuzz Coffee**

| Coffees | |
|---|---|
| House Blend | .89 |
| Dark Roast | .99 |
| Decaf | 1.05 |
| Espresso | 1.99 |
| **Condiments** | |
| Steamed Milk | .10 |
| Mocha | .20 |
| Soy | .15 |
| Whip | .10 |

# Test Drive

```
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.

DarkRoast

```
public class Mocha extends CondimentDecorator {
    Beverage beverage;

    public Mocha(Beverage beverage) {
        this.beverage = beverage;
    }
```

# Test Drive

```
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.


```
public class Mocha extends CondimentDecorator {
    Beverage beverage;

    public Mocha(Beverage beverage) {
        this.beverage = beverage;
    }
```

# Test Drive

```
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.

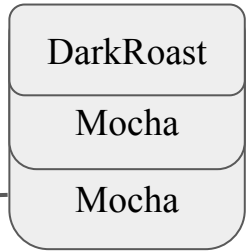
Make a DarkRoast object.

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.




```
public class Mocha extends CondimentDecorator {
    Beverage beverage;

    public Mocha(Beverage beverage) {
        this.beverage = beverage;
    }
```

# Test Drive

```
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.


```
public class Mocha extends CondimentDecorator {
    Beverage beverage;

    public Mocha(Beverage beverage) {
        this.beverage = beverage;
    }
```

# Test Drive

```
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.


```
% java StarbuzzCoffee
Espresso $1.99
Dark Roast Coffee, Mocha, Mocha, Whip $1.49
House Blend Coffee, Soy, Mocha, Whip $1.34
```

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

public double cost() {
    return .20 + beverage.cost();
}
```

# Test Drive

```
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.

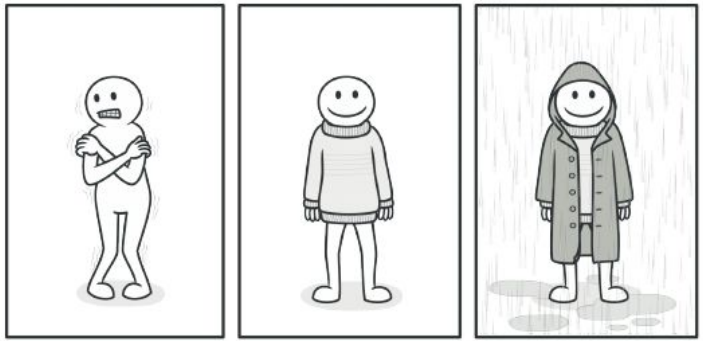
Wrap it with a Mocha.

Wrap it in a second Mocha.

Wrap it in a Whip.

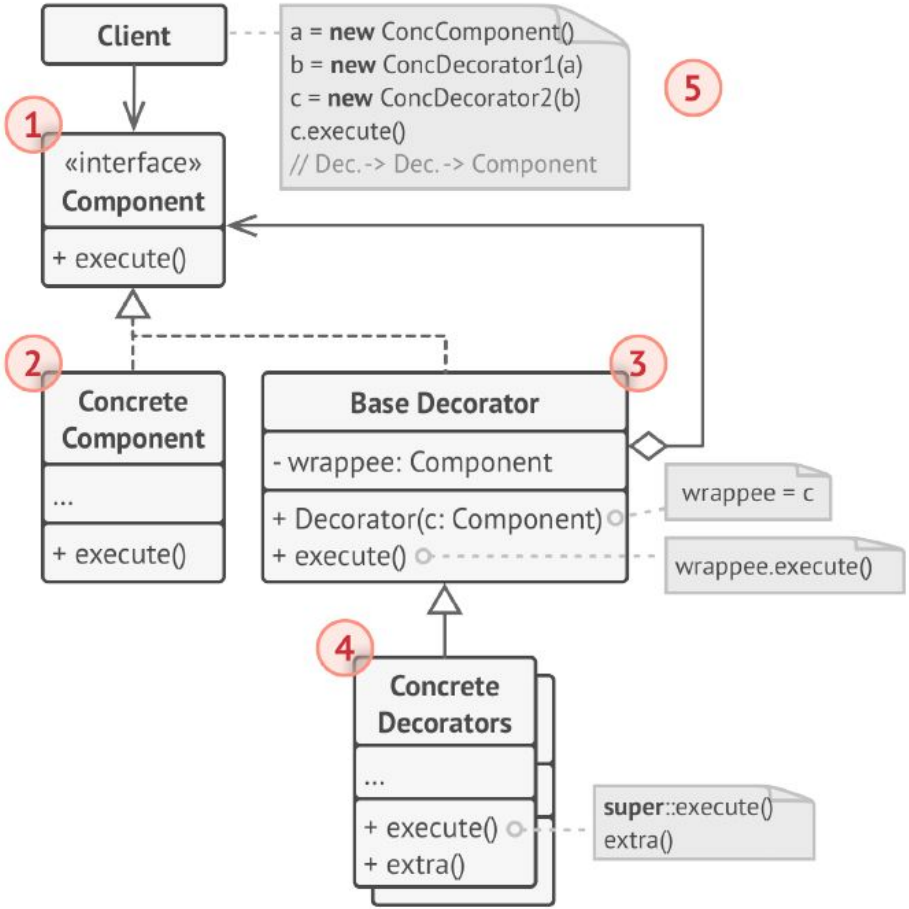
Finally, give us a HouseBlend with Soy, Mocha, and Whip.


```
% java StarbuzzCoffee
Espresso $1.99
Dark Roast Coffee, Mocha, Mocha, Whip $1.49
House Blend Coffee, Soy, Mocha, Whip $1.34
```



# Decorator Pattern Defined

Decorator lets you attach new behaviors to objects by placing **these objects** inside special wrapper objects that contain the *behaviors*.



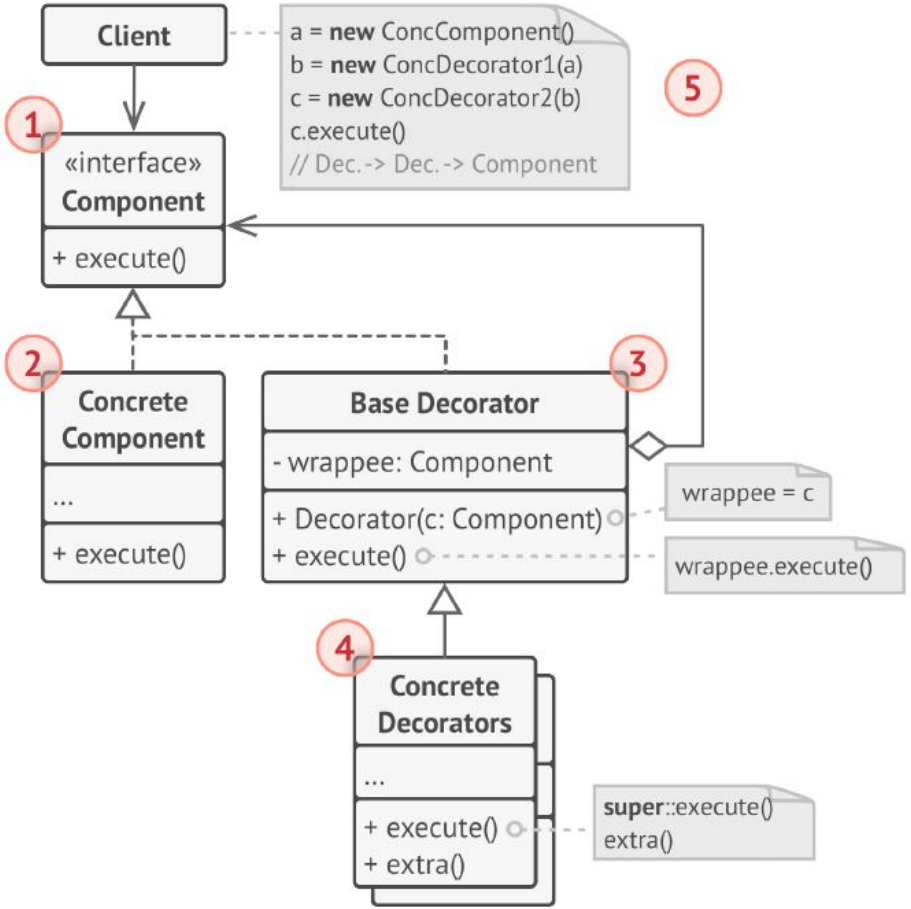
# Decorator Pattern Defined

1. The **Component** declares the common interface for both wrappers and wrapped objects.
2. **Concrete Component** is a class of objects being wrapped. It defines the basic behavior, which can be altered by decorators.

3. The **Base Decorator** class has a field for referencing a wrapped object. The field's type should be declared as the component interface so it can contain both concrete components and decorators. The base decorator delegates all operations to the wrapped object.

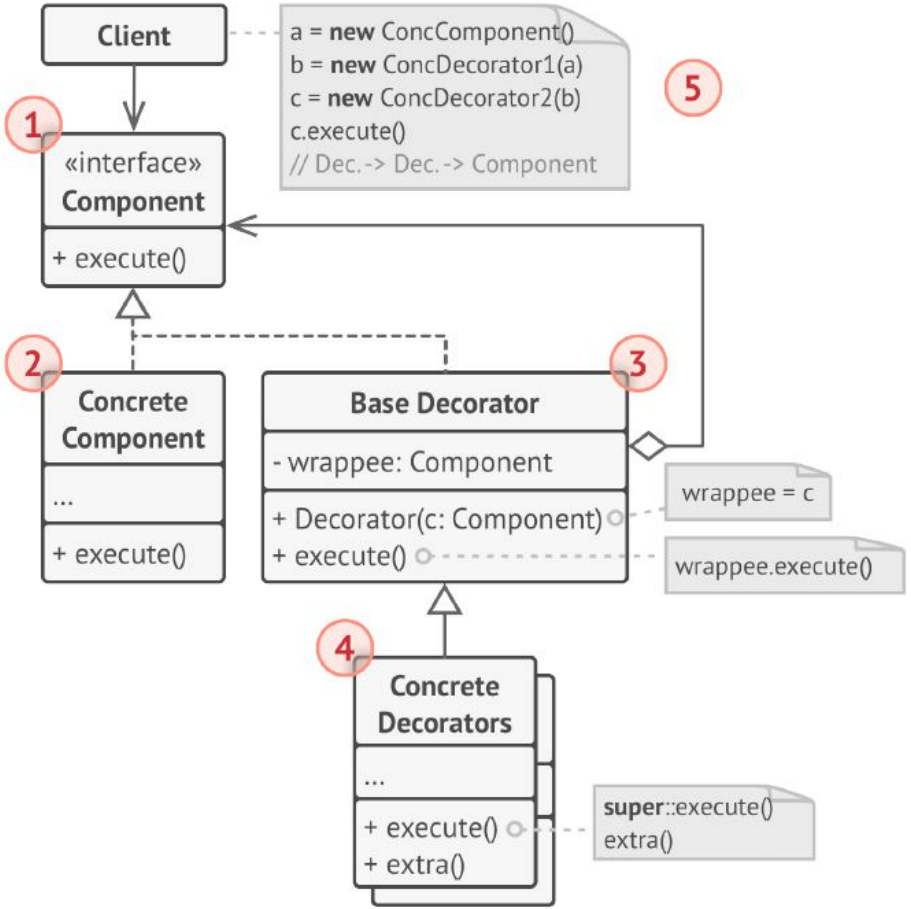
4. **Concrete Decorators** define extra behaviors that can be added to components dynamically. Concrete decorators override methods of the base decorator and execute their behavior either before or after calling the parent method.

5. The **Client** can wrap components in multiple layers of decorators, as long as it works with all objects via the component interface.



# Decorator Pattern Defined

1. The **Component** declares the common interface for both wrappers and wrapped objects.
2. **Concrete Component** is a class of objects being wrapped. It defines the basic behavior, which can be altered by decorators.
3. The **Base Decorator** class has a field for referencing a wrapped object. The field's type should be declared as the component interface so it can contain both concrete components and decorators. The base decorator delegates all operations to the wrapped object.
4. **Concrete Decorators** define extra behaviors that can be added to components dynamically. Concrete decorators override methods of the base decorator and execute their behavior either before or after calling the parent method.
5. The **Client** can wrap components in multiple layers of decorators, as long as it works with all objects via the component interface.



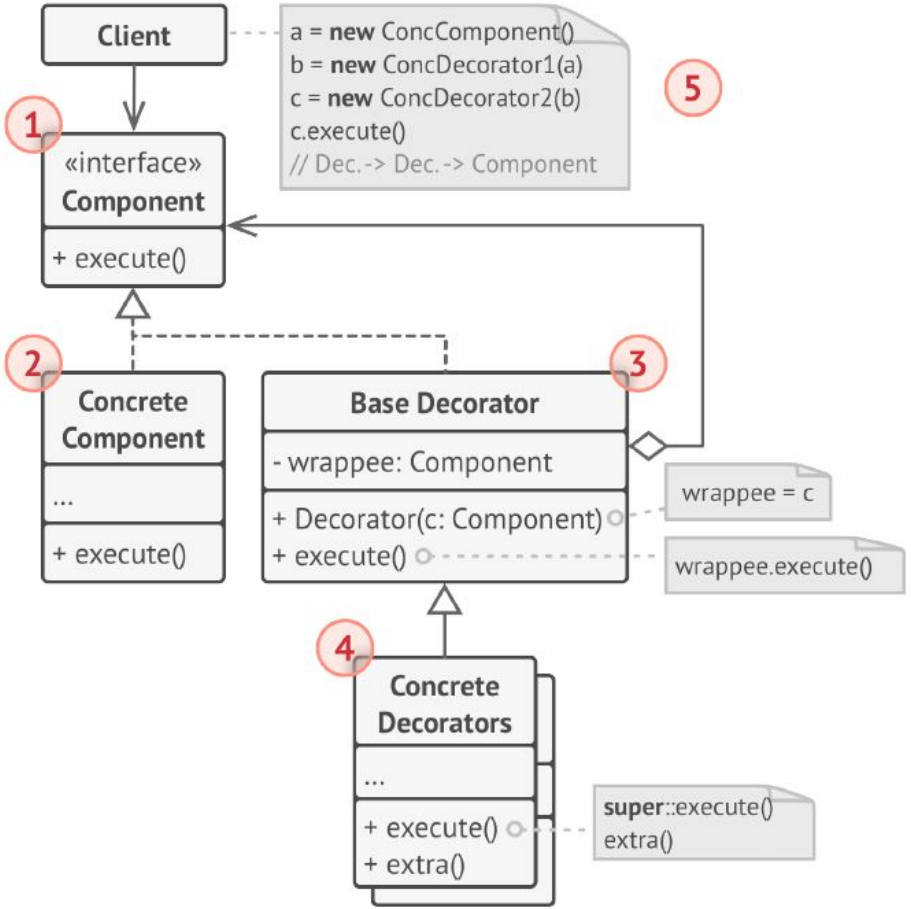
# Decorator Pattern Defined

1. The **Component** declares the common interface for both wrappers and wrapped objects.
2. **Concrete Component** is a class of objects being wrapped. It defines the basic behavior, which can be altered by decorators.

3. The **Base Decorator** class has a field for referencing a wrapped object. The field's type should be declared as the component interface so it can contain both concrete components and decorators. The base decorator delegates all operations to the wrapped object.

4. **Concrete Decorators** define extra behaviors that can be added to components dynamically. Concrete decorators override methods of the base decorator and execute their behavior either before or after calling the parent method.

5. The **Client** can wrap components in multiple layers of decorators, as long as it works with all objects via the component interface.



# Let's talk about Pancake House and Diner

The Diner menu has lots of lunch items, while the Pancake House consists of breakfast items. Every menu item has a name, a description, and a price

Objectville Diner

**Vegetarian BLT**
(Fakin') Bacon with lettu... 2.99
whole wheat
**BLT**
Bacon with lettuce & tom...
**Soup of the day**
A bowl of the soup of the...
a side of potato salad
**Hot Dog**
A hot dog, with saurkrau...
topped with cheese
**Steamed Veggies and Bro...**
A medley of steamed veg...

Objectville Pancake House

**K&B's Pancake Breakfast** 2.99
Pancakes with scrambled eggs, and toast

**Regular Pancake Breakfast** 2.99
Pancakes with fried eggs, sausage

**Blueberry Pancakes** 3.49
Pancakes made with fresh blueberries, and blueberry syrup

**Waffles** 3.59
Waffles, with your choice of blueberries or strawberries

# Let's talk about Pancake House and Diner

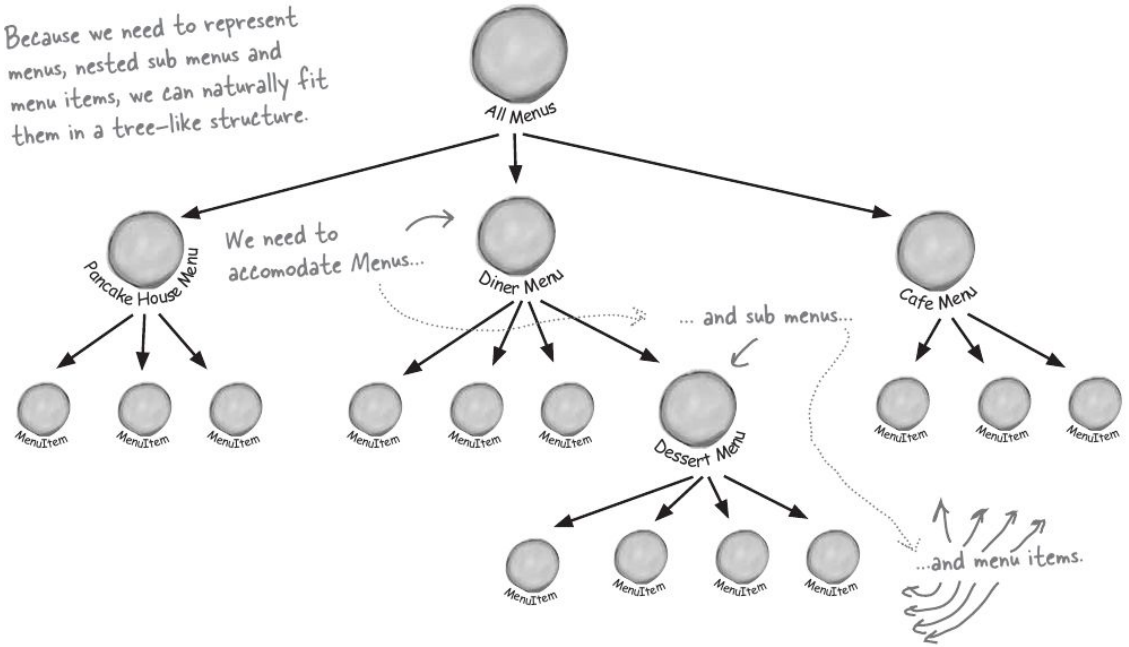
**Java-Enabled Waitress: code-name "Alice"**

```
printMenu()
   - prints every item on the menu

printBreakfastMenu()
   - prints just breakfast items

printLunchMenu()
   - prints just lunch items

printVegetarianMenu()
   - prints all vegetarian menu items

isItemVegetarian(name)
   - given the name of an item, returns true
     if the items is vegetarian, otherwise,
     returns false
```

The Diner menu has lots of lunch items, while the Pancake House consists of breakfast items. Every menu item has a name, a description, and a price

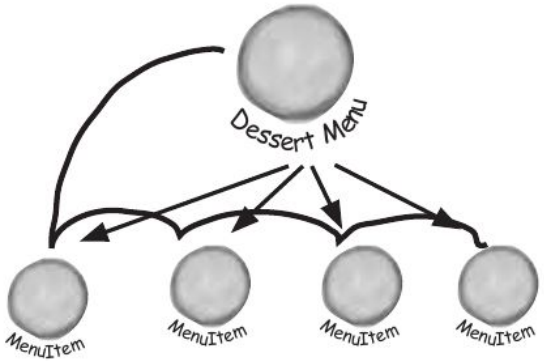
# Something tree-like

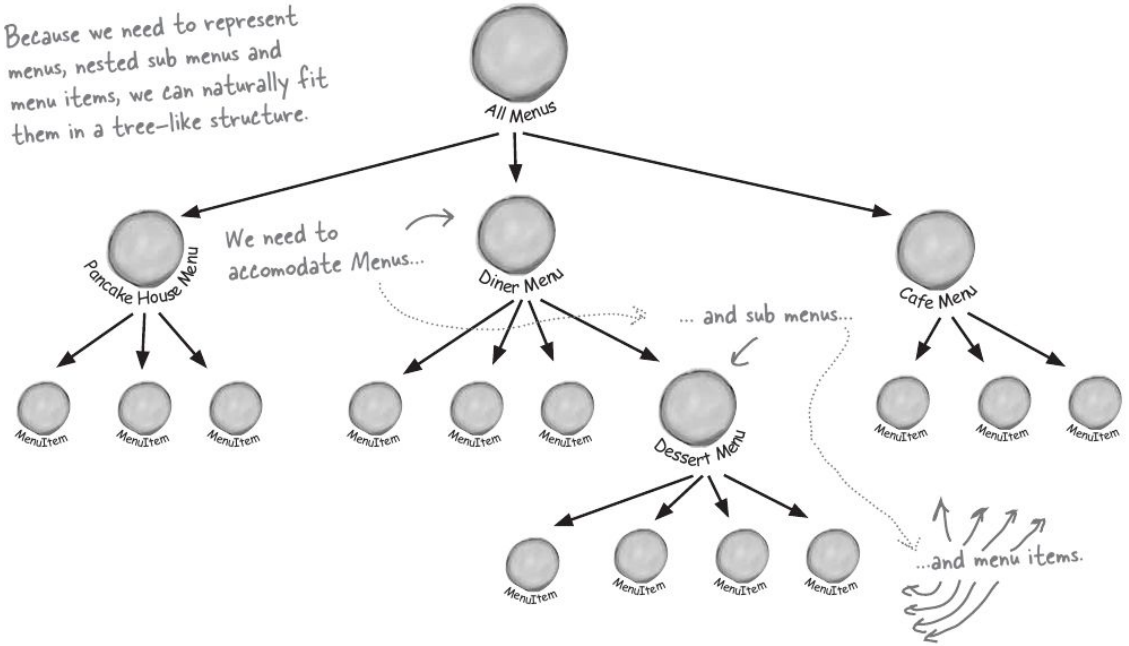
- **Tree-structured** menus with support for *submenus and items*
- Traversal should be **flexible**; allow iterating over a specific submenu (e.g., desserts) or the entire menu hierarchy.



# Something tree-like

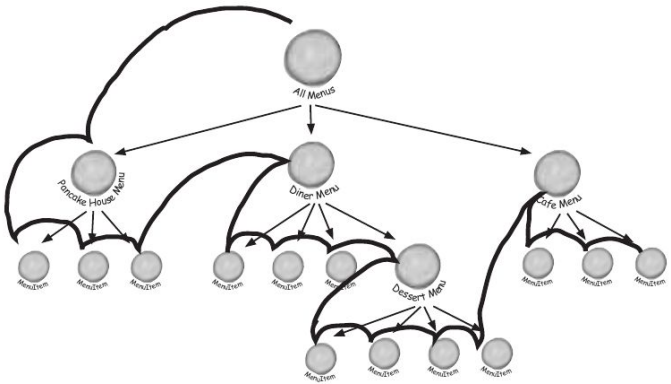
- **Tree-structured** menus with support for *submenus and items*
- Traversal should be **flexible**; allow iterating over a specific **submenu** (e.g., desserts) or the entire menu hierarchy.

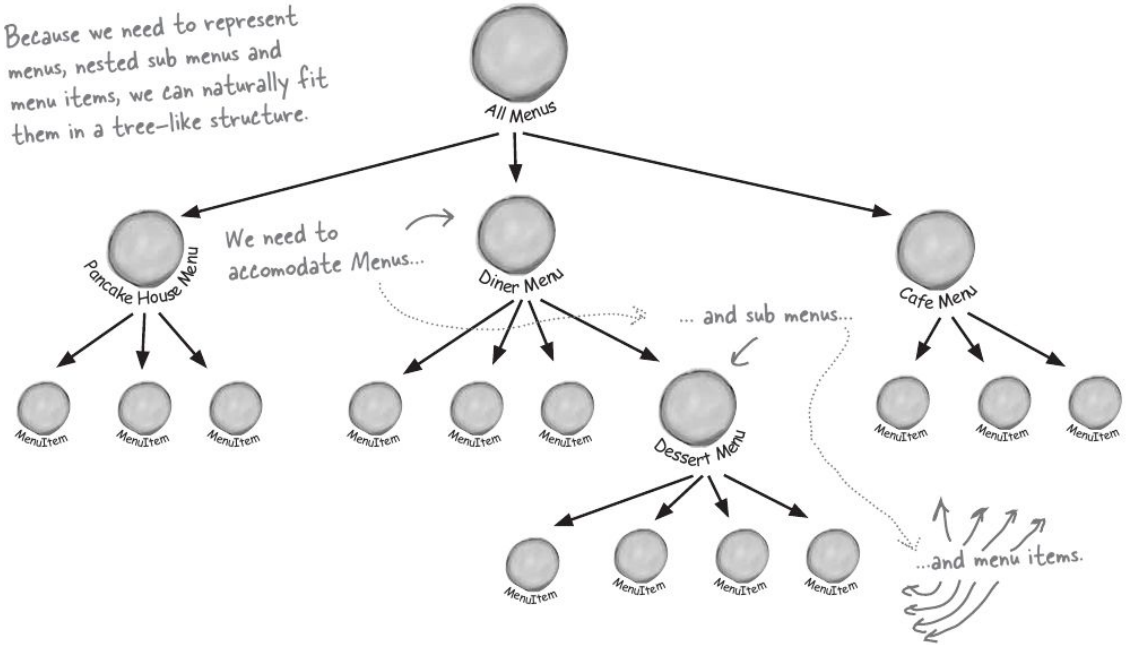


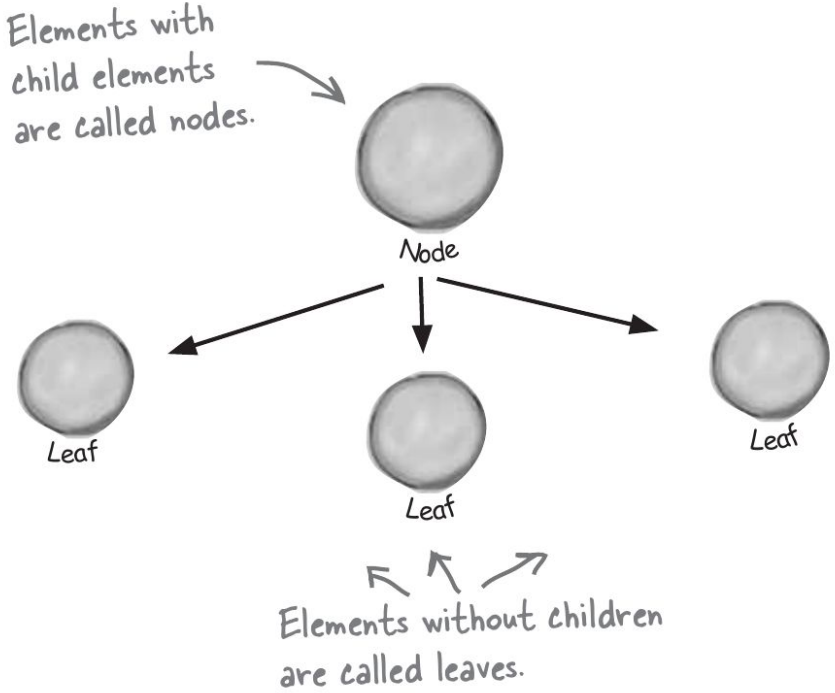
# Something tree-like

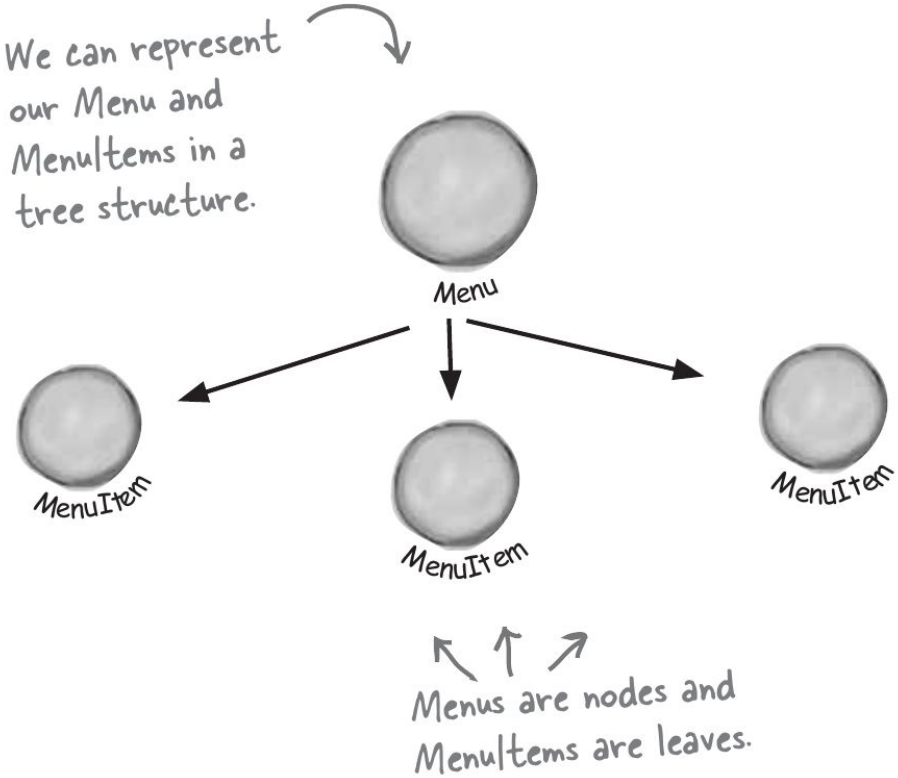
- **Tree-structured** menus with support for *submenus and items*
- Traversal should be **flexible**; allow iterating over a specific submenu (e.g., desserts) or the **entire menu** hierarchy.



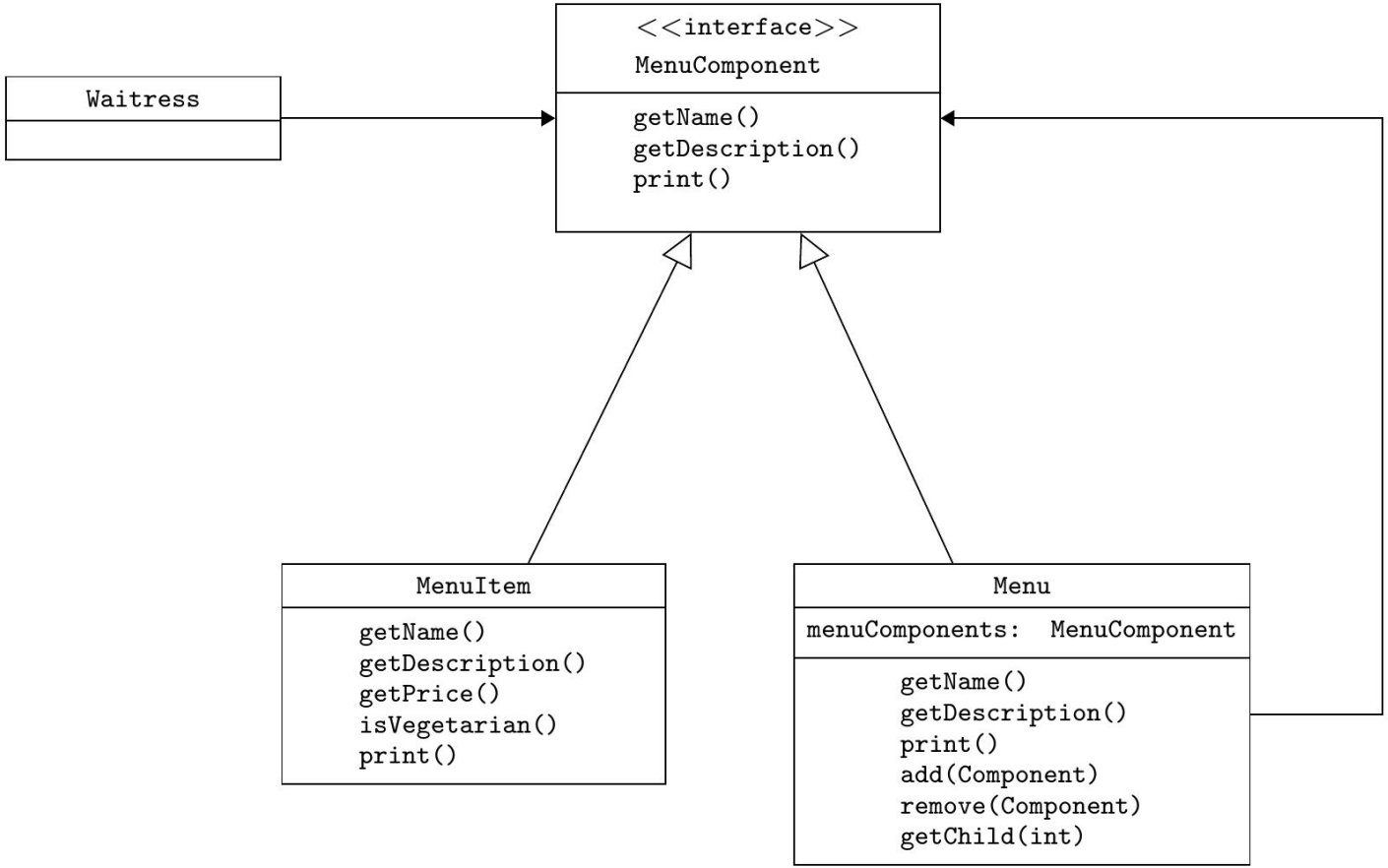



# Nodes and Leaves



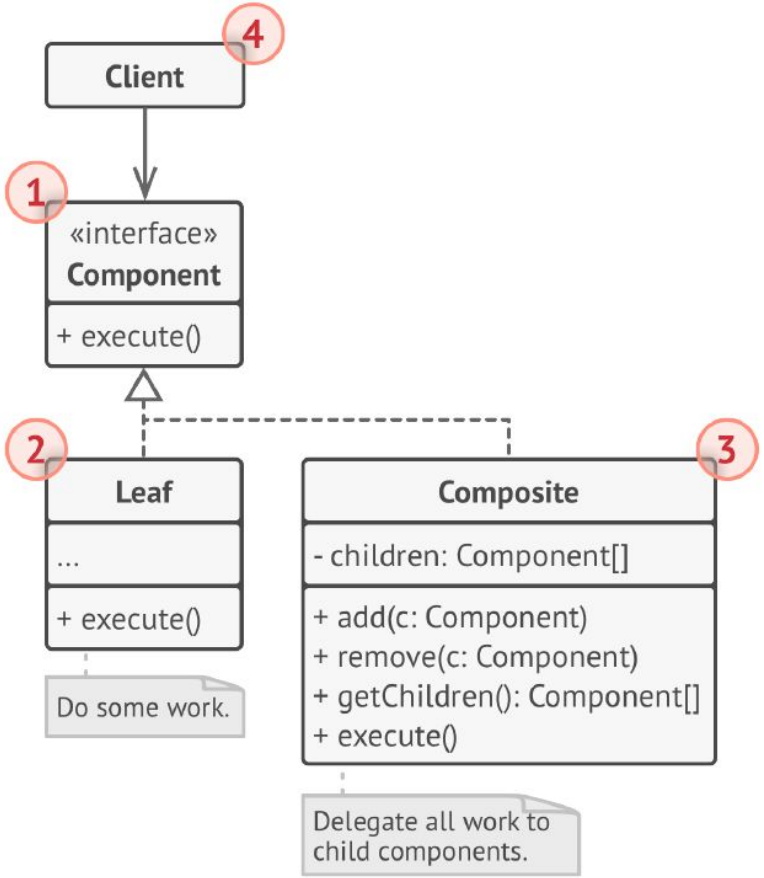



# The Design


<<interface>>
MenuComponent
getName()
getDescription()
print()
Waitress
MenuItem
getName()
getDescription()
getPrice()
isVegetarian()
print()
Menu
menuComponents: MenuComponent
getName()
getDescription()
print()
add(Component)
remove(Component)
getChild(int)

# Composite Pattern Defined

Composite pattern lets you compose objects into tree structures and then work with these structures as if they were *individual objects*.



# Composite Pattern Defined

1. The **Component** interface describes operations that are common to both simple and complex elements of the tree.

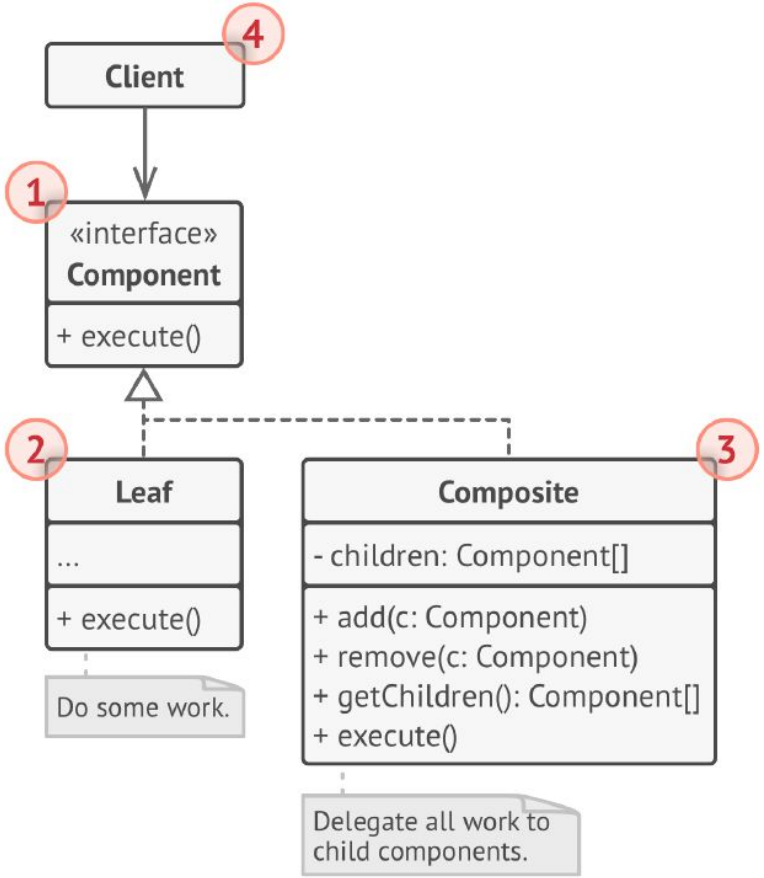
2. The **Leaf** is a basic element of a tree that doesn't have sub-elements.

   Usually, leaf components end up doing most of the real work, since they don't have anyone to delegate the work to.

3. The **Container** (aka *composite*) is an element that has sub-elements: leaves or other containers. A container doesn't know the concrete classes of its children. It works with all sub-elements only via the component interface.

   Upon receiving a request, a container delegates the work to its sub-elements, processes intermediate results and then returns the final result to the client.

4. The **Client** works with all elements through the component interface. As a result, the client can work in the same way with both simple or complex elements of the tree.



# Composite Pattern Defined

1. The **Component** interface describes operations that are common to both simple and complex elements of the tree.

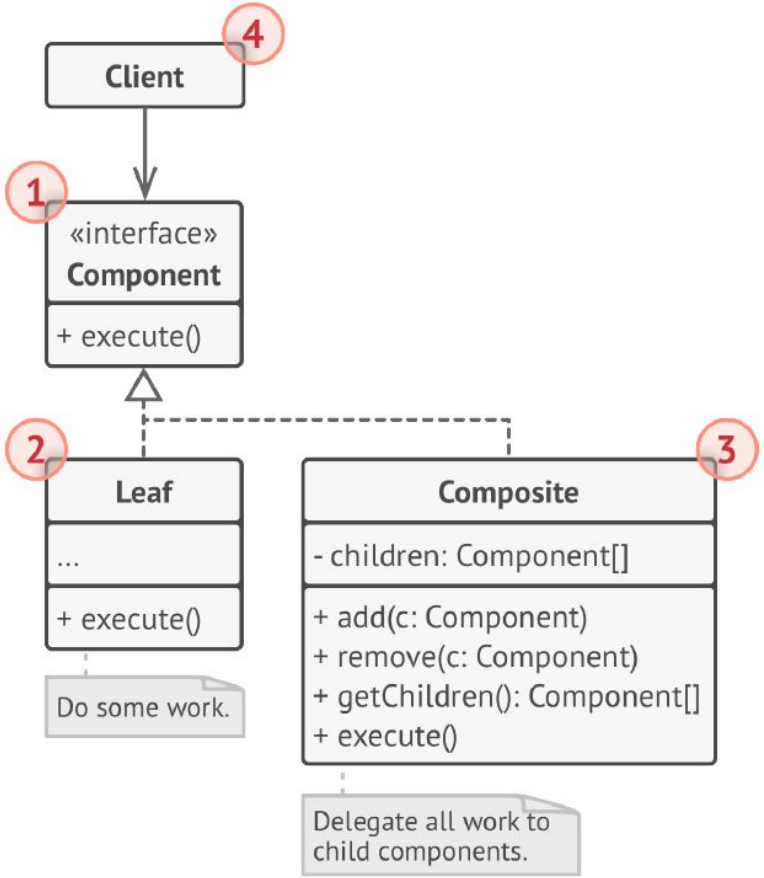
2. The **Leaf** is a basic element of a tree that doesn't have sub-elements.

   Usually, leaf components end up doing most of the real work, since they don't have anyone to delegate the work to.

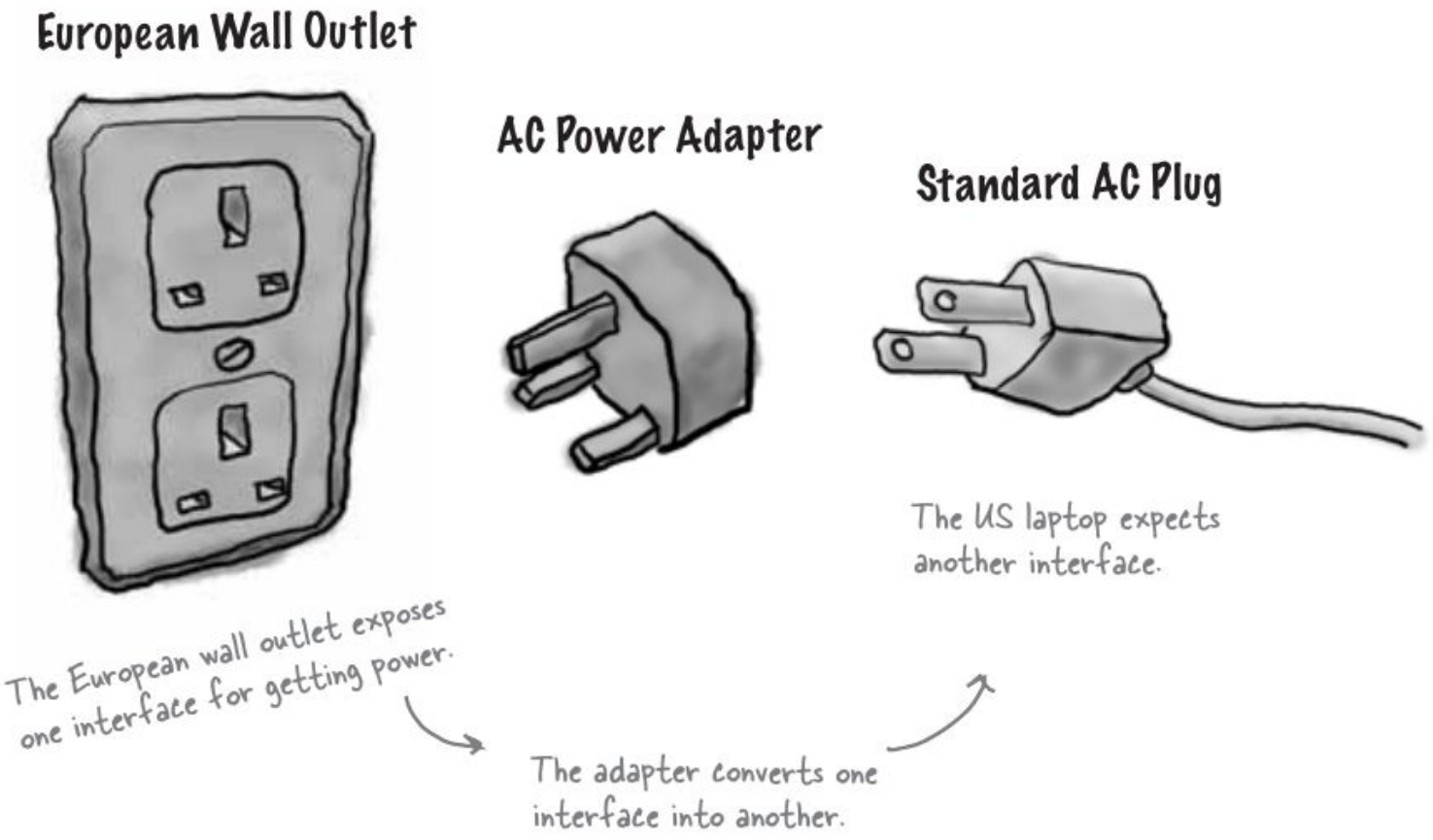
3. The **Container** (aka *composite*) is an element that has sub-elements: leaves or other containers. A container doesn't know the concrete classes of its children. It works with all sub-elements only via the component interface.

   Upon receiving a request, a container delegates the work to its sub-elements, processes intermediate results and then returns the final result to the client.

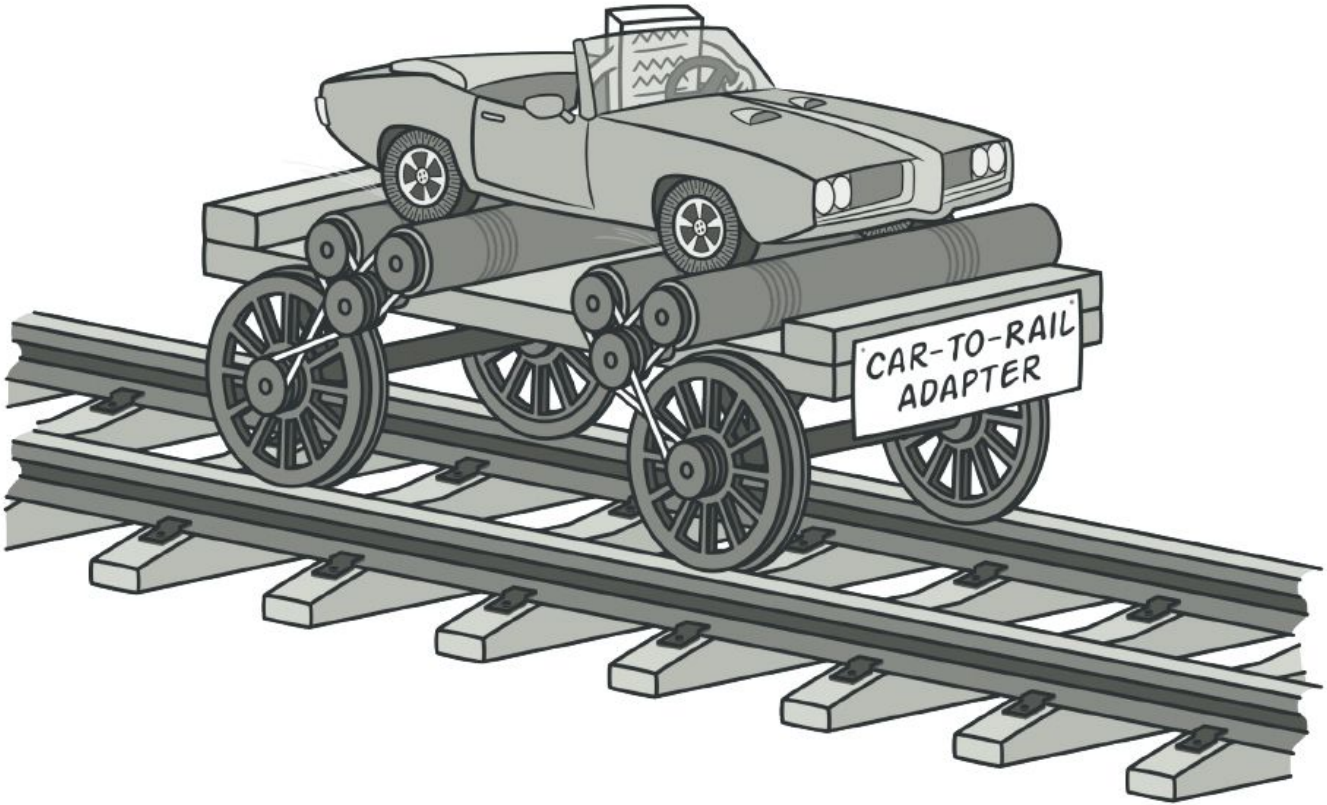
4. The **Client** works with all elements through the component interface. As a result, the client can work in the same way with both simple or complex elements of the tree.



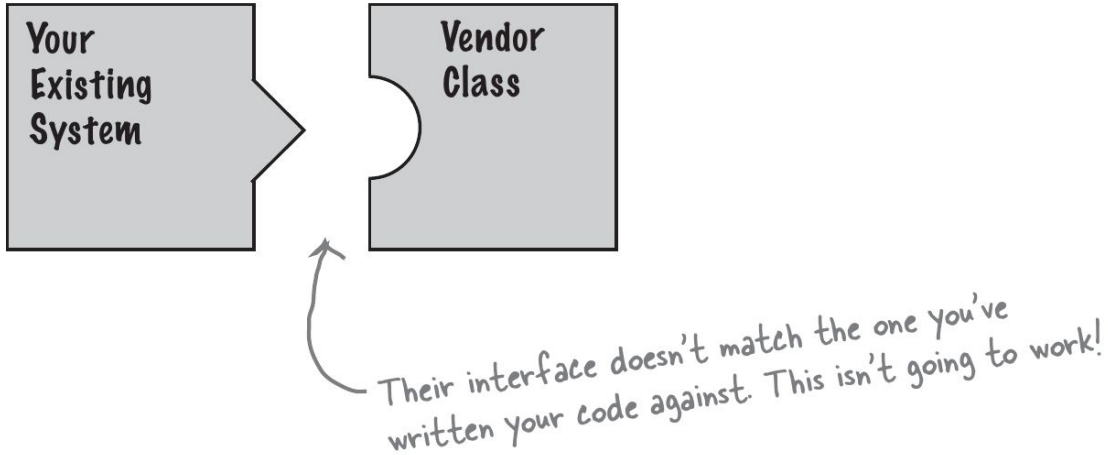
# Adapters in real life



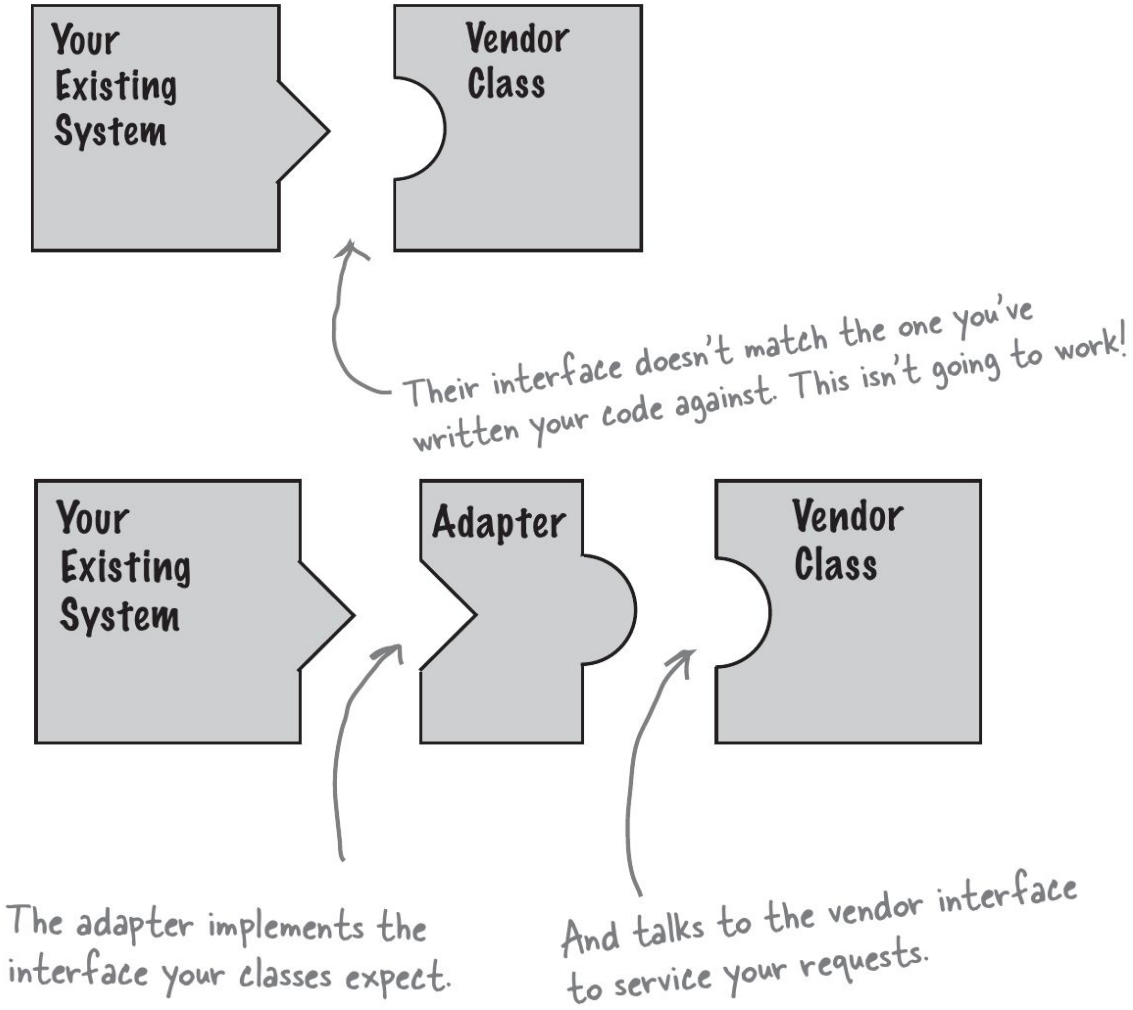
# Adapters in real life



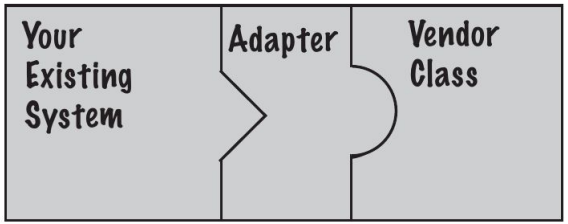
# Adapters in OO System

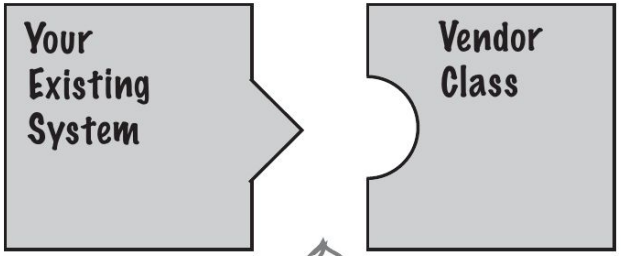


# Adapters in OO System

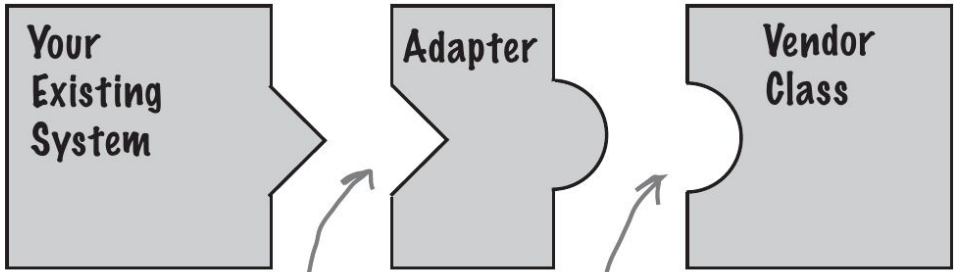


# Adapters in OO System




Their interface doesn't match the one you've written your code against. This isn't going to work!




The adapter implements the interface your classes expect.

And talks to the vendor interface to service your requests.

# Adapter in our Duck Simulator

Existing code

```
public interface Duck {
    public void quack();
    public void fly();
}

public class MallardDuck implements Duck {
    public void quack() {
        System.out.println("Quack");
    }

    public void fly() {
        System.out.println("I'm flying");
    }
}
```

# Adapter in our Duck Simulator

Existing code

```
public interface Duck {
    public void quack();
    public void fly();
}

public class MallardDuck implements Duck {
    public void quack() {
        System.out.println("Quack");
    }

    public void fly() {
        System.out.println("I'm flying");
    }
}
```

Vendor code

```
public interface Turkey {
    public void gobble();
    public void fly();
}

public class WildTurkey implements Turkey {
    public void gobble() {
        System.out.println("Gobble gobble");
    }

    public void fly() {
        System.out.println("I'm flying a short distance");
    }
}
```

# Adapter in our Duck Simulator

Existing code

```
public interface Duck {
    public void quack();
    public void fly();
}

public class MallardDuck implements Duck {
    public void quack() {
        System.out.println("Quack");
    }

    public void fly() {
        System.out.println("I'm flying");
    }
}
```

Vendor code

```
public interface Turkey {
    public void gobble();
    public void fly();
}

public class WildTurkey implements Turkey {
    public void gobble() {
        System.out.println("Gobble gobble");
    }

    public void fly() {
        System.out.println("I'm flying a short distance");
    }
}
```

```
static void testDuck(Duck duck) {
    duck.quack();
    duck.fly();
}
```

Current code uses Duck interface

# Adapter in our Duck Simulator

Existing code

```
public interface Duck {
    public void quack();
    public void fly();
}

public class MallardDuck implements Duck {
    public void quack() {
        System.out.println("Quack");
    }

    public void fly() {
        System.out.println("I'm flying");
    }
}
```

Vendor code

```
public interface Turkey {
    public void gobble();
    public void fly();
}

public class WildTurkey implements Turkey {
    public void gobble() {
        System.out.println("Gobble gobble");
    }

    public void fly() {
        System.out.println("I'm flying a short distance");
    }
}
```

```
static void testDuck(Duck duck) {
    duck.quack();
    duck.fly();
}
```

Current code uses Duck interface

How to incorporate Turkey?

# Adapter in our Duck Simulator

First, you need to implement the interface of the type you're adapting to. This is the interface your client expects to see.

```
public class TurkeyAdapter implements Duck {
    Turkey turkey;

    public TurkeyAdapter(Turkey turkey) {
        this.turkey = turkey;
    }

    public void quack() {
        turkey.gobble();
    }

    public void fly() {
        for(int i=0; i < 5; i++) {
            turkey.fly();
        }
    }
}
```

Next, we need to get a reference to the object that we are adapting; here we do that through the constructor.

Now we need to implement all the methods in the interface; the quack() translation between classes is easy: just call the gobble() method.

Even though both interfaces have a fly() method, Turkeys fly in short spurts – they can't do long-distance flying like ducks. To map between a Duck's fly() method and a Turkey's, we need to call the Turkey's fly() method five times to make up for it.

```
static void testDuck(Duck duck) {
    duck.quack();
    duck.fly();
}
```

# Adapter in our Duck Simulator

First, you need to implement the interface of the type you're adapting to. This is the interface your client expects to see.

```
public class TurkeyAdapter implements Duck {
    Turkey turkey;

    public TurkeyAdapter(Turkey turkey) {
        this.turkey = turkey;
    }

    public void quack() {
        turkey.gobble();
    }

    public void fly() {
        for(int i=0; i < 5; i++) {
            turkey.fly();
        }
    }
}
```

Next, we need to get a reference to the object that we are adapting; here we do that through the constructor.

Now we need to implement all the methods in the interface; the quack() translation between classes is easy: just call the gobble() method.

Even though both interfaces have a fly() method, Turkeys fly in short spurts – they can't do long-distance flying like ducks. To map between a Duck's fly() method and a Turkey's, we need to call the Turkey's fly() method five times to make up for it.

```
static void testDuck(Duck duck) {
    duck.quack();
    duck.fly();
}
```

# Adapter in our Duck Simulator

First, you need to implement the interface of the type you're adapting to. This is the interface your client expects to see.

```
public class TurkeyAdapter implements Duck {
    Turkey turkey;

    public TurkeyAdapter(Turkey turkey) {
        this.turkey = turkey;
    }

    public void quack() {
        turkey.gobble();
    }

    public void fly() {
        for(int i=0; i < 5; i++) {
            turkey.fly();
        }
    }
}
```

Next, we need to get a reference to the object that we are adapting; here we do that through the constructor.

Now we need to implement all the methods in the interface; the quack() translation between classes is easy: just call the gobble() method.

Even though both interfaces have a fly() method, Turkeys fly in short spurts – they can't do long-distance flying like ducks. To map between a Duck's fly() method and a Turkey's, we need to call the Turkey's fly() method five times to make up for it.

```
static void testDuck(Duck duck) {
    duck.quack();
    duck.fly();
}
```

# Test Drive

```
public class DuckTestDrive {
    public static void main(String[] args) {
        MallardDuck duck = new MallardDuck();

        WildTurkey turkey = new WildTurkey();
        Duck turkeyAdapter = new TurkeyAdapter(turkey);

        System.out.println("The Turkey says...");
        turkey.gobble();
        turkey.fly();

        System.out.println("\nThe Duck says...");
        testDuck(duck);

        System.out.println("\nThe TurkeyAdapter says...");
        testDuck(turkeyAdapter);
    }

    static void testDuck(Duck duck) {
        duck.quack();
        duck.fly();
    }
}
```

# Test Drive

```
public class DuckTestDrive {
    public static void main(String[] args) {
        MallardDuck duck = new MallardDuck();

        WildTurkey turkey = new WildTurkey();
        Duck turkeyAdapter = new TurkeyAdapter(turkey);

        System.out.println("The Turkey says...");
        turkey.gobble();
        turkey.fly();

        System.out.println("\nThe Duck says...");
        testDuck(duck);

        System.out.println("\nThe TurkeyAdapter says...");
        testDuck(turkeyAdapter);
    }

    static void testDuck(Duck duck) {
        duck.quack();
        duck.fly();
    }
}
```

```
The Turkey says...
Gobble gobble
I'm flying a short distance

The Duck says...
Quack
I'm flying

The TurkeyAdapter says...
Gobble gobble
I'm flying a short distance
I'm flying a short distance
I'm flying a short distance
I'm flying a short distance
I'm flying a short distance
```

# Test Drive

```
public class DuckTestDrive {
    public static void main(String[] args) {
        MallardDuck duck = new MallardDuck();

        WildTurkey turkey = new WildTurkey();
        Duck turkeyAdapter = new TurkeyAdapter(turkey);

        System.out.println("The Turkey says...");
        turkey.gobble();
        turkey.fly();

        System.out.println("\nThe Duck says...");
        testDuck(duck);

        System.out.println("\nThe TurkeyAdapter says...");
        testDuck(turkeyAdapter);
    }

    static void testDuck(Duck duck) {
        duck.quack();
        duck.fly();
    }
}
```

```
The Turkey says...

Gobble gobble

I'm flying a short distance

The Duck says...
Quack
I'm flying

The TurkeyAdapter says...
Gobble gobble
I'm flying a short distance
I'm flying a short distance
I'm flying a short distance
I'm flying a short distance
I'm flying a short distance
```

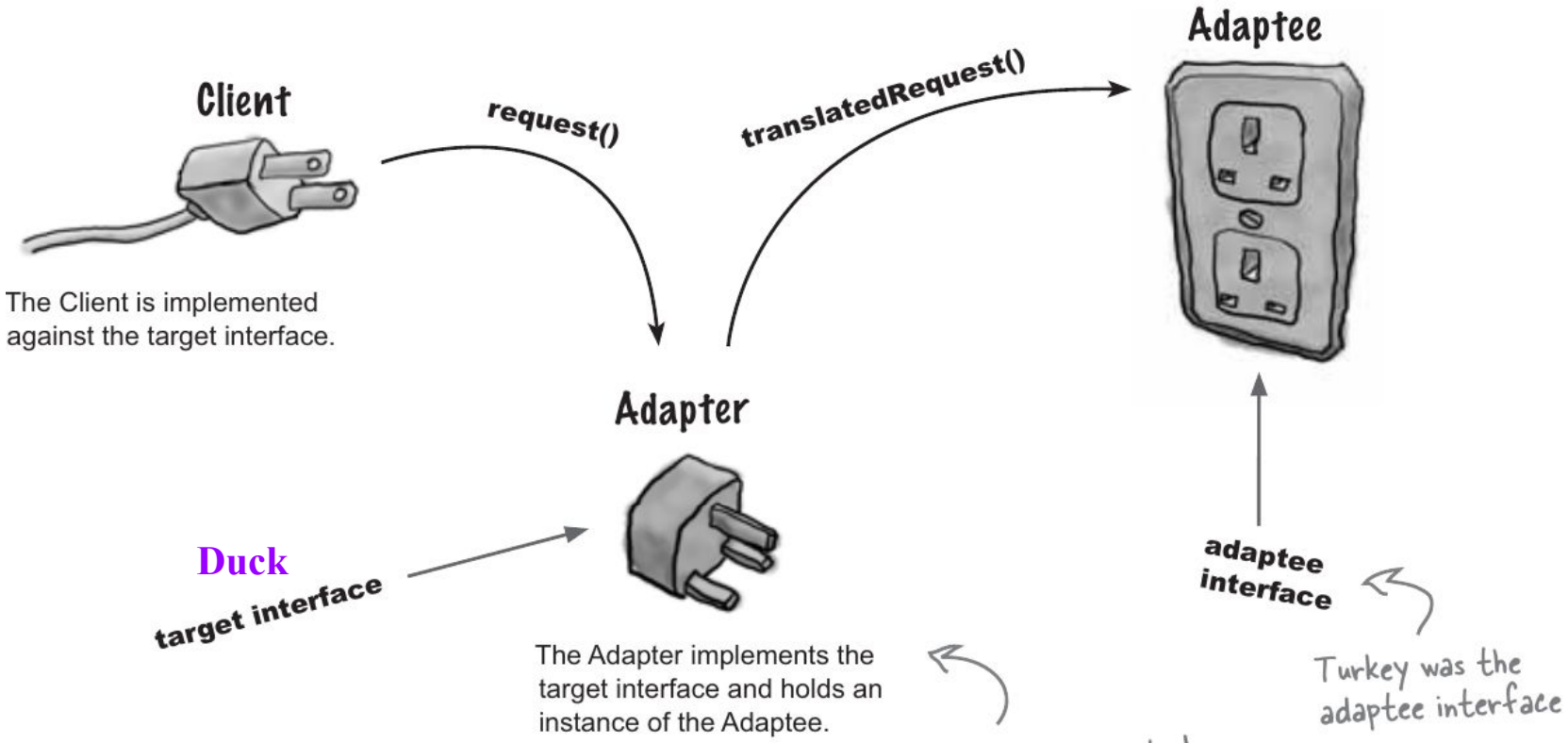
# Test Drive

```
public class DuckTestDrive {
    public static void main(String[] args) {
        MallardDuck duck = new MallardDuck();

        WildTurkey turkey = new WildTurkey();
        Duck turkeyAdapter = new TurkeyAdapter(turkey);

        System.out.println("The Turkey says...");
        turkey.gobble();
        turkey.fly();

        System.out.println("\nThe Duck says...");
        testDuck(duck);

        System.out.println("\nThe TurkeyAdapter says...");
        testDuck(turkeyAdapter);
    }

    static void testDuck(Duck duck) {
        duck.quack();
        duck.fly();
    }
}
```

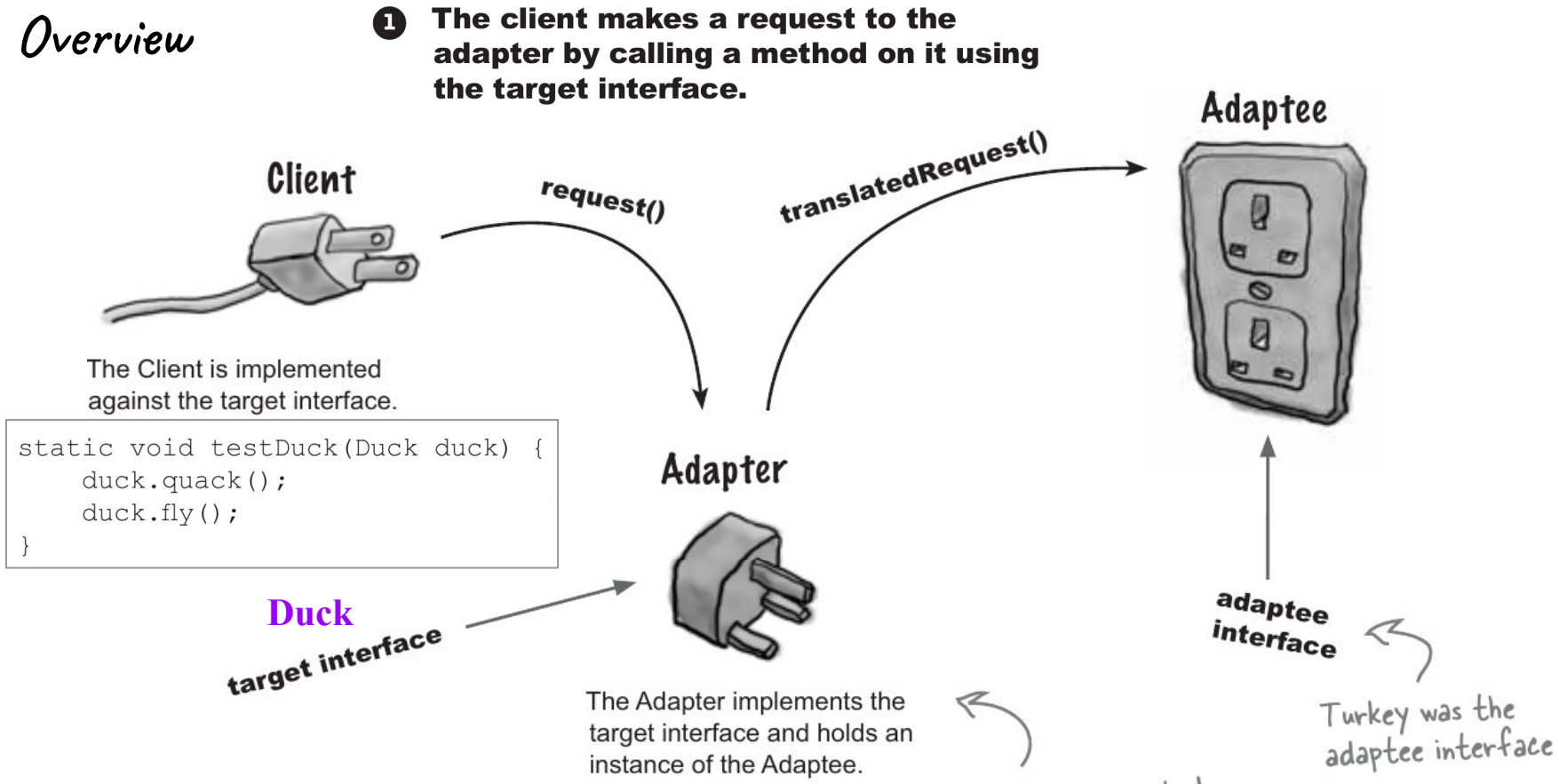
Getting Turkey using existing Duck code

```
The Turkey says...

Gobble gobble

I'm flying a short distance

The Duck says...
Quack
I'm flying

The TurkeyAdapter says...
Gobble gobble
I'm flying a short distance
I'm flying a short distance
I'm flying a short distance
I'm flying a short distance
I'm flying a short distance
```

# Overview


Client
request()
translatedRequest()
Adaptee
The Client is implemented against the target interface.
Adapter
Duck
target interface
The Adapter implements the target interface and holds an instance of the Adaptee.
adaptee interface
Turkey was the adaptee interface
TurkeyAdapter implemented the target interface, Duck.

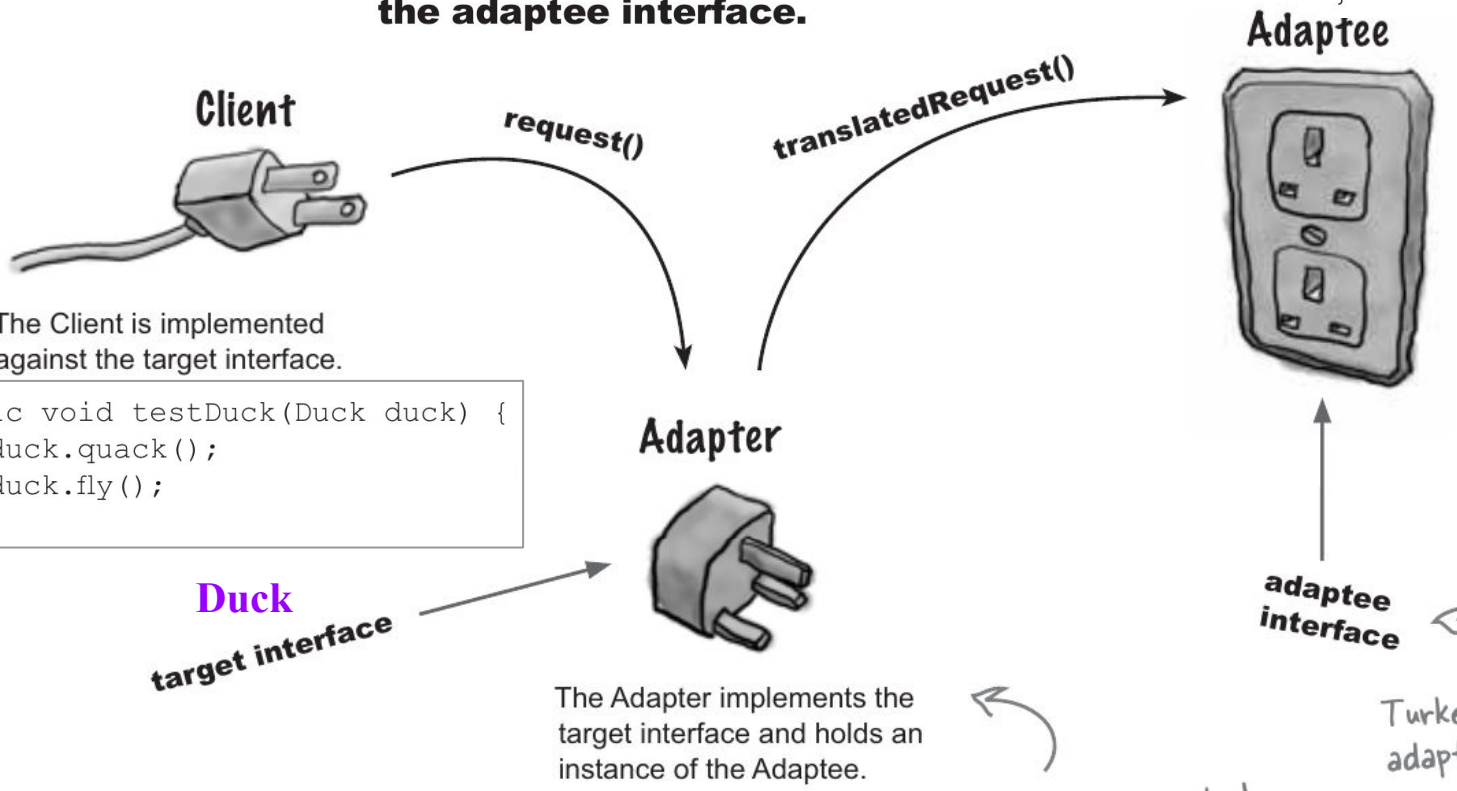
# Overview


1 The client makes a request to the adapter by calling a method on it using the target interface.
Client
request()
translatedRequest()
Adaptee
The Client is implemented against the target interface.
static void testDuck(Duck duck) {
duck.quack();
duck.fly();
}
Adapter
Duck
target interface
adaptee interface
The Adapter implements the target interface and holds an instance of the Adaptee.
Turkey was the adaptee interface
TurkeyAdapter implemented the target interface, Duck.

# Overview

❷ **The adapter translates that request into one or more calls on the adaptee using the adaptee interface.**

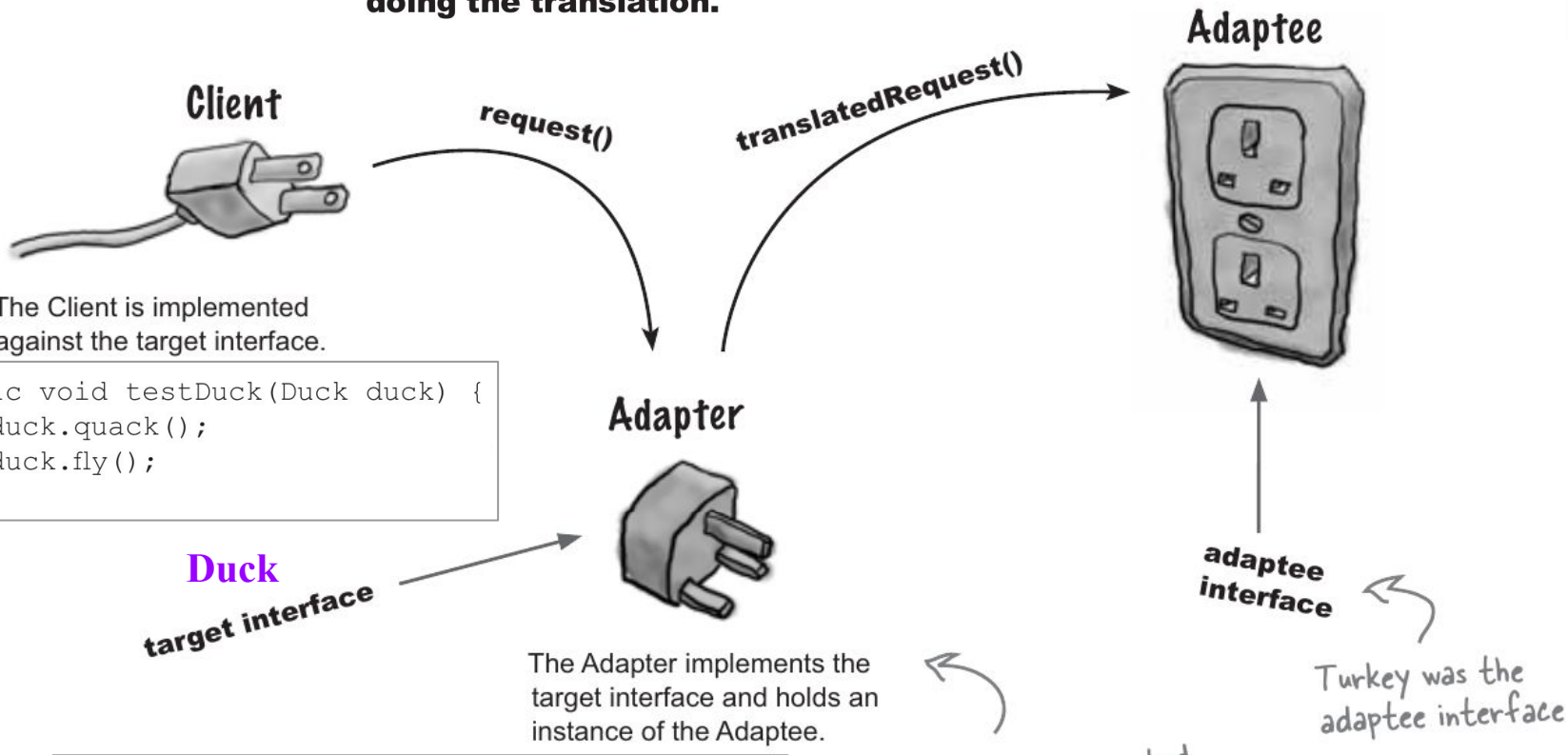
```
public void quack() {
    turkey.gobble();
}
```



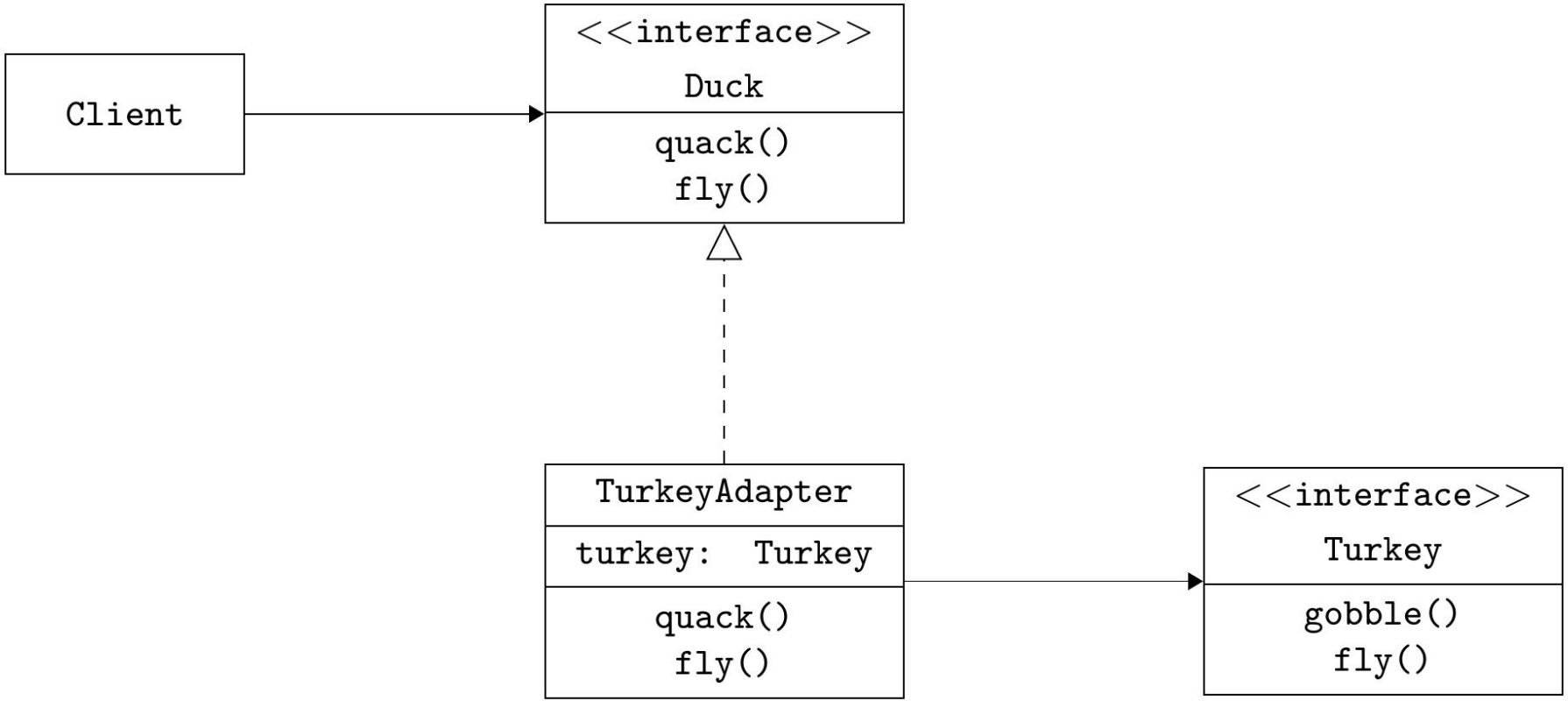
# Overview

❸ **The client receives the results of the call and never knows there is an adapter doing the translation.**

Note that the Client and Adaptee are decoupled – neither knows



# Birds Eye View



# There are no DUMB Questions

**Q: How much "adapting" does an adapter need to do? It seems like if I need to implement a large target interface, I could have a LOT of work on my hands.**

A: You certainly could. The job of implementing an adapter really is proportional to the size of the interface you need to support as your target interface. Think about your options, however. You could rework all your client-side calls to the interface, which would result in a lot of investigative work and code changes. Or, you can cleanly provide one class that encapsulates all the changes in one class.

**Q: Does an adapter always wrap one and only one class?**

A: The Adapter Pattern's role is to convert one interface into another. While most examples of the adapter pattern show an adapter wrapping one adaptee, we both know the world is often a bit more messy. So, you may well have situations where an adapter holds two or more adaptees that are needed to implement the target interface.

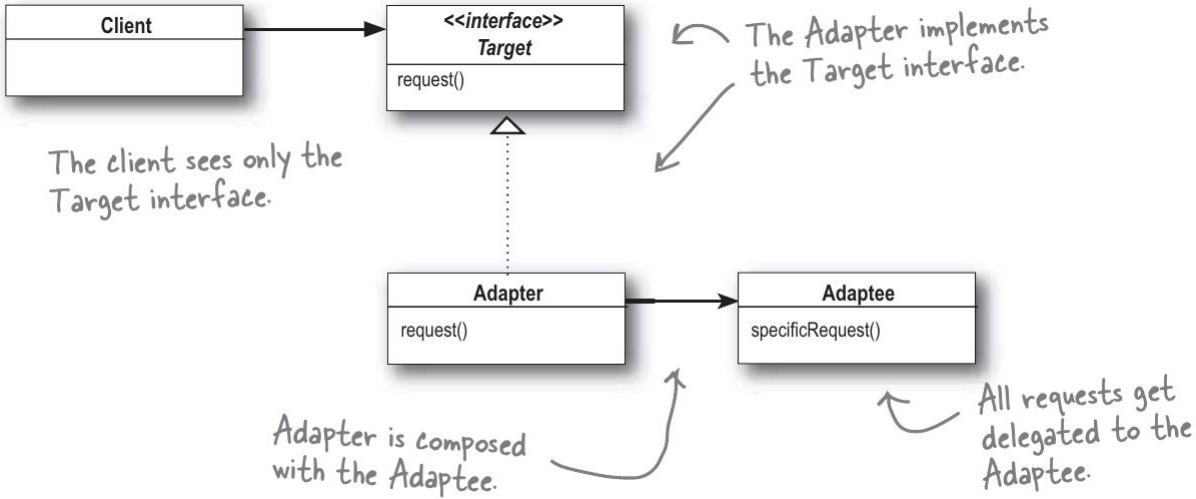
This relates to another pattern called the Facade Pattern; people often confuse the two. Remind us to revisit this point when we talk about facades later in this chapter.

**Q: What if I have old and new parts of my system, the old parts expect the old vendor interface, but we've already written the new parts to use the new vendor interface? It is going to get confusing using an adapter here and the unwrapped interface there. Wouldn't I be better off just writing my older code and forgetting the adapter?**

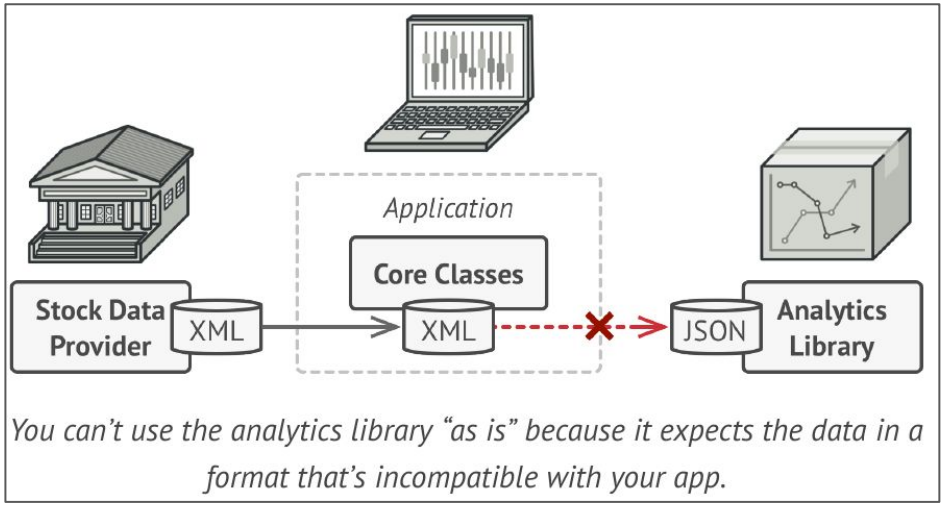
A: Not necessarily. One thing you can do is create a Two Way Adapter that supports both interfaces. To create a Two Way Adapter, just implement both interfaces involved, so the adapter can act as an old interface or a new interface.

# Adapter Pattern Defined

Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of **incompatible** interfaces.

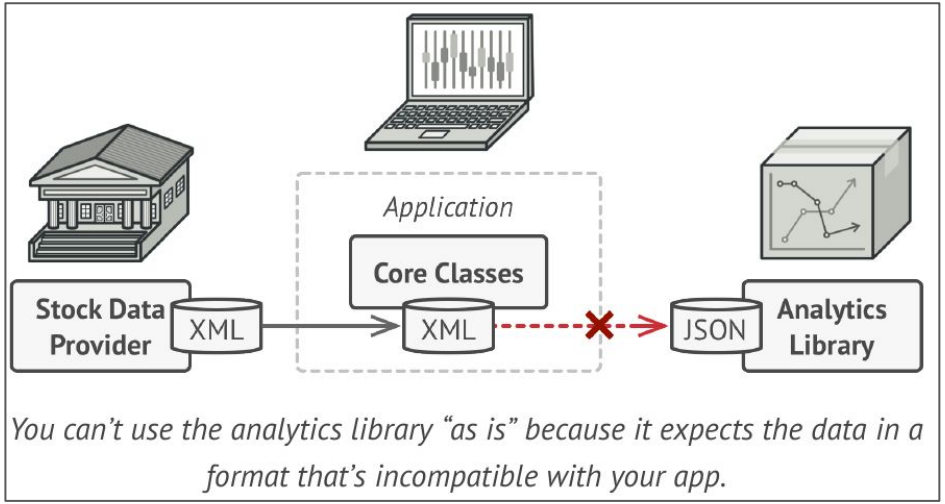


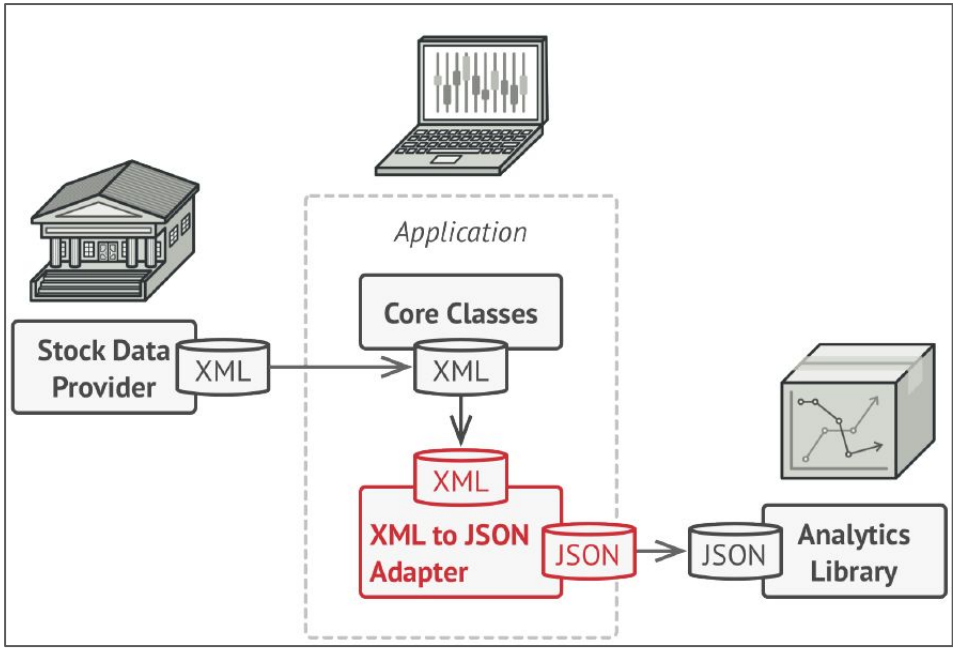
# Yet Another Example




*You can't use the analytics library "as is" because it expects the data in a format that's incompatible with your app.*

# Yet Another Example

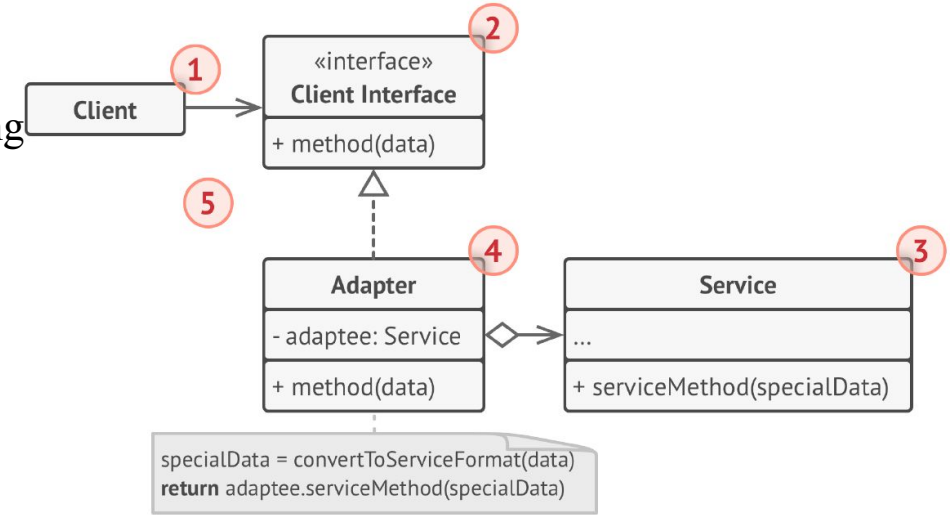


# Object Adapter

Uses the **composition** principle:

- The adapter implements the interface of one object and wraps the other one.
- Can be implemented in all popular programming languages.



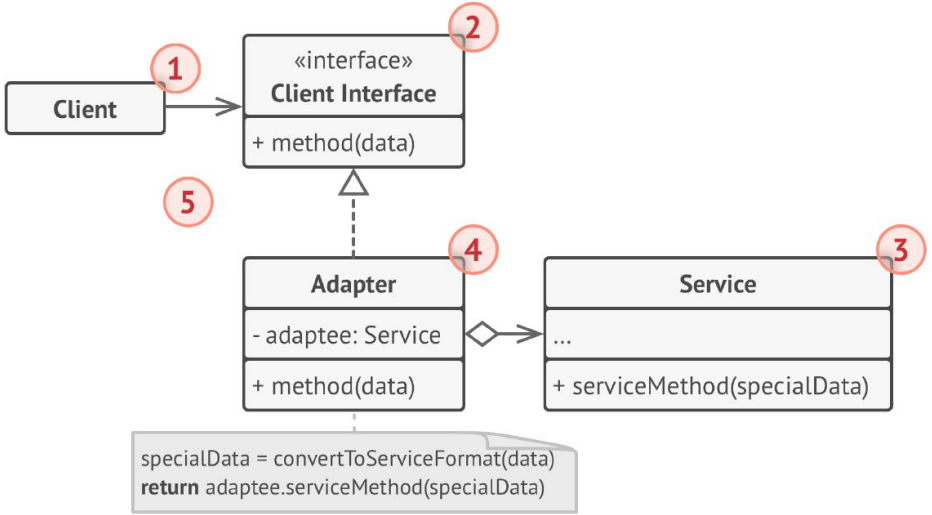
# Object Adapter

1. The **Client** is a class that contains the existing business logic of the program.

2. The **Client Interface** describes a protocol that other classes must follow to be able to collaborate with the client code.

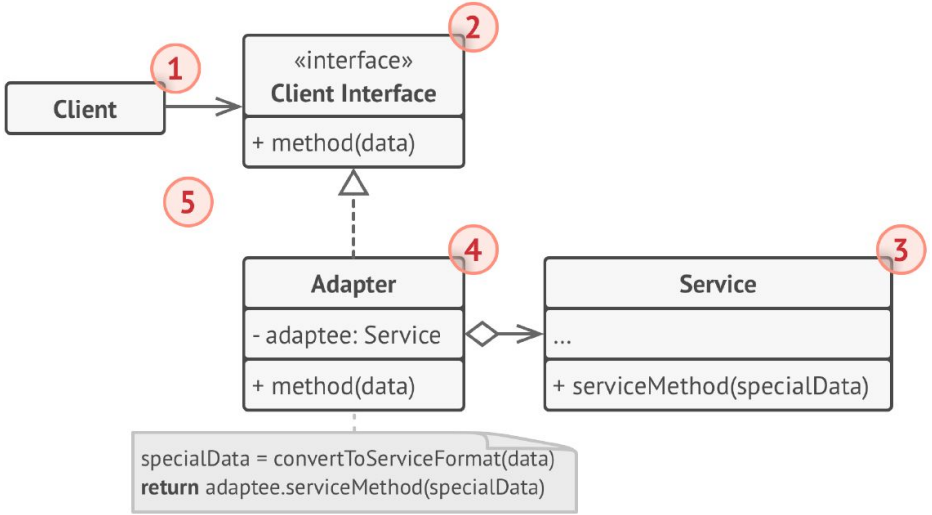
3. The **Service** is some useful class (usually 3rd-party or legacy). The client can't use this class directly because it has an incompatible interface.

4. The **Adapter** is a class that's able to work with both the client and the service: it implements the client interface, while wrapping the service object. The adapter receives calls from the client via the adapter interface and translates them into calls to the wrapped service object in a format it can understand.



# Object Adapter

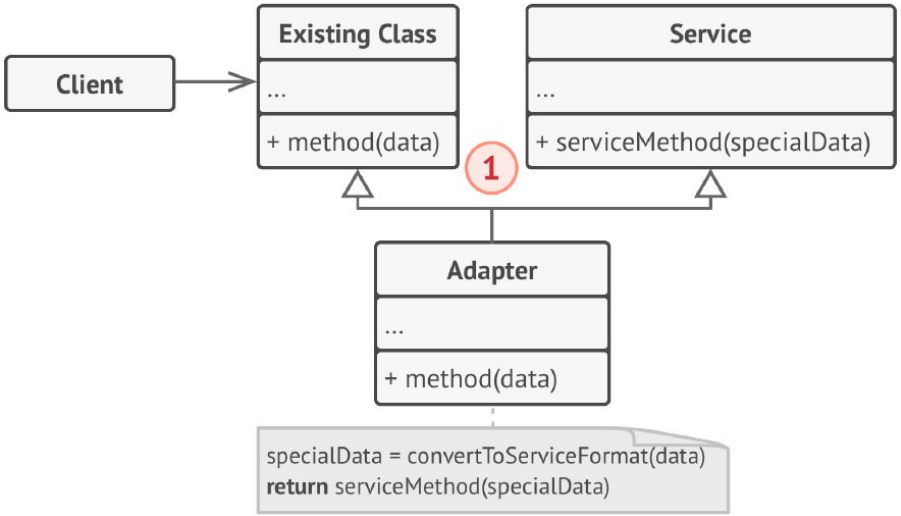
5. The client code doesn't get coupled to the concrete adapter class as long as it works with the adapter via the client interface. Thanks to this, you can introduce new types of adapters into the program without breaking the existing client code. This can be useful when the interface of the service class gets changed or replaced: you can just create a new adapter class without changing the client code.



# Class Adapter

Uses **inheritance**:

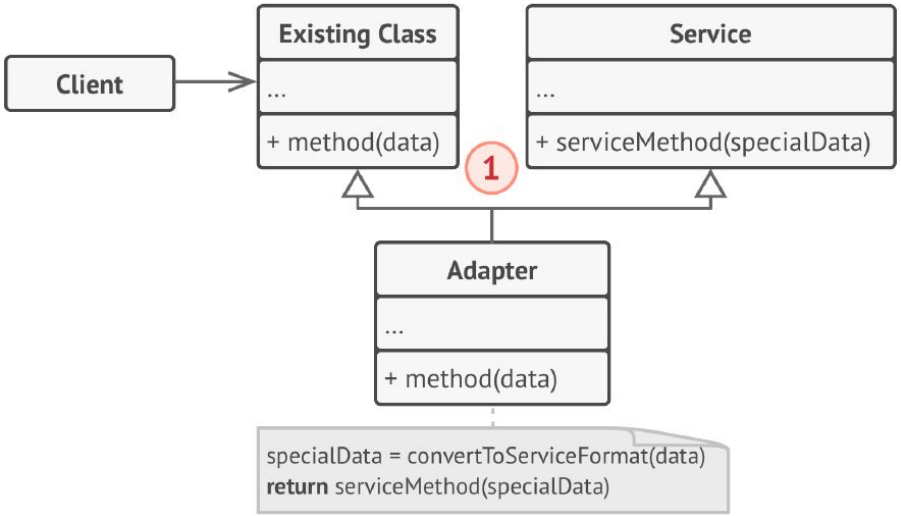
- Adapter inherits interfaces from both objects at the same time.
- Can only be implemented in programming languages that support multiple inheritance, such as C++.



# Class Adapter

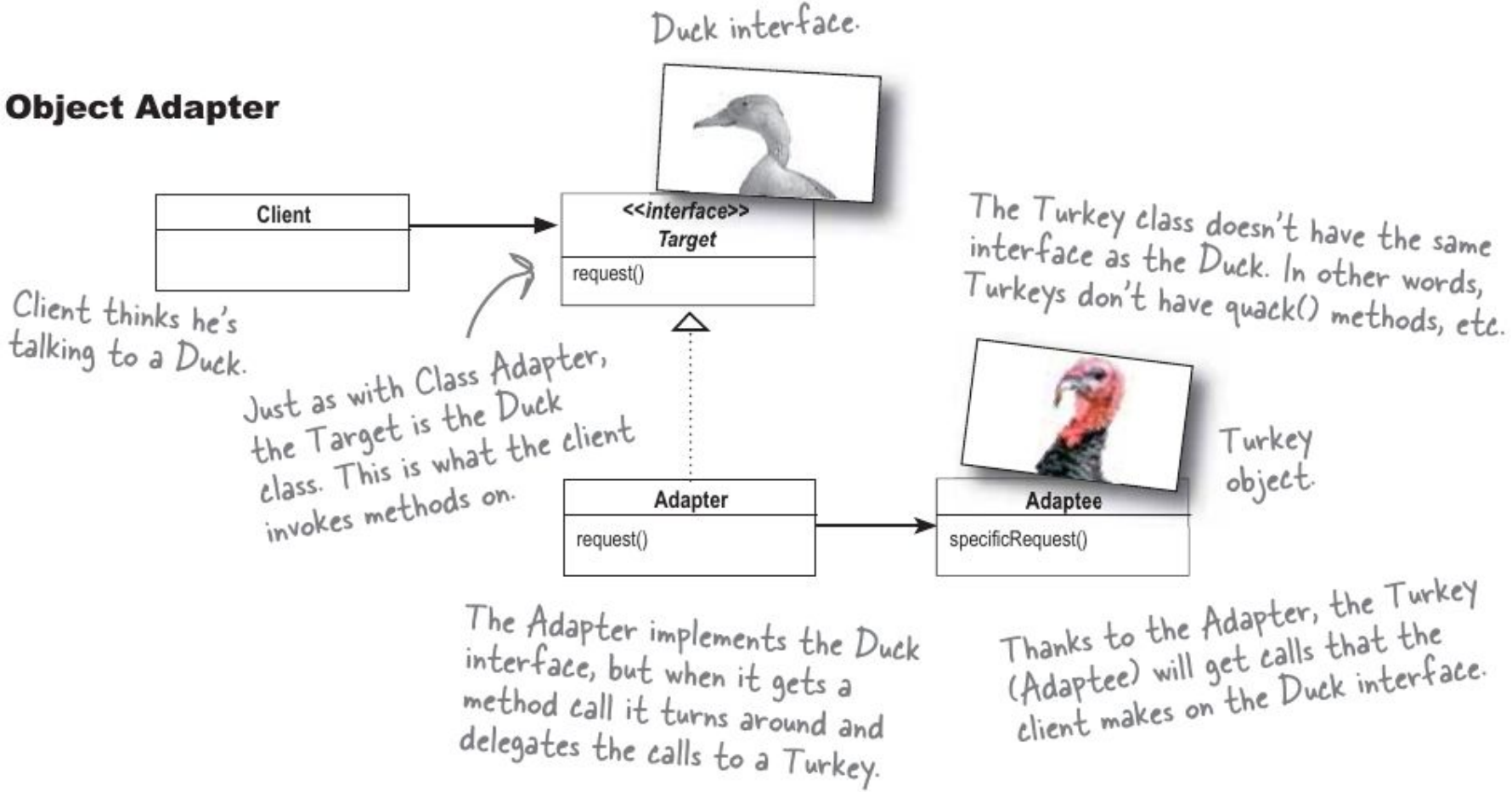
Uses **inheritance**:

- Adapter inherits interfaces from both objects at the same time.
- Can only be implemented in programming languages that support multiple inheritance, such as C++.

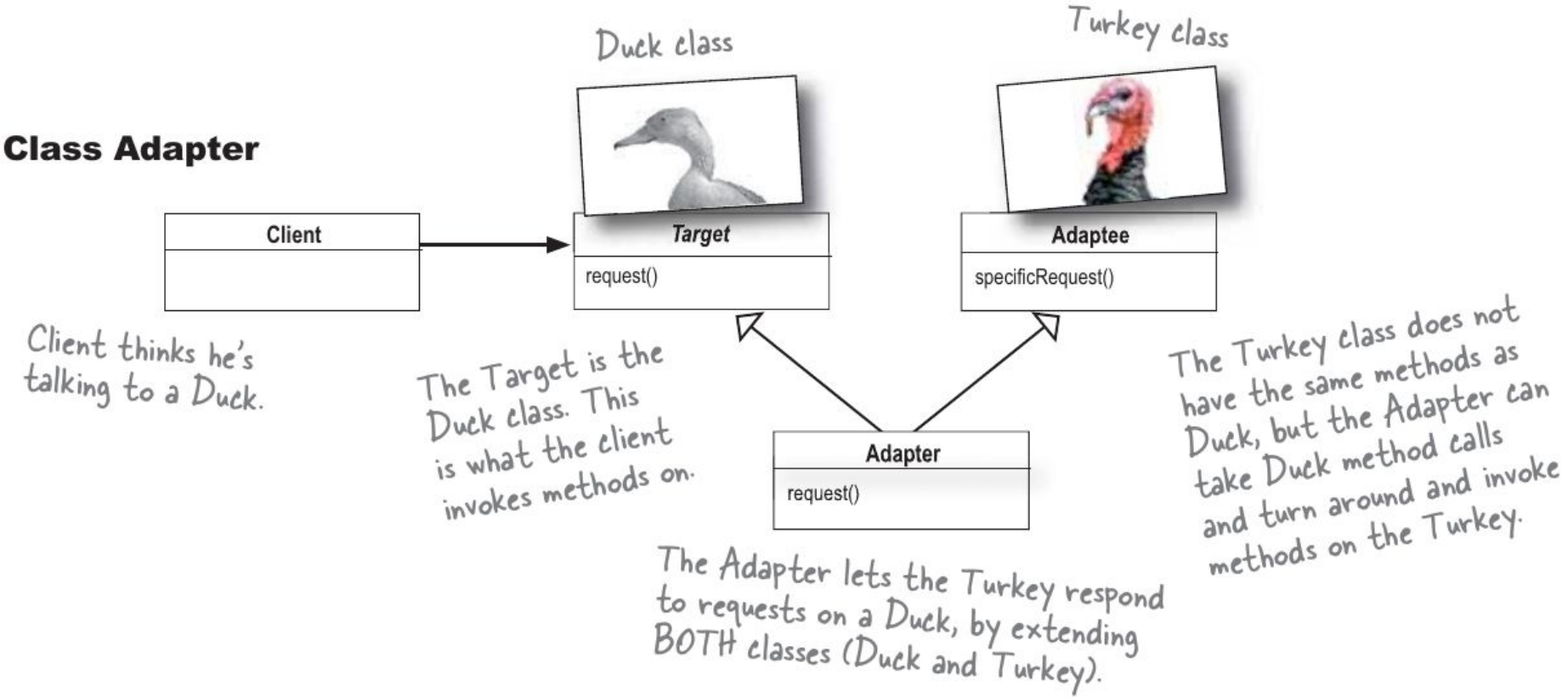


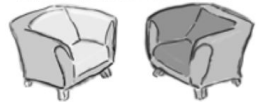

1. The **Class Adapter** doesn't need to wrap any objects because it inherits behaviors from both the client and the service. The adaptation happens within the overridden methods. The resulting adapter can be used in place of an existing client class.

## Object Adapter



## Class Adapter





**Object Adapter**

Because I use composition I've got a leg up. I can not only adapt an adaptee class, but any of its subclasses.

In my part of the world, we like to use composition over inheritance; you may be saving a few lines of code, but all I'm doing is writing a little code to delegate to the adaptee. We like to keep things flexible.

You're worried about one little object? You might be able to quickly override a method, but any behavior I add to my adapter code works with my adaptee class *and* all its subclasses.

Hey, come on, cut me a break, I just need to compose with the subclass to make that work.

You wanna see messy? Look in the mirror!

**Class Adapter**

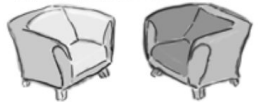
That's true, I do have trouble with that because I am committed to one specific adaptee class, but I have a huge advantage because I don't have to reimplement my entire adaptee. I can also override the behavior of my adaptee if I need to because I'm just subclassing.

Flexible maybe, efficient? No. Using a class adapter there is just one of me, not an adapter and an adaptee.

Yeah, but what if a subclass of adaptee adds some new behavior. Then what?

Sounds messy...

# Fireside Chats



**Decorator**

I'm important. My job is all about *responsibility* – you know that when a Decorator is involved there's going to be some new responsibilities or behaviors added to your design.

That may be true, but don't think we don't work hard. When we have to decorate a big interface, whoa, that can take a lot of code.

Cute. Don't think we get all the glory; sometimes I'm just one decorator that is being wrapped by who knows how many other decorators. When a method call gets delegated to you, you have no idea how many other decorators have already dealt with it and you don't know that you'll ever get noticed for your efforts servicing the request.

**Adapter**

You guys want all the glory while us adapters are down in the trenches doing the dirty work: converting interfaces. Our jobs may not be glamorous, but our clients sure do appreciate us making their lives simpler.

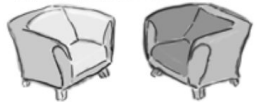
Try being an adapter when you've got to bring several classes together to provide the interface your client is expecting. Now that's tough. But we have a saying: "an uncoupled client is a happy client."

Hey, if adapters are doing their job, our clients never even know we're there. It can be a thankless job.

**Decorator**

**Adapter**

Fireside Chats



But, the great thing about us adapters is that we allow clients to make use of new libraries and subsets without changing *any* code, they just rely on us to do the conversion for them. Hey, it's a niche, but we're good at it.

Well us decorators do that as well, only we allow *new behavior* to be added to classes without altering existing code. I still say that adapters are just fancy decorators – I mean, just like us, you wrap an object.

No, no, no, not at all. We *always* convert the interface of what we wrap, you *never* do. I'd say a decorator is like an adapter; it is just that you don't change the interface!

Uh, no. Our job in life is to extend the behaviors or responsibilities of the objects we wrap, we aren't a *simple pass through*.

Hey, who are you calling a simple pass through? Come on down and we'll see how long *you* last converting a few interfaces!

Maybe we should agree to disagree. We seem to look somewhat similar on paper, but clearly we are *miles* away in our *intent*.

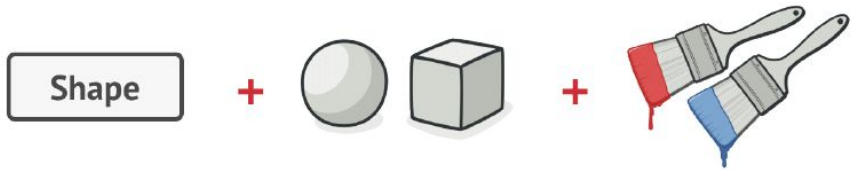
Oh yeah, I'm with you there.

# Colored Shapes

- You have two shapes Circle and Square .

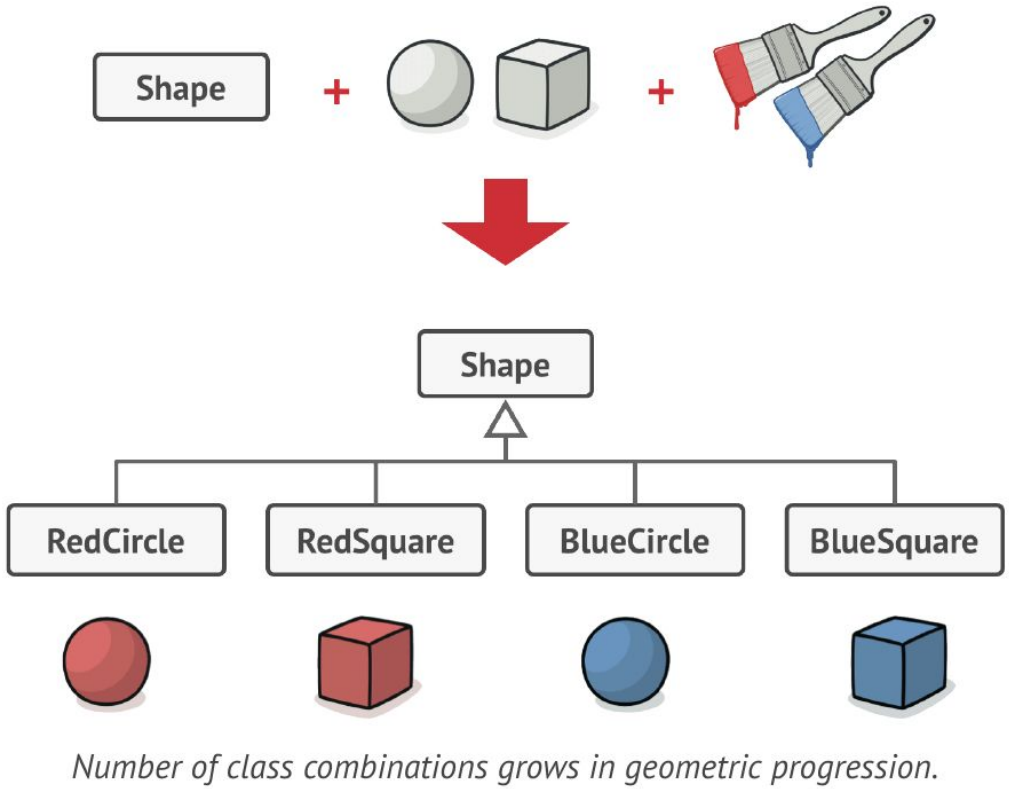
# Colored Shapes

- You have two shapes Circle and Square .
- You want to incorporate colors, Red and Blue



# Colored Shapes

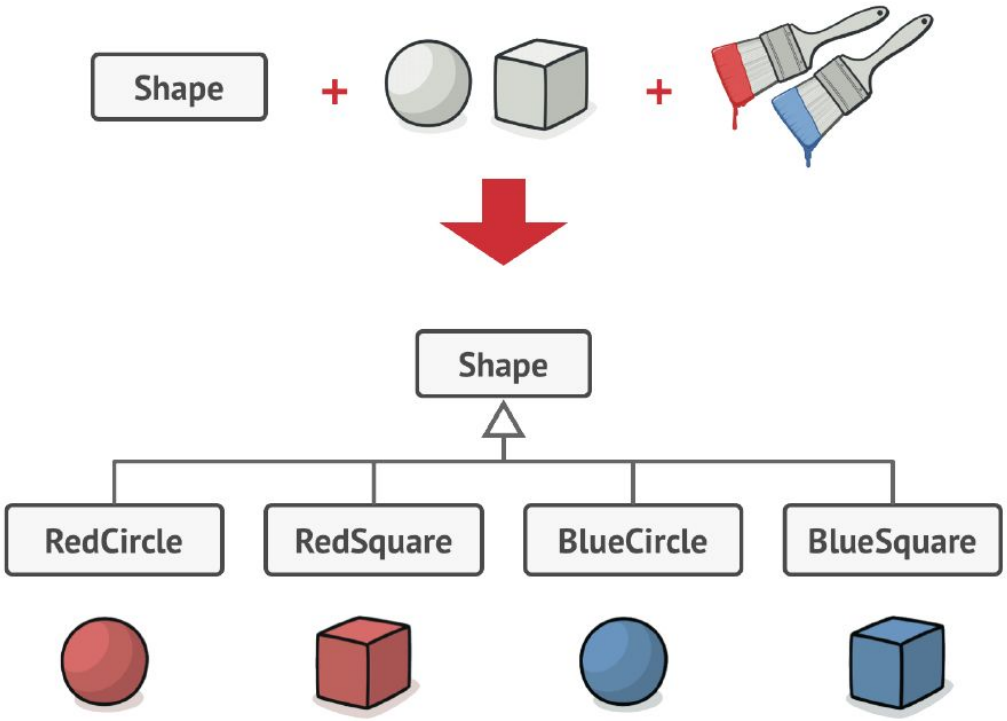
- You have two shapes Circle and Square .
- You want to incorporate colors, Red and Blue
- You create four combinations such as BlueCircle and RedSquare and so on.




*Number of class combinations grows in geometric progression.*

# Colored Shapes

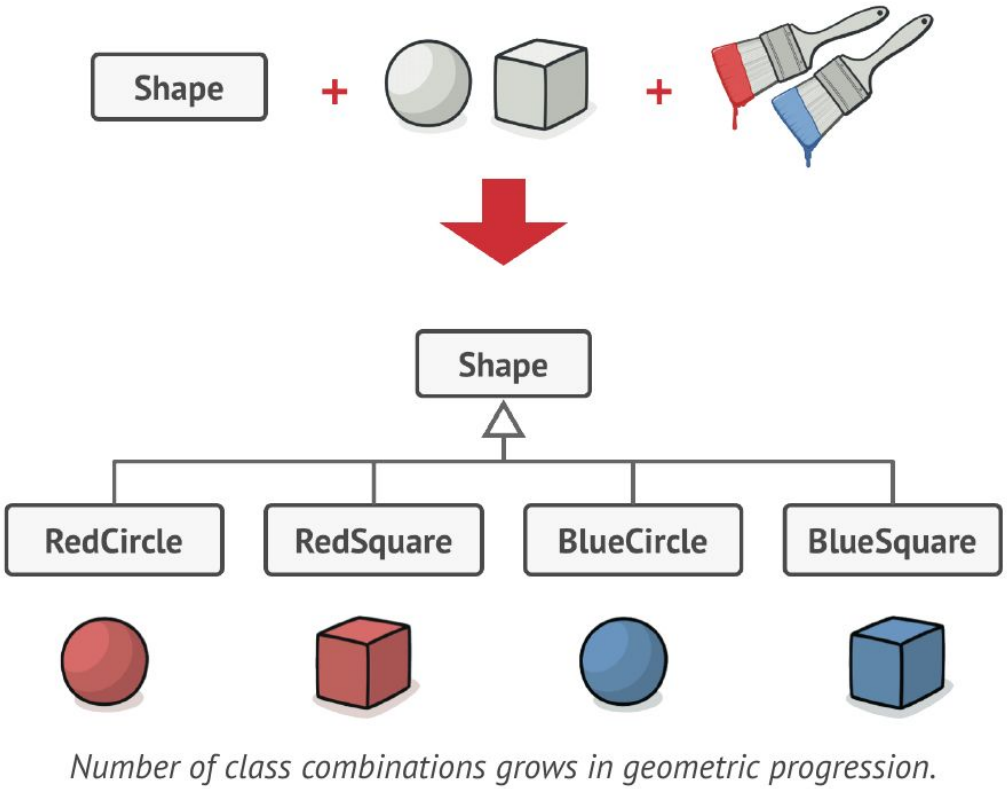
- You have two shapes Circle and Square .
- You want to incorporate colors, Red and Blue
- You create four combinations such as BlueCircle and RedSquare and so on.
- New color (green) or shape (triangle)?




*Number of class combinations grows in geometric progression.*

# Colored Shapes

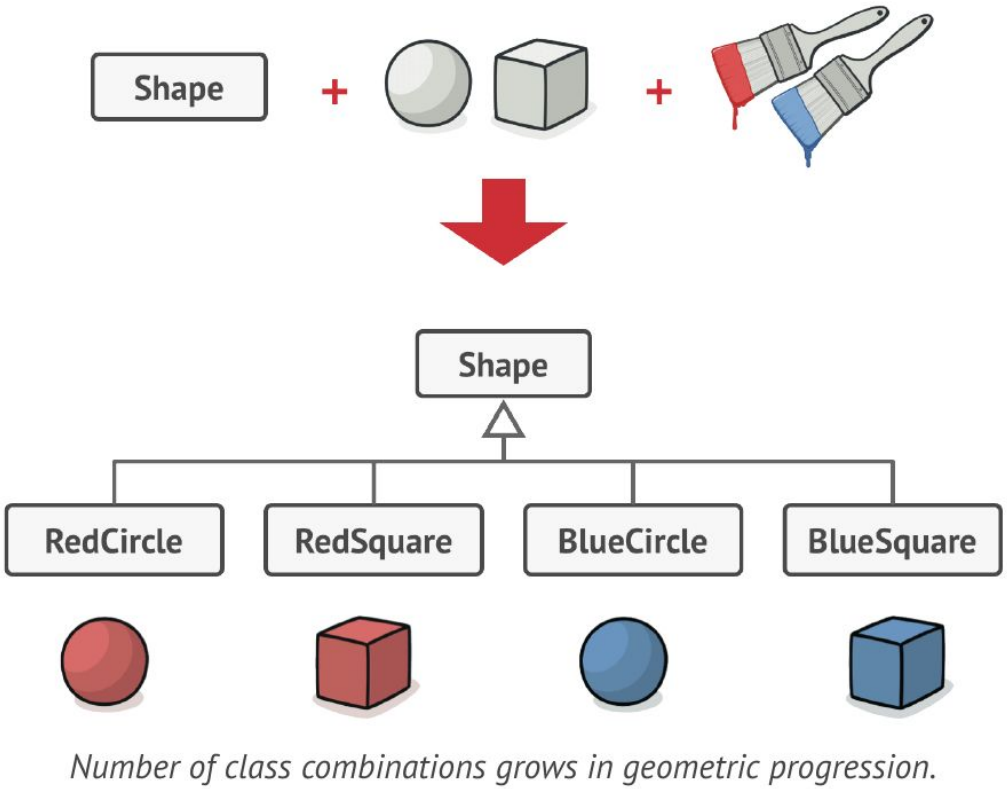
- You have two shapes Circle and Square .
- You want to incorporate colors, Red and Blue
- You create four combinations such as BlueCircle and RedSquare and so on.
- New color (green) or shape (triangle)?
- Hierarchy will grow exponentially




*Number of class combinations grows in geometric progression.*

# Colored Shapes

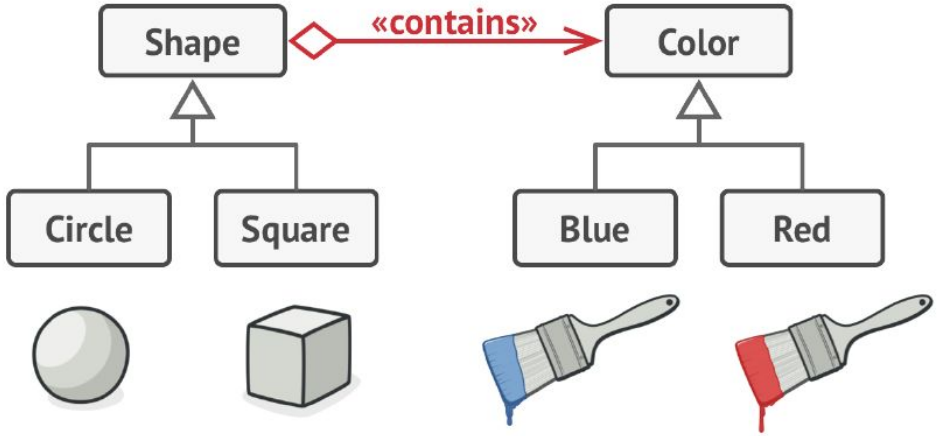
- You have two shapes Circle and Square .
- You want to incorporate colors, Red and Blue
- You create four combinations such as BlueCircle and RedSquare and so on.
- New color (green) or shape (triangle)?
- Hierarchy will grow exponentially
- Why? As things extend in two **independent** dimensions




*Number of class combinations grows in geometric progression.*

# Two Hierarchies

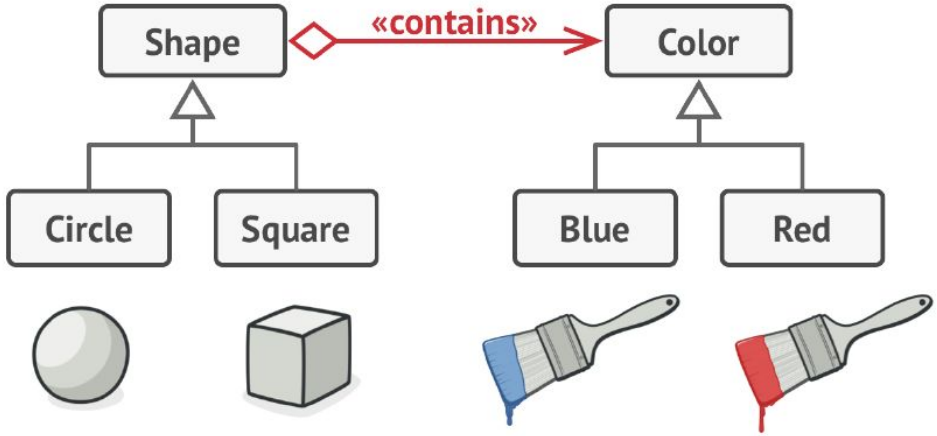
- Two independent dimensions: form and color




*You can prevent the explosion of a class hierarchy by transforming it into several related hierarchies.*

# Two Hierarchies

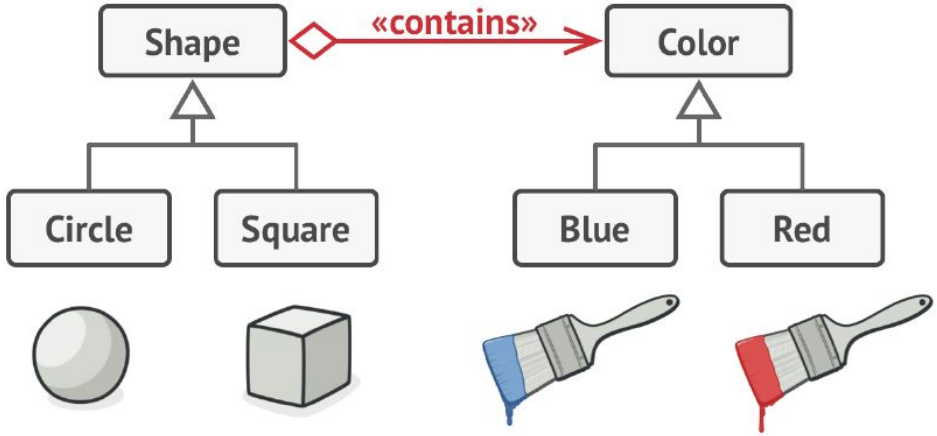
- Two independent dimensions: form and color
- From inheritance to composition




*You can prevent the explosion of a class hierarchy by transforming it into several related hierarchies.*

# Two Hierarchies

- Two independent dimensions: form and color
- From inheritance to composition
- `Shape` delegates color related work to `Color`




*You can prevent the explosion of a class hierarchy by transforming it into several related hierarchies.*

# Some academic terms

The GoF book uses two terms Abstraction & Implementation

- **Abstraction:** high-level control layer for some entity. (also called *interface*)
- This layer isn't supposed to do any **real work** on its own.
- It should **delegate** the work to the **implementation** layer (also called *platform*)

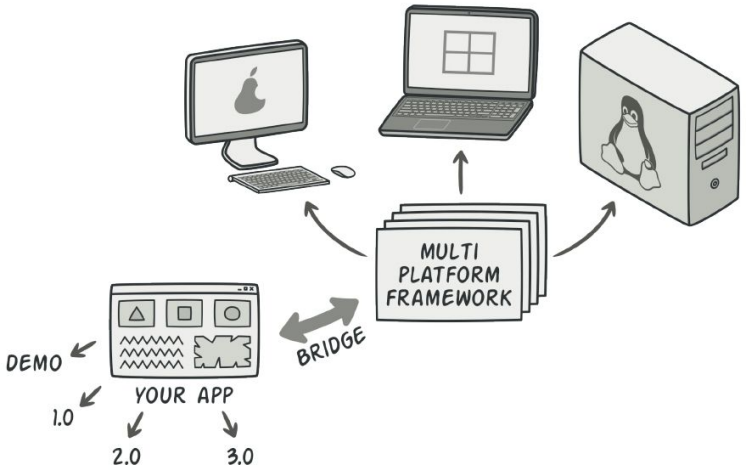
# Some academic terms

The GoF book uses two terms Abstraction & Implementation

- **Abstraction:** high-level control layer for some entity. (also called *interface*)
- This layer isn't supposed to do any **real work** on its own.
- It should **delegate** the work to the **implementation** layer (also called *platform*)

**Real application:**

- **Abstraction:** GUI (regular customer, admins)
- **Implementation:** APIs for Linux, Windows, MacOS etc.




*One of the ways to structure a cross-platform application.*

# Some academic terms

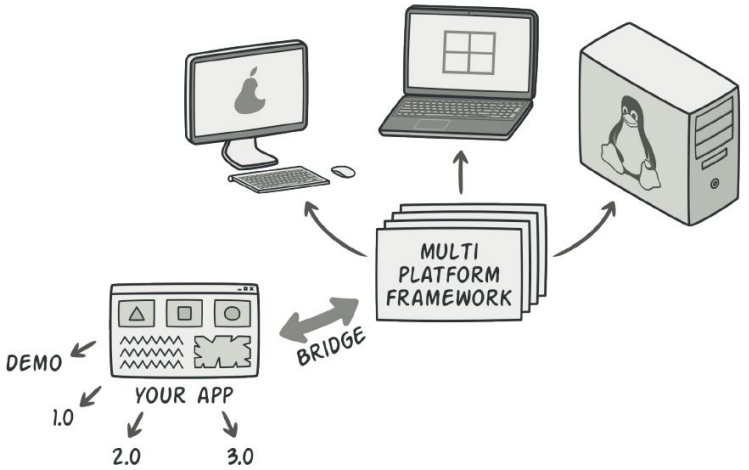
The GoF book uses two terms Abstraction & Implementation

- **Abstraction:** high-level control layer for some entity. (also called *interface*)
- This layer isn't supposed to do any **real work** on its own.
- It should **delegate** the work to the **implementation** layer (also called *platform*)

**Real application:**

- **Abstraction:** GUI (regular customer, admins)
- **Implementation:** APIs for Linux, Windows, MacOS etc.

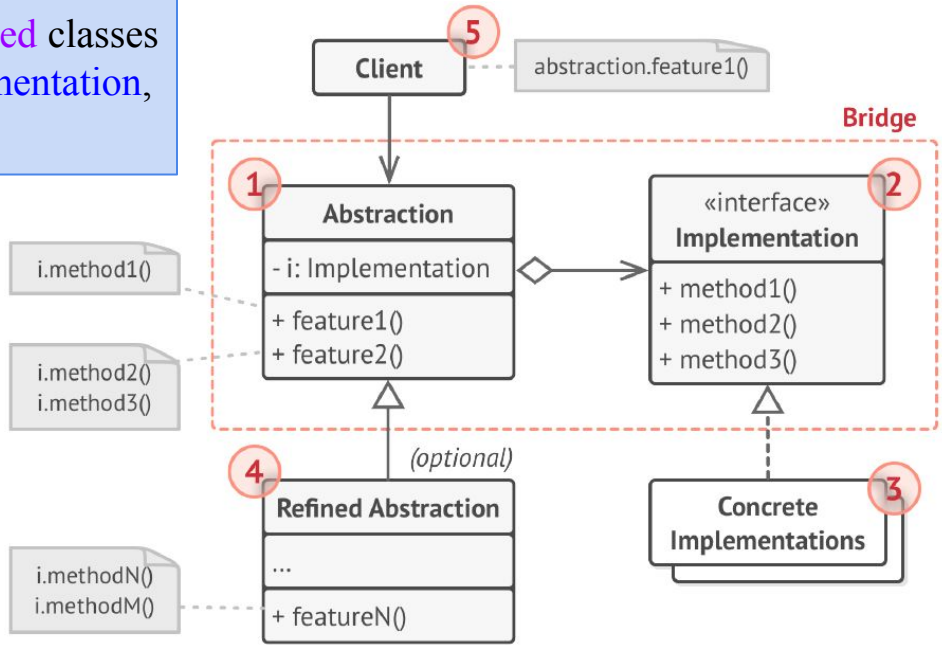
- ❏ Abstraction object controls the appearance of the app, delegates the actual work to the linked implementation object.
- ❏ Implementations are interchangeable
- ❏ Enables same GUI to work under different OS.




*One of the ways to structure a cross-platform application.*

# Bridge Pattern Defined

Bridge lets you split a **large class** or a set of closely related classes into *two separate hierarchies*, abstraction and implementation, which can be developed **independently** of each other.



# Bridge Pattern Defined

1. The **Abstraction** provides high-level control logic. It relies on the implementation object to do the actual low-level work.

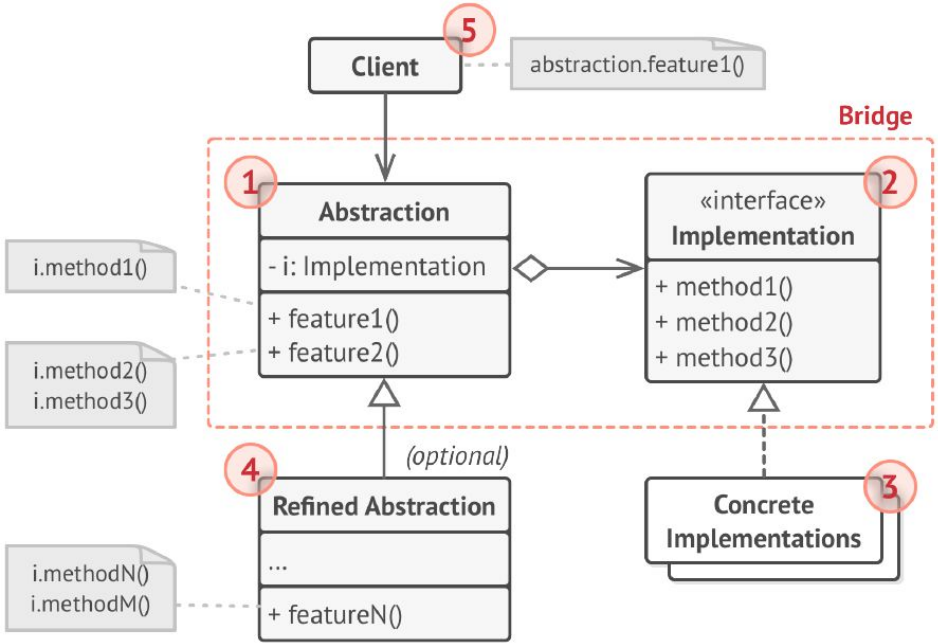
2. The **Implementation** declares the interface that's common for all concrete implementations. An abstraction can only communicate with an implementation object via methods that are declared here.

   The abstraction may list the same methods as the implementation, but usually the abstraction declares some complex behaviors that rely on a wide variety of primitive operations declared by the implementation.

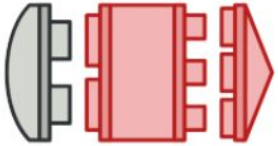
3. **Concrete Implementations** contain platform-specific code.

4. **Refined Abstractions** provide variants of control logic. Like their parent, they work with different implementations via the general implementation interface.

5. Usually, the **Client** is only interested in working with the abstraction. However, it's the client's job to link the abstraction object with one of the implementation objects.

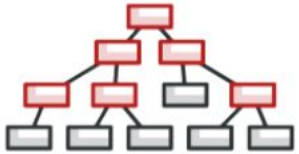


# Summary



## Adapter

Provides a unified interface that allows objects with incompatible interfaces to collaborate.

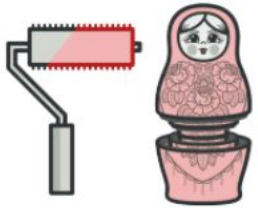


## Composite

Lets you compose objects into tree structures and then work with these structures as if they were individual objects.

## Bridge

Lets you split a large class or a set of closely related classes into two separate hierarchies—abstraction and implementation—which can be developed independently of each other.



## Decorator

Lets you attach new behaviors to objects by placing these objects inside special wrapper objects that contain the behaviors.

# Acknowledgement


- Dive Into Design Patterns - Alexander Shvets
- Head First Design Patterns, 2nd Edition - Eric Freeman, Elisabeth Robson


DESIGN PATTERNS
DESIGN PATTERNS
EVERYWHERE
makeameme.org


Goodbye!