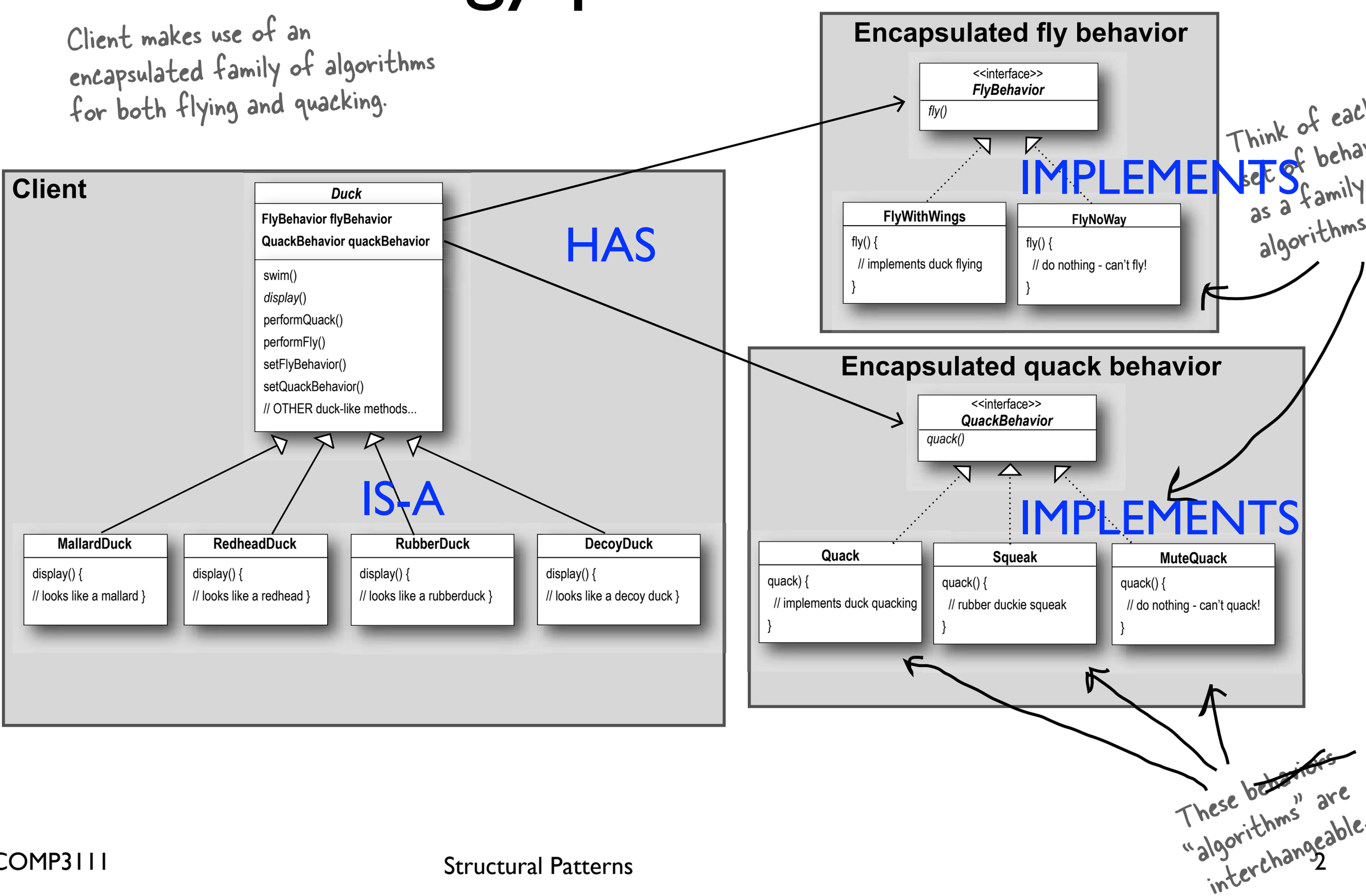
Structural Design Patterns

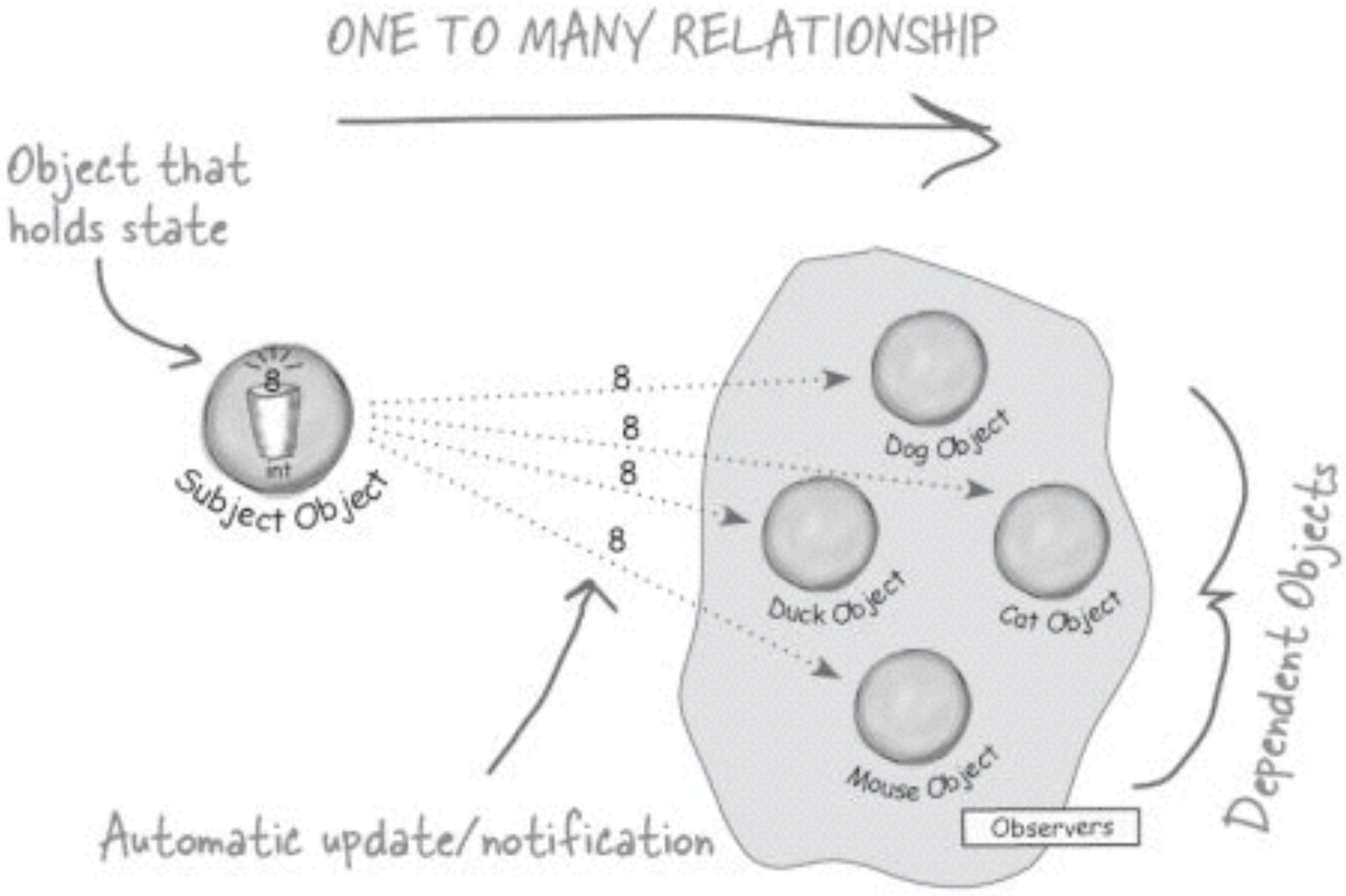
Smart use of inheritance

Strategy pattern

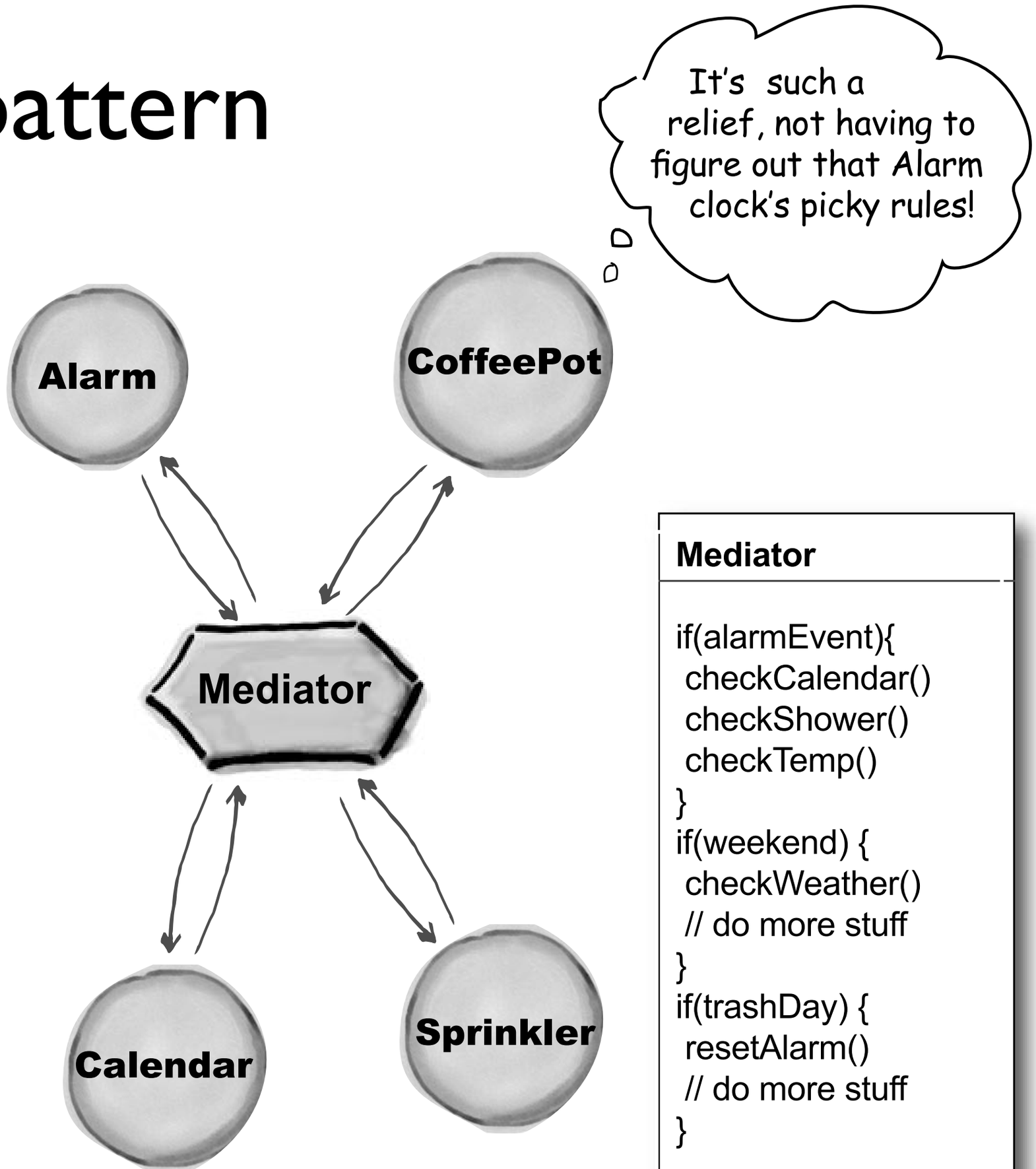
Client makes use of an encapsulated family of algorithms for both flying and quacking.



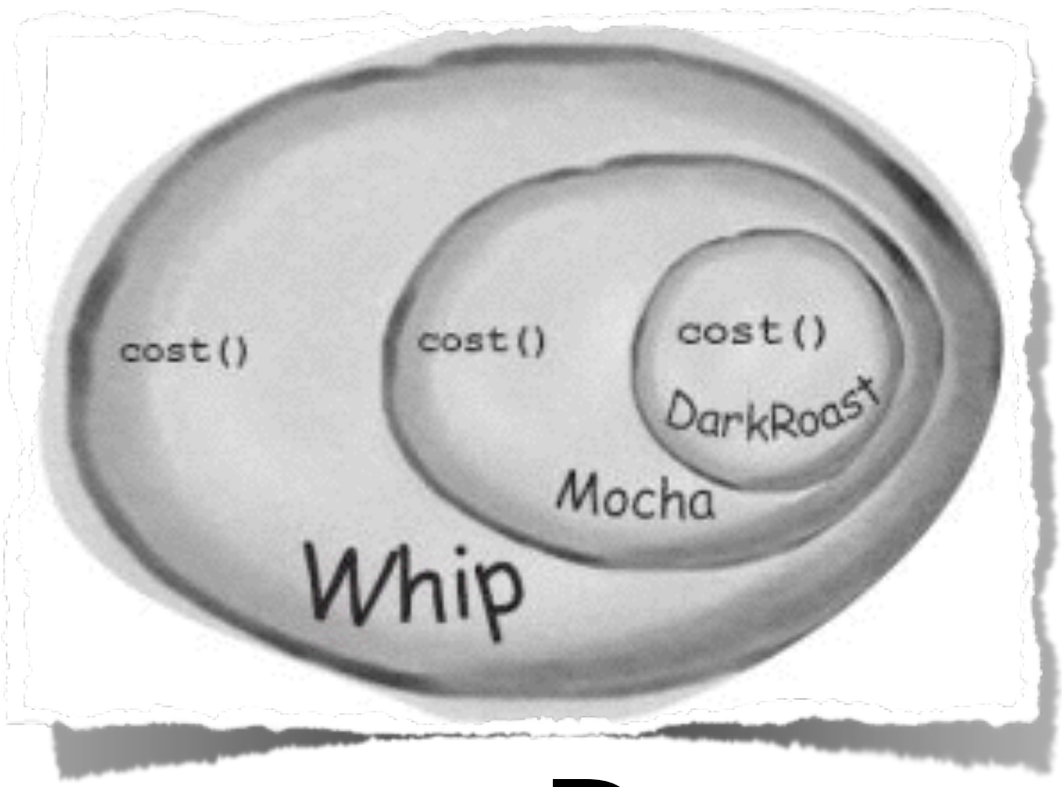
Observable


ONE TO MANY RELATIONSHIP
Object that holds state
8
int
Subject Object
8
8
8
8
Dog Object
Duck Object
Cat Object
Mouse Object
Observers
Dependent Objects
Automatic update/notification

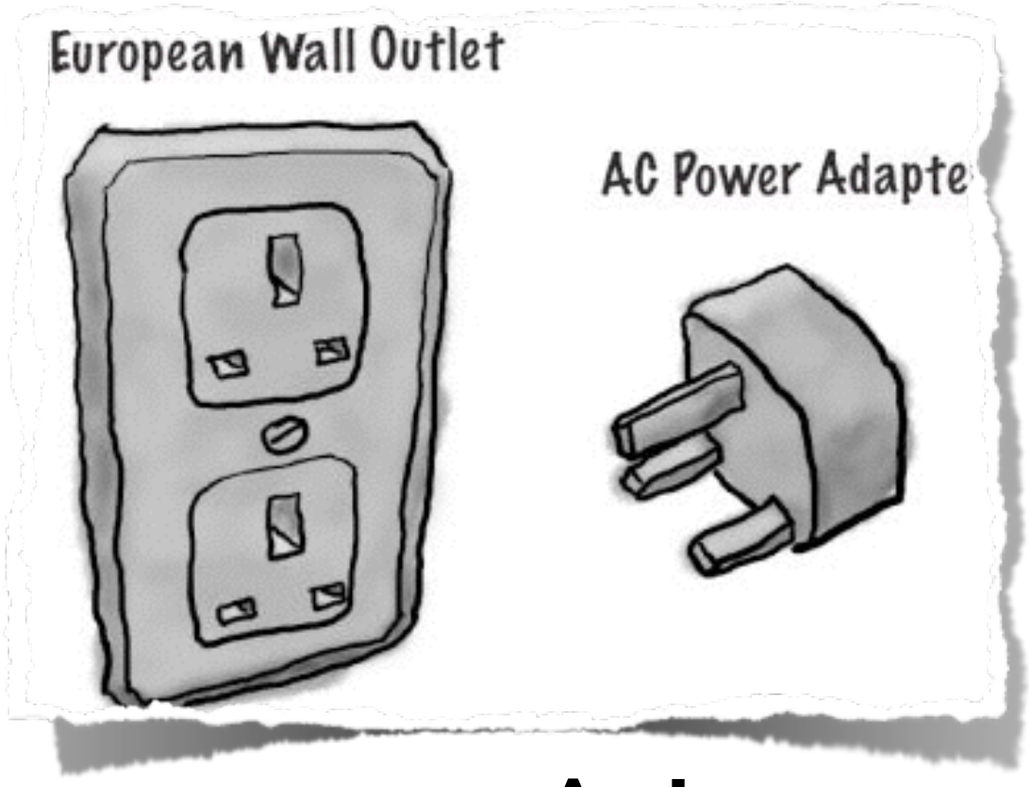
Mediator pattern in action...



Today



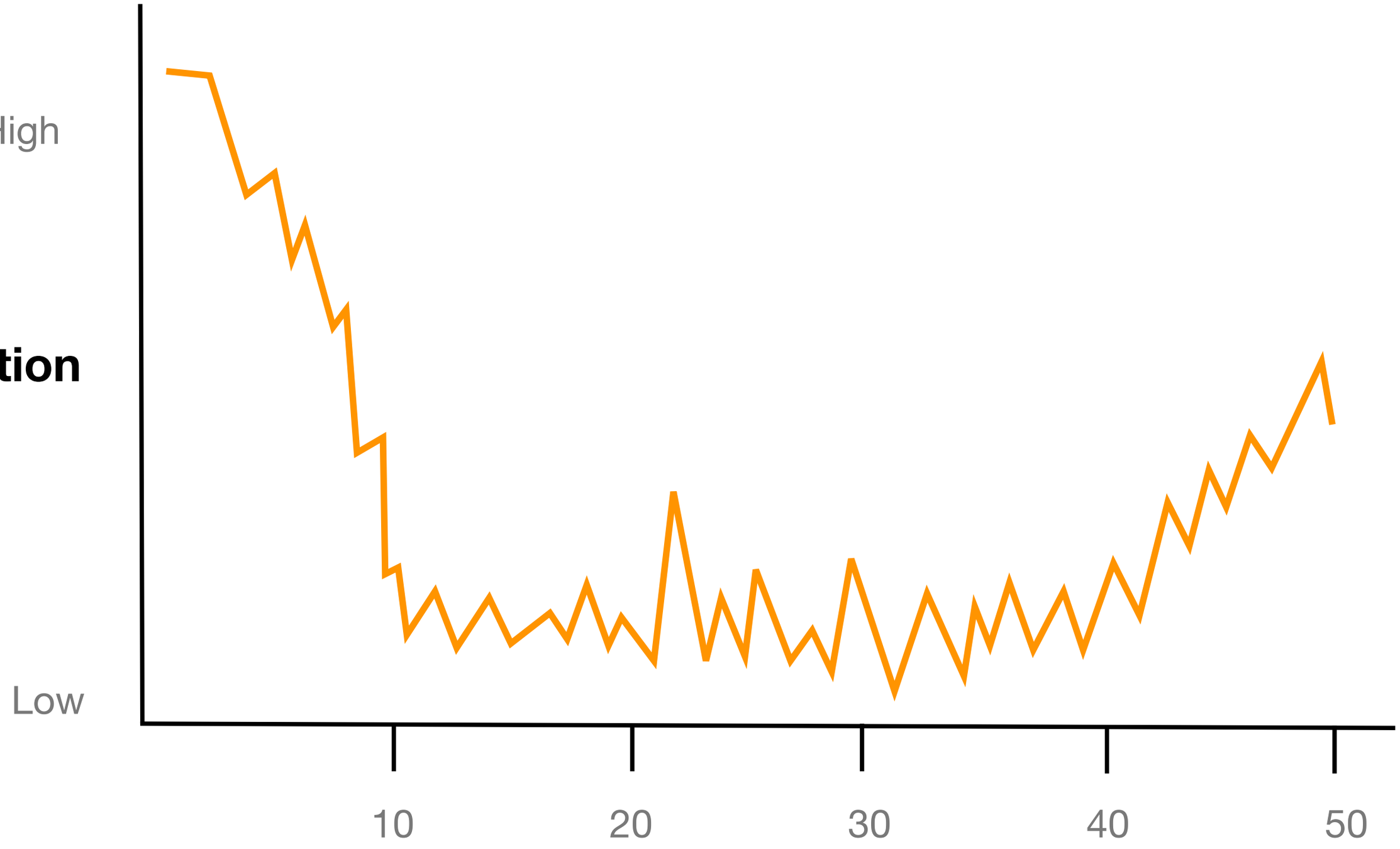
Decorator



Adapter

After 10 minutes, audience attention steadily drops.

The 10-minute rule



Source: www.brainrules.net/attention

Let's get coffee

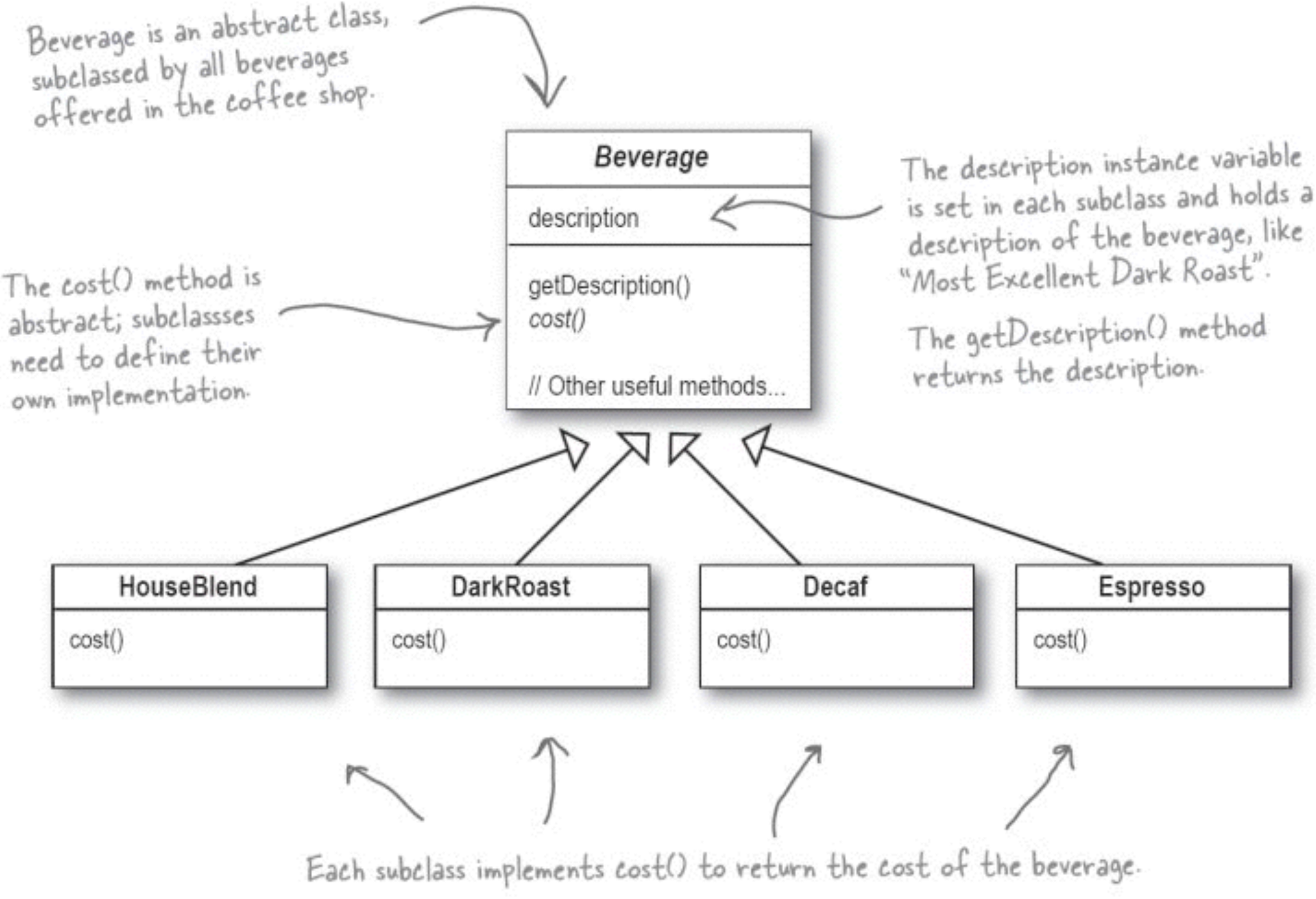
Welcome to Starbuzz Coffee

Starbuzz Coffee has made a name for itself as the fastest growing coffee shop around. If you've seen one on your local corner, look across the street; you'll see another one.

Because they've grown so quickly, they're scrambling to update their ordering systems to match their beverage offerings.

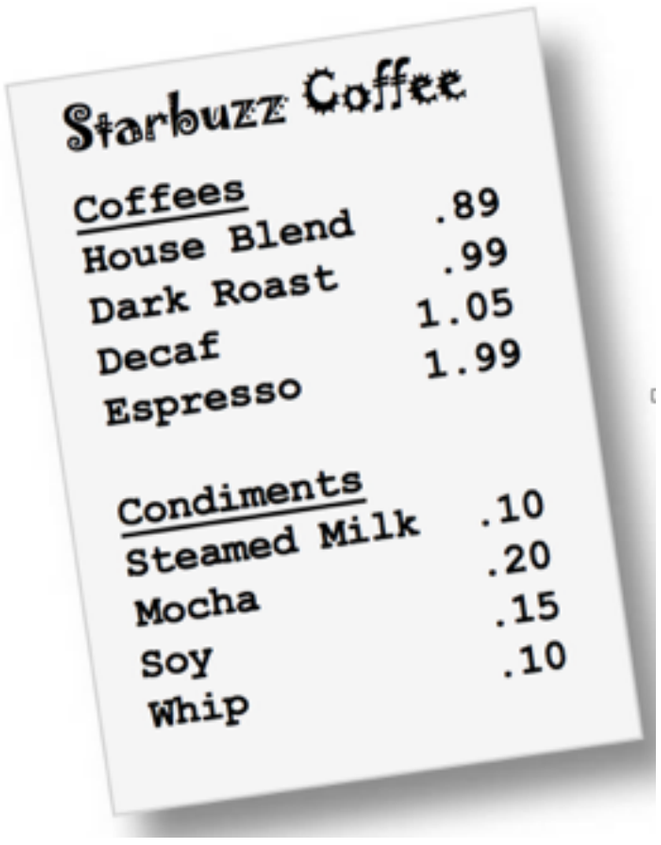
When they first went into business they designed their classes like this...

Initial design of Starbuzz

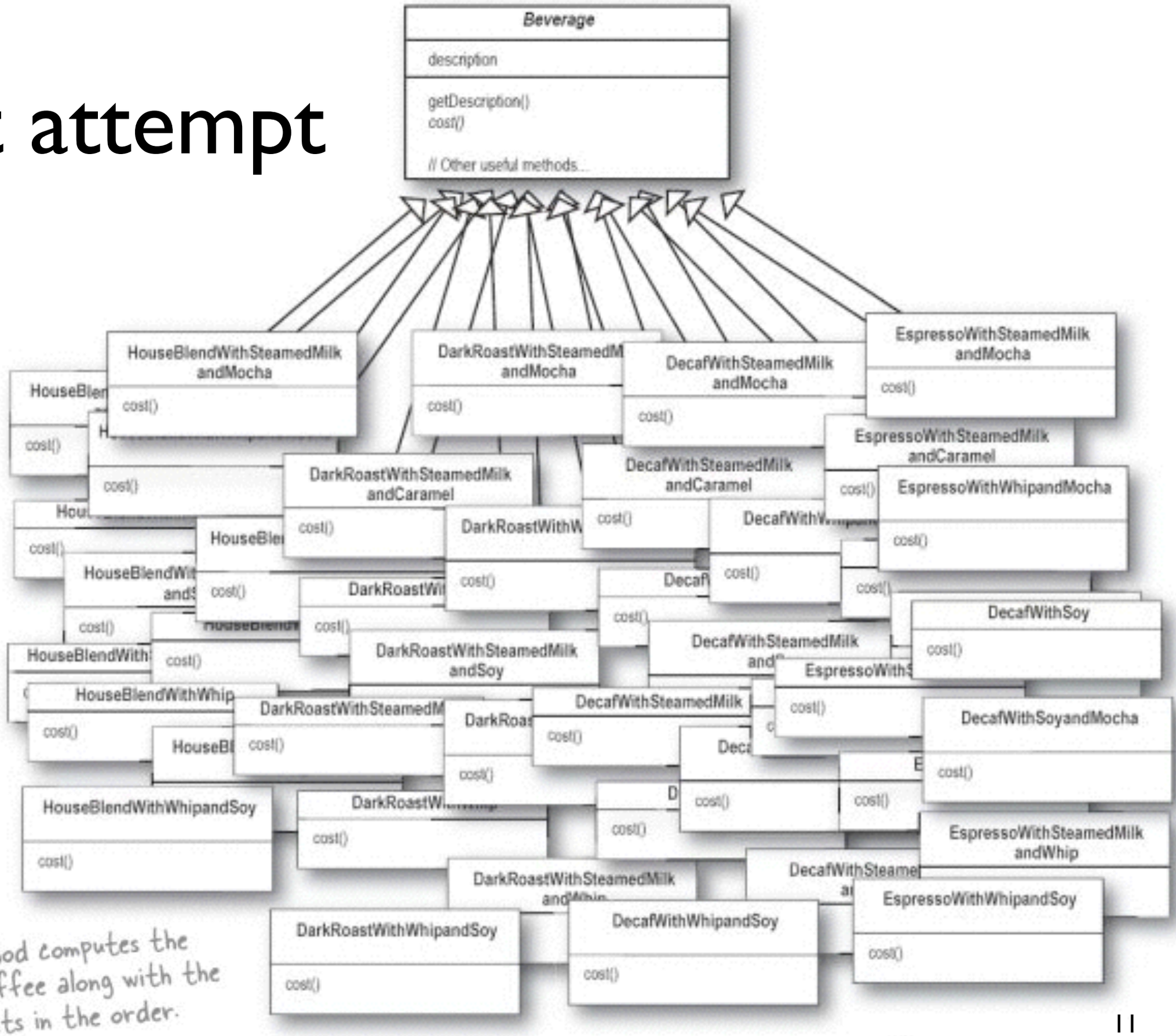


Introducing condiments

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

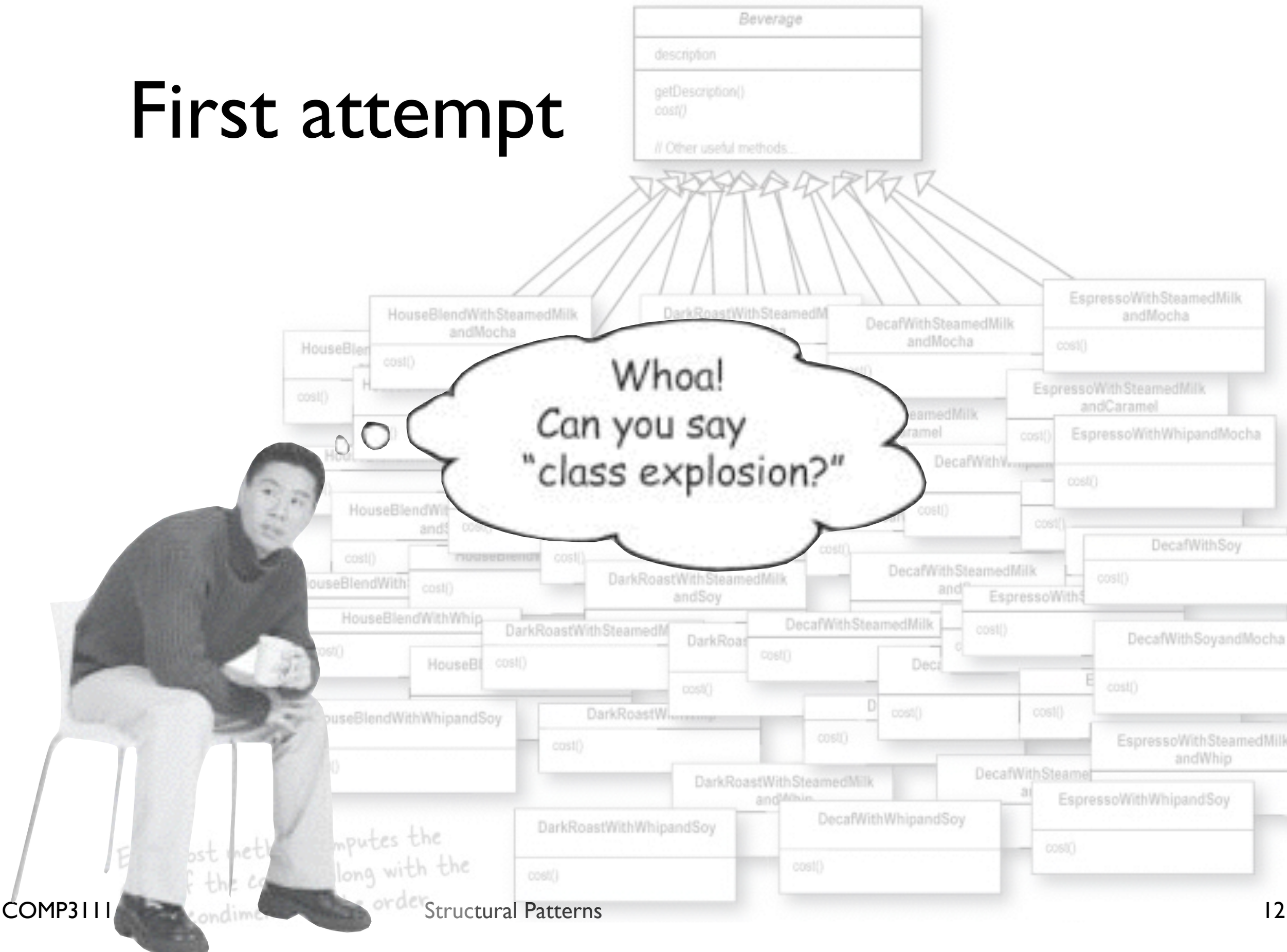


First attempt



Each cost method computes the cost of the coffee along with the other condiments in the order.

First attempt



Challenges

It's pretty obvious that Starbuzz has created a maintenance nightmare for themselves. What happens when the price of milk goes up? What do they do when they add a new caramel topping?

Thinking beyond the maintenance problem, which of the design principles that we've covered so far are they violating?

Hint: they're violating two of them in a big way!

Design Principles

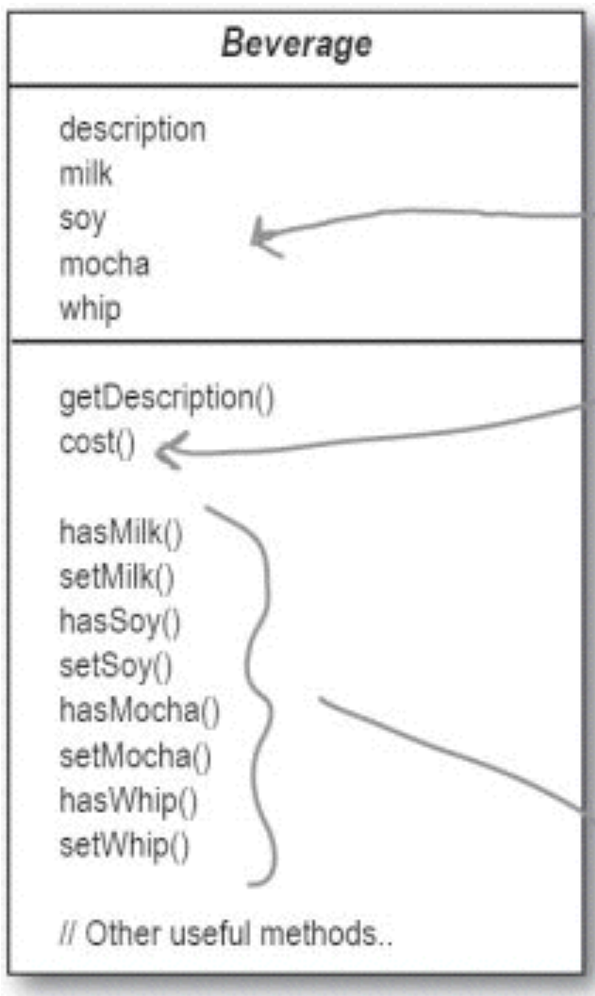
> ***Design Principle***
>
> *Identify the aspects of your application that vary and separate them from what stays the same.*

> ***Design Principle***
>
> *Program to an interface, not an implementation.*

> ***Design Principle***
>
> *Favor composition over inheritance.*

Second attempt

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?



New boolean values for each condiment.

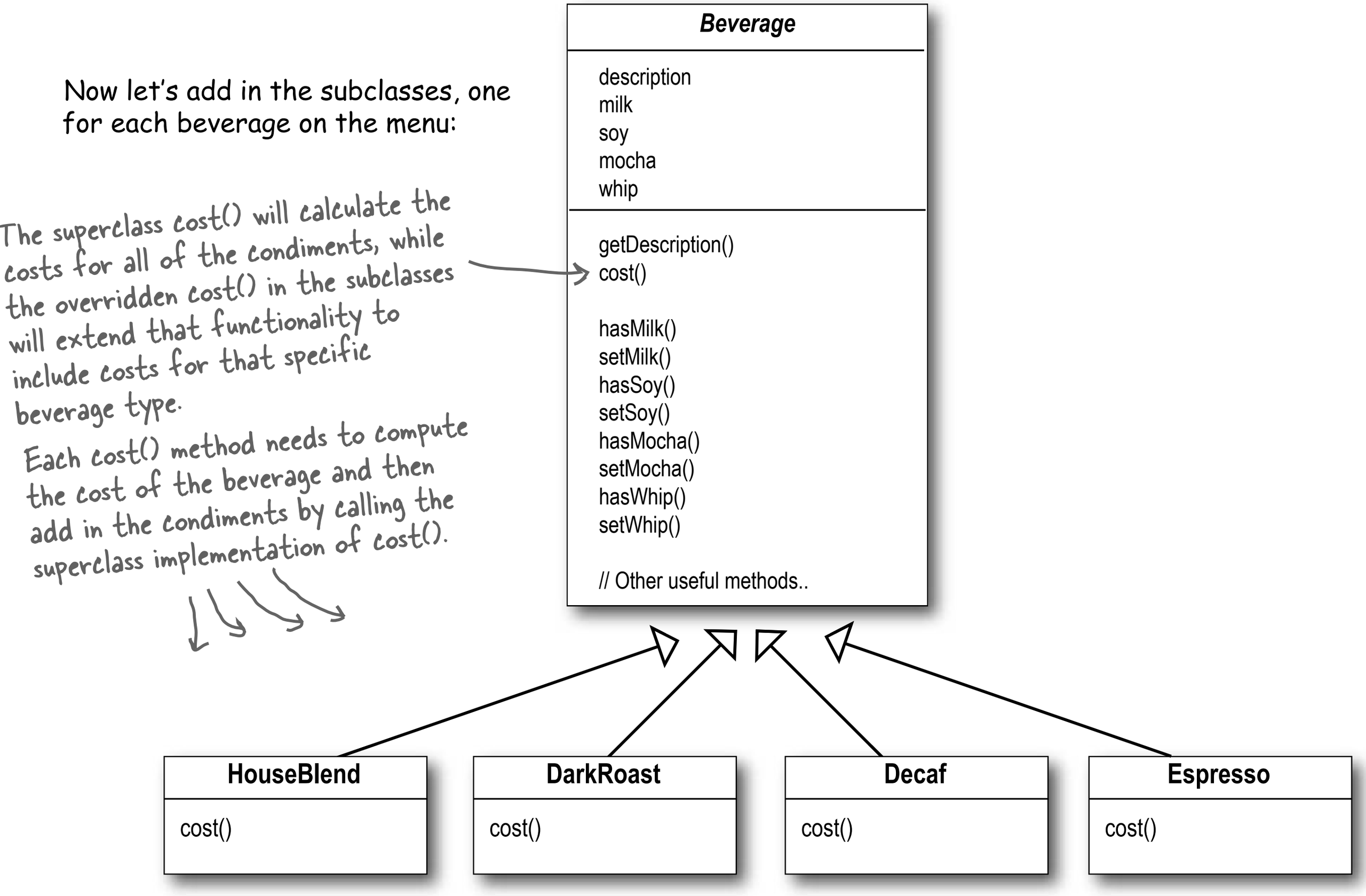
Now we'll implement cost() in Beverage (instead of keeping it abstract), so that it can calculate the costs associated with the condiments for a particular beverage instance. Subclasses will still override cost(), but they will also invoke the super version so that they can calculate the total cost of the basic beverage plus the costs of the added condiments.

These get and set the boolean values for the condiments.

Now let's add in the subclasses, one for each beverage on the menu:

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().



Now let's add in t
for each beverage

will calcu
condiments, while
in the subclasses
ctionality to
t specific

needs to compute
verage and then
nts by calling the
tation of cost().

The
cos
the
wil
inc
be
E
t
a
s

See, five classes total. This is definitely the way to go.

I'm not so sure; I can see some potential problems with this approach by thinking about how the design might need to change in the future.

getDescription()

useBlend

DarkRoast
cost()

Decaf
cost()

Sharpen your pencil

What requirements or other factors might change that will impact this design?

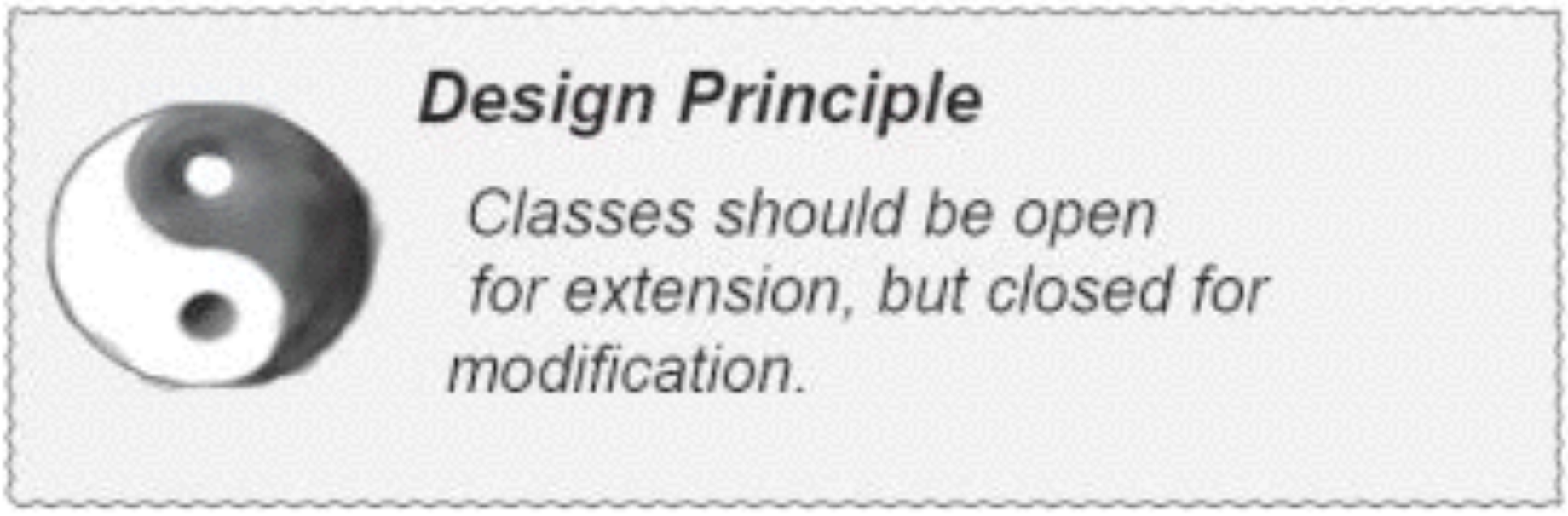
Price changes for condiments will force us to alter existing code.

New condiments will force us to add new methods and alter the cost method in the superclass.

We may have new beverages. For some of these beverages (iced tea?), the condiments may not be appropriate, yet the Tea subclass will still inherit methods like hasWhip().

What if a customer wants a double mocha?

The open-closed principle



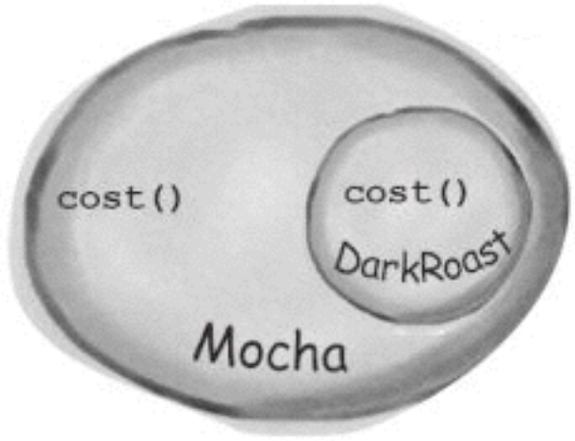
Allow to incorporate new behavior without modifying existing code.

Design Considerations

- Separation of stable things from unstable things
  - All objects are homogenous. They are just different types of coffee
  - Variations are unpredictable and immersive.
- Design decisions
  - We want to use inheritance of a coffee interface to stick to the "programming to interface principle"
  - We want to compose different "coffee configurations".
  - We want inheritance and composition.

Meet the Decorator Pattern

1 Take a DarkRoast object

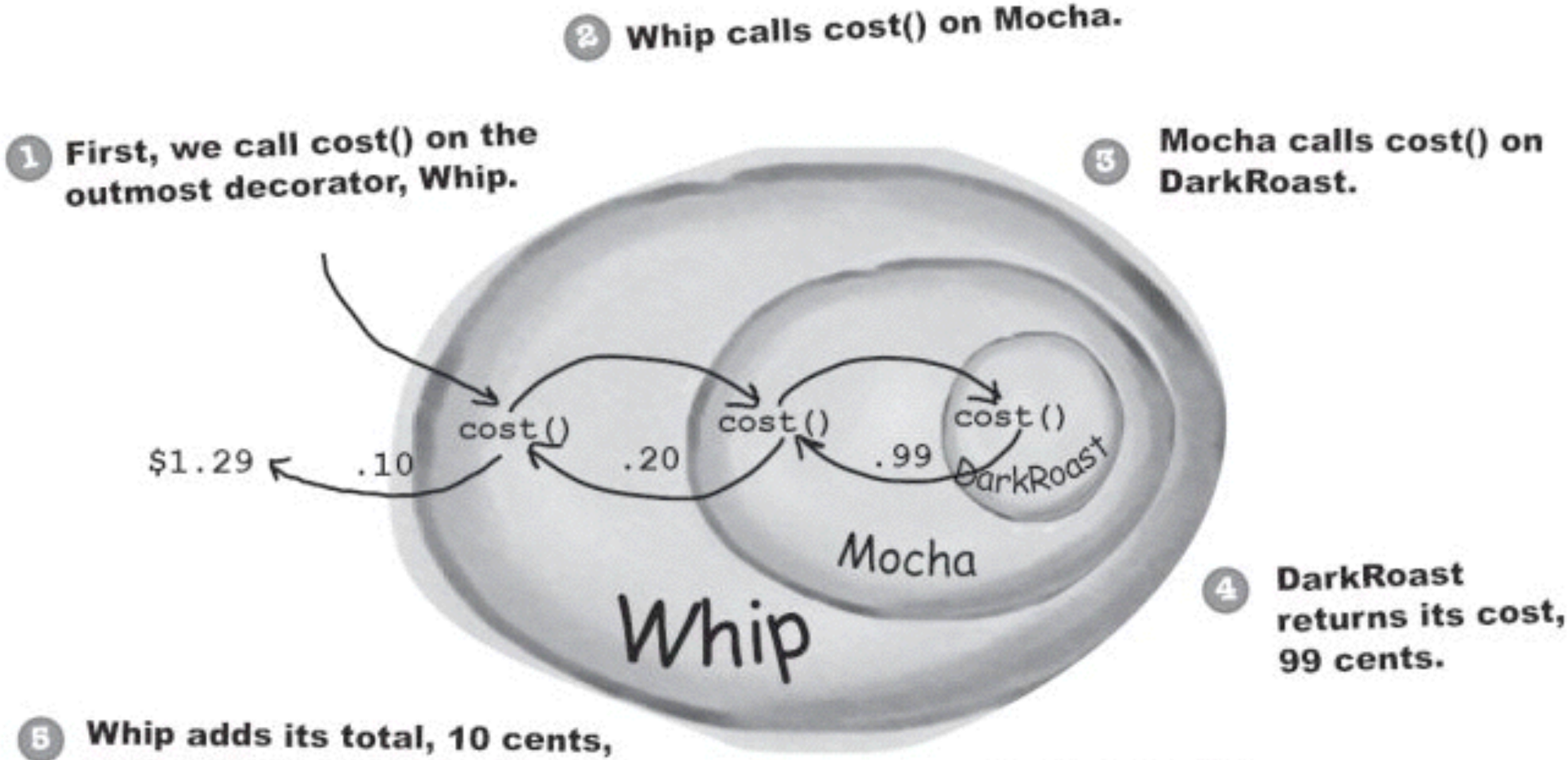


2 Decorate it with a Mocha object

3 Decorate it with a Whip object

4 Call the cost() method and rely on delegation to add on the condiment costs

Meet the Decorator Pattern



The definition

The Decorator Pattern attaches additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

Each component can be used on its own, or wrapped by a decorator.

component

Component
methodA()
methodB()
// other methods

The ConcreteComponent is the object we're going to dynamically add new behavior to. It extends Component.

Each decorator HAS-A (wraps) a component, which means the decorator has an instance variable that holds a reference to a component.

ConcreteComponent
methodA()
methodB()
// other methods

Decorator
methodA()
methodB()
// other methods

Decorators implement the same interface or abstract class as the component they are going to decorate.

ConcereteDecoratorA
Component wrappedObj
methodA()
methodB()
newBehavior()
// other methods

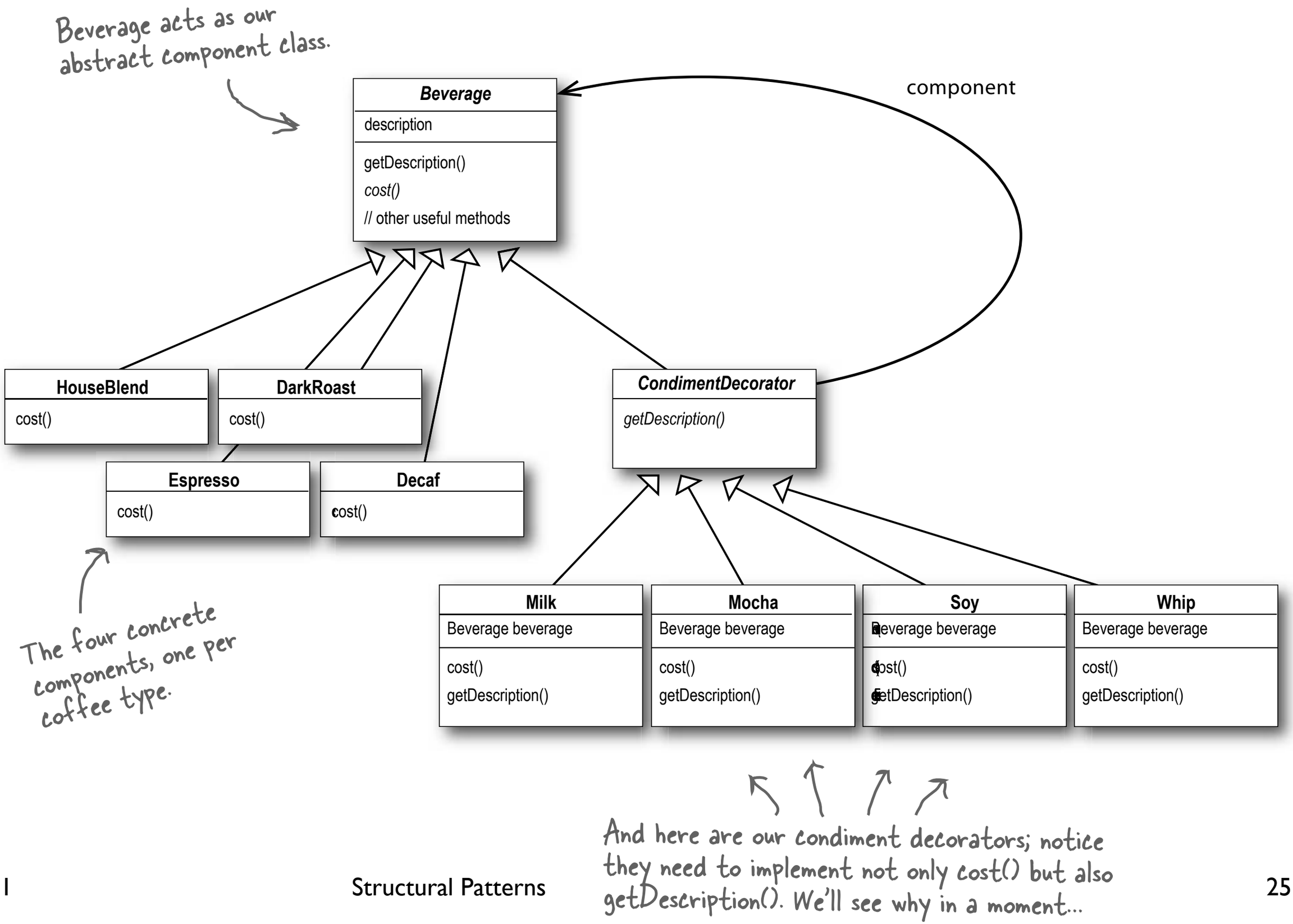
ConcereteDecoratorB
Component wrappedObj
Object newState
methodA()
methodB()
// other methods

The ConcreteDecorator has an instance variable for the thing it decorate (the Component the Decorator wraps).

Decorators can extend the state of the component.

Decorators can add new methods; however, new behavior is typically added by doing computation before or after an existing method in the component.

Decorating our beverages



Writing the Starbuzz code

It's time to whip this design into some real code.

Let's start with the Beverage class, which doesn't need to change from Starbuzz's original design. Let's take a look:

```
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.

Coding beverages

Now that we've got our base classes out of the way, let's implement some beverages. We'll start with Espresso. Remember, we need to set a description for the specific beverage and also implement the cost() method.

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.

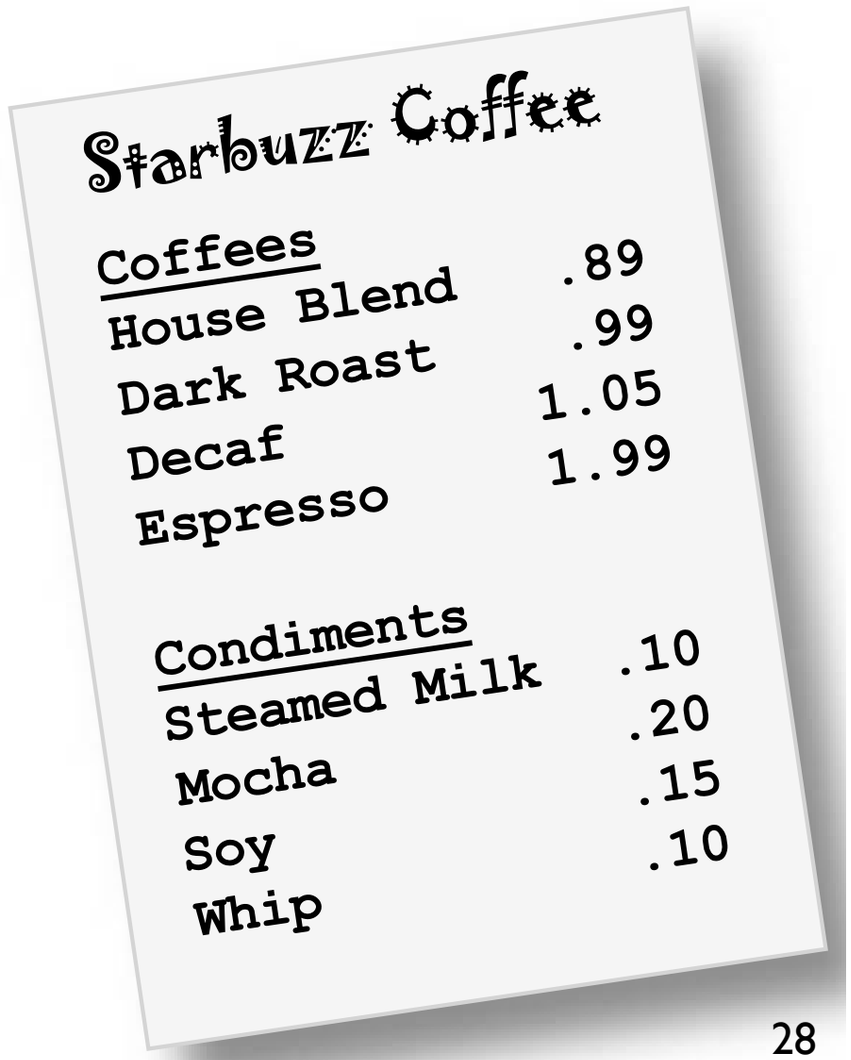
Coding beverages

Now that we've got our base classes out of the way, let's implement some beverages. We'll start with Espresso. Remember, we need to set a description for the specific beverage and also implement the cost() method.

```
public class HouseBlend extends Beverage {
    public HouseBlend() {
        description = "House Blend Coffee";
    }

    public double cost() {
        return .89;
    }
}
```

Okay, here's another Beverage. All we do is set the appropriate description, "House Blend Coffee," and then return the correct cost: 89¢.

You can create the other two Beverage classses (DarkRoast and Decaf) in exactly the same way.



Coding condiments

Mocha is a decorator, so we extend CondimentDecorator.

Remember, CondimentDecorator extends Beverage.

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

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

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

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

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

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

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

We want our description to not only include the beverage – say "Dark Roast" – but also to include each item decorating the beverage, for instance, "Dark Roast, Mocha". So we first delegate to the object we are decorating to get its description, then append ", Mocha" to that description.

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

Serving some coffees

Congratulations. It's time to sit back, order a few coffees and marvel at the flexible design you created with the Decorator Pattern.

Dark roast coffee with double Mocha and Whip

```
Beverage beverage2 = new DarkRoast();
beverage2 = new Mocha(beverage2);
beverage2 = new Mocha(beverage2);
beverage2 = new Whip(beverage2);
System.out.println(beverage2.getDescription()
        + " $" + beverage2.cost());
```

Make a DarkRoast object.

Wrap it with a Mocha.

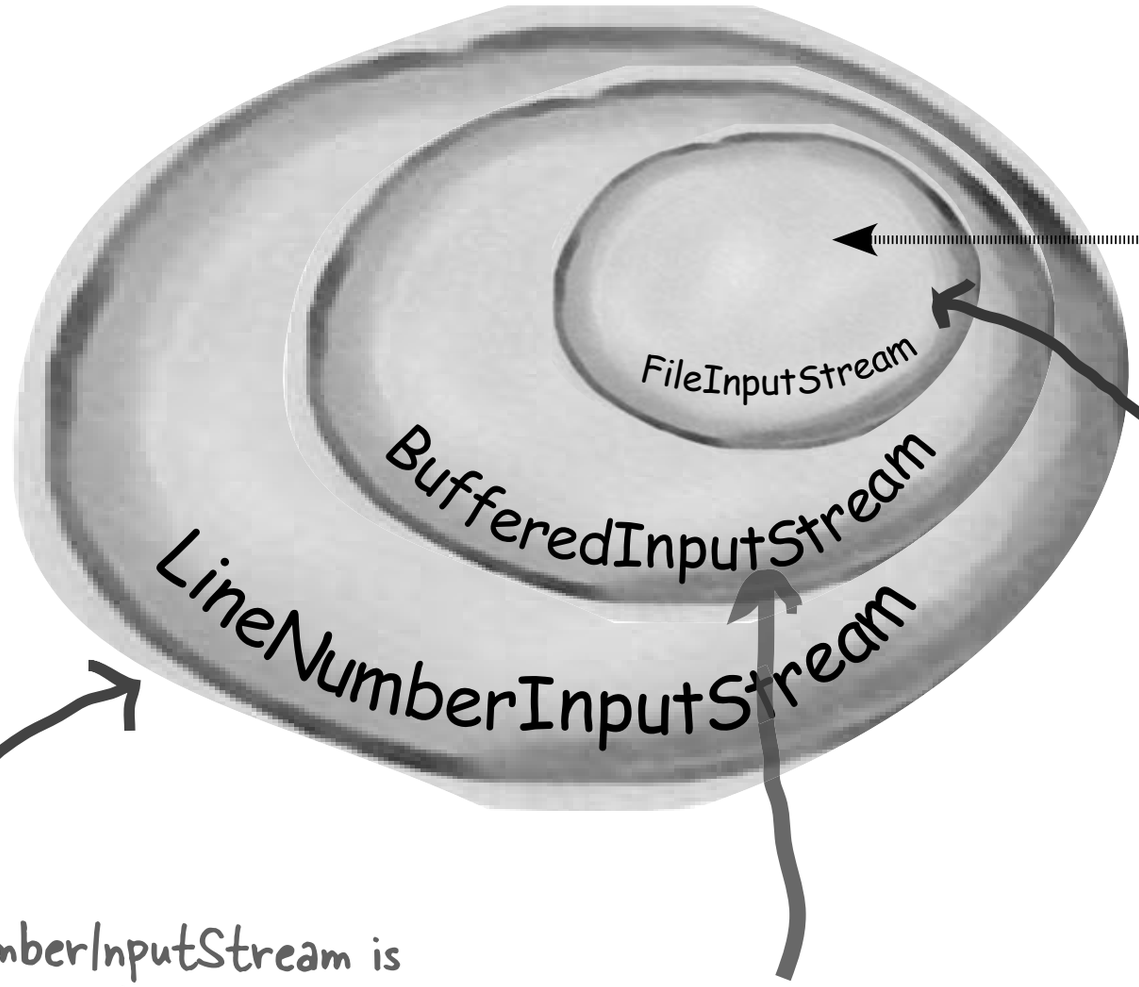
Wrap it in a second Mocha.

Wrap it in a Whip.

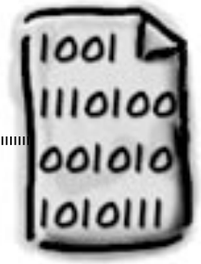
HouseBlend with Soy, Mocha, and Whip?

```
Beverage beverage3 = new HouseBlend();
beverage3 = new Soy(beverage3);
beverage3 = new Mocha(beverage3);
beverage3 = new Whip(beverage3);
System.out.println(beverage3.getDescription()
        + " $" + beverage3.cost());
```

Real world decorators: Java I/O



A text file for reading.

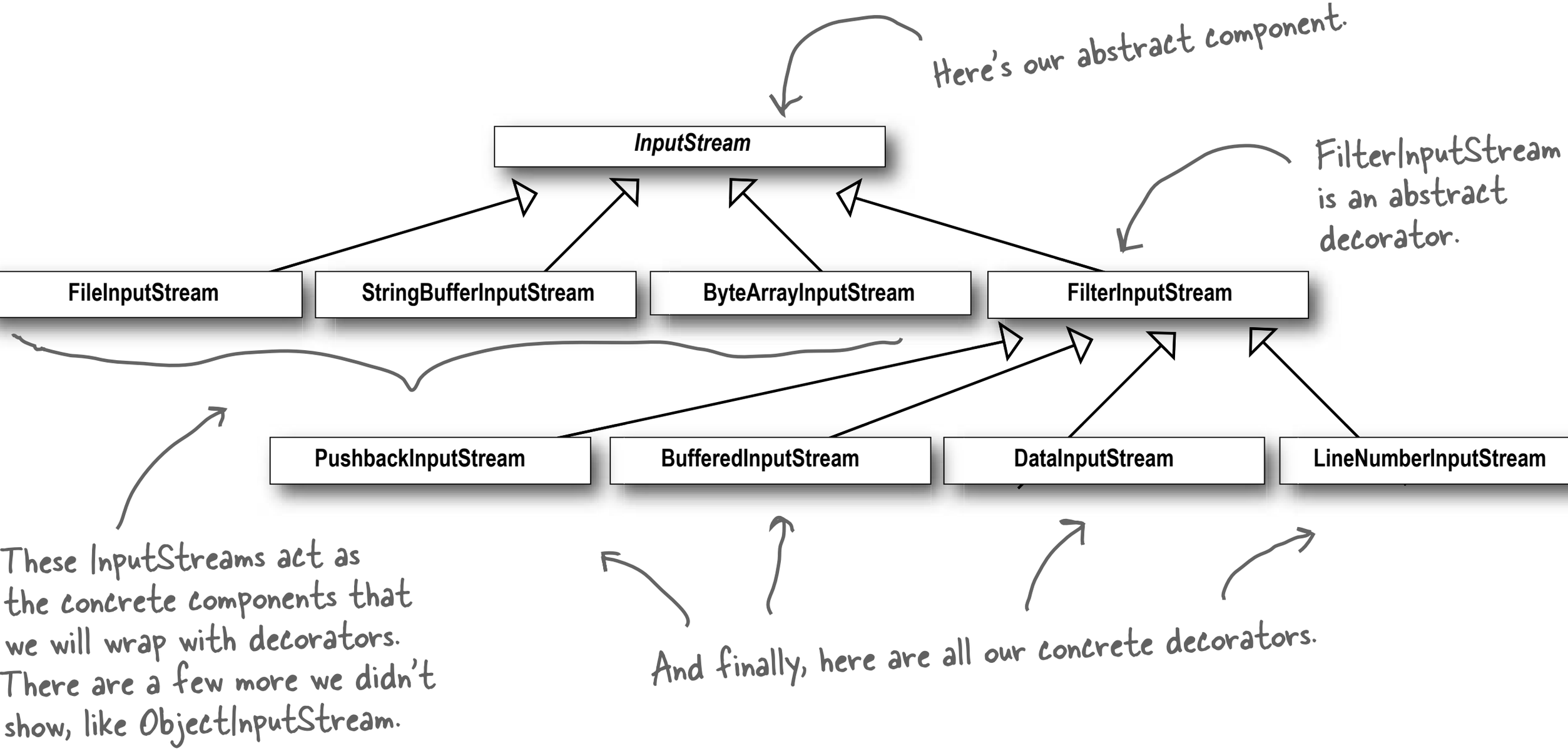


FileInputStream is the component that's being decorated The Java I/O library supplies several components, including FileInputStream, StringBufferInputStream, ByteArrayInputStream and a few others. All of these give us a base component from which to read bytes.

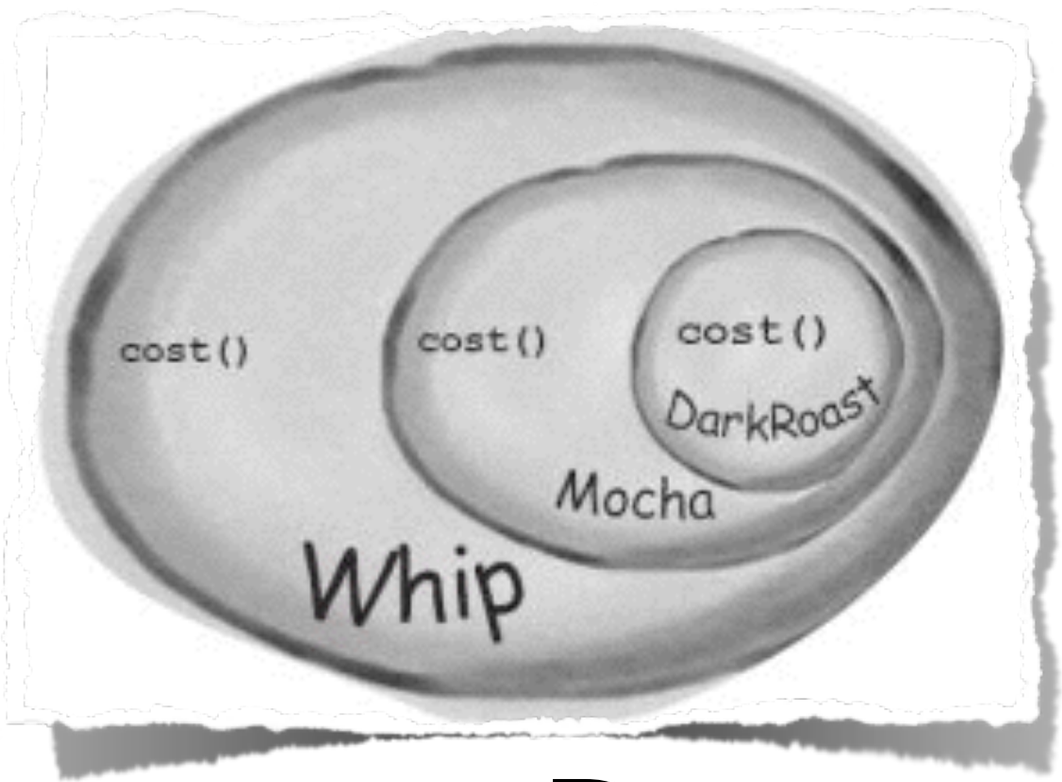
BufferedInputStream is a concrete decorator. BufferedInputStream adds behavior in two ways: it buffers input to improve performance, and also augments the interface with a new method readLine() for reading character-based input, a line at a time.

LineNumberInputStream is also a concrete decorator. It adds the ability to count the line numbers as it reads data.

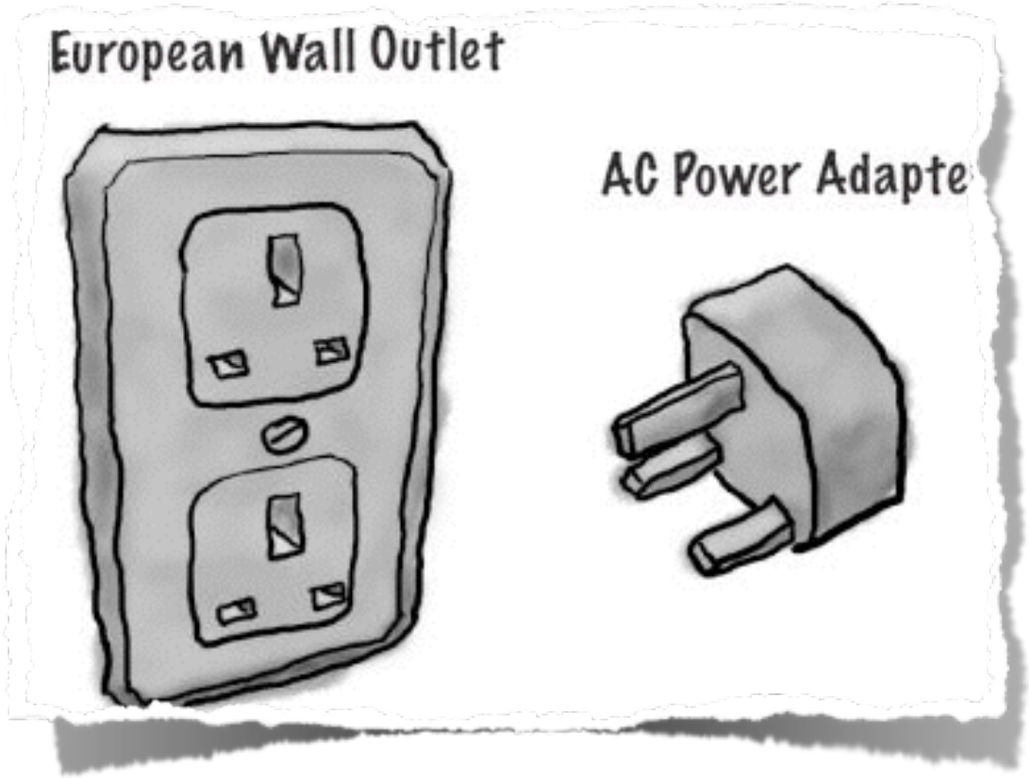
Real world decorators: Java I/O



Half to go!

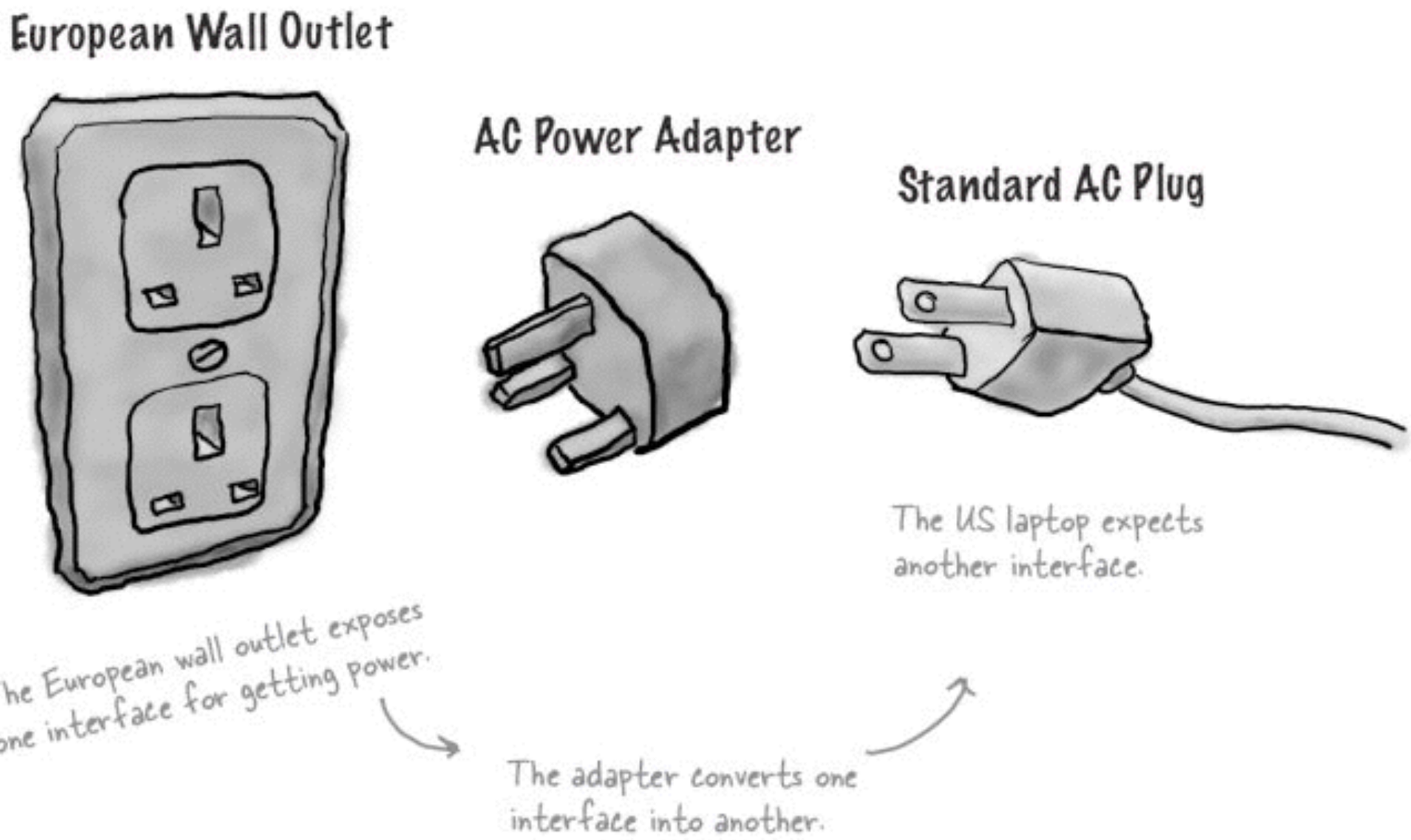


Decorator

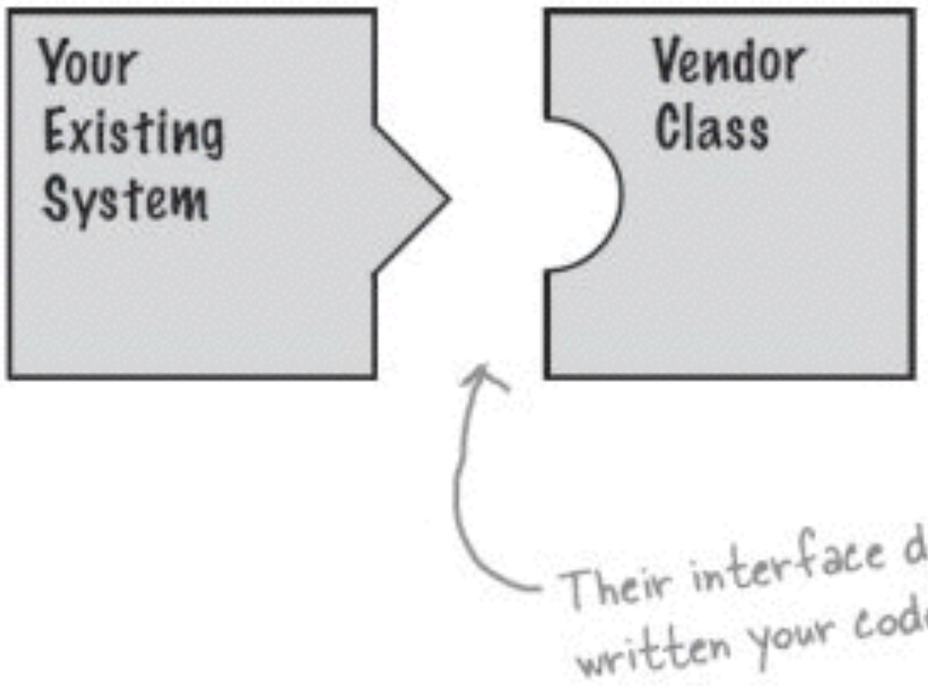


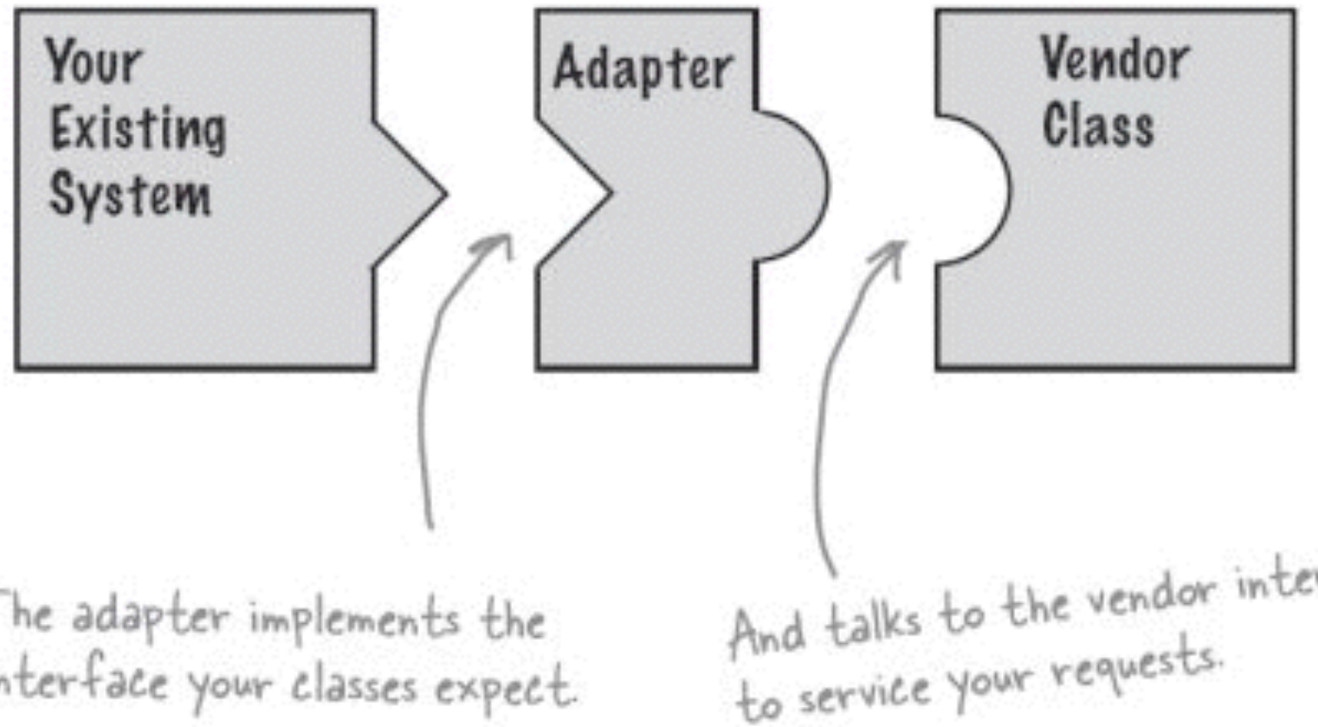
Adapter

Adapters all around us

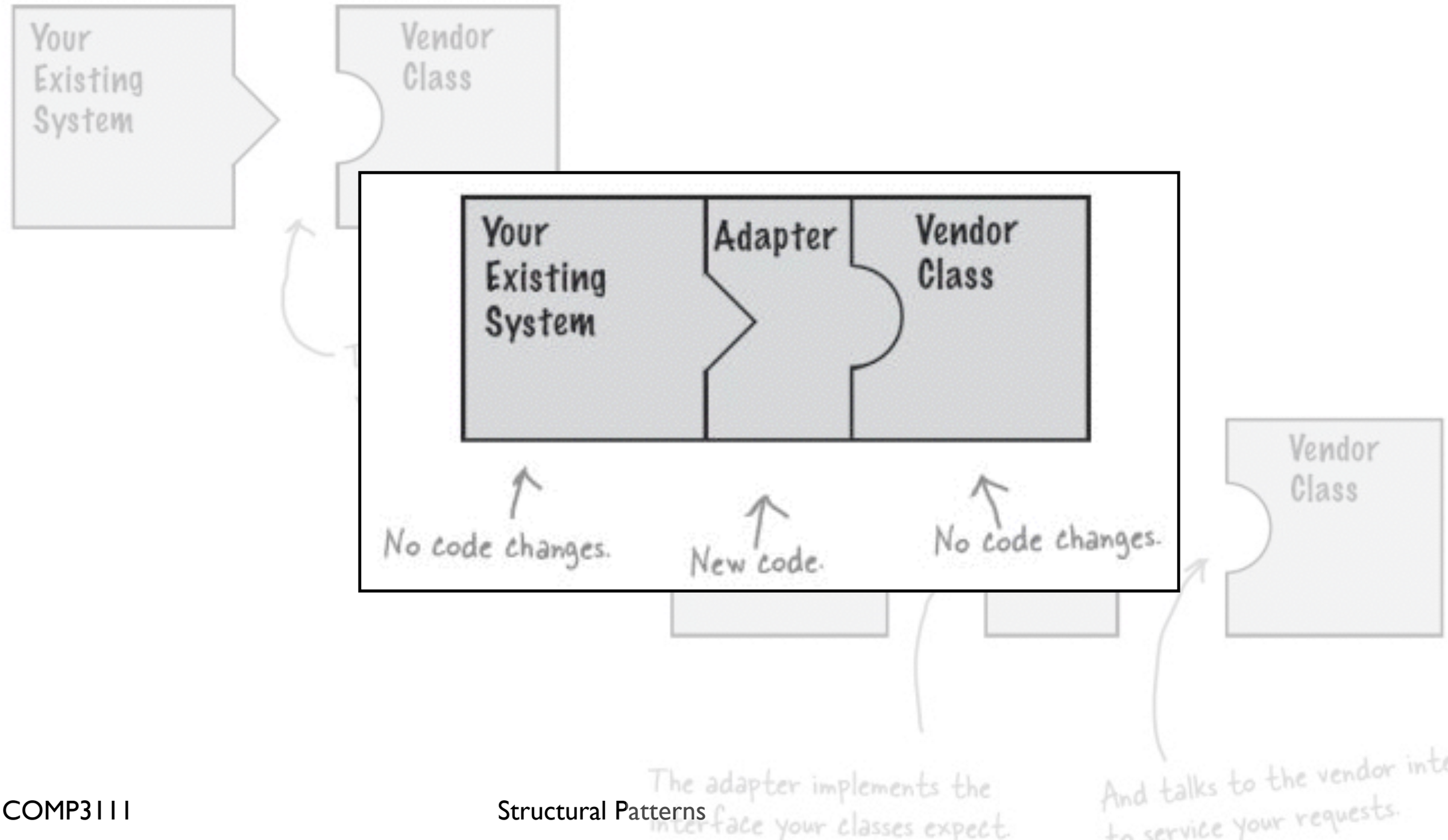


Object-oriented adapters





Object-oriented adapters



turkey adapter

If it walks like a duck and quacks like a duck, then it ~~must~~ might be a ~~duck~~ turkey wrapped with a duck adapter...

It's time to see an adapter in action. Remember our ducks from Chapter 1? Let's review a slightly simplified version of the Duck interfaces and classes:

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

This time around, our ducks implement a Duck interface that allows Ducks to quack and fly.

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

This time around, our ducks implement a Duck interface that allows Ducks to quack and fly.

Here's a subclass of Duck, the MallardDuck.

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

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

Simple implementations: the duck just prints out what it is doing.

Now it's time to meet the newest fowl on the block:

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

Turkeys don't quack, they gobble.

Turkeys can fly, although they can only fly short distances.

```
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");
    }
}
```

Here's a concrete implementation of Turkey; like Duck, it just prints out its actions.

Now, let's say you're short on Duck objects and you'd like to use some Turkey objects in their place. Obviously we can't use the turkeys outright because they have a different interface.

So, let's write an Adapter:

Duck & Turkey

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

This time around, our ducks implement a Duck interface that allows Ducks to quack and fly.

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

Turkeys don't quack, they gobble.

Turkeys can fly, although they can only fly short distances.

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.

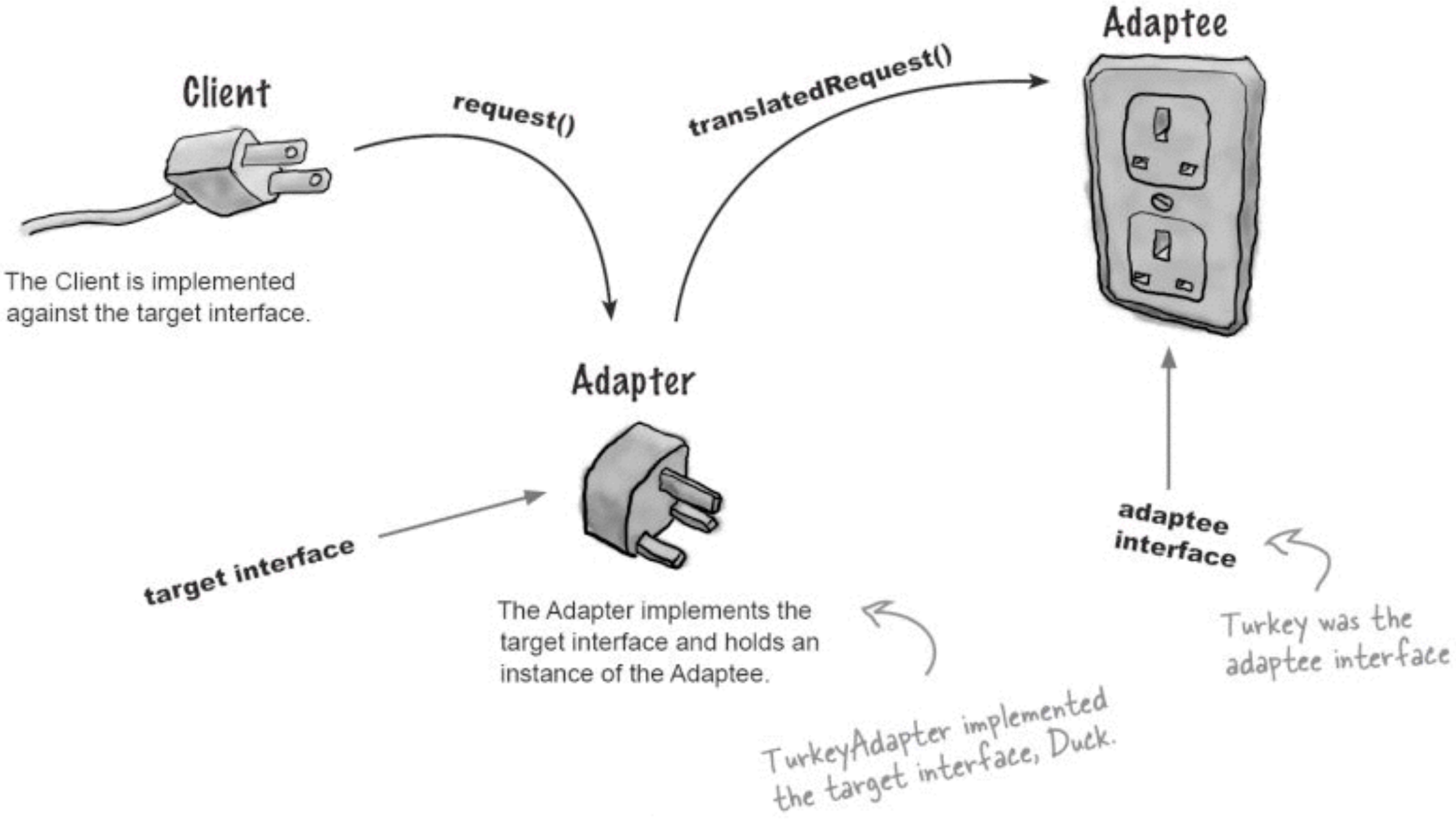
How to make a duck a turkey

```
class DuckAdapter implements Turkey{
   Duck duck;
   DuckAdapter(duck d) {
      duck = d;
   }
   public void fly(){
      if(distance > 500m)
         rest();
   }

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

}
```

The adapter pattern explained

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

The adapter pattern explained



Summary

- Decorator pattern allows small bits of functionality to be added
  - may lead to many small classes
  - hard to understand (Java I/O!!) and debug
- Adapter pattern permits unplanned evolution
  - two classes can have different assumptions
  - translating between them might be hard/ impossible

In-class Exercise

Adapting to Strobing

- General logic
  - The two behaves the same when push "off" button
  - The strobing action can be simulated on turning the flashing light on and off at regular intervals.
- Adaption steps
  - Hold a variable of "Flashing"
  - Delegate the "off" method directly to the "off" method of "Flashing"
  - Implement the "on" method by turning the flashing device on and off at regular intervals

Adapting to Flashing

- General logic
  - Two behaves the same with "off" button
  - For the "on" button, one way is to use two strobing lights that get turned on alternatively at the specified intervals.
- Adaptation steps
  - Delegate the "off" method directly to the "off" method of "Flashing"
  - Implement the "on" method by alternating the strobing devices on and off at regular intervals

Implementation

```
class SpecialStrobeLight implements Strobing{
   Flashing flashing_;
   SpecialStrobeLight(Flashing flashing){
      flashing_ = flashing;
   }

   public void on() {
      setState(true);
      while(getState()){
         flashing_.on();
         sleep(getInterval());
         flashing_off();
         }
   }

   public void off() {
      setState(false);
      flashing_.off();
   }
}
```

Illustration

Implementation

```
class SpecialFlashLight implements Flashing{
   Strobing strobing1_;
   Strobing strobing2_;
   SpecialFlashLight(Strobing strobing1, Strobing strobing2){
      strobing1_ = strobing1;
      stroginb2_ = strobing2;
   }

   public void on() {
      setState(true);
      while(getState()){
      strobing1_.on();
      sleep(getInterval());
      strobing2_.on();
      }
   }

   public void off() {
      setState(false);
      strobing1_.off();
      strobing2_.off();
   }
}
```