MORE DESIGN PATTERNS

DR. VIVEK NALLUR

VIVEK.NALLUR@SCSS.TCD.IE

OUTLINE OF THIS TALK

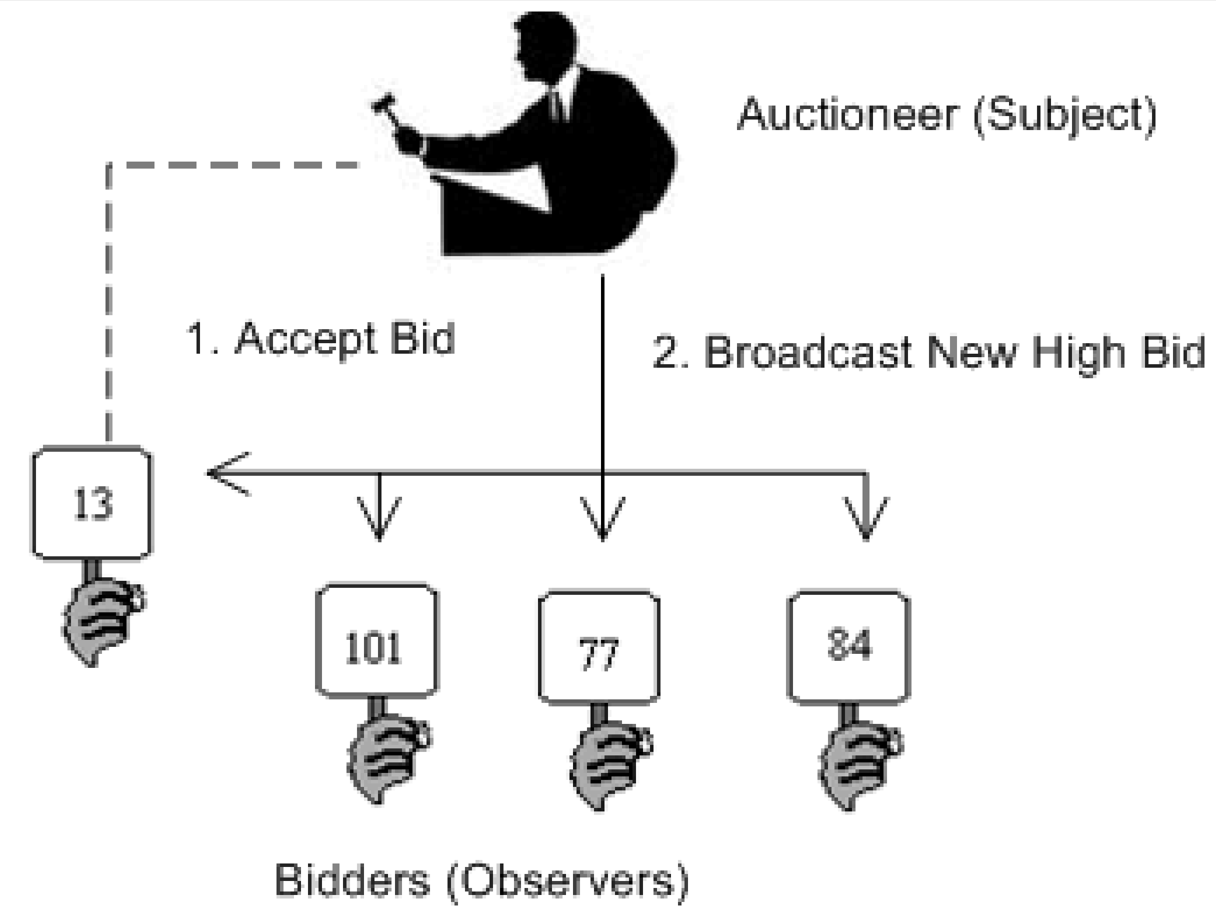
- Observer Pattern
- Decorator Pattern
- Adapter Pattern

START WITH REQUIREMENTS

- Implement an auction house software that allows electronic bidding
- Implement an IDE that has intelli-suggest of methods, syntax highlighting, and auto-save of code, committing code to repository
- Implement a home automation system, where objects inside the house react to the owner entering the house

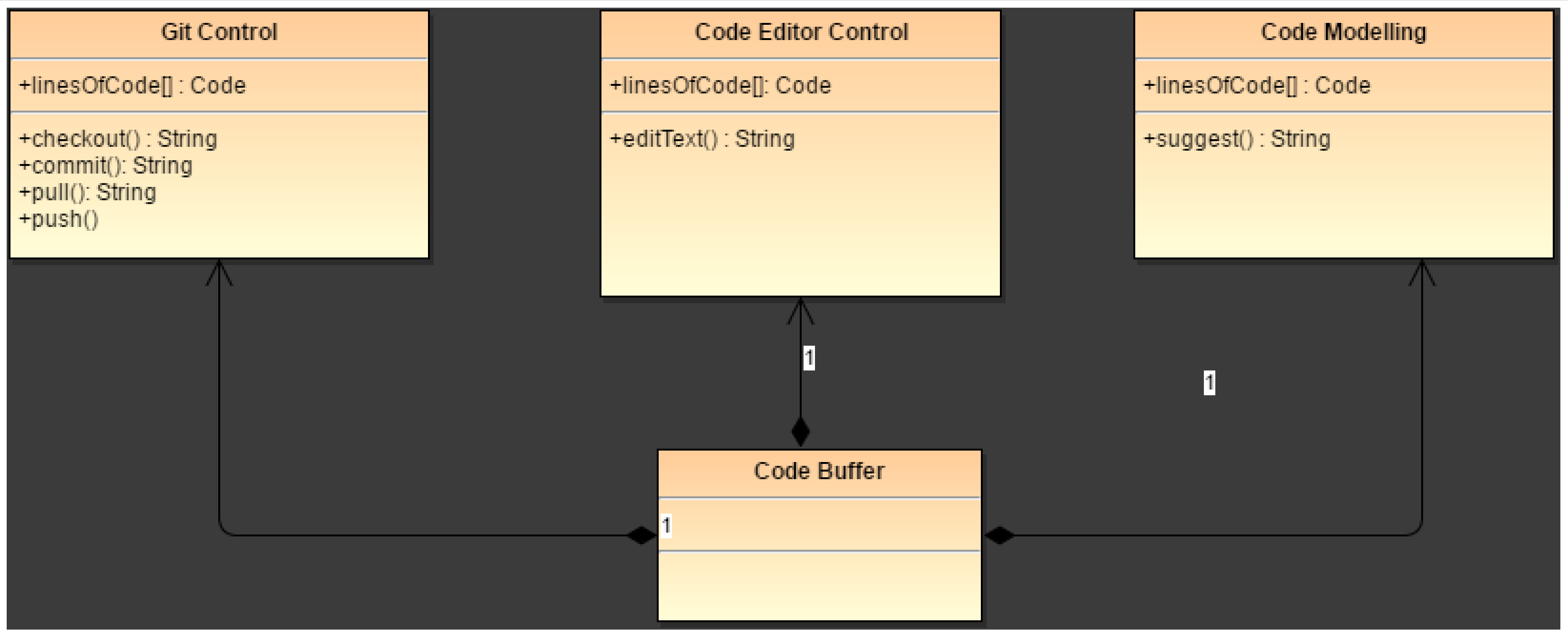
A SIMPLE AUCTION MECHANISM

- Identify the classes: Seller, Buyer(s), Auctioneer, Bids



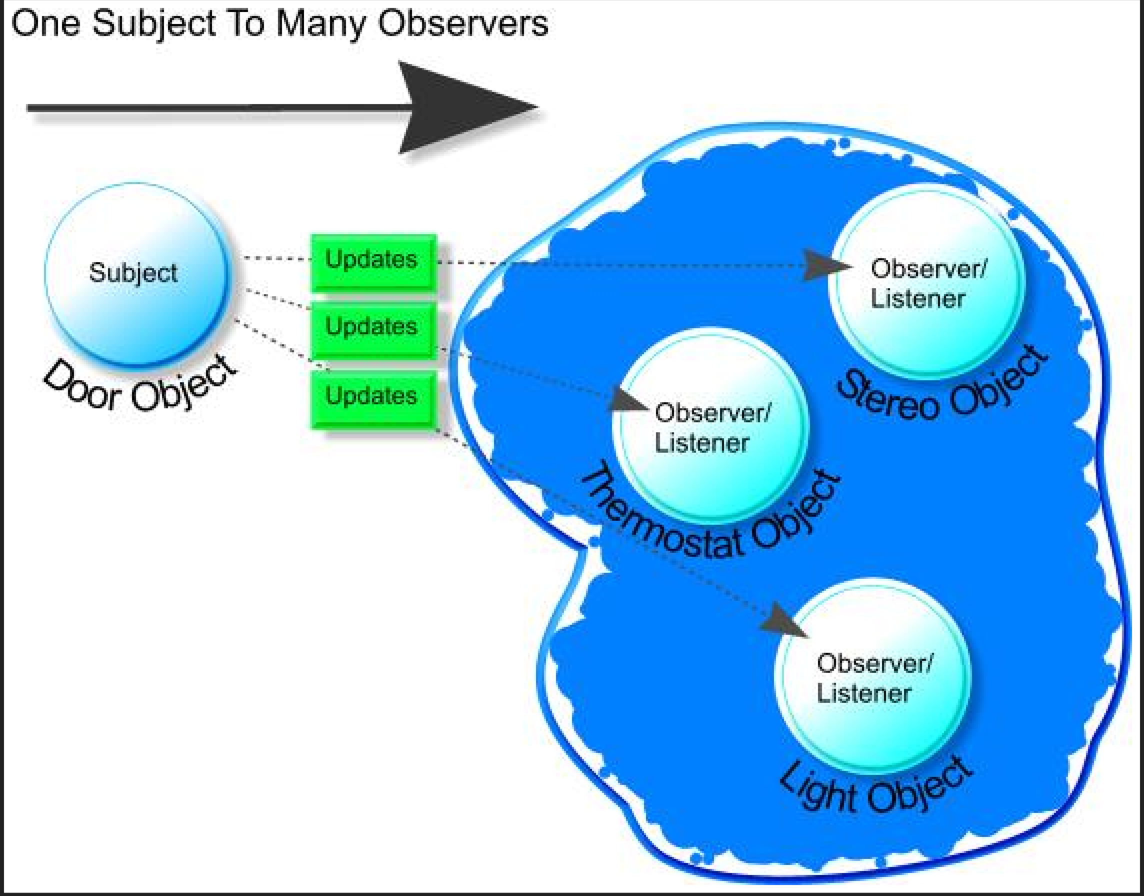
INTELLIGENT IDE

- Identify classes: CodeBuffer, GitControl, CodeModelling, CodeEditor



HOME AUTOMATION

- Identify classes: Door, Thermostat, Stereo, Light

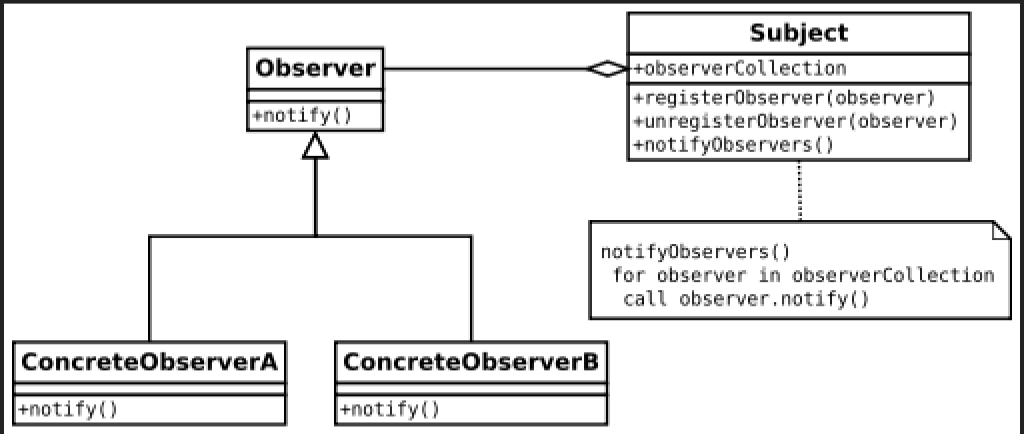


KEY OBSERVATIONS

- One to many relationship
- When the state of the 'One' changes, 'many' objects are notified
- The set of 'many' is *not* static

THE OBSERVER PATTERN

The *Observer Pattern* defines a one-to-many dependency between objects so that when one object changes state, all of its dependents are notified automatically



Observer
+notify()

Subject
+observerCollection
+registerObserver(observer)
+unregisterObserver(observer)
+notifyObservers()

```
notifyObservers()
 for observer in observerCollection
  call observer.notify()
```

ConcreteObserverA
+notify()

ConcreteObserverB
+notify()

IMPLEMENTING OBSERVER PATTERN

```java
import java.util.ArrayList;
import java.util.List;

public class Subject {

   private List<Observer> observers = new ArrayList<Observer>();
   private int state;

   public int getState() {
      return state;
   }

   public void setState(int state) {
      this.state = state;
      notifyAllObservers();
   }
```

IMPLEMENTING OBSERVER PATTERN - II

```java
public abstract class Observer {
   protected Subject subject;
   public abstract void update();
}
```

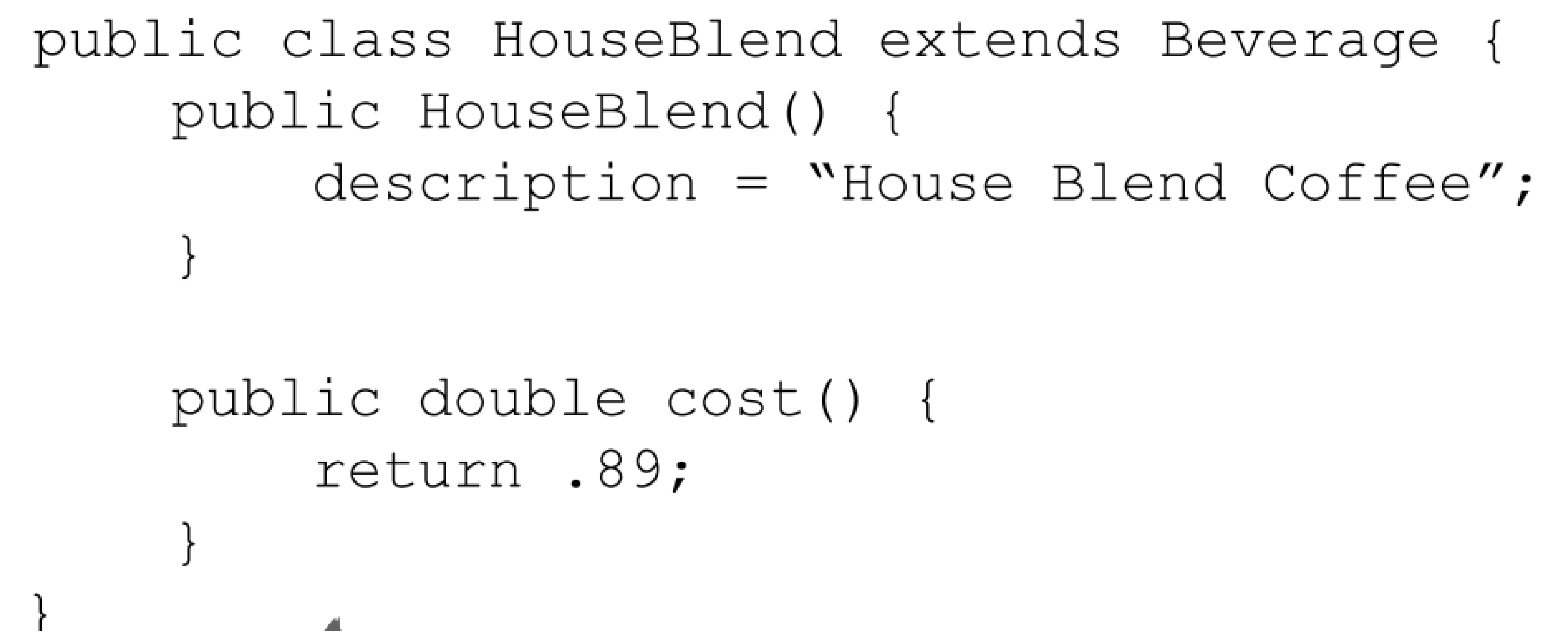
```java
public class BinaryObserver extends Observer{

   public BinaryObserver(Subject subject){
      this.subject = subject;
      this.subject.attach(this);
   }

   @Override
   public void update() {
      System.out.println( "Binary String: " + subject.getState()  );
   }
}
```

POINT-OF-SALE FOR A COFFEE SHOP

- The shop sells many types of coffee: *Dark Roast, Espresso, House Blend, Decaf*
- Each customer can order multiple types of condiments: *Mocha, Soy, Whip, SteamedMilk, …*
- The POS must calculate the cost for a changing set of coffee-types, with a changing set of condiments

NAIVE IMPLEMENTATION

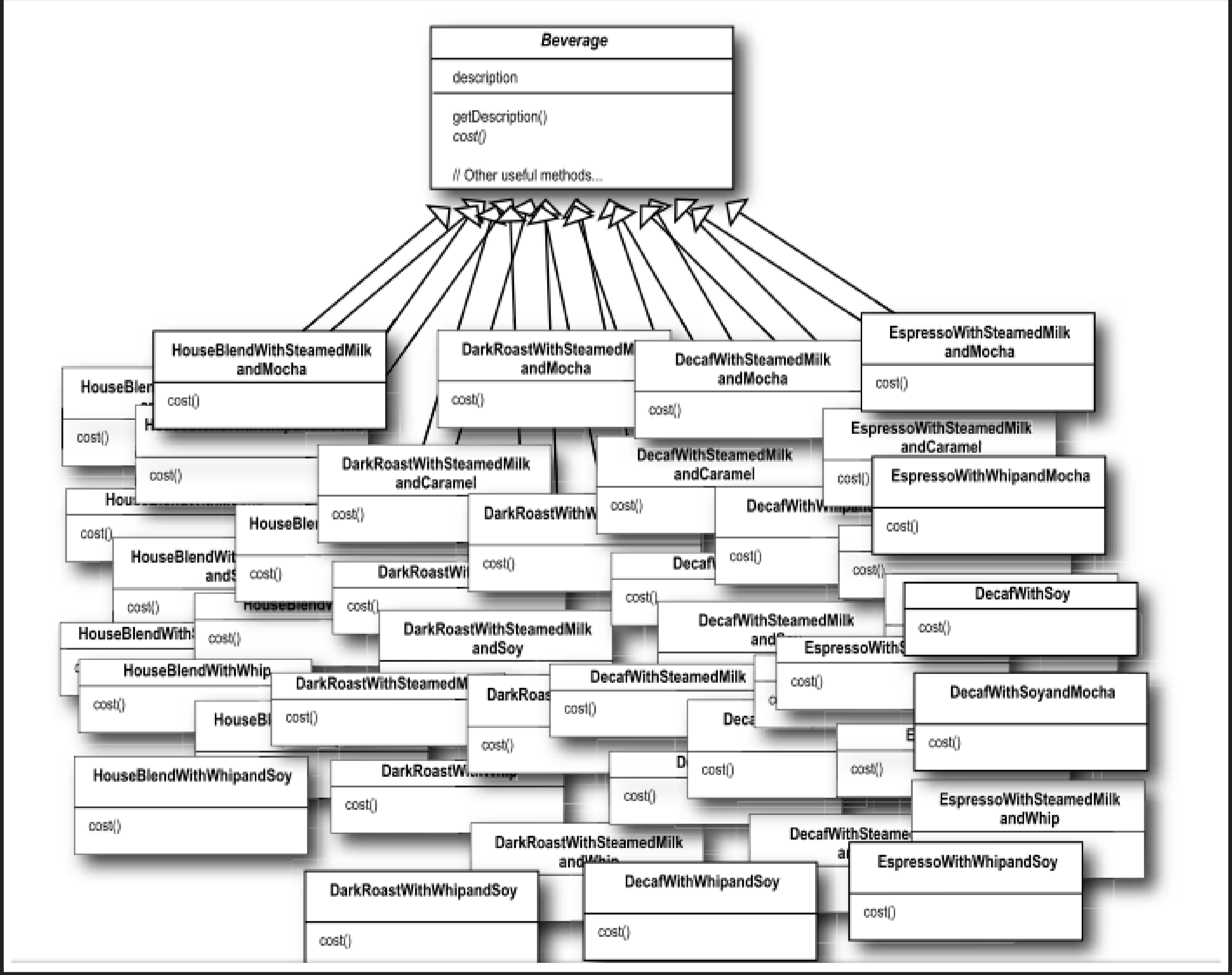
Create an abstract class of `Beverage` and then let all types sub-class it



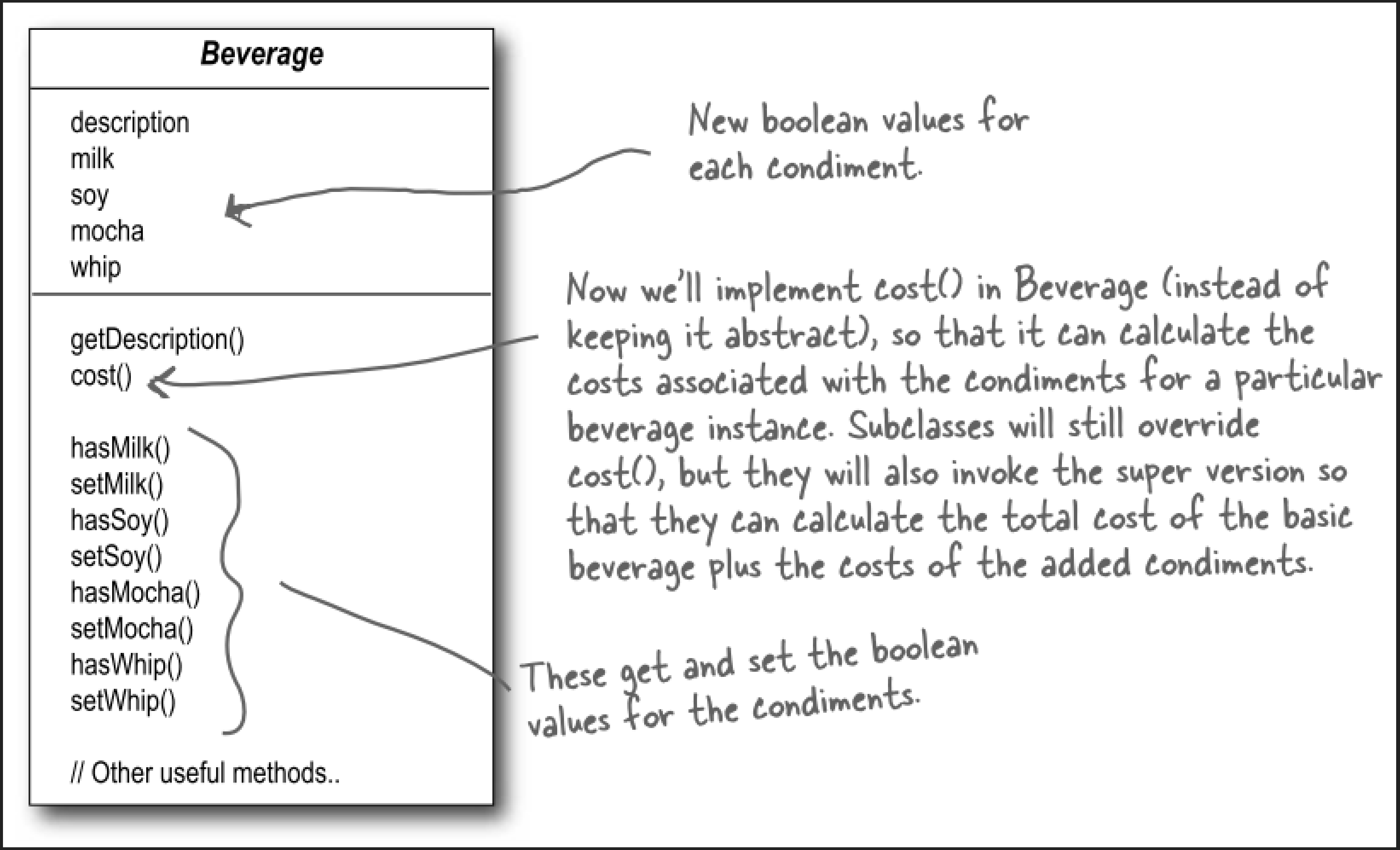
```
public class HouseBlend extends Beverage {
    public HouseBlend() {
        description = “House Blend Coffee”;
    }

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

NAIVE IMPLEMENTATION

Just calculate all combinations and figure out the cost


Beverage
description
getDescription()
cost()
// Other useful methods...
HouseBlendWithSteamedMilk andMocha
DarkRoastWithSteamedMilk andMocha
DecafWithSteamedMilk andMocha
EspressoWithSteamedMilk andMocha
EspressoWithSteamedMilk andCaramel
DarkRoastWithSteamedMilk andCaramel
DecafWithSteamedMilk andCaramel
EspressoWithWhipandMocha
DecafWithSoy
DarkRoastWithSteamedMilk andSoy
DecafWithSoyandMocha
HouseBlendWithWhip
EspressoWithSteamedMilk andWhip
HouseBlendWithWhipandSoy
DarkRoastWithSteamedMilk andWhip
DarkRoastWithWhipandSoy
DecafWithWhipandSoy
EspressoWithWhipandSoy
cost()

BUT WE'RE CLEVER



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.

What's wrong with this approach?

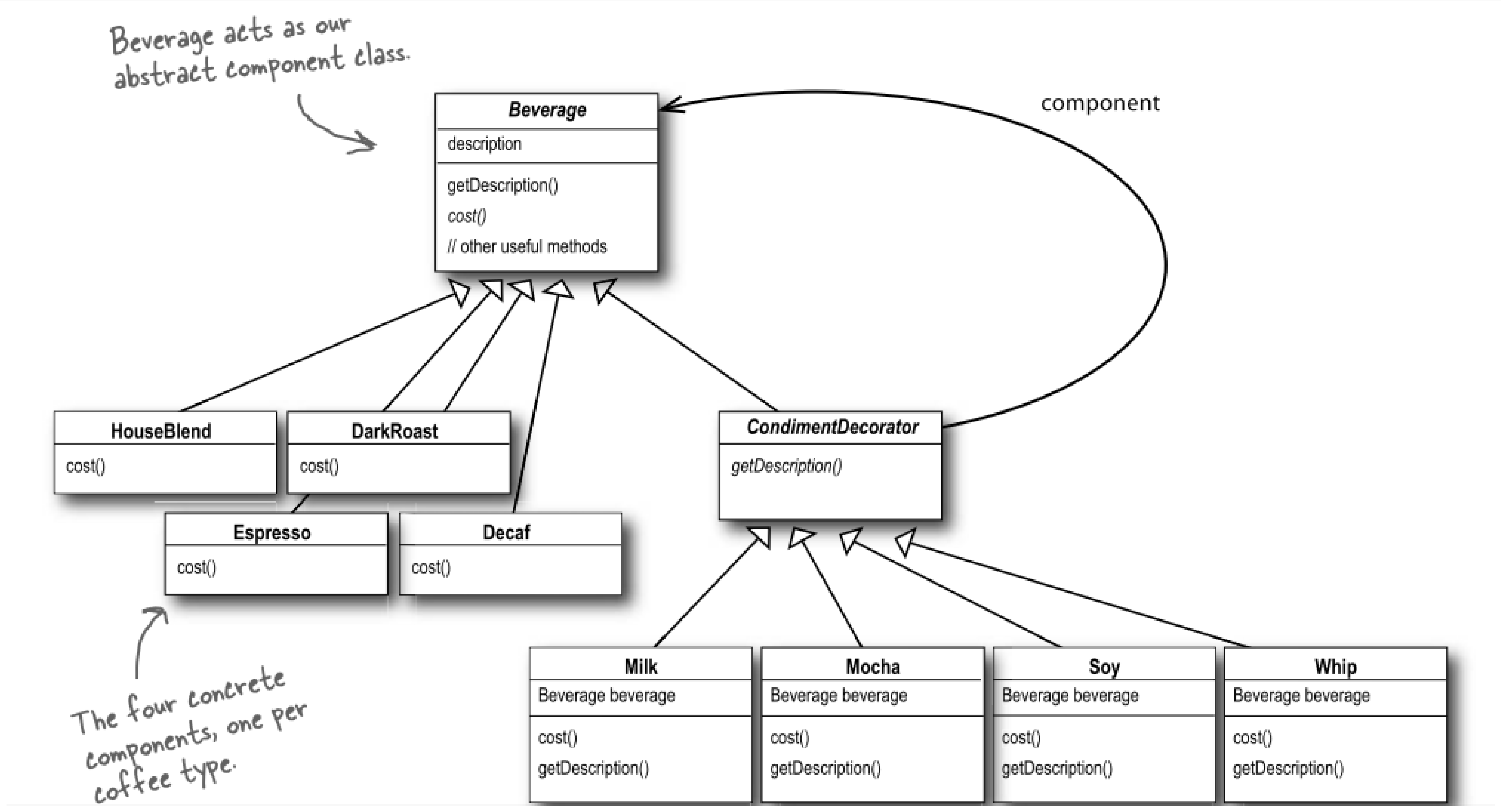
CHANGE IS INEVITABLE

- Price of condiments change
- New condiments appear and unpopular ones disappear
- Different combinations?
- New beverage sold (Organic-kale-antioxidant-juice, iced-tea, ...)

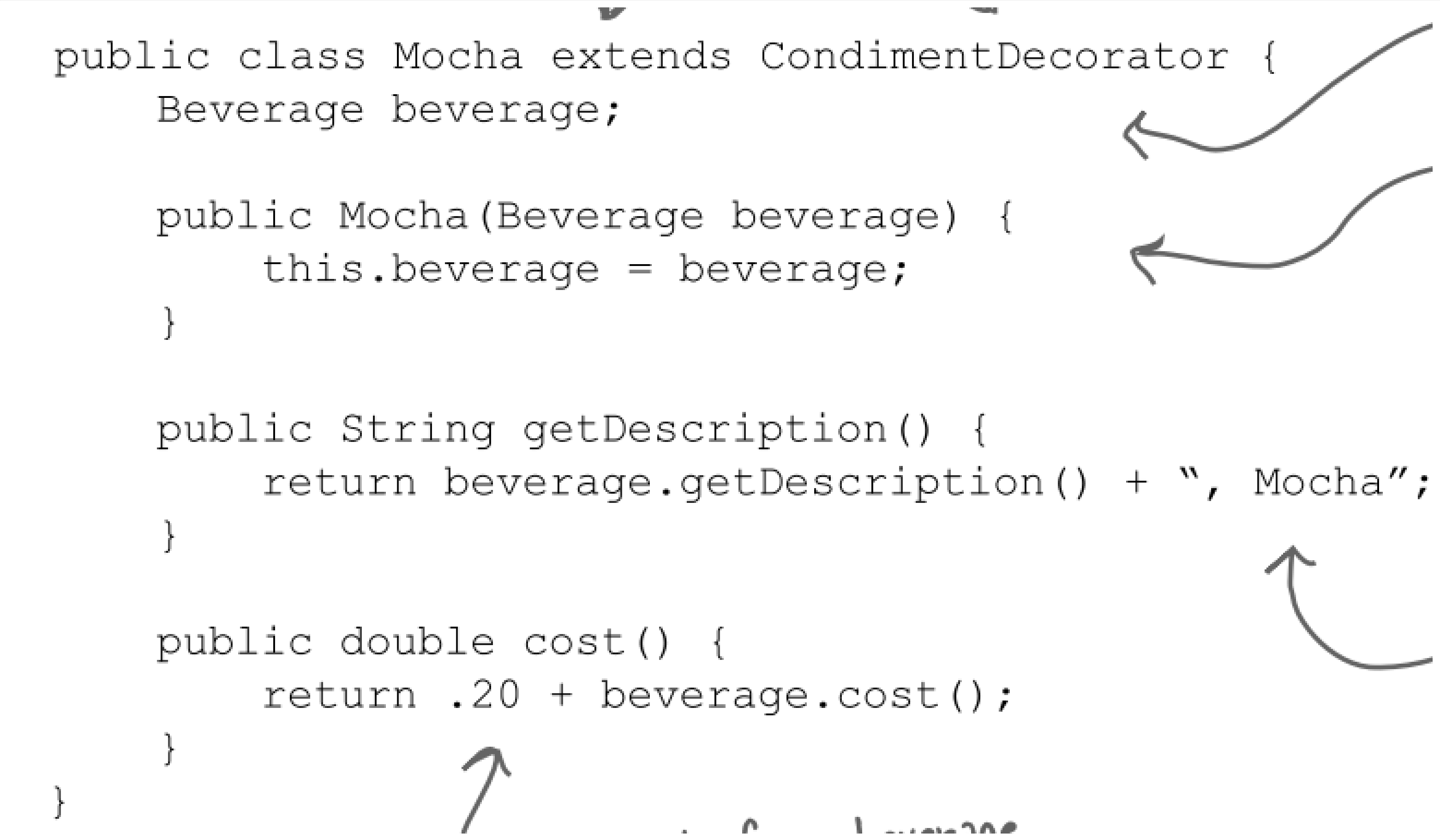
DESIGN PRINCIPLE: OPEN-CLOSED PRINCIPLE

Classes should be open for extension but closed for modification

THE DECORATOR PATTERN



IMPLEMENTING THE DECORATOR



```
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();
    }
}
```

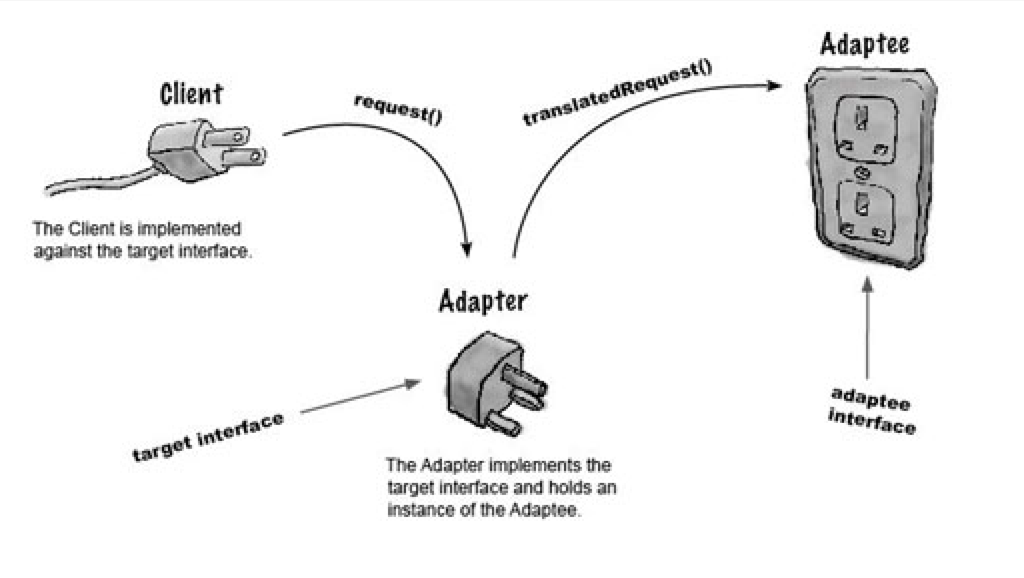
THINGS TO REMEMBER ABOUT DECORATOR

- A decorator has the *same supertype* as the object it decorates
- A decorator can be passed anywhere that the original object can be passed
- The decorator adds its own behaviour *either before* or *after* delegatingto the object it decorates

DECORATOR PATTERN DESCRIBED

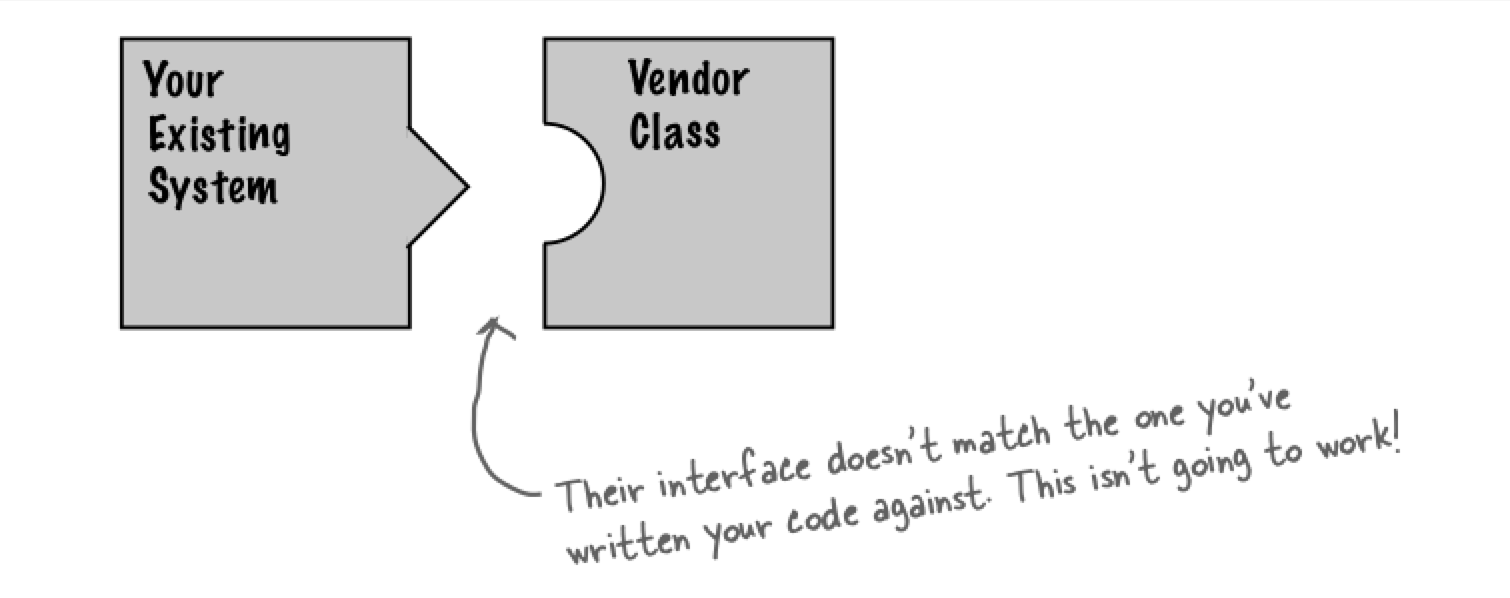
The decorator pattern attaches additional responsibilities to an object dynamically. Decorators provide a flexible alternative to sub-classing for extending functionality

(ALMOST) EVERYONE KNOWS THE ADAPTER PATTERN

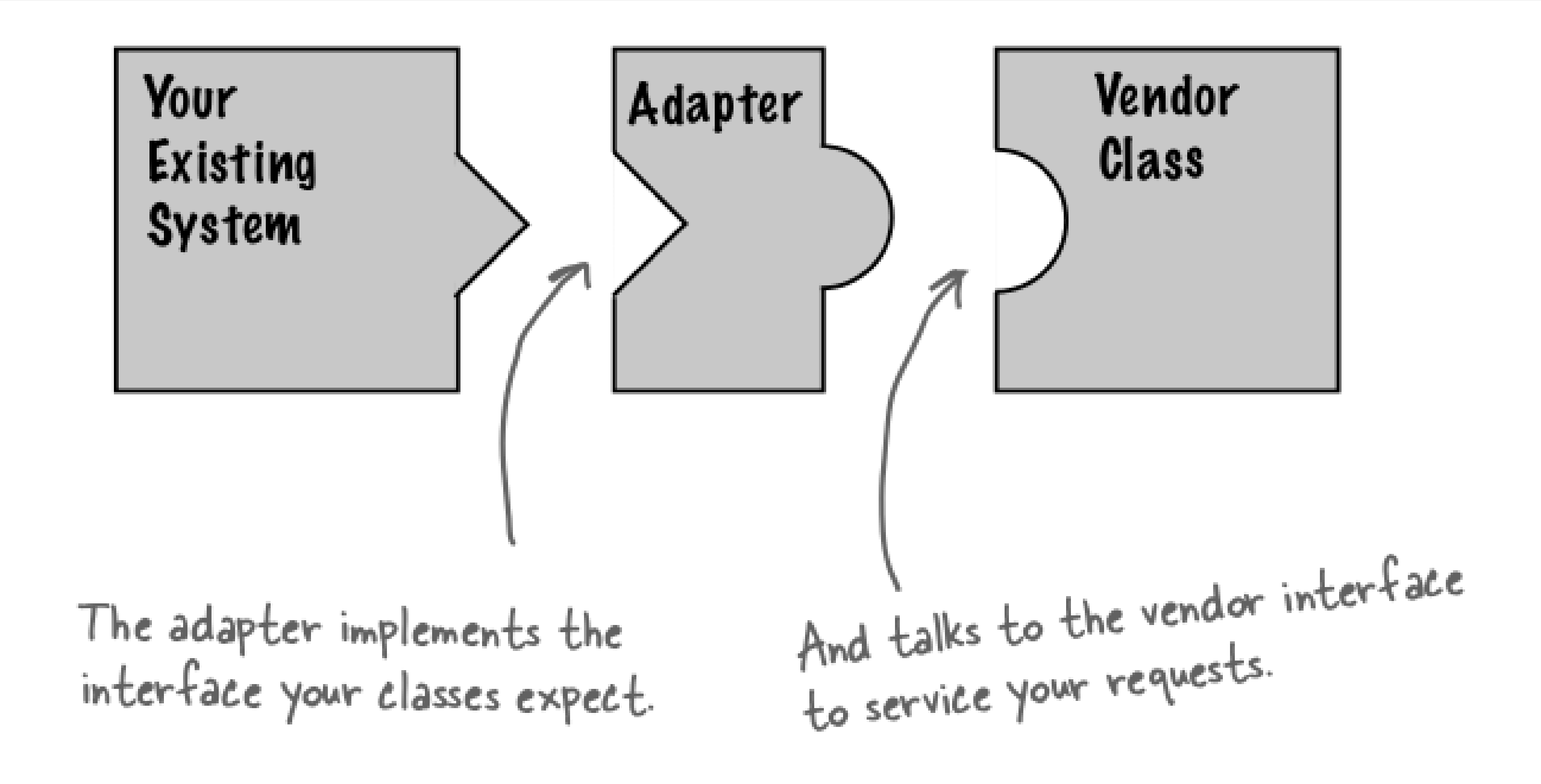


Client
request()
translatedRequest()
Adaptee
The Client is implemented against the target interface.
Adapter
target interface
adaptee interface
The Adapter implements the target interface and holds an instance of the Adaptee.

PROBLEM CONTEXT


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

SOLUTION


Your Existing System
Adapter
Vendor Class
The adapter implements the interface your classes expect.
And talks to the vendor interface to service your requests.

- Client makes a request to the adapter
- Adapter translates the request into one or more calls on the adaptee

IMPLEMENTING ADAPTER PATTERN - I

Before

```
class LegacyLine
{
    public void draw(int x1, int y1, int x2, int y2)
    {
        System.out.println("line from (" + x1 + ',' + y1 + ") to (" +
          + y2 + ')');
    }
}
```

```
class LegacyRectangle
{
    public void draw(int x, int y, int w, int h)
    {
        System.out.println("rectangle at (" + x + ',' + y + ") with w
          + " and height " + h);
    }
}
```

IMPLEMENTING ADAPTER PATTERN - II

After

```
interface Shape
{
  void draw(int x1, int y1, int x2, int y2);
}

class Line implements Shape
{
    private LegacyLine adaptee = new LegacyLine();
    public void draw(int x1, int y1, int x2, int y2)
    {
        adaptee.draw(x1, y1, x2, y2);
    }
}

class Rectangle implements Shape
{
```

ADAPTER PATTERN DESCRIBED

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

THAT'S ALL, FOLKS!

Questions? Comments?