

Design Patterns

Proxy, Factory, Singleton, Adapter,
Façade, Decorator, Strategy and
Observer Patterns

Abdelkarim Erradi
QU

The slides are based on the book:
Head First Design Patterns
Eric Freeman, Elisabeth Freeman & el.
O'Reilly 2004

What is a Design Pattern?

- A proven solution to a common problem.
- Aim = Improve the software Quality Attributes
 - DP = a technique to repeat designer success.
- Many patterns are being implemented in programming languages.

Creational patterns

- *Creational* patterns specialize in abstracting the instantiation process.
 - They help in isolating how objects are created, composed and represented from the rest of the system.
- They are five patterns defined in this category: Abstract Factory, Builder, Factory Method, Prototype, Singleton.

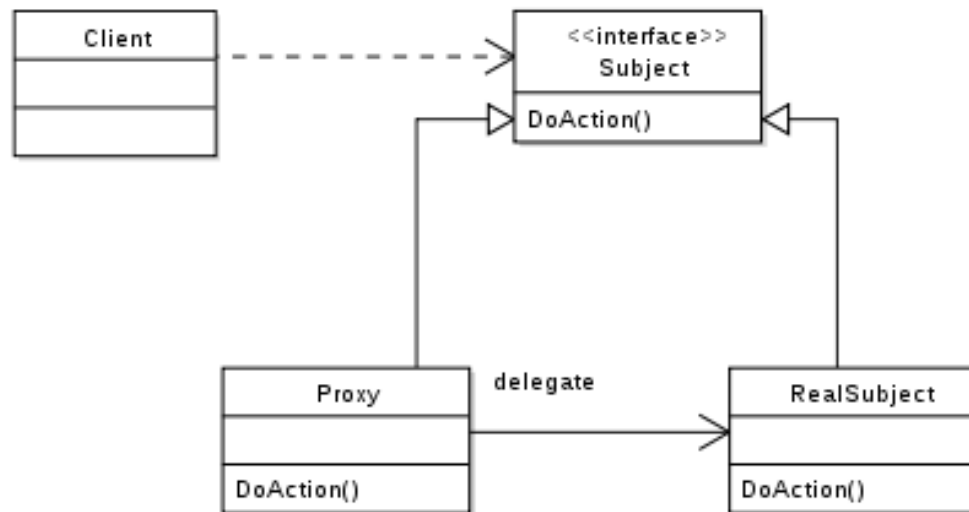
Behavioral patterns

- Behavioral patterns are concerned with algorithms and the **assignment of responsibilities** to objects.
 - They document the patterns of objects as well as the patterns of communication between them.
 - They are eleven patterns defined in this category : Chain of responsibility, Command, Interpreter, Iterator, Mediator, Memento, Observer, State, Strategy, Template Method and Visitor.

Structural Patterns

- **Structural patterns** are concerned with how classes and objects are composed to form larger structures.
- Seven patterns are defined in this category : Adapter, Bridge, Composite, Decorator, Façade, Flyweight and Proxy.

Proxy Pattern

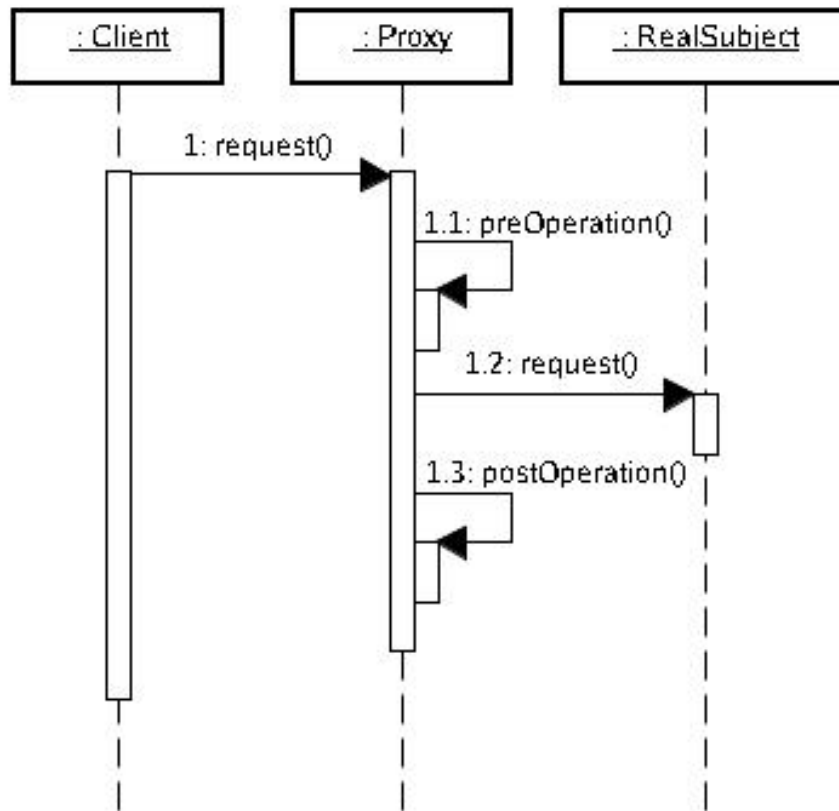


Context

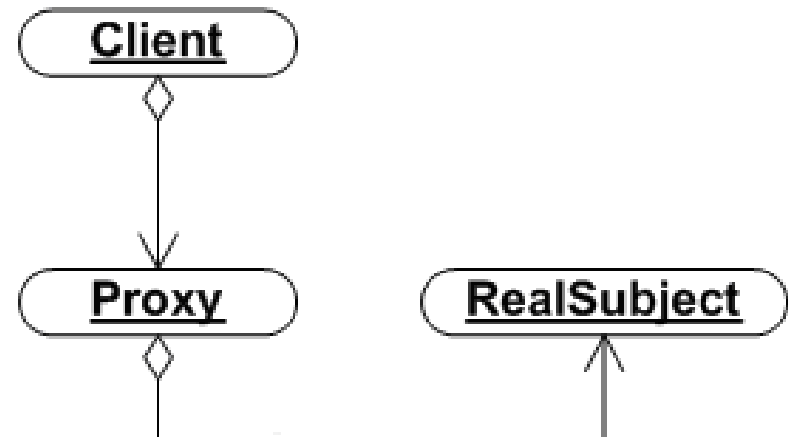
- Use the proxy pattern to create a **representative object** that **controls access to another object**, which may be remote, expensive to create or in need of security.
 - You have a class that you would like to provide controlled access to it
- **Access need to be transparent and simple for the clients**
 - Proxy Handles low level details of network communication
 - clients of a component communicate with a representative rather than to the component itself

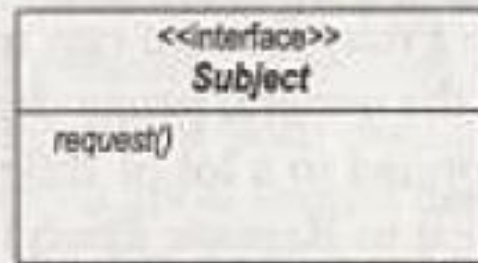
Solution

Sequence diagram of proxy pattern:

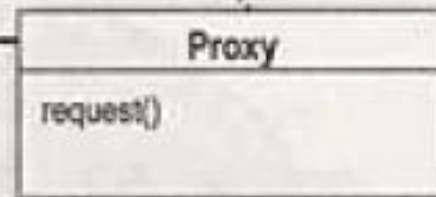
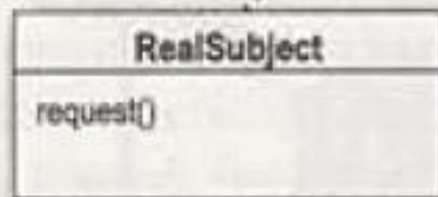


Object Graph of proxy pattern:





Both the Proxy and the RealSubject implement the Subject interface. This allows any client to treat the proxy just like the RealSubject



The RealSubject is usually the object that does most of the real work; the Proxy controls access to it.

The Proxy often instantiates or handles the creation of the RealSubject.

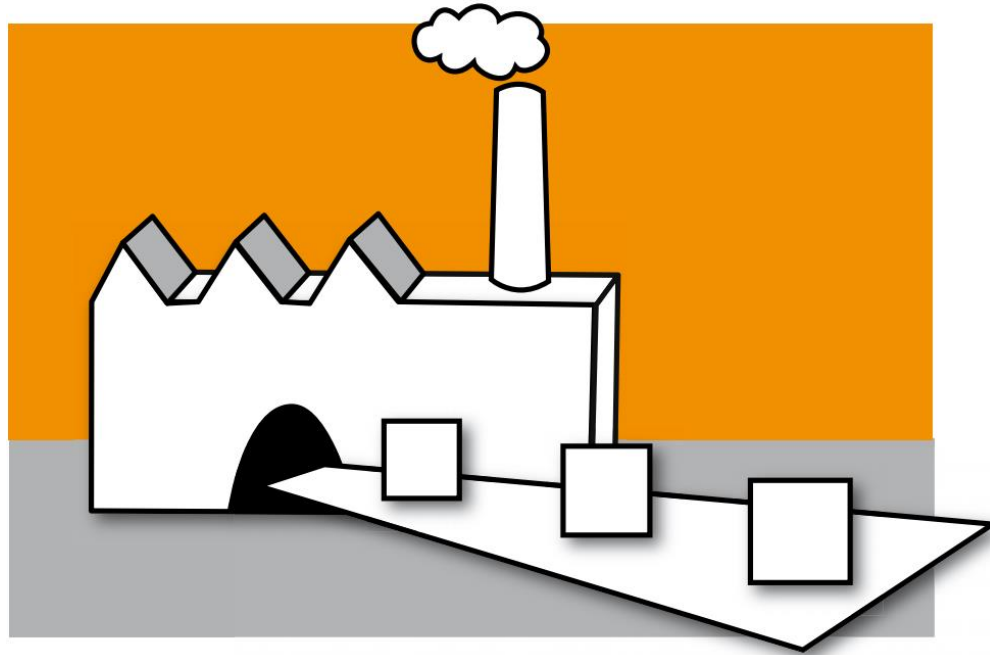
The Proxy keeps a reference to the Subject, so it can forward requests to the Subject when necessary.

General Scenario

- Client need to do a task
- Client calls the method of Proxy (send a service message)
- Proxy internally do pre-processing such as message validation
- Proxy sends the real service message to Original
- Proxy do post-processing and return results back to Client

Factory Pattern

- Handles the creation of complex objects



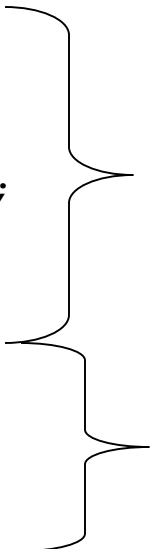
Factory Motivation

- Consider a pizza store that makes different types of pizzas

```
Pizza pizza;
```

```
if (type == CHEESE)
    pizza = new CheesePizza();
else if (type == PEPPERONI)
    pizza = new PepperoniPizza();
else if (type == PESTO)
    pizza = new PestoPizza();
```

```
pizza.prepare();
pizza.bake();
pizza.package();
pizza.deliver();
```



This becomes unwieldy
as we add to our menu

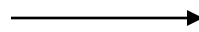
This part stays the same

Idea: pull out the creation code and put it into an object
that only deals with creating pizzas - the PizzaFactory

Abstract Factory Motivation

```
public class PizzaFactory
{
    public Pizza createPizza(int type)
    {
        Pizza pizza = null;
        if (type == CHEESE)
            pizza = new CheesePizza();
        else if (type == PEPPERONI)
            pizza = new PepperoniPizza();
        else if (type == PESTO)
            pizza = new PestoPizza();
        return pizza;
    }
}
```

Replace concrete instantiation with
call to the PizzaFactory to create a
new pizza



Now we don't need to mess with this
code if we add new pizzas

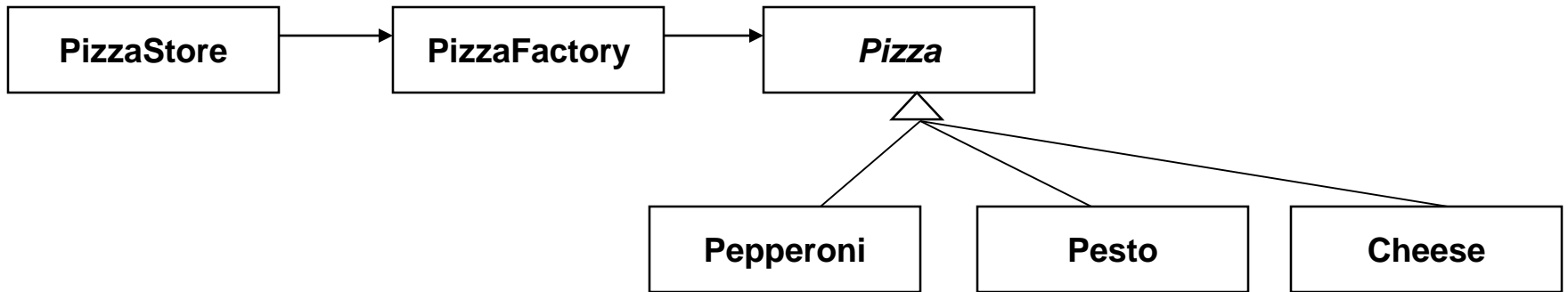
```
Pizza pizza;
PizzaFactory factory;

...

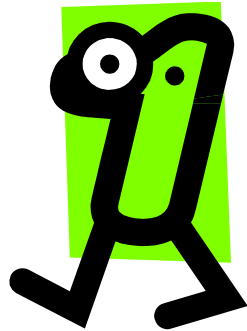
pizza = factory.createPizza(type);

pizza.prepare();
pizza.bake();
pizza.package();
pizza.deliver();
```

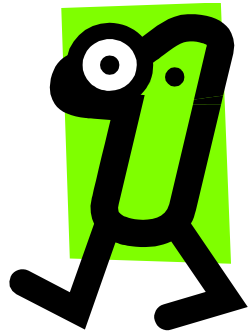
Pizza Classes



Singleton Pattern



Singleton pattern

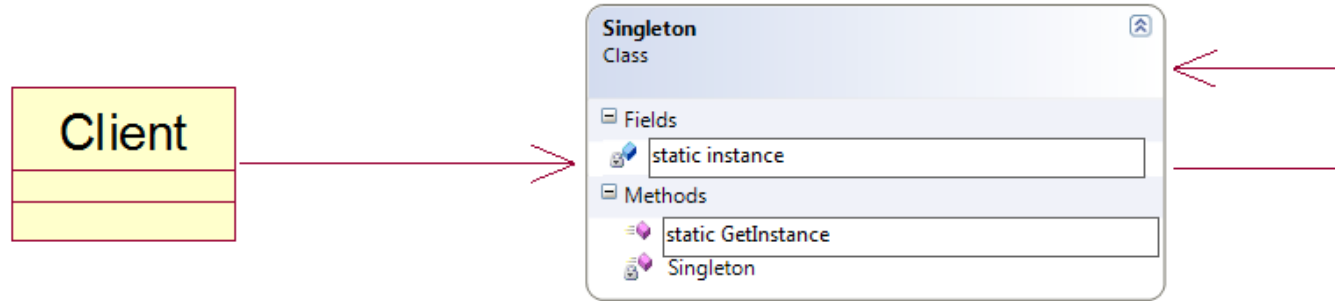


- **Intent**
 - Ensure that a certain class has only **one** instance
 - Provide global access to it
- **Motivation**
 - It's important for some classes to have exactly one instance.
 - This is particularly useful when one object is needed to coordinate actions across a software system.
 - There should be only one file system (or file system manager) and one window manager.

The Singleton pattern ensures that the class (not the programmer) is responsible for the number of instances created.

The Singleton Pattern

The class of the single instance is responsible for access and “initialization on first use”.



- The single instance is a protected static attribute in order to guarantee that a new instance is created if one doesn't already exist, and if one does exist, a reference to that instance is accessible.
- The constructor is private in order to ensure that the object can only be instantiated via the constructor.

Example

```
public class Singleton {  
    private static Singleton instance = null;  
    private Singleton() {  
    }  
    public static Singleton getInstance() {  
        if(instance == null) {  
            instance = new Singleton();  
        }  
        return instance;  
    }  
}  
  
public class TestSingleton{  
    public static void main(String[] args){  
        Singleton s = Singleton.getInstance();  
        ...  
    }  
}
```

Singleton
-instance: Singleton
-Singleton() +getInstance(): Singleton

Important Concepts

1. Why is the constructor private?

Any client that tries to instantiate the Singleton class directly will get an error at compile time.

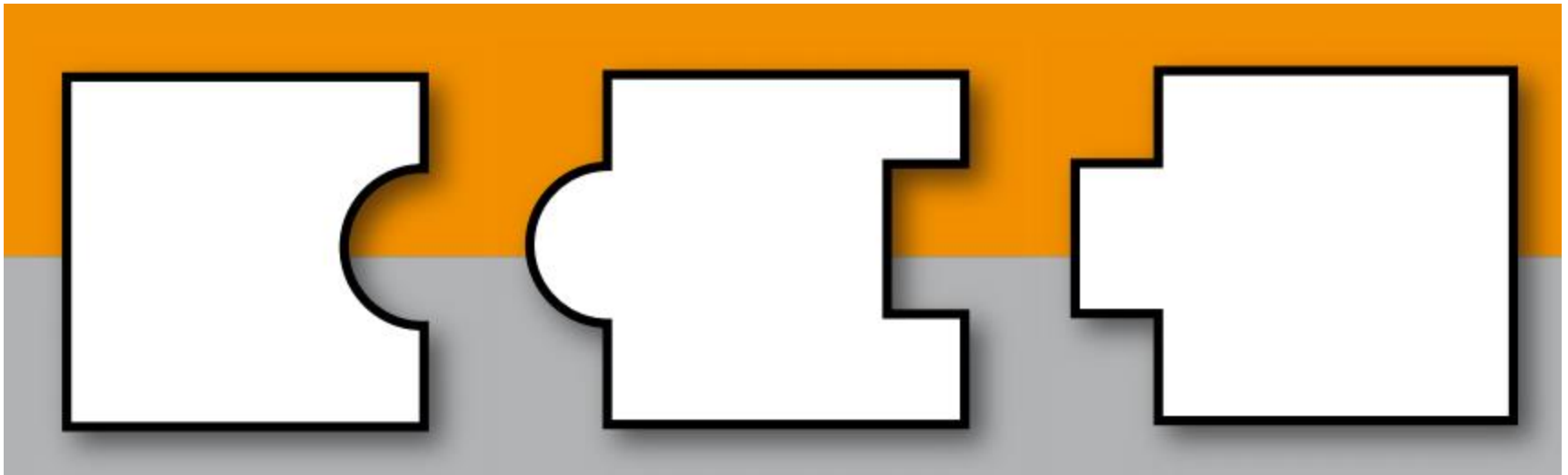
- This ensures that nobody other than the Singleton class can instantiate the object of this class.

2. How can you access the single instance of the Singleton class?

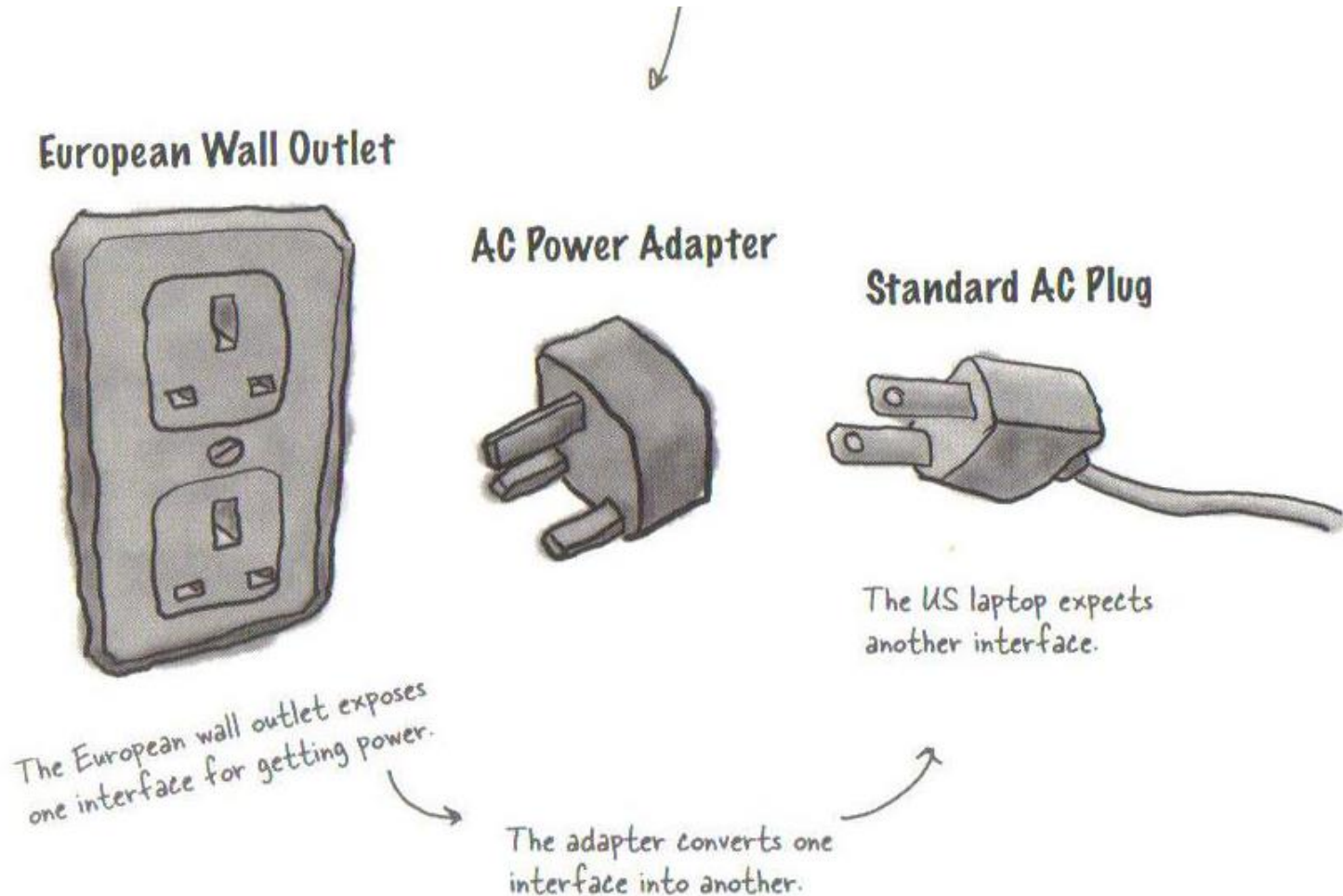
`getInstance()` is a static method. We can use the class name to reference the method.

- `Singleton.getInstance()`
- This is how we can provide global access to the single instance of the Singleton class.

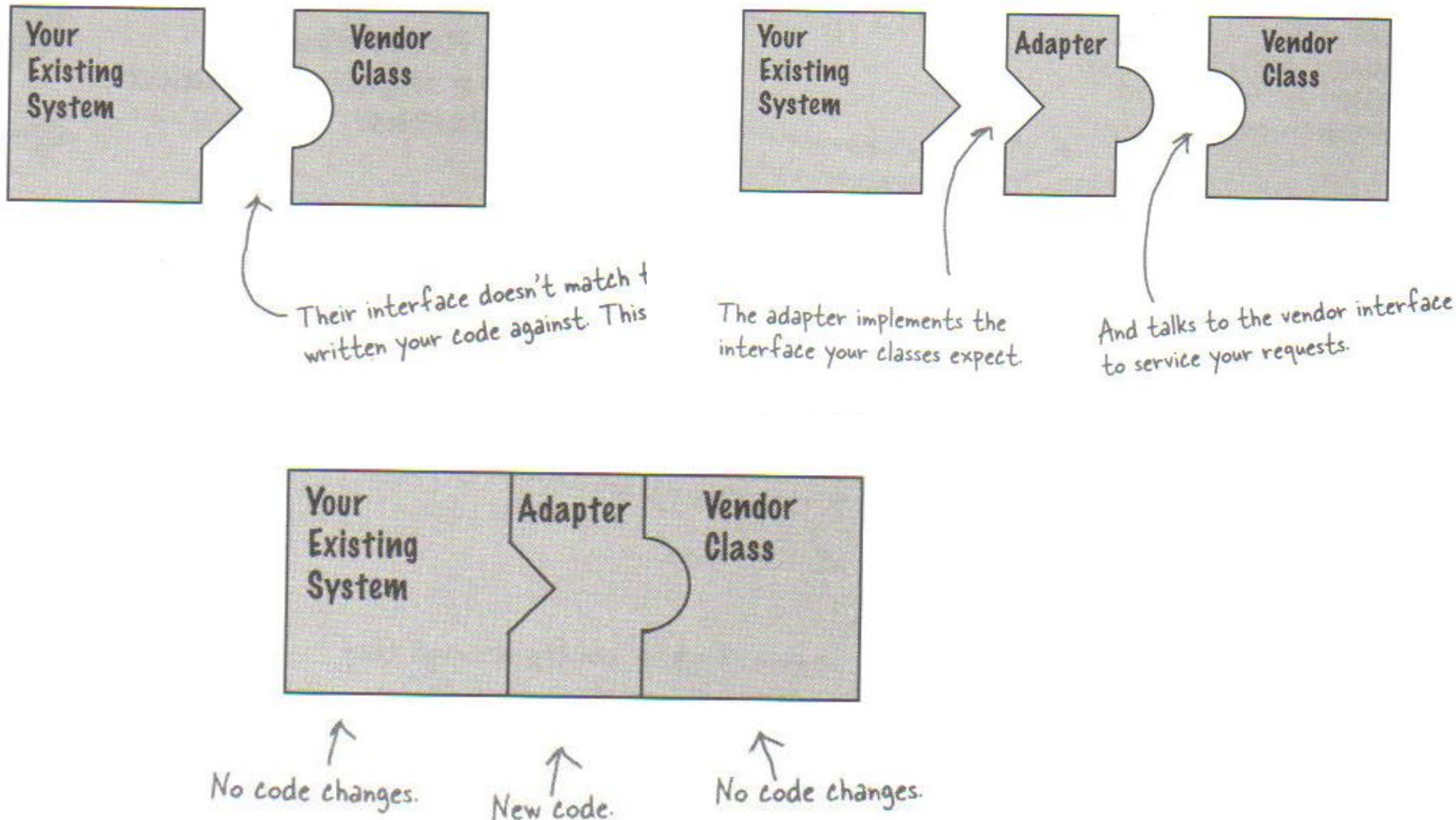
Adapter Pattern



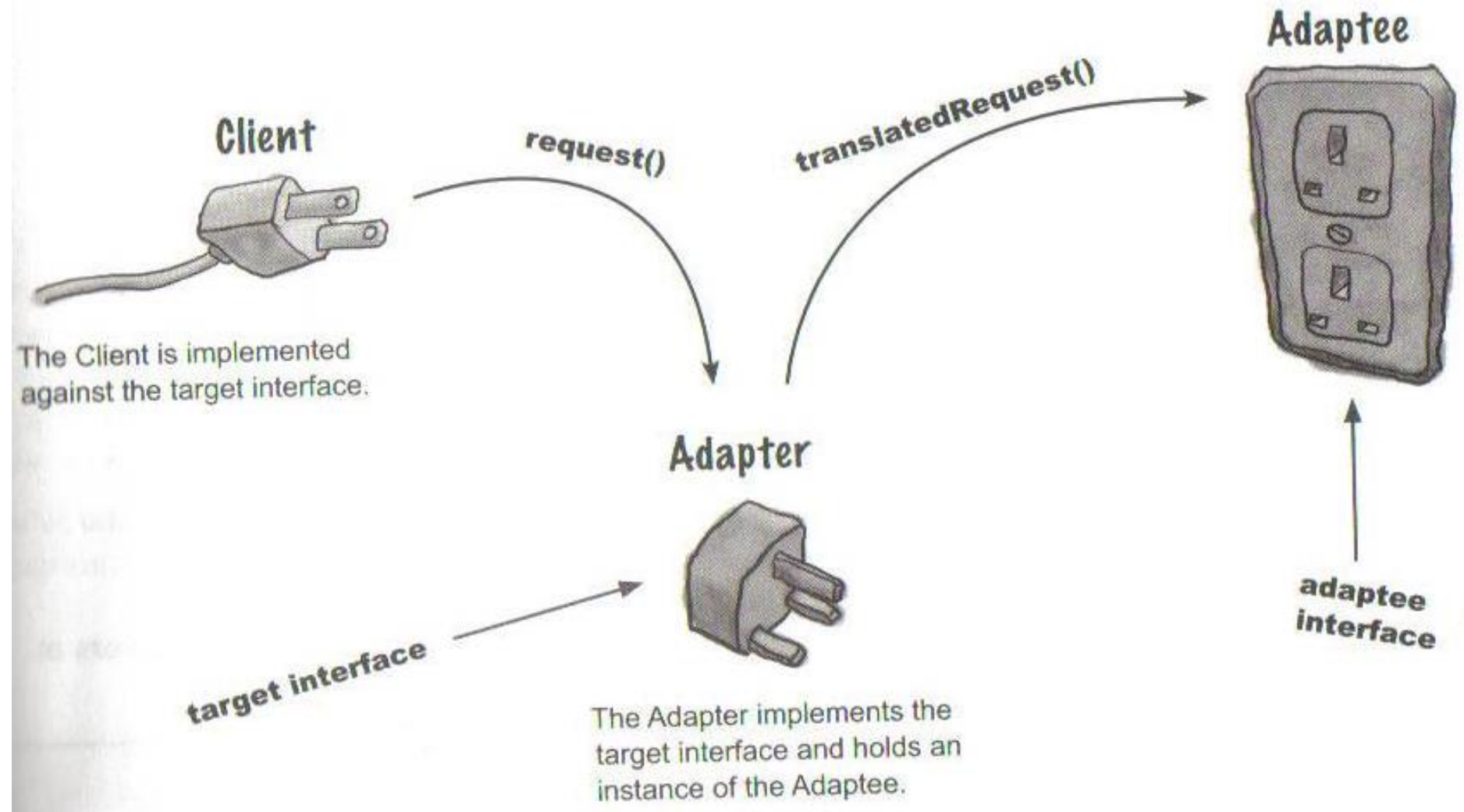
Adapters in real life



Object-Oriented Adapters



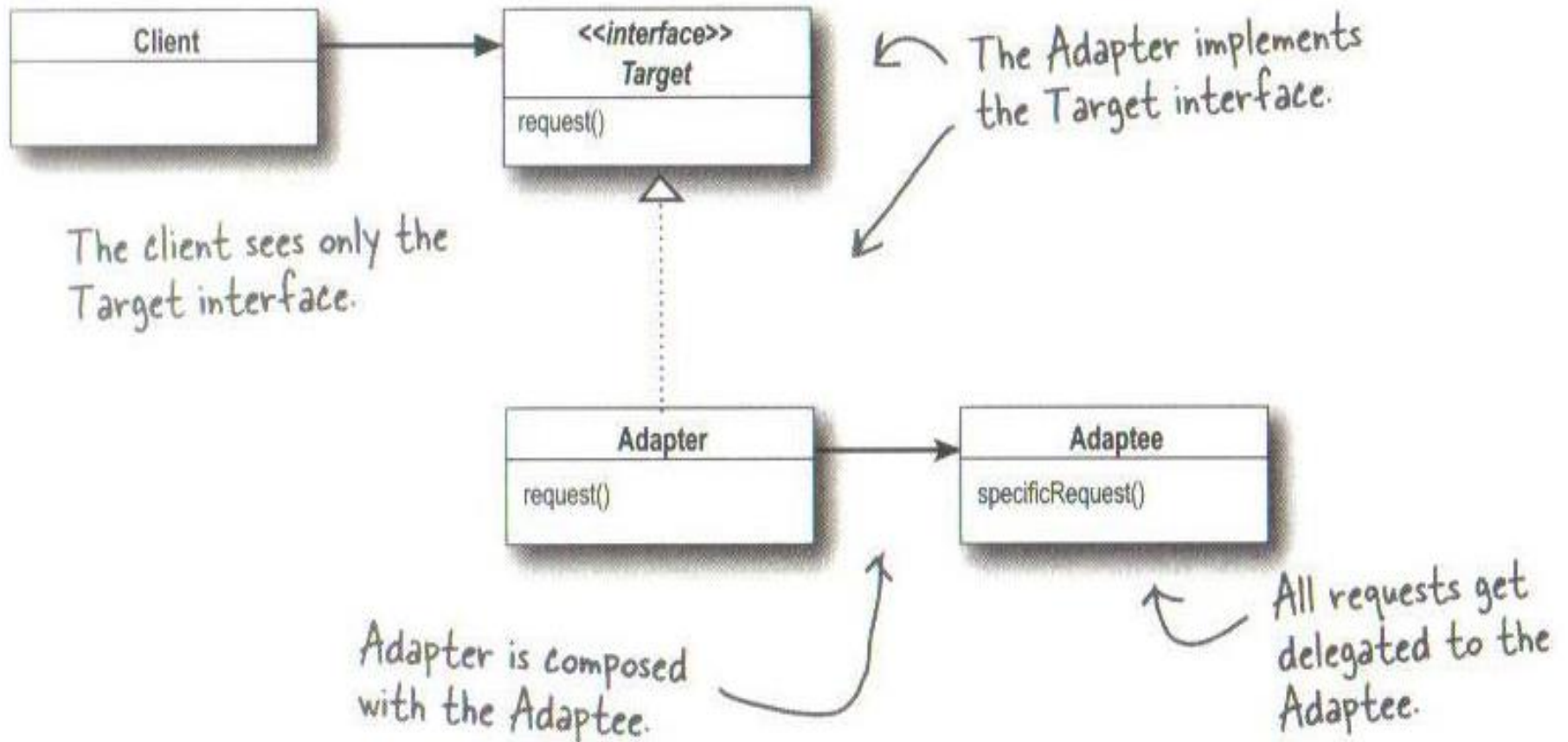
Adapter Pattern explained



Adapter pattern

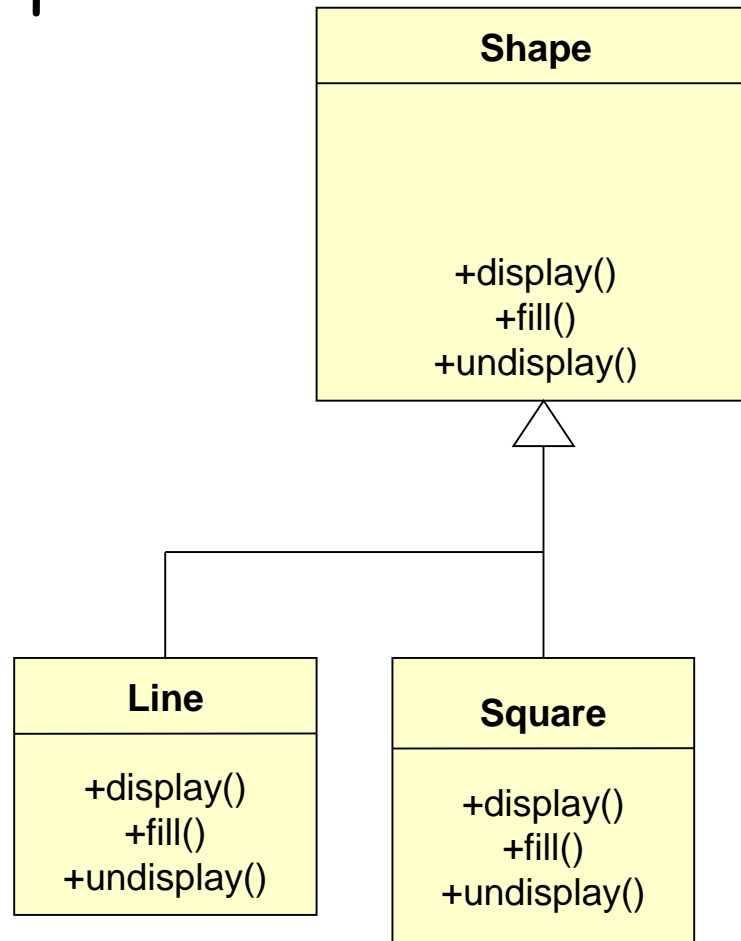
- **Intent**
 - The Adapter Pattern converts the interface of a class into another interface the clients expect.
 - Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.
- **Motivation**
 - Sometimes a toolkit or class library can not be used because its interface is incompatible with the interface required by an application
 - We can not change the library interface, since we may not have its source code
 - Even if we did have the source code, we probably should not change the library for each domain-specific application

Adapter Pattern

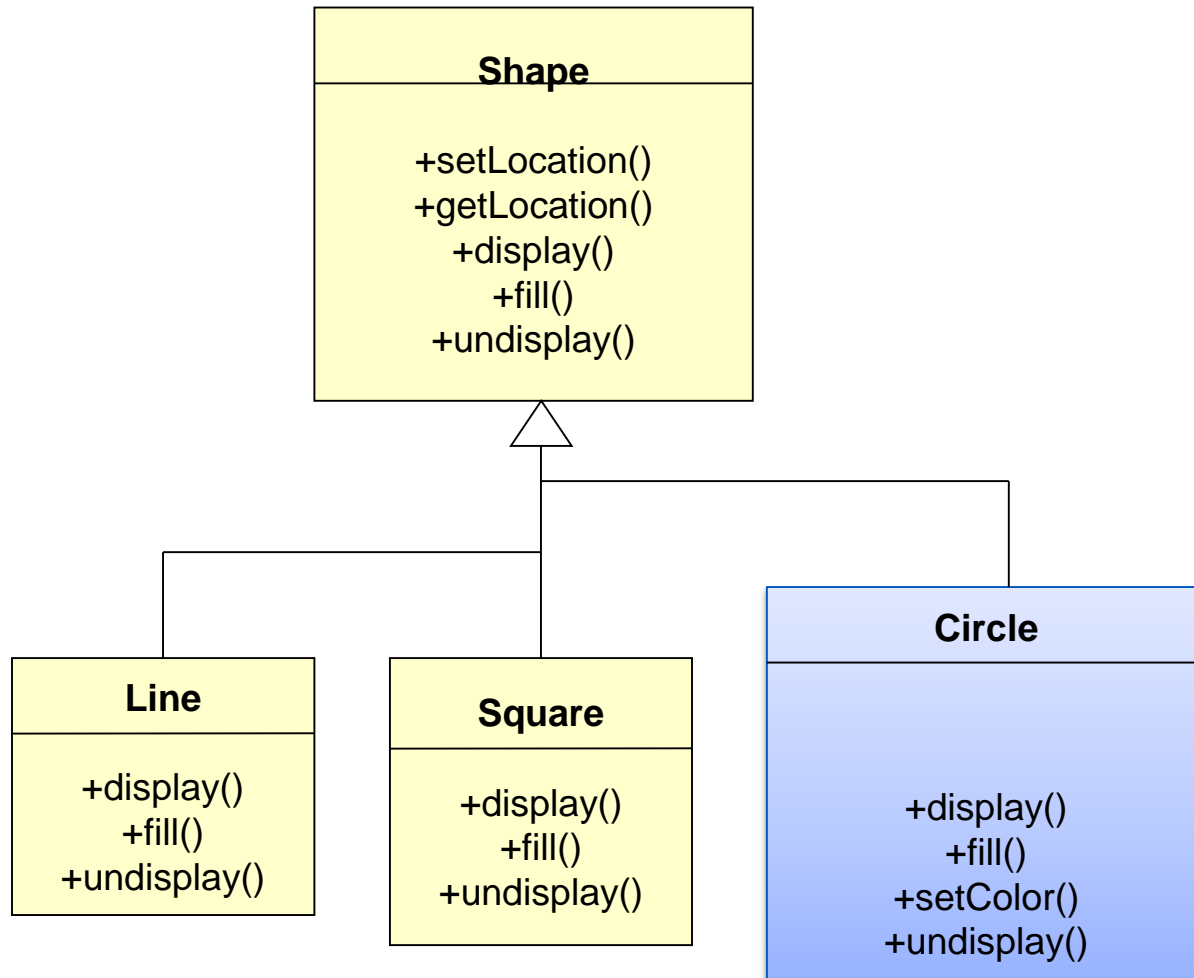


Problem Specification:

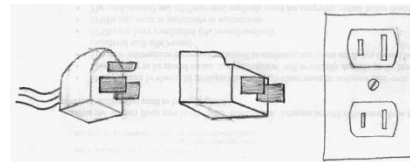
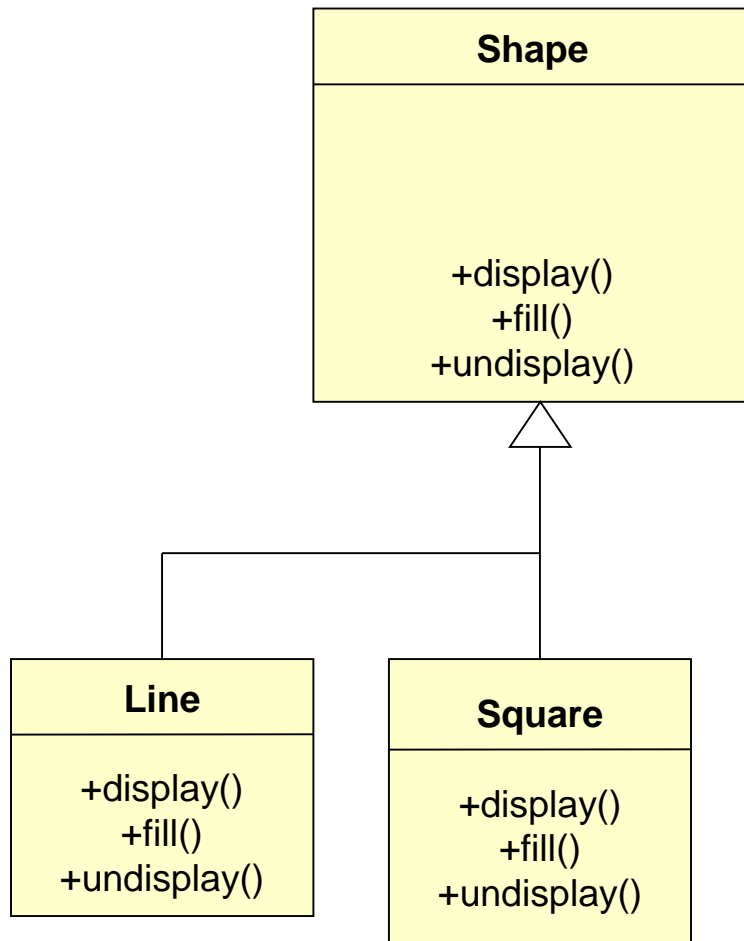
1. You are given the following class library to draw shapes.



2. Now you are asked to add another class to deal with circle. Your plan was:

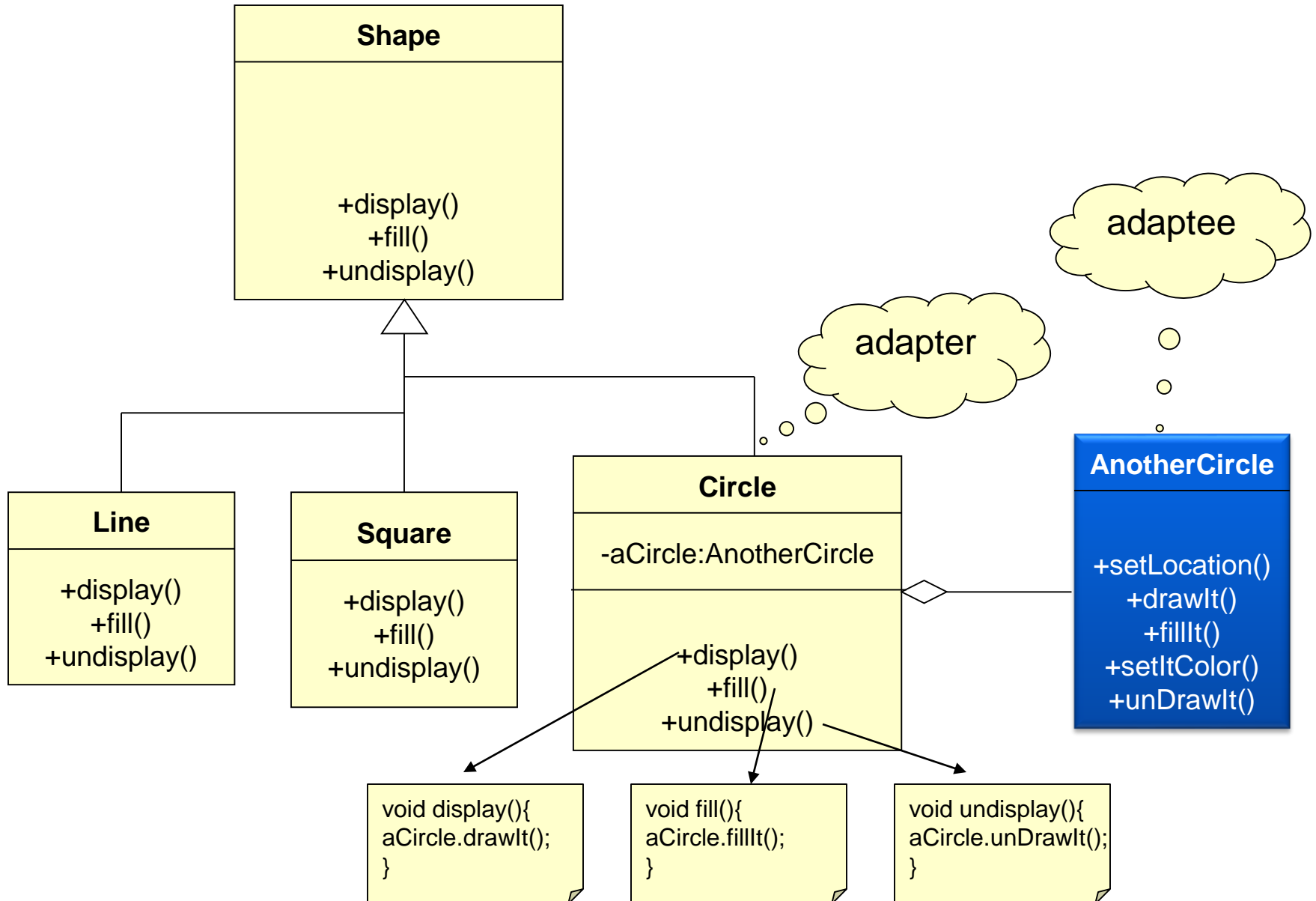


3. Then your client said: "No,no,no". You have to use this nice class library for drawing circles. Unfortunately, it is from a different vendor and it has a different interface.



Adapter Design Pattern Solution

Object Adapter



Example

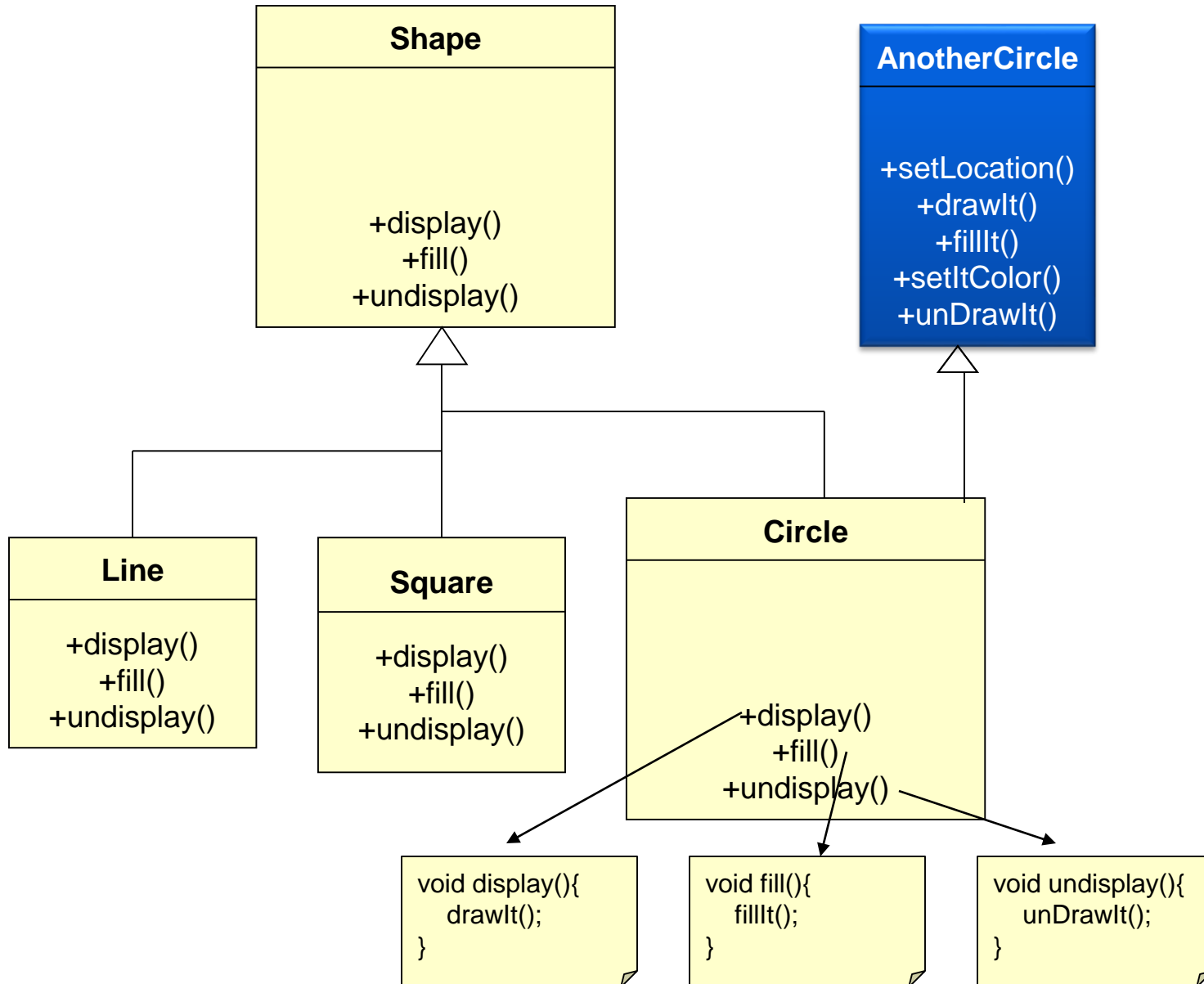
```
class Circle extends Shape {  
    ...  
    private AnotherCircle anotherCircle;  
    public Circle() {  
        anotherCircle = new AnotherCircle();  
    }  
    public void display()  
        another.displayIt();  
    }  
    ...  
}
```

Client Example

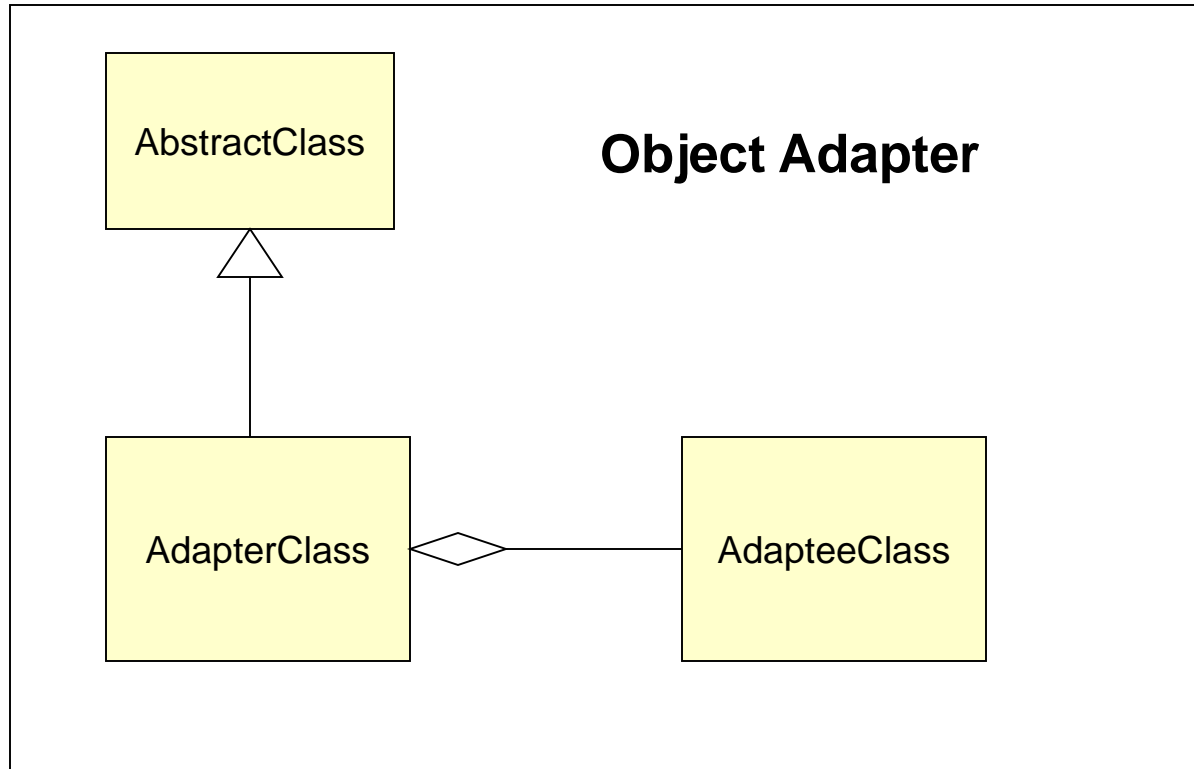
```
class Client {  
    ...  
    Shape aShape = new Circle;  
    aShape.display(); //call Circle::display ()  
    ...  
}
```

Adapter Design Pattern Solution

Class Adapter

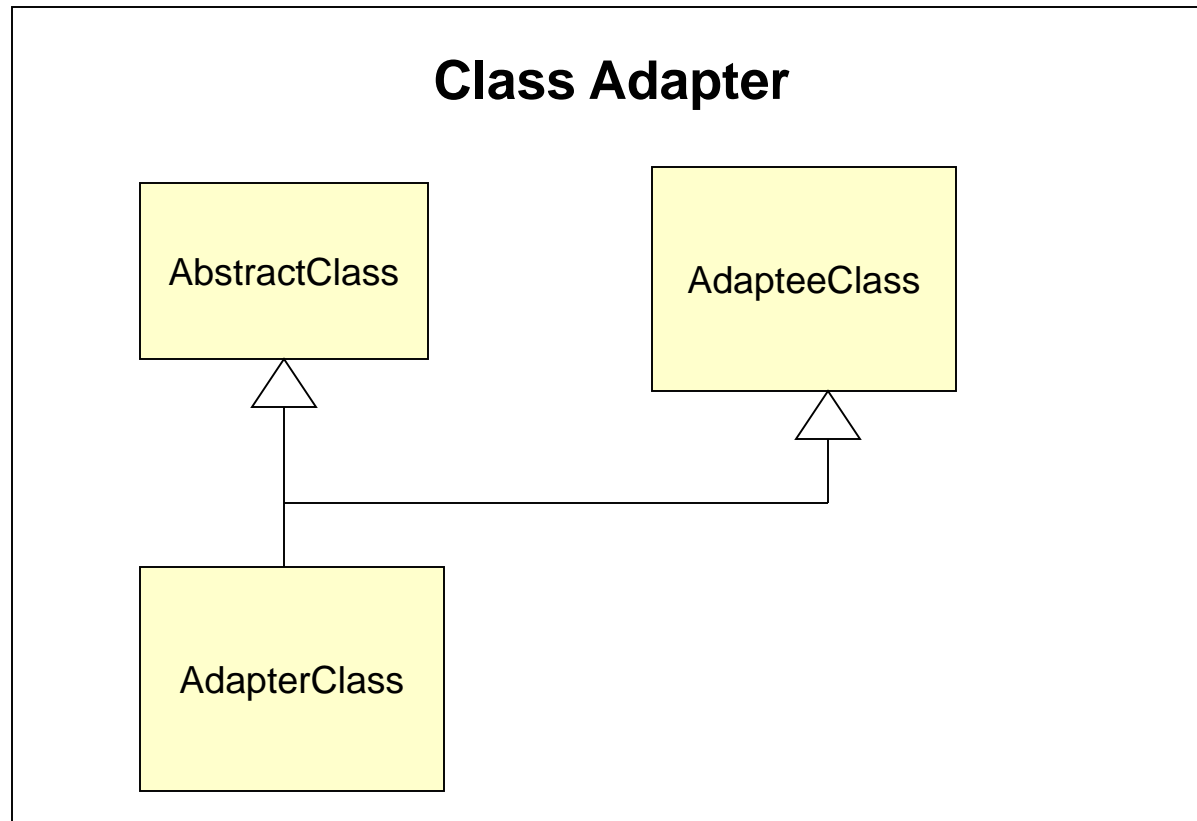


Adapter pattern Class Diagram



- **Object adapter** = The adapter class contains adaptee object

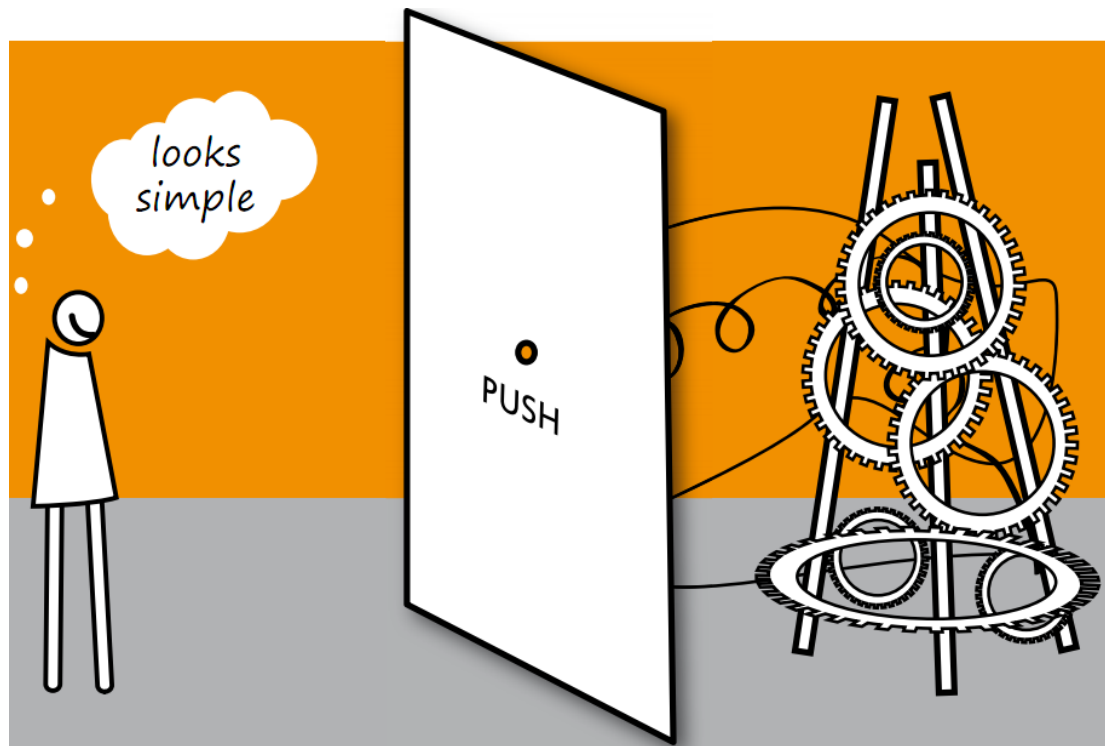
Second way - Class Adapter



- **Class adapter** = Adapter class is inherited from both adaptee and abstract class

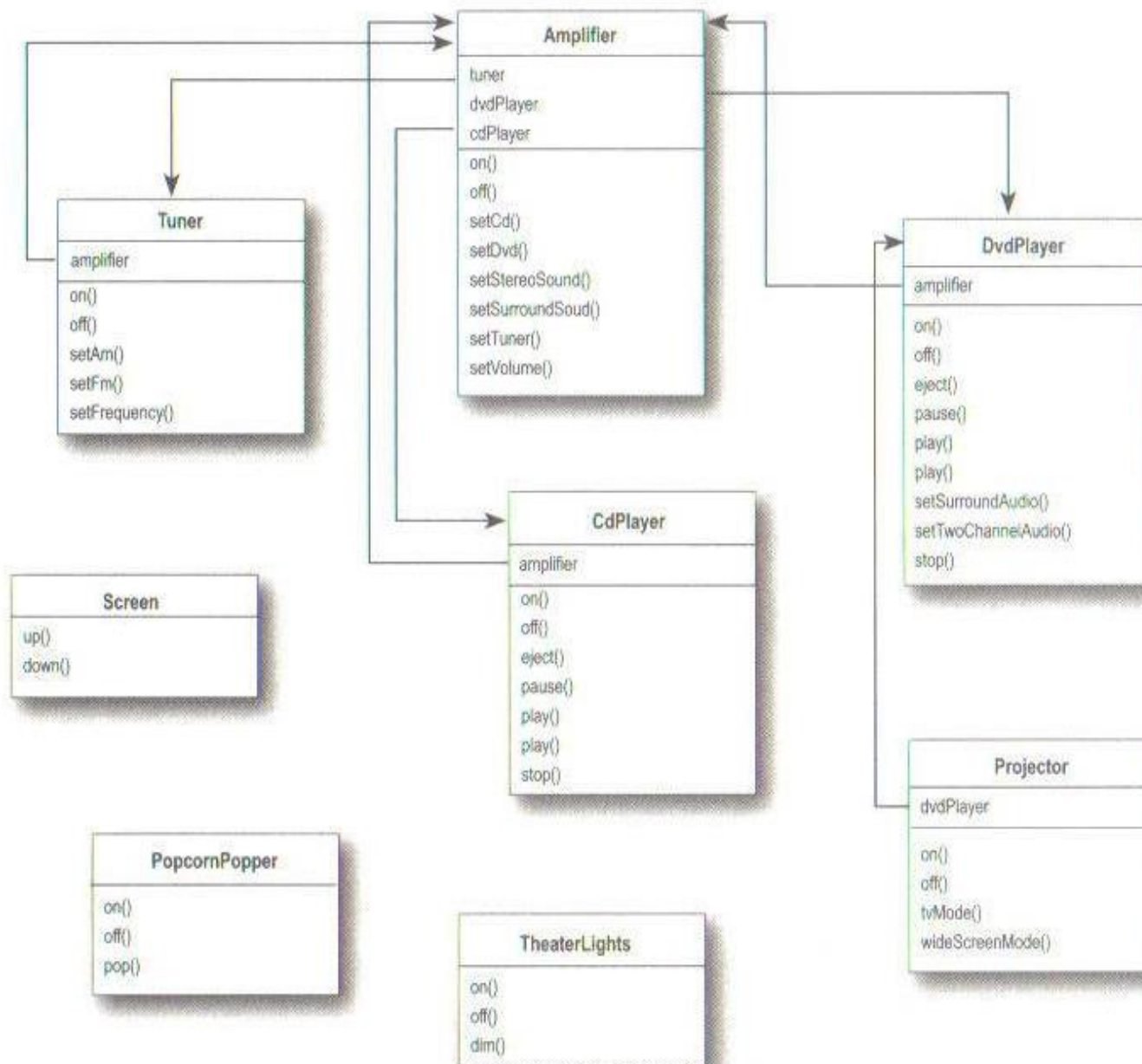
Façade Pattern

Simplifying complex subsystems



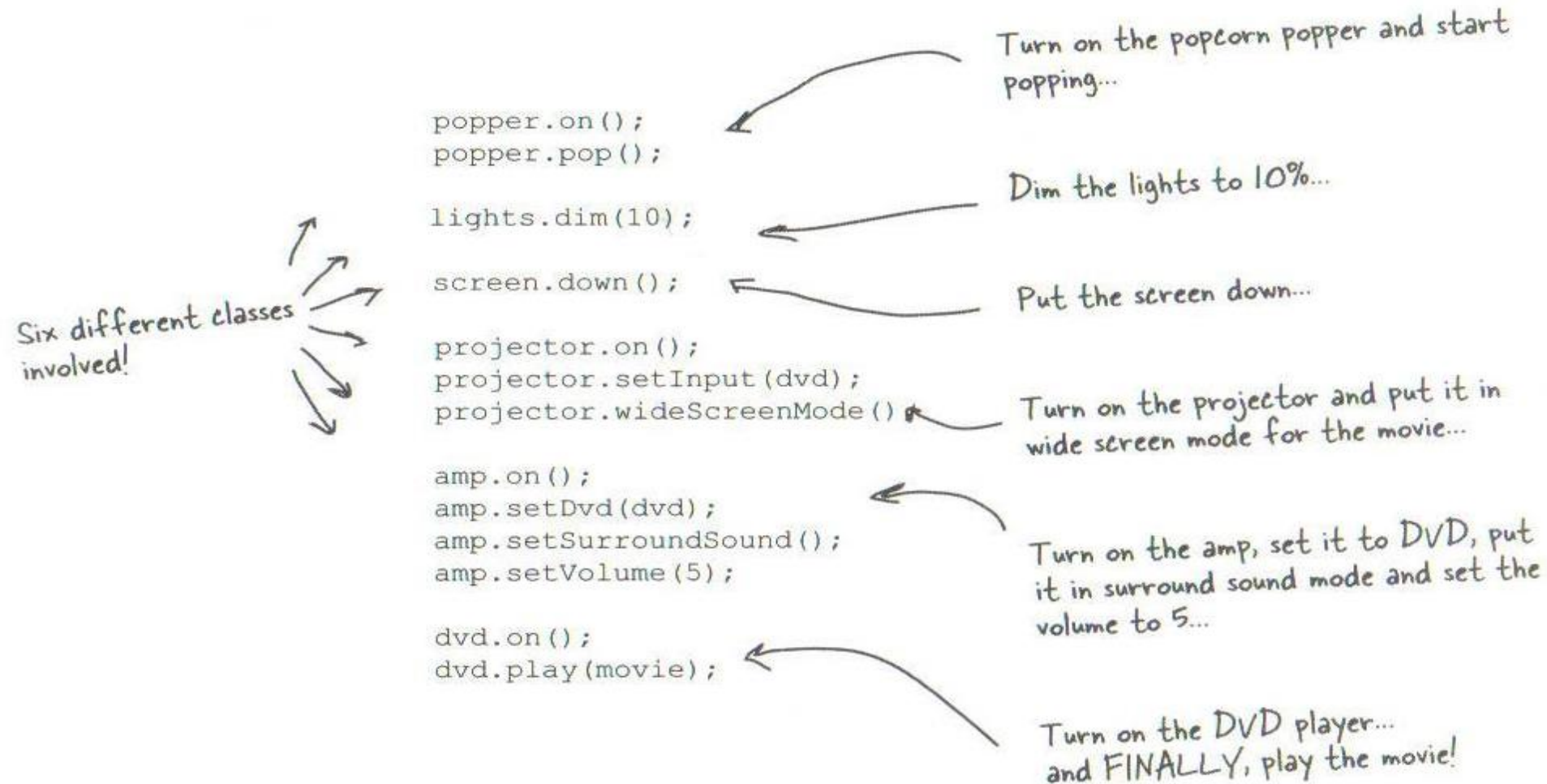
Watching the movie the hard way....

- 1 Turn on the popcorn popper
- 2 Start the popper popping
- 3 Dim the lights
- 4 Put the screen down
- 5 Turn the projector on
- 6 Set the projector input to DVD
- 7 Put the projector on wide-screen mode
- 8 Turn the sound amplifier on
- 9 Set the amplifier to DVD input
- 10 Set the amplifier to surround sound
- 11 Set the amplifier volume to medium (5)
- 12 Turn the DVD Player on
- 13 Start the DVD Player playing



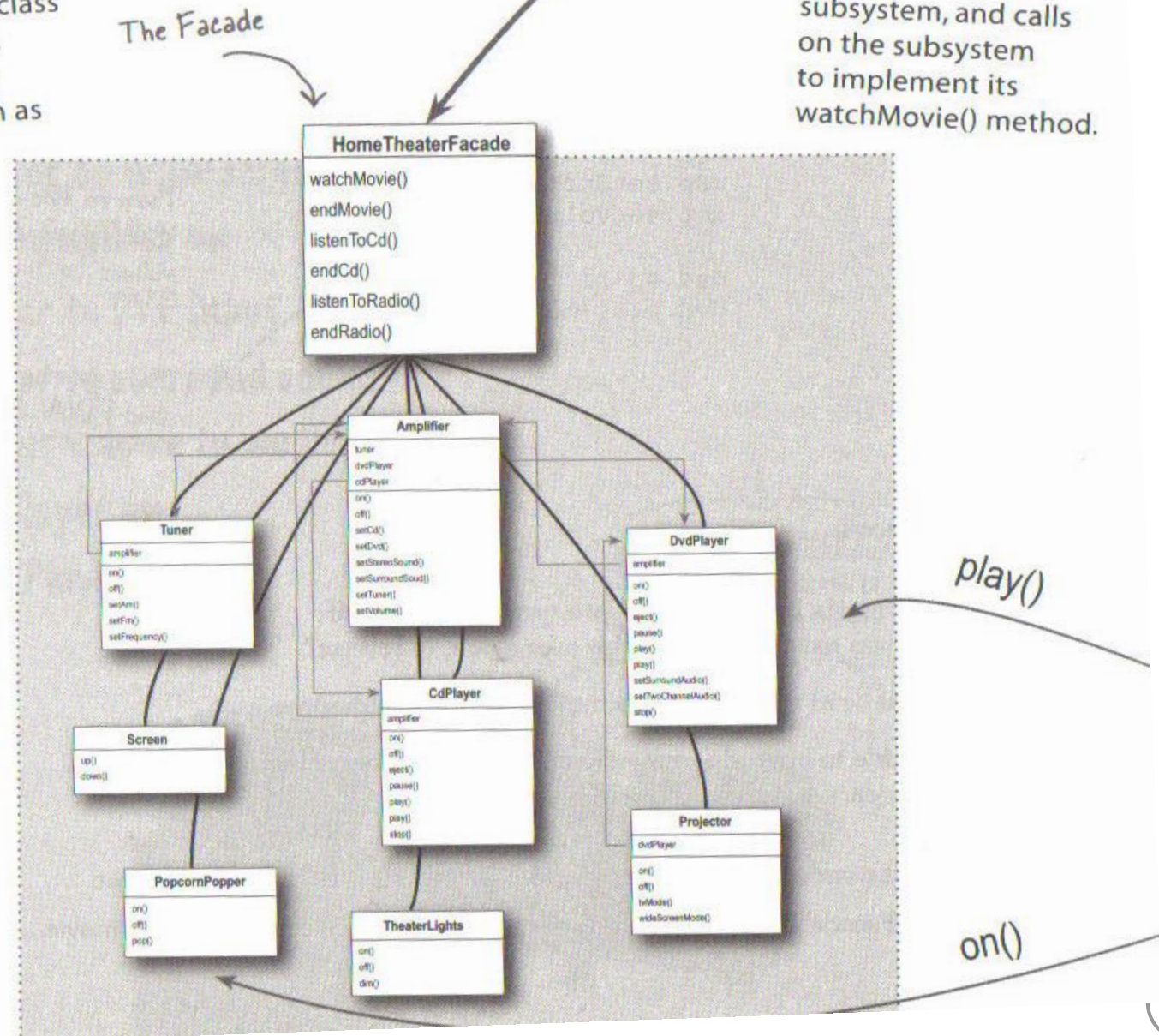
That's a lot of classes, a lot of interactions, and a big set of interfaces to learn and use

What needs to be done to watch a movie....



1 Okay, time to create a Facade for the home theater system. To do this we create a new class HomeTheaterFacade, which exposes a few simple methods such as watchMovie().

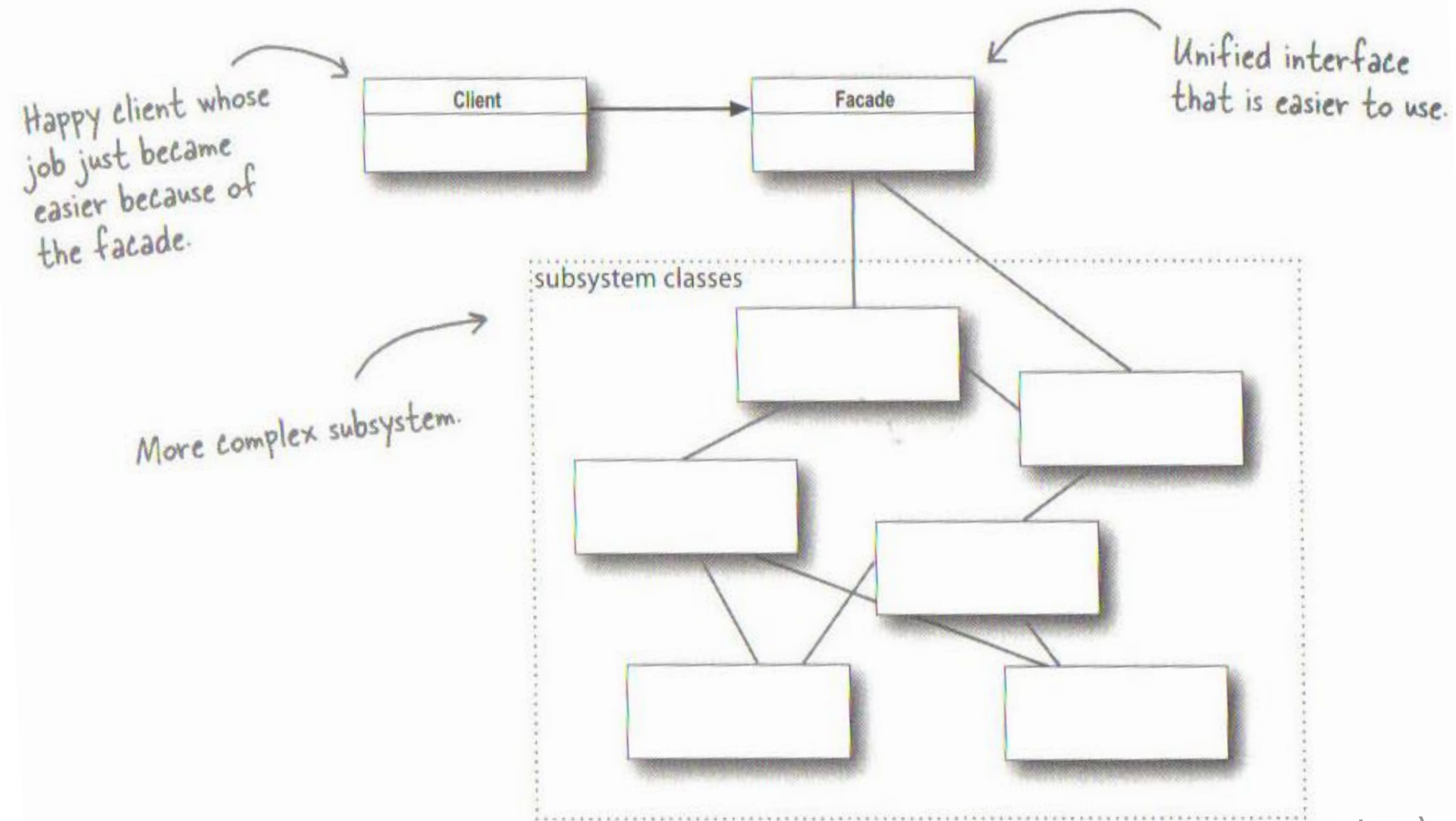
2 The Facade class treats the home theater components as a subsystem, and calls on the subsystem to implement its watchMovie() method.



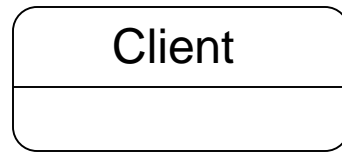
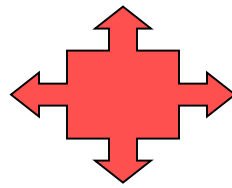
Façade Pattern defined

- The Façade Pattern provides a unified interface to a set of interfaces in a subsystem.
- Façade defines a higher level interface that makes the subsystem easier to use.

Façade pattern - Class Diagram

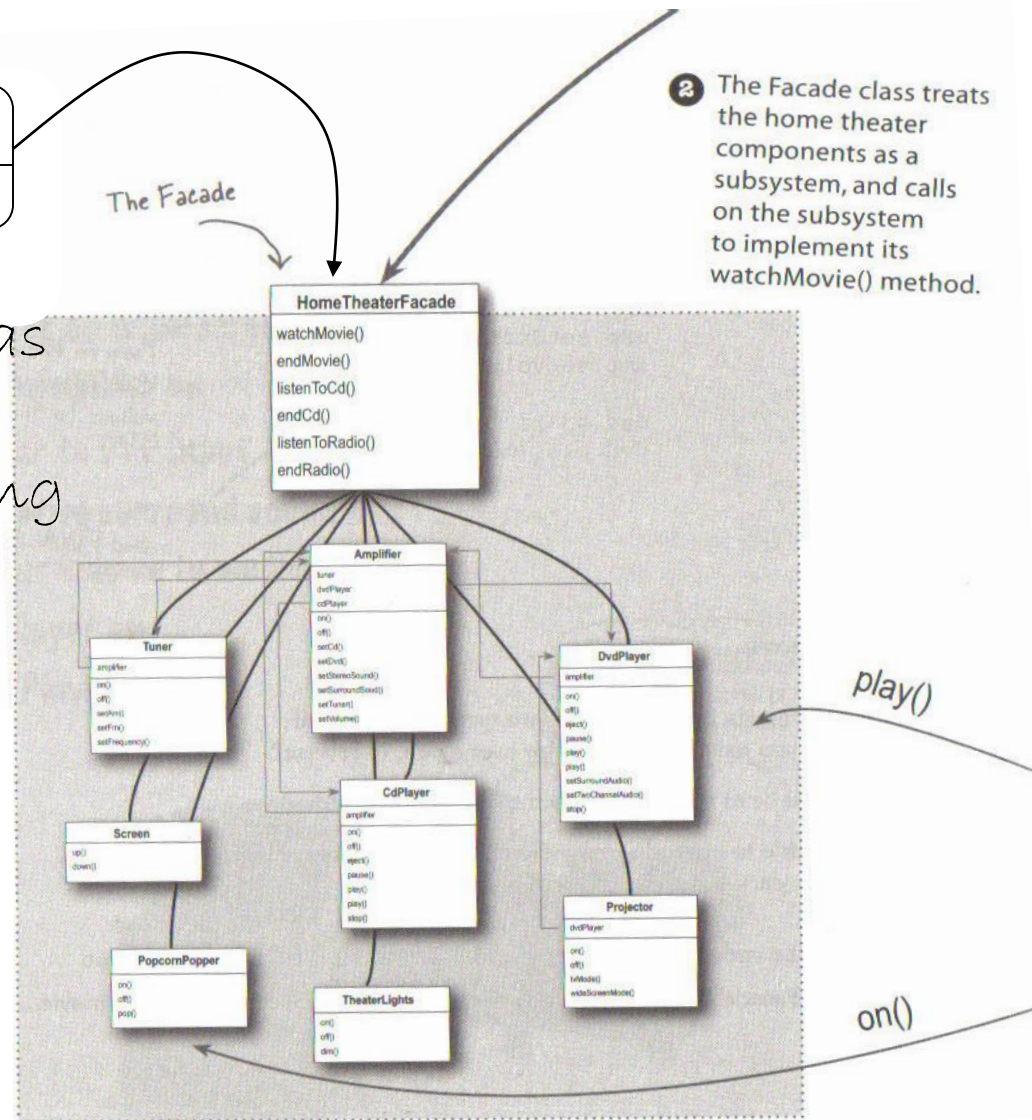


Design Principle



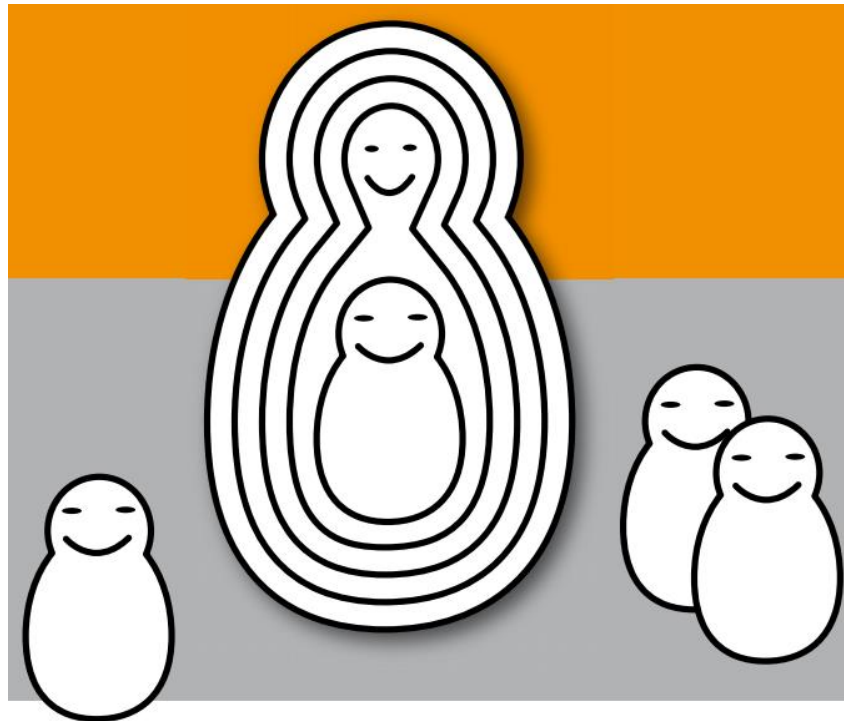
The client only has one friend - and that is a good thing

If the subsystem gets too complicated One can recursively apply the same principle.

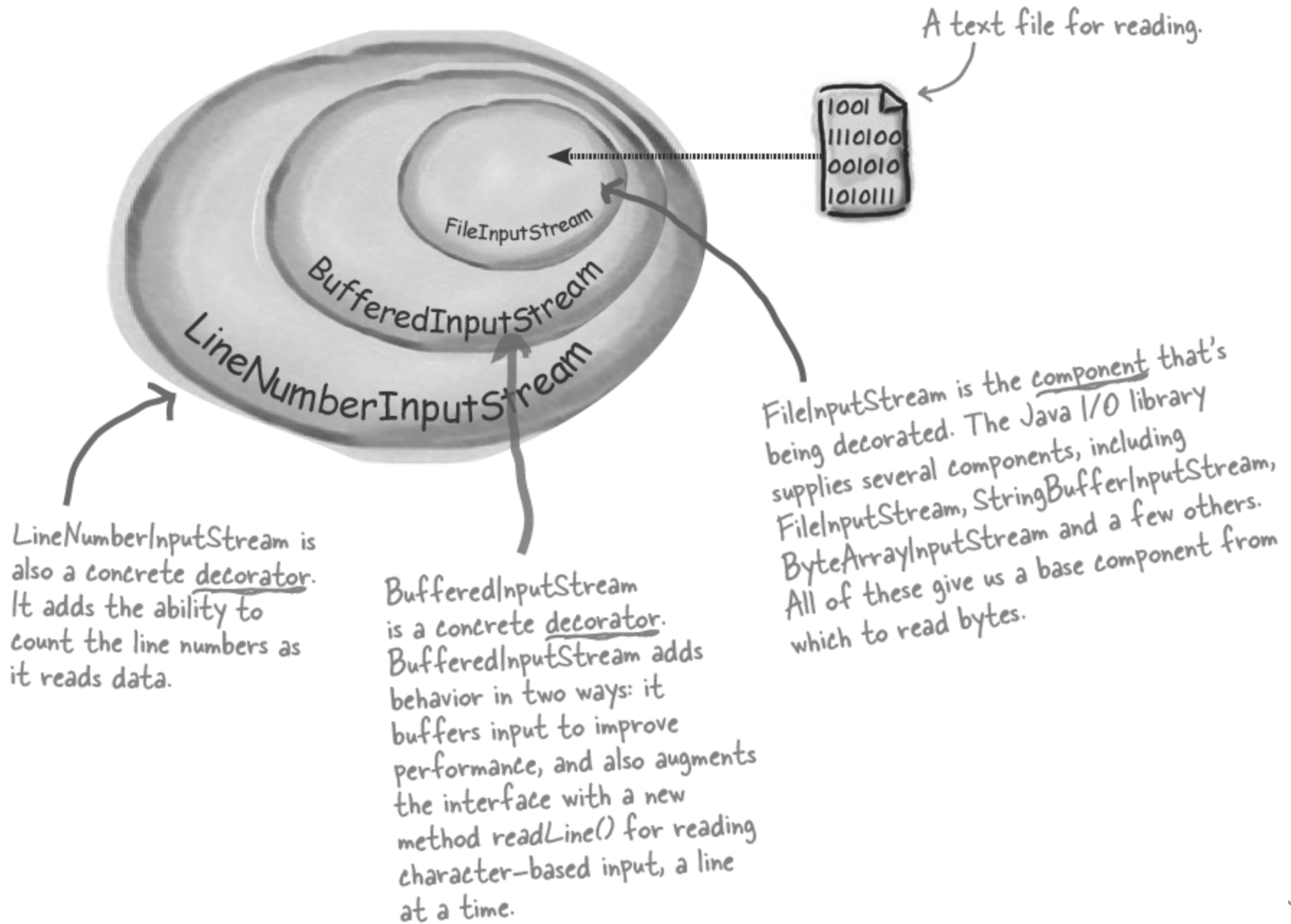


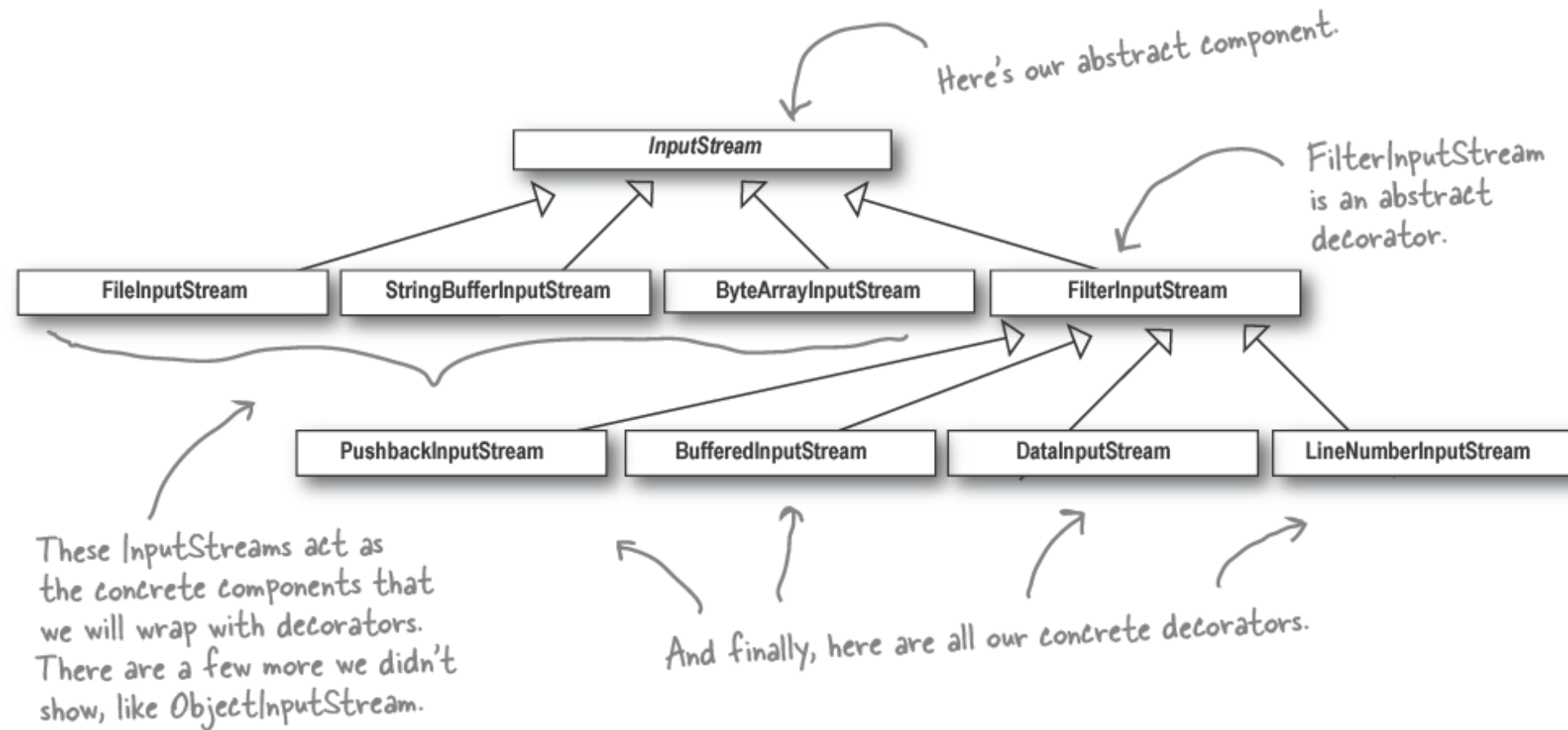
**Principle of Least Knowledge = talk only to your immediate friends
=> Basically this says minimize your dependencies**

Decorator Pattern



Real World Decorators: Java I/O





Writing your own Java I/O Decorator

Don't forget to import
java.io... (not shown)

First, extend the FilterInputStream, the
abstract decorator for all InputStream

No problem. I just have to
extend the FilterInputStream class
and override the read() methods.

```
public class LowerCaseInputStream extends FilterInputStream {  
    public LowerCaseInputStream(InputStream in) {  
        super(in);  
    }  
  
    public int read() throws IOException {  
        int c = super.read();  
        return (c == -1 ? c : Character.toLowerCase((char)c));  
    }  
  
    public int read(byte[] b, int offset, int len) throws IOException {  
        int result = super.read(b, offset, len);  
        for (int i = offset; i < offset+result; i++) {  
            b[i] = (byte)Character.toLowerCase((char)b[i]);  
        }  
        return result;  
    }  
}
```

Now we need to implement two
read methods. They take a
byte (or an array of bytes)
and convert each byte (that
represents a character) to
lowercase if it's an uppercase
character.

Test out your new Java I/O Decorator

```
public class InputTest {  
    public static void main(String[] args) throws IOException {  
        int c;  
        try {  
            InputStream in =  
                new LowerCaseInputStream(  
                    new BufferedInputStream(  
                        new FileInputStream("test.txt")));  
  
            while((c = in.read()) >= 0) {  
                System.out.print((char)c);  
            }  
  
            in.close();  
        } catch (IOException e) {  
            e.printStackTrace();  
        }  
    }  
}
```

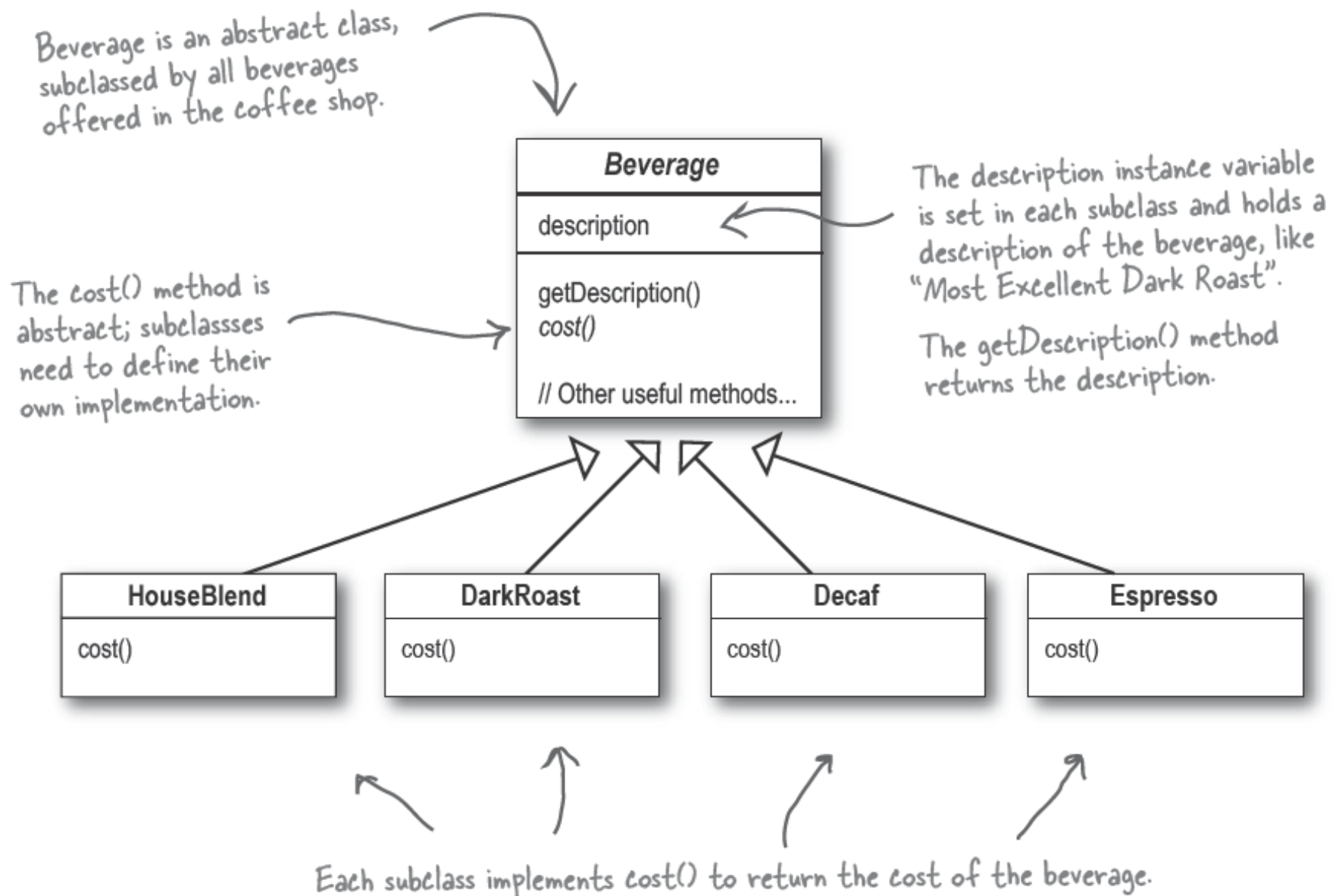
Just use the stream to read characters until the end of file and print as we go.

Set up the `FileInputStream` and decorate it, first with a `BufferedInputStream` and then our brand new `LowerCaseInputStream` filter.

I know the Decorator Pattern therefore I RULE!

test.txt file

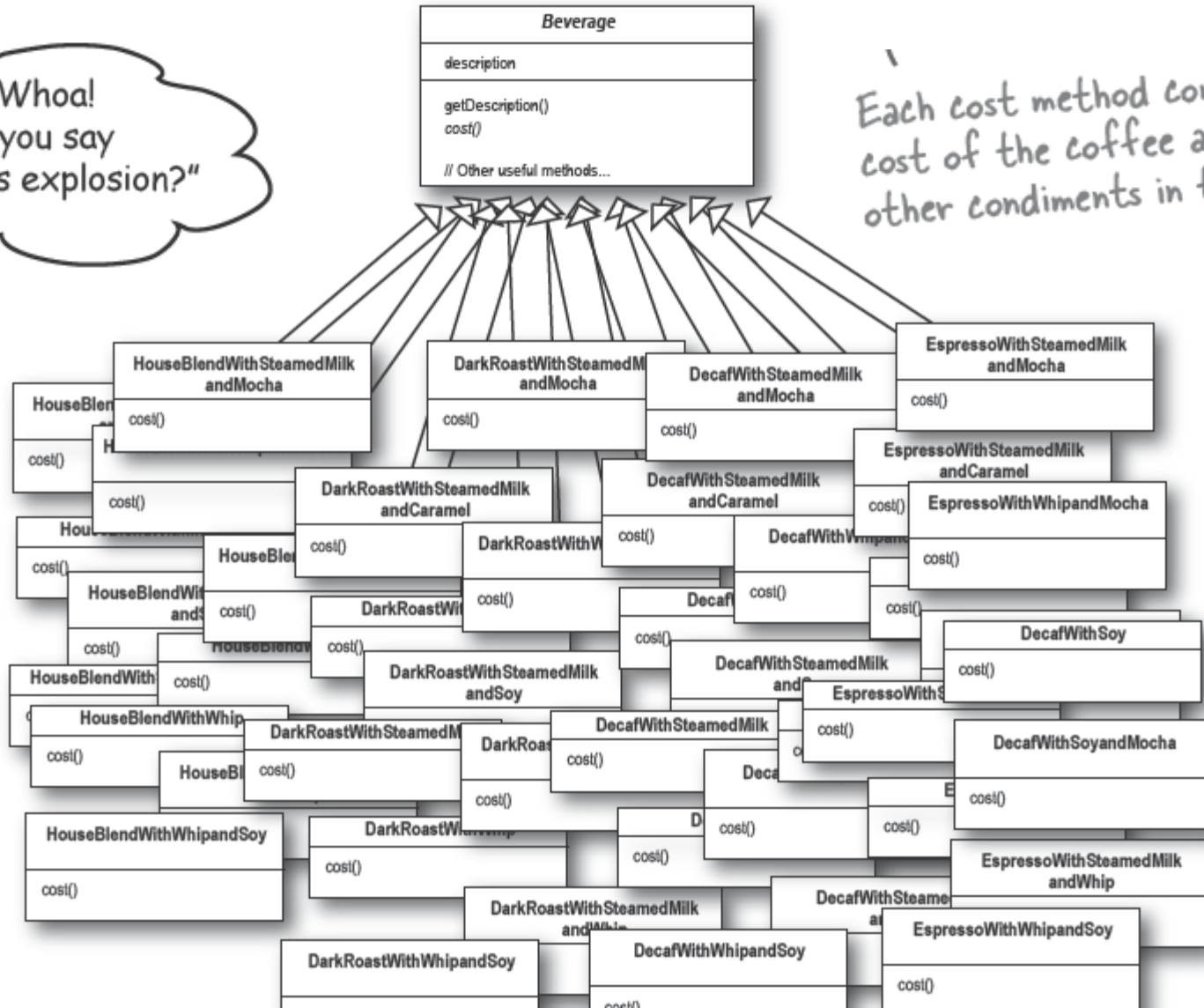
You need to make this file.



What if you want to show the addition of condiments such as steamed milk, soy, mocha and whipped milk?

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.

Here's their first attempt...

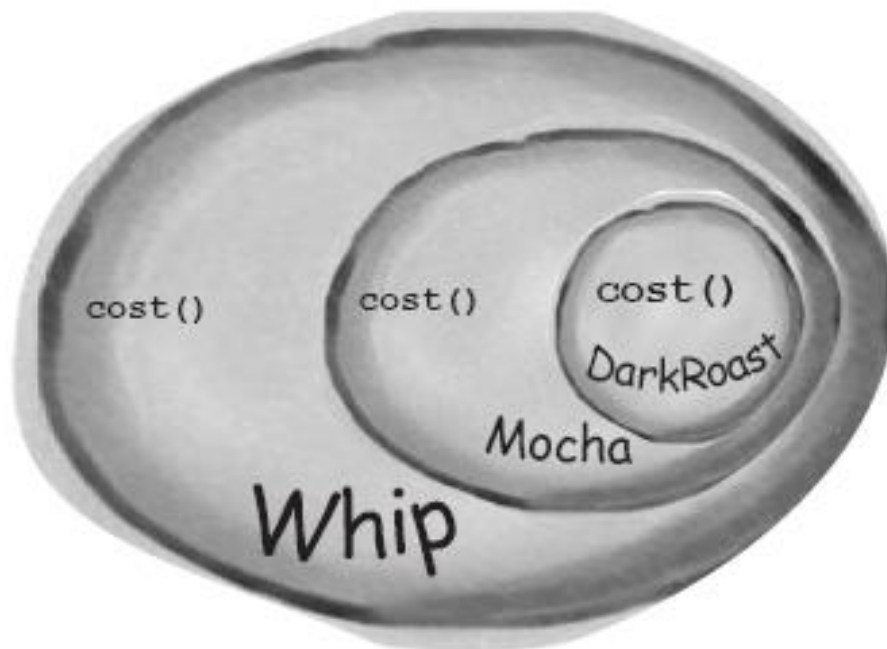


Each cost method computes the cost of the coffee along with the other condiments in the order.

The Decorator Pattern

- Take a coffee beverage object - say DarkRoast object
- Decorate it with Mocha
- Decorate it with Whip
- Call the cost method and rely on delegation to correctly compute the composite cost

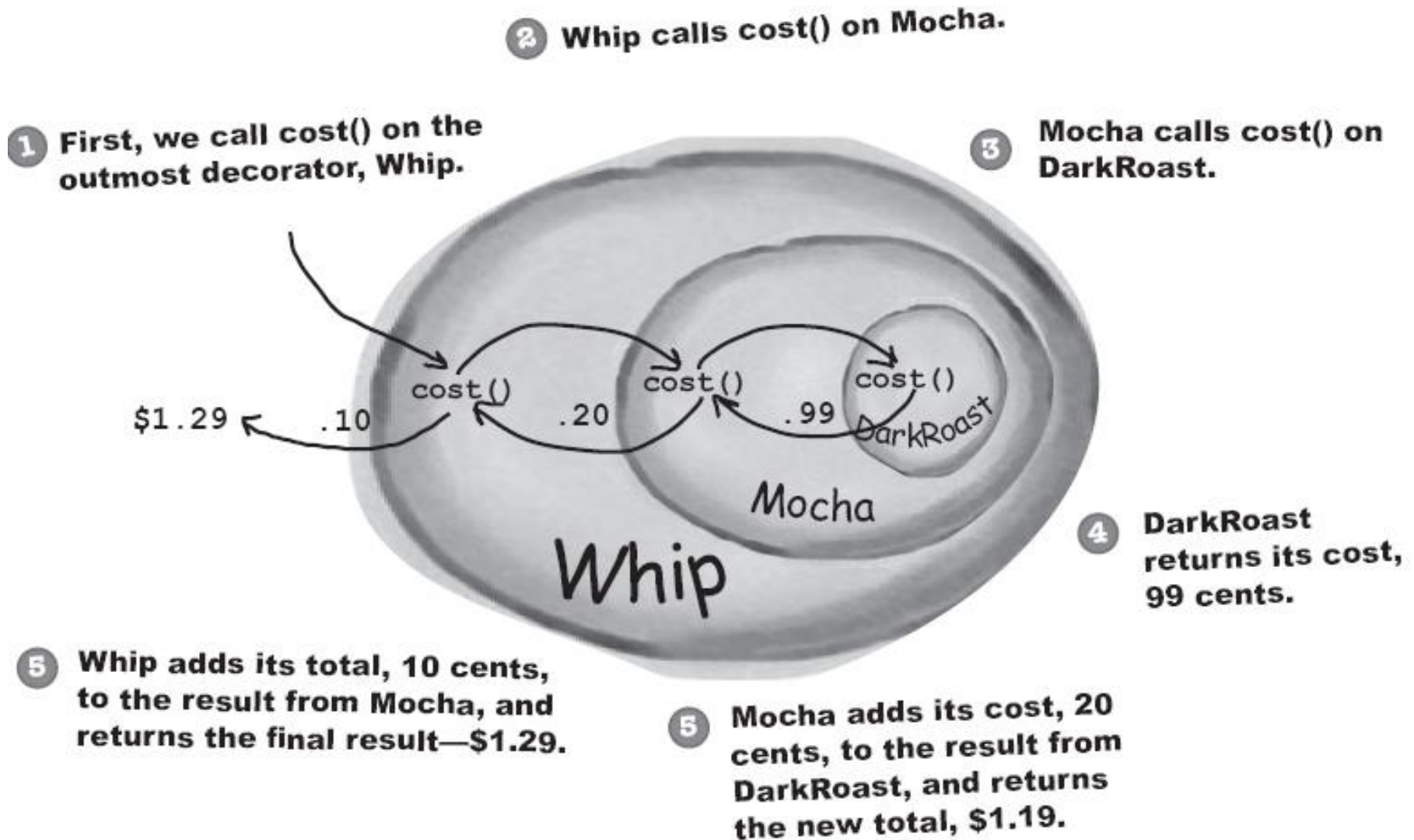
Decorator Pattern approach



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.

Computing Cost using the decorator pattern



Summary

Okay, here's what we know so far...

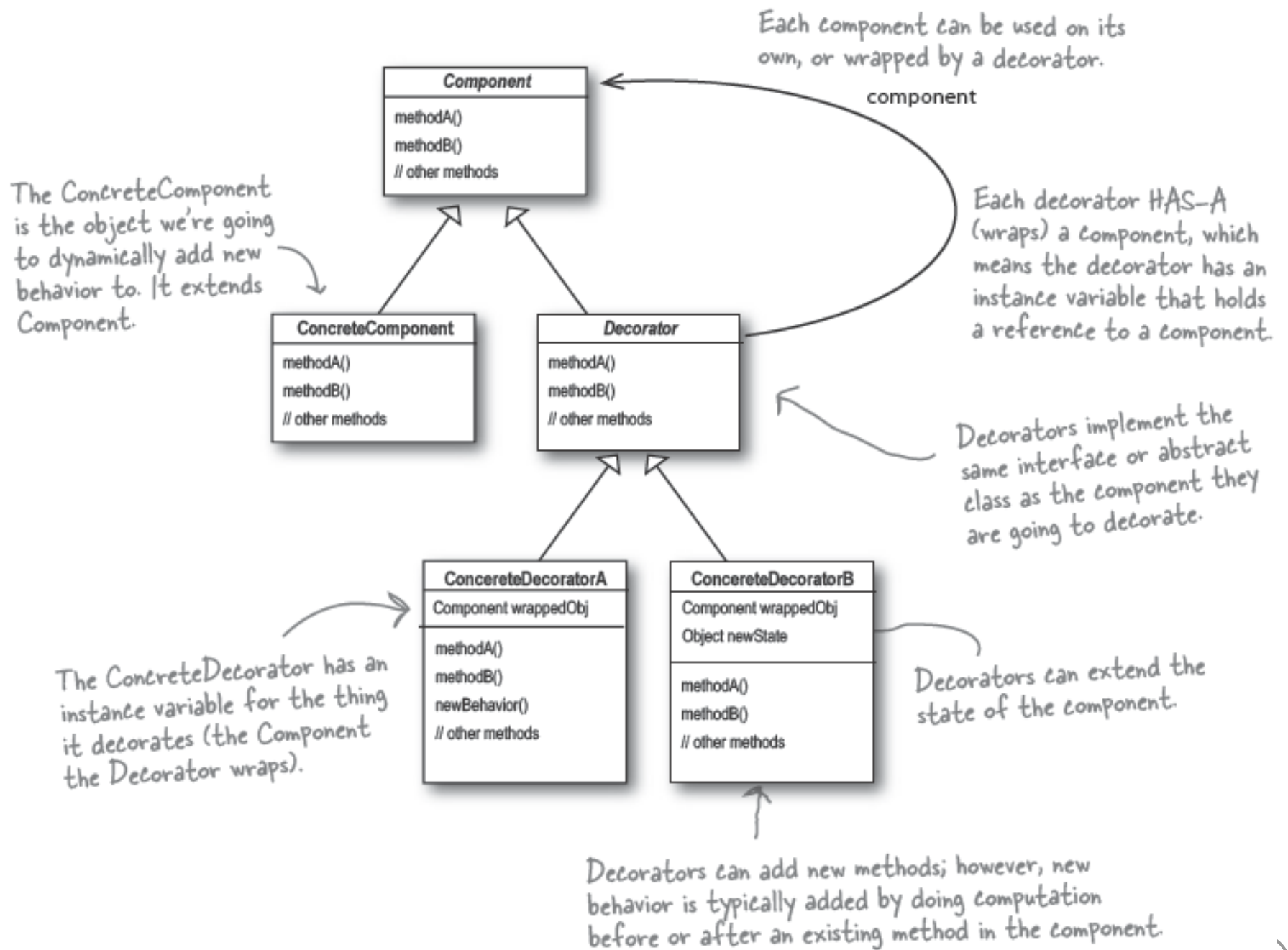
- Decorators have the same supertype as the objects they decorate.
- You can use one or more decorators to wrap an object.
- Given that the decorator has the same supertype as the object it decorates, we can pass around a decorated object in place of the original (wrapped) object.
- The decorator adds its own behavior either before and/or after delegating to the object it decorates to do the rest of the job.
- Objects can be decorated at any time, so we can decorate objects dynamically at runtime with as many decorators as we like.

Key Point!

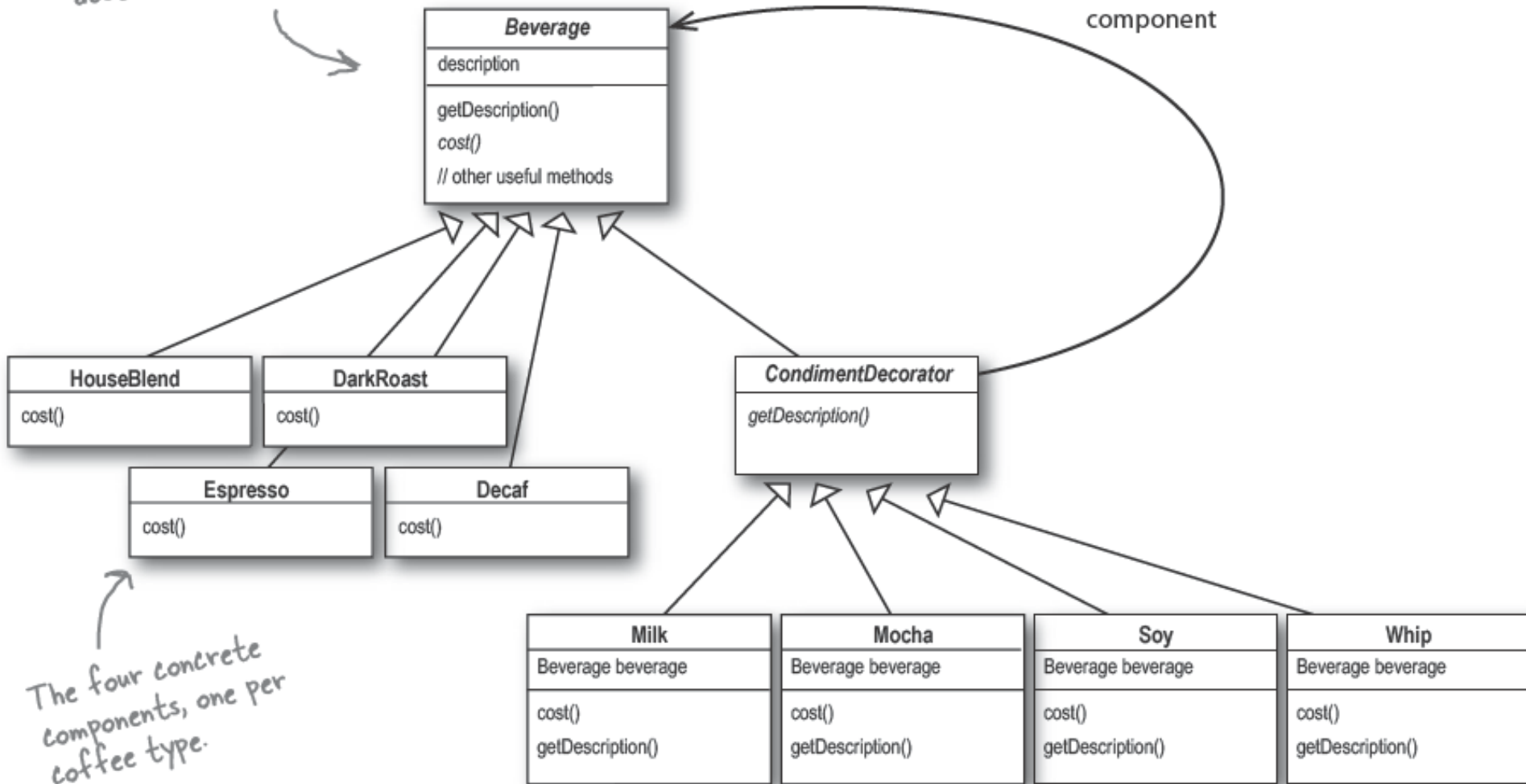
Decorator Pattern

The decorator pattern **attaches additional responsibilities** to an object dynamically.

Decorators provide a flexible alternative to sub-classing for extending functionality.



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...



OO Principles

Encapsulate what varies.

Favor composition over inheritance.

Program to interfaces, not implementations.

Strive for loosely coupled designs between objects that interact.

Classes should be open for extension but closed for modification.

OO Patterns

Strat

encap

inter

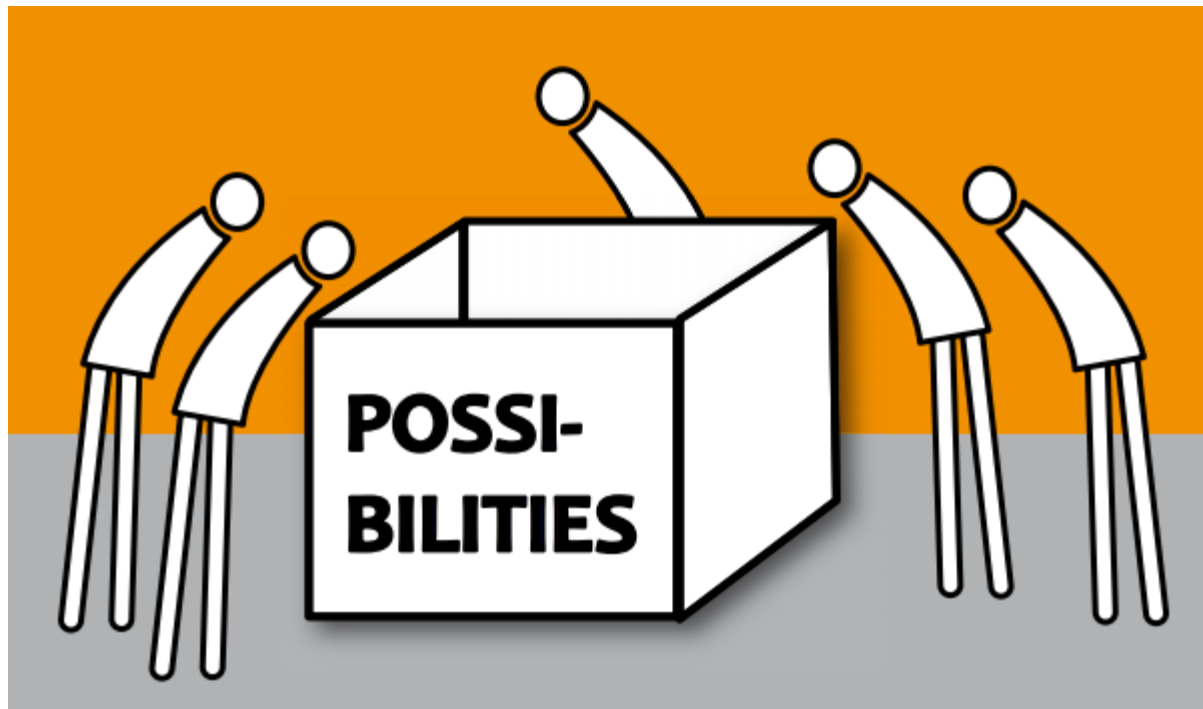
vary

Observer

Decorator - Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

← We now have the Open-Closed Principle to guide us. We're going to strive to design our system so that the closed parts are isolated from our new extensions.

Strategy Pattern



Strategy Pattern

- Define a family of algorithms, encapsulates each one, and makes them interchangeable.
- Strategy lets the algorithm vary independently from clients that use it.
- Design Principle:

Identify the aspects of the application that vary and separate them from what stays the same.

Multiple *SalePricingStrategy* classes with polymorphic *getTotal* method

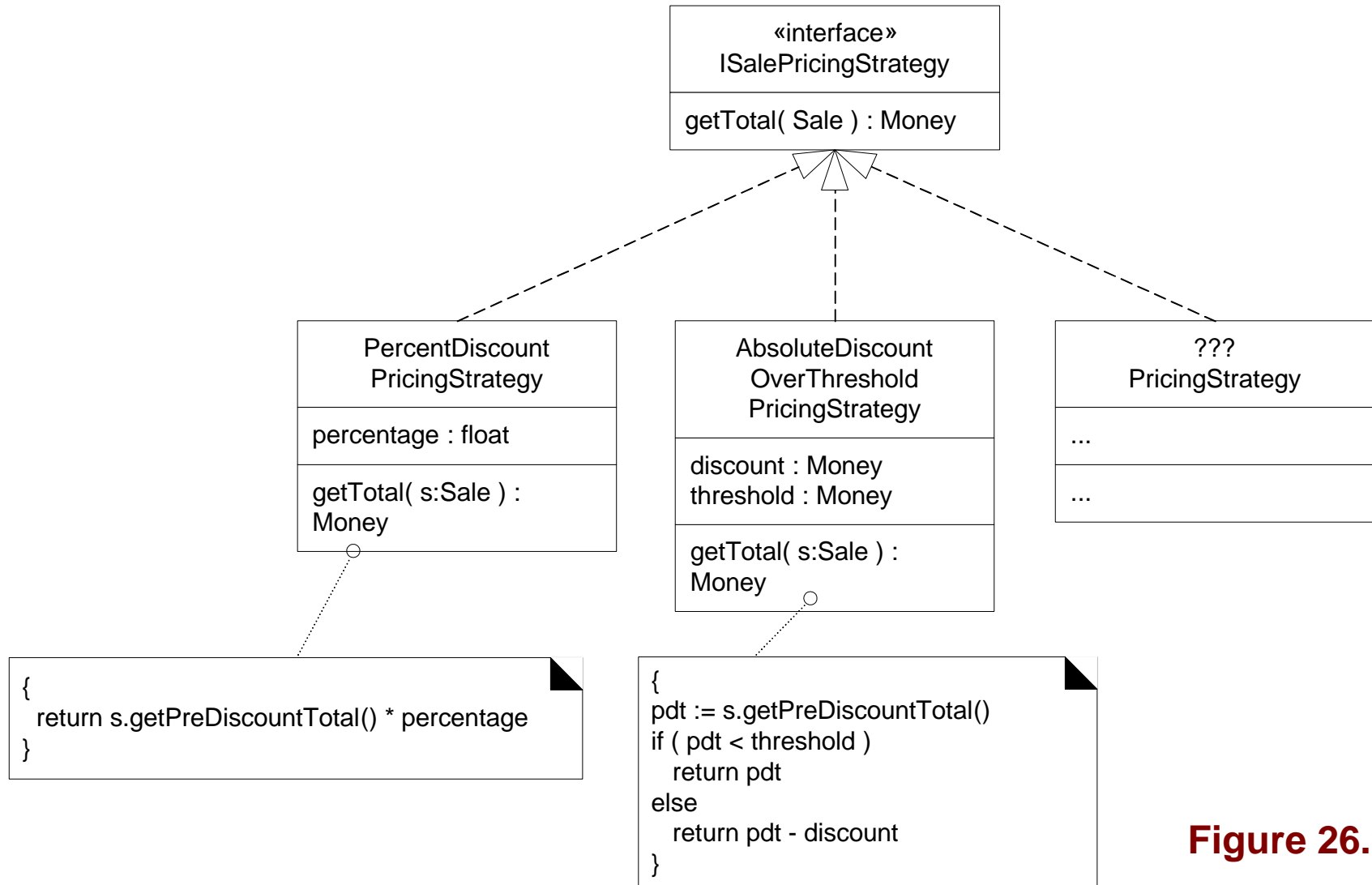
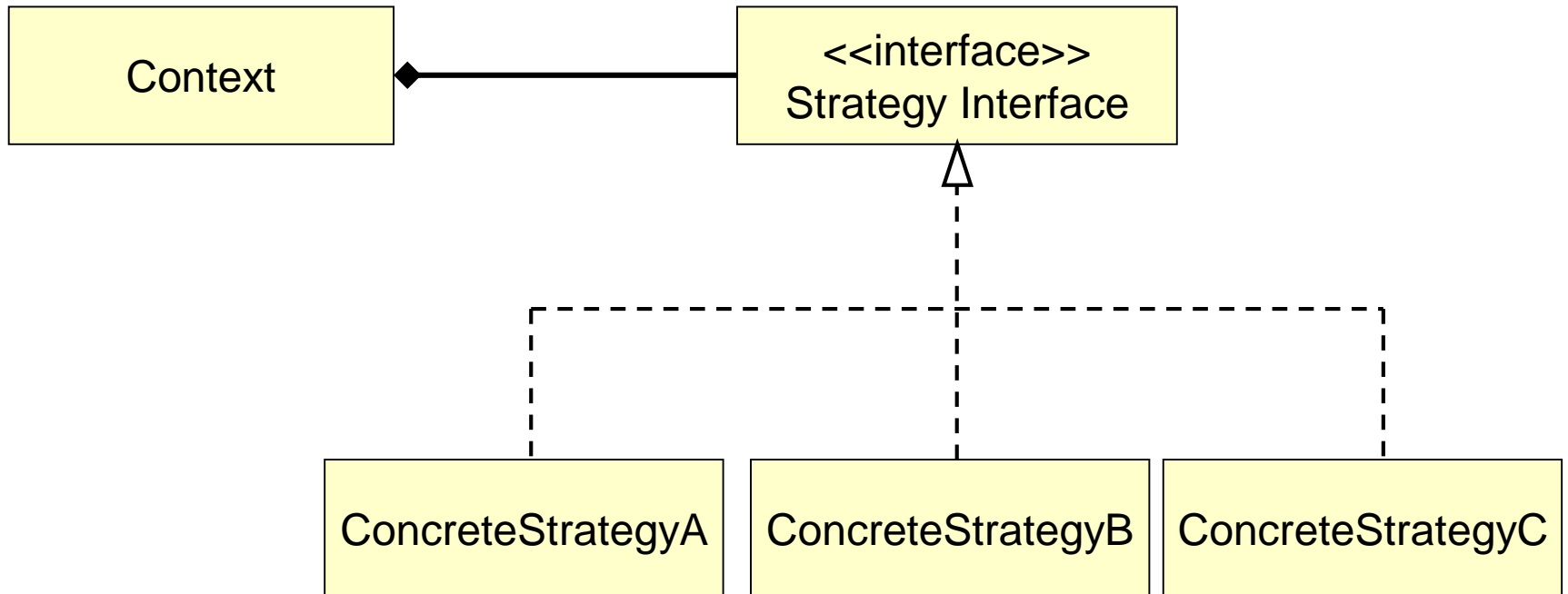


Figure 26.9

Strategy Class Diagram



Implement the Strategy pattern

- Define a Strategy interface for your strategy objects
- Implement ConcreteStrategy classes that implement the Strategy interface, as appropriate
- In your Context class, maintain a private reference to a Strategy object.
- In your Context class, implement public setter and getter methods for the Strategy object .

Context class

```
class Context {  
    IStrategy strategy;  
    // Constructor  
    public Context(IStrategy strategy) {  
        this.strategy = strategy;  
    }  
    public void execute() {  
        strategy.execute();  
    }  
}
```

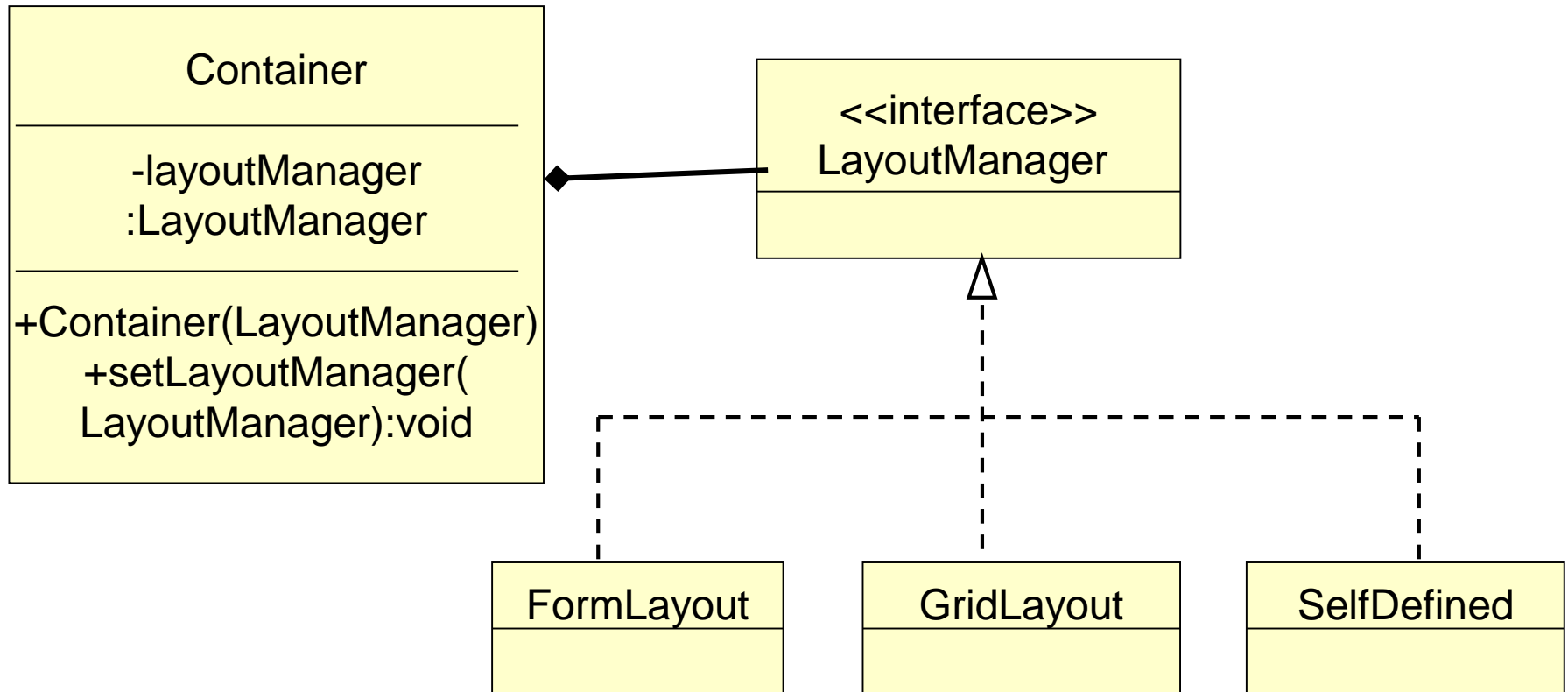
Strategy interface and concrete strategies

```
interface IStrategy {  
    void execute();  
}  
// Implements the algorithm using the strategy interface  
class ConcreteStrategyA implements IStrategy {  
    public void execute() {  
        System.out.println( "Called ConcreteStrategyA.execute()" );  
    }  
}  
class ConcreteStrategyB implements IStrategy {  
    public void execute() {  
        System.out.println( "Called ConcreteStrategyB.execute()" );  
    }  
}  
class ConcreteStrategyC implements IStrategy {  
    public void execute() {  
        System.out.println( "Called ConcreteStrategyC.execute()" );  
    }  
}
```

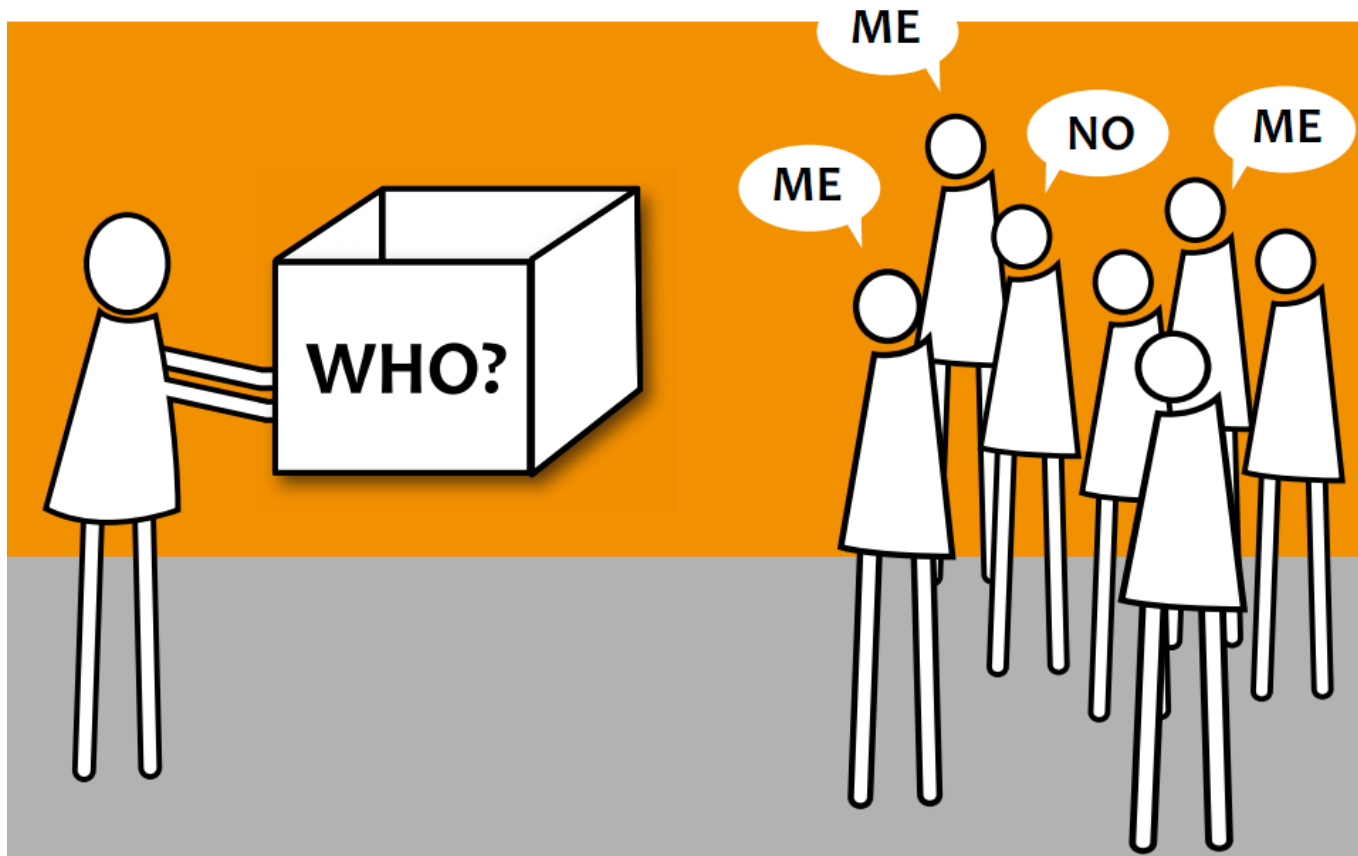

Applications

```
class MainApp {  
    public static void main() {  
        Context context;  
        // Three contexts following different strategies  
        context = new Context(new ConcreteStrategyA());  
        context.execute();  
        context = new Context(new ConcreteStrategyB());  
        context.execute();  
        context = new Context(new ConcreteStrategyC());  
        context.execute();  
    }  
}
```

Layout Manager Sample



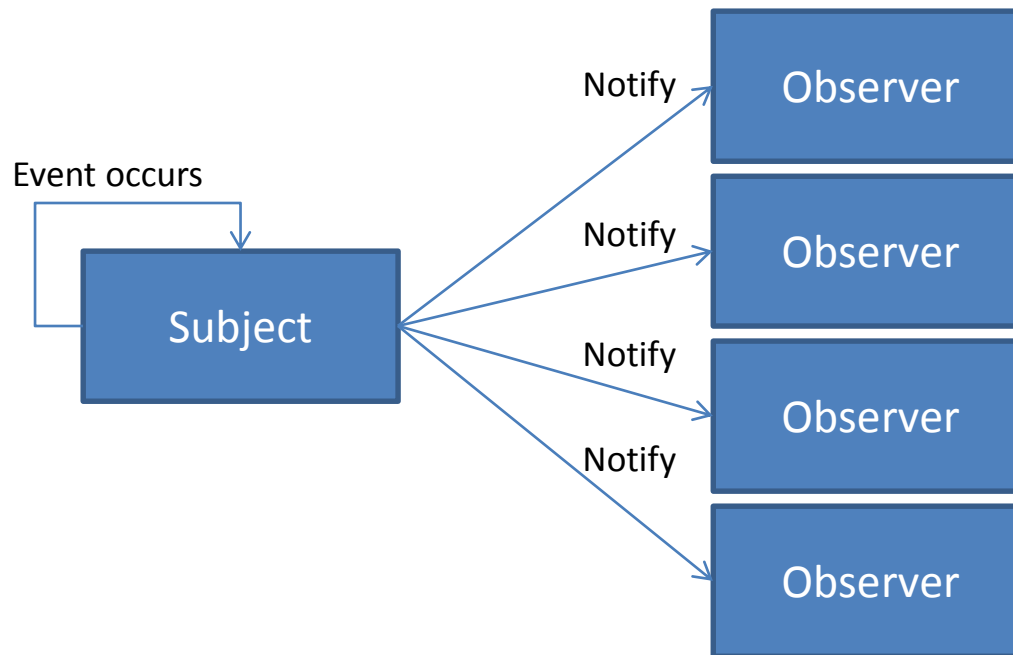
Observer Pattern



Observer Pattern

Observer Pattern

The observer pattern allows for an object to maintain a list of all objects that wish to be notified should a certain *Event* occur.



Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

```
static void Main()
{
    // Create IBM stock and attach investors
    Stock ibm = new Stock("IBM", 120.00);
    ibm.Attach(new Investor("Sorros"));
    ibm.Attach(new Investor("Berkshire"));

    // Fluctuating prices will notify investors
    ibm.Price = 120.10;
    ibm.Price = 121.00;
    ibm.Price = 120.50;
    ibm.Price = 120.75;

    // Wait for user
    Console.ReadKey();
}
```

```
abstract class Stock {
    private string _symbol;    private double _price;
    private List<IInvestor> _investors = new List<IInvestor>();

    // Constructor
    public Stock(string symbol, double price) {
        this._symbol = symbol;  this._price = price;    }

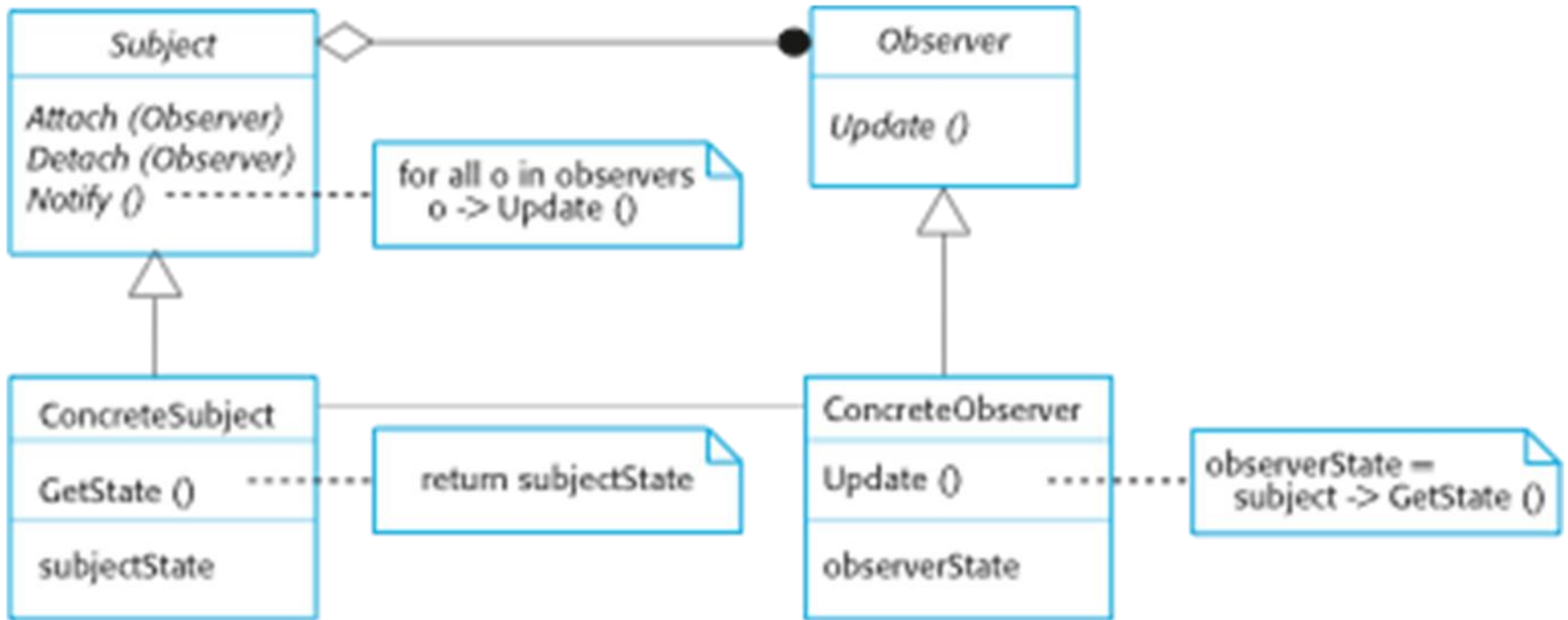
    public void Attach(IInvestor investor) {
        _investors.Add(investor);    }

    public void Detach(IInvestor investor) {
        _investors.Remove(investor);    }

    public void Notify() {
        foreach (IInvestor investor in _investors) {
            investor.Update(this);    }
        Console.WriteLine("");    }

    // Gets or sets the price
    public double Price {
        get { return _price; }
        set { if (_price != value) {
                _price = value; Notify();
            }
        }
    }
}
```

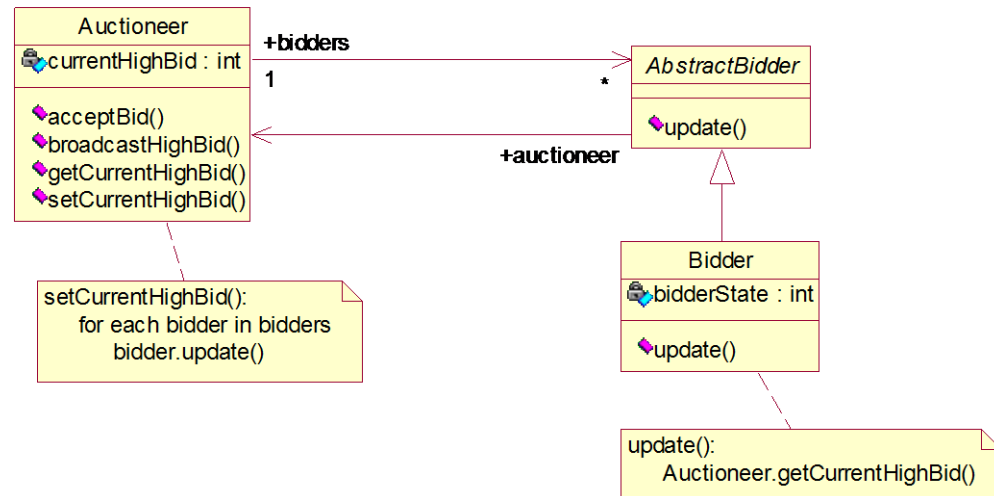
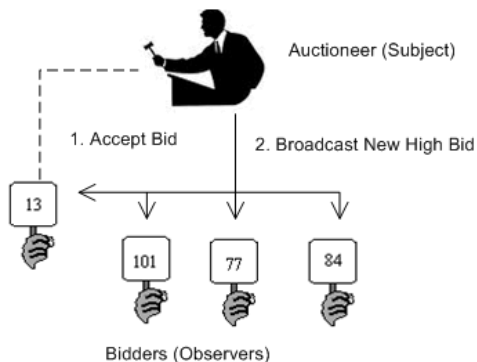
A UML model of the Observer pattern



Observer Pattern Example

IN AN AUCTION/BIDDERS SCENARIO, THE BIDDERS MAY BE VIEWED AS OBSERVERS, WITH THE HIGH BID AS THE OBJECT BEING VIEWED.

The bidders are observers, receiving notification from the auctioneer whenever a new high bid is placed.

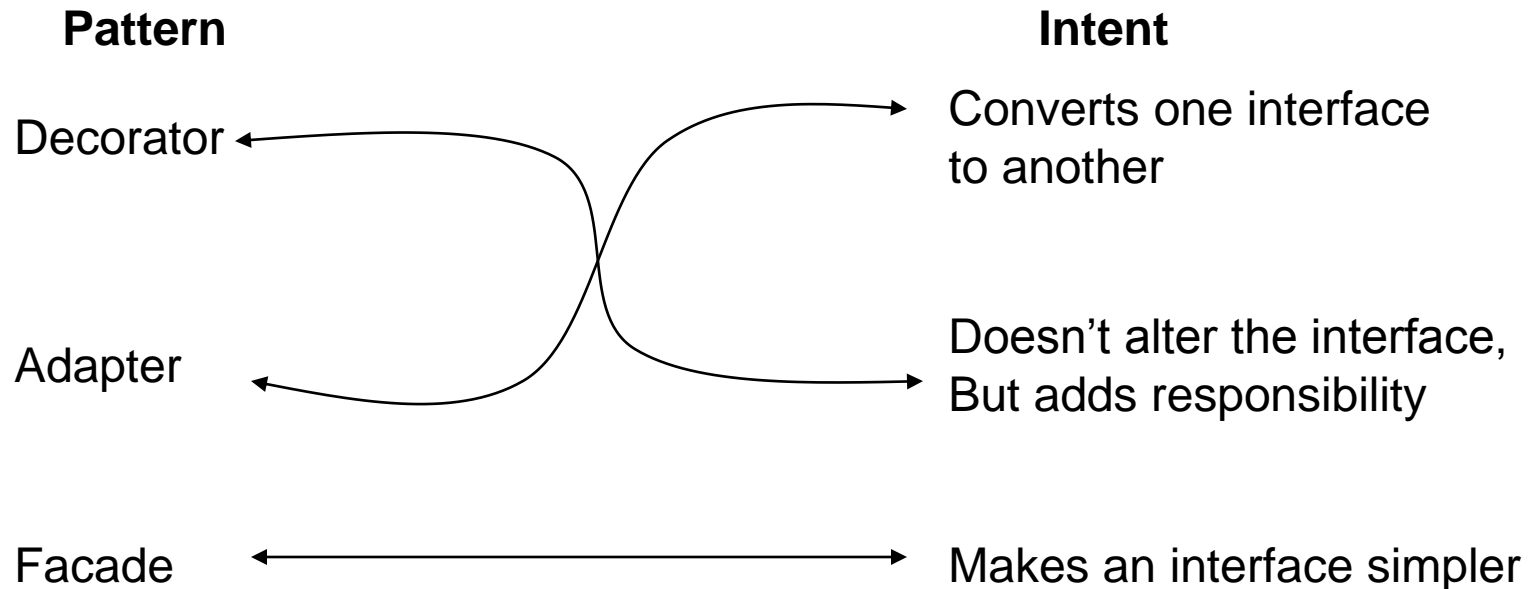


The acceptance of a bid by the auctioneer results in a broadcast of the new high bid to the bidders.

OO Patterns - Summary

- **Strategy Pattern** defines a family of algorithms, Encapsulates each one, and makes them interchangeable. Strategy lets the algorithm vary independently from clients that use it.
- **Observer Pattern** defines a one-to-many dependency between objects so that when one object changes state, all of its dependents are notified and updated automatically.
- **Decorator Pattern** – attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative for sub-classing for extending functionality
- **Factory Pattern** – Assign the responsibility of creating complex objects to a Factory Class. Let the Factory decide which class to instantiate.
- **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 Façade Pattern** provides a unified interface to a set of interfaces in a subsystem. Façade defines a higher level interface that makes the subsystem easier to use.

A little comparison



Proxy vs. Decorator

- The Proxy pattern is very similar in structure to the Decorator pattern. Both patterns **create a “wrapper” around another object.**
 - Adapter is a third kind of “wrapper”, but is clearly different from Decorator and Proxy because an **adapter has a different interface than the wrapped object.**
- Here are some differences between the two patterns:
 - The intents of the patterns are different:
 - **Decorator is used to add additional responsibilities to an object** (without using inheritance)
 - **Proxy is used to “control access” to an object.** Rather than adding responsibilities, a Proxy might actually prevent access to some functionality (e.g., read-only collections, Secure objects)
 - **A Decorator always stores a reference to the wrapped object;** a Proxy may or may not, depending on what it does
 - Distributed objects (proxy never stores direct object reference)
 - Lazy Loading (proxy sometimes stores a direct object reference)

Consequences

- Advantages

- The Proxy Pattern provides an **additional level of indirection** to support distributed, controlled, or intelligent access to the target object, protecting the target from undue complexity.
 - Separation of housekeeping code from functionality
- Decoupling clients from the location of remote server components
- Proxies are useful whenever there is a need for a more sophisticated reference to an object than can be provided by a simple pointer.

- Disadvantages

- Less efficiency due to indirection
- Complex implementation

Quiz - Match each pattern with its description

Pattern	Description
Decorator	Wraps another object and provides a different interface to it
Facade	Wraps another object and provides additional behavior for it
Proxy	Wraps another object to control access to it
Adapter	Wraps a bunch of objects to simplify their interface

Golden OO-principles

- encapsulate what varies
- 1class, 1 responsibility
- program against an interface, not against an implementation
- prefer composition over inheritance
- loose coupling, modular black boxes

Summary

- To become more effective in selecting patterns for a particular scenario you need to understand *whys* and *whats* of the patterns