# CS525 Advanced Software Development

## Lesson 7 – The Adapter & Facade Patterns

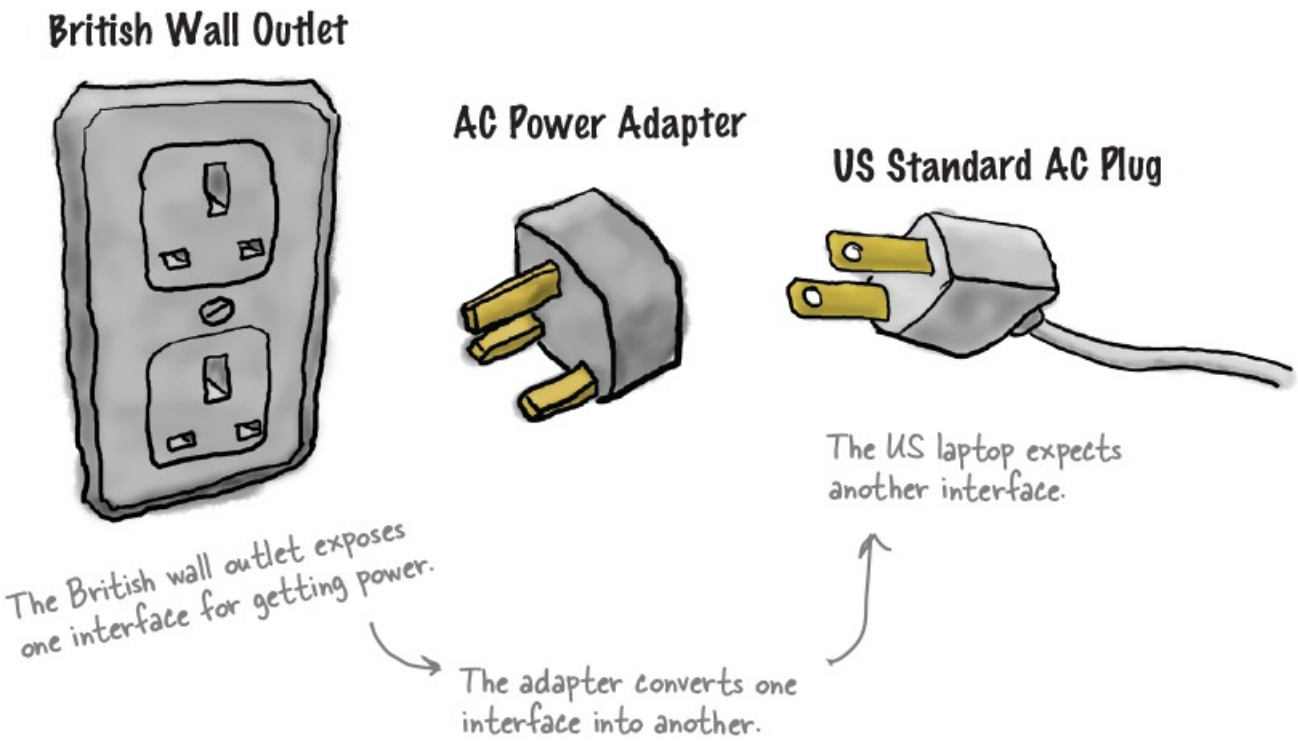
Design Patterns
*Elements of Reusable Object-Oriented Software*

Payman Salek, M.S.
March 2022

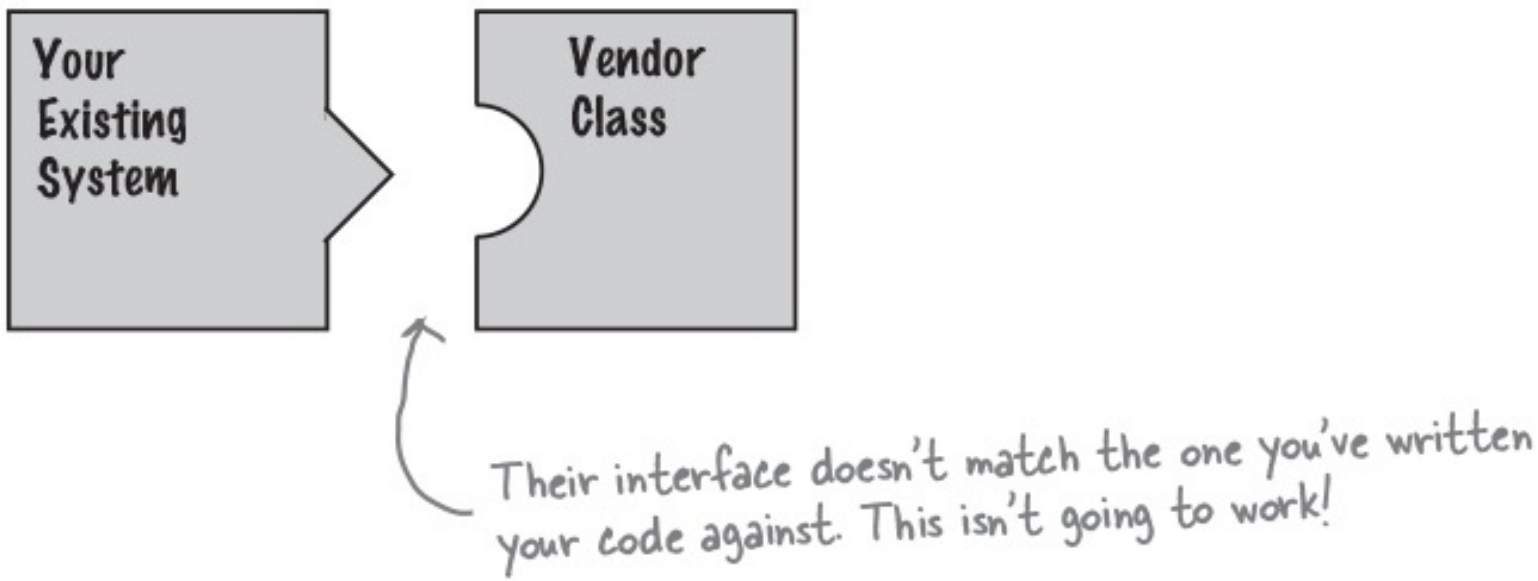
# Remember the Decorator Pattern?

We wrapped objects to give them new responsibilities. Now we're going to wrap some objects with a different purpose: to make their interfaces look like something they're not.

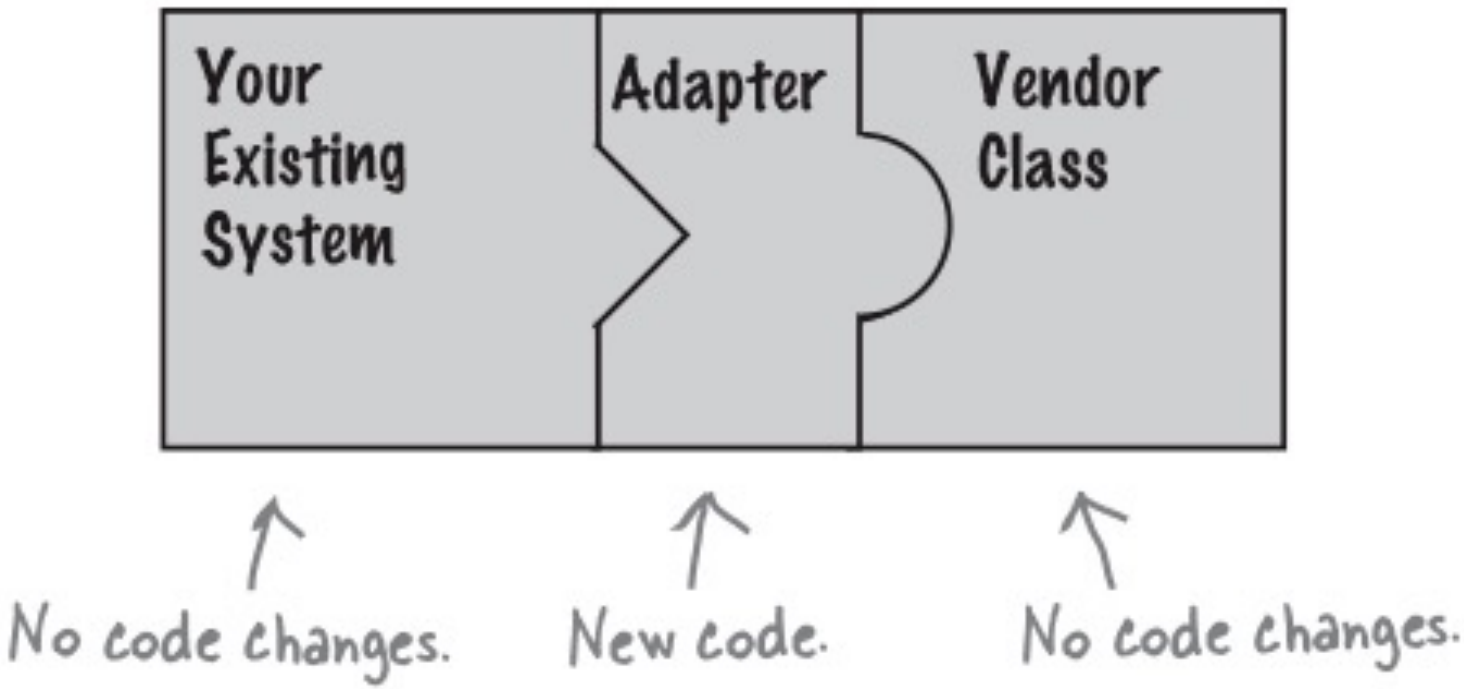
# Setting the stage (AC Plugs)



# Putting a square peg in a round hole



# Putting a square peg in a round hole



# Remember the Duck interface?

```
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.

# …and the implementation??

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

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

Simple implementations: MallardDuck just prints out what it is doing.

# Now consider the Turkey hierarchy

```
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 MallardDuck, it just prints out its actions.

# Now consider the Turkey hierarchy

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

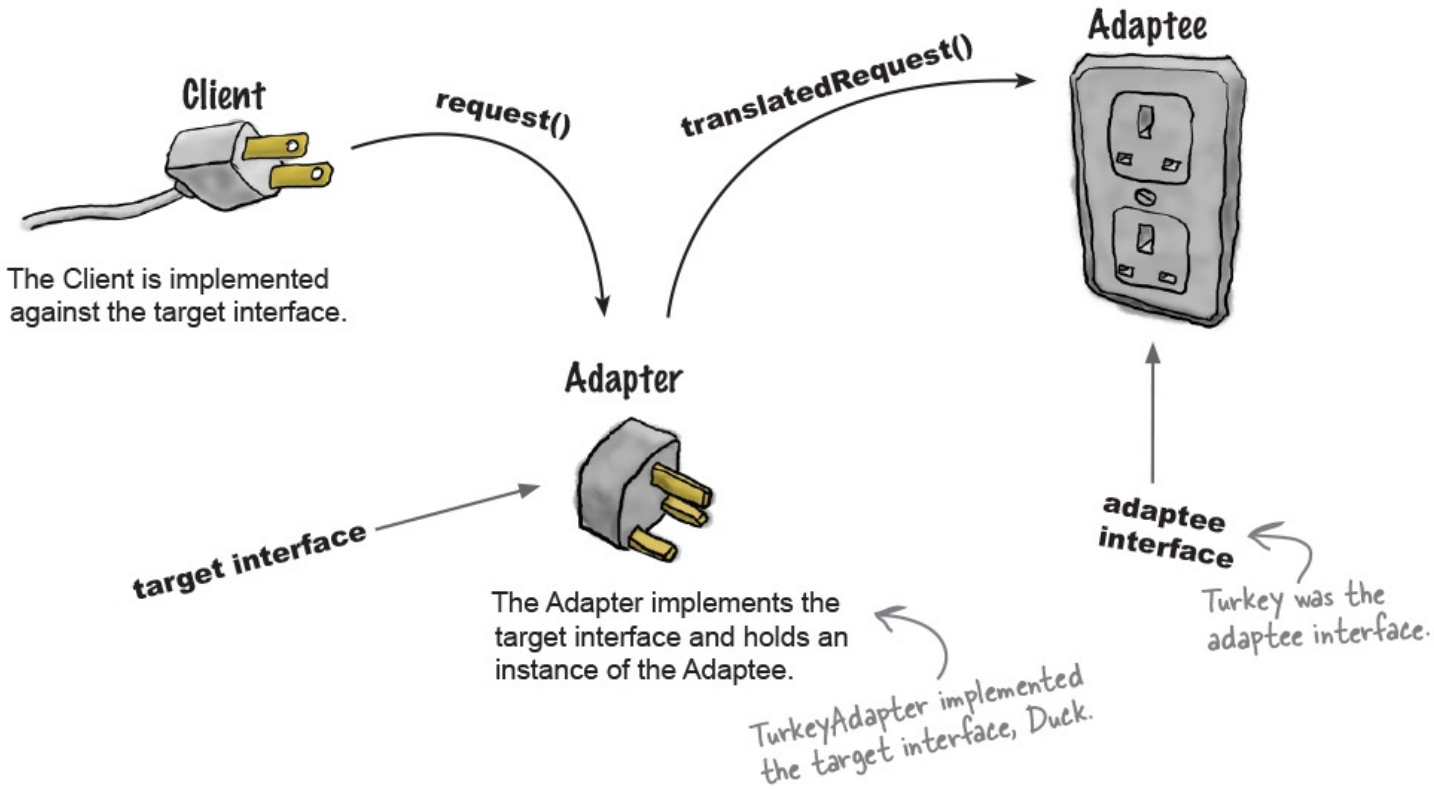
```
public class TurkeyAdapter implements Duck {
    Turkey turkey;

    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.

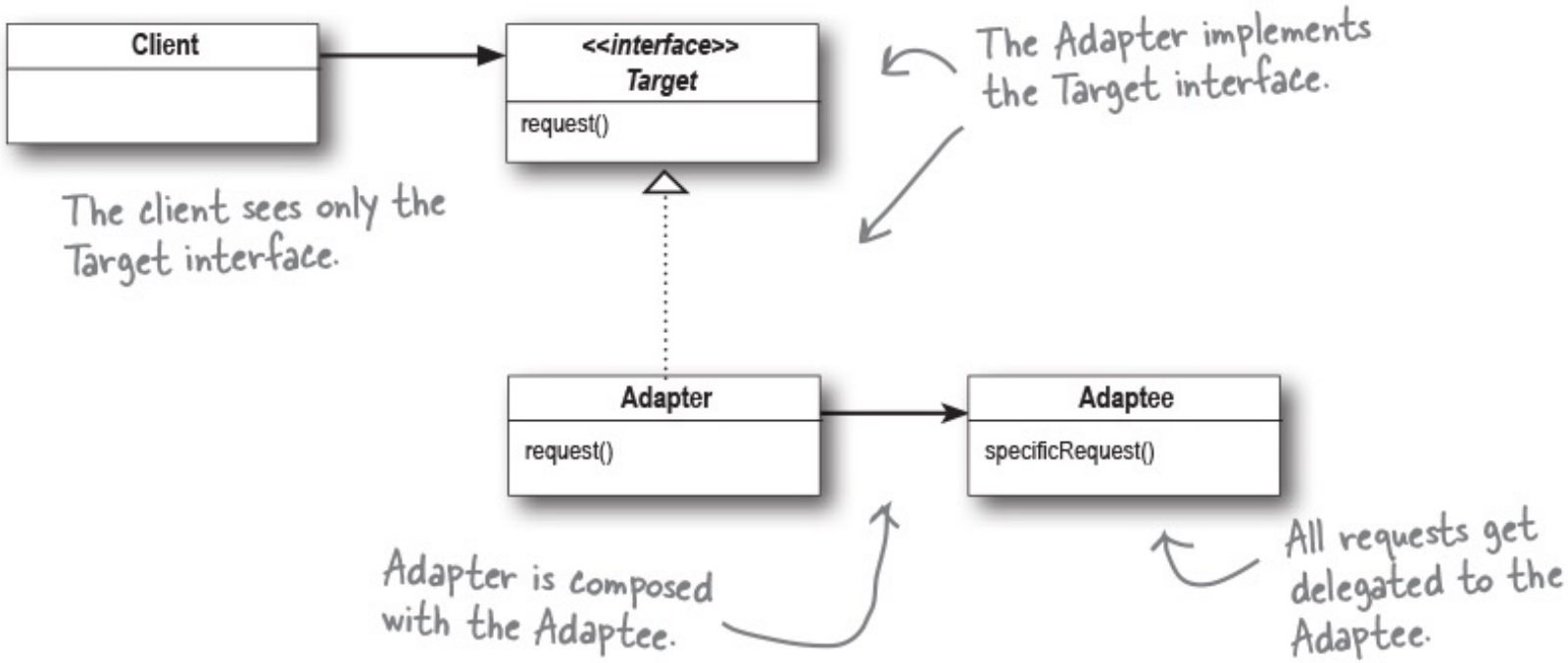
# The Adapter Pattern Explained



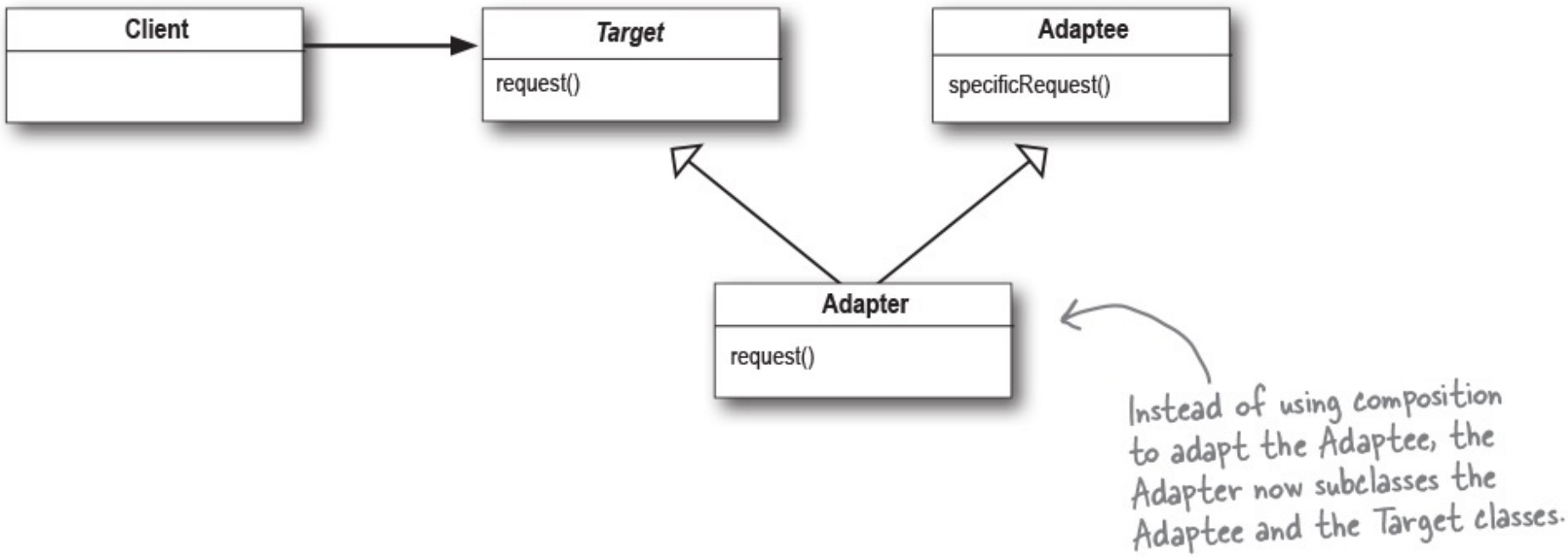
# The Adapter Pattern

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 (Object Adapter)

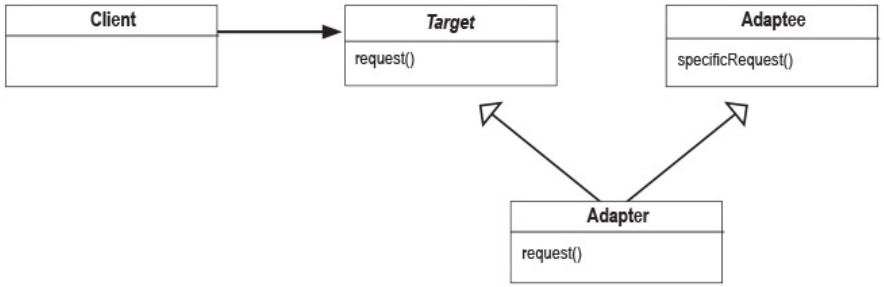


# The Adapter Pattern (Class Adapter)

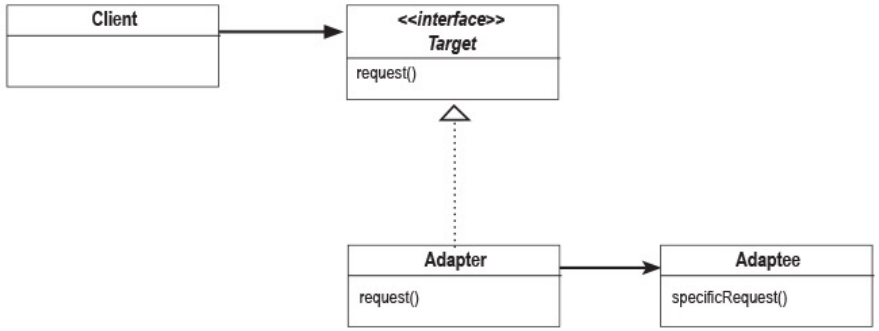


# The Adapter Pattern

**Class Adapter**



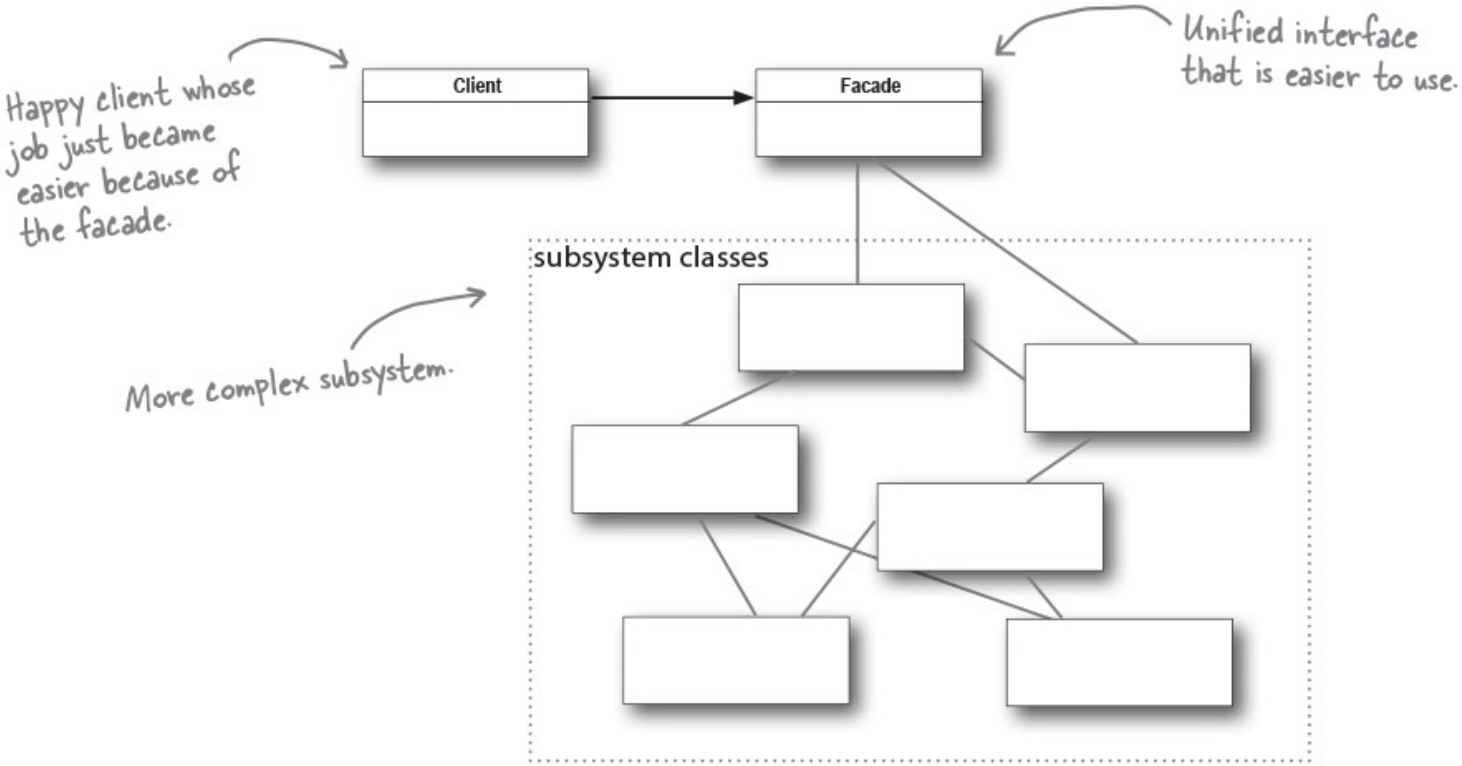

**Object Adapter**



# The Facade Pattern

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

# The Facade Pattern



# Principle of Least Knowledge

Talk only to your immediate friends!

# Principle of Least Knowledge

Without the Principle

```
public float getTemp() {
    Thermometer thermometer = station.getThermometer();
    return thermometer.getTemperature();
}
```

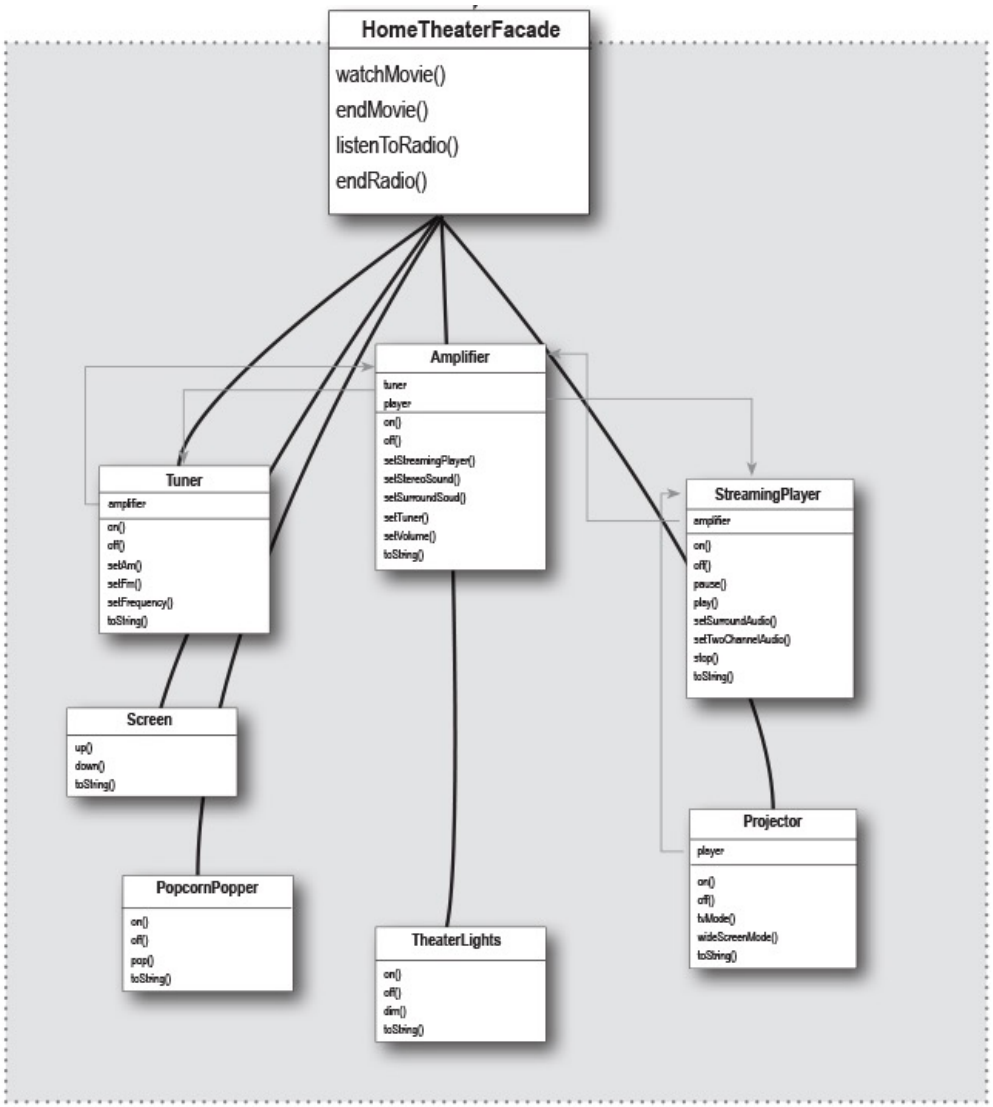
Here we get the thermometer object from the station and then call the getTemperature() method ourselves.

With the Principle

```
public float getTemp() {
    return station.getTemperature();
}
```

When we apply the principle, we add a method to the Station class that makes the request to the thermometer for us. This reduces the number of classes we're dependent on.

# The Façade Pattern



# The Façade Pattern