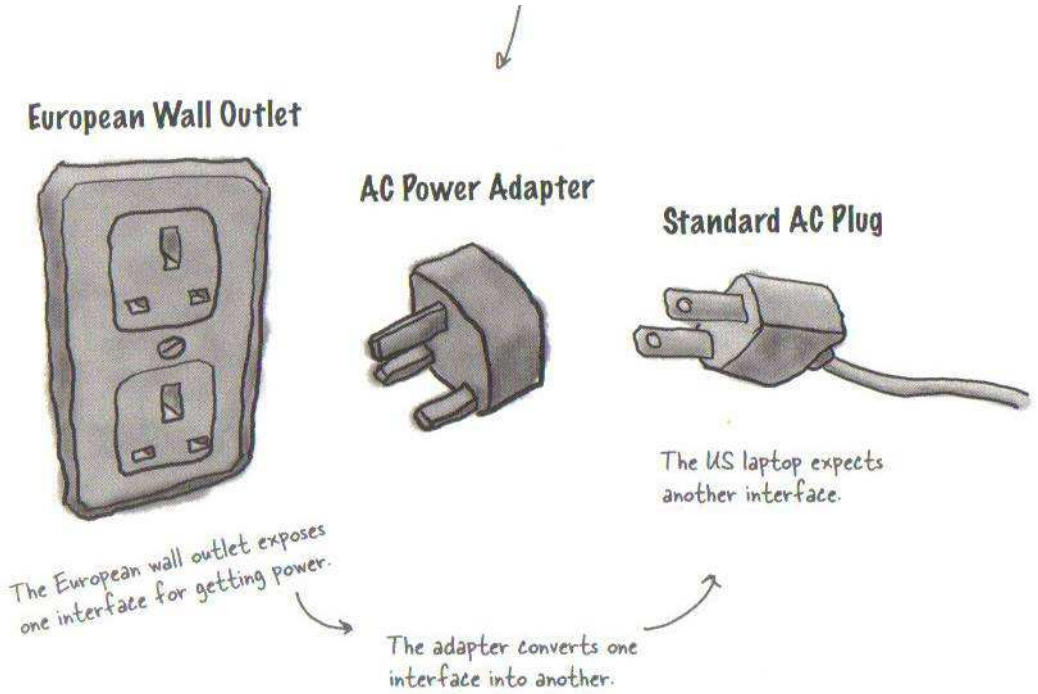
Adapter Pattern

Presented by,

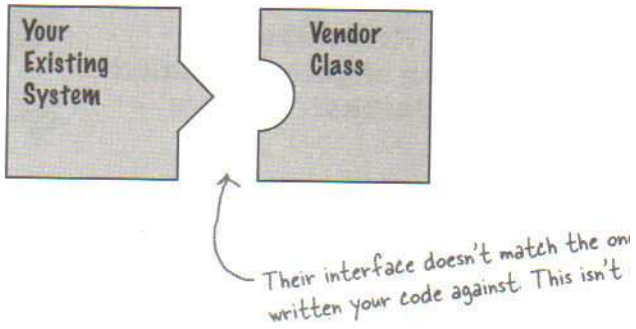
K.Vallidevi,

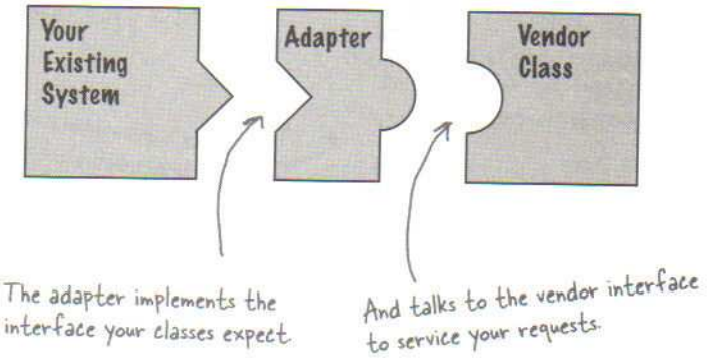
SSN College of Engineering

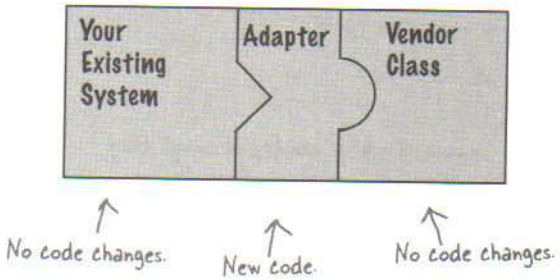
Adapters in real life



Object-Oriented Adapters







Turkey that wants to be a duck example

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

Subclass of a duck – Mallard Duck

```
public class MallardDuck implements Duck {
    public void quack() {
        System.out.println("Quack");
    }
    public void fly() {
        System.out.println("I'm flying");
    }
}
```

Turkey Interface

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

An instance of a turkey

```
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");
    }
}
```

Turkey adapter – that makes a turkey look like a duck

```
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();
        }
    }
}
```

Duck 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();
    }
}
```

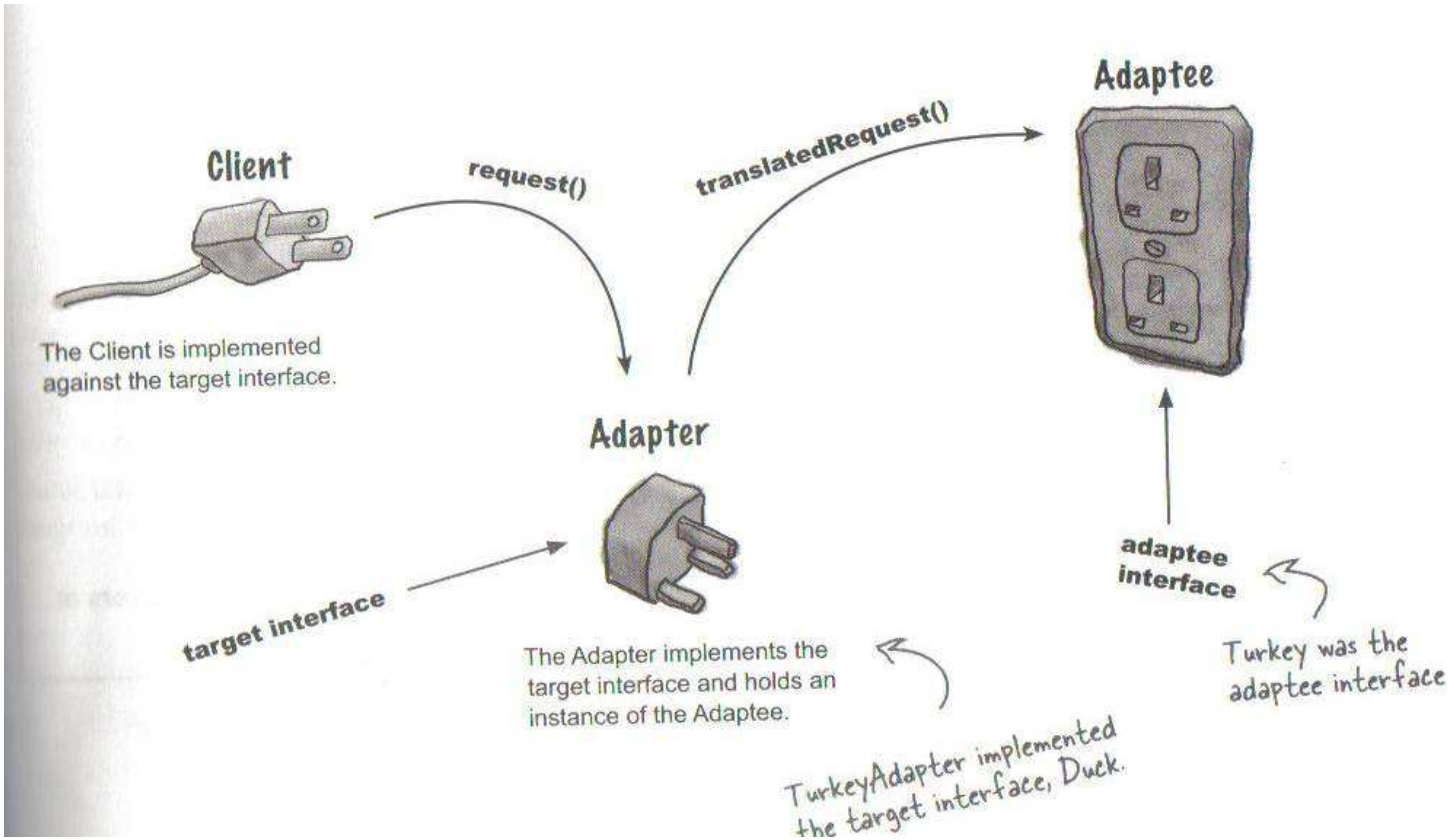
Test run – turkey that behaves like a duck

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

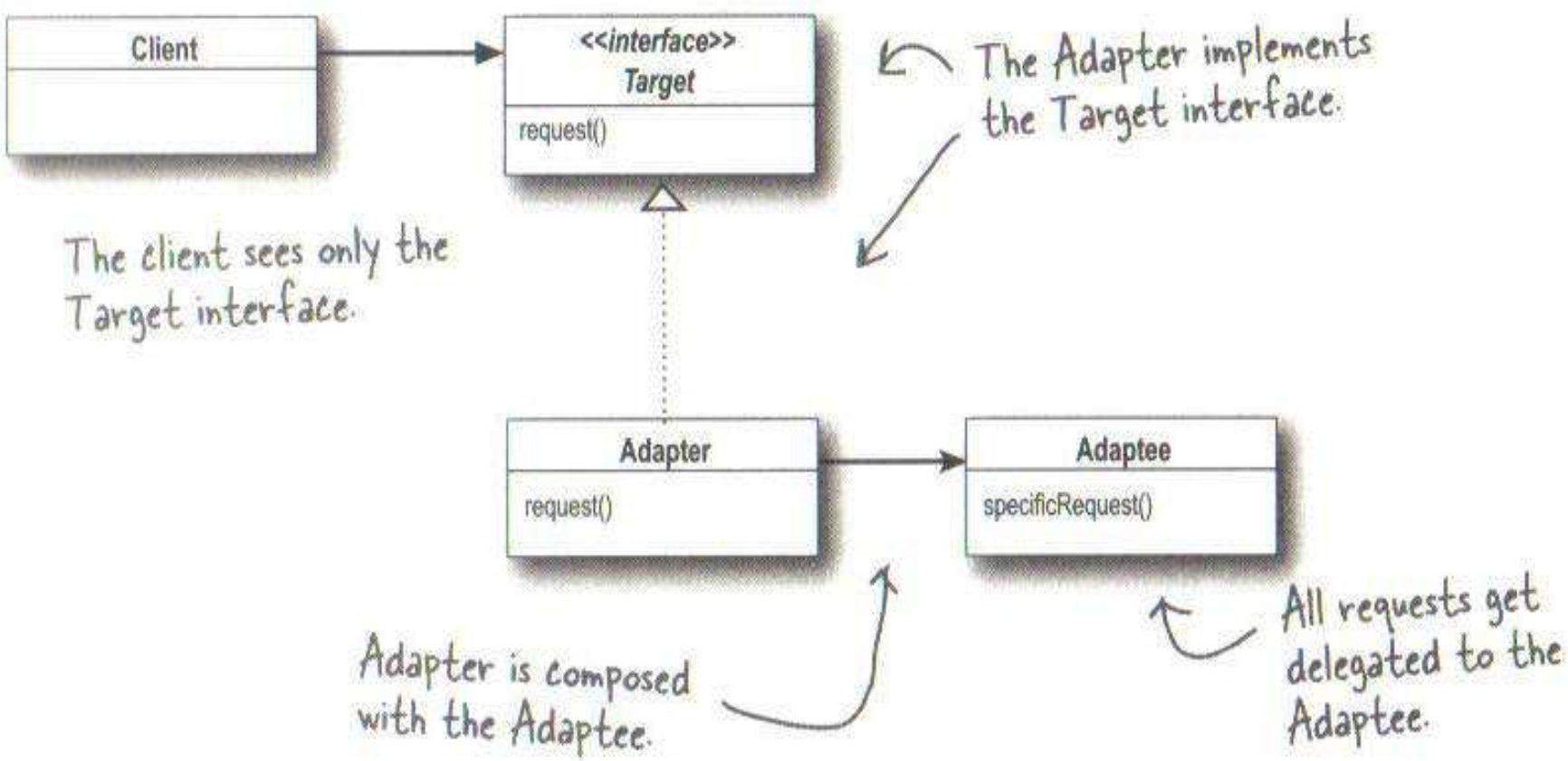
Adapter Pattern explained



Adapter Pattern defined

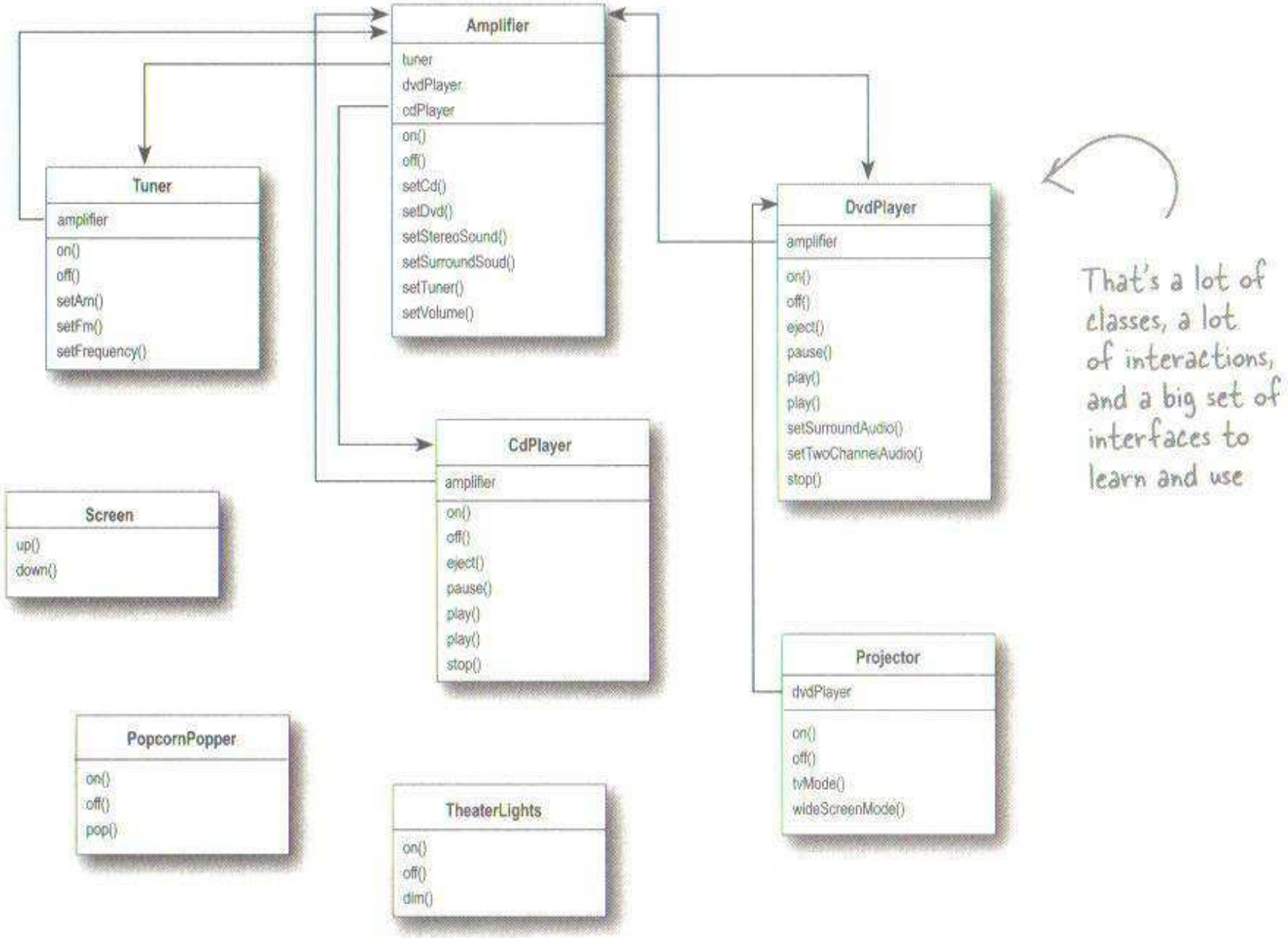
The Adapter Pattern converts the interface of a class into another interface the clients expect. Adapter lets classes work together that couldn’t otherwise because of incompatible interfaces.

Adapter Pattern



Façade Pattern

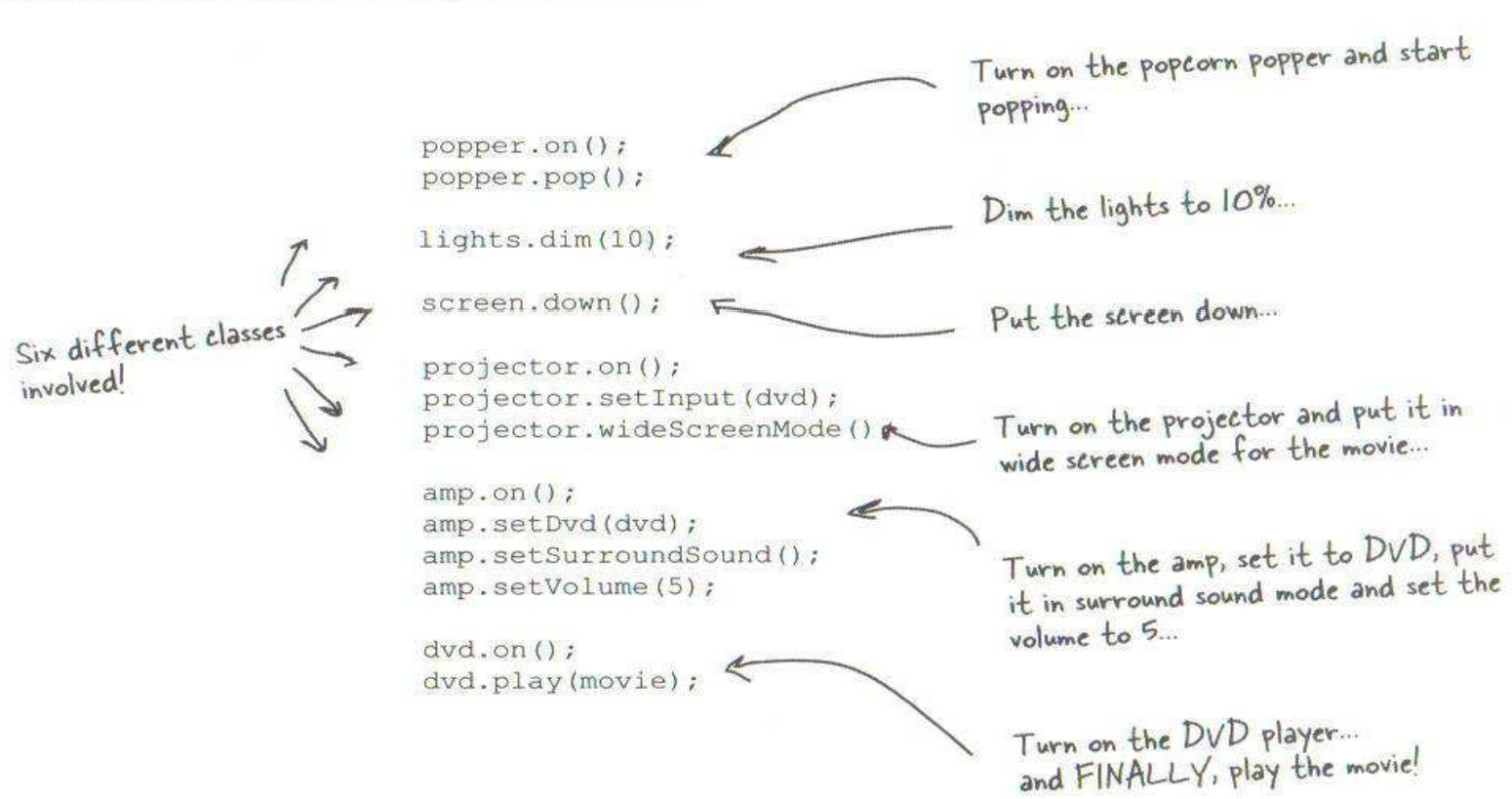
Simplifying complex subsystems

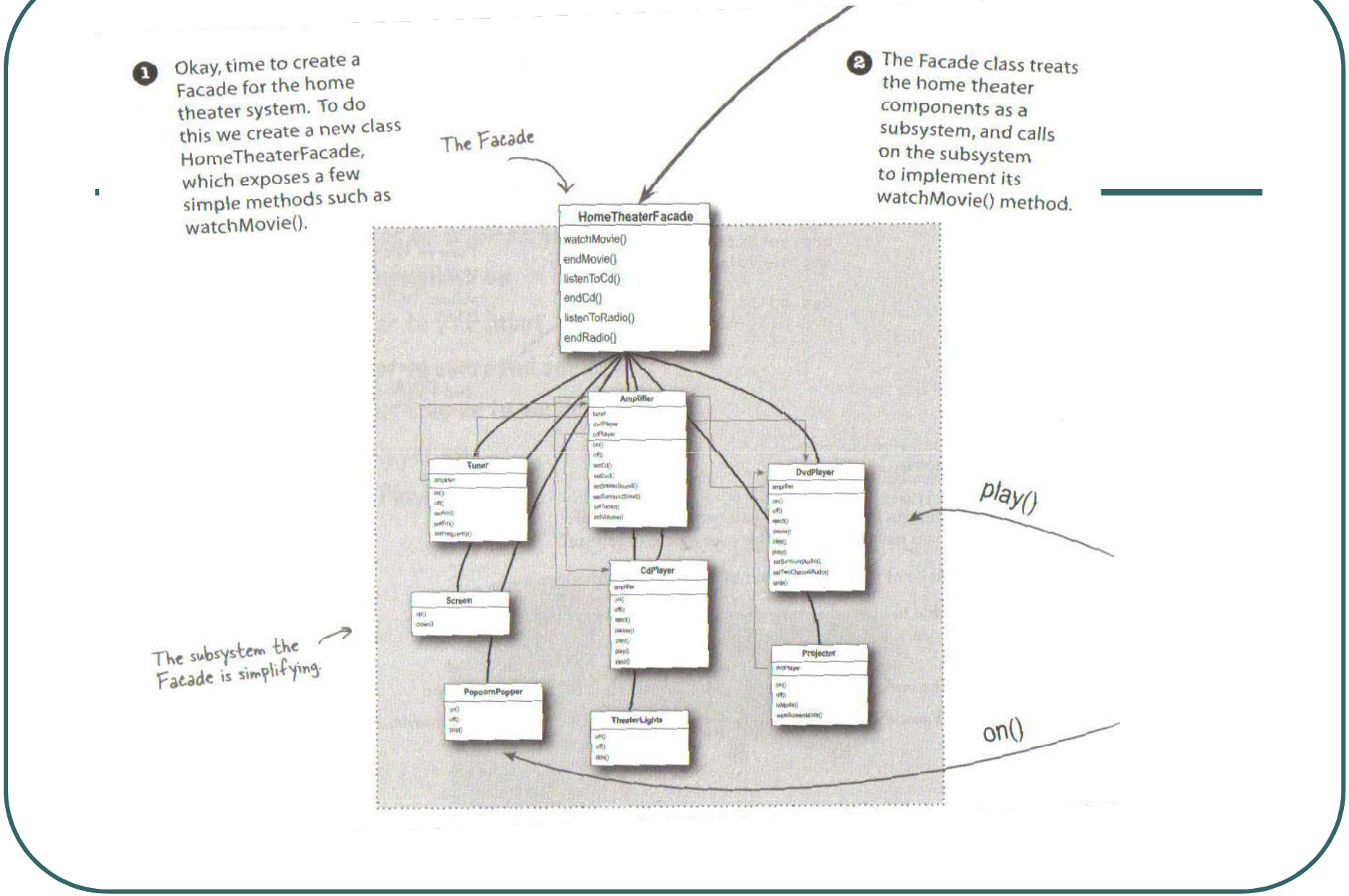

Amplifier
tuner
dvdPlayer
cdPlayer
on()
off()
setCd()
setDvd()
setStereoSound()
setSurroundSoud()
setTuner()
setVolume()
Tuner
amplifier
on()
off()
setAm()
setFm()
setFrequency()
DvdPlayer
amplifier
on()
off()
eject()
pause()
play()
play()
setSurroundAudio()
setTwoChannelAudio()
stop()
That's a lot of classes, a lot of interactions, and a big set of interfaces to learn and use
CdPlayer
amplifier
on()
off()
eject()
pause()
play()
play()
stop()
Screen
up()
down()
Projector
dvdPlayer
on()
off()
tvMode()
wideScreenMode()
PopcornPopper
on()
off()
pop()
TheaterLights
on()
off()
dim()

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

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().
The Facade
2 The Facade class treats the home theater components as a subsystem, and calls on the subsystem to implement its watchMovie() method.
HomeTheaterFacade
watchMovie()
endMovie()
listenToCd()
endCd()
listenToRadio()
endRadio()
Amplifier
Tuner
DvdPlayer
CdPlayer
Screen
Projector
PopcornPopper
TheaterLights
play()
on()
The subsystem the Facade is simplifying.

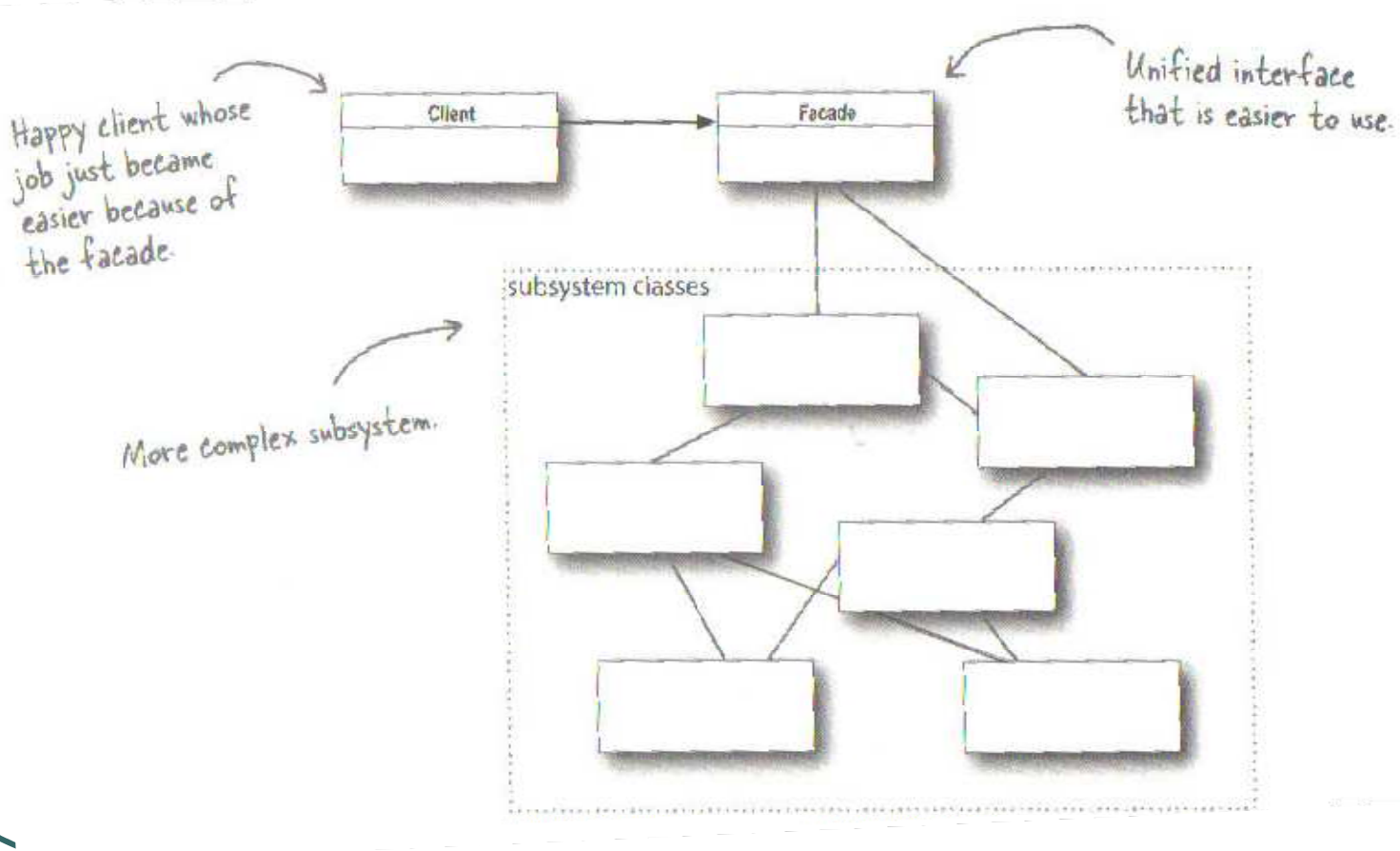
Façade example

- Look at Eclipse code for home theater façade

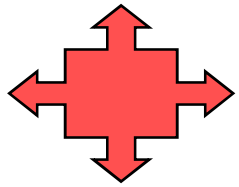
Façade Pattern defined

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



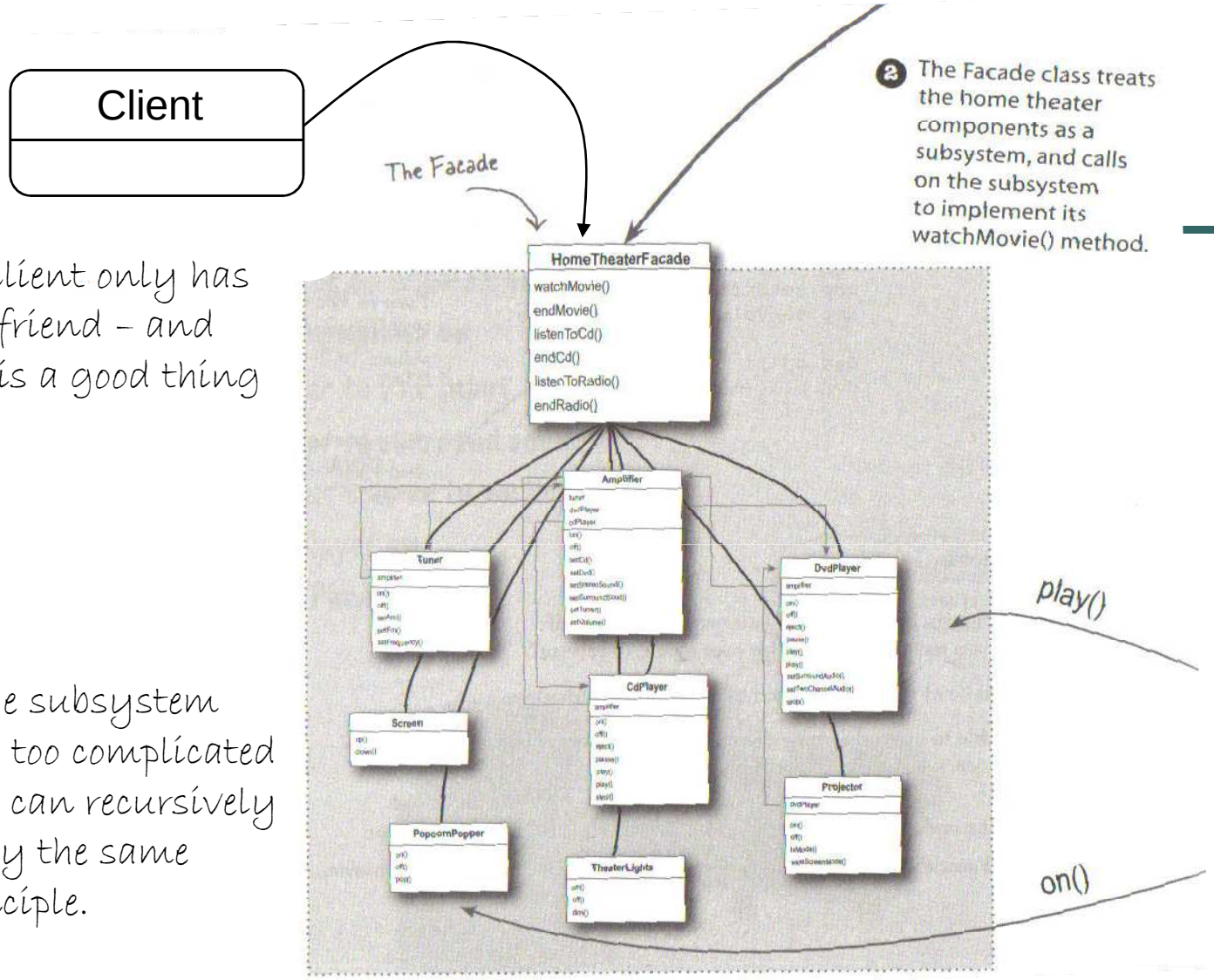
Design Principle



Principle of Least Knowledge

talk only to your immediate friends

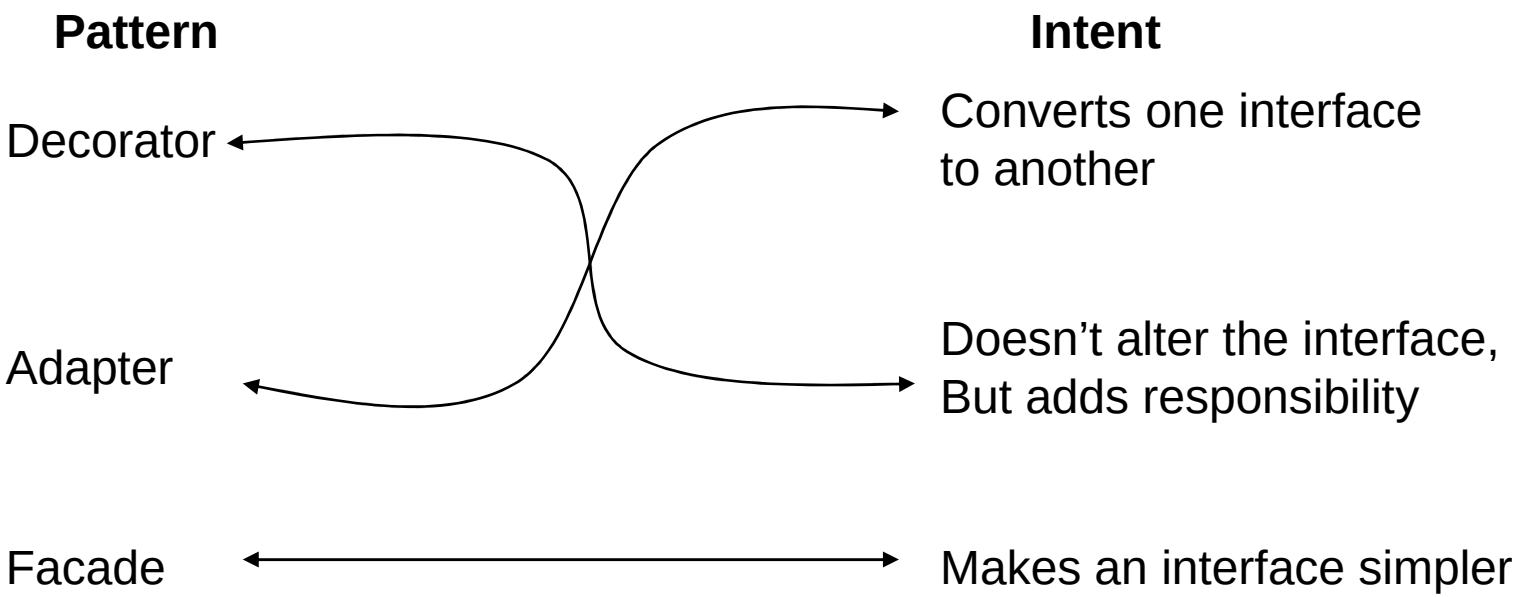
Basically this says minimize your dependencies



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.

A little comparison



Summary so far..

- OO Basics
  - Abstraction
  - Encapsulation
  - Inheritance
  - Polymorphism
- OO Principles
  - Encapsulate what varies
  - Favor composition over inheritance
  - Program to interfaces not to implementations
  - Strive for loosely coupled designs between objects that interact
  - Classes should be open for extension but closed for modification.
  - Depend on abstracts. Do not depend on concrete classes.
  - Only talk to your friends

Summary so far…

- OO Patterns
  - **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
  - **Abstractor Factory** – Provide an interface for creating families of related or dependent objects without specifying their concrete classes.
  - **Factory Method** – Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory method lets a class defer instantiation to the subclasses.
  - **Command Pattern** – Encapsulates a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.
  - **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.