

CS 130 SOFTWARE ENGINEERING

DESIGN PATTERNS

SINGLETON, COMMAND, ADAPTER, FAÇADE

Professor Miryung Kim
UCLA Computer Science
Based on Materials from Miryung Kim,
Christine Julien and Adnan Aziz

AGENDA

- ▶ Singleton
- ▶ Command
- ▶ Adaptor
- ▶ Façade
- ▶ State (It will be on HW)

RECAP

► Strategy

Encapsulate a family of algorithms and vary them

► Factory Method:

Separate creation of varying products from other shared uses

► Abstract Factory

Create varying families of products

Separate creation of varying families from other shared uses

Reduce duplication of parallel product classes



THE SINGLETON PATTERN

MOTIVATION FOR SINGLETON

- ▶ There are many objects we only need one of:
 - ▶ Thread pools, caches, dialog boxes, logging objects, device drivers, etc.
 - ▶ In many cases, instantiating more than one of such objects creates all kinds of problems (e.g., incorrect program behavior, resource overuse, inconsistent results)

THINK-PAIR-SHARE



- ▶ What's wrong with just using a global variable?

TOWARDS A SINGLETON

- ▶ In Java, how do you create a single object?
 - ▶ `new myObject();`
- ▶ And if you call that a second time?
 - ▶ You get a second, distinct object
- ▶ Can you always instantiate a class this way?
- ▶ How could you prevent such instantiation?
 - ▶

```
public class MyClass {  
    private MyClass() {}  
}
```
- ▶ Who can use such a private constructor?
 - ▶ Only code within `MyClass`

TOWARDS A SINGLETON (CONT.)

► How can you get access to code within **MyClass** if you can't instantiate it?

► What does this do:

```
► public class MyClass {  
    public static MyClass getInstance() {  
        ...  
    }  
}
```

► How would you call that?

```
► MyClass.getInstance();
```

► How would you fill out the implementation to make sure that only a single instance of **MyClass** is ever created?

NOW LET'S TAKE A LOOK AT SKELETON CODE

► [Handout-DesignPatternSkeletons](#)

THE CLASSIC SINGLETON



We have a static variable to hold our one instance of the class Singleton

```
public class Singleton {  
    private static Singleton uniqueInstance;  
    // ...  
    private Singleton() {}  
    public static Singleton getInstance() {  
        if (uniqueInstance == null) {  
            uniqueInstance = new Singleton();  
        }  
        return uniqueInstance;  
    }  
    // ...  
}
```

Our constructor is declared private; only Singleton can instantiate this class!

The getInstance() method gives us a way to instantiate the class and also to return an (the) instance of it.

THE SINGLETON PATTERN

The **Singleton Pattern** ensures a class has only one instance and provides a global point of access to that instance.

THE SINGLETON CLASS DIAGRAM

The getInstance() method is static, which means it's a class method, so you can conveniently access this method from anywhere in your code using Singleton.getInstance(). That's just as easy as accessing a global variable, but we get benefits like lazy instantiation from the Singleton.

```
Singleton
static uniqueInstance
// Other useful Singleton data...
static getInstance()
// Other useful Singleton methods...
```

The uniqueInstance class variable holds our one and only instance of Singleton.

↑ A class implementing the Singleton Pattern is more than a Singleton; it is a general purpose class with its own set of data and methods.

THREAD SAFETY

- ▶ The Singleton pattern, as we have implemented it, is not **thread safe**
- ▶ Let's explore what happens when multiple threads access the singleton pattern simultaneously.
- ▶ Draw a sequence diagram with three actors
 - ▶ The Singleton object
 - ▶ Two “client” threads

THREAD SAFE SINGLETON



```
public class Singleton {  
    private static Singleton uniqueInstance;  
    // ...  
    private Singleton() {}  
    public static synchronized Singleton getInstance() {  
        if (uniqueInstance == null) {  
            uniqueInstance = new Singleton();  
        }  
        return uniqueInstance;  
    }  
    // ...  
}
```

By adding the **synchronized keyword**, we force every thread to wait its turn before it can enter the method. That is, no two threads may be in the method at the same time

This turns out to not be a great idea. Why?

Synchronization is expensive. But that's not all...

When is synchronization *really* necessary?

WHOOPS. WHAT ARE THE OPTIONS, THEN?

► Just roll with synchronized getInstance () anyway. Maybe the performance of getInstance () isn't a big deal for you.



► Use **eager instantiation** instead of **lazy instantiation**

```
► public class Singleton {  
    private static Singleton uniqueInstance = new  
    Singleton();  
    private Singleton() {}  
    public static Singleton getInstance() {  
        return uniqueInstance;  
    }  
}
```

ONE MORE OPTION...



► Double checked locking (Java 5 and later)

```
► public class Singleton {  
    private volatile static Singleton uniqueInstance;  
    private Singleton () {}  
    public static Singleton getInstance() {  
        if (uniqueInstance == null) {  
            synchronized (Singleton.class) {  
                if (uniqueInstance == null) {  
                    uniqueInstance = new Singleton();  
                }  
            }  
        }  
        return uniqueInstance;  
    }  
}
```

Check for an instance; if there isn't one,
enter the synchronized block

Once in the block, check again and if still
null, create an instance

The volatile keyword ensures that
multiple threads handle the
uniqueInstance correctly when it is
being initialized to the Singleton
instance

Notice that we only synchronize the first
time through!

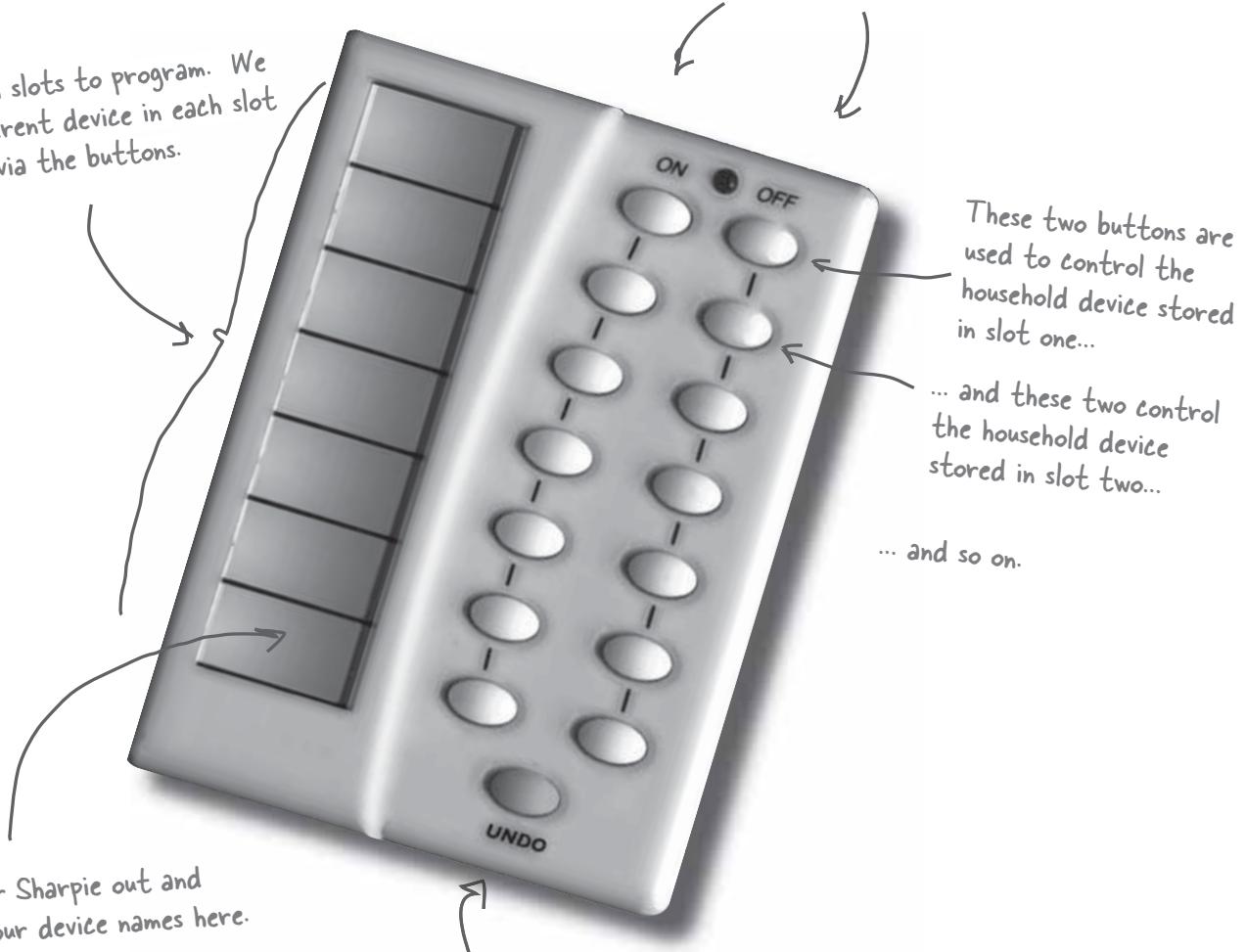
This approach can drastically reduce the
overhead of synchronization



THE COMMAND PATTERN

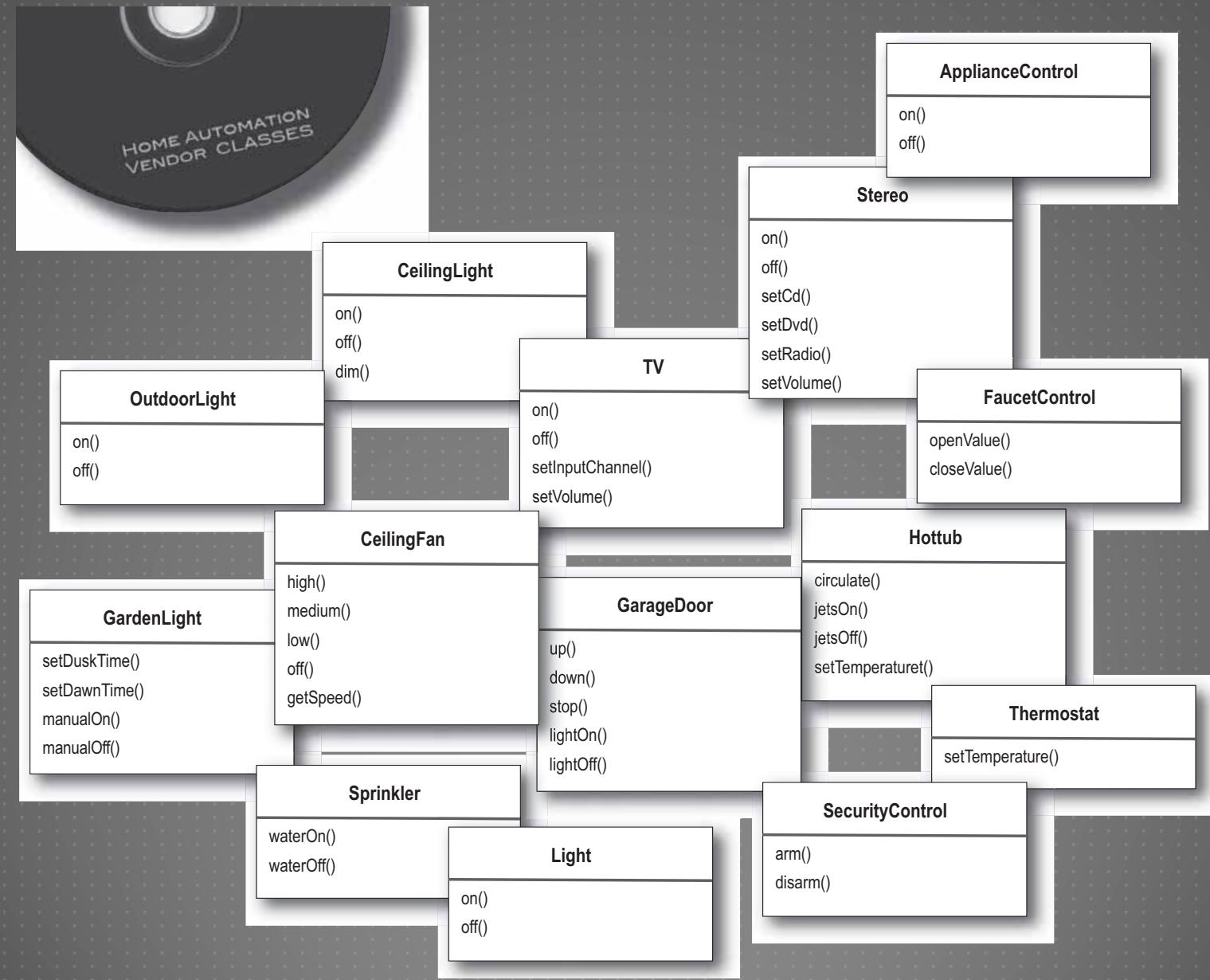
THE MISSION:A REMOTE CONTROL

We've got seven slots to program. We can put a different device in each slot and control it via the buttons.



Here's the global "undo" button that undoes the last button pressed.

THE VENDOR CLASSES



TOWARDS A DESIGN

We need some
information hiding
and *separation of concerns*

Yeah, the remote
shouldn't have a bunch
of switch statements
that select between
devices...

Yeah! What's with
all the different
method names?

The remote is
simple, but the
devices are not!

We really need to
decouple the requester
of the action from the
object that performs
the action



MOTIVATION FOR COMMAND

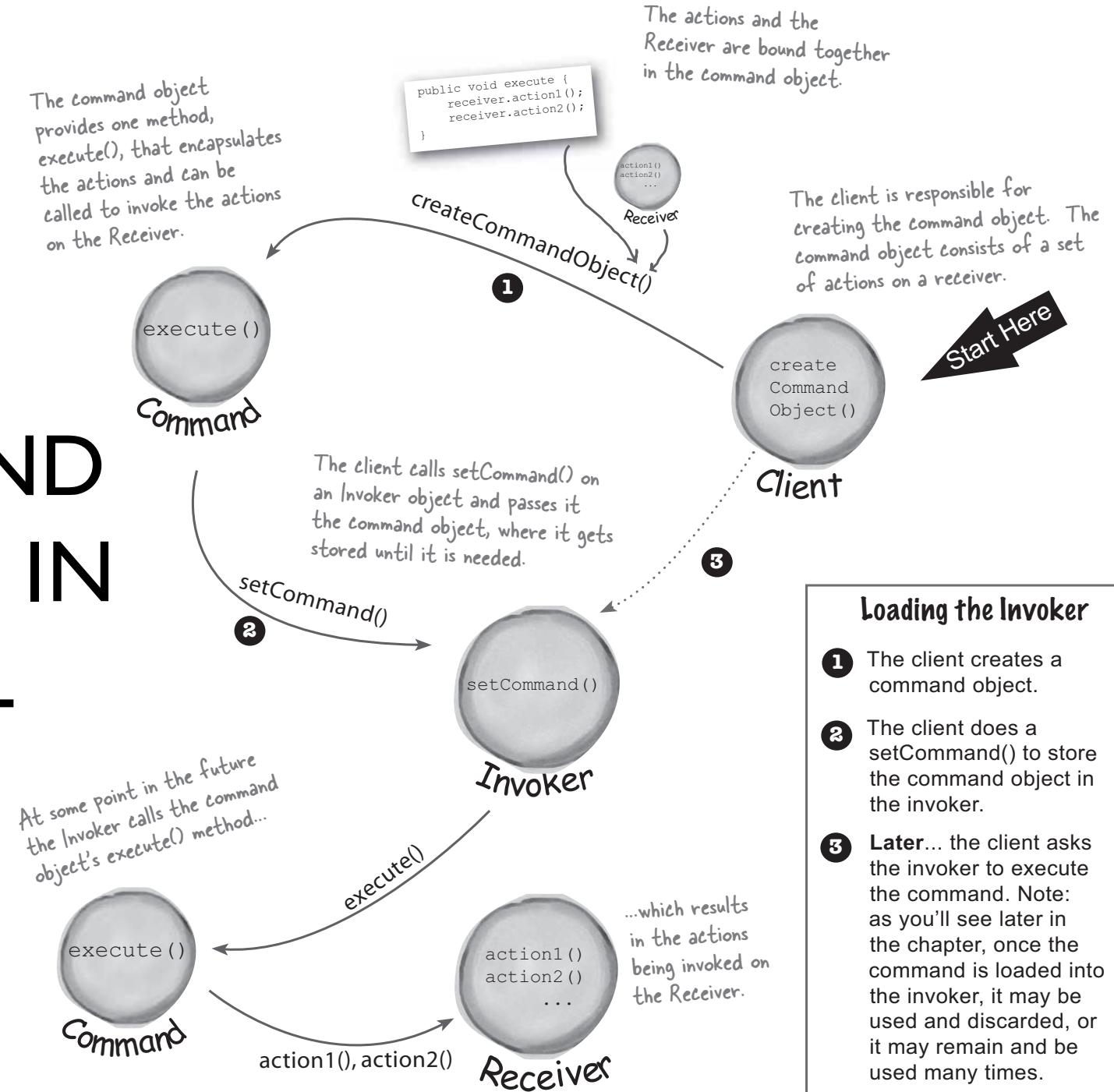
- ▶ There are multiple varying objects to be controlled
- ▶ We want different actions on different objects
- ▶ We need a common interface to request commands to be invoked on varying objects.

COMMAND OBJECTS (IN CONTEXT)



- ▶ We introduce **command objects** into the design
 - ▶ A command object encapsulates a request to do something (e.g., turn on a light) on a specific object (e.g., the living room lamp)
 - ▶ We can then just store a command object for each button such that when the button is pressed, the command is invoked
 - ▶ The button doesn't have to know *anything* about the command

THE COMMAND PATTERN IN GENERAL



Loading the Invoker

- 1 The client creates a command object.
 - 2 The client does a setCommand() to store the command object in the invoker.
 - 3 Later... the client asks the invoker to execute the command. Note: as you'll see later in the chapter, once the command is loaded into the invoker, it may be used and discarded, or it may remain and be used many times.

ROLES

- ▶ Client
- ▶ Command
 - ▶ execute()
 - ▶ “public void execute() { receiver.action();}”
- ▶ Invoker
 - ▶ setCommand()
- ▶ Receiver
 - ▶ action()

A FIRST COMMAND OBJECT

```
public interface Command {  
    public void execute();  
}
```

```
public class LightOnCommand implements Command {  
    Light light;  
    public LightOnCommand (Light light) {  
        this.light = light;  
    }  
    public void execute () {  
        light.on();  
    }  
}
```

Simple. Just one method called execute()

Light

on()
off()

The constructor is passed the specific light that this command controls. When execute() gets called, this is the light that will be the receiver of the request

USING THE COMMAND OBJECT

```
public class SimpleRemoteControl {  
    Command slot;  
  
    public SimpleRemoteControl () {  
  
        public void setCommand(Command command) {  
            slot = command;  
        }  
  
        public void buttonWasPressed() {  
            slot.execute();  
        }  
    }  
}
```

We have one slot to hold a command,
which will control one device

We have a method to set the command
the slot will control; could be called
multiple times to change the behavior of
the button

This method is called when the button is
pressed. All we do is take the current
command bound to the slot and call its
execute () method

A SIMPLE TEST OF THE REMOTE

```
public class RemoteControlTest {  
    public static void main(String[] args) {  
        SimpleRemoteControl remote =  
            new SimpleRemoteControl();  
  
        Light light = new Light();  
  
        LightOnCommand lightOn = new LightOnCommand(light);  
  
        remote.setCommand(lightOn);  
        remote.buttonWasPressed();  
    }  
}
```

The RemoteControlTest class is the Client

The remote is the Invoker; it will be passed a command object that can be used to make requests

The Light object is the Receiver of the request

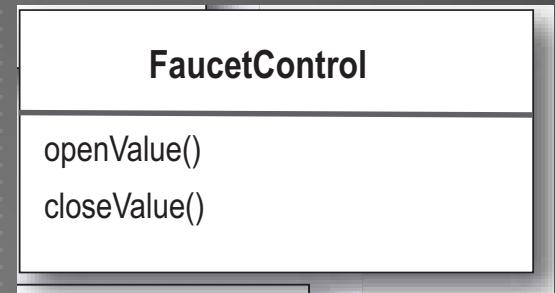
We create a command and pass it the Receiver

Then we pass the command to the Invoker.

IMPLEMENTING A DIFFERENT COMMAND ON A NEW OBJECT

- ▶ Implement the `FaucetOffCommand` class
- ▶ Here's the new “test” code:

```
public class RemoteControlTest {  
    public static void main(String[] args) {  
        SimpleRemoteControl remote = new SimpleRemoteControl();  
        Faucet faucet = new FaucetControl();  
        FaucetOffCommand faucetOff = new FaucetOffCommand(faucet);  
        remote.setCommand(faucetOff);  
        remote.buttonWasPressed();  
    }  
}
```



SOLUTION

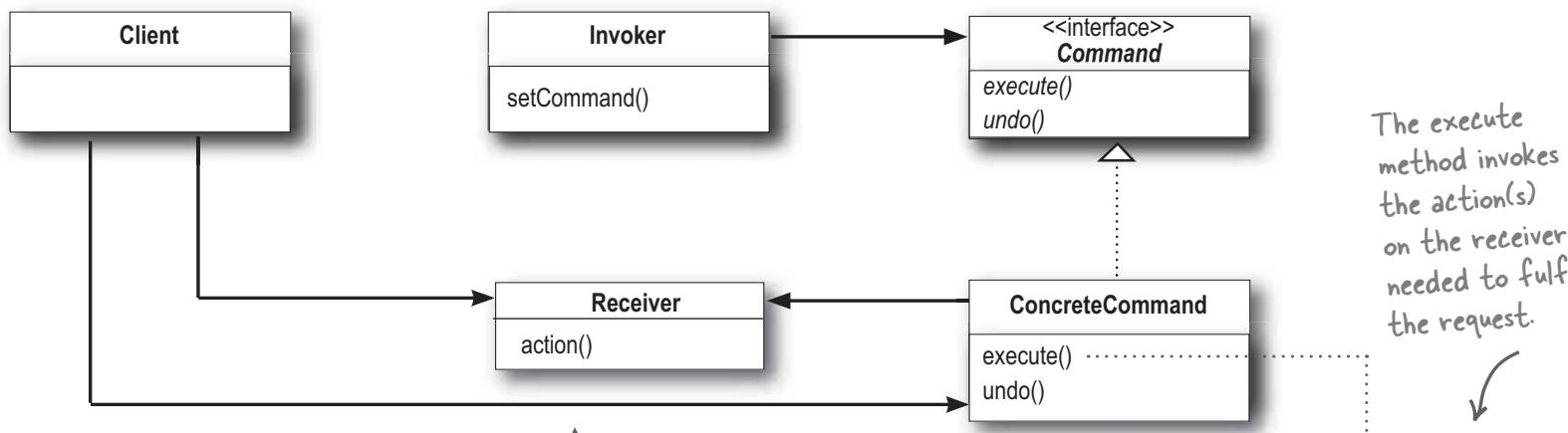
```
public class FaucetOffCommand implements Command {  
    FaucetControl faucet;  
  
    public FaucetOffCommand(Faucet faucet) {  
        this.faucet = faucet;  
    }  
  
    public void execute () {  
        faucet.closeValve();  
    }  
}
```

THE COMMAND PATTERN CLASS DIAGRAM

The Client is responsible for creating a ConcreteCommand and setting its Receiver.

The Invoker holds a command and at some point asks the command to carry out a request by calling its execute() method.

Command declares an interface for all commands. As you already know, a command is invoked through its execute() method, which asks a receiver to perform an action. You'll also notice this interface has an undo() method, which we'll cover a bit later in the chapter.



The Receiver knows how to perform the work needed to carry out the request. Any class can act as a Receiver.

The ConcreteCommand defines a binding between an action and a Receiver. The Invoker makes a request by calling execute() and the ConcreteCommand carries it out by calling one or more actions on the Receiver.

```
public void execute() {
    receiver.action()
}
```

The execute method invokes the action(s) on the receiver needed to fulfill the request.

THE COMMAND PATTERN

The Command Pattern encapsulates a request as an object, thereby letting you parameterize other objects with different requests, queue or log requests, and support undoable operations.

NOW LET'S TAKE A LOOK AT SKELETON CODE

► [Handout-DesignPatternSkeletons](#)

QUESTION 1.



- ▶ How does the remote know the difference between the kitchen light and the living room light?

QUESTION 2.

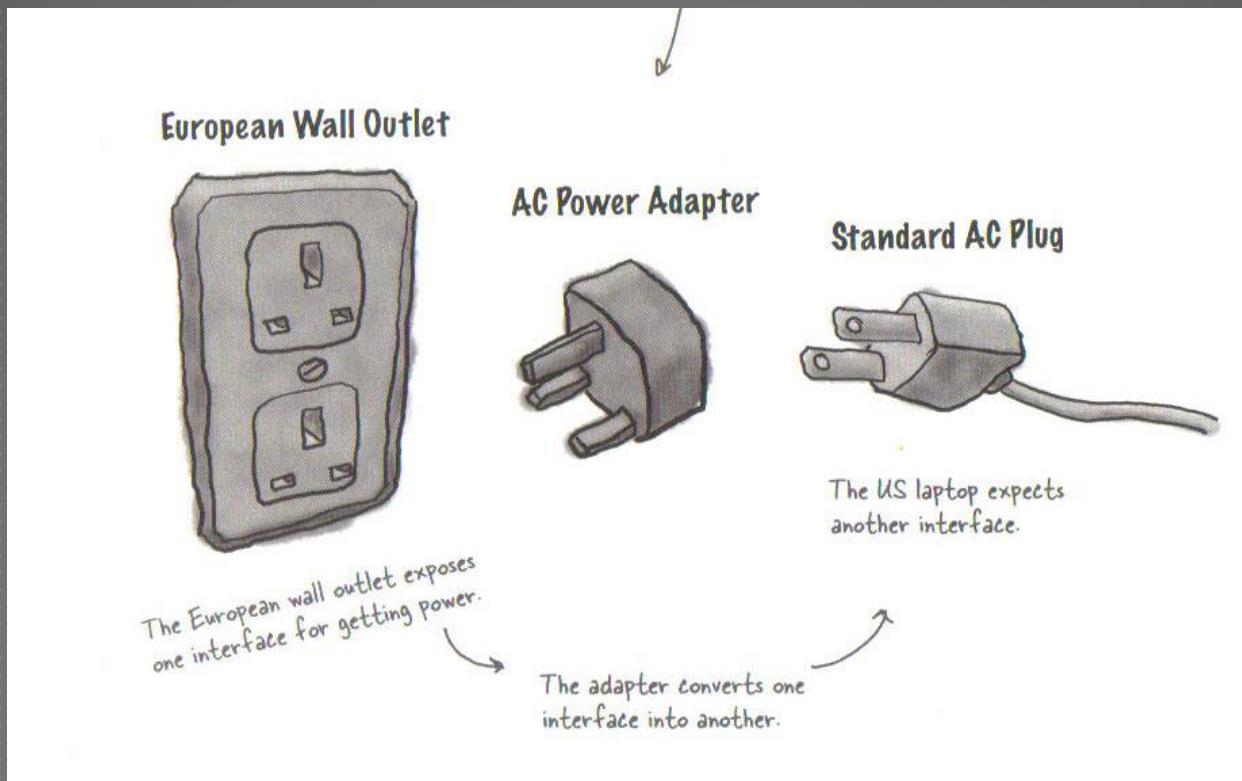


- ▶ How would you implement a history of undo operations?

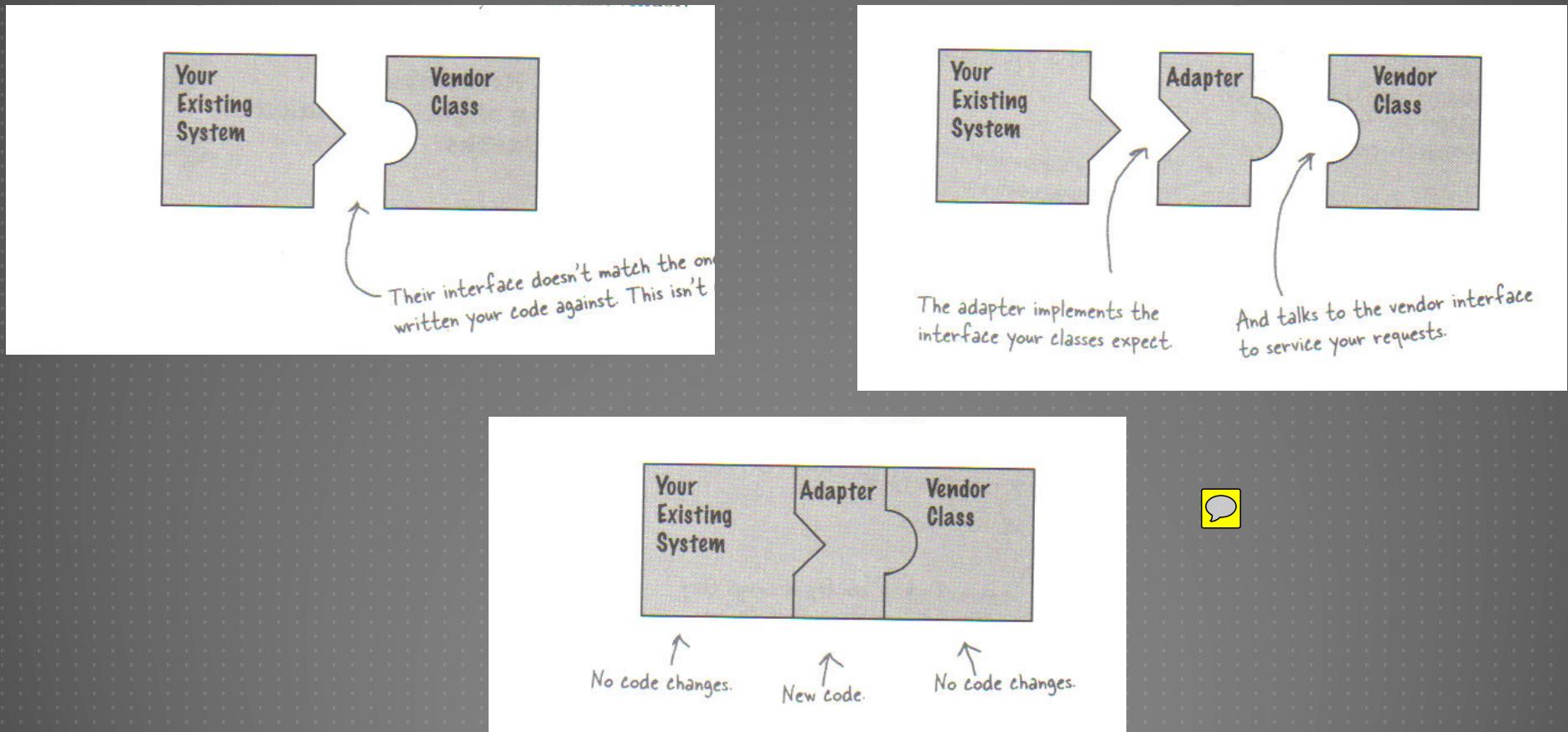


ADAPTER

ADAPTERS IN REAL LIFE



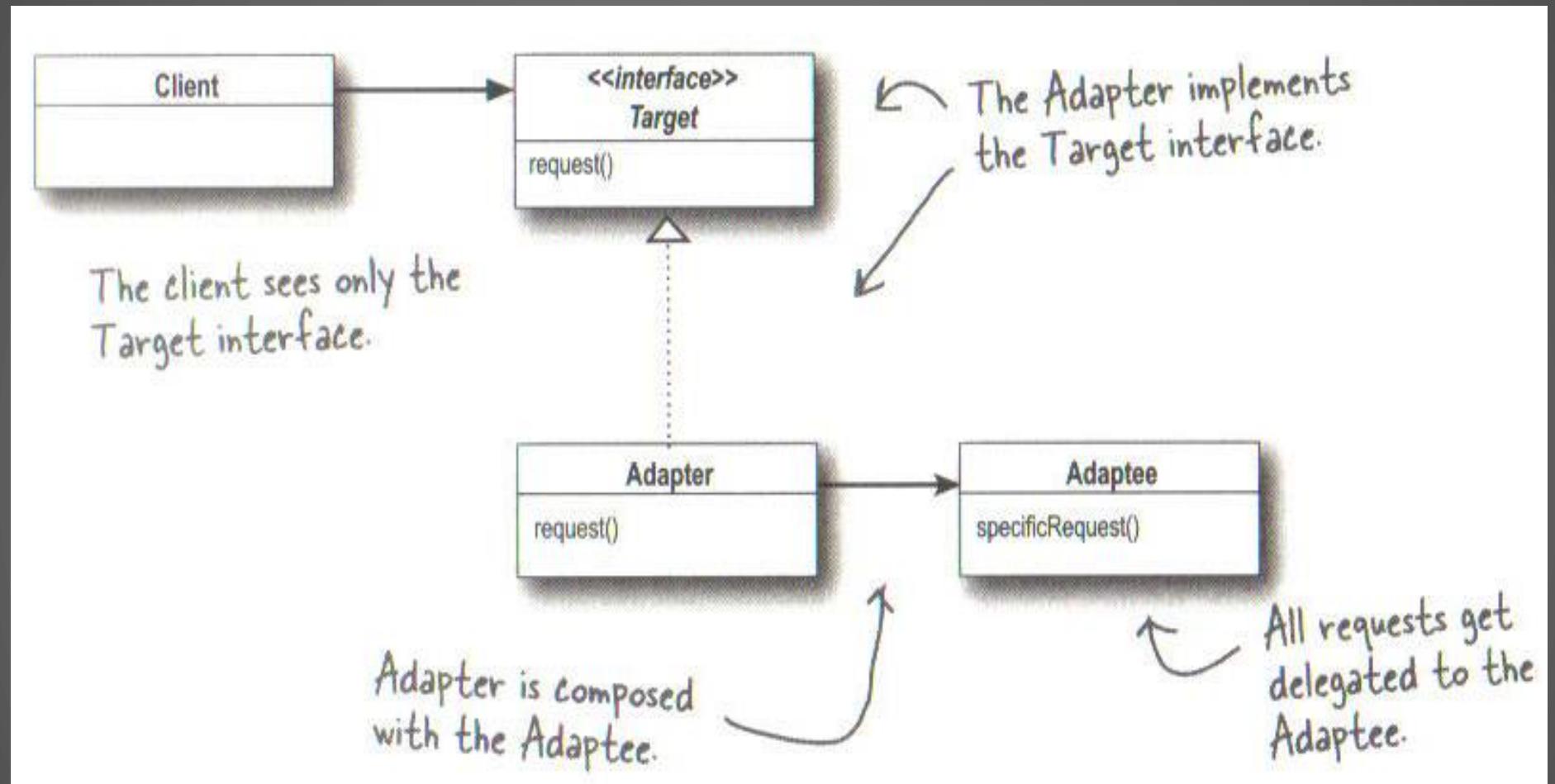
OBJECT-ORIENTED ADAPTERS



MOTIVATION FOR ADAPTOR

- ▶ You have legacy code that does not fit the target interface you desire.

ADAPTER PATTERN



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();  
        }  
    }  
}
```

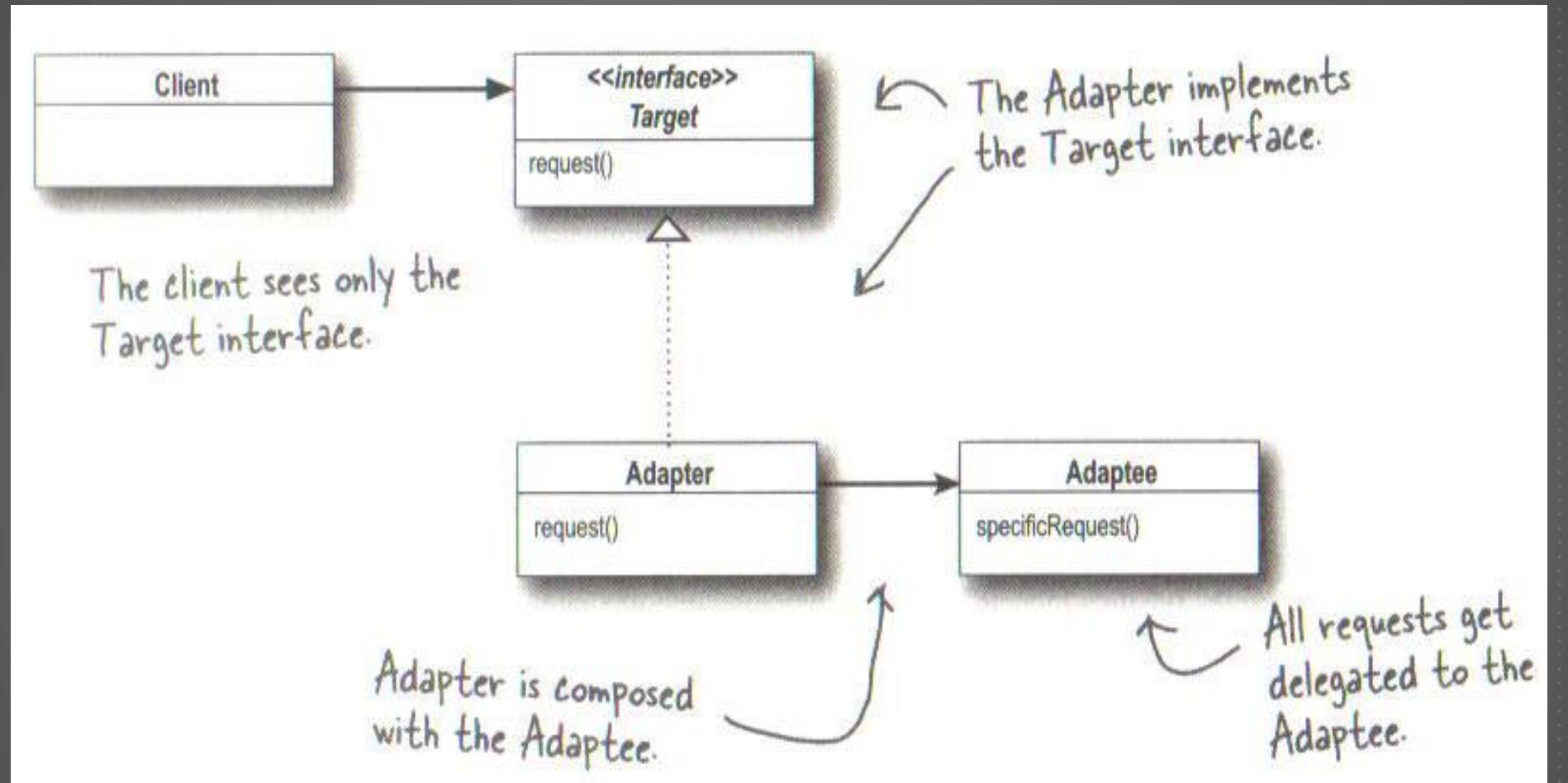
NOW LET'S TAKE A LOOK AT SKELETON CODE

► [Handout-DesignPatternSkeletons](#)

ADAPTER PATTERN DEFINED

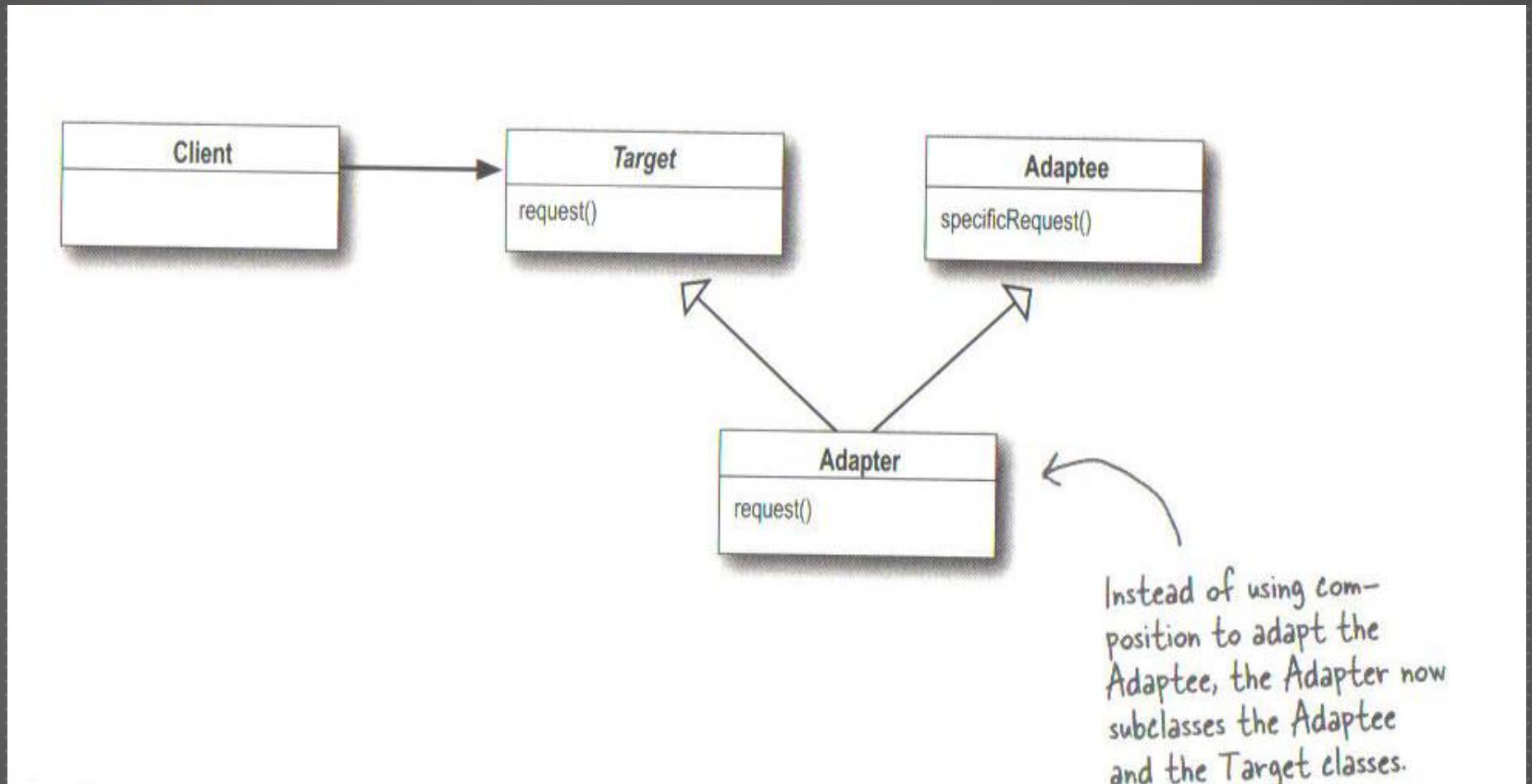
The Adapter Pattern converts the interface of a class into another interface the clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

THINK PAIR SHARE: IDENTIFY ROLES FROM CODE.

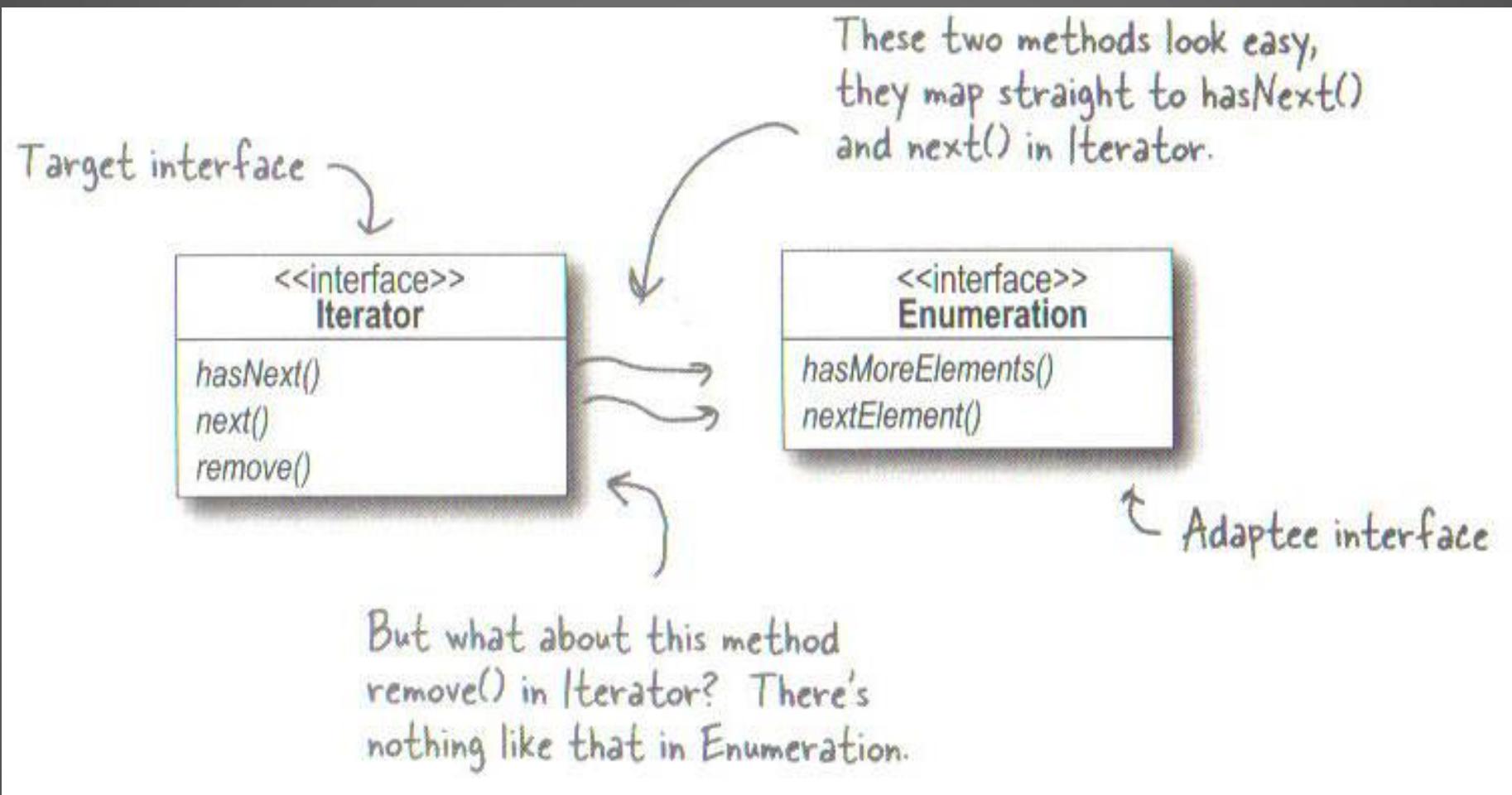


```
public class EnumerationIterator implements Iterator {  
    Enumeration enumeration;  
  
    public EnumerationIterator(Enumeration enumeration) {  
        this.enumeration = enumeration;  
    }  
  
    public boolean hasNext() {  
        return enumeration.hasMoreElements();  
    }  
  
    public Object next() {  
        return enumeration.nextElement();  
    }  
  
    public void remove() {  
        throw new UnsupportedOperationException();  
    }  
}
```

C++ STYLE CLASS ADAPTER



REAL WORLD ADAPTERS



THINK PAIR SHARE

- ▶ (1) Identify Roles: Client, Adaptor, Adaptee, Target
 - ▶ What is the not ideal interface of legacy code?
 - ▶ What is the ideal interface that Client wants to use?
- ▶ (2) Critique the first design according IH principle. The secret that Rectangle draws based on height and width is now exposed to Client code OldDemo. If you want to change how to draw, you have to change client code.
- ▶ (3) What is the benefit of the second design?

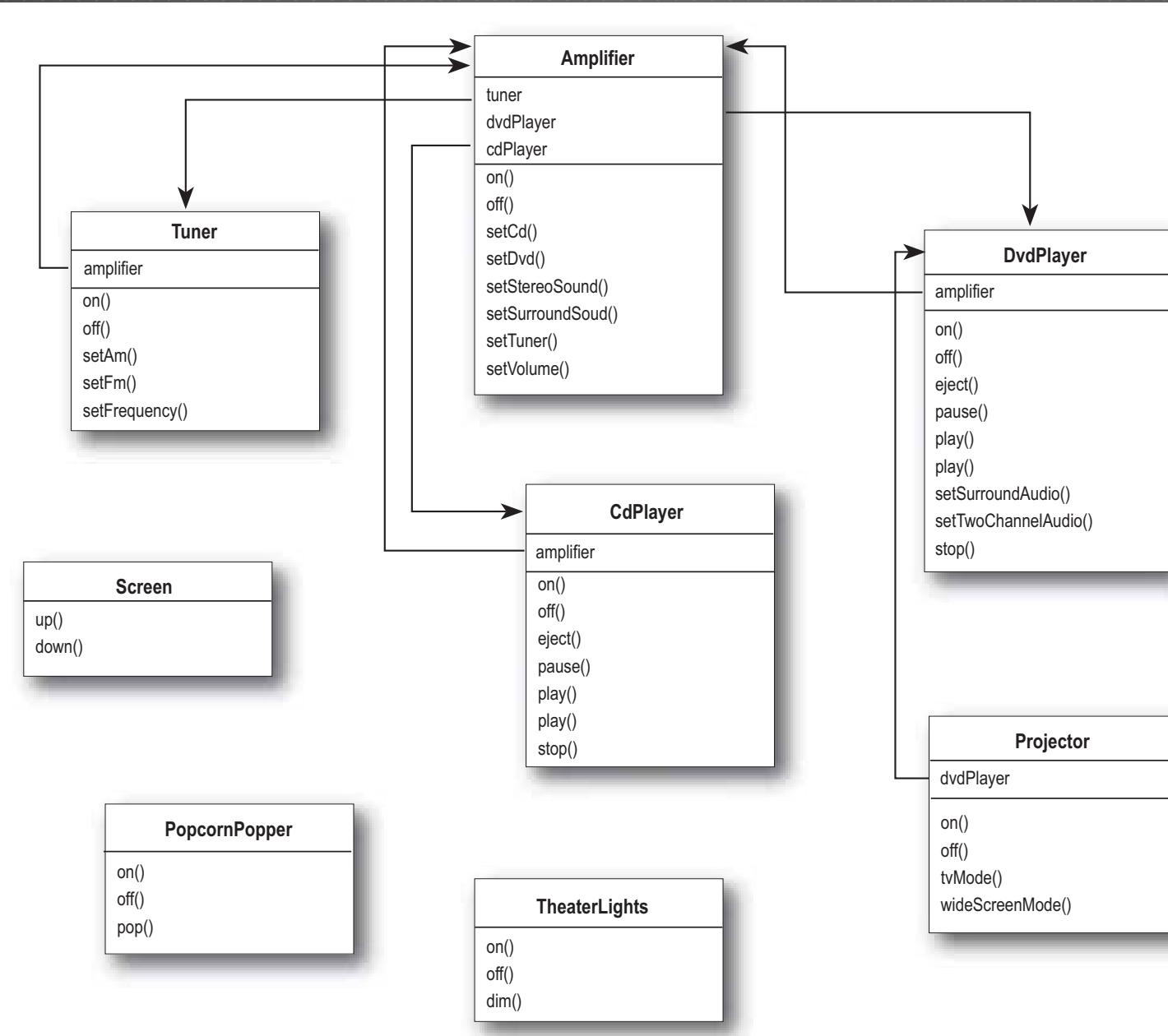
Now as long as new types of Shape conform to the interface of requiring end and star coordinates, you can change how to draw.



A special kind of adapter...

THE FAÇADE PATTERN

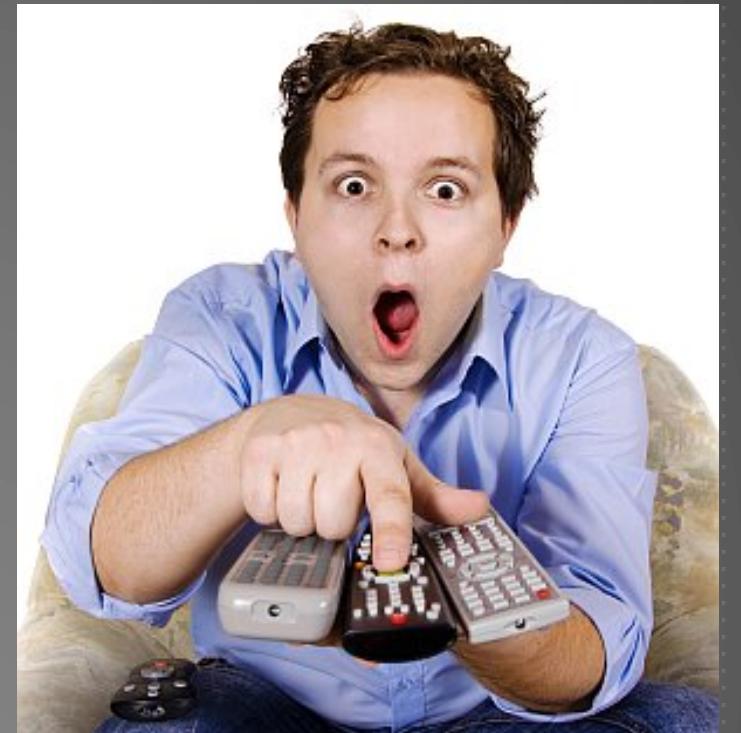
A HOME THEATER



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

WATCHING A MOVIE

- ▶ Sit back, relax, and...
 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 in 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



FURTHER COMPLICATIONS...

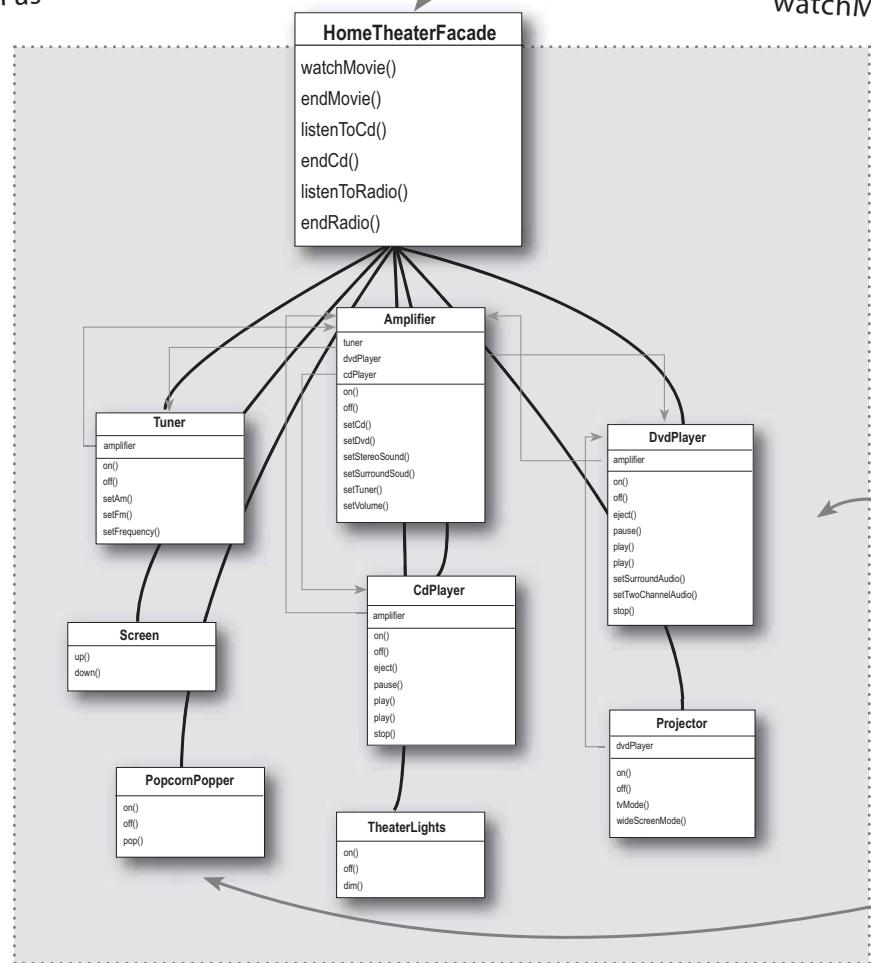
- ▶ When the movie finishes, you have to do it all in reverse!
- ▶ Doing a slightly different task (e.g., listen to streaming audio) is equally complex
- ▶ When you upgrade your system, you have to learn a slightly different procedure

watchMovie()

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.

The subsystem the Facade is simplifying.



MOTIVATION FOR FACADE

- ▶ You want to create a simple interface for a family of complex subsystems

SIDE BAR: FAÇADE VS. ADAPTER

- A façade not only simplifies an interface, but it also decouples a client from a subsystem of components
- Facades and adapters may wrap multiple classes
 - A façade's intent is to **simplify**
 - An adapter's intent is to **convert** the interface into something different
- A façade does not encapsulate; it just provides a simplified interface

THE FAÇADE PATTERN

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.

RECAP

- ▶ Singleton ensures a single object creation.
- ▶ Command decouples a receiver object's actions from invokers.
- ▶ Adaptor adapts legacy code to a target interface.
- ▶ Façade simplifies complex interfaces of multiple subsystems.

PREVIEW

- ▶ Chapter 8 Template Method
- ▶ Chapter 10 State

QUESTIONS?