CS 230 SOFTWARE ENGINEERING

DESIGN PATTERNS

SINGLETON, COMMAND, ADAPTER, FAÇADE

Professor Miryung Kim

UCLA Computer Science

AGENDA

- Singleton
- Command
- Adaptor

- Façade
- State

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

RECAP: STRATEGY VS. FACTORY METHOD

- What is the similarity between Strategy and Factory Method?
- What is the main difference between Strategy and Factory Method?

RECAP: STRATEGY VS. FACTORY METHOD

- What is the similarity between Strategy and Factory Method?
  - Use polymorphism through extends or implements
  - Flyable obj = ..
  - obj.performFly(); // this was the use of strategy
  - Factory obj =…
  - Pizza p = obj.createPizza(str);// this is the use of factory method.
- What is the main difference between Strategy and Factory Method?
  - The difference is that factory method has a return type that is of generic Product and it's about constructing objects

RECAP: FACTORY METHOD VS. ABSTRACT FACTORY

- What is the difference between FactoryMethod and AbstractFactory?

RECAP: FACTORY METHOD VS. ABSTRACT FACTORY

- What is the difference between FactoryMethod and AbstractFactory?
- // factory method version
- Pizza p = factory.createPizza(str); // how the factory method was invoked on the client side
- // abstract factory version
- IngredientCollection ingCol = new NYCollection();
- Factory factory = new NYPizzaFactory(ingColl);
- or Factory factory = new NYPizzaFactory();
  - this.ingredientCollection = new NYCollection(); // inside NYPizzaFactory constructor
- Pizza p = factory.createPizza(str) // this is also how the factory method was invoked on the client side when using abstract factory
- Regardless of how it was done, NYPizzaFactory knows which ingredient collection to use

RECAP: FACTORY METHOD VS. ABSTRACT FACTORY

- Abstract factory includes a factory method pattern (TRUE)
- Family/ Ingredients
- Pizza consists of a set of ingredients.
- But the constraints of choosing a set of ingredients is determined by "Concrete Factory"
- Ingredient collection object is passed around as an argument or set to a field by another method.

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?
  - Any one can mess around it.
  - Could be modified with another reference.
  - You might want to instantiate in the beginning of a program (e.g. class loading) => eager instantiation.

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

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

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

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

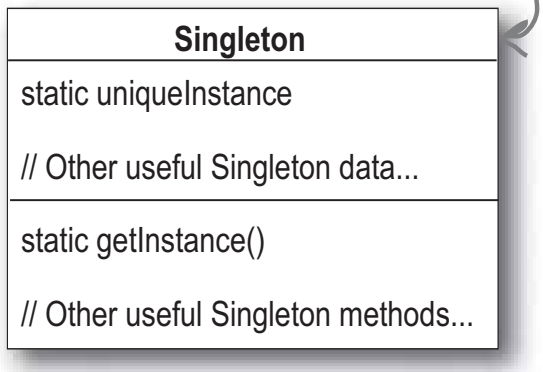
The `getInstance()` method gives us a way to instantiate the class and also to return an (the) instance of it.

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.



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

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

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

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

Notice that we only synchronize the first time through!

This approach can drastically reduce the overhead of synchronization

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

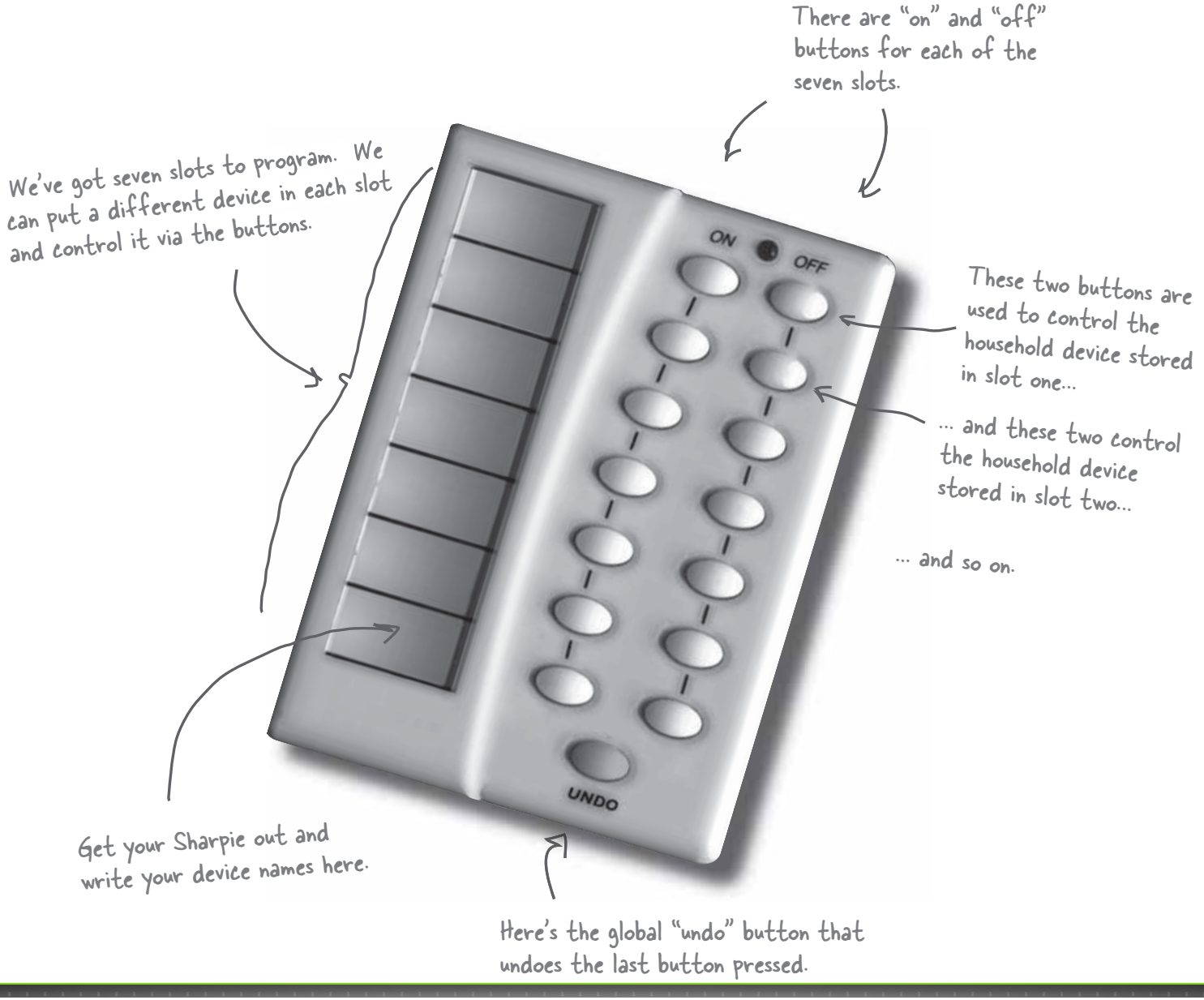
MENTI

- volatile keyword is also used to communicate the content of memory between threads.
- volatile variable, it's guaranteed that all reader thread will see updated value of the volatile variable once write operation completed.
- volatile keyword in Java guarantees that value of the volatile variable will always be read from main memory and not from Thread's local cache.

- volatile variable will prevent the compiler from doing any reordering or any kind of optimization which is not desirable in a multi-threaded environment.
- The volatile keyword in Java is a field modifier while synchronized modifies code blocks and methods.
- Java volatile keyword doesn't mean atomic, its common misconception that after declaring volatile will be atomic, to make the operation atomic you still need to ensure exclusive access using synchronized method or block in Java.

THE COMMAND PATTERN

THE MISSION: A REMOTE CONTROL



THE VENDOR CLASSES

ApplianceControl
- on()
- off()

Stereo
- on()
- off()
- setCd()
- setDvd()
- setRadio()
- setVolume()

CeilingLight
- on()
- off()
- dim()

OutdoorLight
- on()
- off()

TV
- on()
- off()
- setInputChannel()
- setVolume()

FaucetControl
- openValue()
- closeValue()

CeilingFan
- high()
- medium()
- low()
- off()
- getSpeed()

Hottub
- circulate()
- jetsOn()
- jetsOff()
- setTemperaturet()

GardenLight
- setDuskTime()
- setDawnTime()
- manualOn()
- manualOff()

GarageDoor
- up()
- down()
- stop()
- lightOn()
- lightOff()

Thermostat
- setTemperature()

Sprinkler
- waterOn()
- waterOff()

SecurityControl
- arm()
- disarm()

Light
- on()
- off()

TOWARDS A DESIGN

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

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

We need some *information hiding* and *separation of concerns*

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

The remote is simple, but the devices are not!

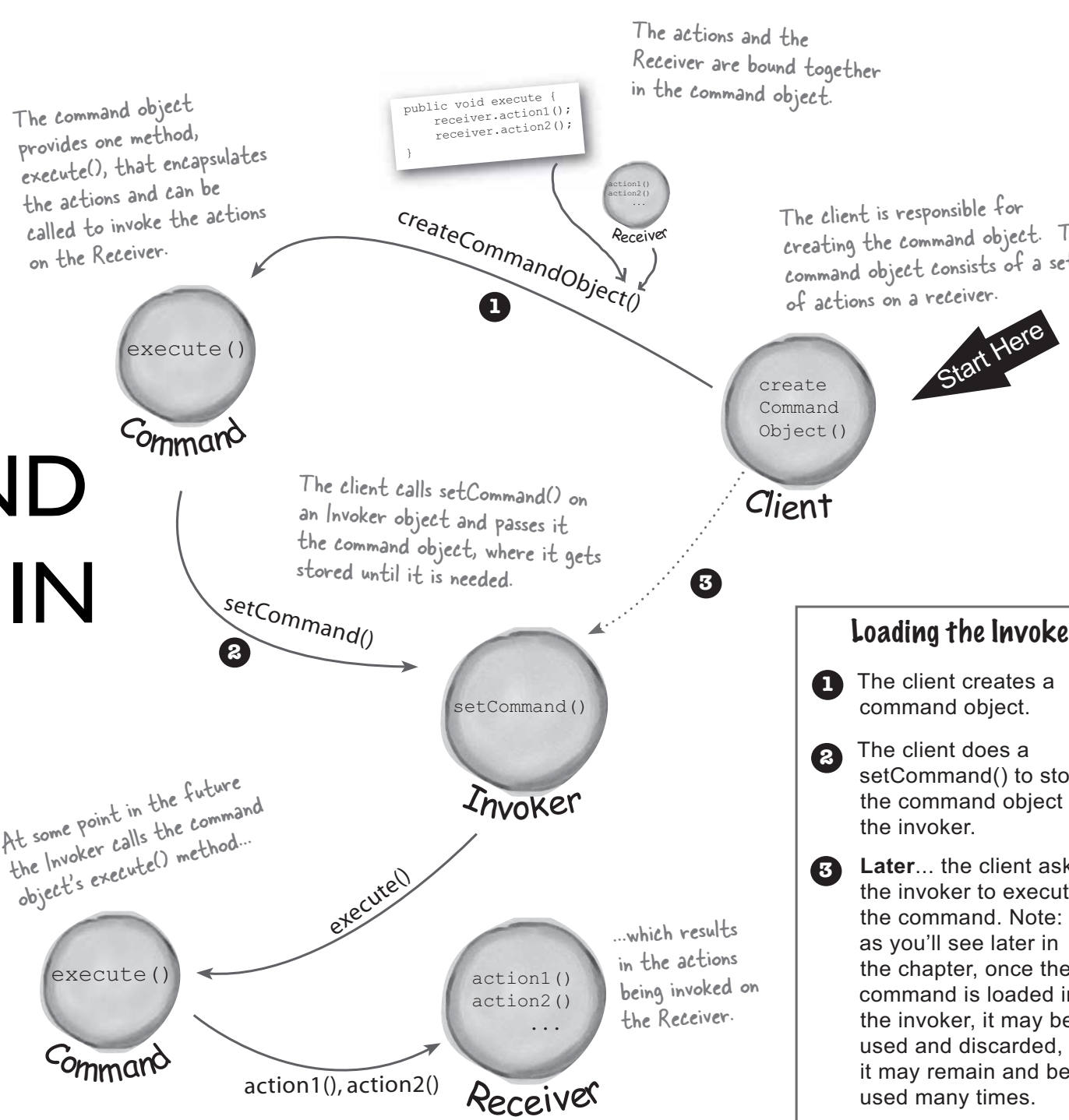
MOTIVATION FOR COMMAND

- 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.

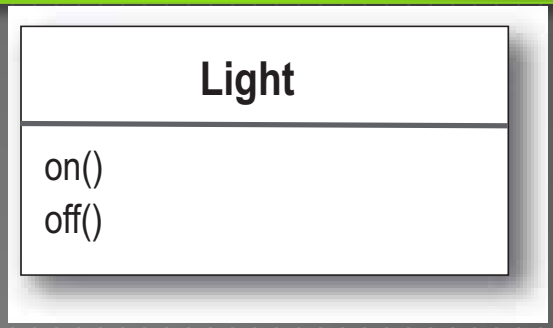
ROLES

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

A FIRST COMMAND OBJECT

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

Simple. Just one method called `execute()`




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

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

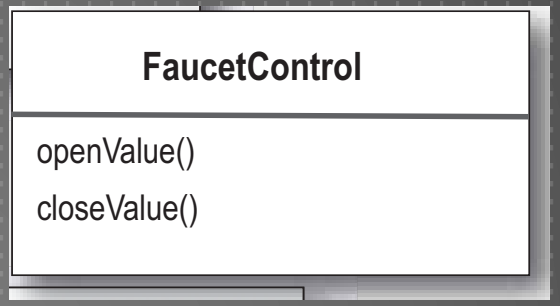
Then we pass the command to the Invoker.

WHAT IS DIFFERENCE BETWEEN COMMAND AND MEDIATOR?

- Mediator is an intermediate where multiple colleagues communicates. Reduces n-n complex communication to n-1 communication.
- Command is an interface for an action that encapsulates an underlying device.
- Command is used to create an interface that is not device dependent, because devices (receiver objects performing the actions) could come and go.
- Devices have different sets of names for their actions.

IMPLEMENTING A DIFFERENT COMMAND ON A NEW OBJECT

- Implement the `FaucetOffCommand` class
- Here's the new "test" code:

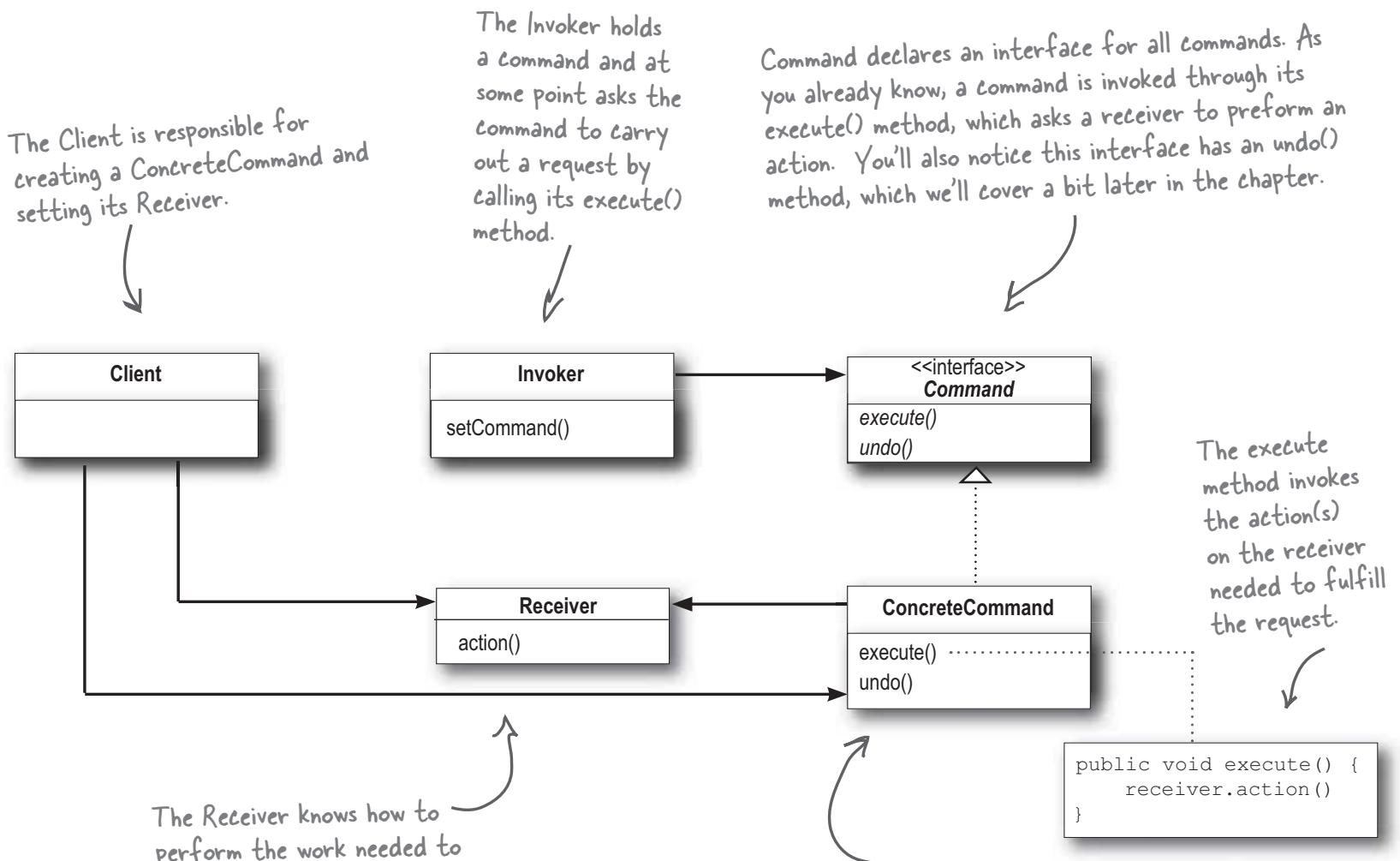



```
public class RemoteControlTest {
  public static void main(String[] args) {
    SimpleRemoteControl remote = new SimpleRemoteControl();
    Faucet faucet = new FaucetControl();
    FaucetOffCommand faucetOff = new FaucetOffCommand(faucet);
    remote.setCommand(faucetOff);
    remote.buttonWasPressed();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

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 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.

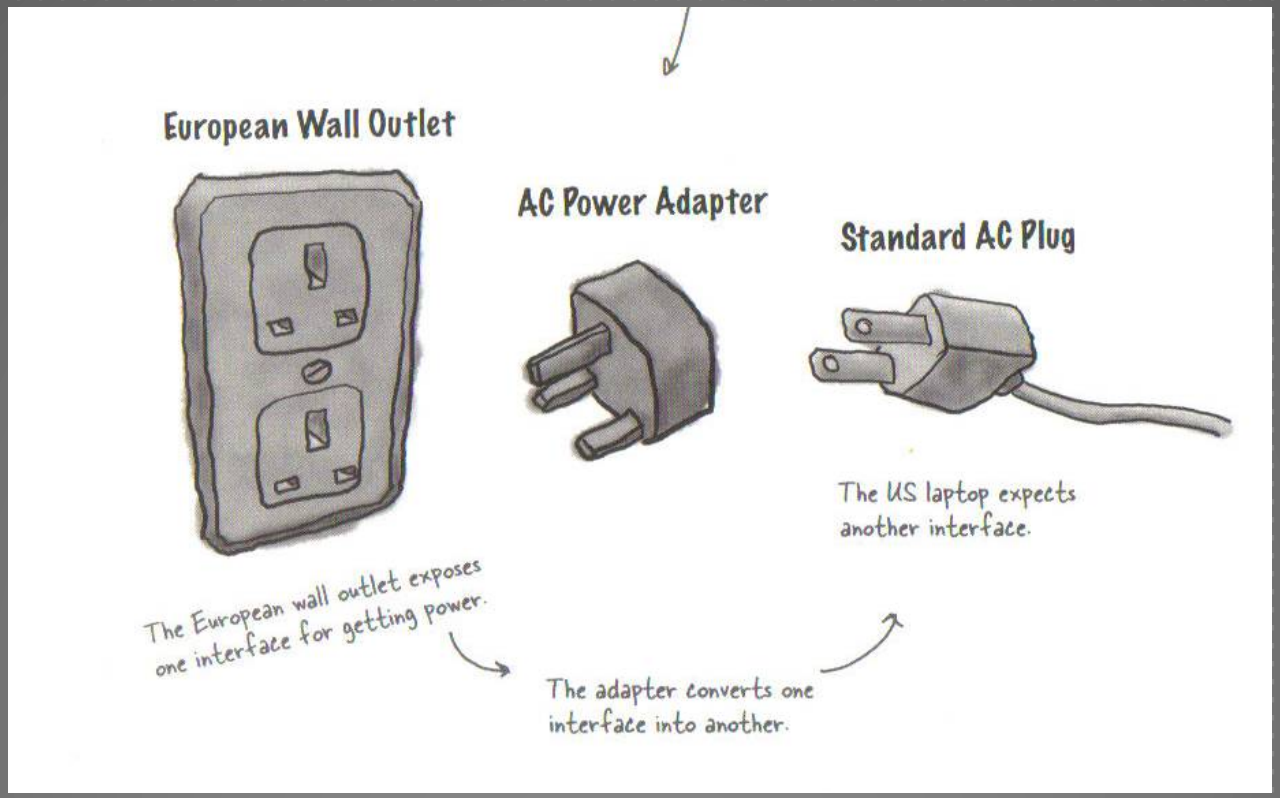
- How does the remote know the difference between the kitchen light and the living room light?

QUESTION 2. (MENTI)

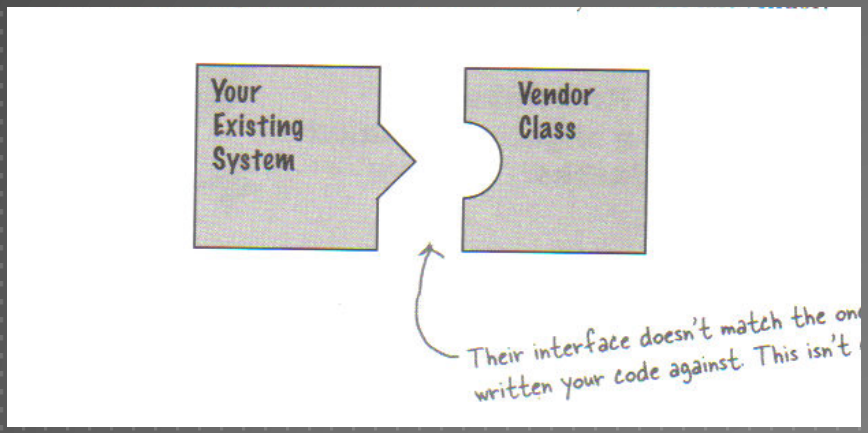
- How would you implement a history of undo operations?

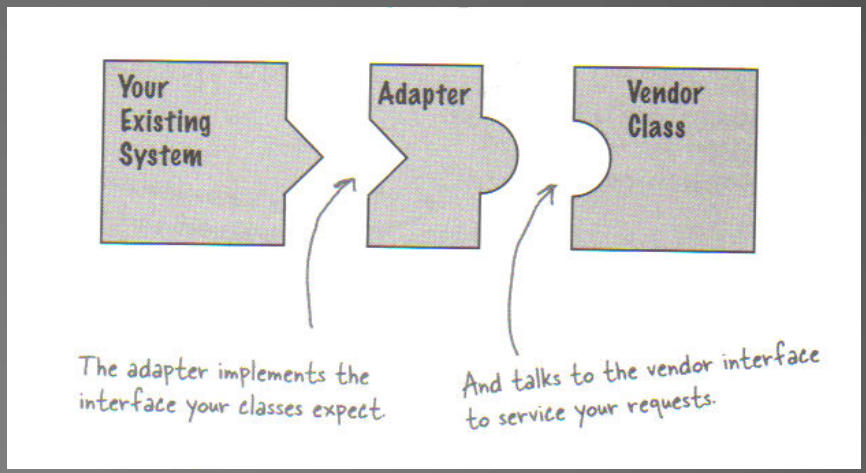
ADAPTER

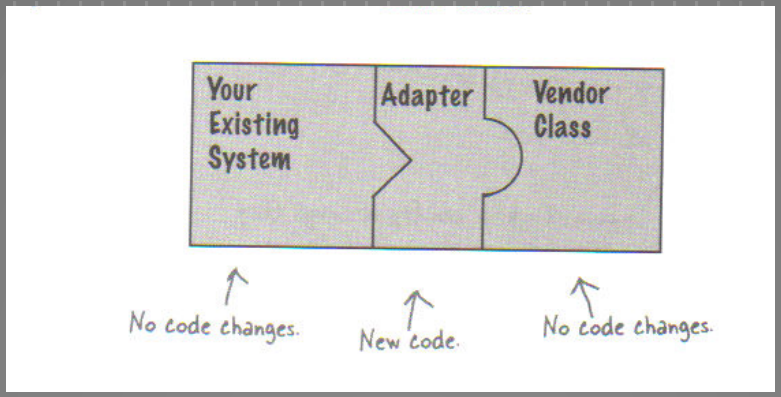
ADAPTERS IN REAL LIFE



OBJECT-ORIENTED ADAPTERS





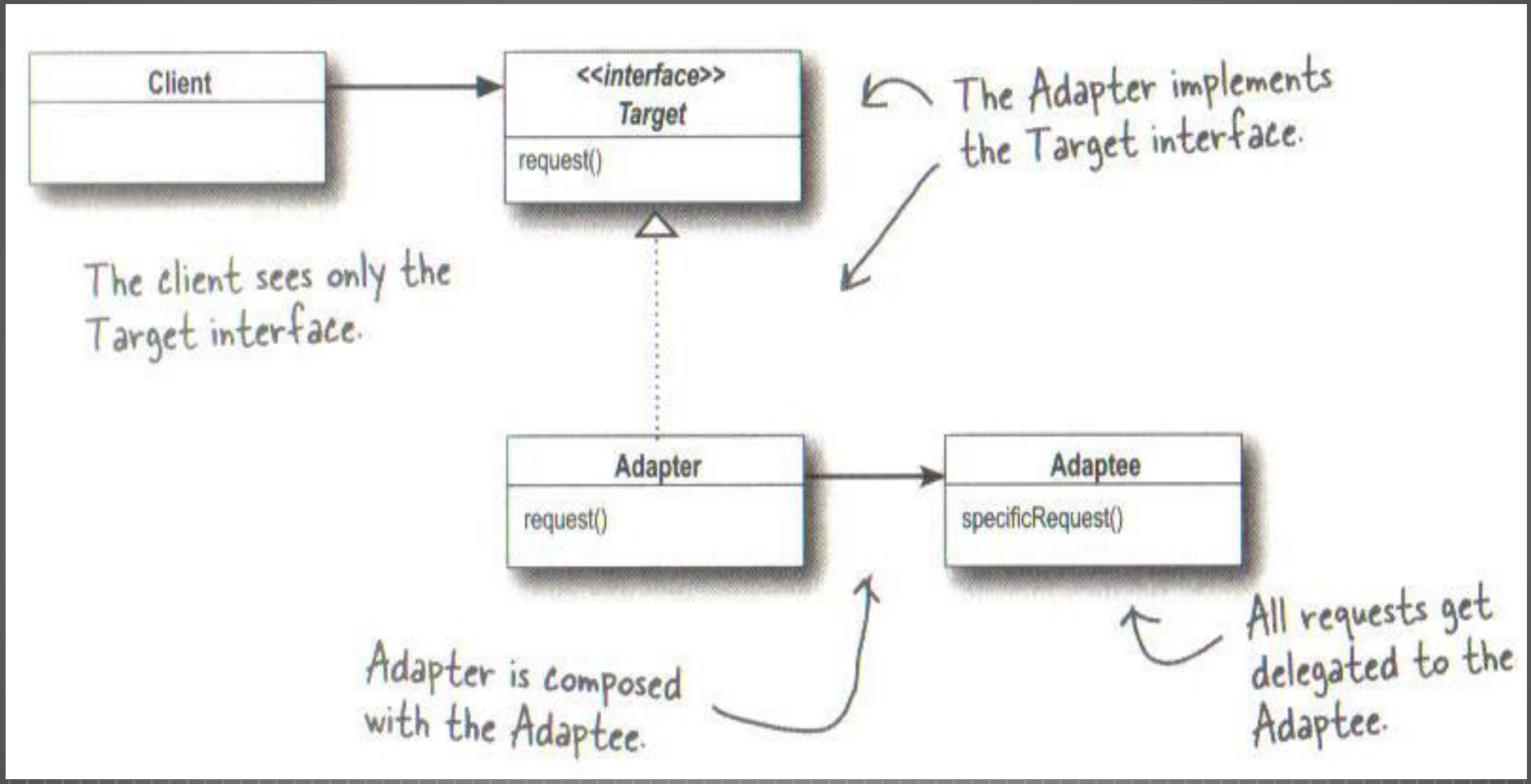


Page 237 Head First Design Patterns

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();
            }
    }
}
```

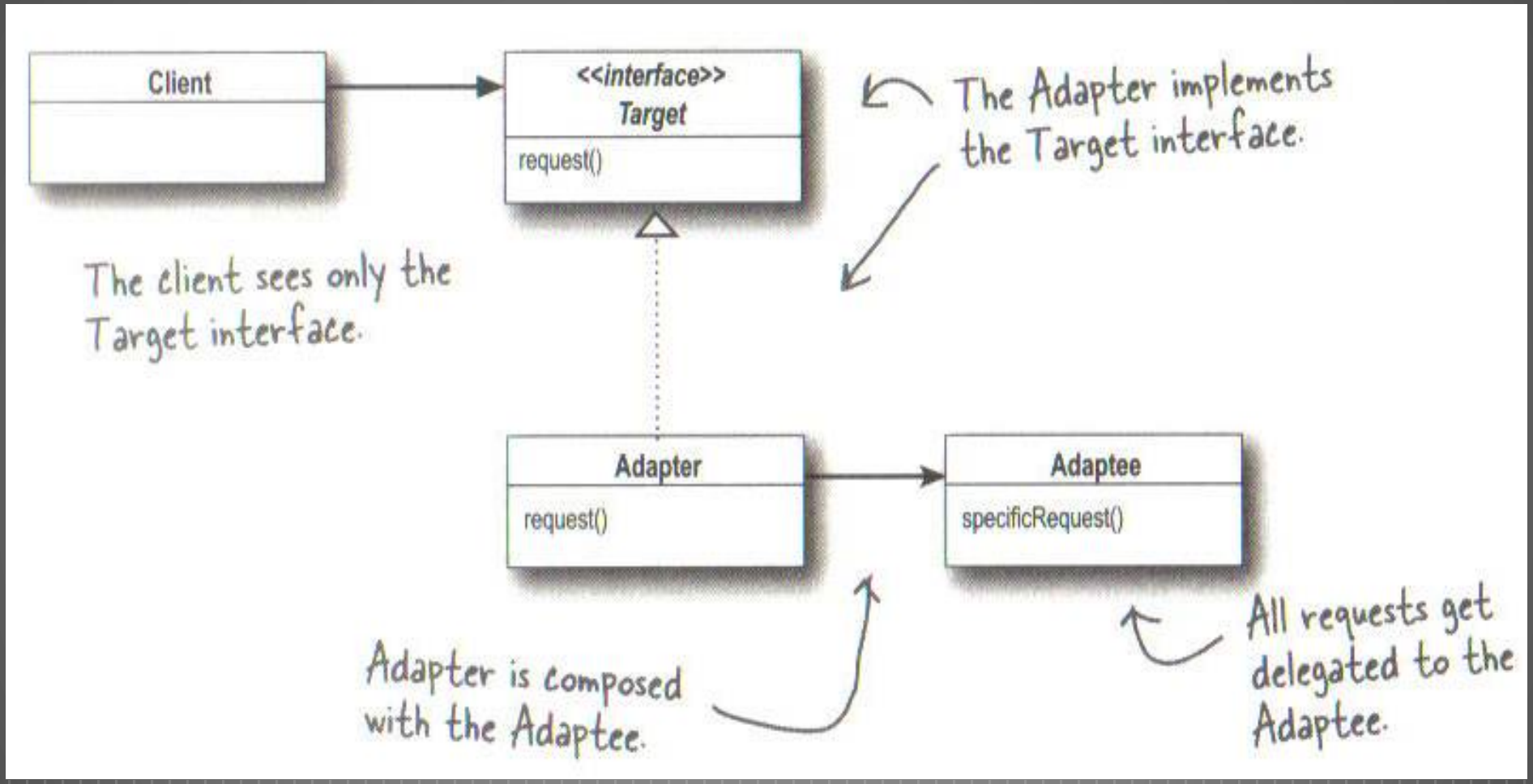
NOW LET'S TAKE A LOOK AT SKELETON CODE

- Handout-DesignPatternSkeletons

ADAPTER PATTERN DEFINED

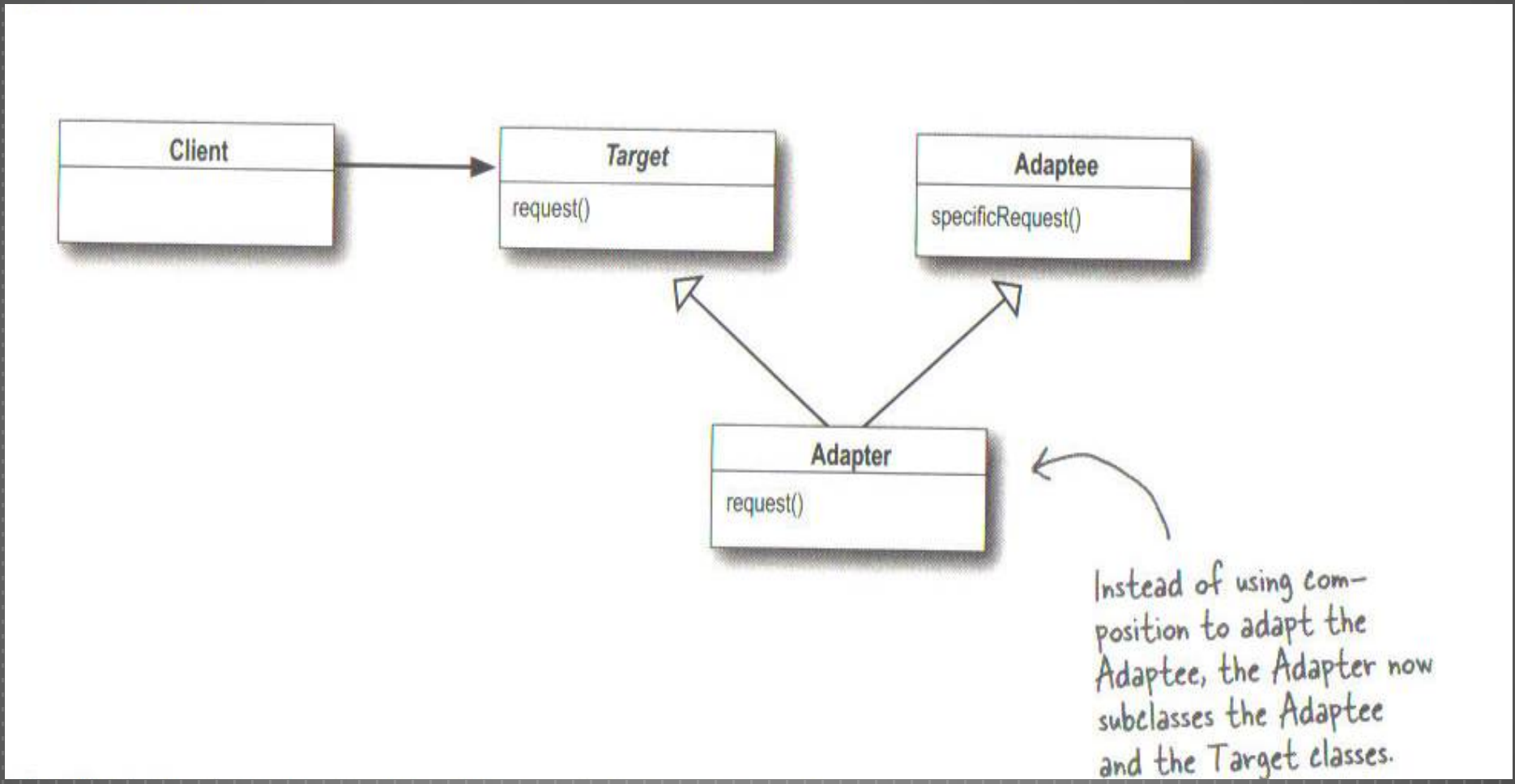
The Adapter Pattern converts the interface of a class into another interface the clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

THINK PAIR SHARE: IDENTIFY ROLES FROM CODE.

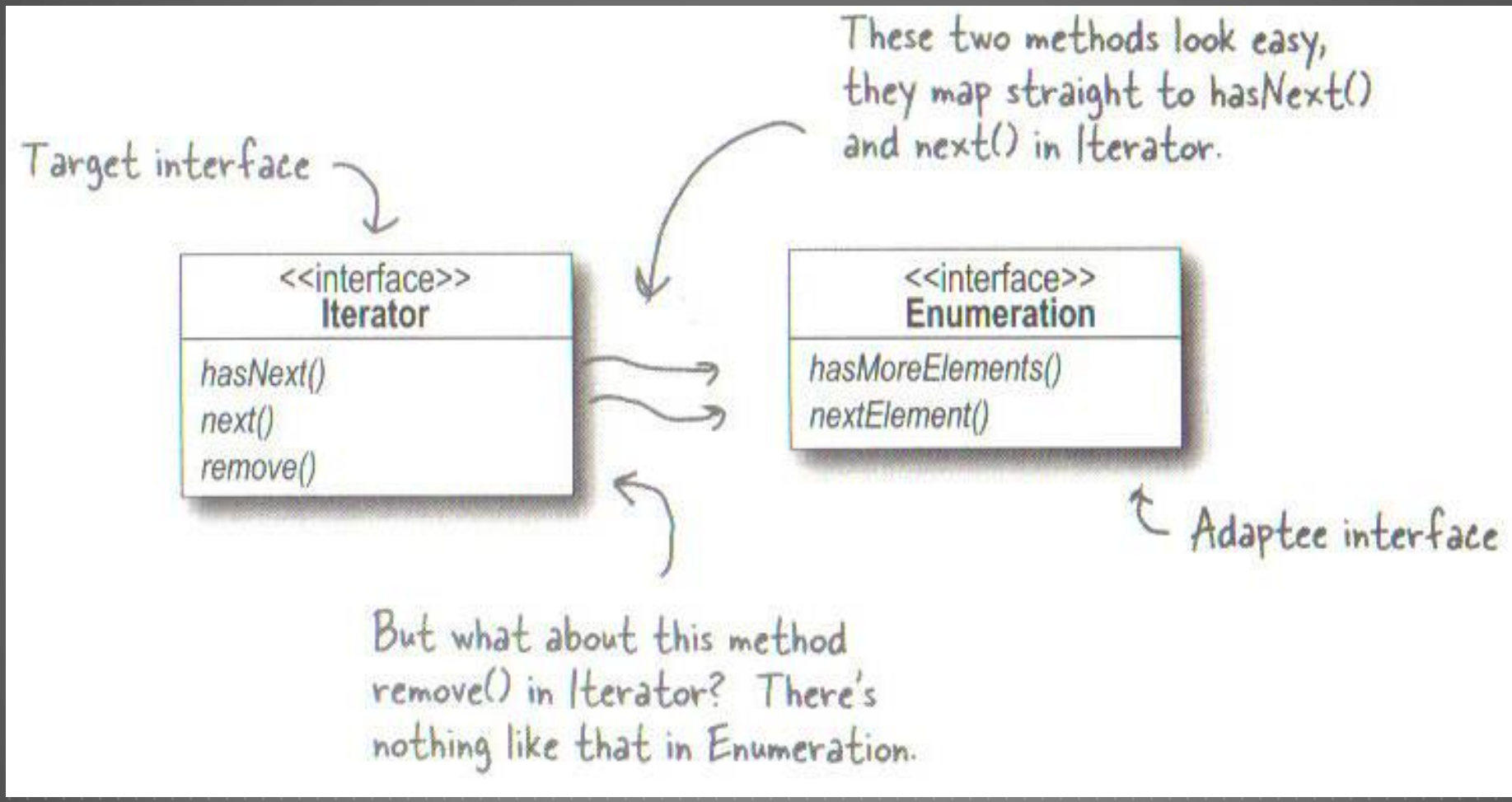


```
public class EnumerationIterator implements Iterator {
    Enumeration enumeration;

    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?

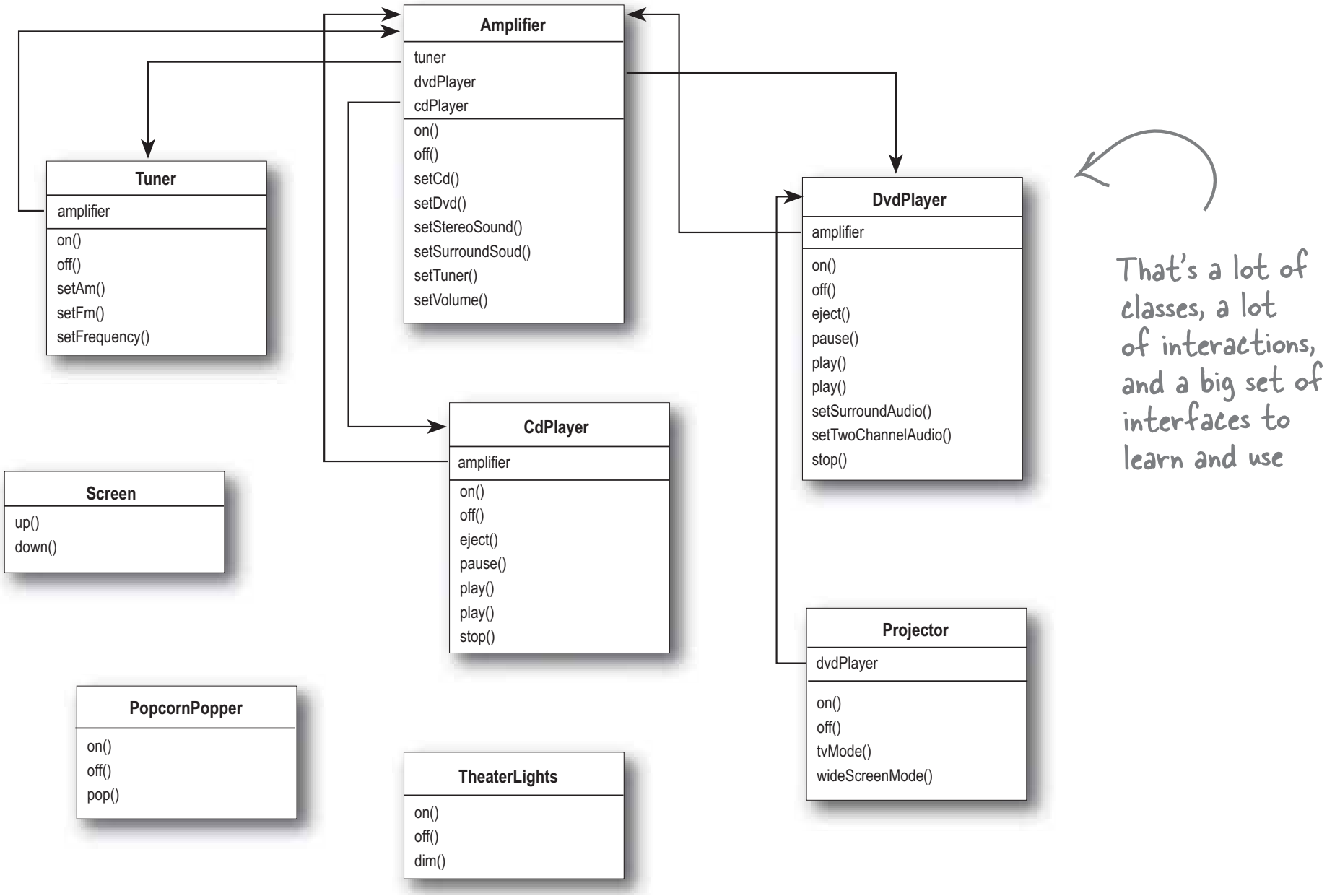
  Now as long as new types of Shape conform to the interface of requiring end and star coordinates, you can change how to draw.

See Handout-DesignPatternsMerged.pdf

A special kind of adapter…

THE FAÇADE PATTERN

A HOME THEATER



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…

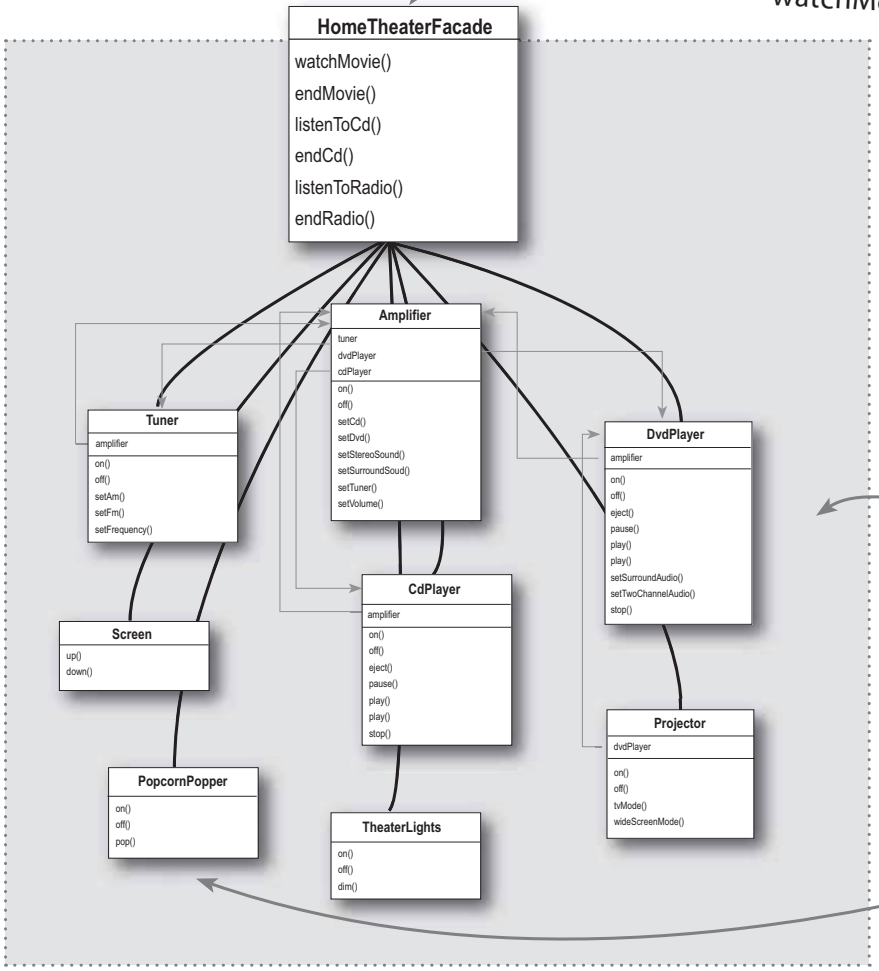
- When the movie finishes, you have to do it all in reverse!
- Doing a slightly different task (e.g., listen to streaming audio) is equally complex
- When you upgrade your system, you have to learn a slightly different procedure

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().

watchMovie()

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

The Facade



The subsystem the Facade is simplifying.

play()

on()

MOTIVATION FOR FACADE

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

SIDEBAR: 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 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.

QUESTIONS?