DESIGN PATTERNS III

Yuqun Zhang
CS304

Figures from Head First Design Patterns

THE SINGLETON PATTERN

The Notion of a 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)
- We could just use global (static) variables
  - The Singleton pattern gives all of the upsides without the downsides
    - E.g., object isn't forced to be created when the application starts
- Basically, the Singleton is used anytime you want every object in the application to use the same global resource

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?
    ```

The Classic Singleton

```
public class Singleton {
  private static Singleton uniqueInsta
  // …
  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!

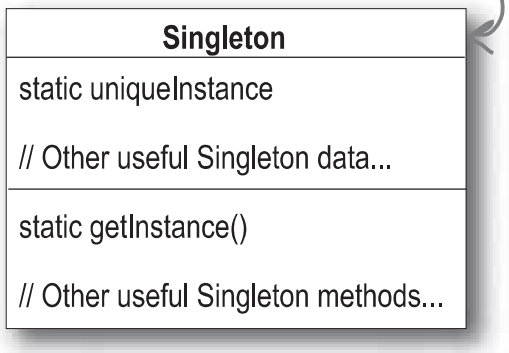
The `getInstance()` method gives us a way to instantiate the class and also to return an (the) instance of it.

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.

We have a problem…

- 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
- Show a sequence of legal calls to the `Singleton` object's `getInstance()` method (including its internals) that results in the instantiation of two objects

An example

```
1  public static ChocolateBoiler getInstance() {
2      if (uniqueInstance == null) {
3          uniqueInstance = new ChocolateBoiler();
       }
4      return uniqueInstance;
}
```

The "Fix" is Easy

```
public class Singleton {
  private static Singleton uniqueInstance;
  // …
  private Singleton() {}
  public static synchronized S
    if (uniqueInstance == null
      uniqueInstance = new Sin
    }
    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
-

```
public class Singleton {
   private volatile static S
   private Singleton () {}
   public static Singleton getInstance() {
      if (uniqueInstance == n
         synchronized (Singlet
            if (uniqueInstance
               uniqueInstance =
            }
         }
      }
      return uniqueInstance;
   }
}
```

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

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

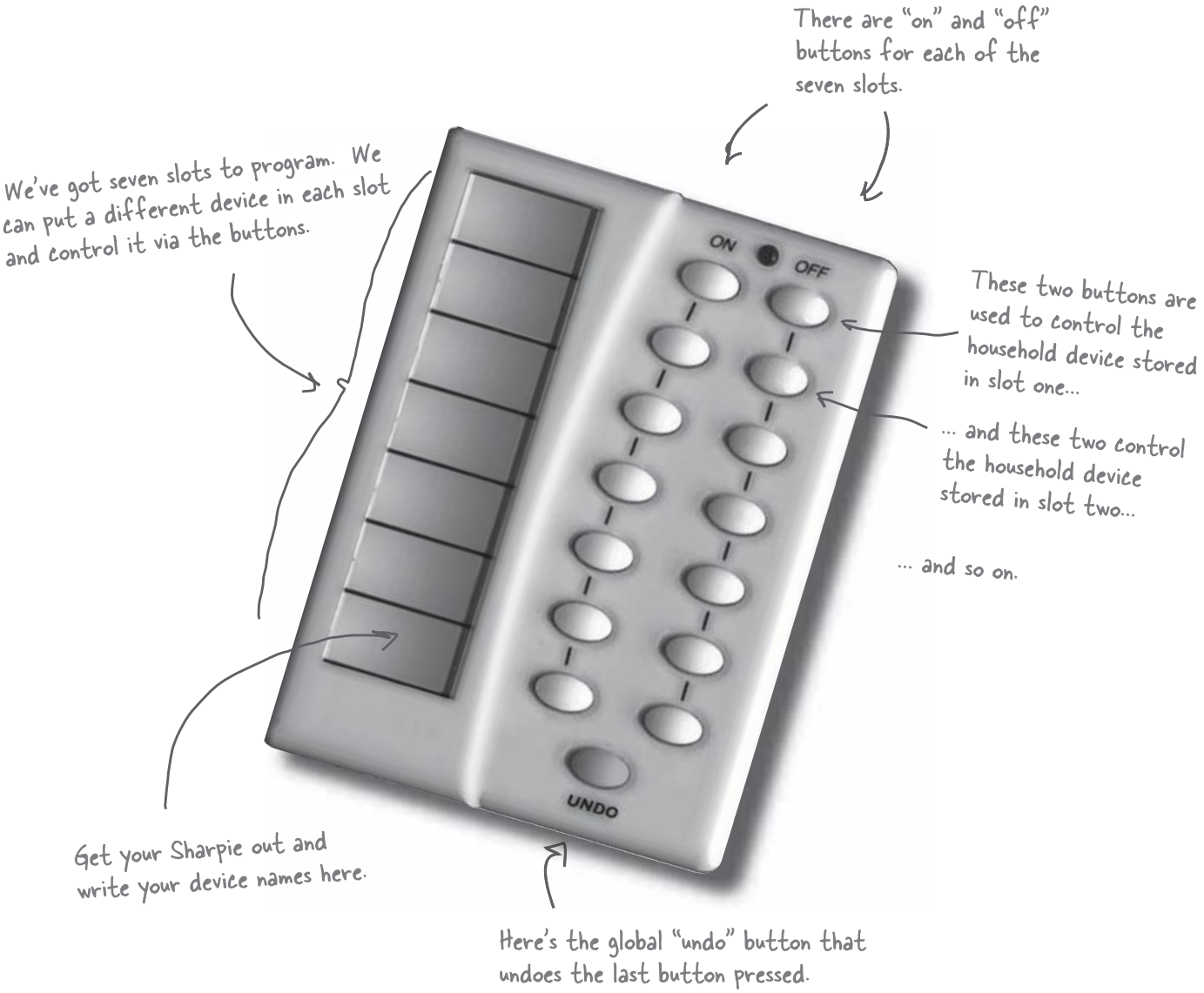
Once in the block, check again and if still null, create an instance

Notice that we only synchronize the first time through!

This approach can drastically reduce the overhead of synchronization

THE COMMAND PATTERN

The Mission: A Remote Control



The Vendor Classes

Towards a Design

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

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

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

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

The remote is simple, but the devices are not!

Command Objects (in context)

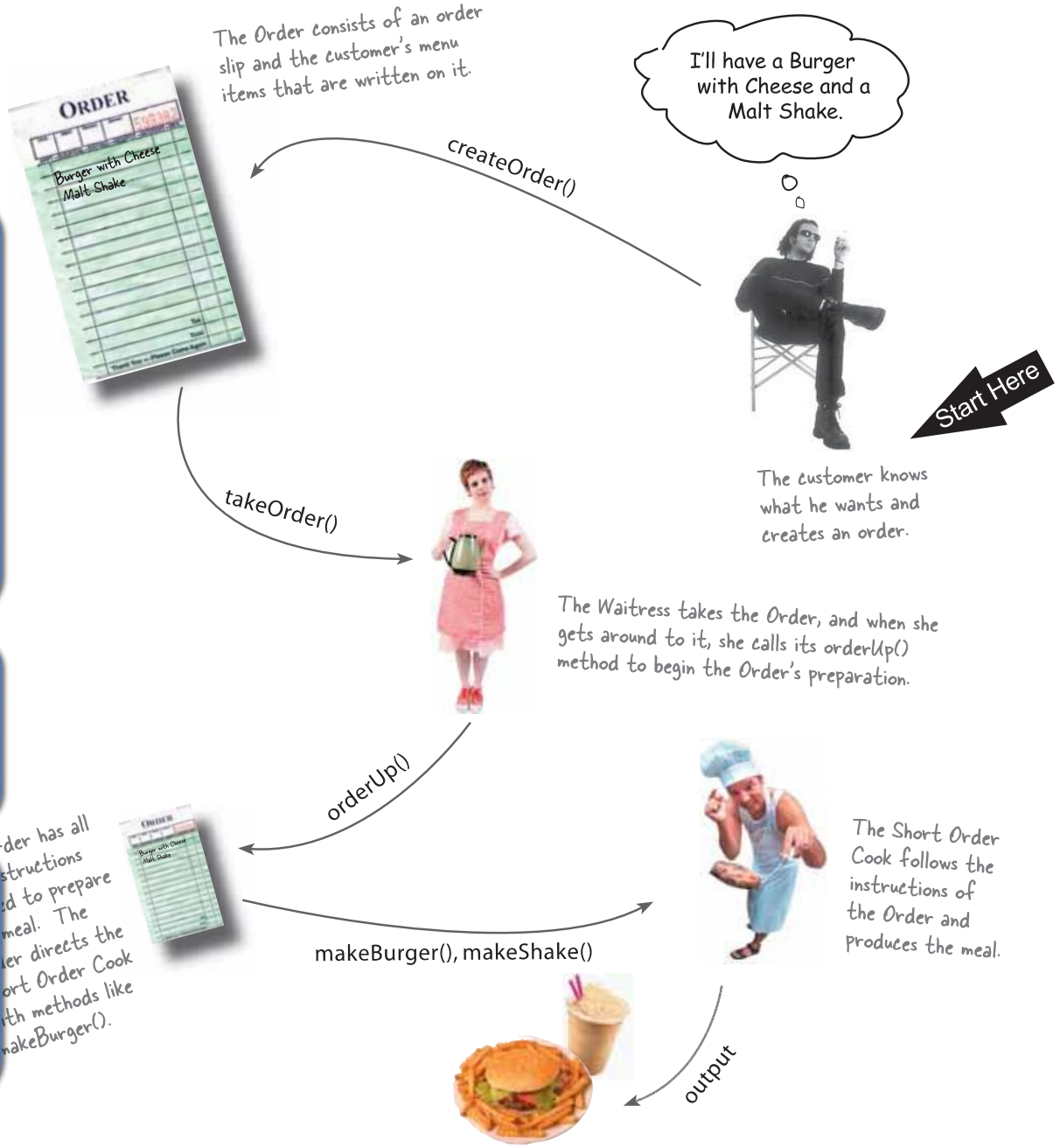
- We introduce **command objects** into the design
  - A command object encapsulates a request to do something (e.g., turn on a light) on a specific object (e.g., the living room lamp)
  - We can then just store a command object for each button such that when the button is pressed, the command is invoked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

An Analogy

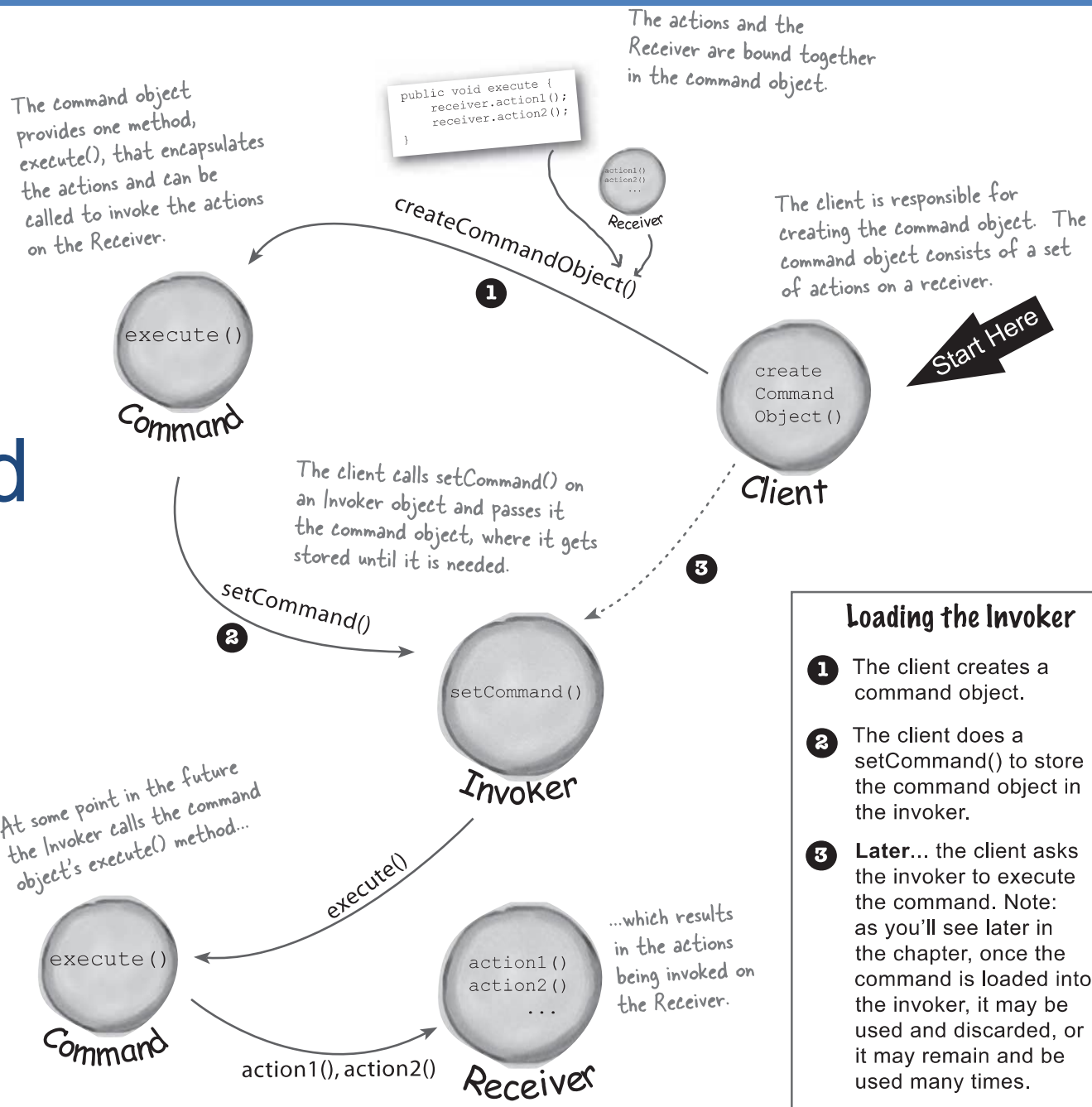
An order slip encapsulates a request to prepare a meal. It's method `orderUp()` encapsulates the actions needed to prepare the meal; it also carries its own reference to the appropriate `Cook`

The waitress just creates order slips and invokes the `orderUp()` method.

The Cook knows how to prepare the meals; but he's completely decoupled from the waitress (they need never directly communicate)



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.

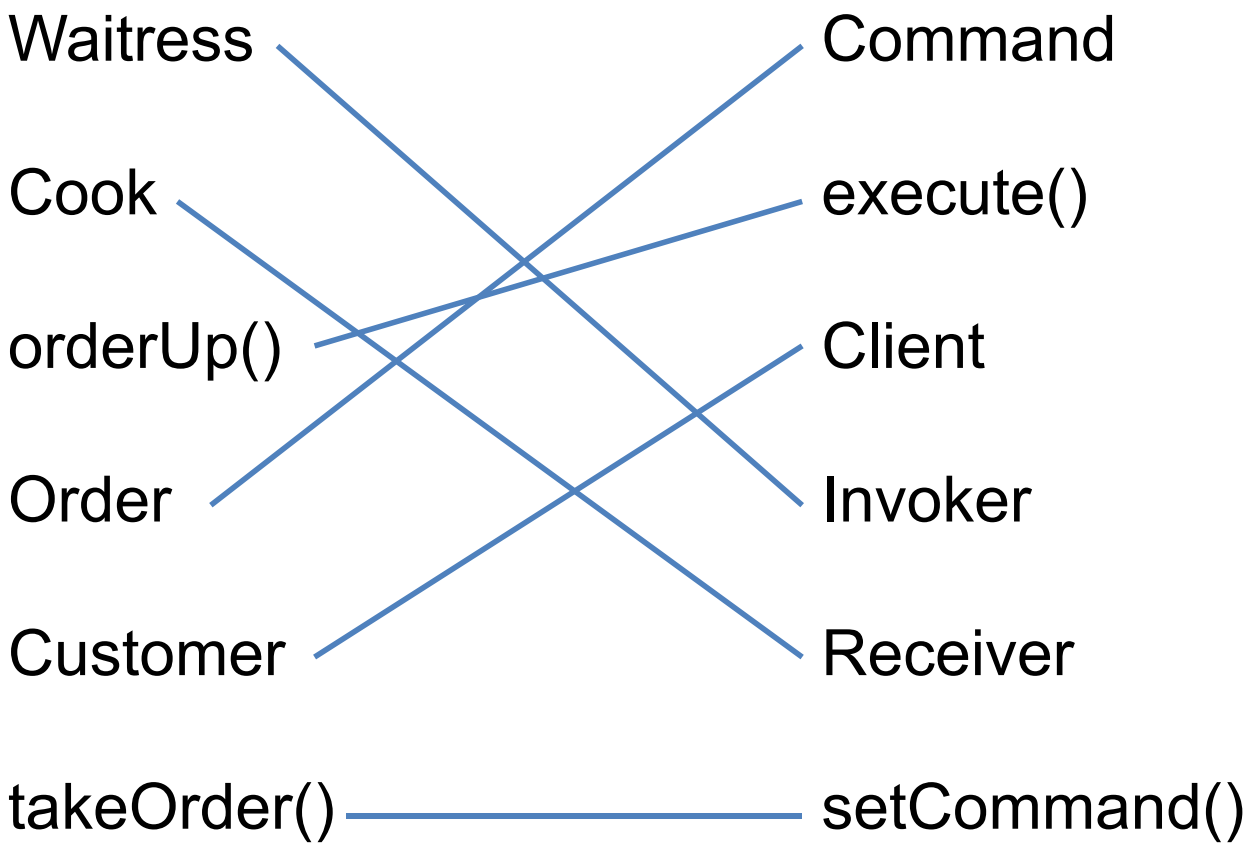
Quick Quiz

- Match the diner objects to the Command Pattern

Waitress	Command
Cook	execute()
orderUp()	Client
Order	Invoker
Customer	Receiver
takeOrder()	setCommand()

Quick Quiz

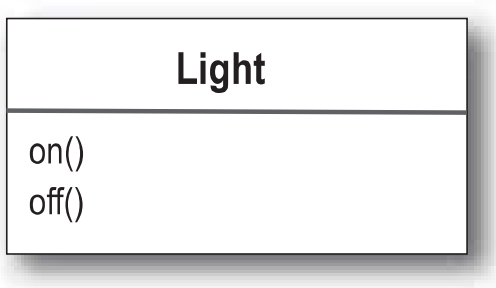
- Match the diner objects to the Command Pattern



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

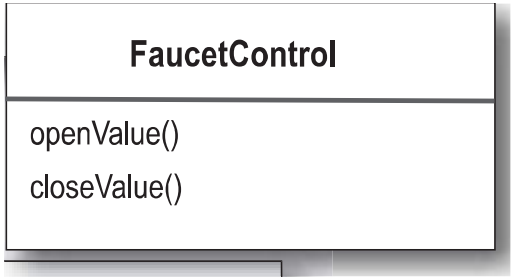
The `Light` object is the Receiver of the request

We create a command and pass it the Receiver

Then we pass the command to the Invoker.

Can you do it?

- 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();
  }
}
```

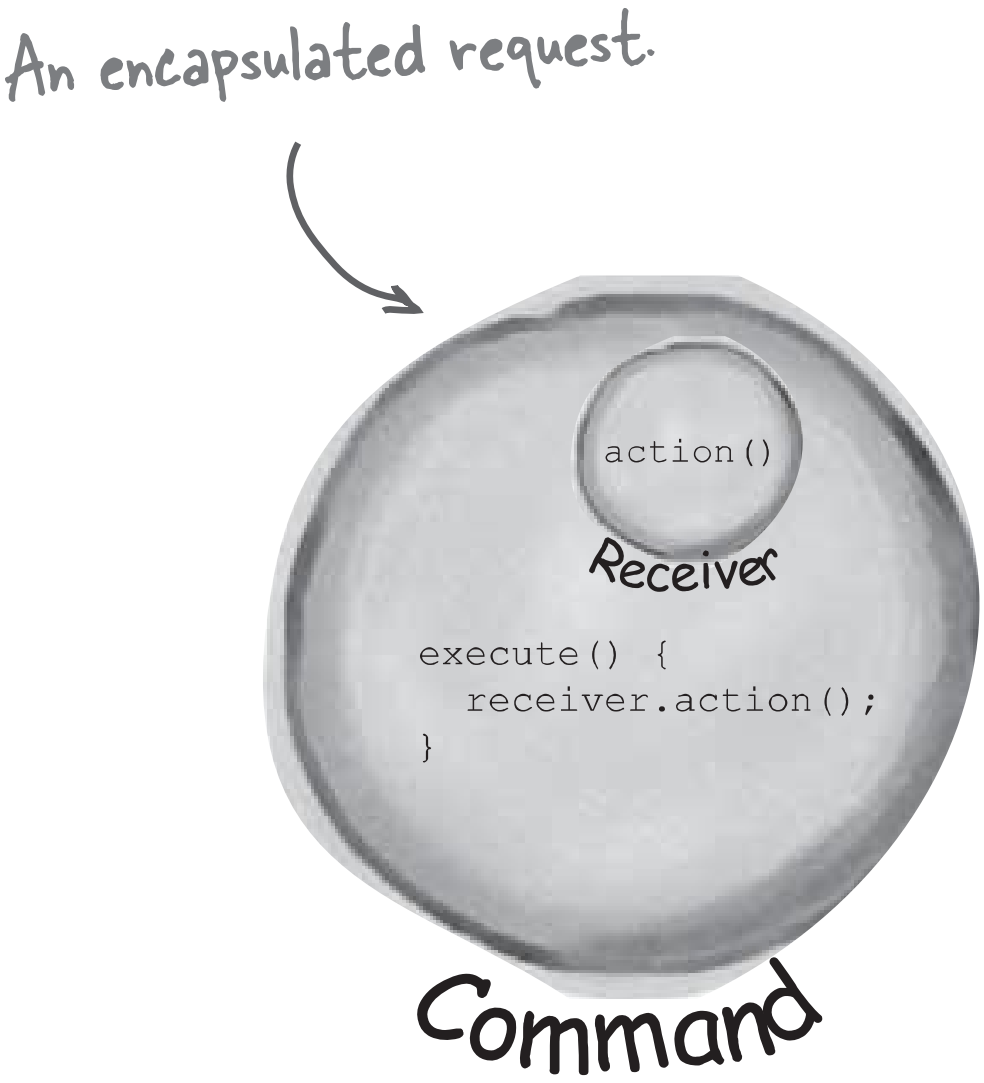
Solution

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

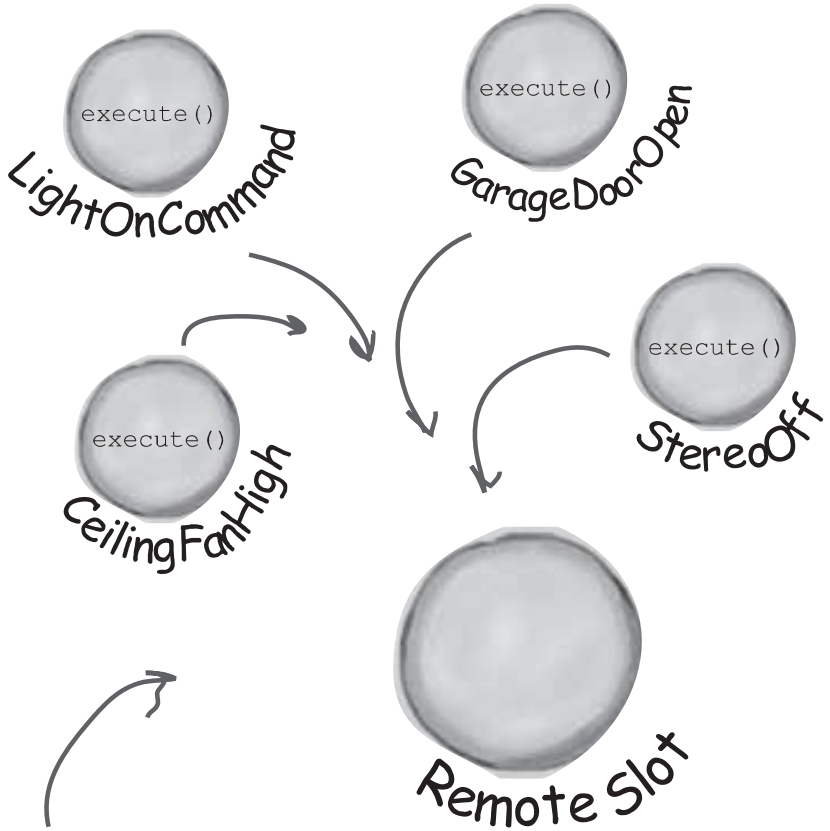
The Command Pattern

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.

The Command Pattern

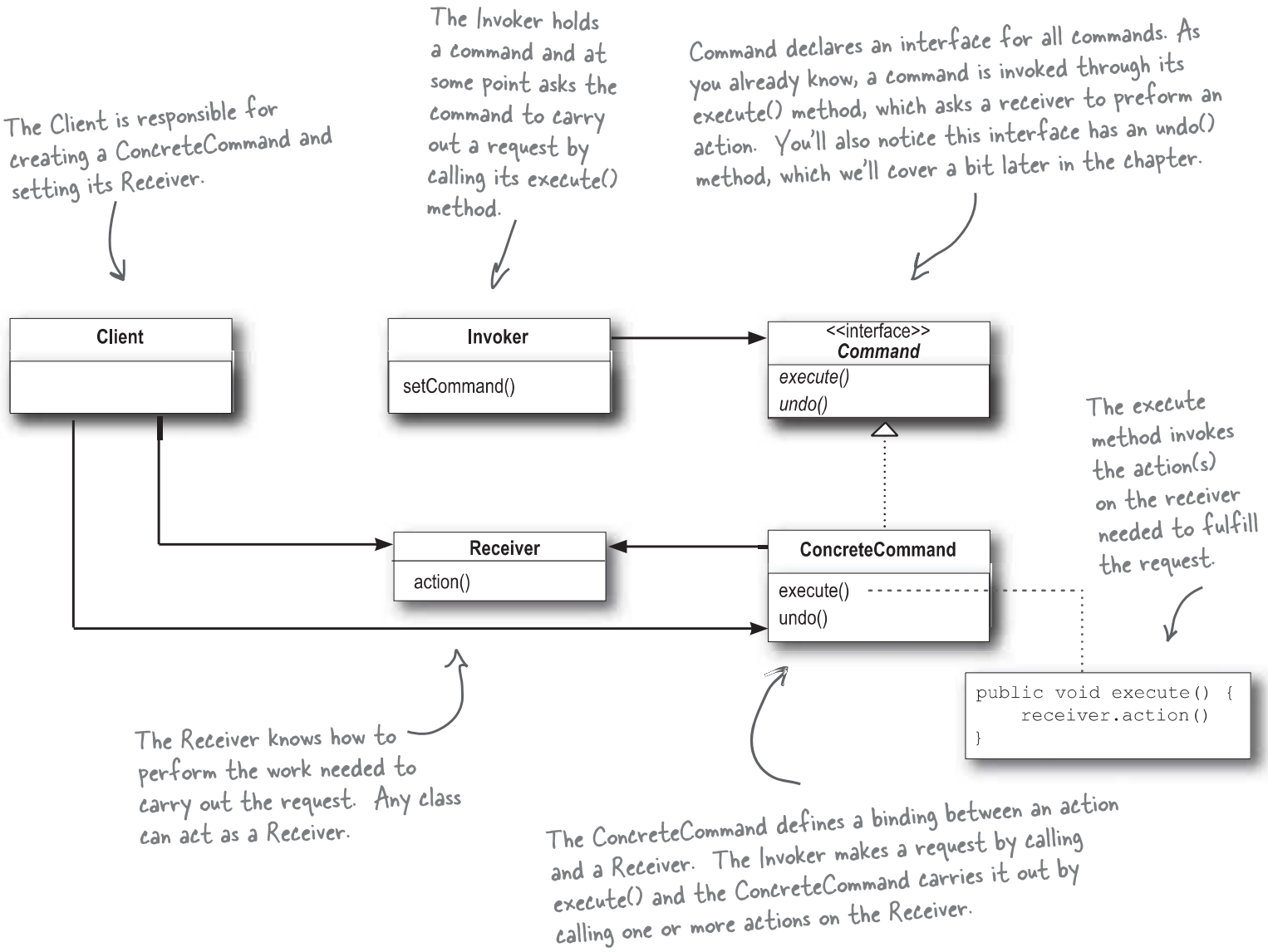


The Command Pattern in Action

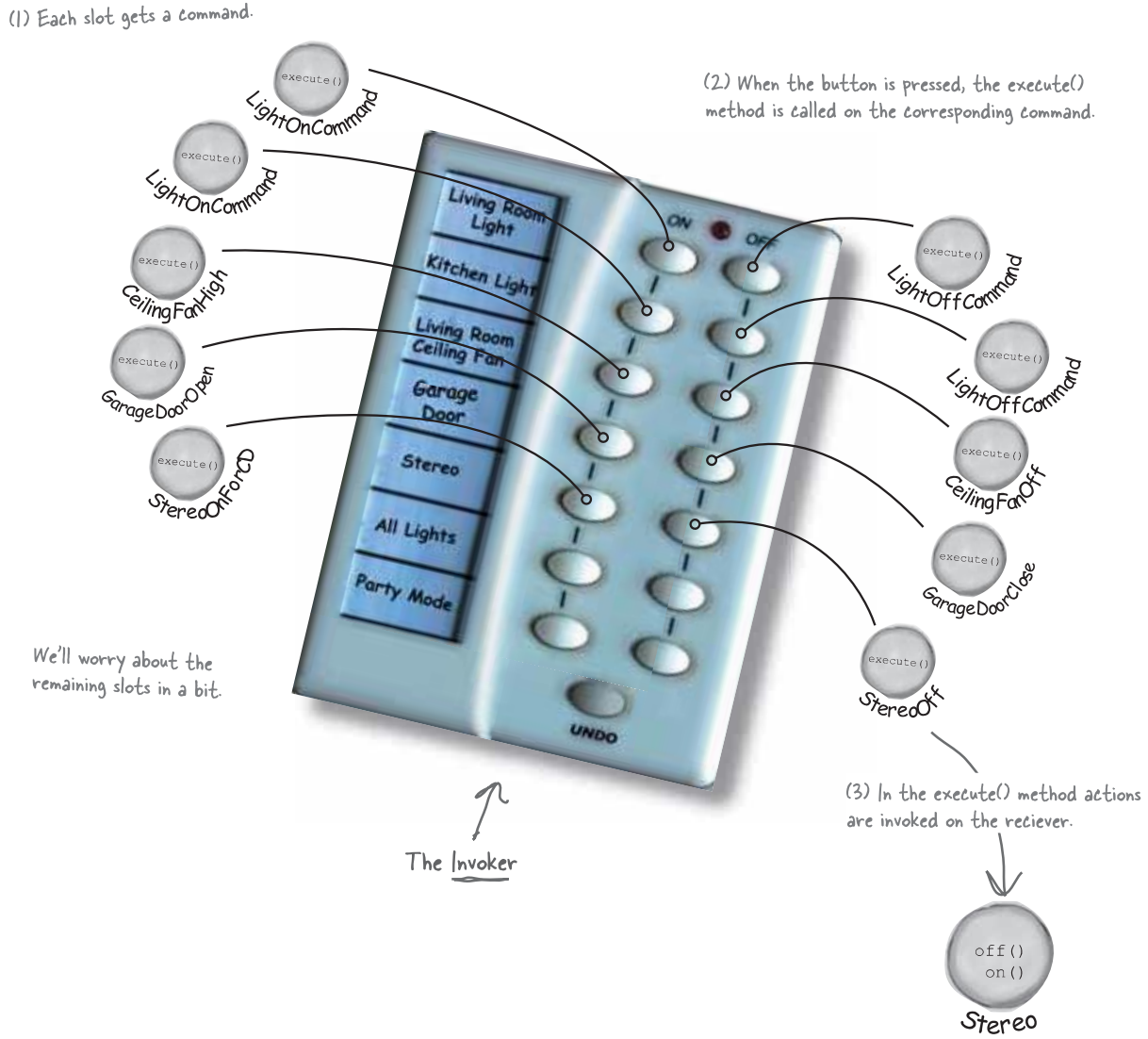


An invoker – for instance one slot of the remote – can be parameterized with different requests.

The Command Pattern Class Diagram



Back to our Remote…



Question

- How does the remote know the difference between the kitchen light and the living room light?
  - It doesn't have to! The Receiver is encapsulated in the command that is inserted in the "slot."

Implement Remote Control

```
public class RemoteControl{
    Command[] onCommands;
    Command[] offCommands;

    public RemoteControl(){
        onCommands = new Command[7];
        ...;
    }

    public void setCommand(int slot, Command onCommand, Command offCommand){
        onCommands[slot] = onCommand;
        ...;
    }

    public void onButtonWasPushed (int slot){
        onCommands[slot].execute();
    }

    ...;
}
```

Let's Add Support for the Undo Button

- First, let's expand the `Command` interface:

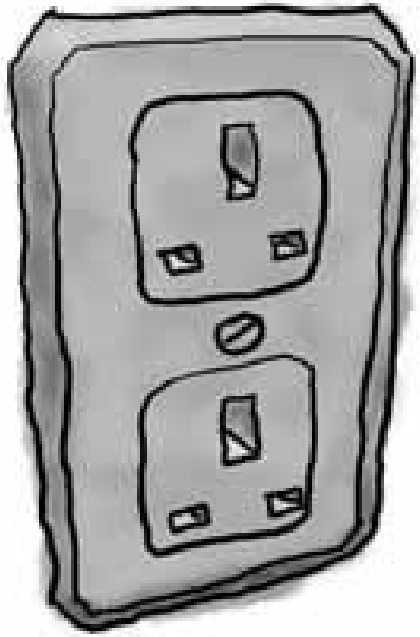
```
public interface Command {
   public void execute ();
   public void undo ();
}
```

- Now, every `Command` should be undoable
- So we add an implementation for `undo()` for every command that we implement
  - E.g., for `LightOnCommand`, `undo()` simply calls `light.off()`
  - Some `undo()` method implementations are more complicated; e.g., undoing a change in speed of a ceiling fan
- All that's left is to add support to the remote control to handle tracking which `undo()` method to call
  - Store the last command executed; if the undo button is pressed, we can just invoke the `undo()` method on that command

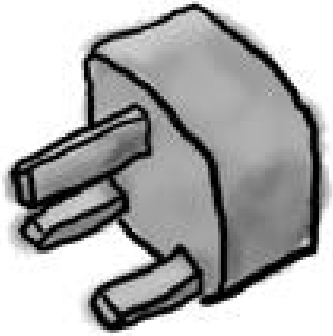
THE ADAPTER PATTERN

The Adapter Analogy

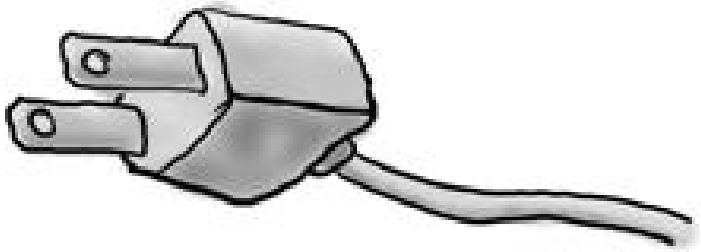
European Wall Outlet



AC Power Adapter



Standard AC Plug

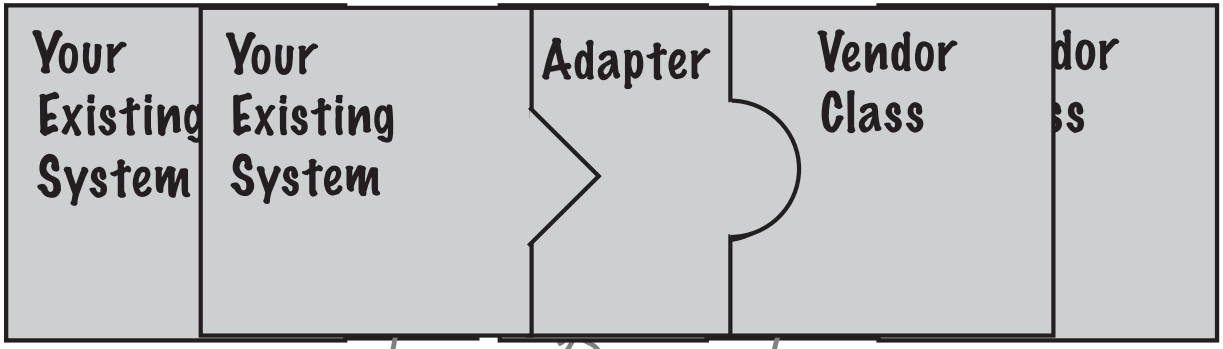


The US laptop expects another interface.

The European wall outlet exposes one interface for getting power.

The adapter converts one interface into another.

The OO Adapter



Writing an Adaptor

- Back to Ducks… but with interfaces this time:

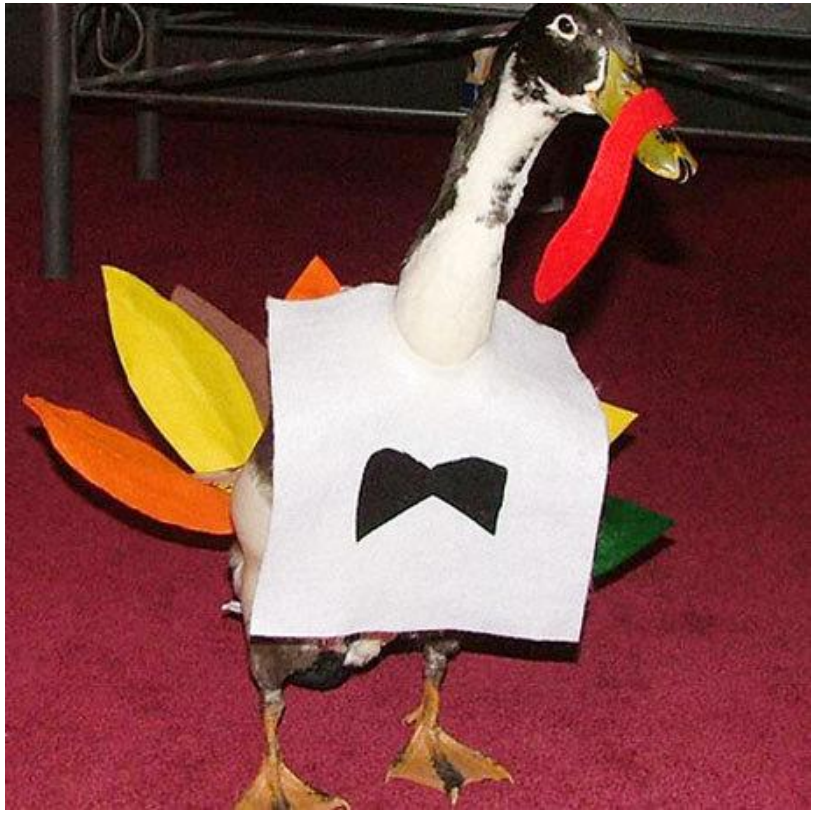
```
public interface Duck {
  public void quack();
  public void fly();
}
public class MallardDuck implements Duck {
  public void quack() {
    System.out.println("Quack");
  }
  public void fly() {
    System.out.println("I'm flying!");
  }
}
```

A New Fowl…

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

Now here's my problem

- I need turkeys. But all I have are ducks. Is there anyway to turn a duck into a turkey?



The TurkeyAdapter

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

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

We need to get a reference to the object that we're adapting; here we just do that through the constructor.

We have to implement all of the methods in the interface

Even though both interfaces have the fly() method, the turkey's fly is different than the duck's. So we have to implement that logic.

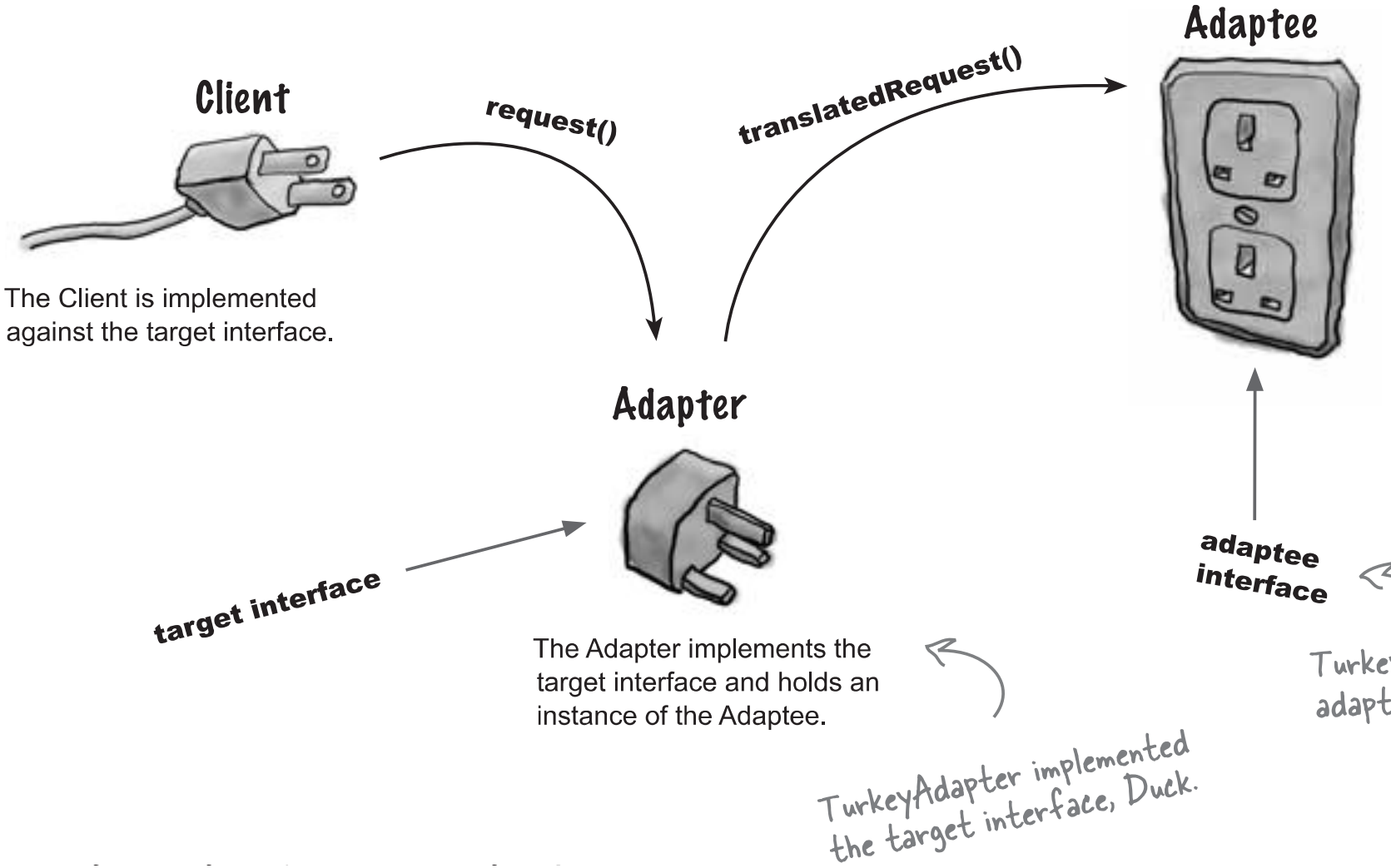
Let's test it

```
public class DuckTestDrive{
    public static void main(String[] args){
        WildTurkey turkey = new WildTurkey();
        Duck turkeyAdapter = new TurkeyAdapter(turkey);

        testDuck(turkeyAdapter);
    }

    static void testDuck(Duck duck){
        duck.quack();
        duck.fly();
    }
}
```

The Adapter Pieces



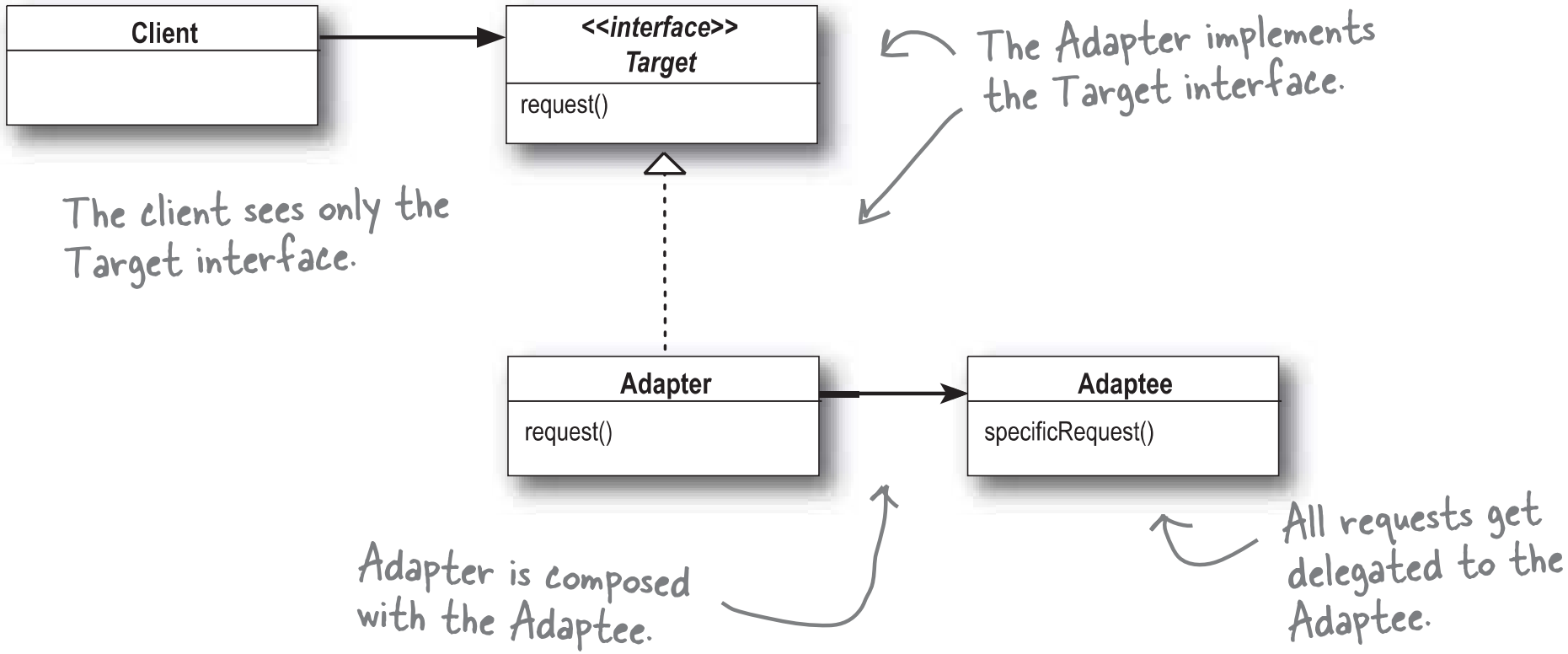
Questions

- How much adapting can be done in an adapter?
  - That really depends on the particular situation and the particular interfaces. It could be just basic translation or massive amounts of work
- Does an adapter always wrap only one class?
  - The real world can be messier; an adapter could wrap two or more adaptees needed to implement the target interface
  - However, the Adapter *always* converts one interface to another (the point is just that the definition of “interface” may not be limited to a single class)

The Adapter Pattern

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

The Adapter Class Diagram

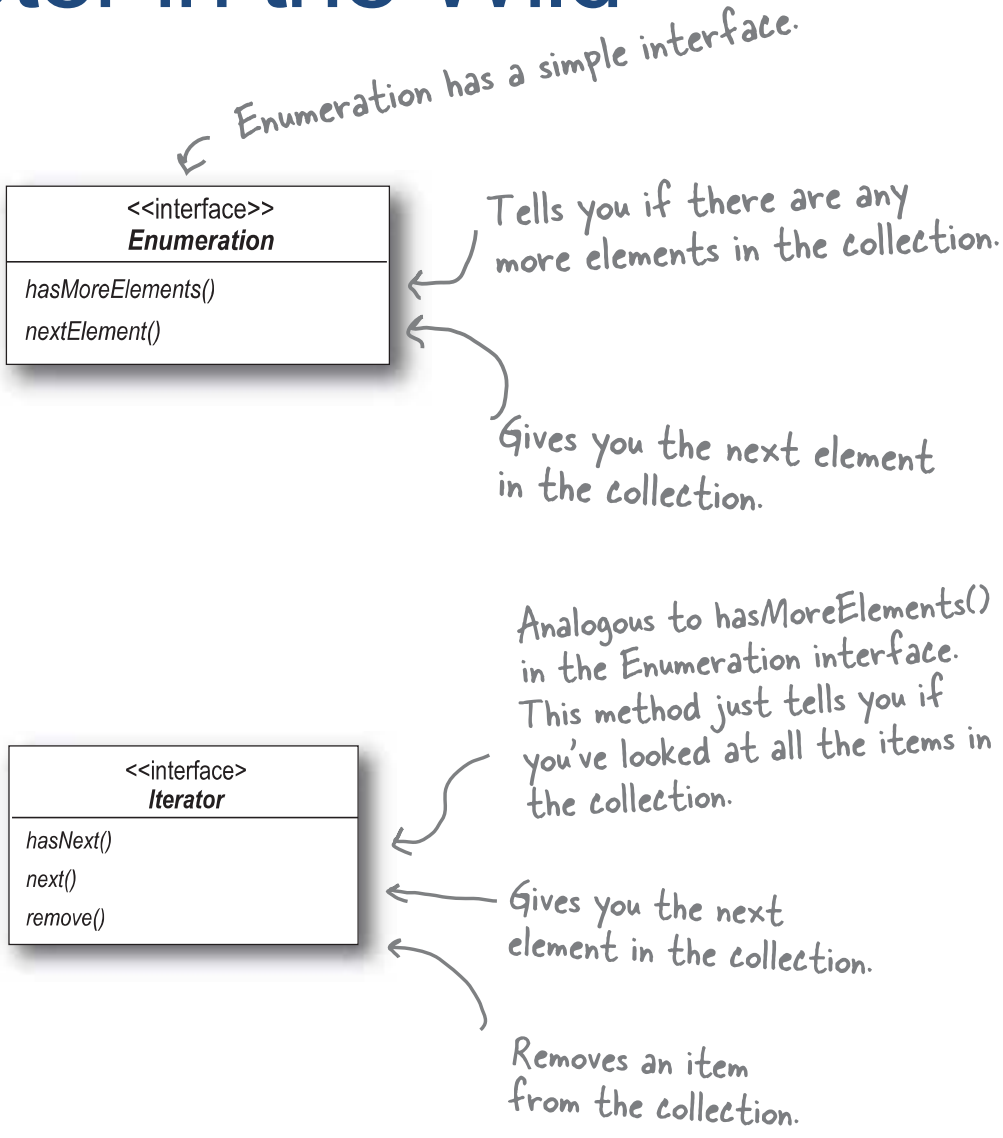


Adapter Pattern and Good OO Design

- Favor composition
  - The adaptee is wrapped with an altered interface using composition
- Programming to an interface not an implementation
  - The client is only aware of the target interface

An Example Adapter in the Wild

- Old World Enumerators…
  - Early Java collections types implemented an `elements()` method that returned an `Enumeration` of the elements that you could then step through without knowing how the collection was implemented
- New World Iterators…
  - The Collections classes use an `Iterator` interface that implements a similar capability but also allows item removal
- Never the twain shall meet
  - Oh, wait…

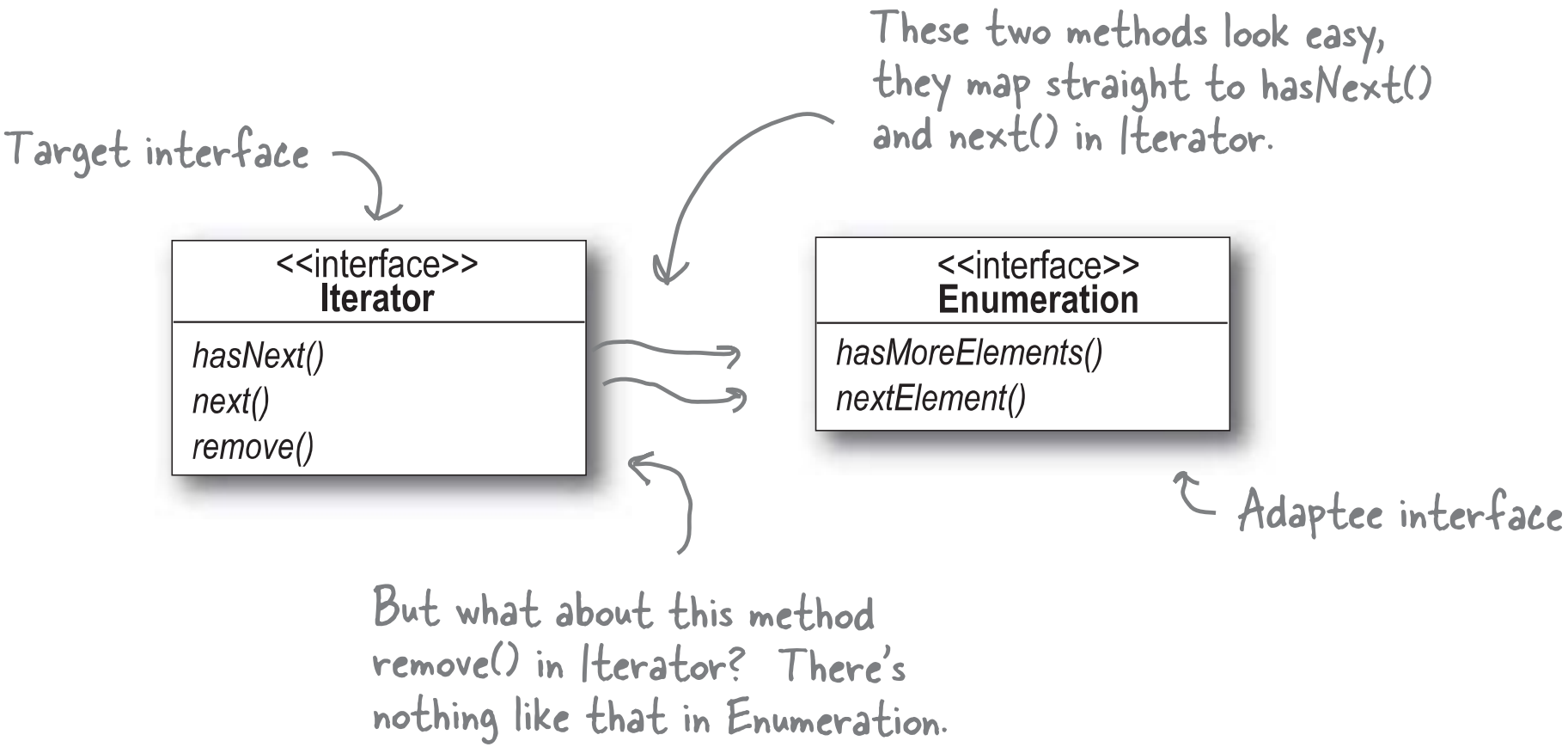


Backwards Compatibility

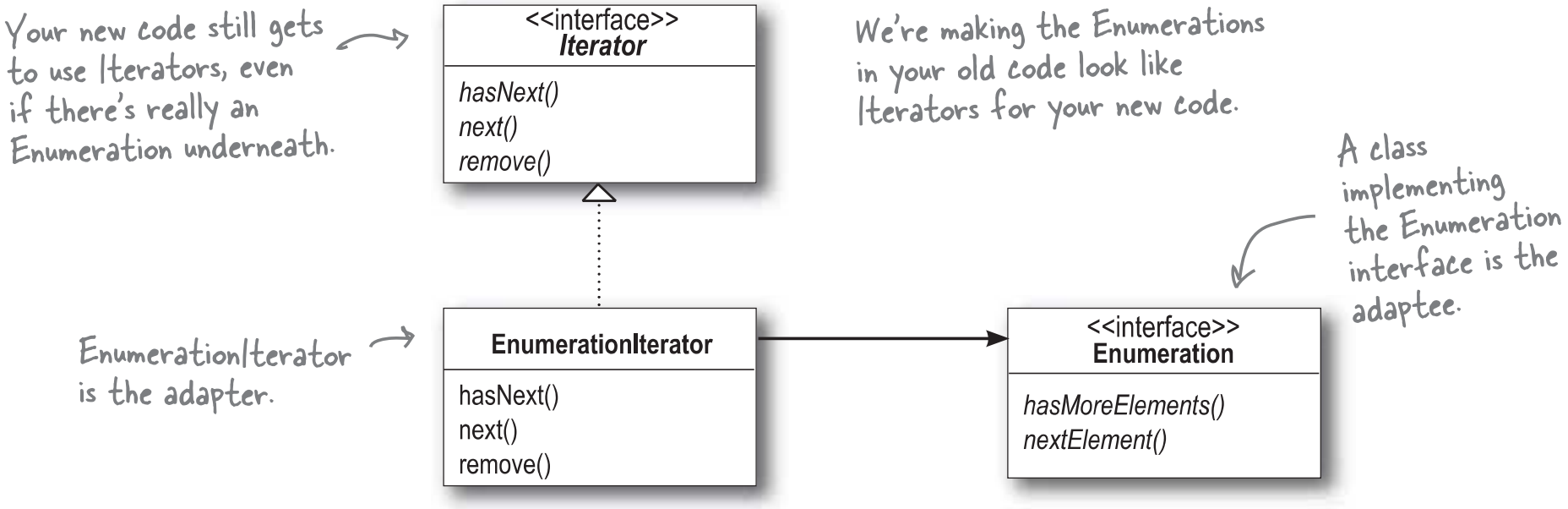
- You know what that means, right?

- We often have to work with legacy code that does it the old way.
  - But our clients insist (and they should) on using the new way.

Let's Examine the Interfaces



The Class Diagram



How to Handle the `remove()` method

- Enumeration simply doesn't support remove; it's "read only"
- We can, however, throw a runtime exception if someone tries to call `remove()` on an `EnumerationIterator`
  - The Iterator class supports this; it's remove method supports throwing an `UnsupportedOperationException`

- In the end, the adapter isn't perfect; clients will still have to deal with potential exceptions

The `EnumerationIterator` Adapter

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

Our adapter has to implement the Iterator interface

Use composition to stash the enum

The Iterator's `hasNext()` method is a direct map to the Enumeration's `hasMoreElements()` method

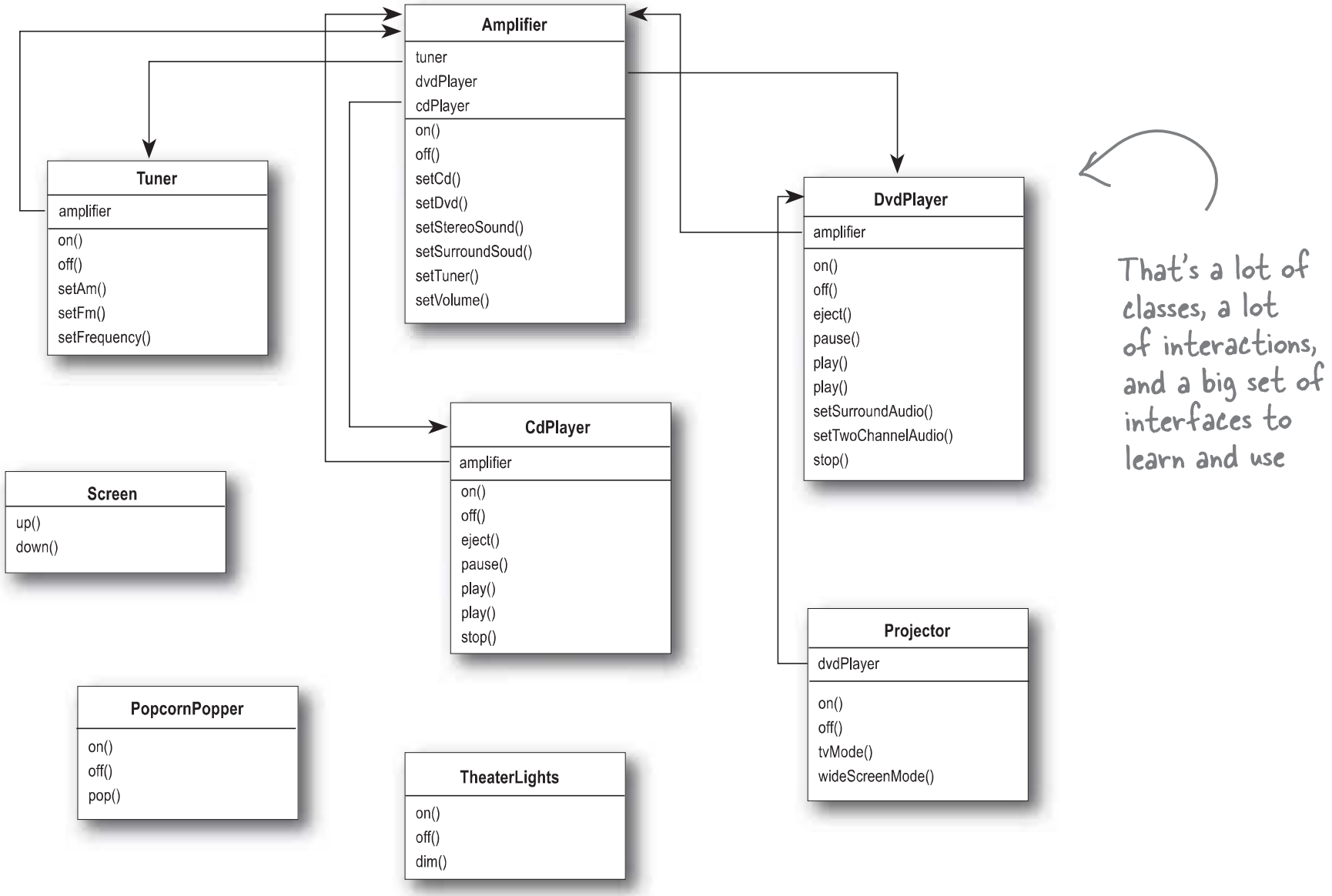
The Iterator's `next()` method is a direct map to the Enumeration's `nextElement()` method

Throw the `UnsupportedOerationException` if someone tries to call remove

THE FACADE PATTERN

A special kind of adapter…

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

Codes

- Popper.on();
- Popper.pop();
- Lights.dim(10);
- Screen.down();
- Projector.on();
- Projector.setInput(dvd);
- Projector.widScreenMode();
- Amp.on();
- Amp.setDvd(dvd);
- Amp.setSurroundSound();
- Amp.setVolume(5);
- Dvd.on();
- Dvd.play(movie);

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().

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

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 does not encapsulate; it just provides a simplified interface

The Home Theater Façade

```
public class HomeTheaterFacade {
    Amplifier amp;
    Tuner tuner;
    DvdPlayer dvd;
    CdPlayer cd;
    Projector projector;
    TheaterLights lights;
    Screen screen;
    PopcornPopper popper;

    public HomeTheaterFacade(Amplifier amp,
                             Tuner tuner,
                             DvdPlayer dvd,
                             CdPlayer cd,
                             Projector projector,
                             Screen screen,
                             TheaterLights lights,
                             PopcornPopper popper) {

        this.amp = amp;
        this.tuner = tuner;
        this.dvd = dvd;
        this.cd = cd;
        this.projector = projector;
        this.screen = screen;
        this.lights = lights;
        this.popper = popper;
    }

        // other methods here

}
```

Here's the composition; these are all the components of the subsystem we are going to use.

The facade is passed a reference to each component of the subsystem in its constructor. The facade then assigns each to the corresponding instance variable.

We're just about to fill these in...

The Home Theater Façade

```
public void watchMovie(String movie) {
    System.out.println("Get ready to watch a movie...");
    popper.on();
    popper.pop();
    lights.dim(10);
    screen.down();
    projector.on();
    projector.wideScreenMode();
    amp.on();
    amp.setDvd(dvd);
    amp.setSurroundSound();
    amp.setVolume(5);
    dvd.on();
    dvd.play(movie);
}

public void endMovie() {
    System.out.println("Shutting movie theater down...");
    popper.off();
    lights.on();
    screen.up();
    projector.off();
    amp.off();
    dvd.stop();
    dvd.eject();
    dvd.off();
}
```

watchMovie() follows the same sequence we had to do by hand before, but wraps it up in a handy method that does all the work. Notice that for each task we are delegating the responsibility to the corresponding component in the subsystem.

And endMovie() takes care of shutting everything down for us. Again, each task is delegated to the appropriate component in the subsystem.

One Remote to Rule them All

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.

A Sidebar: The Principle of Least Knowledge

- As a general design principle, you should reduce the interactions between objects to just a few “close friends”
  - Be careful of the number of classes an object interacts with and also how it comes to interact with those classes
  - Prevents creating designs that have a very high degree of coupling among classes (these systems are much more fragile)
- General guidelines:
  - Only invoke methods that belong to
    - The object itself
    - Objects passed as parameters to the method
    - Any object the method creates or instantiates
    - Any components of the object
  - Do **not** invoke methods on objects that were returned from calling other methods!

An Example

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.

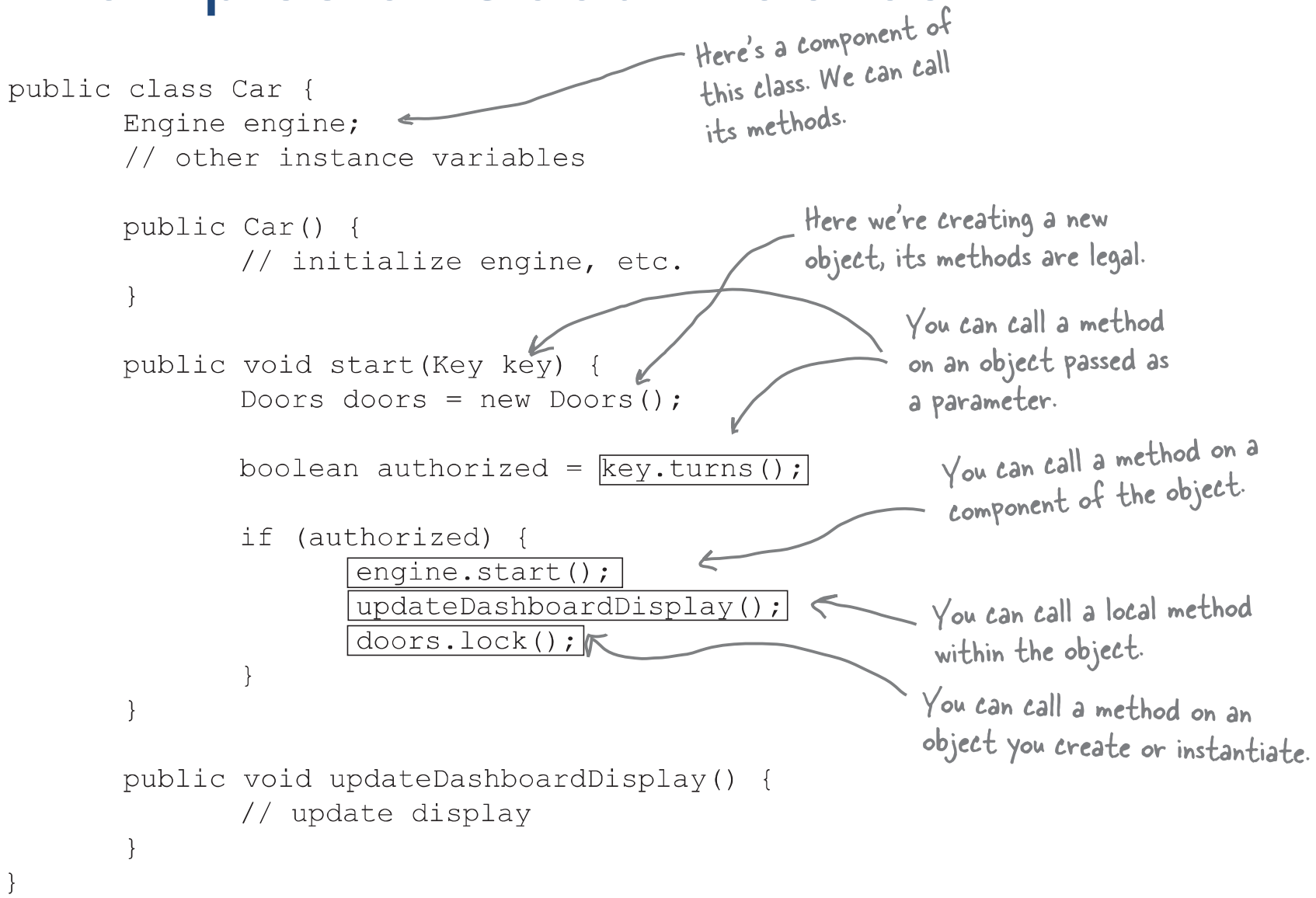
Principle of Least Knowledge

- a.k.a. Law of Demeter

- Advantages
  - Reduces dependencies between objects, which has been demonstrated to reduce maintenance costs

- Disadvantages
  - Results in more “wrapper” classes to avoid method calls to other components
  - This can result in increased complexity and development time

Examples of Good Practice




```
public class Car {
        Engine engine;
        // other instance variables

        public Car() {
                // initialize engine, etc.
        }

        public void start(Key key) {
                Doors doors = new Doors();

                boolean authorized = key.turns();

                if (authorized) {
                        engine.start();
                        updateDashboardDisplay();
                        doors.lock();
                }
        }

        public void updateDashboardDisplay() {
                // update display
        }
}
```

Exercise

- Does this violate the Principle of Least Knowledge?

```
public House {
    WeatherStation station;

    // other methods and constructor

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

Exercise

- Does this violate the Principle of Least Knowledge?

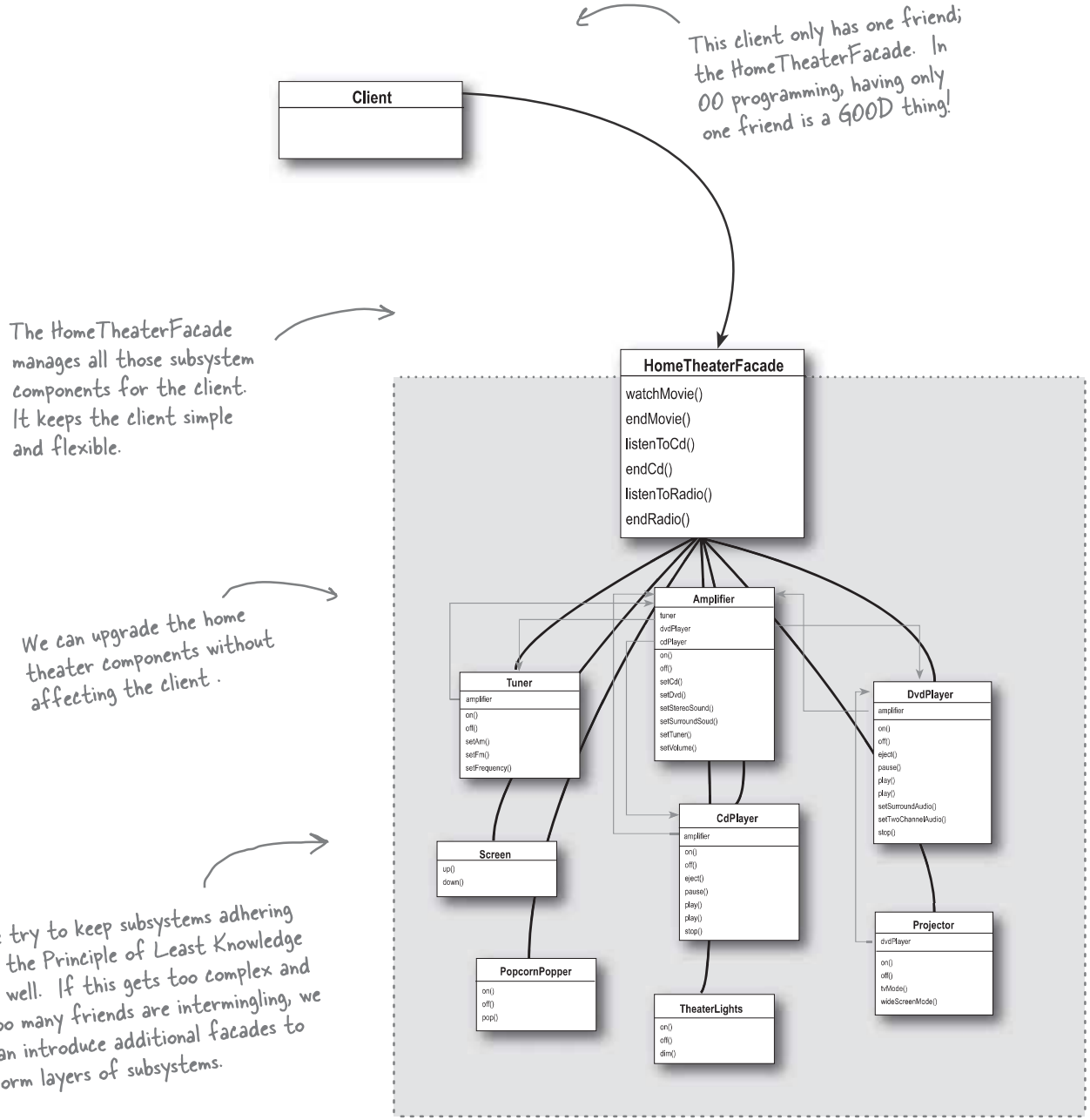
```
public House {
     WeatherStation station;

    // other methods and constructor

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

    public float getTempHelper(Thermometer thermometer) {
        return thermometer.getTemperature();
    }
}
```

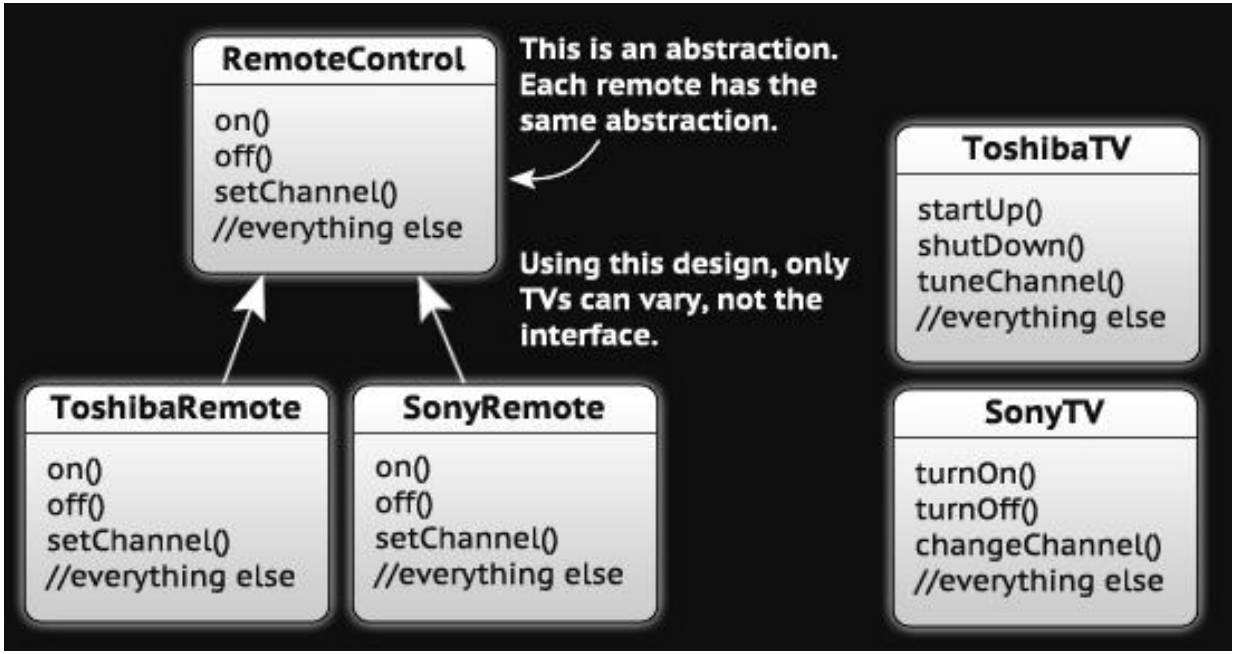
Façade and the Principle of Least Knowledge



BRIDGE PATTERN

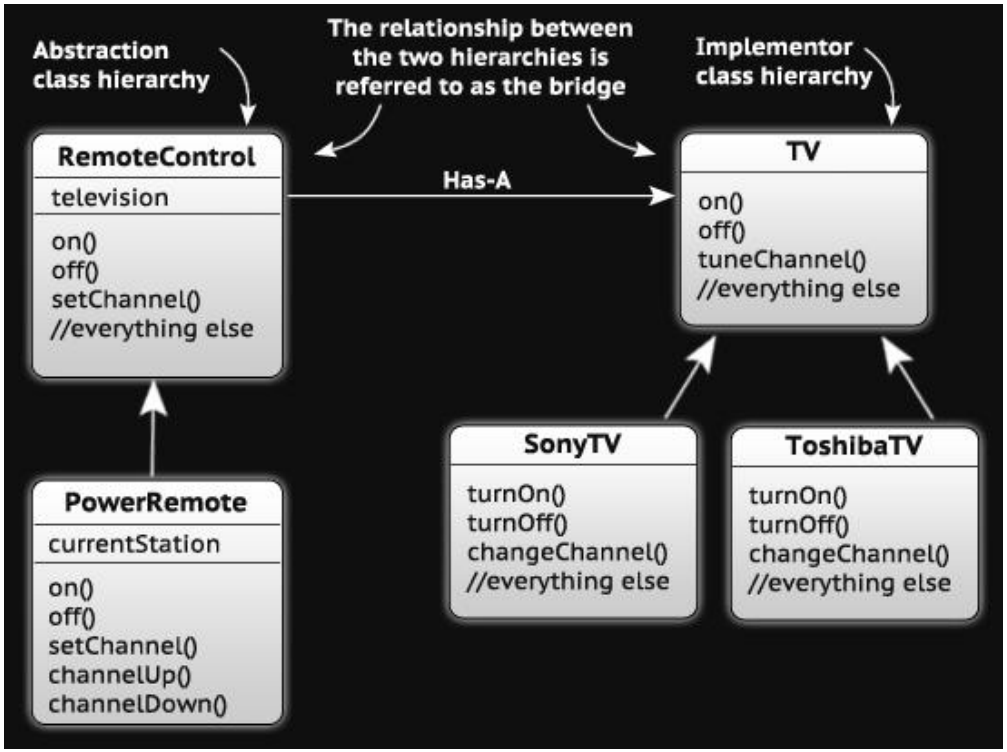
The Scenario

- You're writing code for a new remote control for TVs
- You're a good OO designer, so you'll make good use of abstraction
  - You'll have a RemoteControl base class and multiple implementations, one for each model of TV



The Dilemma

- The problem is that only the implementations can vary
- You need to make it possible for the abstraction to vary
- Do you remember “favor composition over inheritance”?
  - How can we employ that?



The Result

- Now you have two separate class hierarchies: one for the abstraction, and one for the implementation
  - The “has-a” relationship between the two is the “bridge”
- Benefits
  - Decouples an implementation so it is not permanently bound to an unchanging interface
  - Abstraction and implementation can be extended independently
  - Changes to the concrete abstraction classes don’t affect the client
- Uses
  - Useful in graphics and windowing systems that need to run over multiple platforms
  - Useful anytime you need to vary an interface and an implementation in different ways
- Disadvantages
  - Increases complexity

Some codes

```
public interface TV {
    void On();
    void Off();
    void tuneChannel();
}

public class SonyTV implements TV{
    @Override
    public void On() {
        System.out.println("Sony on”);
    }

    @Override
    public void Off() {
        System.out.println("Sony off");
    }

    @Override
    public void tuneChannel() {
        System.out.println("Sony tune channel");
    }
}
```

Some codes

```
public abstract class RemoteControl {
    public TV tv;
    public void On(){
        tv.On();
    }
    public void Off(){
        tv.Off();
    }
    public void SetChannel(){
        tv.tuneChannel();
    }
}

public class PowerRemote extends RemoteControl{
    @Override
    public void SetChannel() {
        System.out.println("reset the channel");
        super.SetChannel();
    }
}
```

Some codes

```
public class Client {
    public static void main(String[] args) {
        RemoteControl remoteControl = new PowerRemote();
        remoteControl.tv = new SonyTV();
        remoteControl.On();
        remoteControl.Off();
        remoteControl.SetChannel();
}
```

An Example

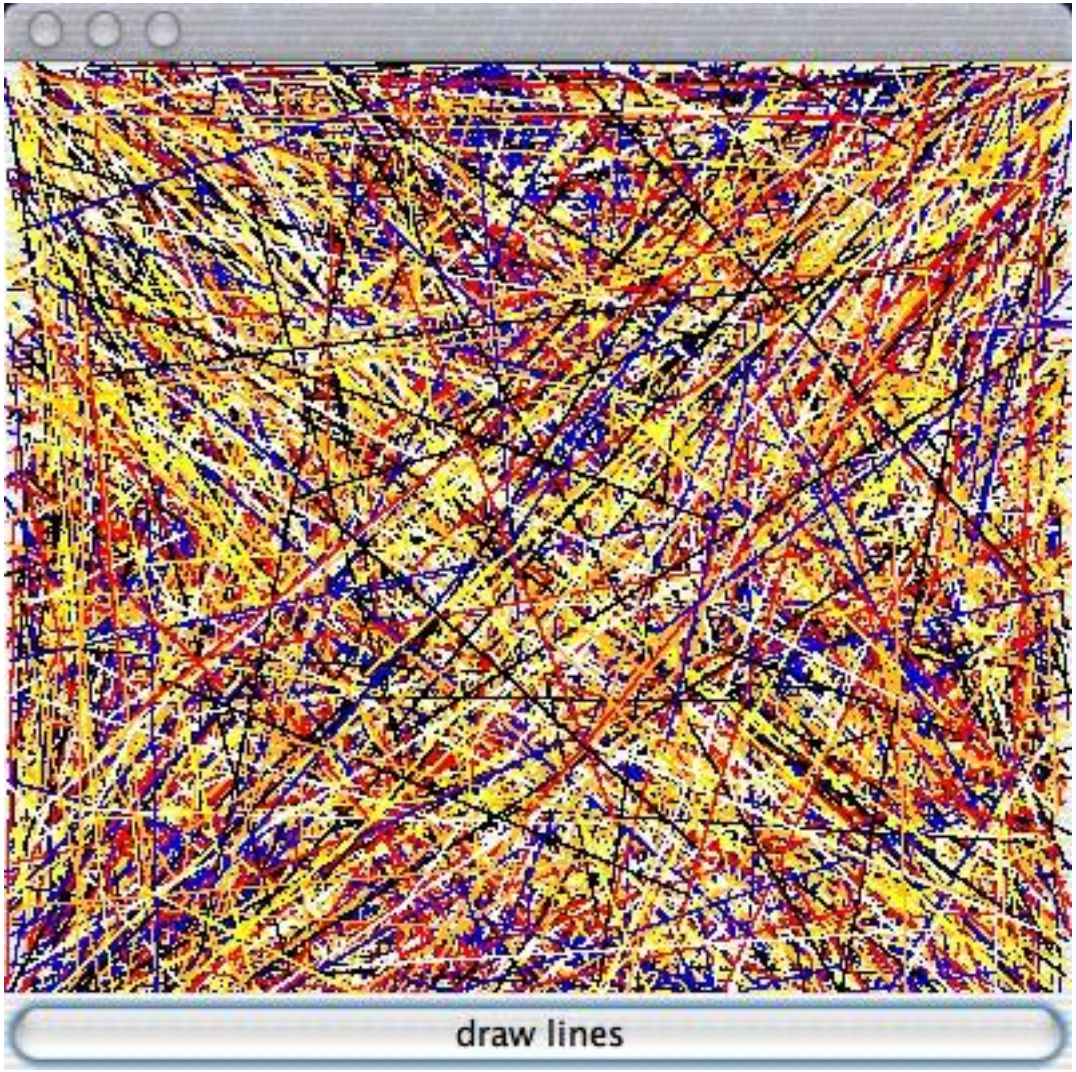
- http://cleanbugs.com/item/bridging-mode-for-java-design-patterns-413908.html

FLYWEIGHT PATTERN

The Scenario

- You want to add trees to a landscape design application
  - Trees are pretty dumb; they have x-y locations, they can draw themselves dynamically (depending on an age attribute)
  - A user might want to add lots and lots of trees
- A user of your application is complaining that, when he creates large groves of trees, the app gets sluggish
  - Yikes! There are THOUSANDS of tree objects!
- When you have tons of objects that are basically the same
  - Create a single instance of the object
  - A single client object that manages the state of all of those objects

An Example



An Example (cont.)

```
public class Test extends JFrame{
private static final Color colors[] = { Color.red, Color.blue,
                                        Color.yellow, Color.orange,
                                        Color.black,  Color.white };
   public Test() {
      // …
      button.addActionListener(new ActionListener() {
         public void actionPerformed(ActionEvent event) {
            Graphics g = panel.getGraphics();
            for(int i=0; i < NUMBER_OF_LINES; ++i) {
               g.setColor(getRandomColor());
               g.drawLine(getRandomX(), getRandomY(),
                          getRandomX(), getRandomY());
            }
         }
      });
   }
   private Color getRandomColor() {
      return colors[(int)(Math.random()*colors.length)];
   }
   // helper methods
}
```

An Example with Classes

```
public class Test extends JFrame{
   public Test() {
      //...
      button.addActionListener(new ActionListener() {
         public void actionPerformed(ActionEvent event) {
            Graphics g = panel.getGraphics();
            for(int i=0; i < NUMBER_OF_LINES; ++i) {
               Color color = getRandomColor();
               Line line = new Line(color,
                                    getRandomX(), getRandomY(),
                                    getRandomX(), getRandomY());
               line.draw(g);
            }
         }
      });
   }
   ...
}

public class Line {
   private Color color = Color.black;
   private int x, y, x2, y2;
   public Line(Color color, int x, int y, int x2, int y2) {
      this.color = color;  this.x = x;   this.y = y;  this.x2 = x2; this.y2 = y2;
   }
   public void draw(Graphics g) {
      g.setColor(color);
      g.drawLine(x, y, x2, y2);
   }
}
```

Yikes!

- We made 10,000 Line objects. That's insane
- Instead, let's group lines by some fundamental characteristic, say color
  - Now we only need to create six lines; then we can reuse the implementation…

Flyweight Lines

```
public class Test extends JFrame {
   //...
   public Test() {
      //...
      button.addActionListener(new ActionListener() {
         public void actionPerformed(ActionEvent event) {
            Graphics g = panel.getGraphics();
            for(int i=0; i < NUMBER_OF_LINES; ++i) {
               Line line = LineFactory.getLine(getRandomColor());
               line.draw(g, getRandomX(), getRandomY(),
                            getRandomX(), getRandomY());
            }
         }
      });
   }
   //...
}
```

Flyweight Lines (cont.)

```
public class LineFactory {
   private static final HashMap linesByColor = new HashMap();
   public static Line getLine(Color color) {
      Line line = (Line)linesByColor.get(color);
      if(line == null) {
         line = new Line(color);
         linesByColor.put(color, line);
         System.out.println("Creating " + color + " line");
      }
      return line;
   }
}

public class Line {
   private Color color;
   public Line(Color color) {
      this.color = color;
   }
   public void draw(Graphics g, int x, int y, int x2, int y2) {
      g.setColor(color);
      g.drawLine(x, y, x2, y2);
   }
}
```

QUESTIONS?

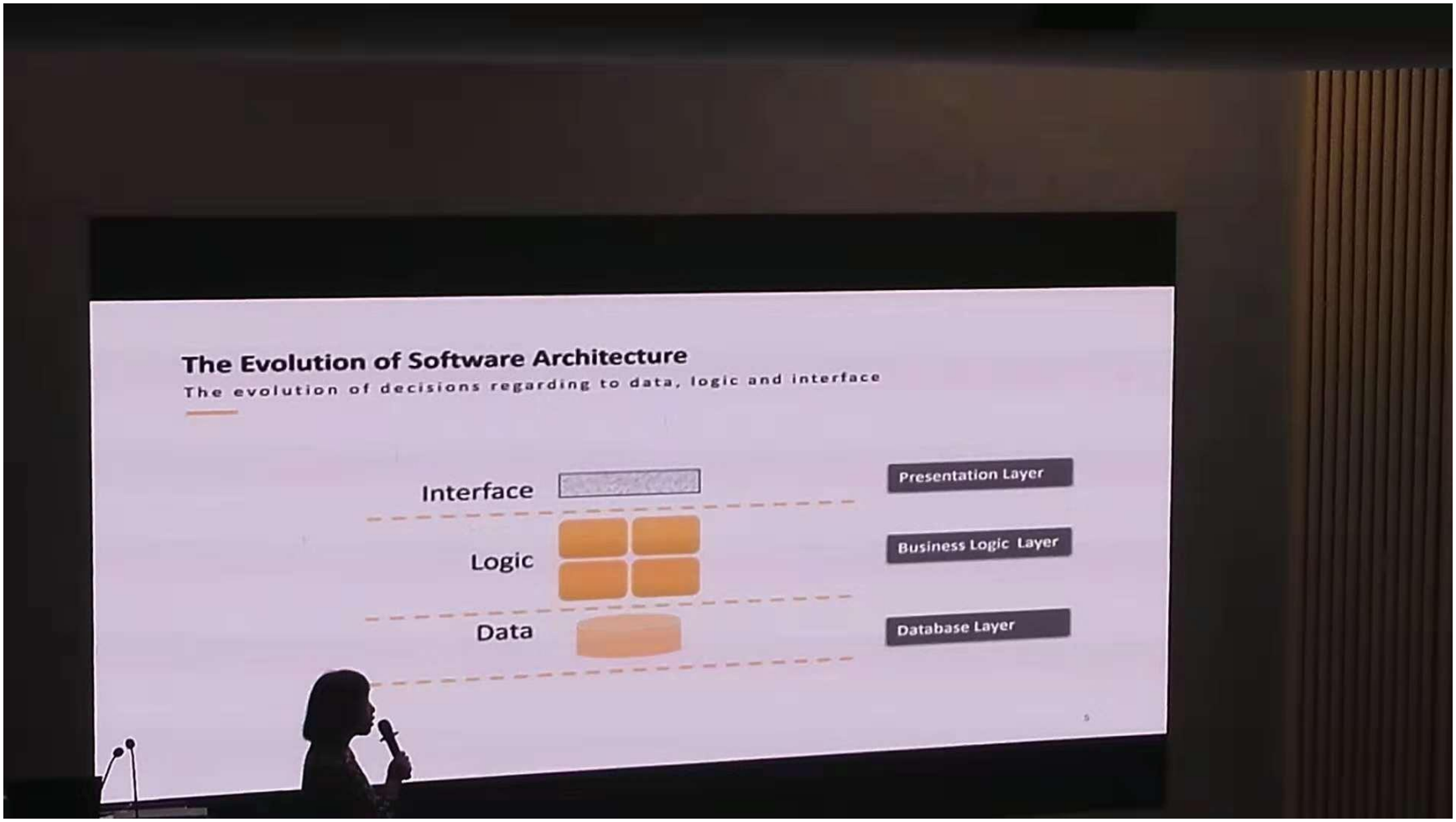
Summarization

- Creational patterns: dealing with object creations
- Structural patterns: easing the design by identifying simple ways to realize relationships among entities.
- Behavioral patterns: identifying communication patterns among entities

Principles

- encapsulate what varies
- favor composition over inheritence
- program to interface, not to implementation
- liskov substitute
- open for extensions, closed for modification
- depend upon abstractions, not upon implementations
- least knowledge

A little extras



A little extras