Lecture 07: The Command Pattern

SE313, Software Design and Architecture

Damla Oguz

Chapter 6: The Command Pattern

- “Sometimes it is necessary to issue requests to objects without knowing anything about the operation being requested or the receiver of the request.”
- In this lecture, we take encapsulation to a whole new level:
  - We’re going to encapsulate method invocation.
  - We’re going to encapsulate a “request” as an object.

A brief introduction to the Command Pattern

- Let’s study the interactions between the customers, the waitress, the orders and the short-order cook.

Let’s study the interaction in a little more detail

- The Customer knows what he wants and creates an order.
  - I’ll have a Burger with Cheese and a Malt Shake.

createOrder()

- The Order consists of an order slip and the customer’s menu items that are written on it.

takeOrder()

- The Waitress takes the Order, and when she gets around to it, she calls its orderUp() method to begin the Order’s preparation.

Let’s study the interaction in a little more detail (cont.)

orderUp()

- The Order has all the instructions needed to prepare the meal. The Order directs the Short Order Cook with methods like makeBurger().

makeBurger(), makeShake()

- The Short Order Cook follows the instructions of the Order and produces the meal.

output


ORDER
Burger with Cheese
Milk Shake
createOrder()
Start Here
takeOrder()
orderUp()
makeBurger(), makeShake()
output

The Objectville Diner roles and responsibilities

- An Order Slip encapsulates a request to prepare a meal.
  - Think of the Order Slip as an object, an object that acts as a request to prepare a meal.
  - Like any object, it can be passed around - from the Waitress to the order counter.
  - It has an interface that consists of only one method, orderUp(), that encapsulates the actions needed to prepare the meal.

The Objectville Diner roles and responsibilities (cont.)

- It also has a reference to the object that needs to prepare it (in our case, the Cook).
- It's encapsulated in that the Waitress doesn't have to know what's in the order or even who prepares the meal; she only needs to pass the slip through the order window and call "Order up!"

Okay, in real life a waitress would probably care what is on the Order Slip and who cooks it, but this is Objectville... work with us here!

The Objectville Diner roles and responsibilities (cont.)

- The Waitress's job is to take Order Slips and invoke the orderUp() method on them. The Waitress has it easy:
  - take an order from the customer,
  - continue helping customers until she makes it back to the order counter,
  - then invoke the orderUp() method to have the meal prepared.
  - As we've already discussed, in Objectville, the Waitress really isn't worried about what's on the order or who is going to prepare it; she just knows Order Slips have an orderUp() method she can call to get the job done.

The Objectville Diner roles and responsibilities (cont.)

- The Waitress' takeOrder() method gets parameterized with different Order Slips from different customers, but that doesn't faze her; she knows all Order Slips support the orderUp() method and she can call orderUp() any time she needs a meal prepared.

The Objectville Diner roles and responsibilities (cont.)

- The Short Order Cook has the knowledge required to prepare the meal.
  - The Short Order Cook is the object that really knows how to prepare meals.
  - Once the Waitress has invoked the orderUp() method; the Short Order Cook takes over and implements all the methods that are needed to create meals.

The Objectville Diner roles and responsibilities (cont.)

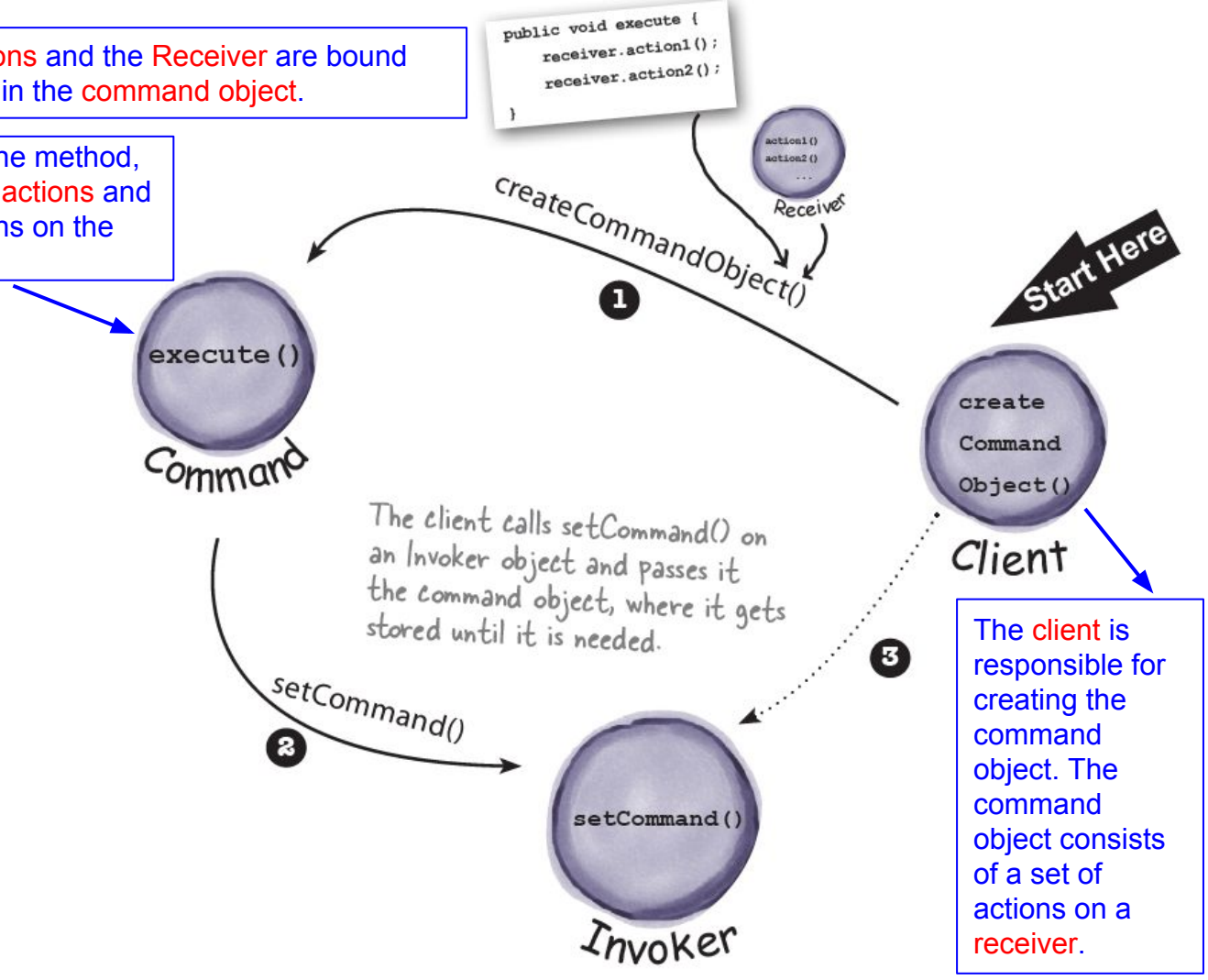
- Notice the Waitress and the Cook are totally **decoupled**:
  - The Waitress has Order Slips that encapsulate the details of the meal; she just calls a method on each order to get it prepared.
  - Likewise, the Cook gets his instructions from the Order Slip; he never needs to directly communicate with the Waitress.

The Objectville Diner roles and responsibilities (cont.)

- So, we have a Diner with a Waitress who is decoupled from the Cook by an Order Slip.
- Think of the Diner as a model for an OO design pattern that allows us **to separate “an object making a request” from “the objects that receive and execute those requests”.**
- Let’s map all this Diner talk to the Command Pattern…

The actions and the Receiver are bound together in the command object.

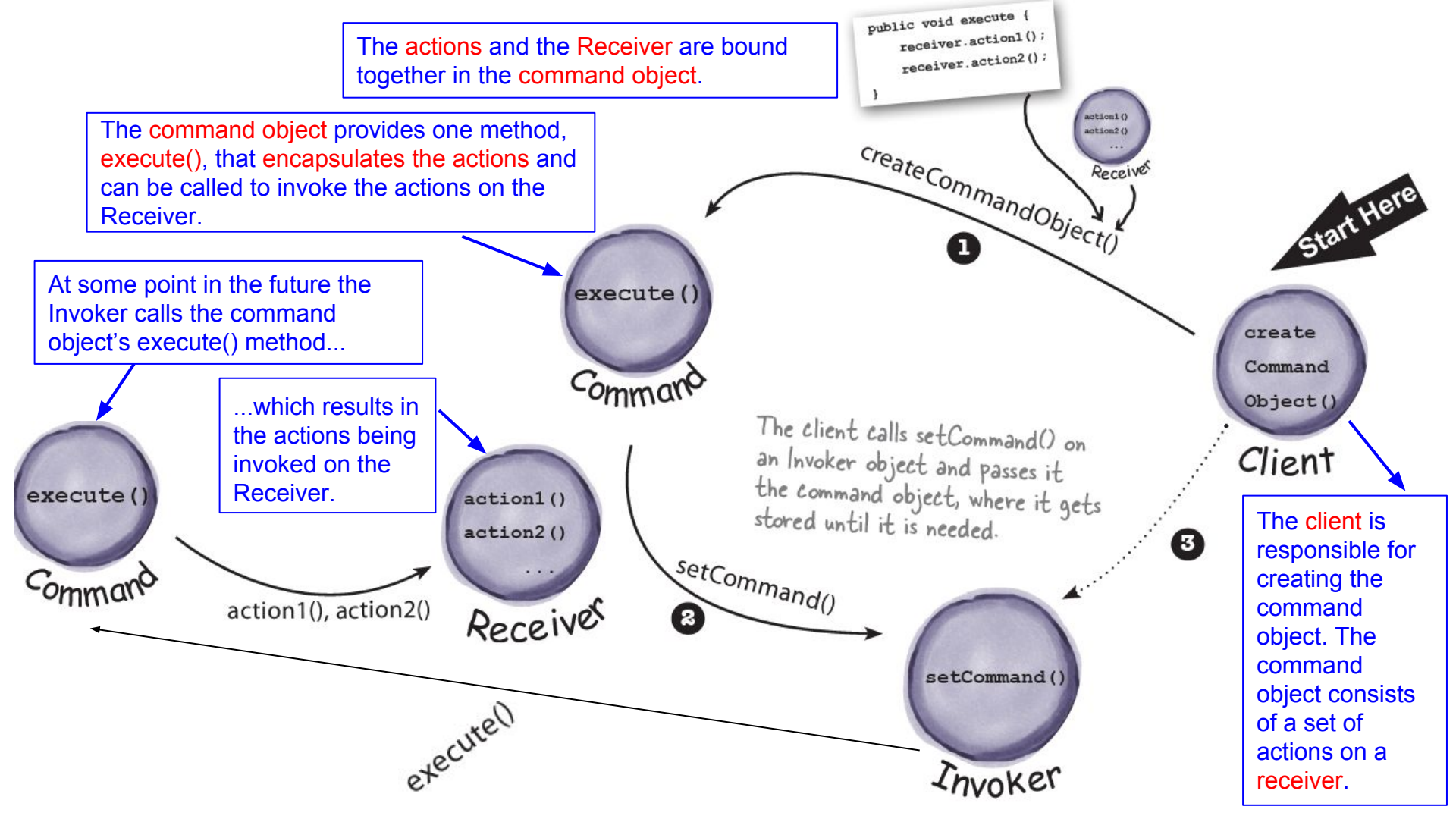
The command object provides one method, execute(), that encapsulates the actions and can be called to invoke the actions on the Receiver.



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.

The client is responsible for creating the command object. The command object consists of a set of actions on a receiver.


The actions and the Receiver are bound together in the command object.
public void execute {
receiver.action1();
receiver.action2();
}
The command object provides one method, execute(), that encapsulates the actions and can be called to invoke the actions on the Receiver.
action1()
action2()
...
Receiver
createCommandObject()
1
Start Here
execute()
Command
At some point in the future the Invoker calls the command object's execute() method...
create
Command
Object()
Client
...which results in the actions being invoked on the Receiver.
execute()
Command
action1()
action2()
...
Receiver
action1(), action2()
The client calls setCommand() on an Invoker object and passes it the command object, where it gets stored until it is needed.
setCommand()
2
3
setCommand()
Invoker
execute()
The client is responsible for creating the command object. The command object consists of a set of actions on a receiver.

From the Diner to the Command Pattern

- All the players are the same; only the names have changed.
- Match the diner objects and methods with the corresponding names from the Command Pattern.

○ Customer	● Invoker
○ createOrder()	● execute()
○ Order	● Client
○ takeOrder()	● createCommand()
○ Waitress	● Receiver
○ orderUp()	● Command
○ Short Order Cook	● setCommand()

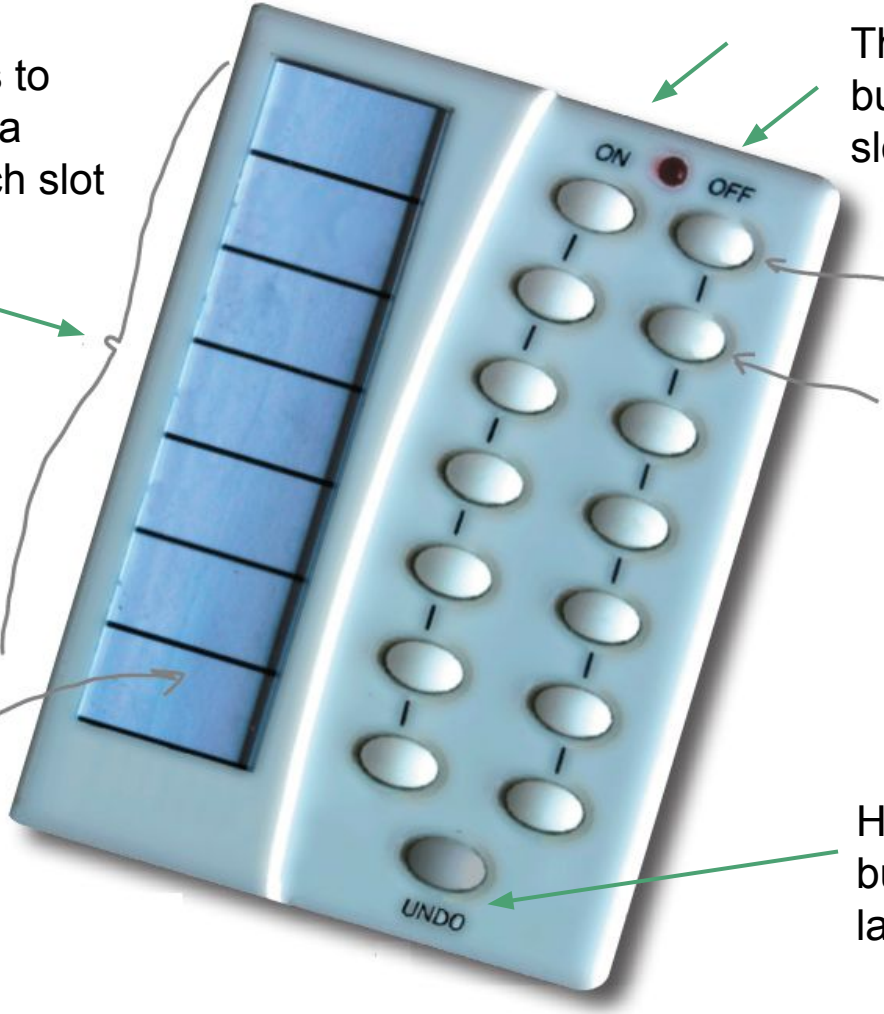
From the Diner to the Command Pattern

- All the players are the same; only the names have changed.
  - Customer --- Client
  - createOrder() --- createCommand()
  - Order --- Command
  - takeOrder() --- setCommand()
  - Waitress --- Invoker
  - orderUp() --- execute()
  - Short Order Cook --- Receiver

Home Automation Remote Control

- We will design the API for a new Home Automation Remote Control.
- Let’s check the hardware: the Remote Control...

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



There are "on" and "off" buttons for each of the seven slots.

These two buttons are used to control the household device stored in slot one...

... and these two control the household device stored in slot two…

... and so on.

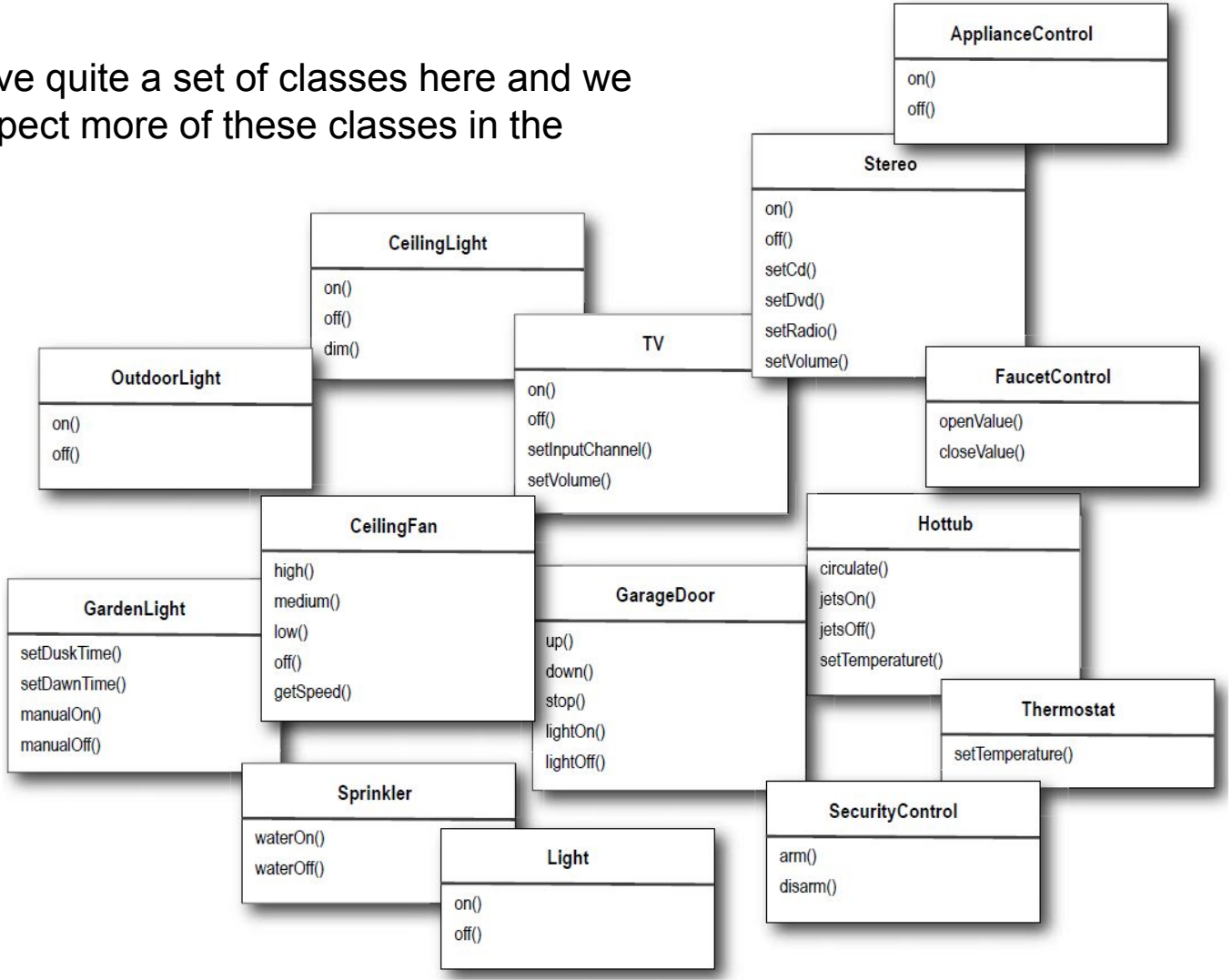
Write your device names here.

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

Taking a look at the vendor classes

- We will check out the vendor classes that are already exist.
- These should give us some idea of the interfaces of the objects we need to control from the remote.

We have quite a set of classes here and we can expect more of these classes in the future.



Discussing how to design the remote control API

- We have a bunch of classes with on() and off() methods, but here we've got methods like dim(), setTemperature(), setVolume(), setDirection().
- Also we can expect more vendor classes in the future with just as diverse methods.
- It's important we view this as a separation of concerns:
  - The remote should know how to interpret button presses and make requests.
  - But it shouldn't know a lot about home automation or how to turn on a hot tub.

Discussing how to design the remote control API (cont.)

- The Command Pattern allows us to decouple “the requester of an action” from “the object that actually performs the action”.
- So, here the requester would be the remote control and the object that performs the action would be an instance of one of our vendor classes.
  - We’re going to assign **each slot** to **a command** in the remote control.
  - This makes the **remote control** our **invoker**.

Discussing how to design the remote control API (cont.)

- A command object encapsulates a request to do something (like turn on a light) on a specific object (say, the living room light object).
- So, if we store a command object for each button, when the button is pressed we ask the command object to do some work.
- The remote doesn’t have any idea what the work is, it just has a command object that knows how to talk to the right object to get the work done.
- So, you see, the remote is decoupled from the light object!

Our first command object

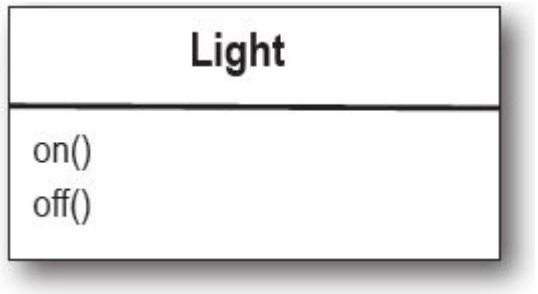
- **Implementing the Command interface**
  - First things first: all command objects implement the same interface, which consists of one method.

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

Simple. All we need is one method called execute().

Our first command object (cont.)

- **Implementing a command to turn a light on**
  - Let’s say you want to implement a command for turning a light on.
  - Referring to our set of vendor classes, the Light class has two methods: on() and off().



This is a command, so we need to implement the Command interface.

```
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 is going to control – say the living room light– and stashes it in the light instance variable. When execute gets called, this is the light object that is going to be the Receiver of the request.

The execute method calls the on() method on the receiving object, which is the light we are controlling.

Using the command object

- Let’s make things simple: say we’ve got a remote control with only one button and corresponding slot to hold a device to control…

```
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 our command, which will control one device.

We have a method for setting the command the slot is going to control. This could be called multiple times if the client of this code wanted to change the behavior of the remote 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.

Creating a simple test to use the Remote Control

```
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();
    }
}
```

This is our Client in Command Pattern-speak.

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

Now we create a Light object. This will be the Receiver of the request.

Here, create a command and pass the Receiver to it.

Here, pass the command to the Invoker.

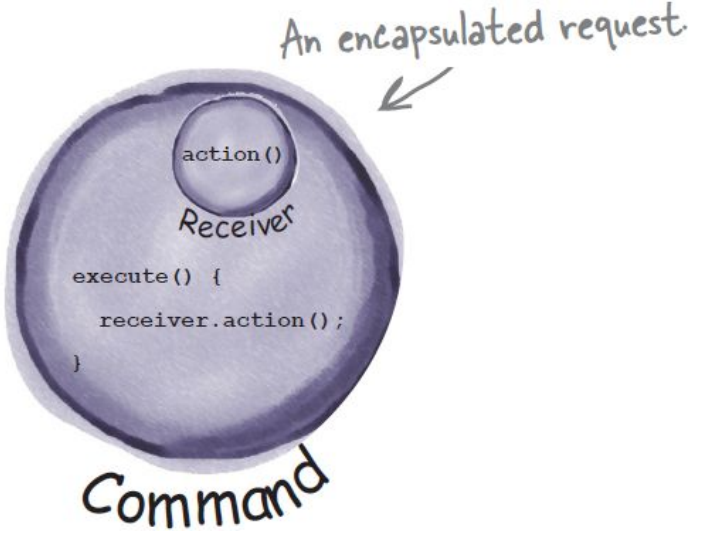
And then we simulate the button being pressed.

Here's the output of running this test code.

```
File  Edit  Window  Help  DinerFoodYum
%java RemoteControlTest

Light is On

%
```

The Command Pattern defined

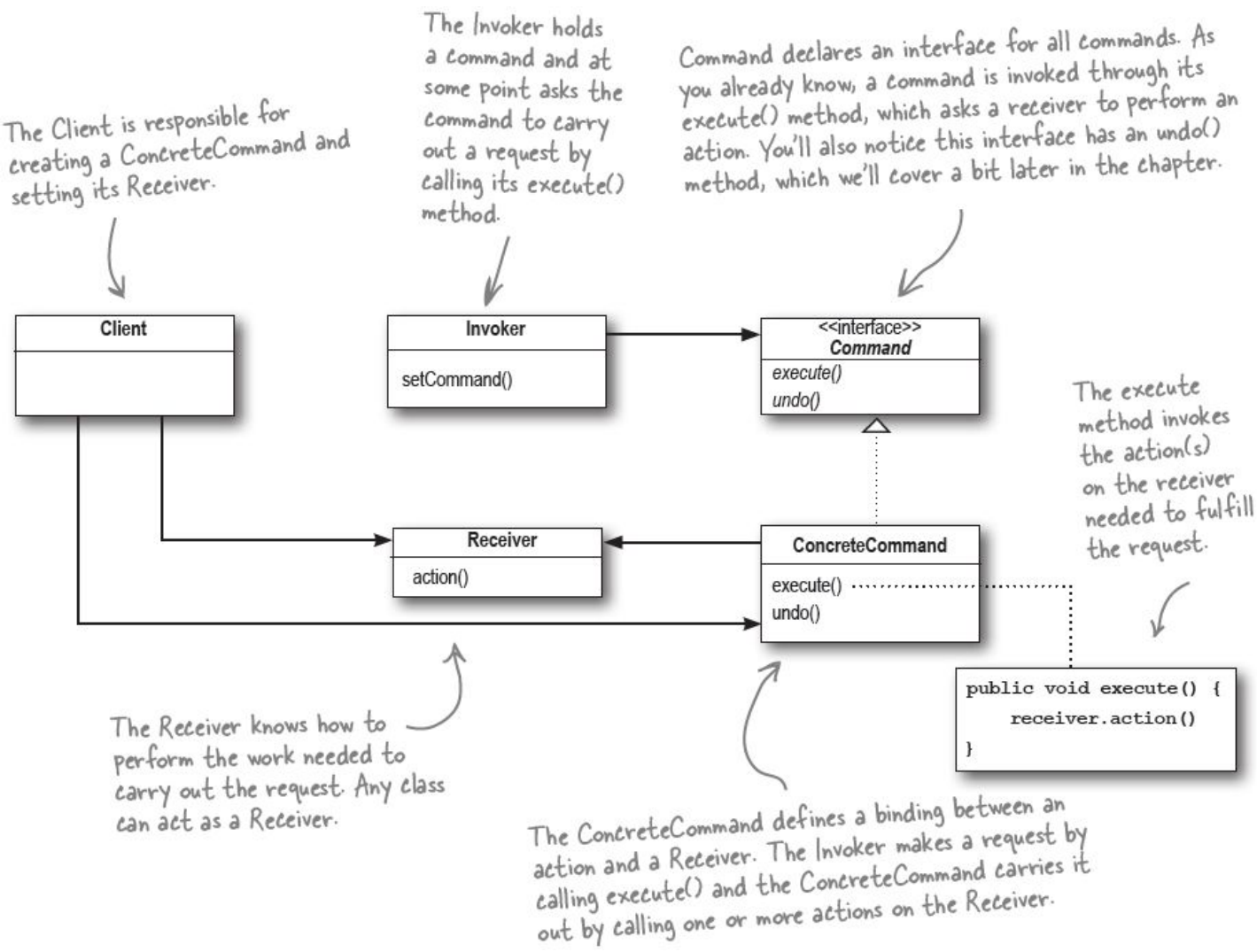
The Command Pattern encapsulates a request as an object, thereby letting you parameterize other objects with different requests, queue or log requests, and support undoable operations.



- We know that a command object ***encapsulates a request*** by binding together a set of actions on a specific receiver. To achieve this, it packages the actions and the receiver up into an object that exposes just one method, execute().

The Command Pattern defined (cont.)

- When called, execute() causes the actions to be invoked on the receiver.
- We've also seen some examples of ***parameterizing an object*** with a command.
  - Back at the diner, the Waitress can be parameterized with multiple orders throughout the day.
  - In the simple remote control, we can first load the button slot with a "light on" command and then later we can replace it with a "garage door open" command.
  - Like the Waitress, the remote slot does not care what command object it has, as long as it implements the Command interface.

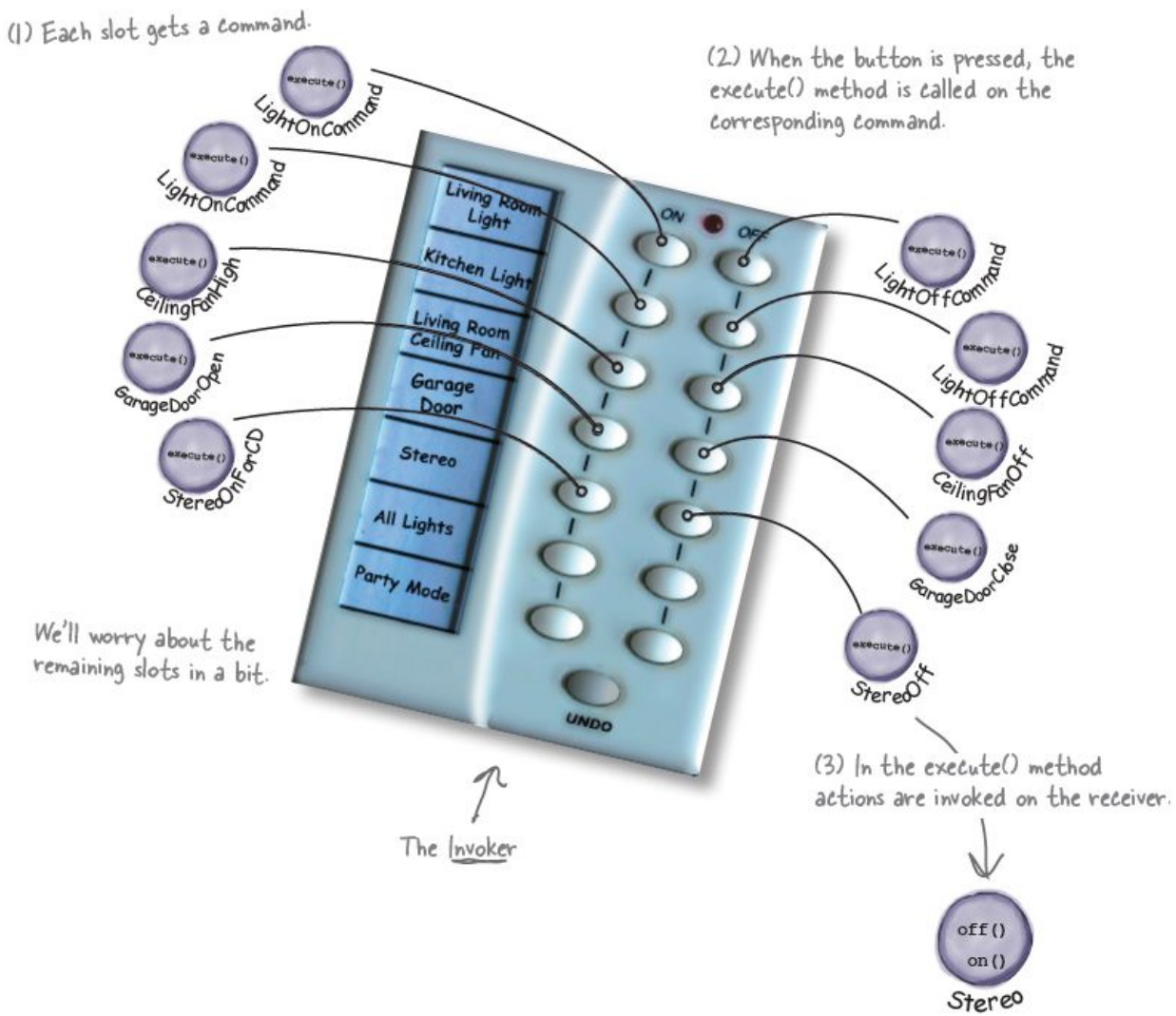

The Client is responsible for creating a ConcreteCommand and setting its Receiver.
The Invoker holds a command and at some point asks the command to carry out a request by calling its execute() method.
Command declares an interface for all commands. As you already know, a command is invoked through its 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.
Client
Invoker
setCommand()
<<interface>>
Command
execute()
undo()
Receiver
action()
ConcreteCommand
execute()
undo()
The execute method invokes the action(s) on the receiver needed to fulfill the request.
public void execute() {
receiver.action()
}
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.

Participants of the Command Pattern

- Command
  - declares an interface for executing an operation.
- ConcreteCommand
  - defines a binding between a Receiver object and an action.
  - implements execute() by invoking the corresponding operation(s) on Receiver.
- Client
  - creates a ConcreteCommand object and sets its receiver.
- Invoker
  - asks the command to carry out the request. (issues a request by calling execute() on the command.)
- Receiver
  - knows how to perform the operations associated with carrying out a request.

Assigning Commands to slots

- *We're going to assign **each slot** to **a command** in the remote control.*
- *This makes the **remote control** our **invoker**.*
- When a button is pressed the execute() method is going to be called on the corresponding command, which results in actions being invoked on the receiver (like lights, ceiling fans, and stereos).


(1) Each slot gets a command.
(2) When the button is pressed, the execute() method is called on the corresponding command.
execute()
LightOnCommand
execute()
LightOnCommand
execute()
CeilingFanHigh
execute()
GarageDoorOpen
execute()
StereoOnForCD
Living Room Light
Kitchen Light
Living Room Ceiling Fan
Garage Door
Stereo
All Lights
Party Mode
ON
OFF
UNDO
execute()
LightOffCommand
execute()
LightOffCommand
execute()
CeilingFanOff
execute()
GarageDoorClose
execute()
StereoOff
We'll worry about the remaining slots in a bit.
The Invoker
(3) In the execute() method actions are invoked on the receiver.
off()
on()
Stereo

Implementing the Remote Control

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

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

        Command noCommand = new NoCommand();
        for (int i = 0; i < 7; i++) {
            onCommands[i] = noCommand;
            offCommands[i] = noCommand;
        }
    }
```

This time around the remote is going to handle seven On and Off commands, which we'll hold in corresponding arrays.

In the constructor all we need to do is instantiate and initialize the on and off arrays.

Implementing the Remote Control (cont.)

The setCommand() method takes a slot position and an On and Off command to be stored in that slot.

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

It puts these commands in the on and off arrays for later use.

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

public void offButtonWasPushed(int slot) {
    offCommands[slot].execute();
}

public String toString() {
    StringBuffer stringBuff = new StringBuffer();
    stringBuff.append("\n------ Remote Control -------\n");
    for (int i = 0; i < onCommands.length; i++) {
        stringBuff.append("[slot " + i + "] " + onCommands[i].getClass().getName()
            + "    " + offCommands[i].getClass().getName() + "\n");
    }
    return stringBuff.toString();
}
```

When an On or Off button is pressed, the hardware takes care of calling the corresponding methods onButtonWasPushed() or offButtonWasPushed().

We've overwridden toString() to print out each slot and its corresponding command. You'll see us use this when we test the remote control.

Implementing the Commands

- We’ve already implemented the LightOnCommand for the SimpleRemoteControl.
- We can plug that same code in here and everything works beautifully.
- Off commands are no different; in fact, the LightOffCommand looks like this…

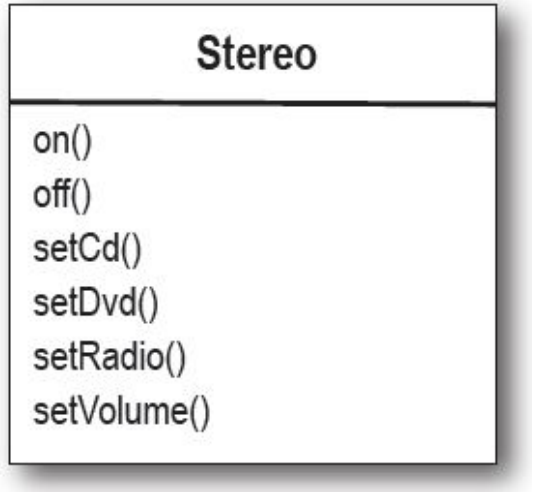
Implementing the Commands (cont.)

```
public class LightOffCommand implements Command {
    Light light;

    public LightOffCommand(Light light) {
        this.light = light;
    }

    public void execute() {
        light.off();
    }
}
```

The LightOffCommand works exactly the same way as the LightOnCommand, except that we are binding the receiver to a different action: the off() method.

Implementing the Commands

- Let’s try something a little more challenging; how about writing on and off commands for the Stereo?
- Okay, off is easy, we just bind the Stereo to the off() method in the StereoOffCommand.
- On is a little more complicated; let’s say we want to write a StereoOnWithCDCommand...



```
public class StereoOnWithCDCommand implements Command {
    Stereo stereo;

    public StereoOnWithCDCommand(Stereo stereo) {
        this.stereo = stereo;
    }

    public void execute() {
        stereo.on();
        stereo.setCD();
        stereo.setVolume(11);
    }
}
```

Just like the LightOnCommand, we get passed the instance of the stereo we are going to be controlling and we store it in a local instance variable.

To carry out this request, we need to call three methods on the stereo: first, turn it on, then set it to play the CD, and finally set the volume to 11. Why 11? Well, it's better than 10, right?

Putting the Remote Control through its paces

- Now we need to run some tests and get some documentation together to describe the API.
- We've managed to come up with a design that is going to allow them to produce a remote that is easy to maintain and they're going to have no trouble convincing the vendors to write some simple command classes in the future since they are so easy to write.

```
public class RemoteLoader {

    public static void main(String[] args) {
        RemoteControl remoteControl = new RemoteControl();

        Light livingRoomLight = new Light("Living Room");
        Light kitchenLight = new Light("Kitchen");
        CeilingFan ceilingFan= new CeilingFan("Living Room");
        GarageDoor garageDoor = new GarageDoor("");
        Stereo stereo = new Stereo("Living Room");

        LightOnCommand livingRoomLightOn =
                new LightOnCommand(livingRoomLight);
        LightOffCommand livingRoomLightOff =
                new LightOffCommand(livingRoomLight);
        LightOnCommand kitchenLightOn =
                new LightOnCommand(kitchenLight);
        LightOffCommand kitchenLightOff =
                new LightOffCommand(kitchenLight);
```

Create all the devices in their proper locations.

Create all the Light Command objects.

```
CeilingFanOnCommand ceilingFanOn =
        new CeilingFanOnCommand(ceilingFan);
CeilingFanOffCommand ceilingFanOff =
        new CeilingFanOffCommand(ceilingFan);

GarageDoorUpCommand garageDoorUp =
        new GarageDoorUpCommand(garageDoor);
GarageDoorDownCommand garageDoorDown =
        new GarageDoorDownCommand(garageDoor);

StereoOnWithCDCommand stereoOnWithCD =
        new StereoOnWithCDCommand(stereo);
StereoOffCommand  stereoOff =
        new StereoOffCommand(stereo);
```

Create the On and Off for the ceiling fan.

Create the Up and Down commands for the Garage.

Create the stereo On and Off commands.

```
        remoteControl.setCommand(0, livingRoomLightOn, livingRoomLightOff);
        remoteControl.setCommand(1, kitchenLightOn, kitchenLightOff);
        remoteControl.setCommand(2, ceilingFanOn, ceilingFanOff);
        remoteControl.setCommand(3, stereoOnWithCD, stereoOff);

        System.out.println(remoteControl);

        remoteControl.onButtonWasPushed(0);
        remoteControl.offButtonWasPushed(0);
        remoteControl.onButtonWasPushed(1);
        remoteControl.offButtonWasPushed(1);
        remoteControl.onButtonWasPushed(2);
        remoteControl.offButtonWasPushed(2);
        remoteControl.onButtonWasPushed(3);
        remoteControl.offButtonWasPushed(3);
    }
}
```

Now that we've got all our commands, we can load them into the remote slots.

Here's where we use our toString() method to print each remote slot and the command that it is assigned to.

All right, we are ready to roll! Now, we step through each slot and push its On and Off button.

Let's check out the execution of our remote control test...

File Edit Window Help CommandsGetThingsDone

```
% java RemoteLoader
------ Remote Control -------
[slot 0] LightOnCommand             LightOffCommand
[slot 1] LightOnCommand             LightOffCommand
[slot 2] CeilingFanOnCommand        CeilingFanOffCommand
[slot 3] StereoOnWithCDCommand      StereoOffCommand
[slot 4] NoCommand                  NoCommand
[slot 5] NoCommand                  NoCommand
[slot 6] NoCommand                  NoCommand
```

On slots

Off slots

```
Living Room light is on
Living Room light is off
Kitchen light is on
Kitchen light is off
Living Room ceiling fan is on high
Living Room ceiling fan is off
Living Room stereo is on
Living Room stereo is set for CD input
Living Room Stereo volume set to 11
Living Room stereo is off
%
```

Our commands in action! Remember, the output from each device comes from the vendor classes. For instance, when a light object is turned on it prints "Living Room light is on."

A question


Wait a second, what is with that NoCommand that is loaded in slots four through six? Trying to pull a fast one?

Null object

- Good catch. We did sneak a little something in there. In the remote control, we didn't want to check to see if a command was loaded every time we referenced a slot. For instance, in the onButtonWasPushed() method, we would need code like this:

```
public void onButtonWasPushed(int slot) {
    if (onCommands[slot] != null) {
        onCommands[slot].execute();
    }
}
```

Null object (cont.)

- So, how do we get around that? Implement a command that does nothing!

```
public class NoCommand implements Command {
    public void execute() { }
}
```

- Then, in our RemoteControl constructor, we assign every slot a NoCommand object by default and we know we'll always have some command to call in each slot.

```
Command noCommand = new NoCommand();
for (int i = 0; i < 7; i++) {
    onCommands[i] = noCommand;
    offCommands[i] = noCommand;
}
```

Null object (cont.)

Pattern Honorable Mention

The NoCommand object is an example of a *null object*. A null object is useful when you don't have a meaningful object to return, and yet you want to remove the responsibility for handling **null** from the client. For instance, in our remote control we didn't have a meaningful object to assign to each slot out of the box, so we provided a NoCommand object that acts as a surrogate and does nothing when its execute method is called.

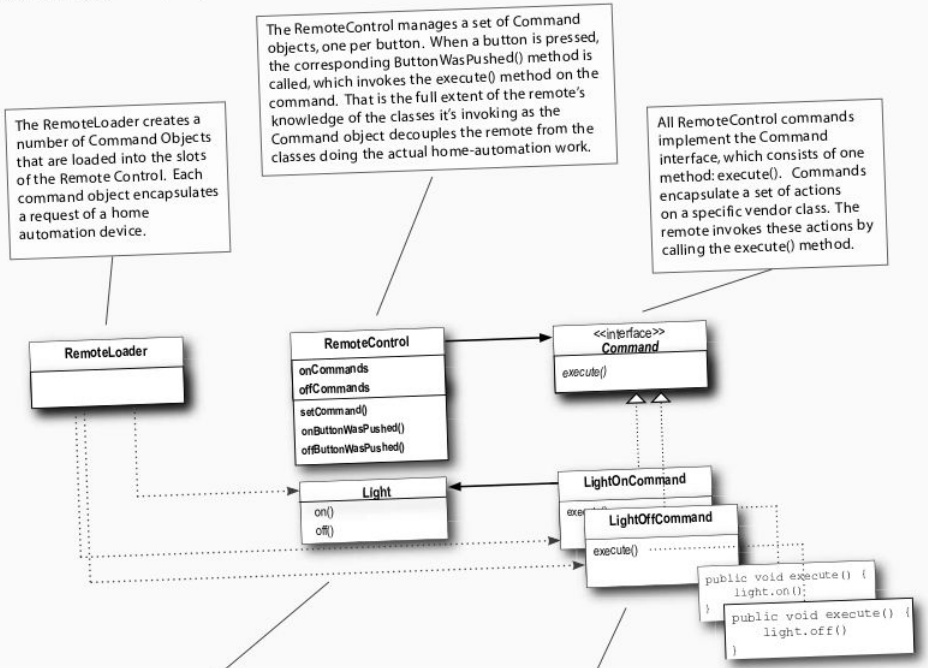
You'll find uses for Null Objects in conjunction with many Design Patterns and sometimes you'll even see Null Object listed as a Design Pattern.

Documentation

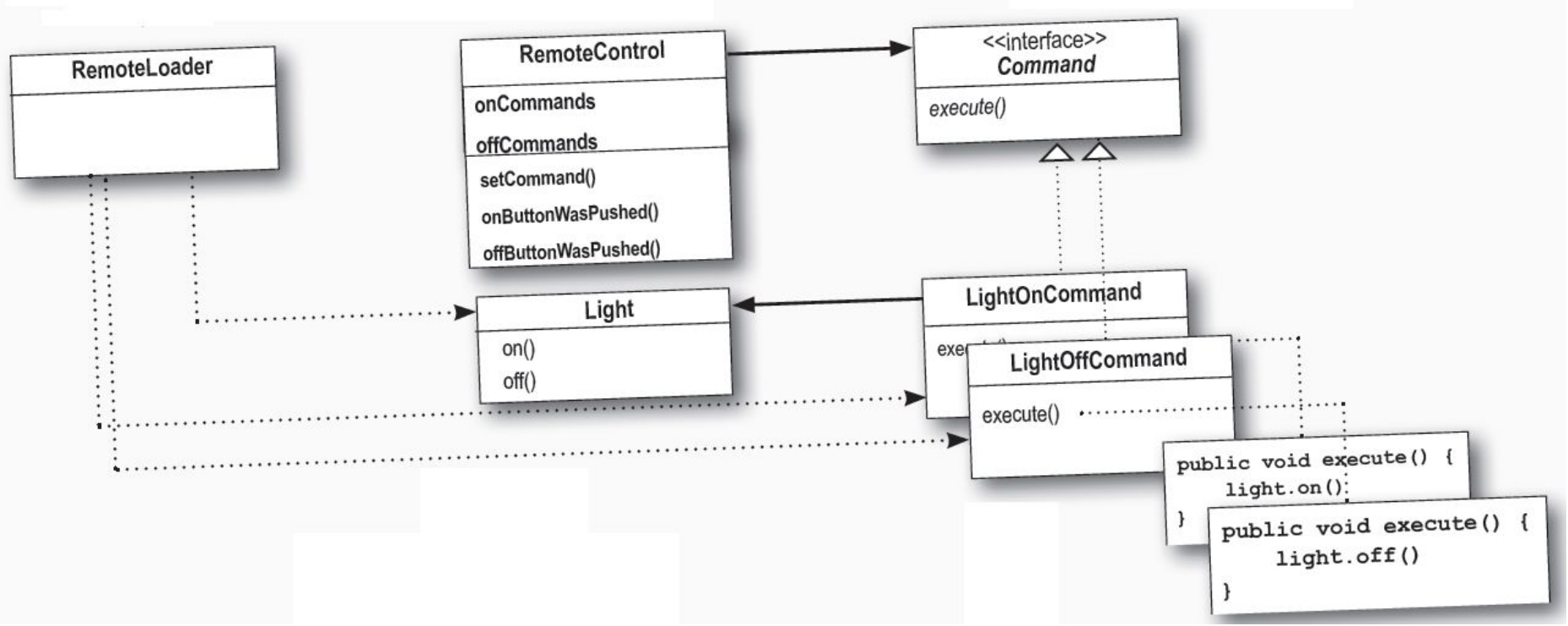
Remote Control API Design for Home Automation or Bust, Inc.,

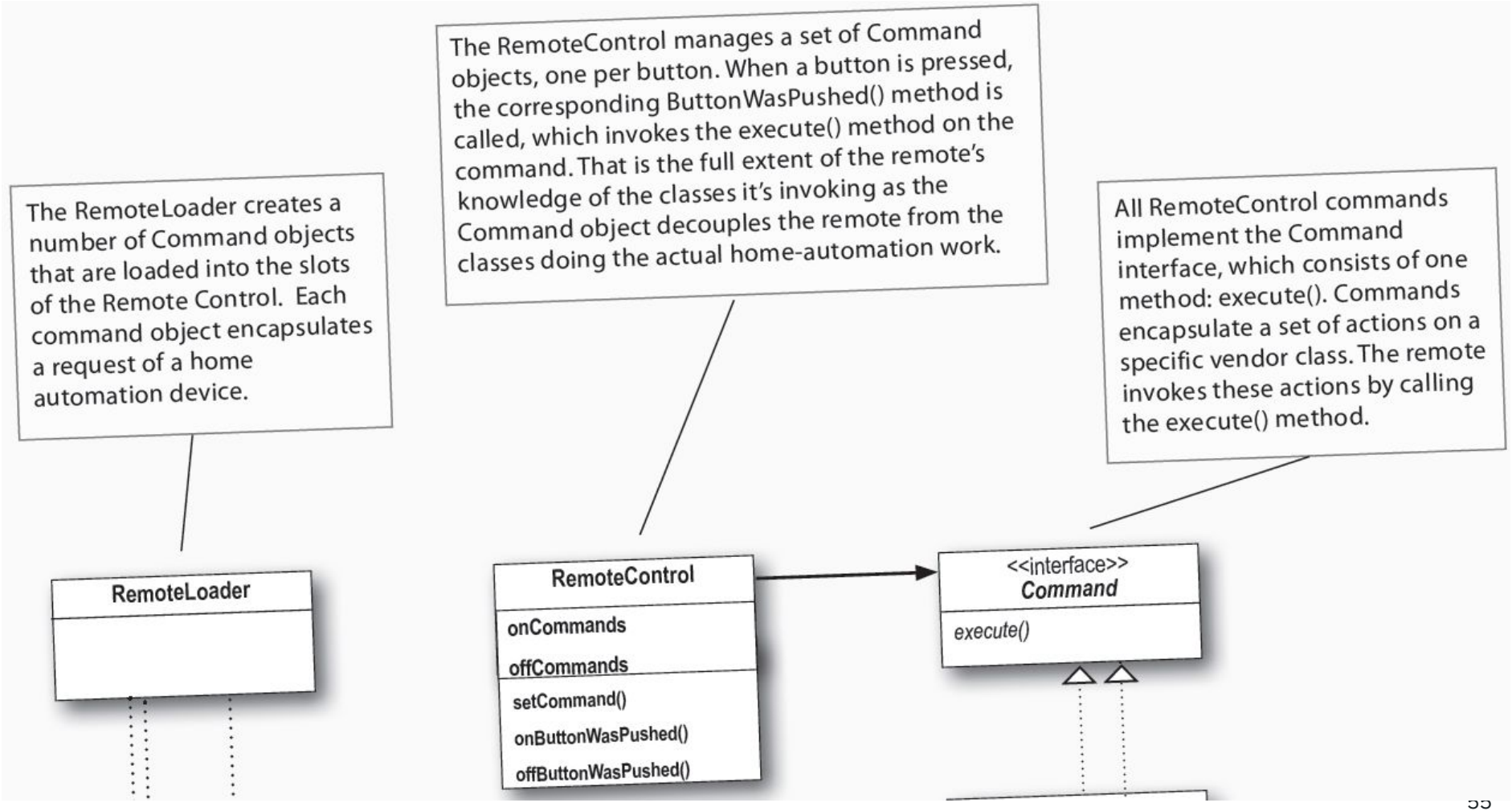
We are pleased to present you with the following design and application programming interface for your Home Automation Remote Control. Our primary design goal was to keep the remote control code as simple as possible so that it doesn't require changes as new vendor classes are produced. To this end we have employed the Command Pattern to logically decouple the RemoteControl class from the Vendor Classes. We believe this will reduce the cost of producing the remote as well as drastically reduce your ongoing maintenance costs.

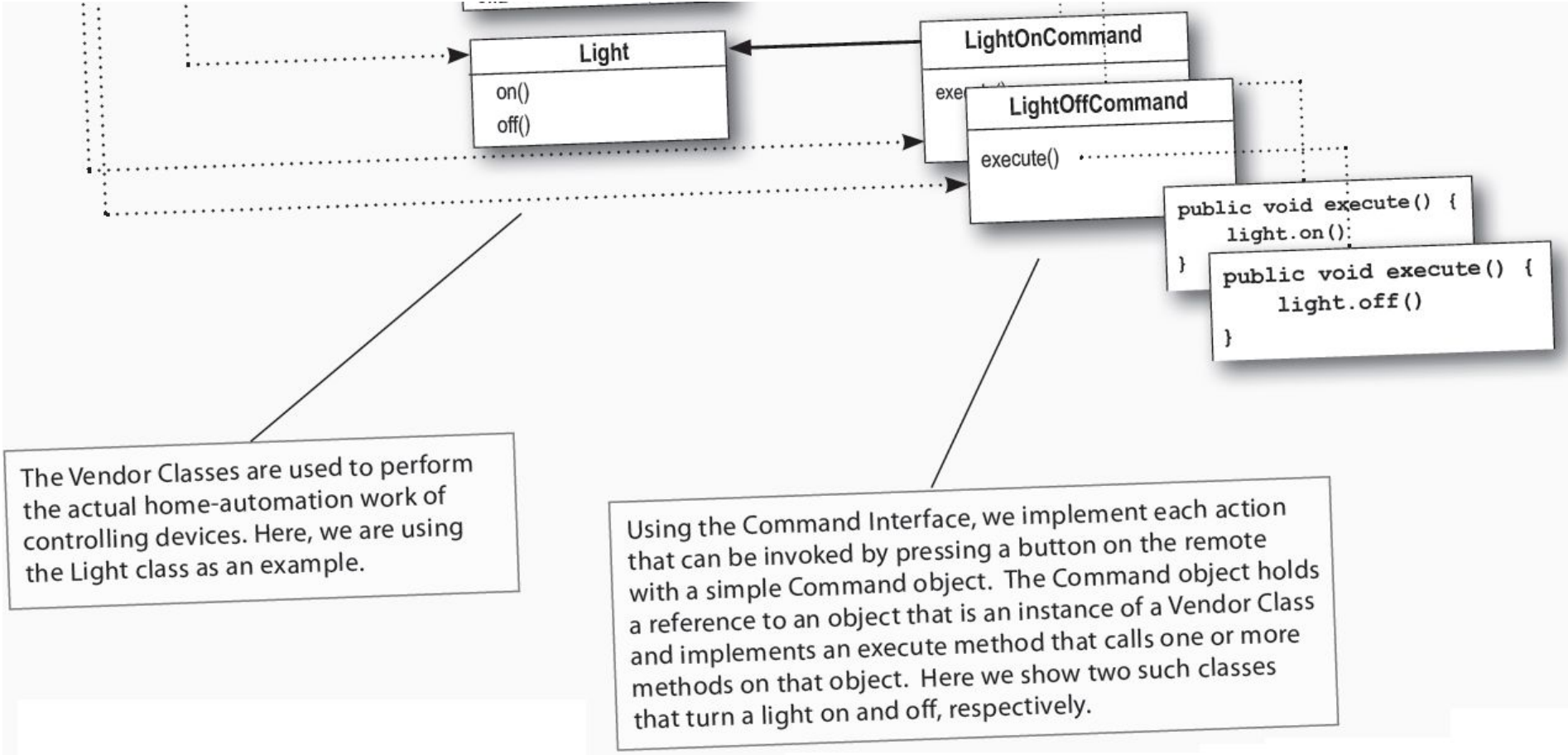
The following class diagram provides an overview of our design:



Zooming in to the documentation




The RemoteLoader creates a number of Command objects that are loaded into the slots of the Remote Control. Each command object encapsulates a request of a home automation device.
The RemoteControl manages a set of Command objects, one per button. When a button is pressed, the corresponding ButtonWasPushed() method is called, which invokes the execute() method on the command. That is the full extent of the remote's knowledge of the classes it's invoking as the Command object decouples the remote from the classes doing the actual home-automation work.
All RemoteControl commands implement the Command interface, which consists of one method: execute(). Commands encapsulate a set of actions on a specific vendor class. The remote invokes these actions by calling the execute() method.
RemoteLoader
RemoteControl
onCommands
offCommands
setCommand()
onButtonWasPushed()
offButtonWasPushed()
<<interface>>
Command
execute()


Light
on()
off()
LightOnCommand
LightOffCommand
execute()
public void execute() {
light.on();
}
public void execute() {
light.off()
}
The Vendor Classes are used to perform the actual home-automation work of controlling devices. Here, we are using the Light class as an example.
Using the Command Interface, we implement each action that can be invoked by pressing a button on the remote with a simple Command object. The Command object holds a reference to an object that is an instance of a Vendor Class and implements an execute method that calls one or more methods on that object. Here we show two such classes that turn a light on and off, respectively.

Enabling Undo

- The execute() method of a command performs a sequence of actions.
- The undo() method reverses the sequence of actions.

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

Here's the new undo() method.

- Let's implement the undo() method to the commands…
  - Let's begin with the Light command.

Updating the LightOnCommand

```
public class LightOnCommand implements Command {
    Light light;

    public LightOnCommand(Light light) {
        this.light = light;
    }

    public void execute() {
        light.on();
    }

    public void undo() {
        light.off();
    }
}
```

execute() turns the light on, so undo() simply turns the light back off.

Updating the LightOffCommand

```
public class LightOffCommand implements Command {
    Light light;

    public LightOffCommand(Light light) {
        this.light = light;
    }

    public void execute() {
        light.off();
    }

    public void undo() {
        light.on();
    }
}
```

And here, undo() turns the light back on.

Updating the Remote Control class

- Here’s how we’re going to do it:
  - We’ll add a new instance variable to track the last command invoked; then, whenever the undo button is pressed, we retrieve that command and invoke its undo() method.

```
public class RemoteControlWithUndo {
    Command[] onCommands;
    Command[] offCommands;
    Command undoCommand;

    public RemoteControlWithUndo() {
        onCommands = new Command[7];
        offCommands = new Command[7];

        Command noCommand = new NoCommand();
        for(int i=0;i<7;i++) {
            onCommands[i] = noCommand;
            offCommands[i] = noCommand;
        }
        undoCommand = noCommand;
    }

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

This is where we'll stash the last command executed for the undo button.

Just like the other slots, undo starts off with a NoCommand, so pressing undo before any other button won't do anything at all.

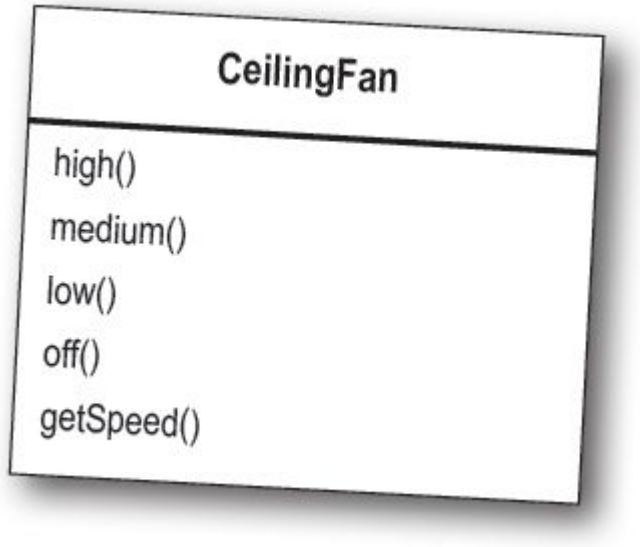
```
public void onButtonWasPushed(int slot) {
    onCommands[slot].execute();
    undoCommand = onCommands[slot];
}

public void offButtonWasPushed(int slot) {
    offCommands[slot].execute();
    undoCommand = offCommands[slot];
}

public void undoButtonWasPushed() {
    undoCommand.undo();
}

public String toString() {
    // toString code here...
}
```

When a button is pressed, we take the command and first execute it; then we save a reference to it in the undoCommand instance variable. We do this for both "on" commands and "off" commands.

When the undo button is pressed, we invoke the undo() method of the command stored in undoCommand. This reverses the operation of the last command executed.

Using state to implement Undo

- Let’s try something a little more interesting, like the CeilingFan from the vendor classes.
- The CeilingFan allows a number of speeds to be set along with an off method.



```
public class CeilingFan {
    public static final int HIGH = 3;
    public static final int MEDIUM = 2;
    public static final int LOW = 1;
    public static final int OFF = 0;
    String location;
    int speed;

    public CeilingFan(String location) {
        this.location = location;
        speed = OFF;
    }

    public void high() {
        speed = HIGH;
        // code to set fan to high
    }

    public void medium() {
        speed = MEDIUM;
        // code to set fan to medium
    }

    public void low() {
        speed = LOW;
        // code to set fan to low
    }

    public void off() {
        speed = OFF;
        // code to turn fan off
    }

    public int getSpeed() {
        return speed;
    }
}
```

Notice that the CeilingFan class holds local state representing the speed of the ceiling fan.

These methods set the speed of the ceiling fan.

We can get the current speed of the ceiling fan using getSpeed().

Adding Undo to the CeilingFan commands

- Let’s tackle adding undo to the various CeilingFan commands.
- To do so, we need to track the last speed setting of the fan and, if the undo() method is called, restore the fan to its previous setting.
- Here’s the code for the CeilingFanHighCommand…

```
public class CeilingFanHighCommand implements Command {
    CeilingFan ceilingFan;
    int prevSpeed;

    public CeilingFanHighCommand(CeilingFan ceilingFan) {
        this.ceilingFan = ceilingFan;
    }

    public void execute() {
        prevSpeed = ceilingFan.getSpeed();
        ceilingFan.high();
    }

    public void undo() {
        if (prevSpeed == CeilingFan.HIGH) {
            ceilingFan.high();
        } else if (prevSpeed == CeilingFan.MEDIUM) {
            ceilingFan.medium();
        } else if (prevSpeed == CeilingFan.LOW) {
            ceilingFan.low();
        } else if (prevSpeed == CeilingFan.OFF) {
            ceilingFan.off();
        }
    }
}
```

We've added local state to keep track of the previous speed of the fan.

In execute, before we change the speed of the fan, we need to first record its previous state, just in case we need to undo our actions.

To undo, we set the speed of the fan back to its previous speed.

Macro Commands

- We can make a new kind of Command that can execute other Commands…
- Macro Commands are a simple extension of Command that allow multiple commands to be invoked.
- Likewise, Macro Commands can easily support undo().

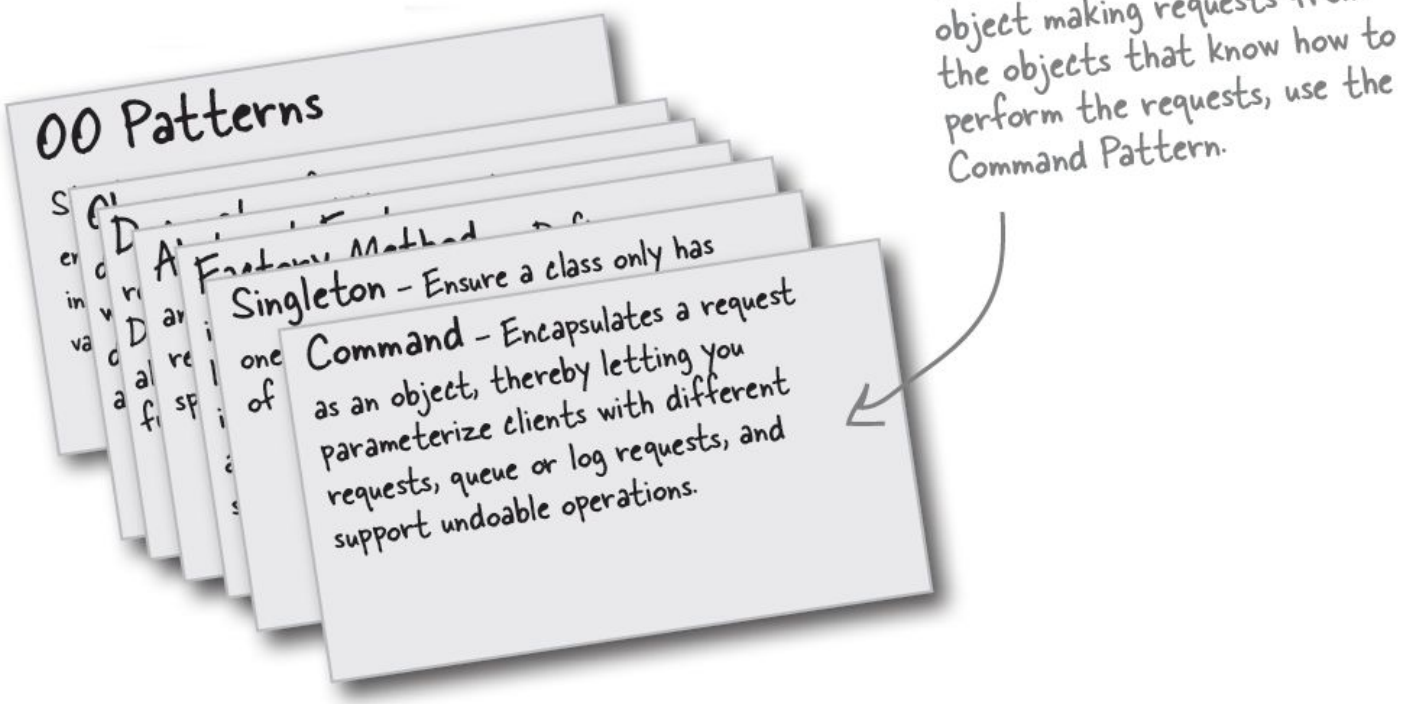
Macro Commands (cont.)

```
public class MacroCommand implements Command {
    Command[] commands;

    public MacroCommand(Command[] commands) {
        this.commands = commands;
    }

    public void execute() {
        for (int i = 0; i < commands.length; i++) {
            commands[i].execute();
        }
    }
}
```

Take an array of Commands and store them in the MacroCommand.

When the macro gets executed by the remote, execute those commands one at a time.

Tools for your Design Toolbox



References

Material in this lecture is taken from Freeman, E., Robson, E., Bates, B., & Sierra, K., *Head First Design Patterns: A Brain-Friendly Guide*, O'Reilly Media, Inc., 2014.