Agenda

- Command Pattern

Command Pattern

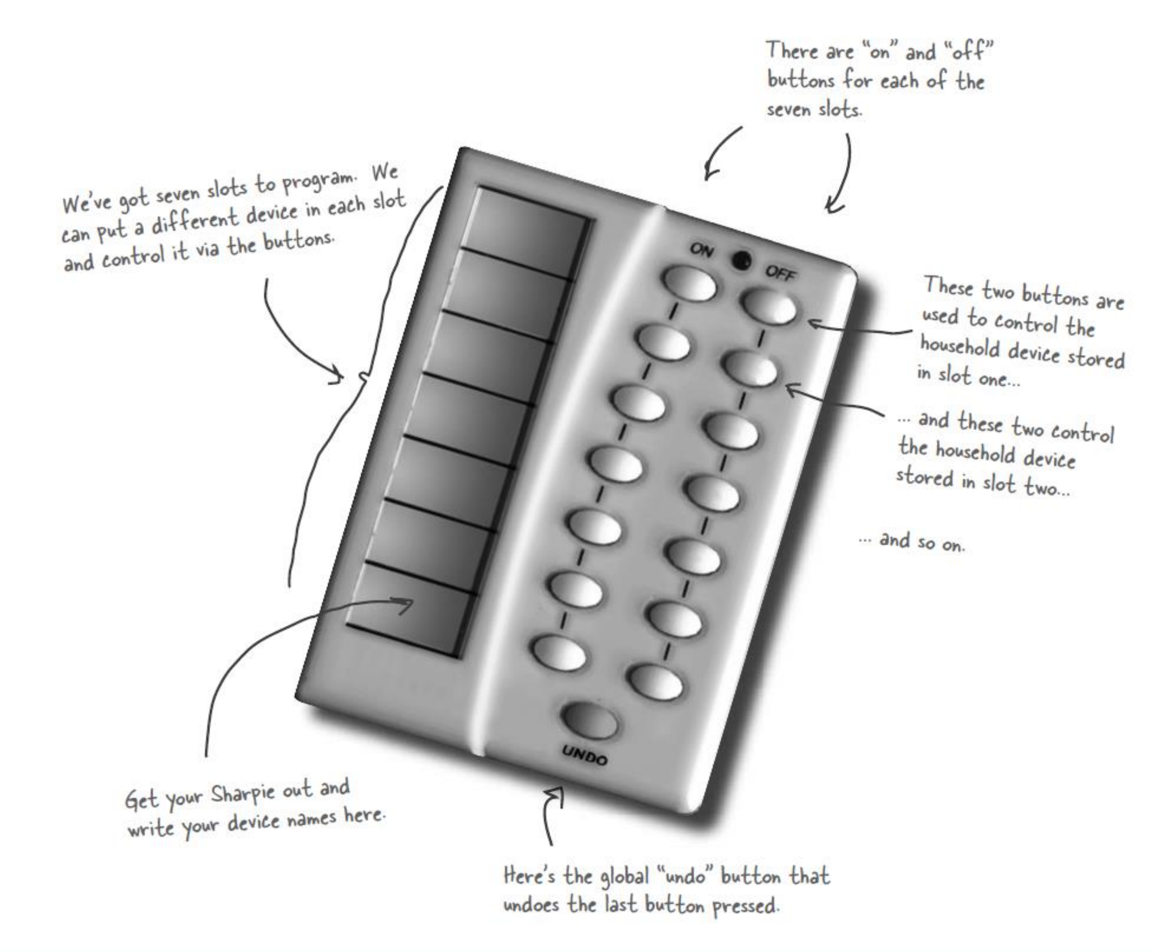
Encapsulating Invocation

Motivating problem description

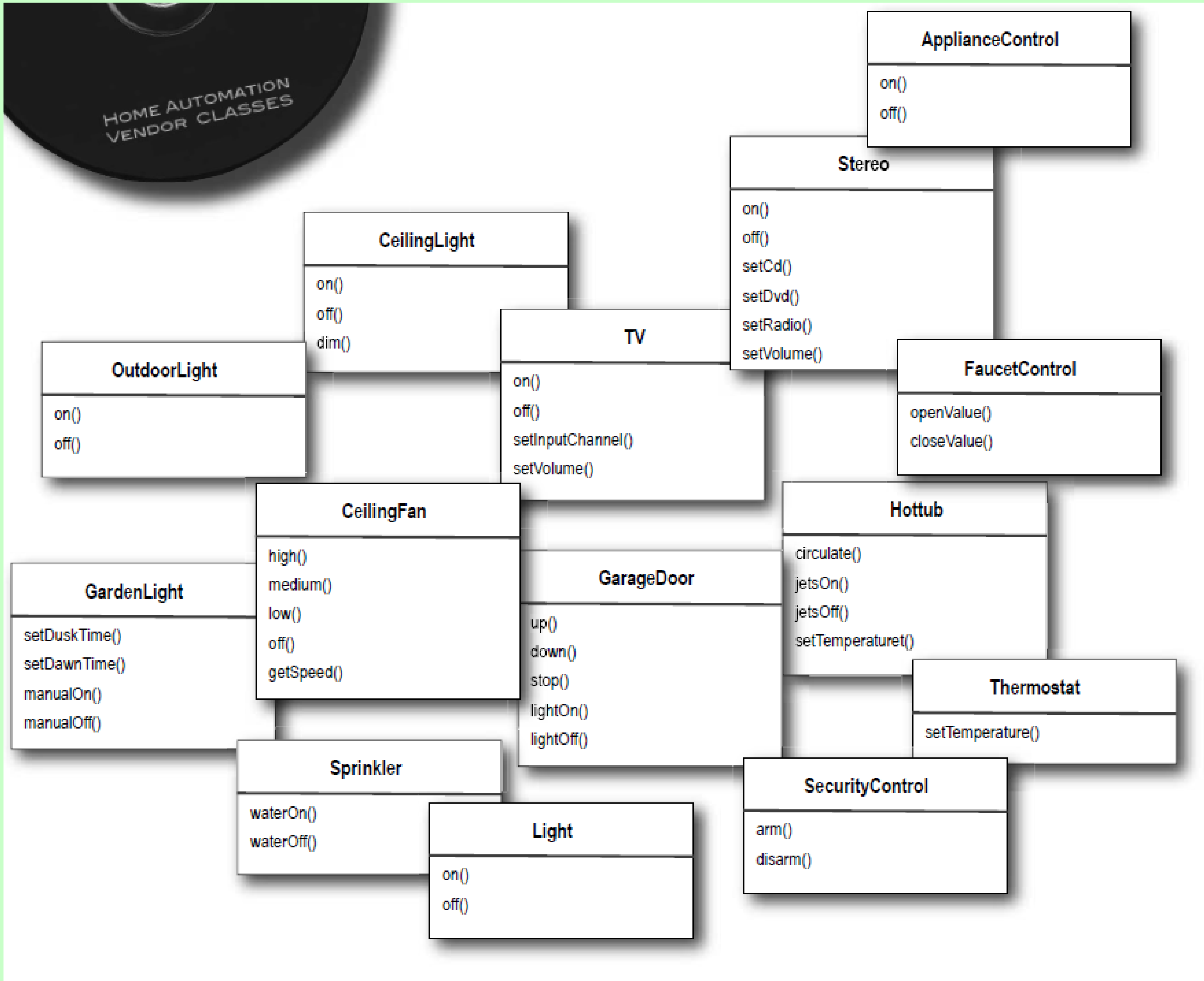
- Build a remote that will control variety of home devices

- Sample devices: lights, stereo, TV, ceiling light, thermostat, sprinkler, hot tub, garden light, ceiling fan, garage door

Motivating problem description


There are "on" and "off" buttons for each of the seven slots.
We've got seven slots to program. We can put a different device in each slot and control it via the buttons.
ON
OFF
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.
UNDO
Get your Sharpie out and write your device names here.
Here's the global "undo" button that undoes the last button pressed.

Motivating problem description



ApplianceControl
on()
off()

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

CeilingLight
on()
off()
dim()

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

OutdoorLight
on()
off()

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()

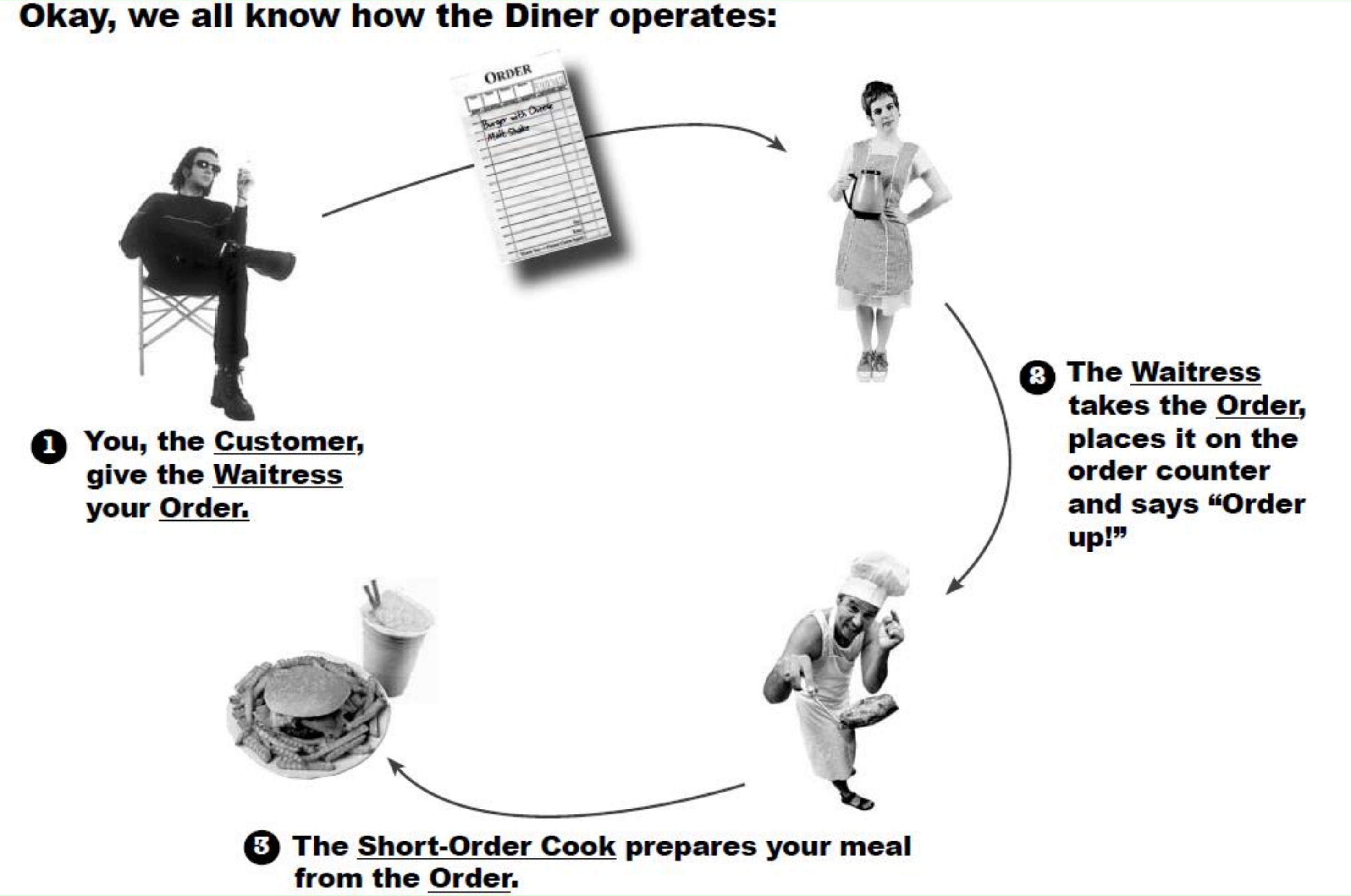
SecurityControl
arm()
disarm()

Light
on()
off()

Command Pattern

- The Command Pattern allows you to decouple the requester of an action from the object that actually performs the action.
- A command object encapsulates a request to do something (like turn on a light) on a specific object (say, the living room light object).

Command Pattern Analogy

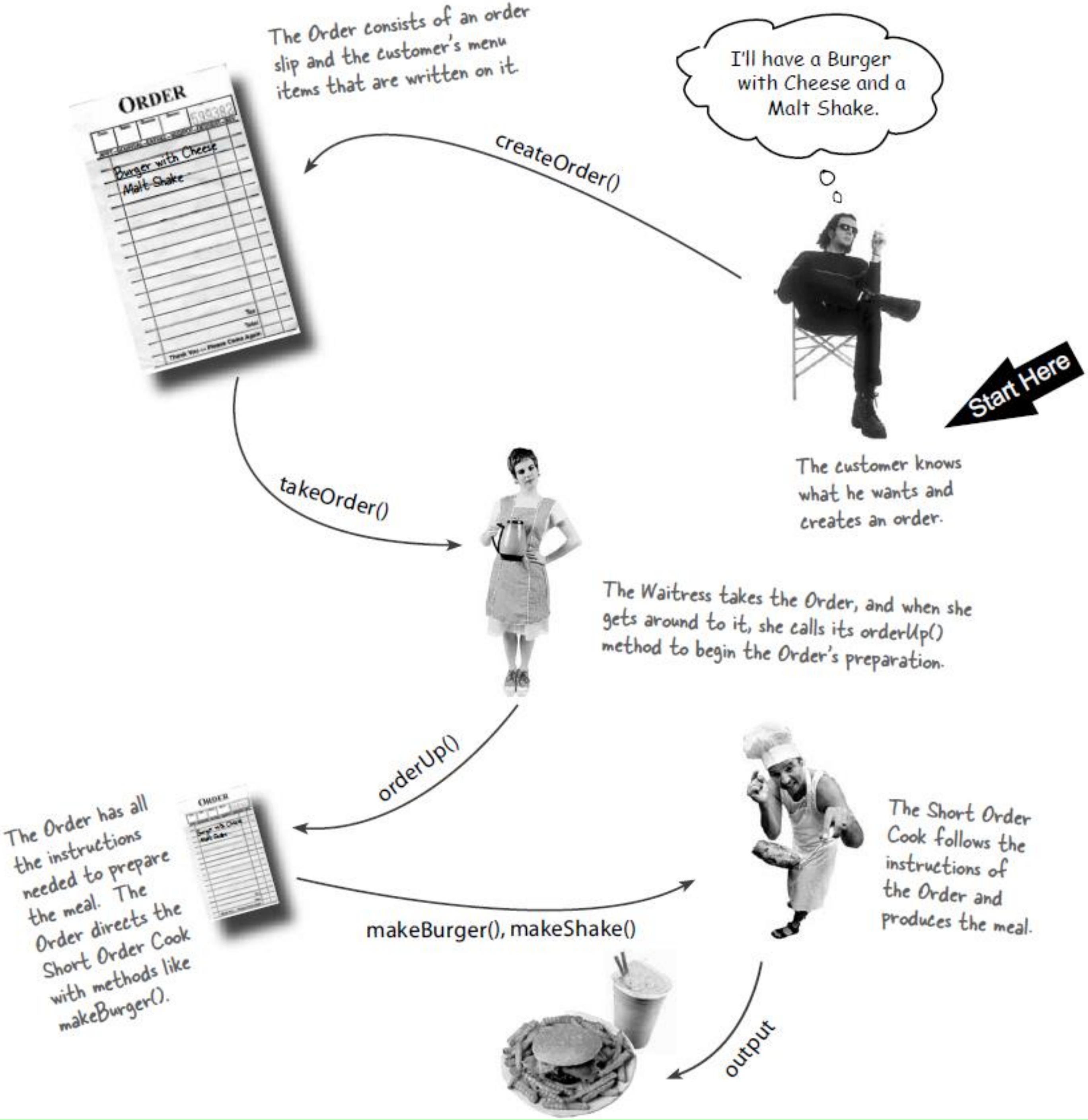


Okay, we all know how the Diner operates:

1. You, the Customer, give the Waitress your Order.

2. The Waitress takes the Order, places it on the order counter and says “Order up!”

3. The Short-Order Cook prepares your meal from the Order.



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

The customer knows what he wants and creates an order.

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

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

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

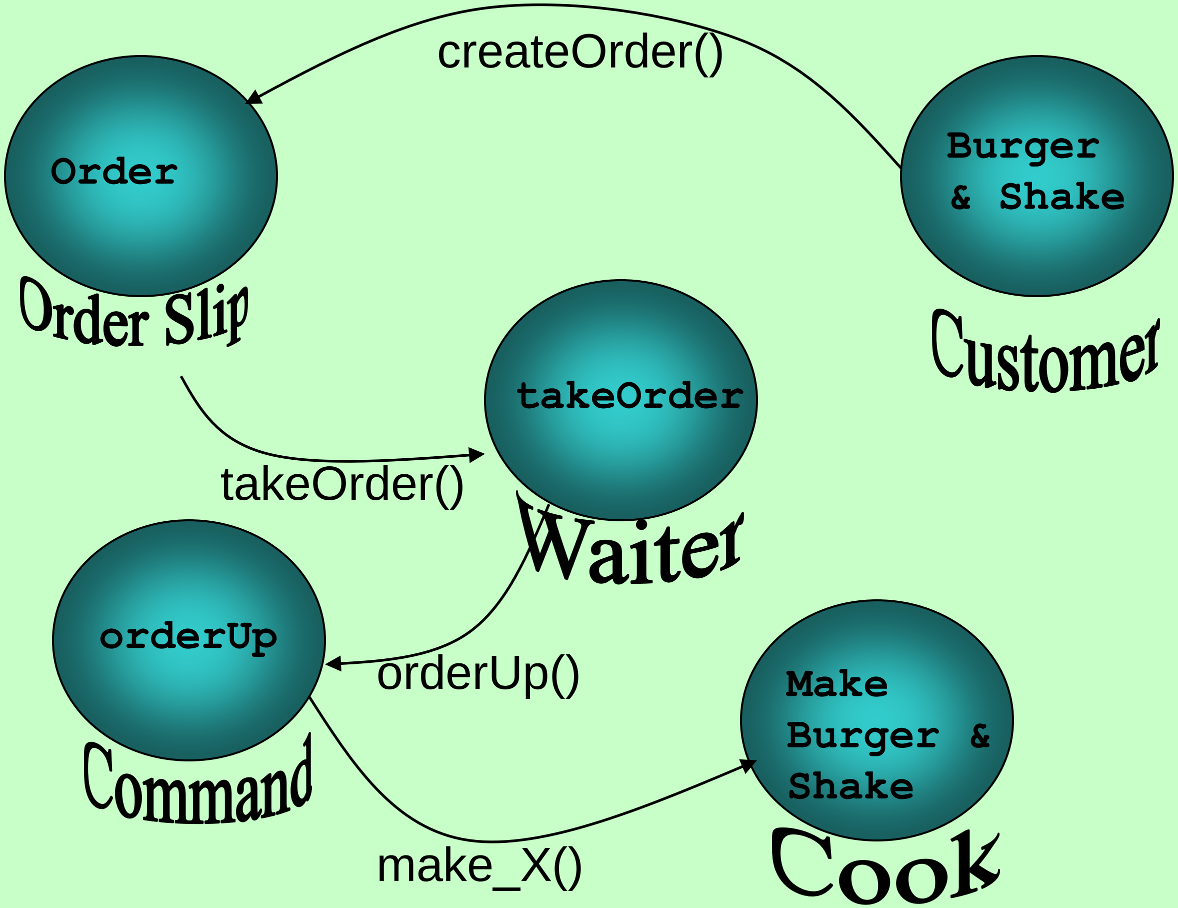
Diner roles and responsibilities

- An Order Slip encapsulates a request to prepare a meal.

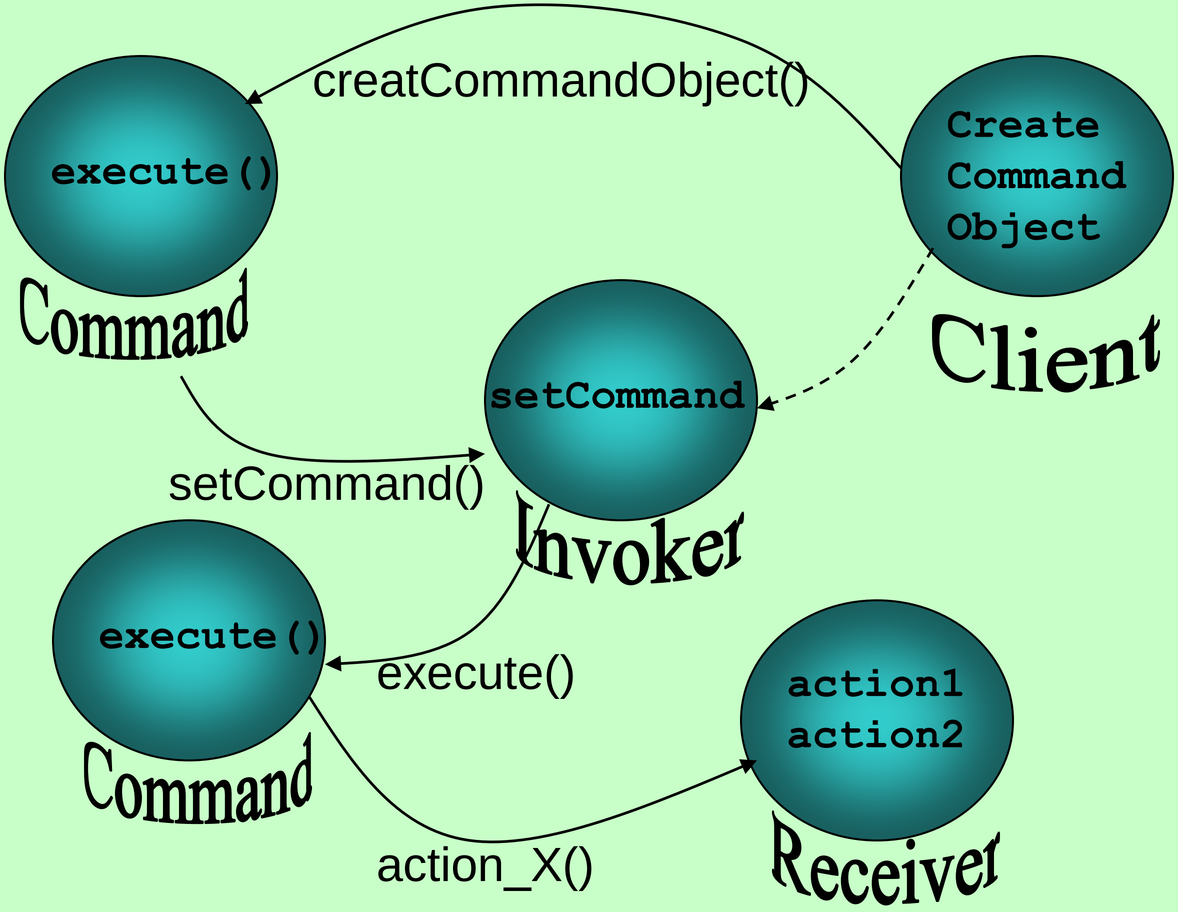
- The Waitress’s job is to take Order Slips and invoke the orderUp() method on them.

- The Short Order Cook has the knowledge required to prepare the meal.

Introducing the command pattern – Diner example



Introducing the command pattern



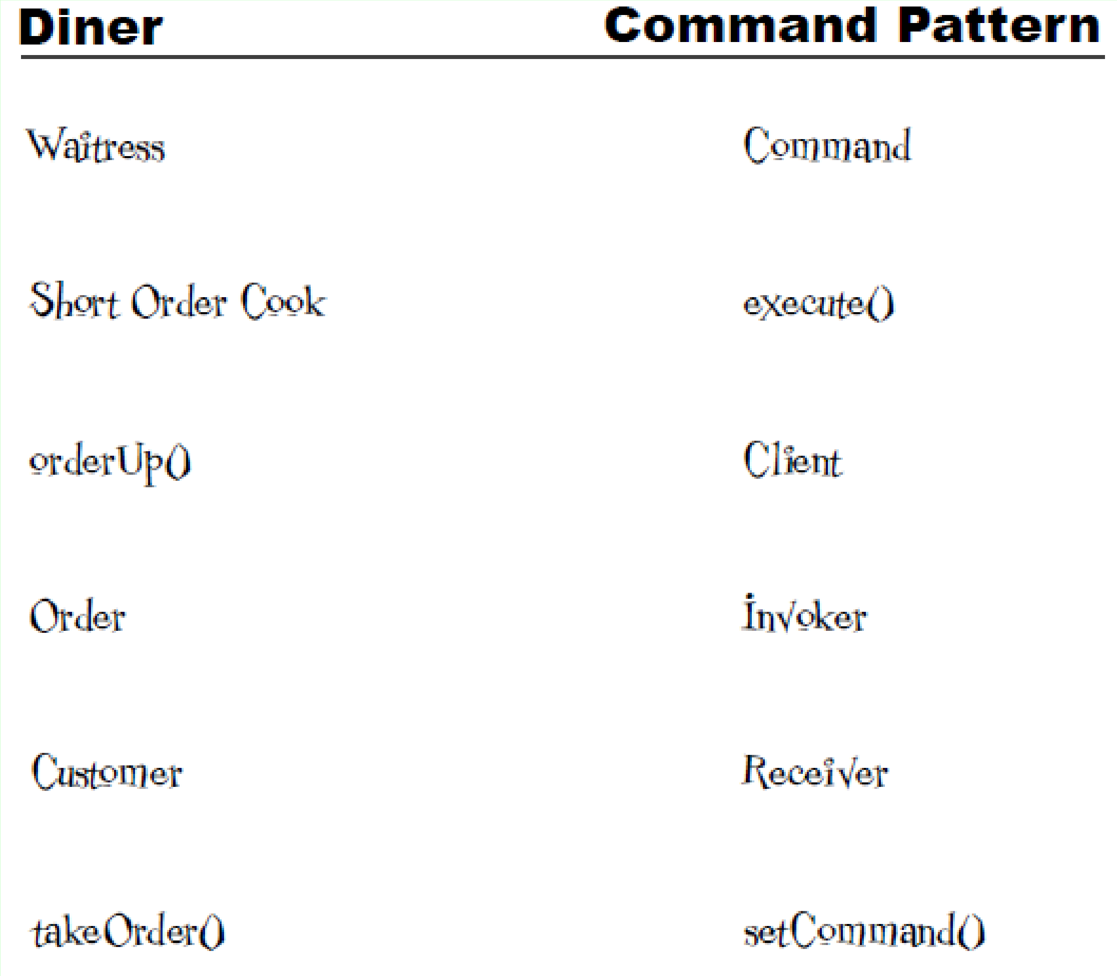
Introducing the command pattern

- The client creates a command object .

- The client does a set Command( ) to store the command object in the invoker.

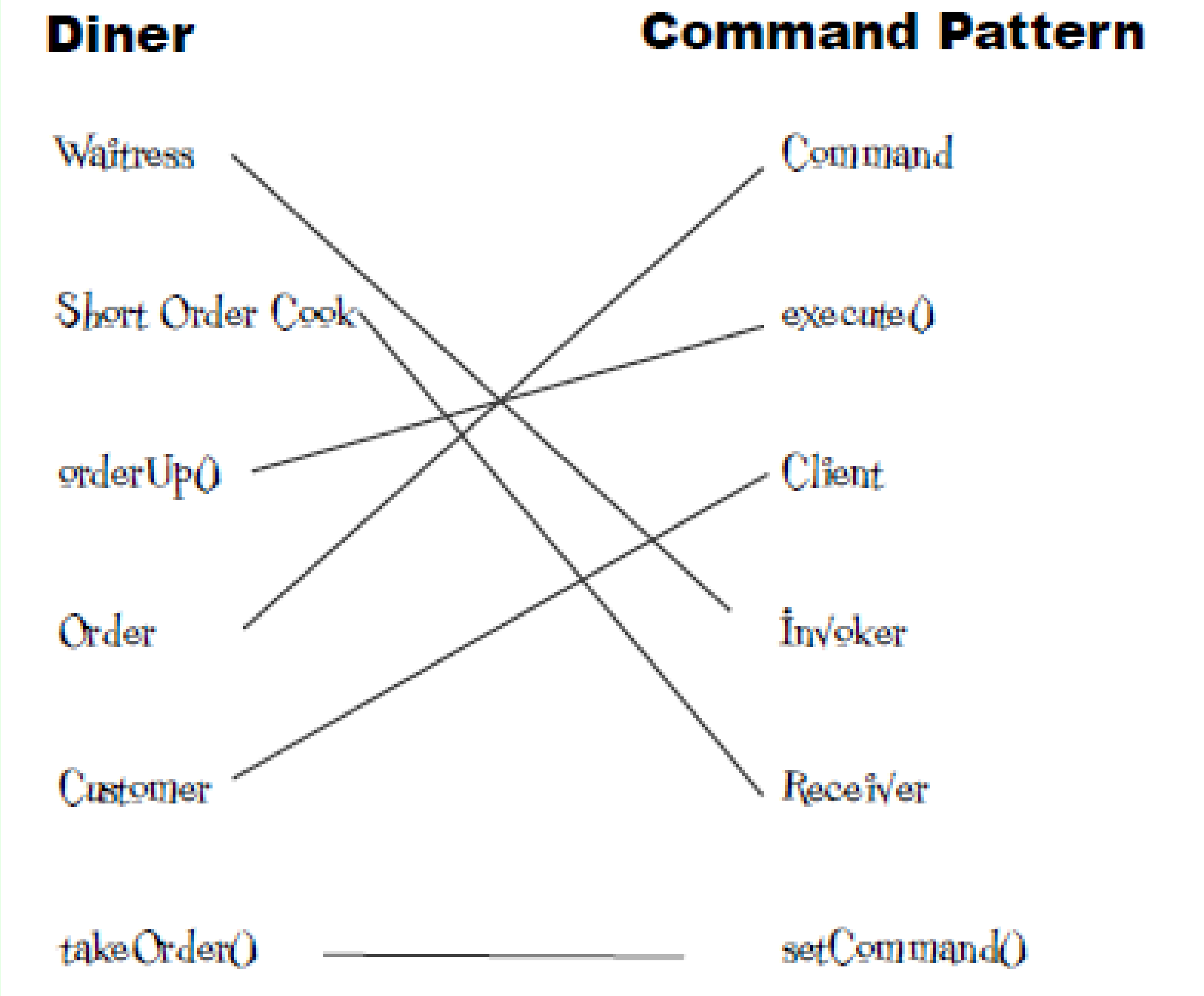
- Later... the client asks the invoker to execute the command.

Who does what?



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

Who does what?



Implementing the Command interface

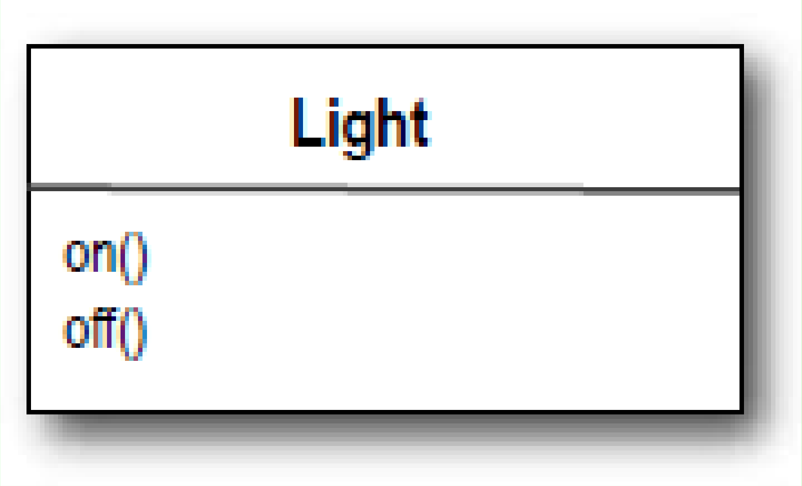
- All command objects implement the same interface, which consists of one method.
- In the Diner we called this method orderUp(); however, we typically just use the name **execute().**

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

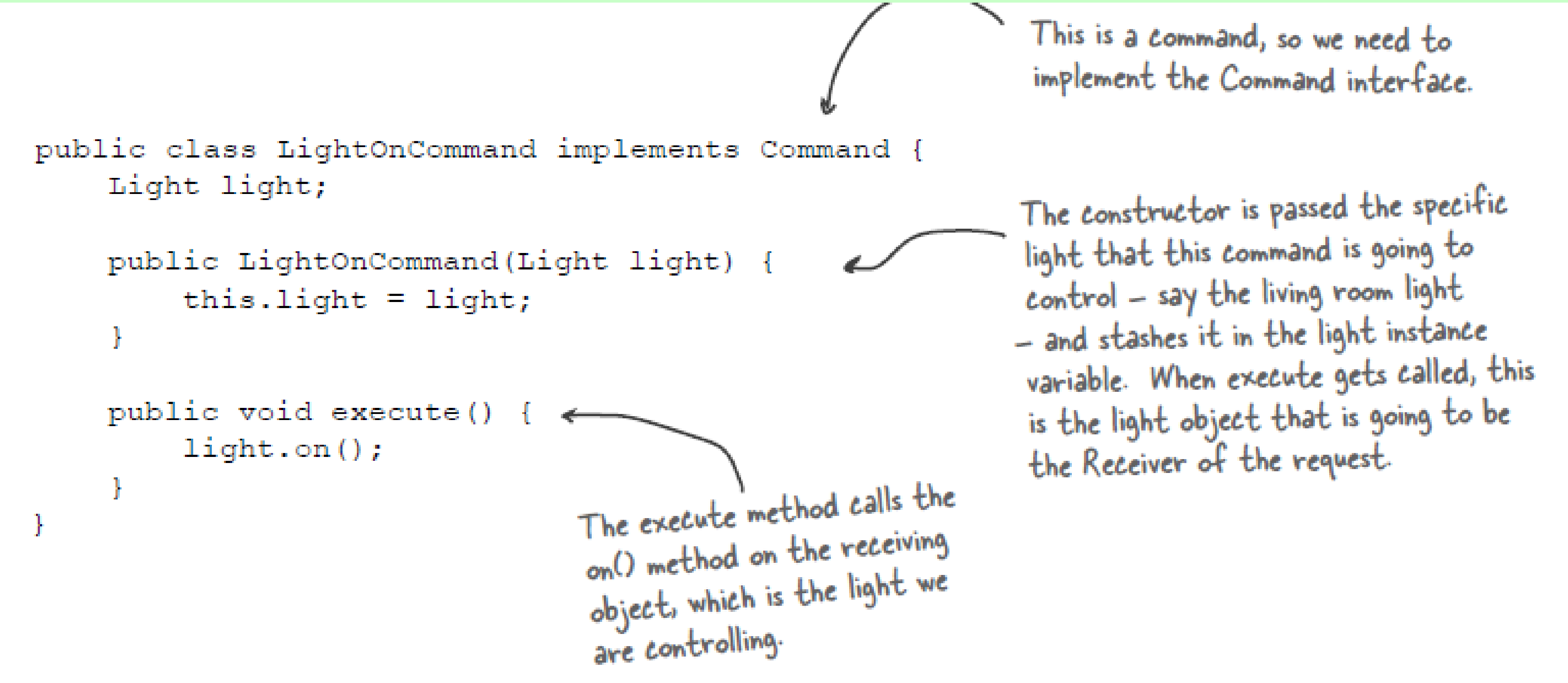
Implementing a Command to turn a light on

- Now, 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()



Implementing a Command to turn a light on



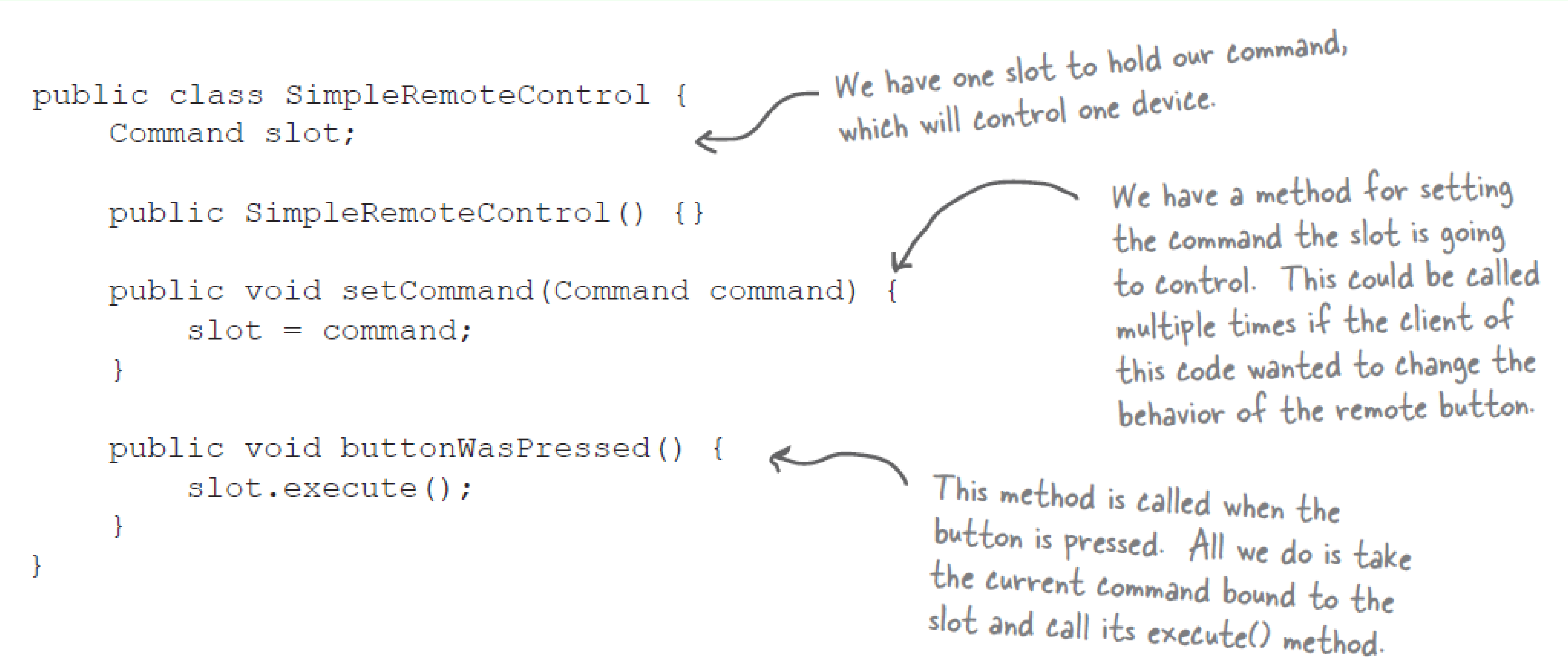
This is a command, so we need to implement the Command interface.

```
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.

A remote control with only one button



```
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.

Simple test to use the Remote Control

This is our Client in Command Pattern-speak.

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

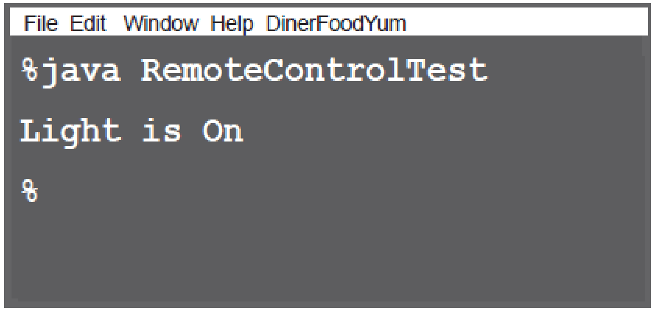
Now we create a Light object, this will be the Receiver of the request.

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!



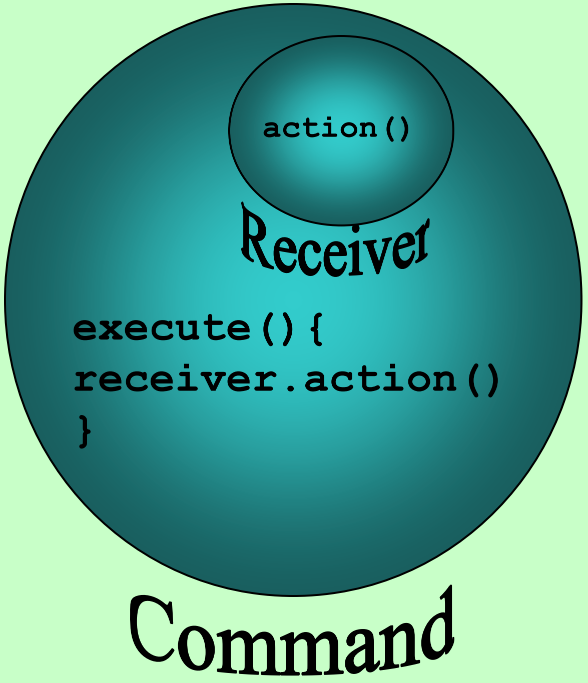
```
%java RemoteControlTest
Light is On
%
```

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.

Command Pattern for home automation

An encapsulated Request

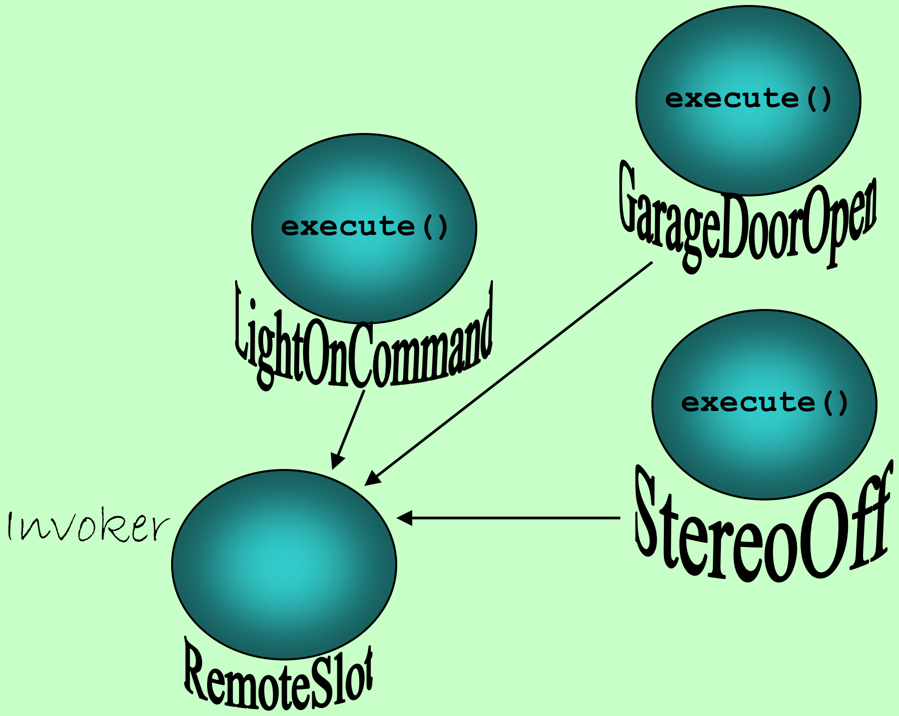


action()

Receiver

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

Command



execute()

GarageDoorOpen

execute()

LightOnCommand

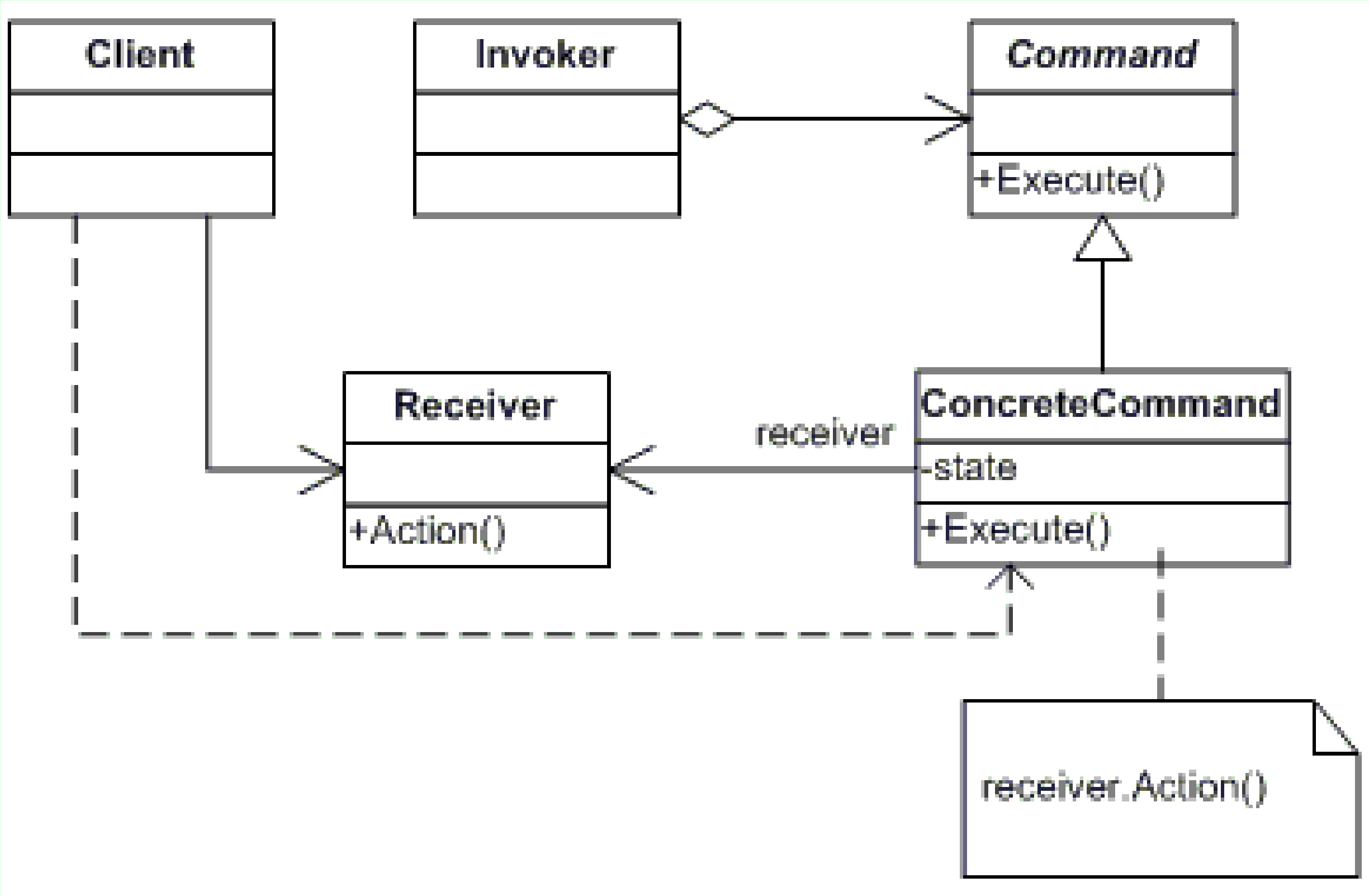
execute()

StereoOff

Invoker

RemoteSlot

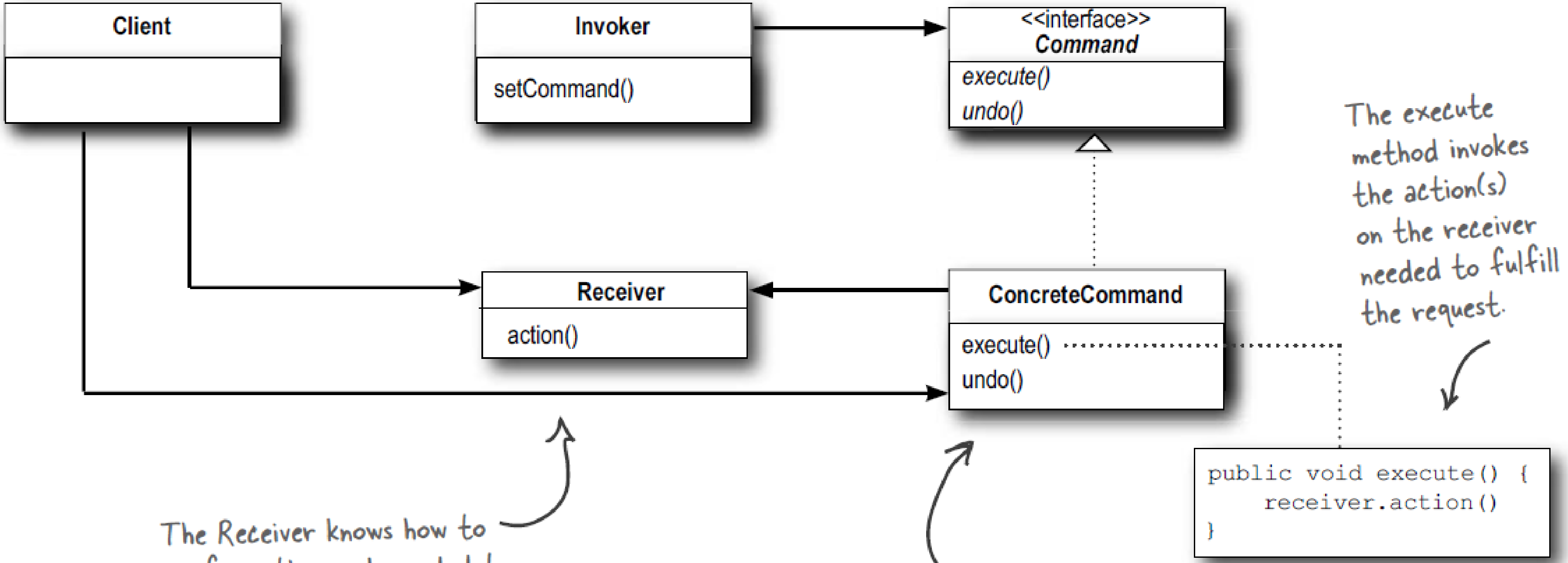
General Form



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.



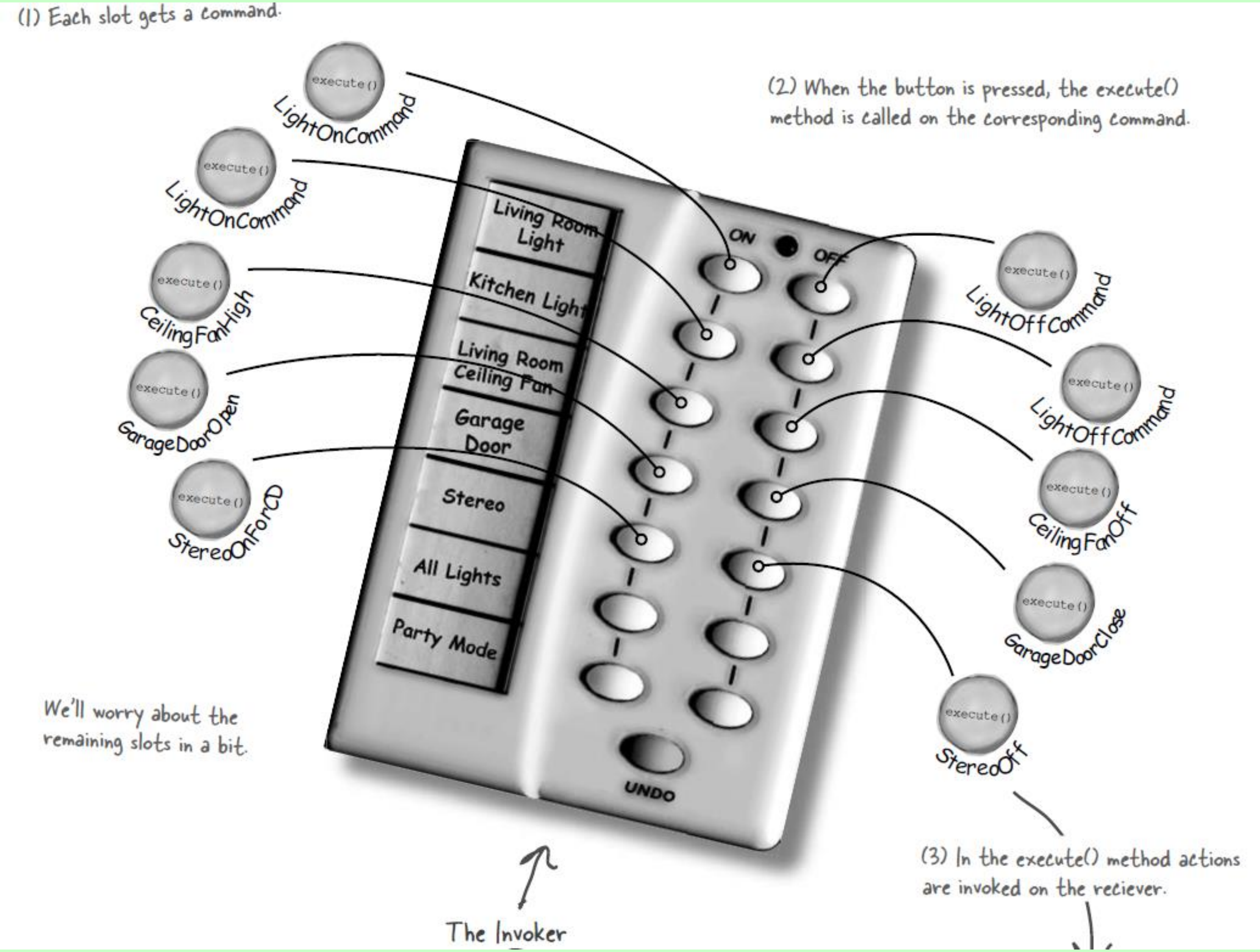
```
public void execute() {
    receiver.action()
}
```

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

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.

Assigning Commands to Slots



(1) Each slot gets a command.

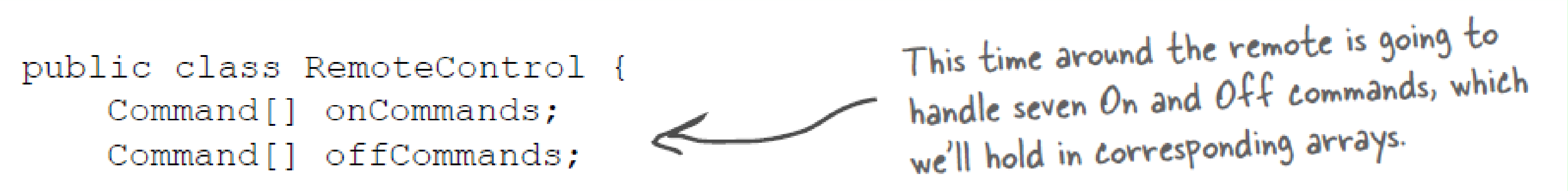
(2) When the button is pressed, the execute() method is called on the corresponding command.

We'll worry about the remaining slots in a bit.

(3) In the execute() method actions are invoked on the reciever.

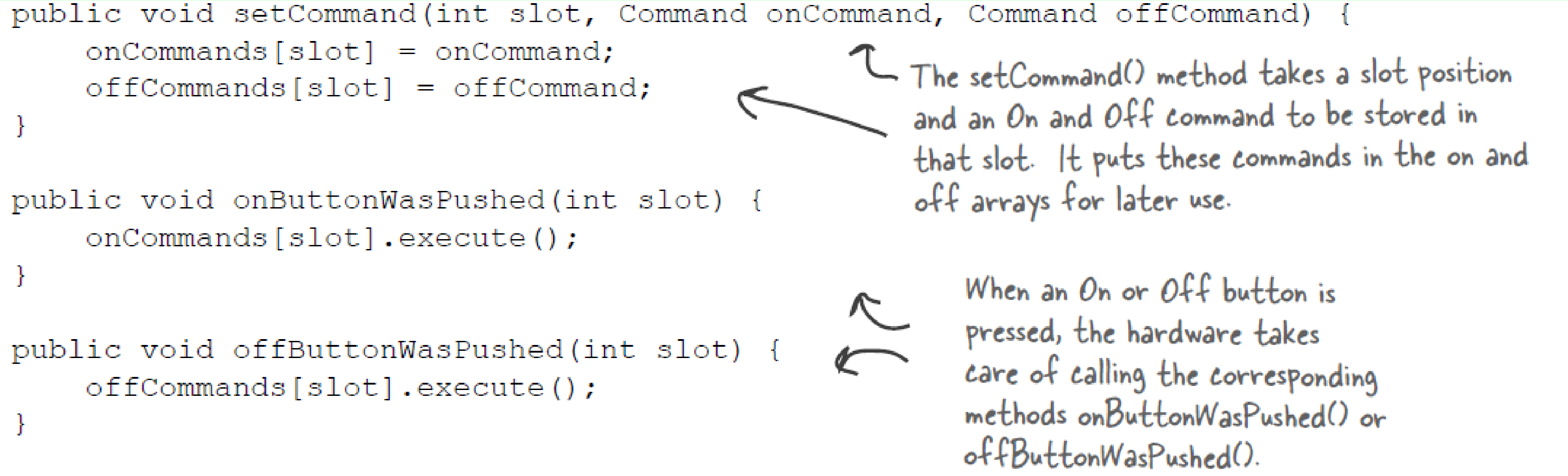
The Invoker

Implementing the Remote Control



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

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

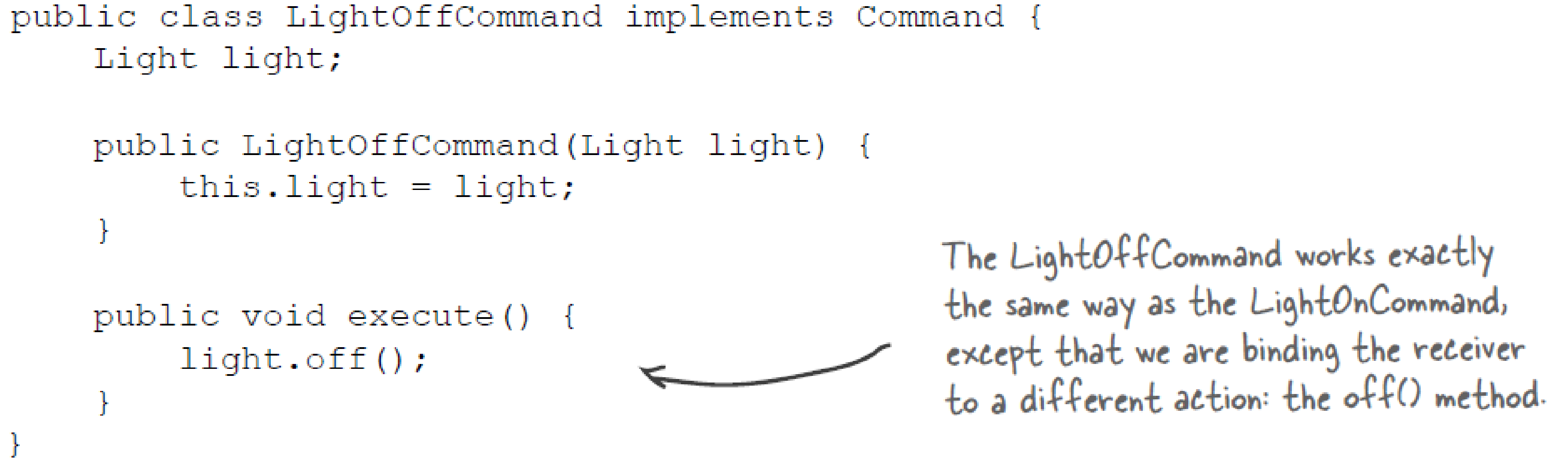


```
public void setCommand(int slot, Command onCommand, Command offCommand) {
    onCommands[slot] = onCommand;
    offCommands[slot] = offCommand;
}

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

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

The setCommand() method takes a slot position and an On and Off command to be stored in that slot. It puts these commands in the on and off arrays for later use.

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

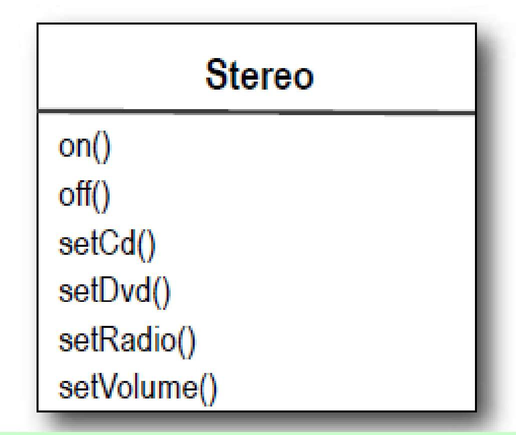
Implementing the Commands



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

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

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

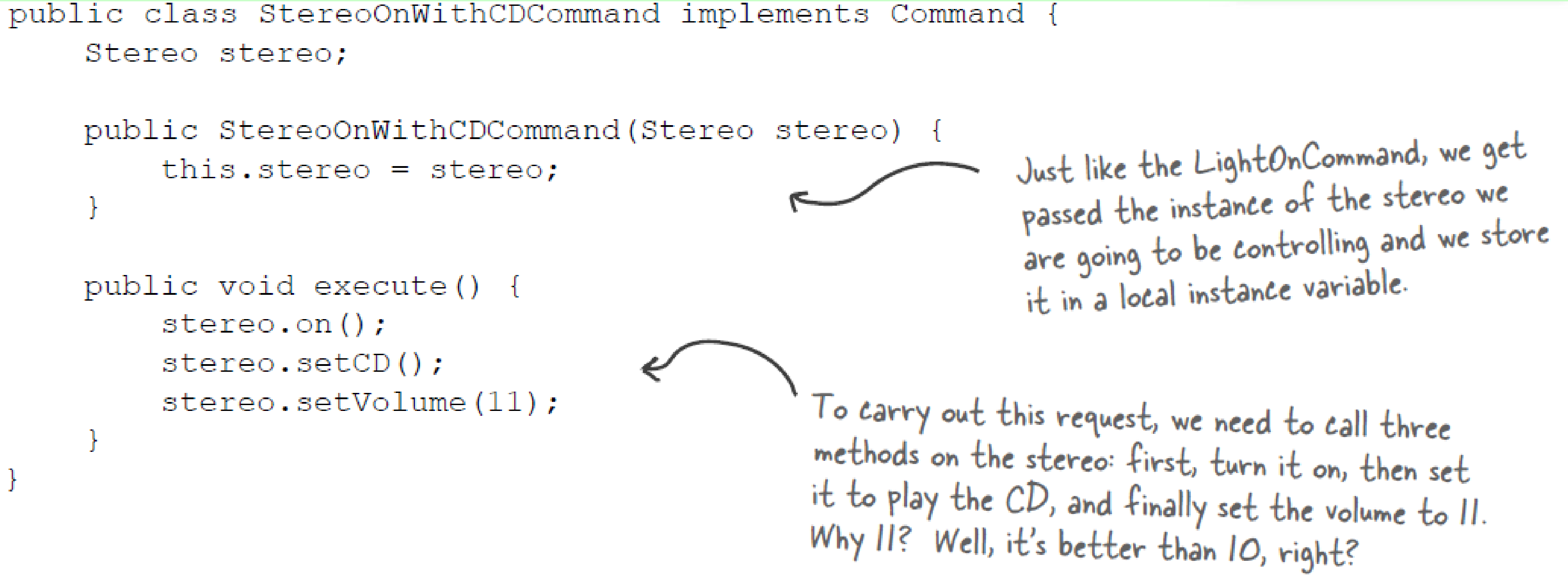
The LightOffCommand works exactly the same way as the LightOnCommand, except that we are binding the receiver to a different action: the off() method.

Implementing the Commands



Stereo

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

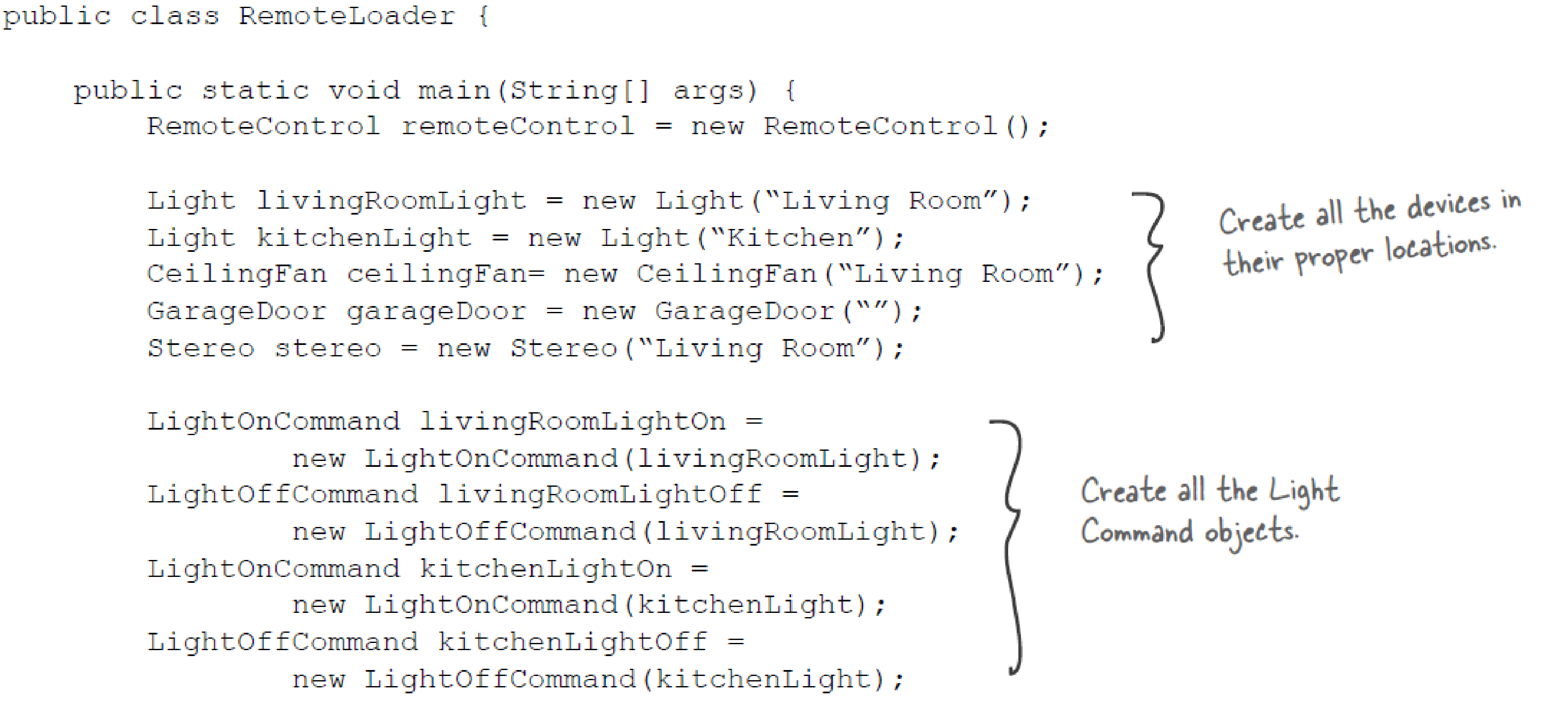


```
public class StereoOnWithCDCommand implements Command {
    Stereo stereo;

    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?

Remote Loader (Client)

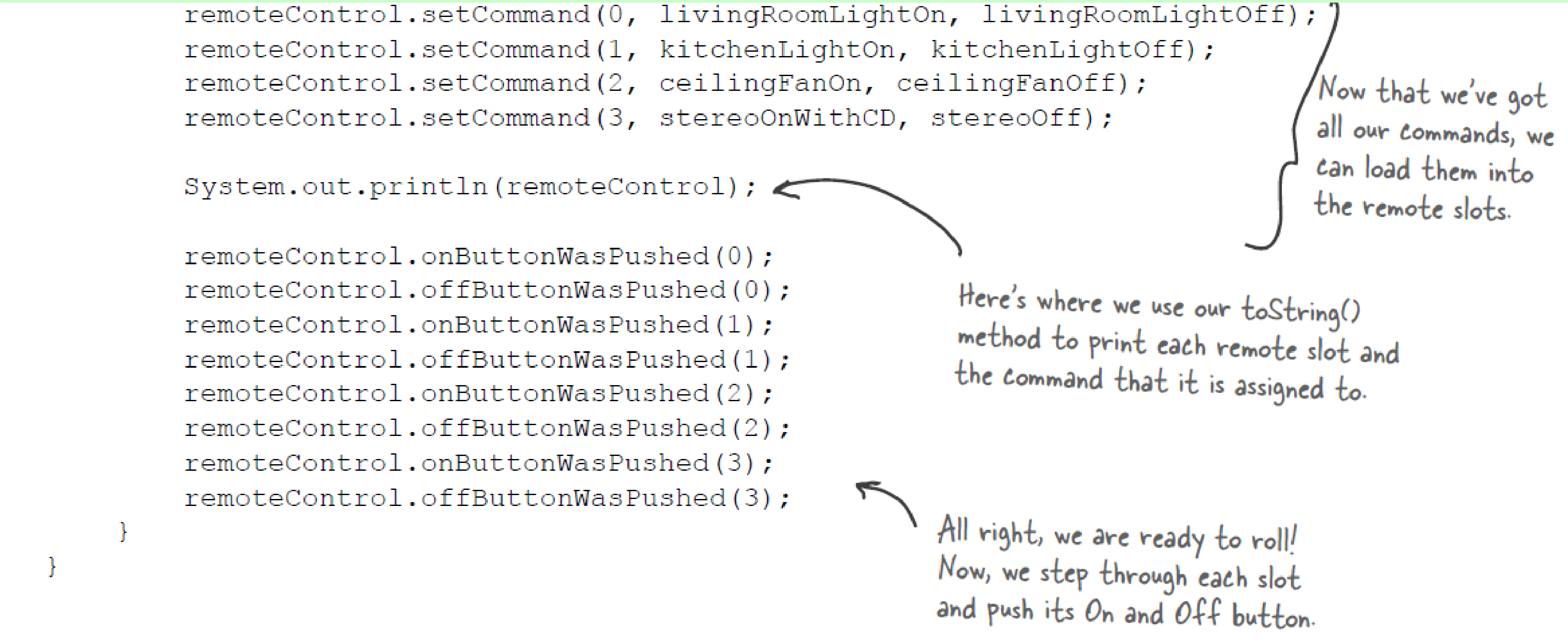


```
public class RemoteLoader {

    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.

Remote Loader (Client)



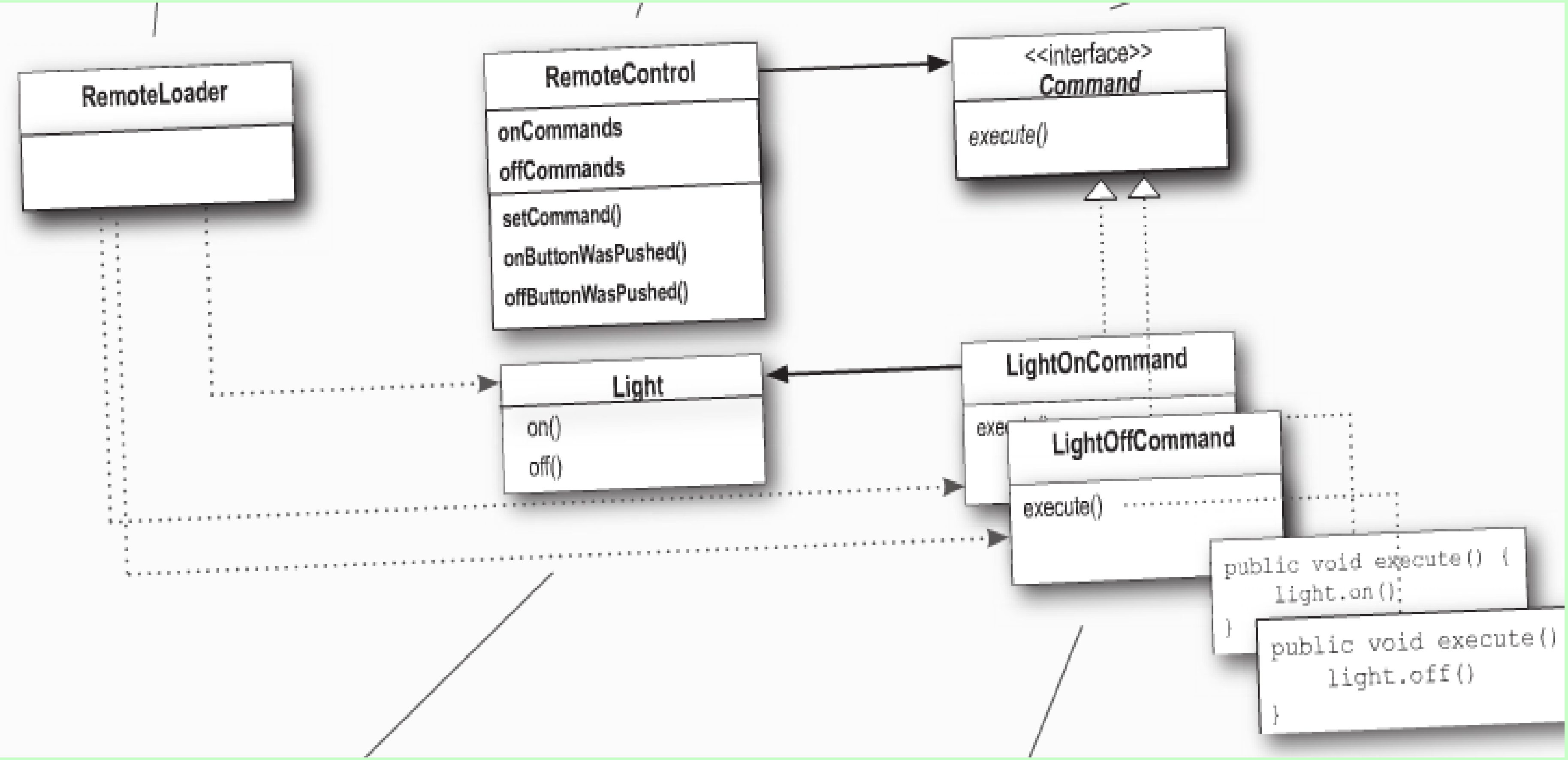
```
        remoteControl.setCommand(0, livingRoomLightOn, livingRoomLightOff);
        remoteControl.setCommand(1, kitchenLightOn, kitchenLightOff);
        remoteControl.setCommand(2, ceilingFanOn, ceilingFanOff);
        remoteControl.setCommand(3, stereoOnWithCD, stereoOff);

        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.

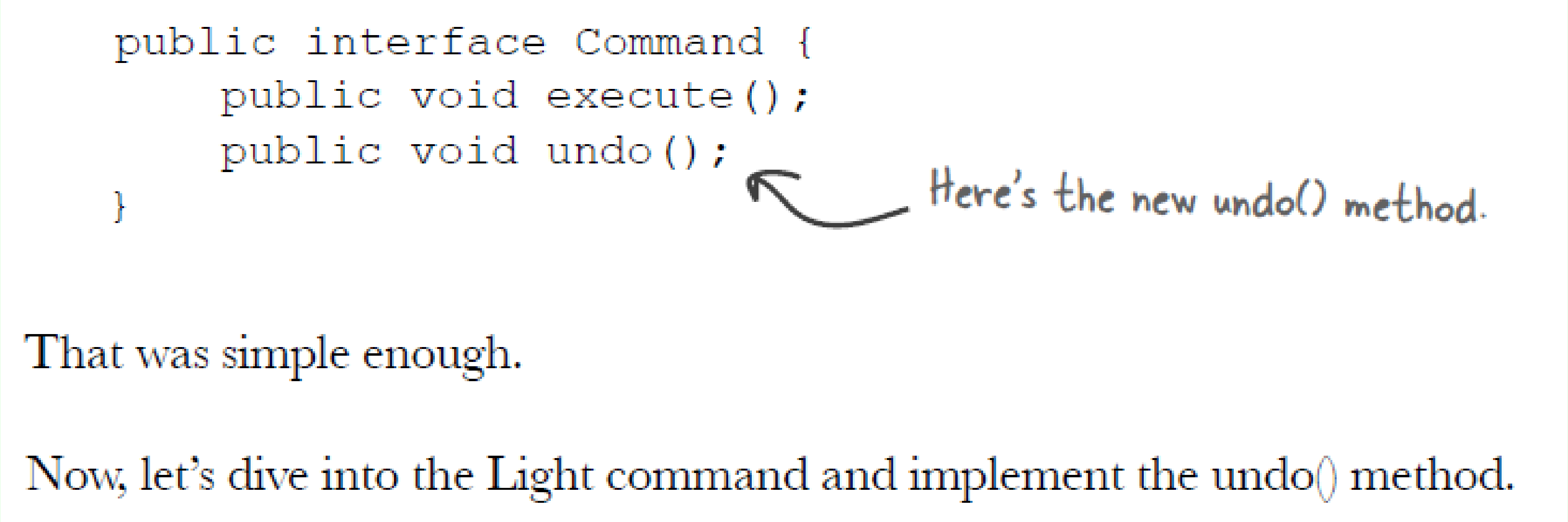
Remote Control API Design



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

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

What about undo?



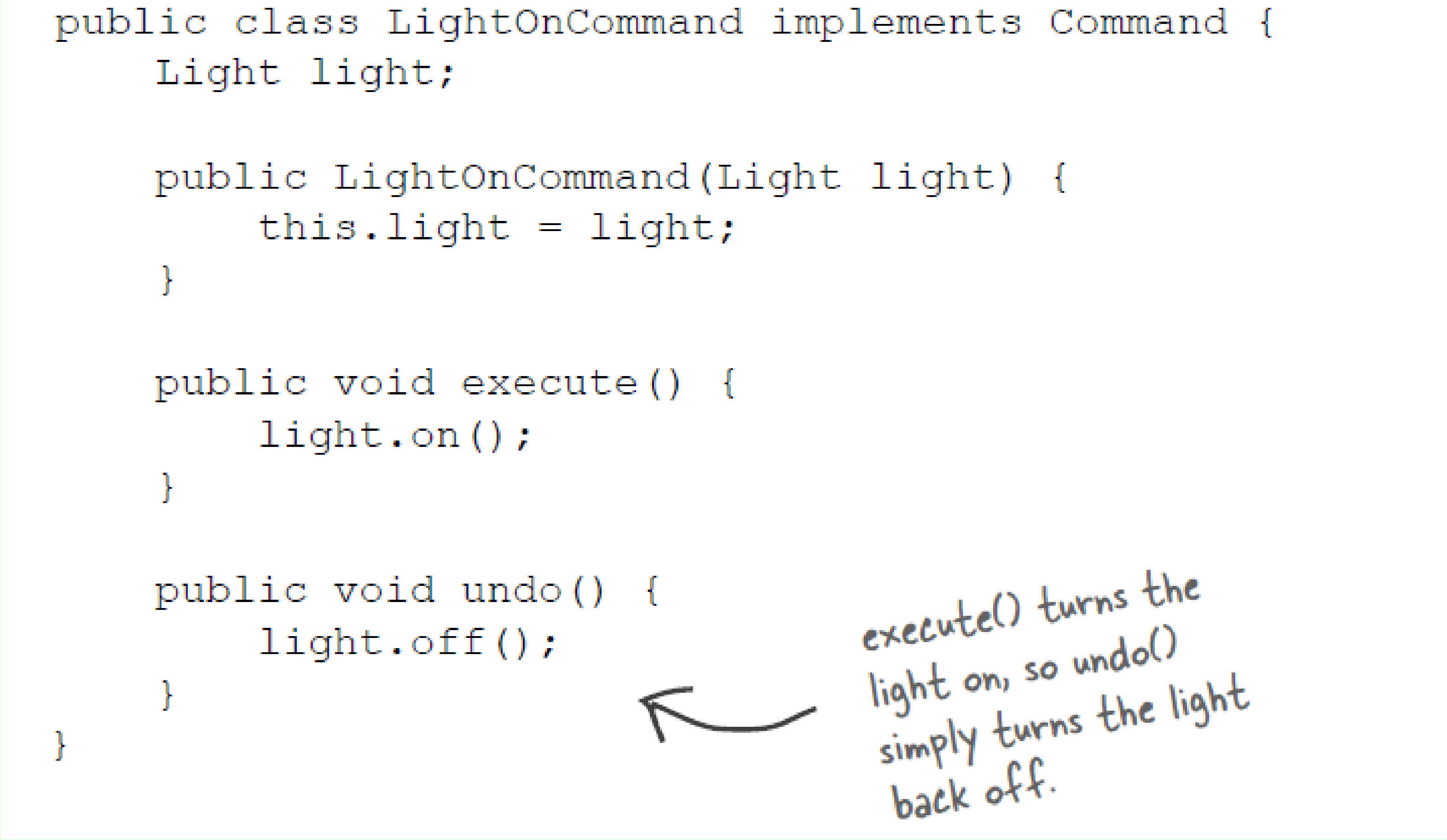
```
public interface Command {
    public void execute();
    public void undo();
}
```

Here's the new undo() method.

That was simple enough.

Now, let's dive into the Light command and implement the undo() method.

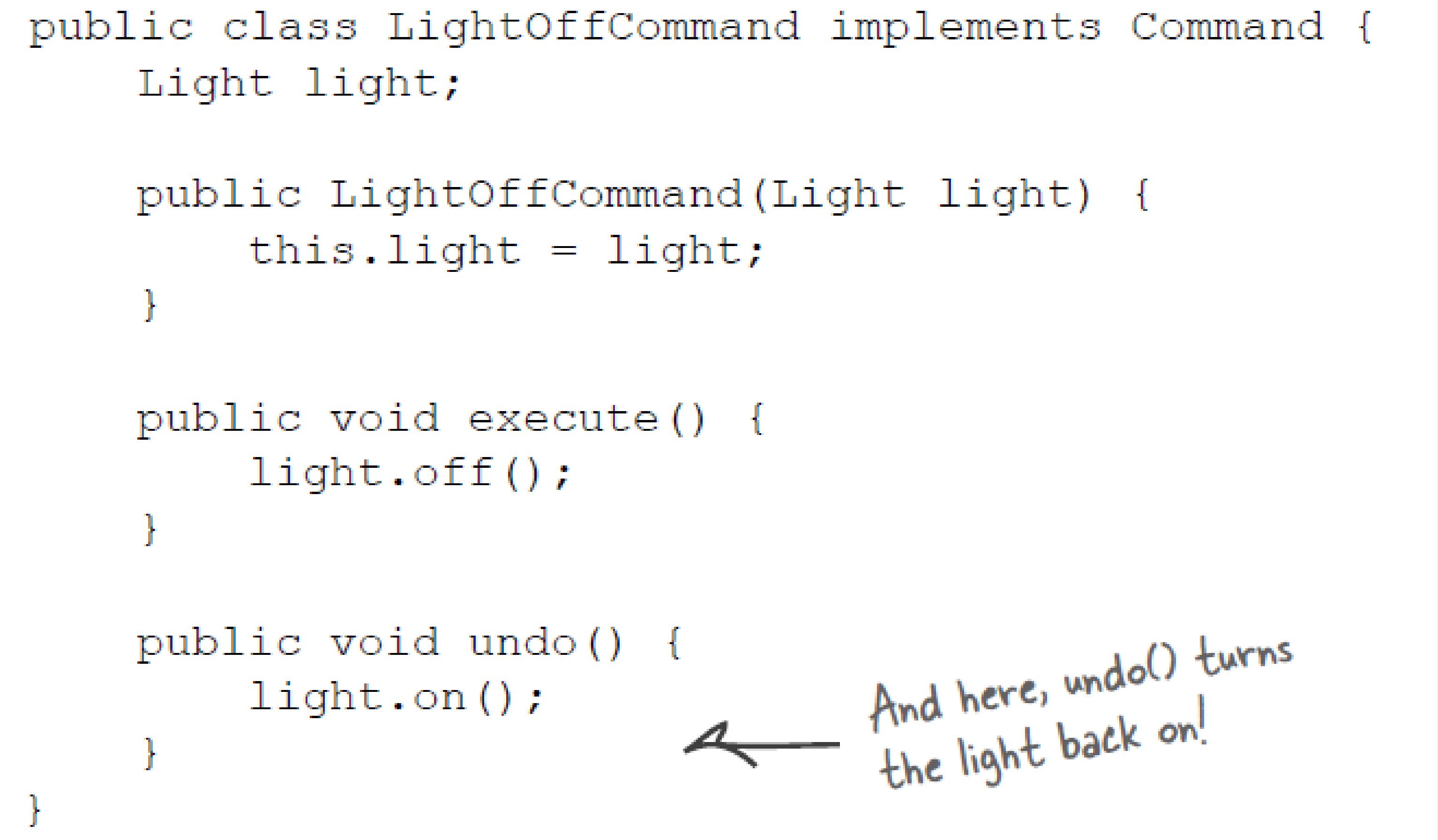
Implement the undo



```
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.

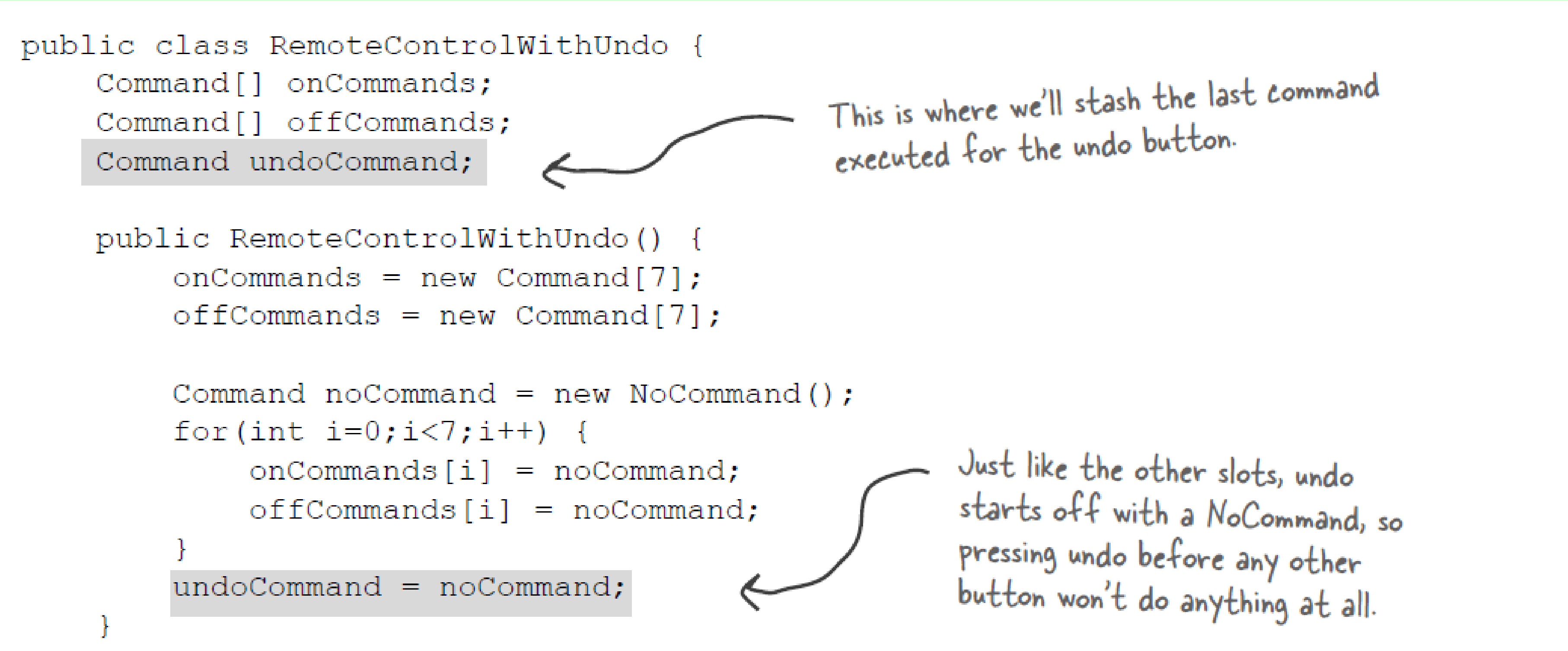
Implement the undo



```java
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!

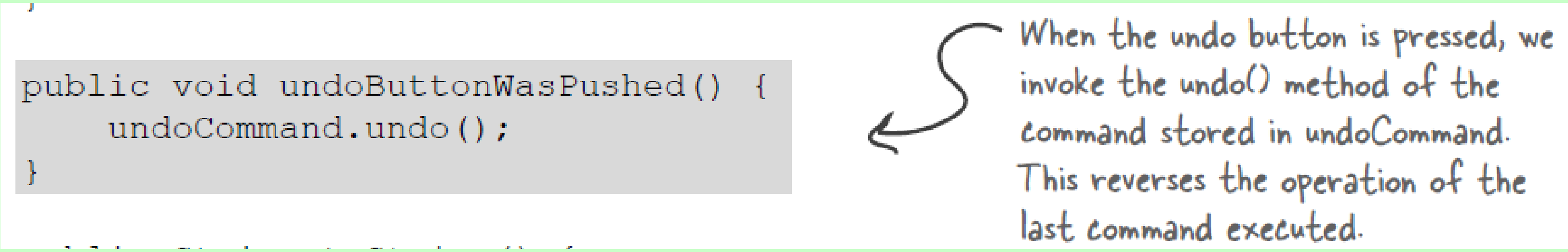
Remote Control with Undo



```
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;
    }
```

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

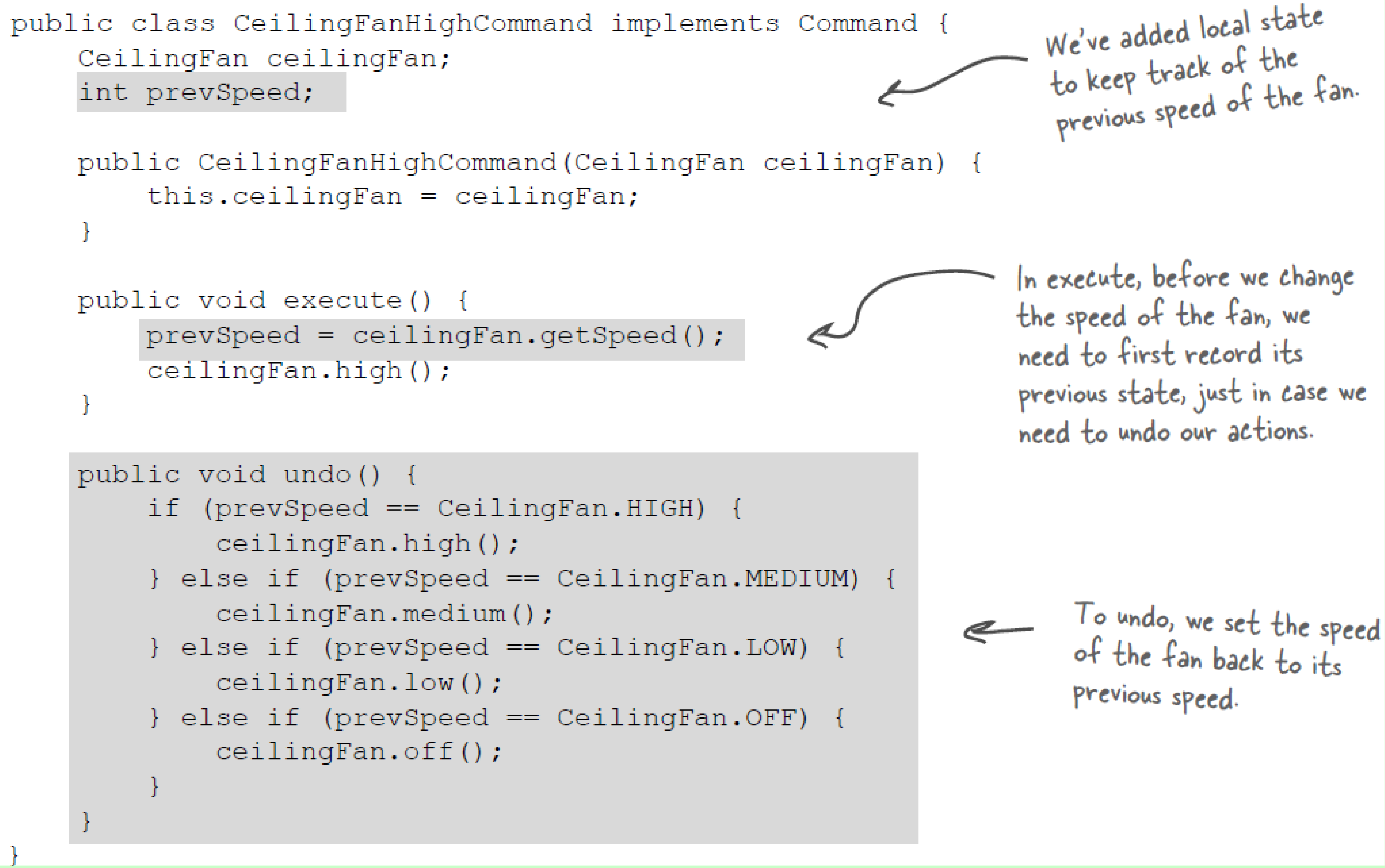
Just like the other slots, undo starts off with a NoCommand, so pressing undo before any other button won't do anything at all.



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

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.

Adding Undo to the ceiling fan commands



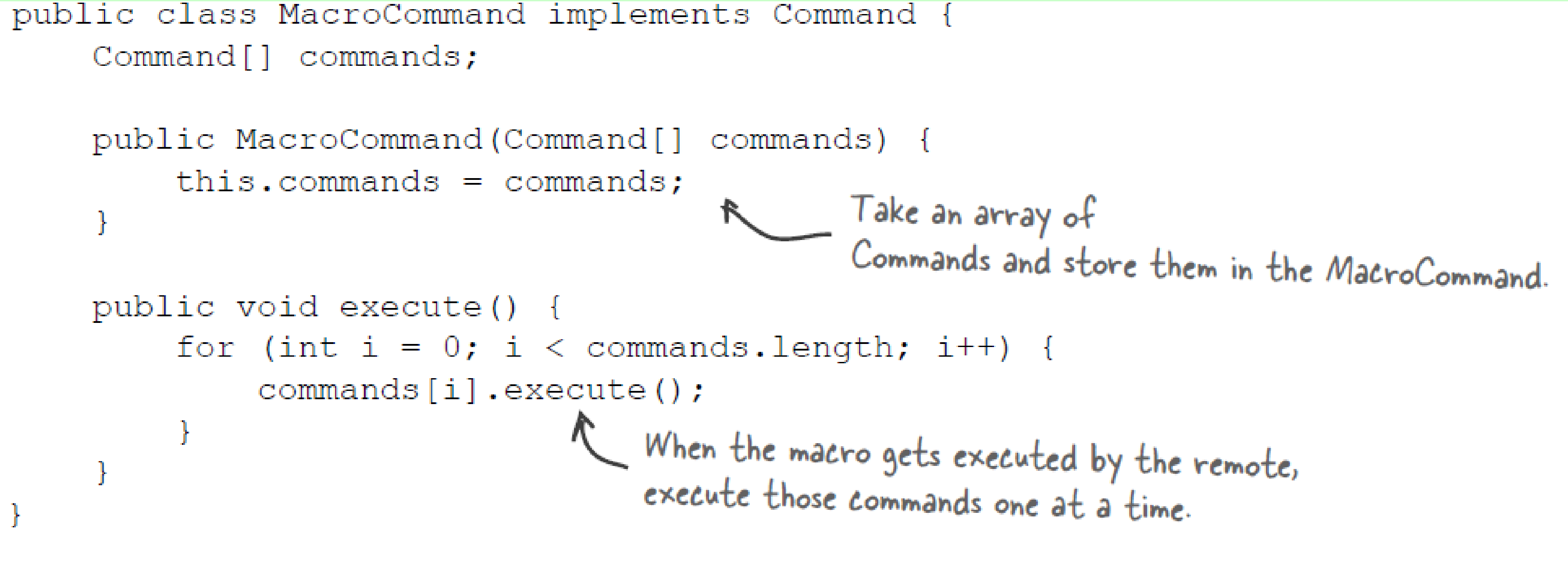
```
public class CeilingFanHighCommand implements Command {
    CeilingFan ceilingFan;
    int prevSpeed;

    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.

Adding Macro Command

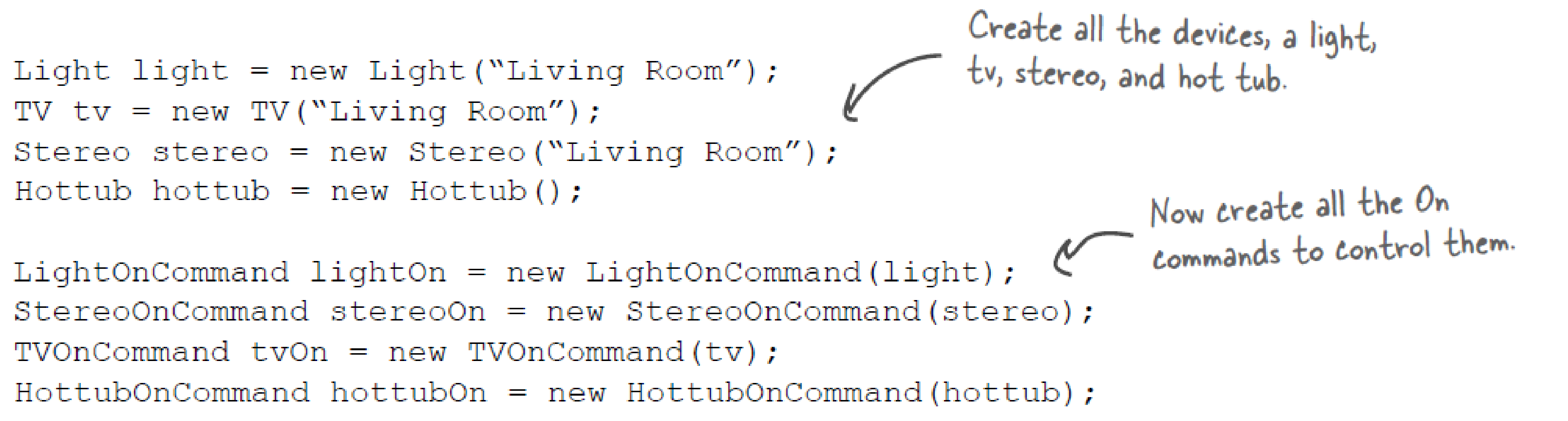


```
public class MacroCommand implements Command {
    Command[] commands;

    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.

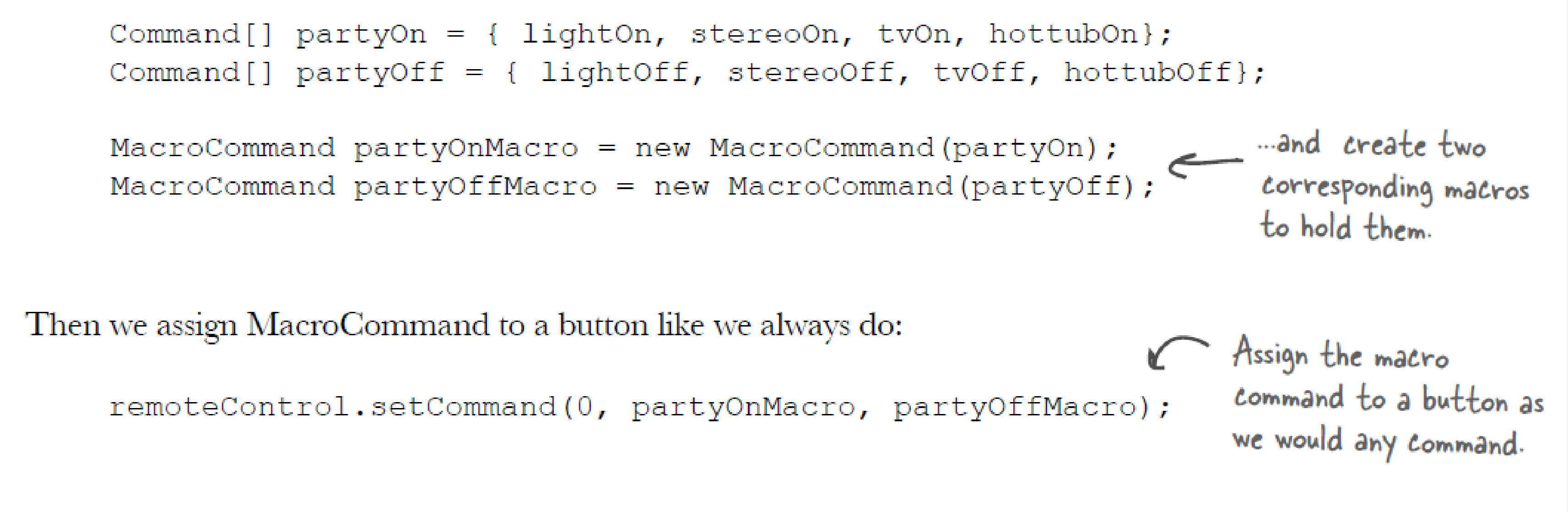
Adding Macro Command



```
Light light = new Light(“Living Room”);
TV tv = new TV(“Living Room”);
Stereo stereo = new Stereo(“Living Room”);
Hottub hottub = new Hottub();

LightOnCommand lightOn = new LightOnCommand(light);
StereoOnCommand stereoOn = new StereoOnCommand(stereo);
TVOnCommand tvOn = new TVOnCommand(tv);
HottubOnCommand hottubOn = new HottubOnCommand(hottub);
```

Create all the devices, a light, tv, stereo, and hot tub.

Now create all the On commands to control them.



```
Command[] partyOn = { lightOn, stereoOn, tvOn, hottubOn};
Command[] partyOff = { lightOff, stereoOff, tvOff, hottubOff};

MacroCommand partyOnMacro = new MacroCommand(partyOn);
MacroCommand partyOffMacro = new MacroCommand(partyOff);
```

...and create two corresponding macros to hold them.

Then we assign MacroCommand to a button like we always do:

```
remoteControl.setCommand(0, partyOnMacro, partyOffMacro);
```

Assign the macro command to a button as we would any command.

More uses of the Command Pattern: queuing requests

- Commands give us a way to package a piece of computation (a receiver and a set of actions) and pass it around as a first-class object.

- Now, the computation itself may be invoked long after some client application creates the command object.

- In fact, it may even be invoked by a different thread.

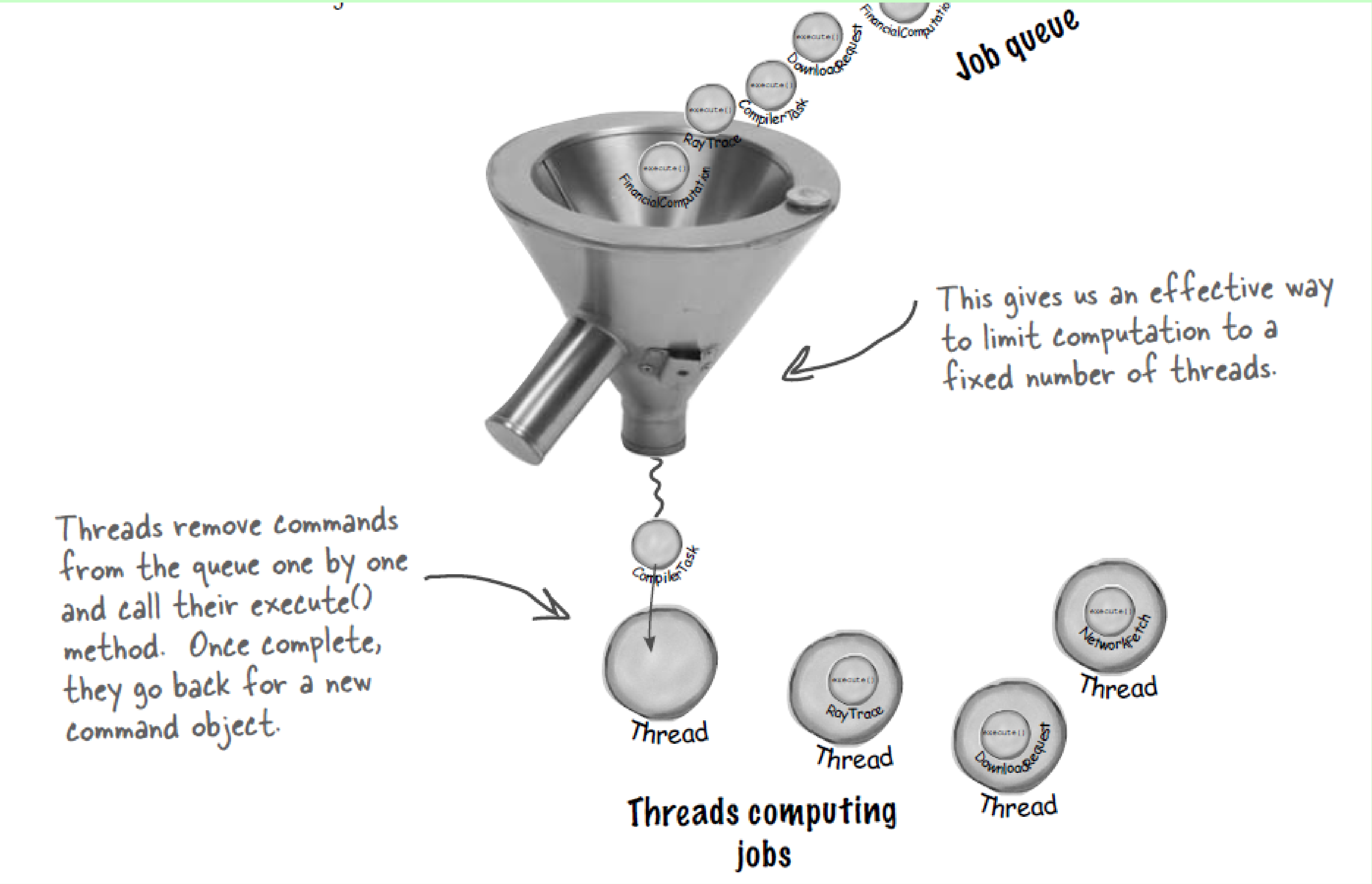
More uses of the Command Pattern: queuing requests

- Imagine a job queue: you add commands to the queue on one end, and on the other end sit a group of threads.

- Threads run the following script: they remove a command from the queue, call its execute() method, wait for the call to finish,

- Then discard the command object and retrieve a new one.

More uses of the Command Pattern: queuing requests

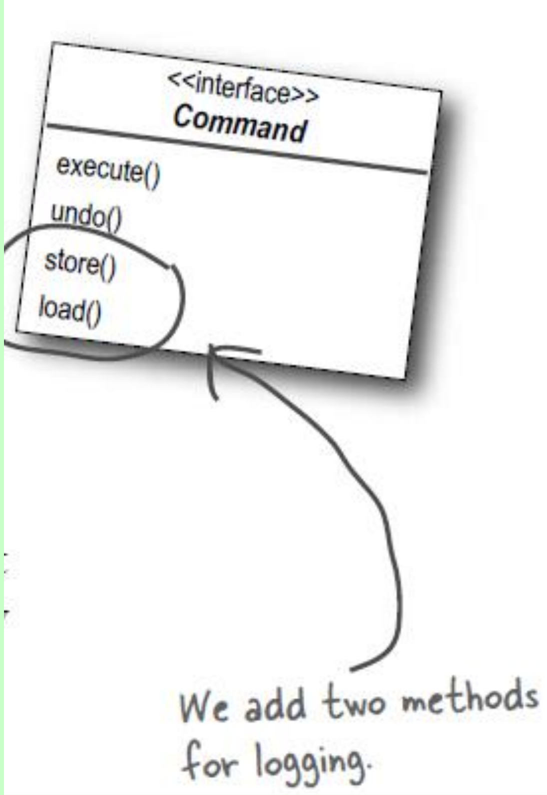


This gives us an effective way to limit computation to a fixed number of threads.

Threads remove commands from the queue one by one and call their execute() method. Once complete, they go back for a new command object.

More uses of the Command Pattern: logging requests

- The semantics of some applications require that we log all actions and be able to recover after a crash by reinvoking those actions.
- As we execute commands, we store a history of them on disk. When a crash occurs, we reload the command objects and invoke their execute() methods in batch and in order.
- we might want to implement our failure recovery by logging the actions on the spreadsheet rather than writing a copy of the spreadsheet to disk every time a change occurs.



We add two methods for logging.

References

- Design Patterns: Elements of Reusable Object-Oriented Software By Gamma, Erich; Richard Helm, Ralph Johnson, and John Vlissides (1995). Addison-Wesley. ISBN 0-201-63361-2.
- **Head First Design Patterns** By Eric Freeman, Elisabeth Freeman, Kathy Sierra, Bert Bates
  First Edition October 2004
  ISBN 10: 0-596-00712-4
- http://www.csee.wvu.edu/classes/cs210/Fall_2006/CS_210_Sept_26.ppt