CS525 Advanced Software Development

Lesson 6 – The Command Pattern

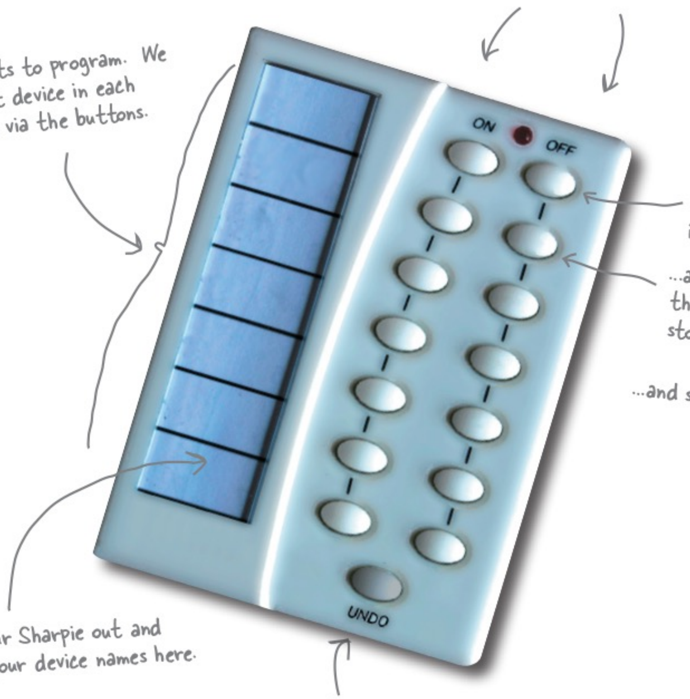
Design Patterns
Elements of Reusable Object-Oriented Software

Payman Salek, M.S.
March 2022

Introduction

In this chapter, we take encapsulation to a whole new level: we’re going to encapsulate method invocation.

Setting the stage (The Remote)



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.

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...

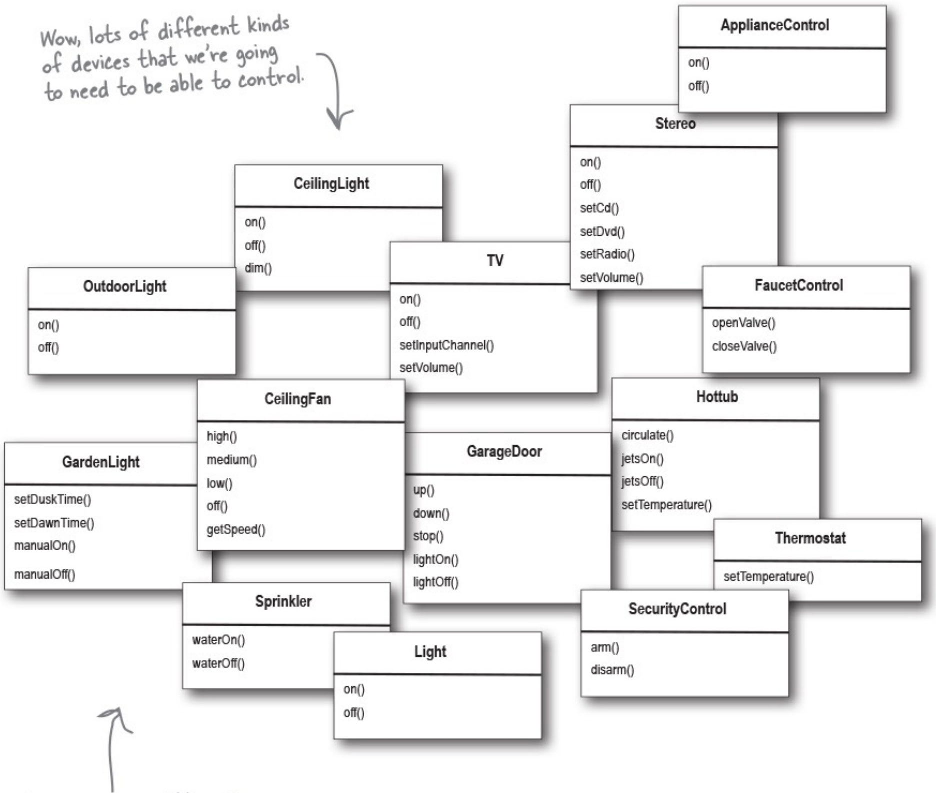
...and so on.

Get your Sharpie out and write your device names here.

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

Setting the stage (vendor classes)

Wow, lots of different kinds of devices that we're going to need to be able to control.



ApplianceControl
on()
off()

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

CeilingLight
on()
off()
dim()

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

OutdoorLight
on()
off()

FaucetControl
openValve()
closeValve()

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

Hottub
circulate()
jetsOn()
jetsOff()
setTemperature()

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

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

Thermostat
setTemperature()

Sprinkler
waterOn()
waterOff()

SecurityControl
arm()
disarm()

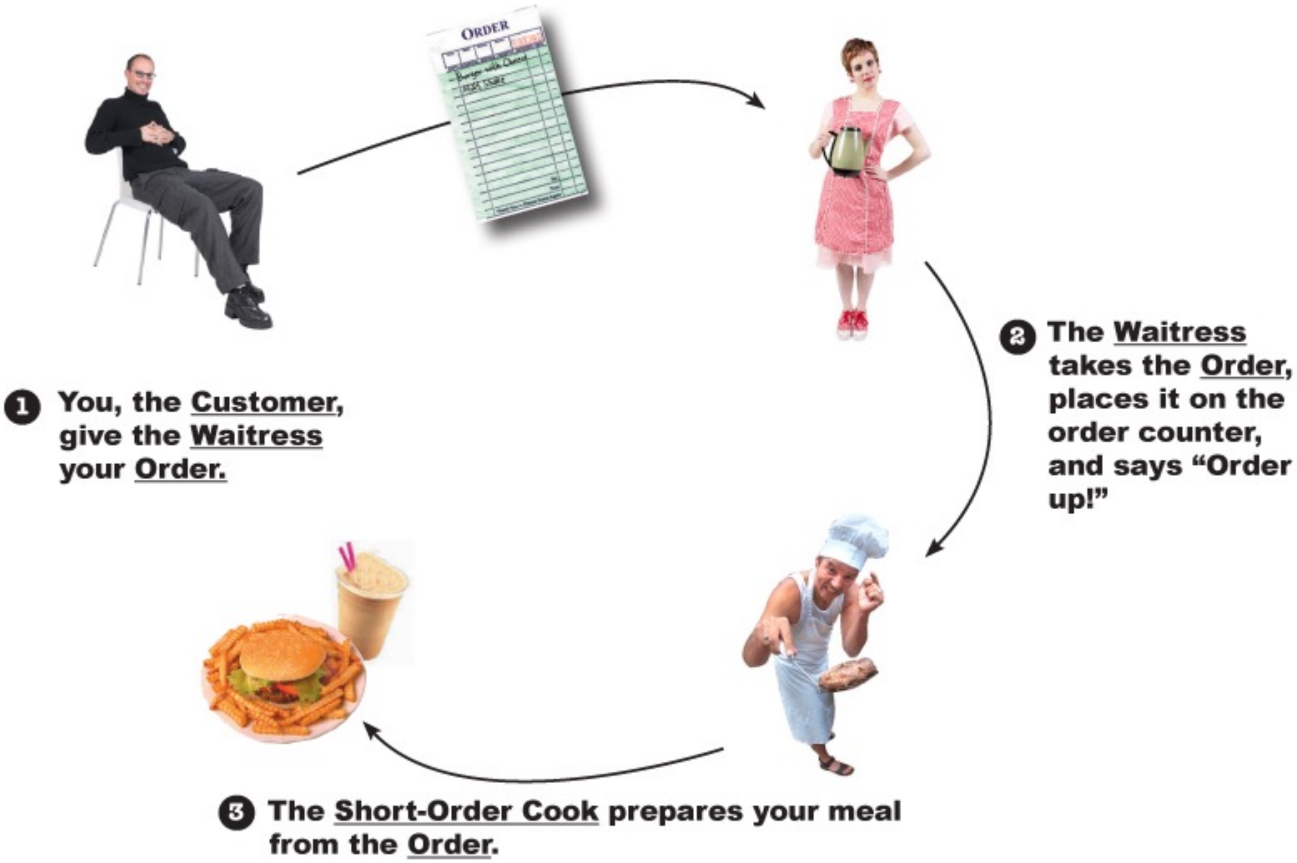
Light
on()
off()

And some very different kinds of interfaces across these devices.

Solution: The Command

The Command Pattern allows you 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 your vendor classes.

The Command Pattern

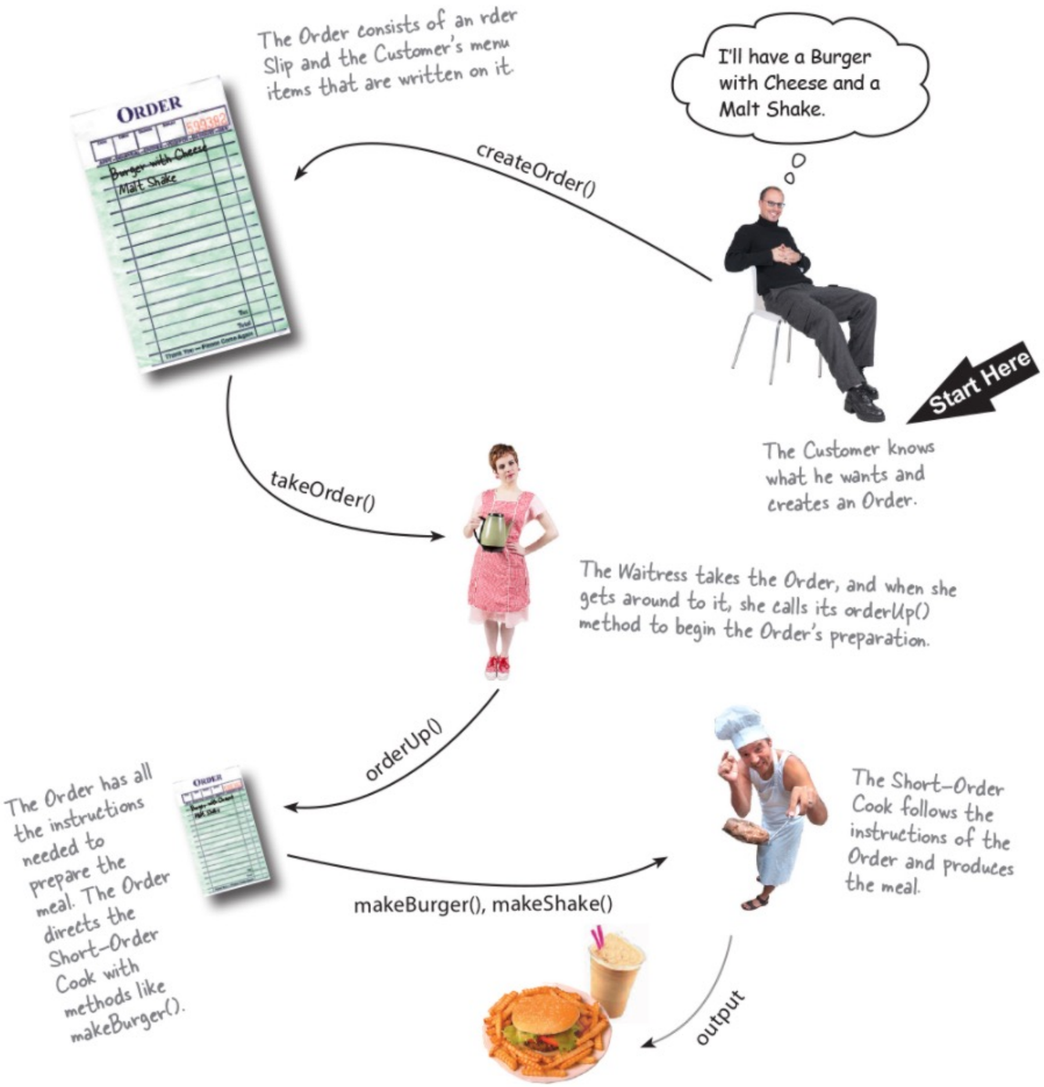


1 You, the Customer, give the Waitress your Order.

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 Command Pattern Explained



The Order consists of an rder Slip and the Customer's menu items that are written on it.

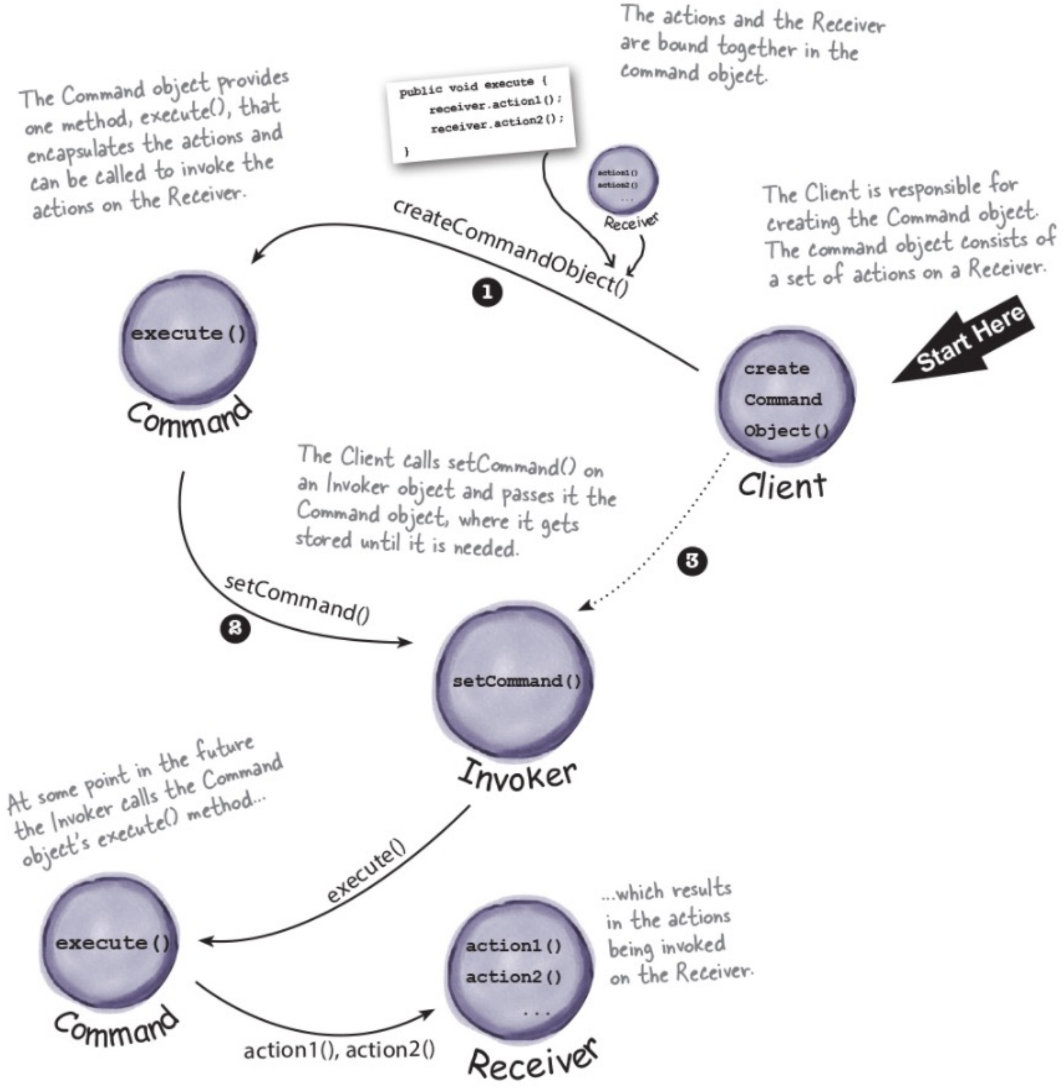
The Customer knows what he wants and creates an Order.

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.

The Command Pattern Abstraction



```
public void execute {
    receiver.action1();
    receiver.action2();

}
```

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.

The Client is responsible for creating the Command object. The command object consists of a set of actions on a Receiver.

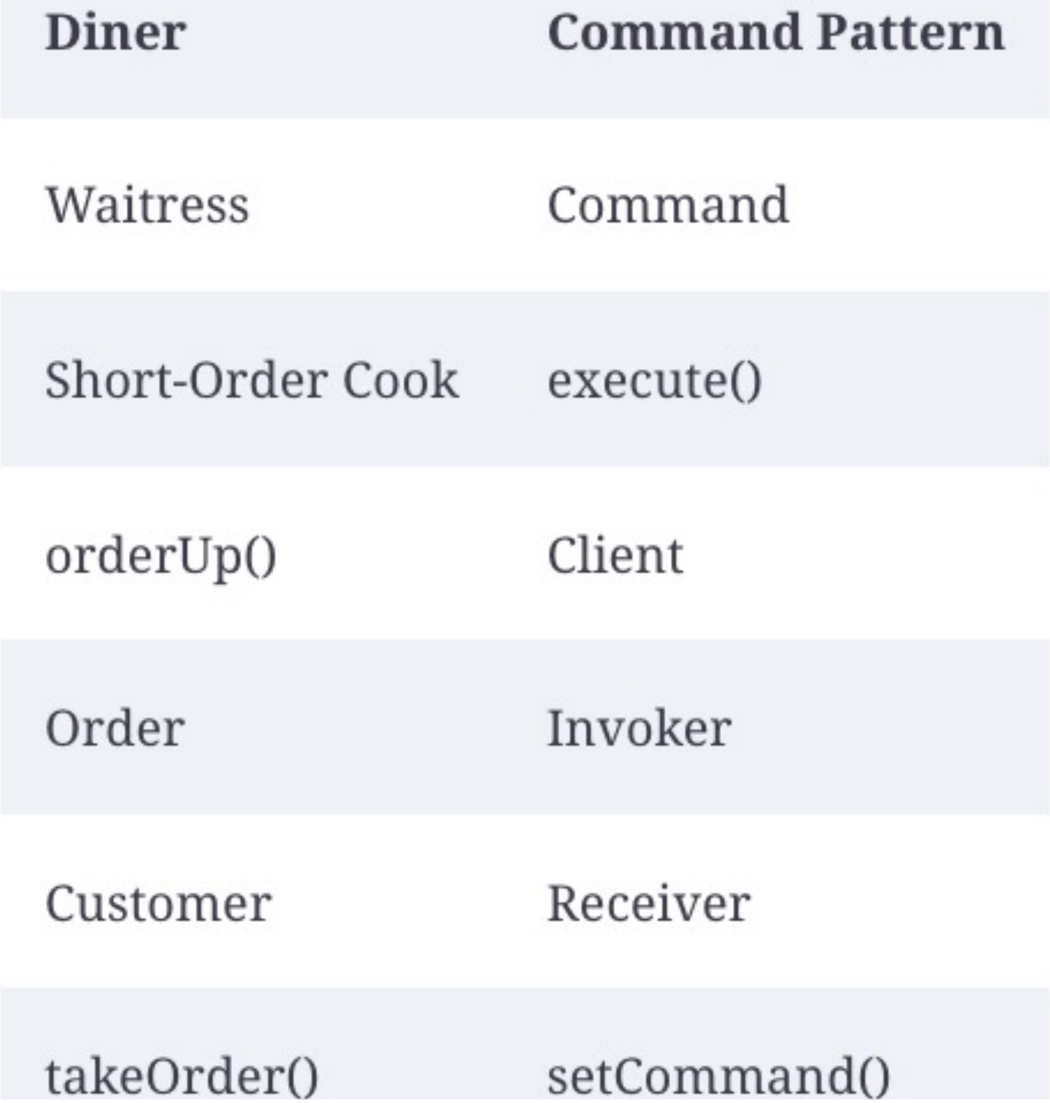
The Client calls setCommand() on an Invoker object and passes it the Command object, where it gets stored until it is needed.

At some point in the future the Invoker calls the Command object's execute() method...

...which results in the actions being invoked on the Receiver.

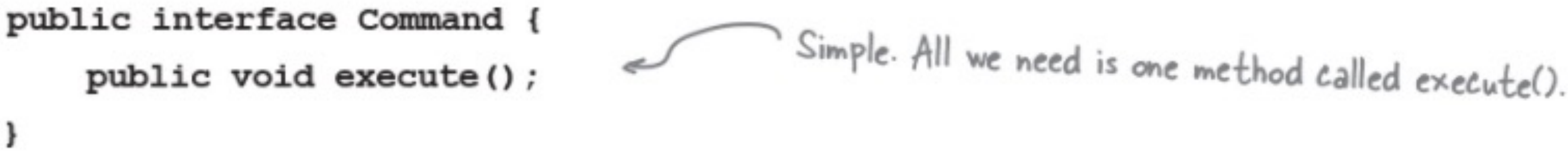
2/27/22

Applying the commend pattern to the diner



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

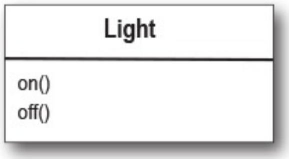
The Command Interface

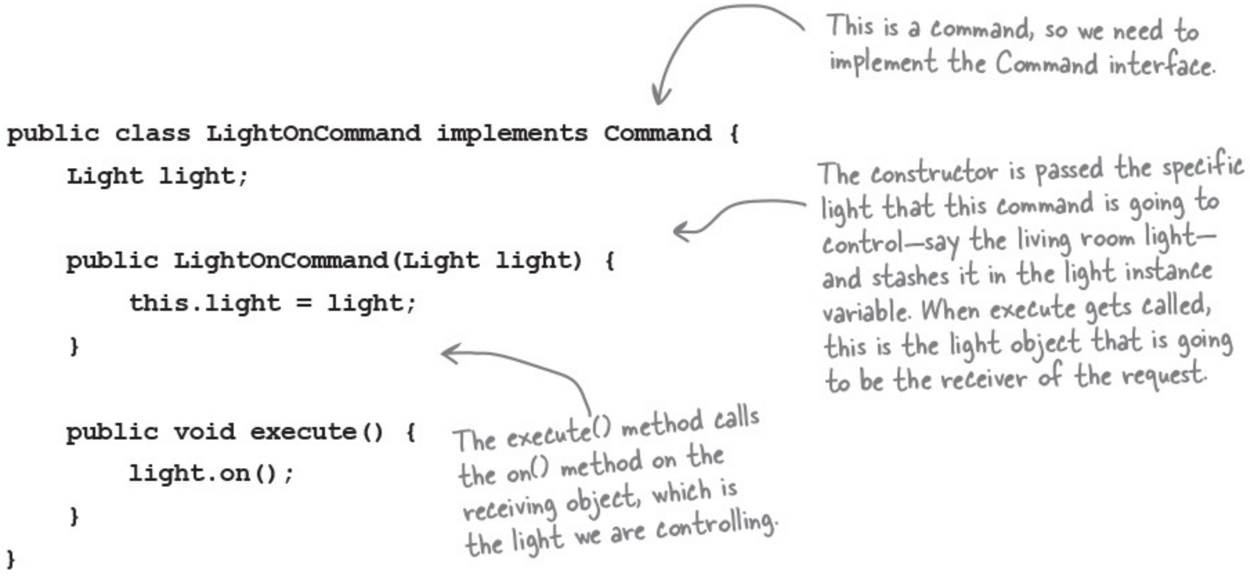


```
public interface Command {
    public void execute();
}
```

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

Applying the Command Pattern





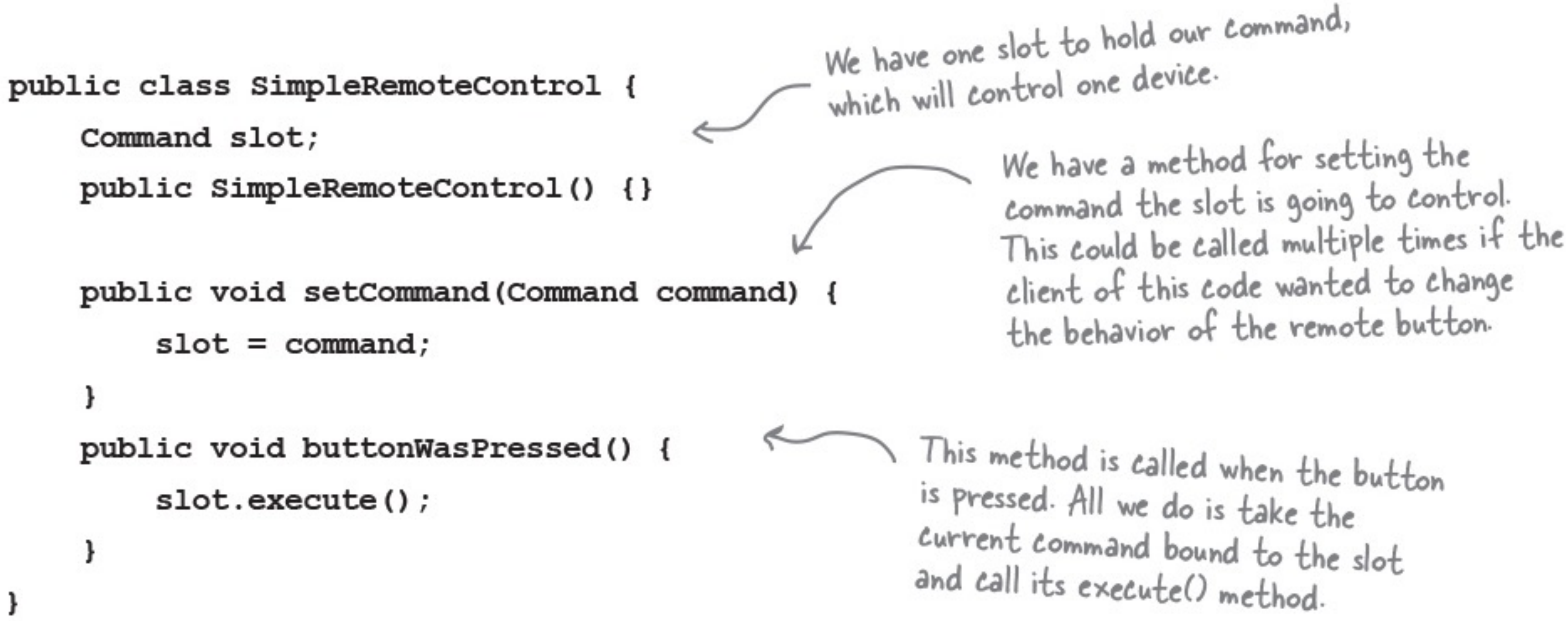
```
public class LightOnCommand implements Command {
    Light light;

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

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

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

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.

Applying the Command Pattern



```
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.

Applying the Command Pattern

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.

```
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();
    }
}
```

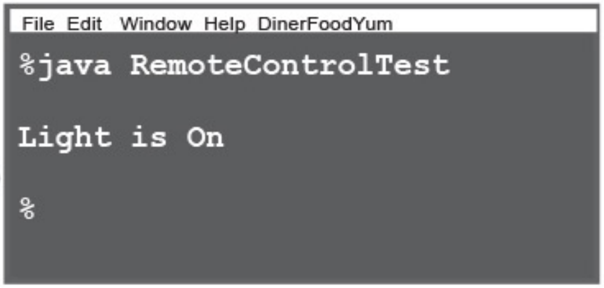
Now we create a Light object. This will be the Receiver of the request.

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

Here, pass the command to the Invoker.

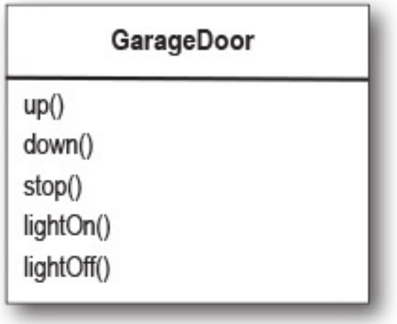
And then we simulate the button being pressed.

Here's the output of running this test code.



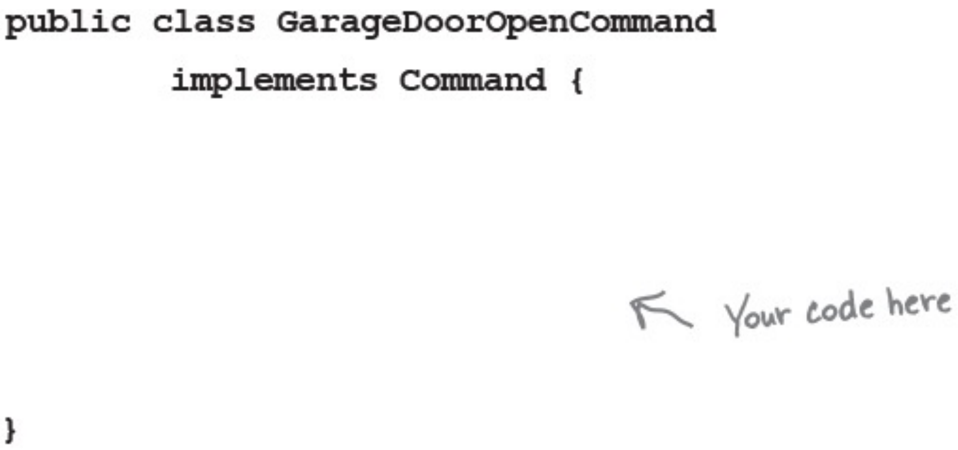
```
%java RemoteControlTest

Light is On

%
```

Practice: Apply the pattern



GarageDoor

up()
down()
stop()
lightOn()
lightOff()



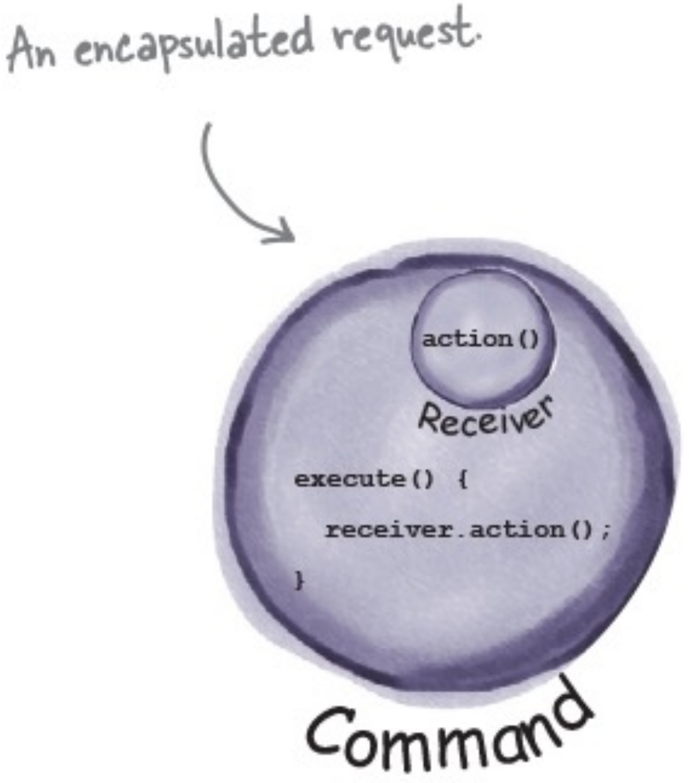
```
public class GarageDoorOpenCommand
        implements Command {




}
```

↖ Your code here

The Command Pattern Defined

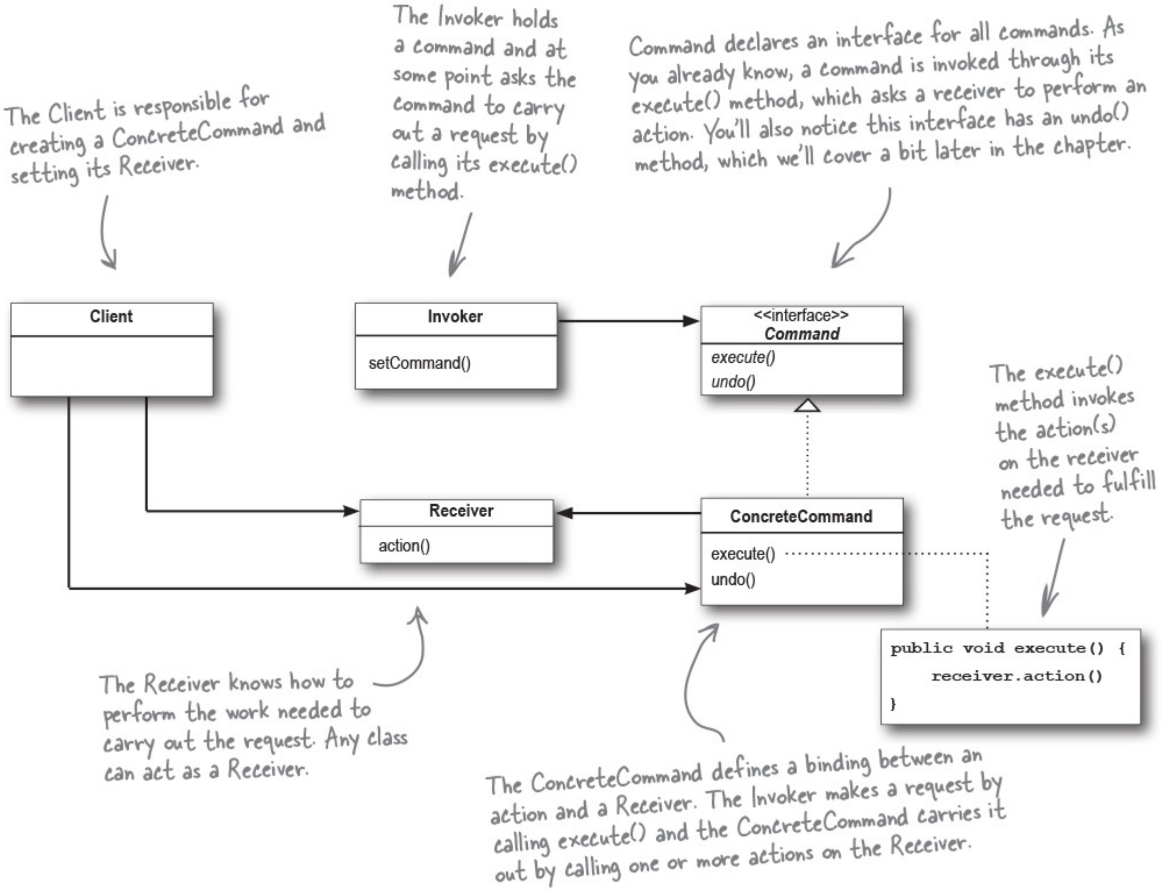
encapsulates a request as an object, thereby letting you parameterize other objects with different requests, queue or log requests, and support undoable operations.

An Encapsulated Request



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

The Command Pattern UML



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

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

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

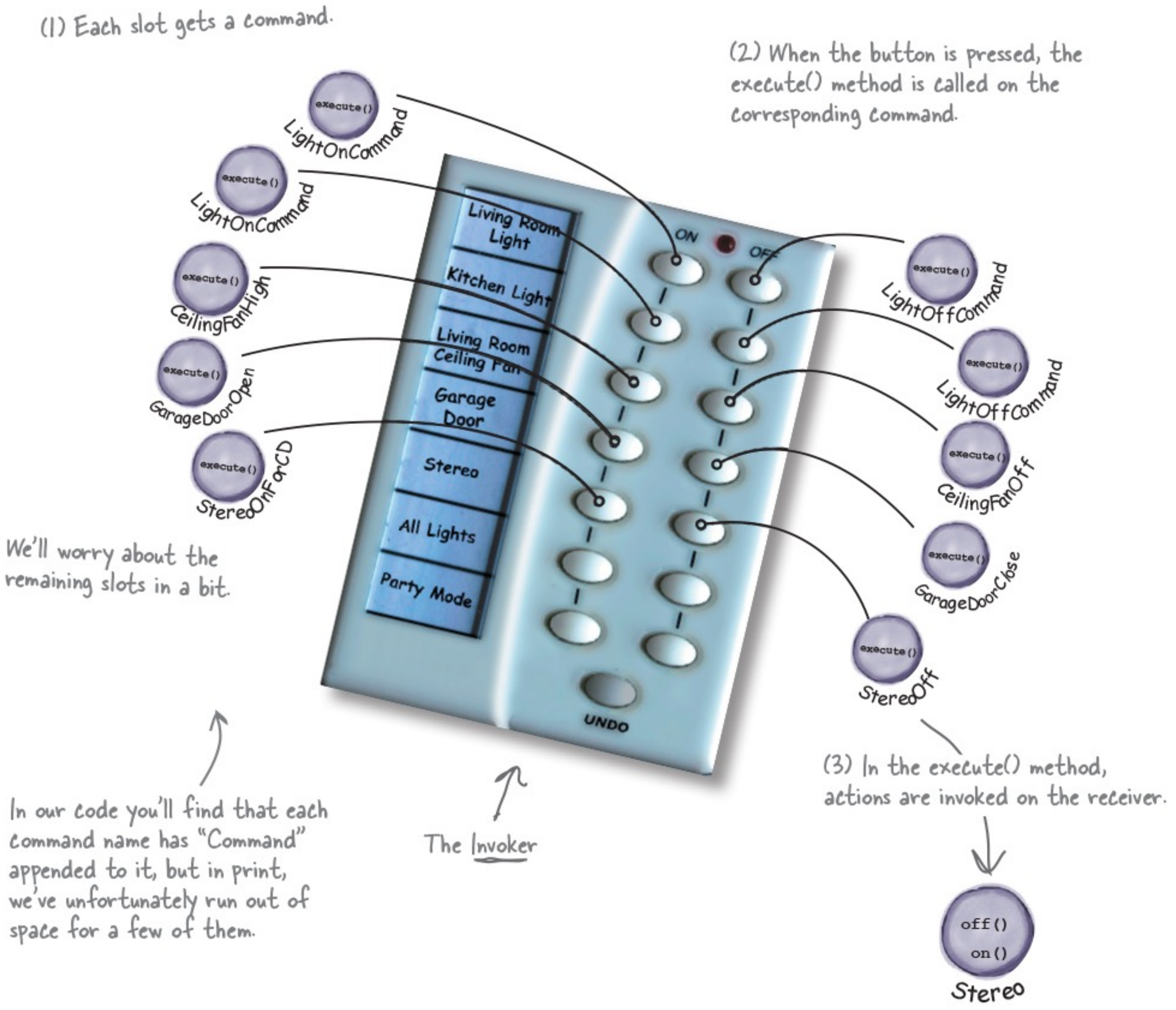
Command declares an interface for all commands. As you already know, a command is invoked through its execute() method, which asks a receiver to perform an action. You'll also notice this interface has an undo() method, which we'll cover a bit later in the chapter.

The 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.

Configuring the Remote



(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.

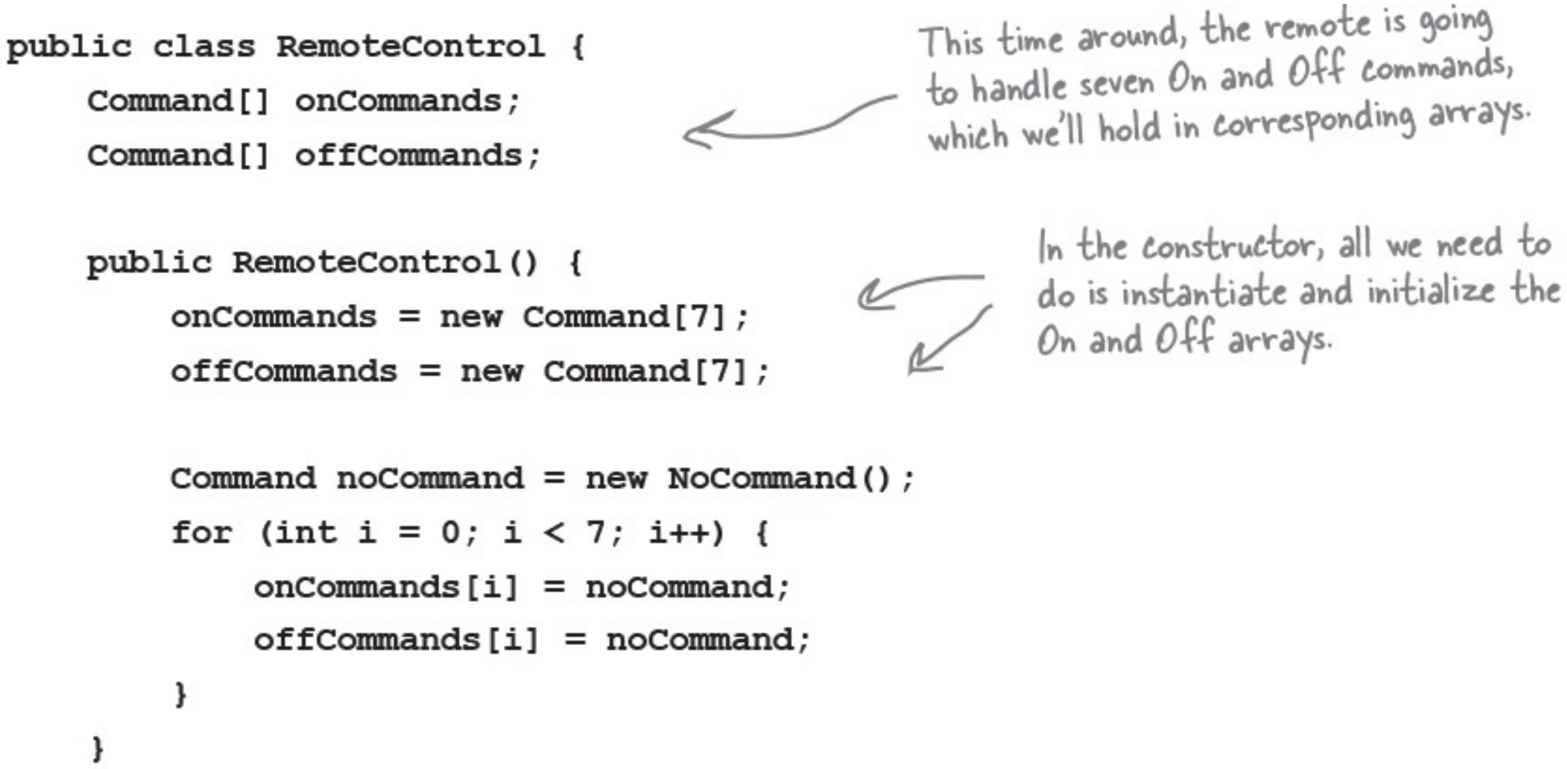
In our code you'll find that each command name has "Command" appended to it, but in print, we've unfortunately run out of space for a few of them.

The Invoker

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

2/27/22

The Remote Control

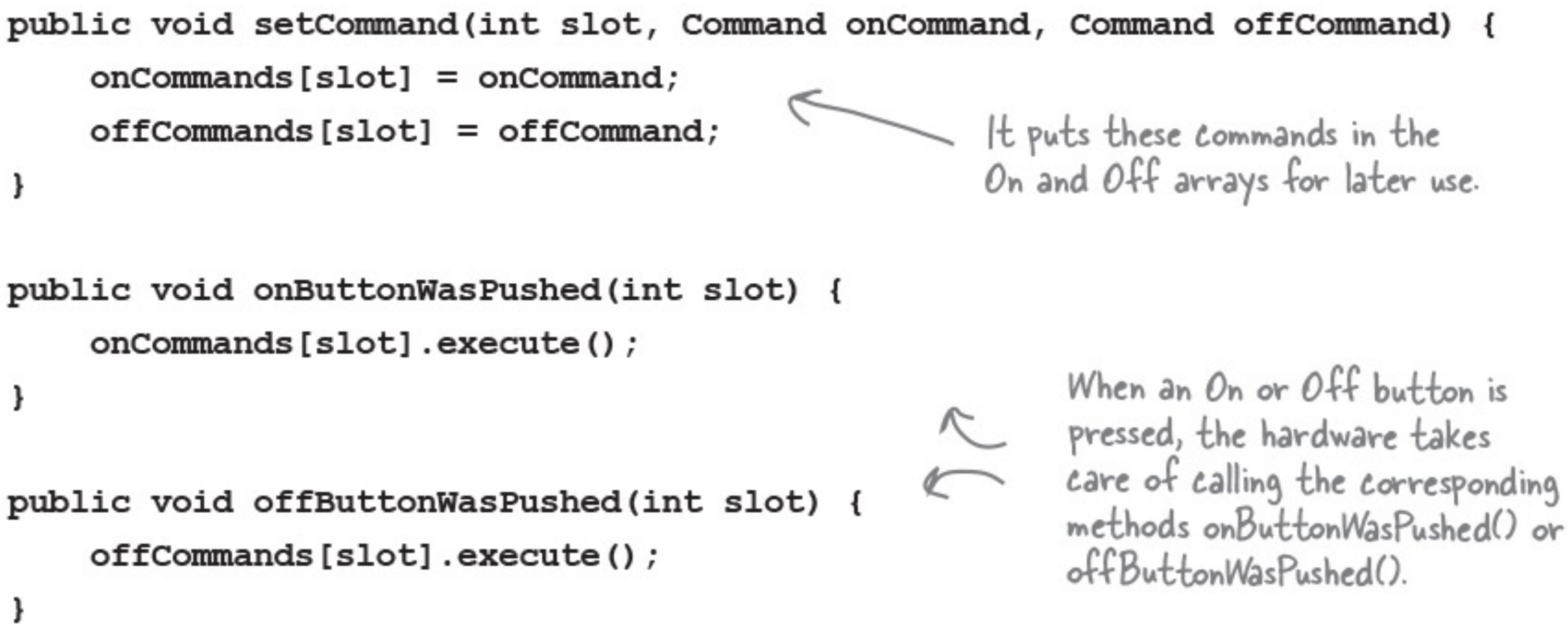


```
public class RemoteControl {
    Command[] onCommands;
    Command[] offCommands;

    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.

The Remote Control

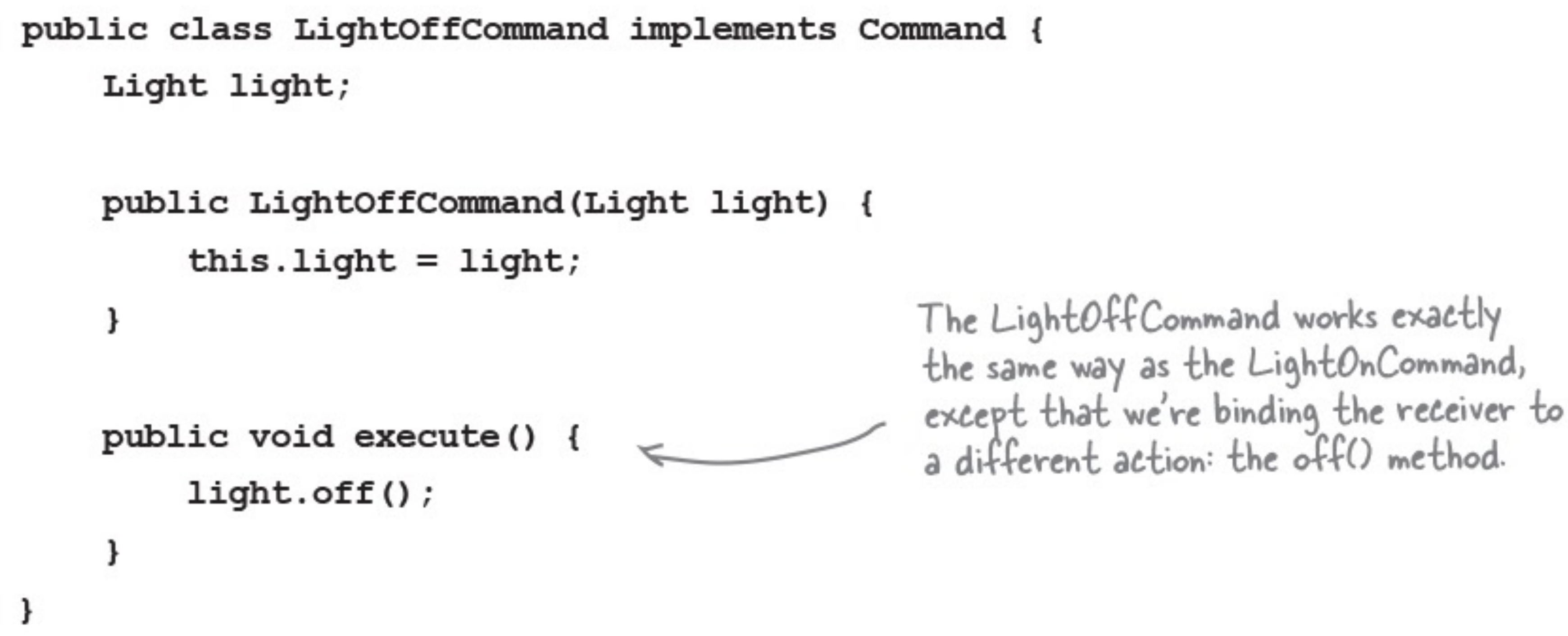


```
public void setCommand(int slot, Command onCommand, Command offCommand) {
    onCommands[slot] = onCommand;
    offCommands[slot] = offCommand;
}

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

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

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().

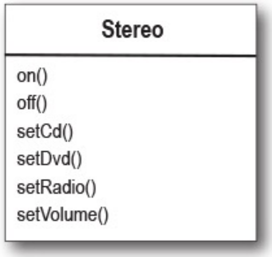
Implementing the Commands



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

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

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

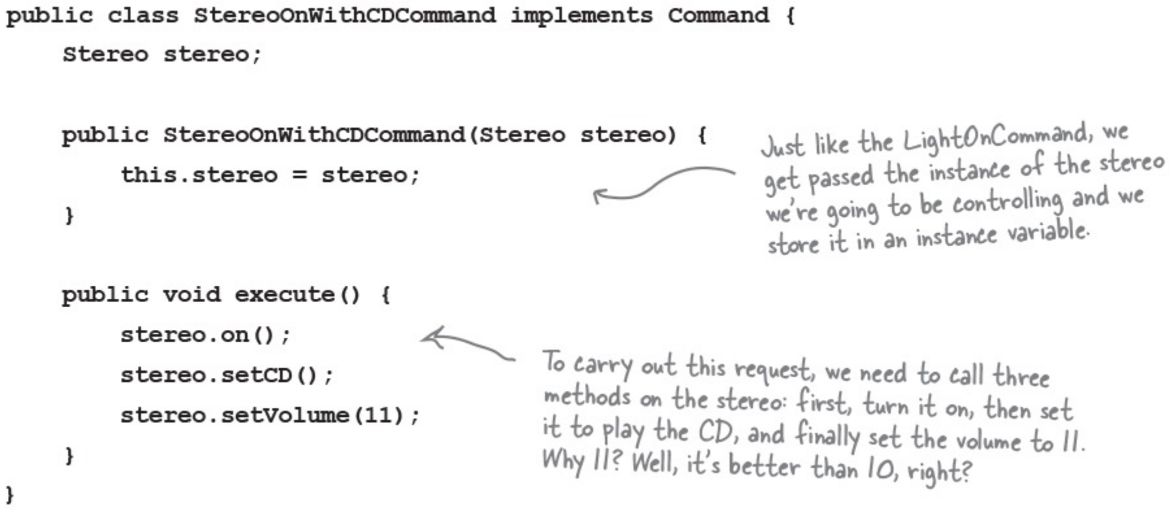
The LightOffCommand works exactly the same way as the LightOnCommand, except that we're binding the receiver to a different action: the off() method.

A more sophisticated Command



Stereo

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

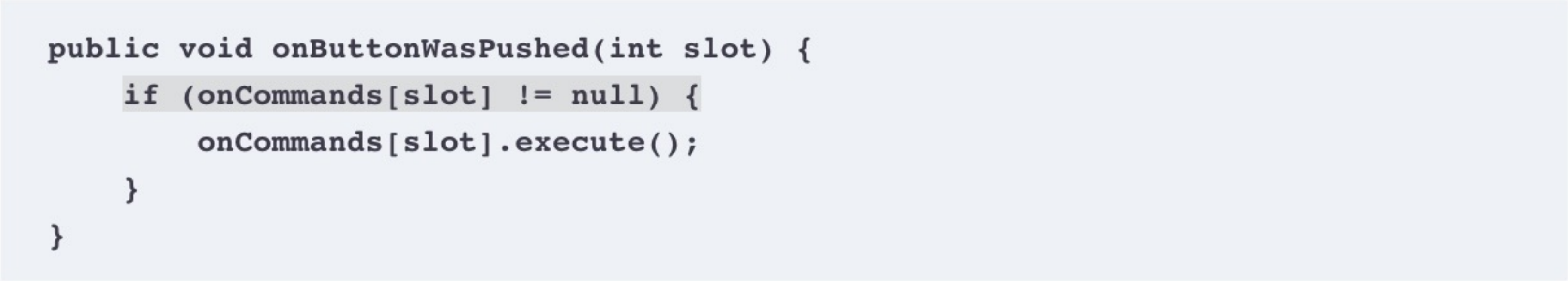


```
public class StereoOnWithCDCommand implements Command {
    Stereo stereo;

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

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

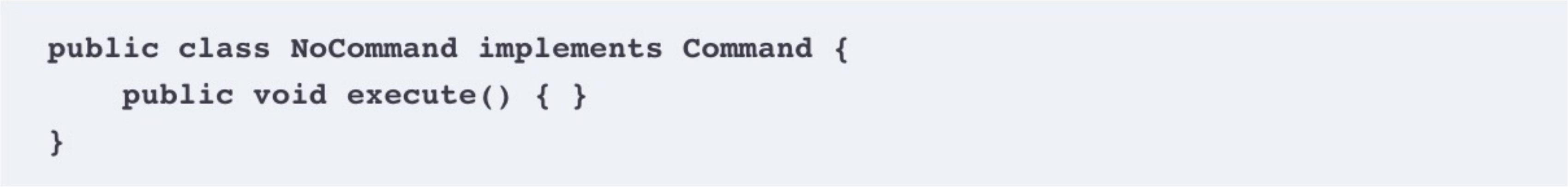
Just like the LightOnCommand, we get passed the instance of the stereo we're going to be controlling and we store it in an instance variable.

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?

The “NoCommand”



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

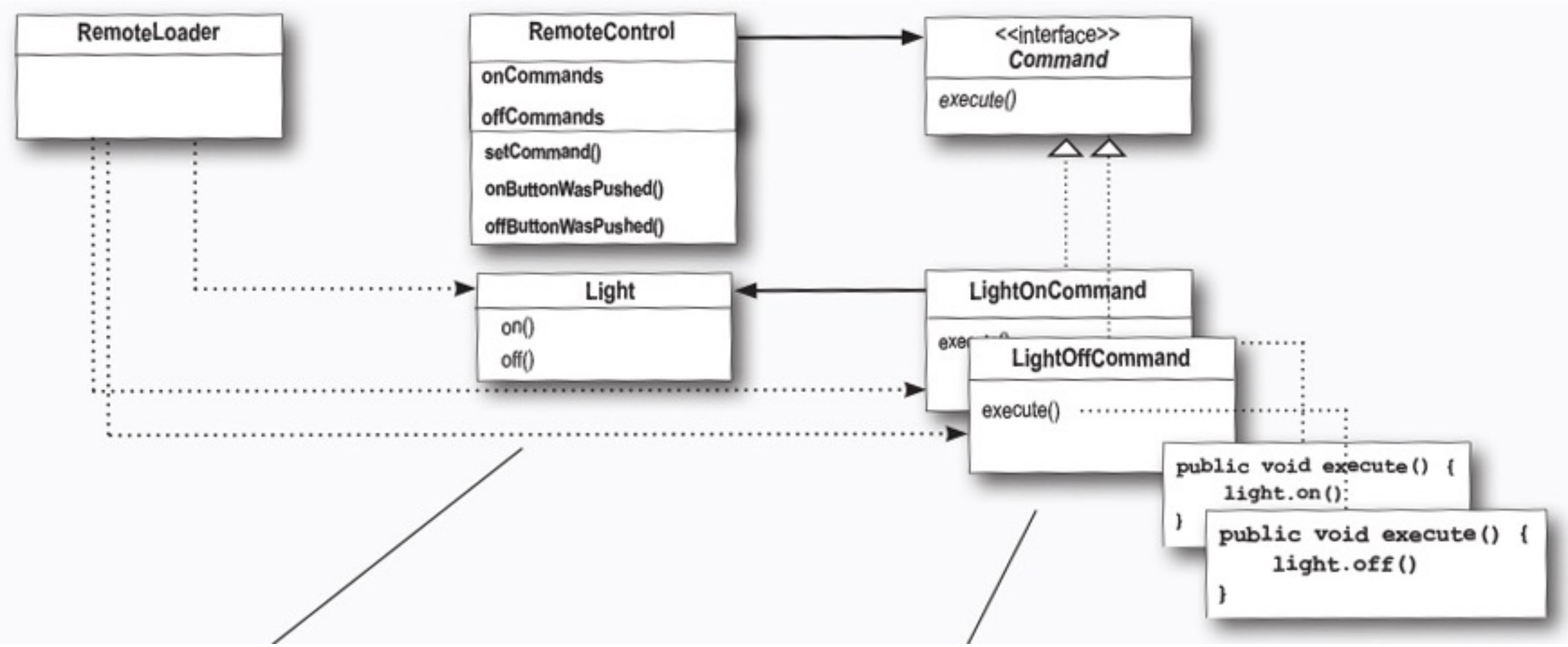


```
public class NoCommand implements Command {
    public void execute() { }
}
```

The Null Object

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.

The Reworked Remote



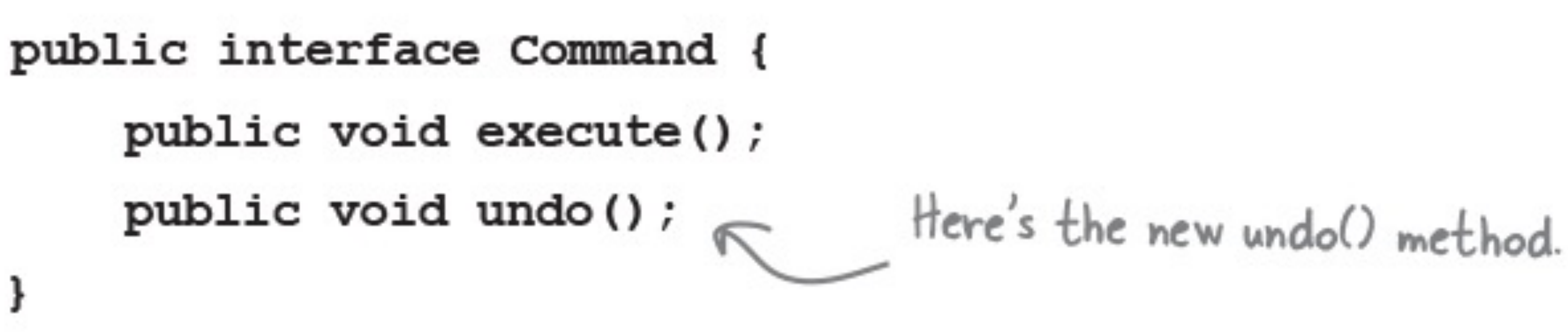
```
public void execute() {
    light.on();
}
```

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

The “Undo” Button

When commands support undo, they have an undo() method that mirrors the execute() method. Whatever execute() last did, undo() reverses.

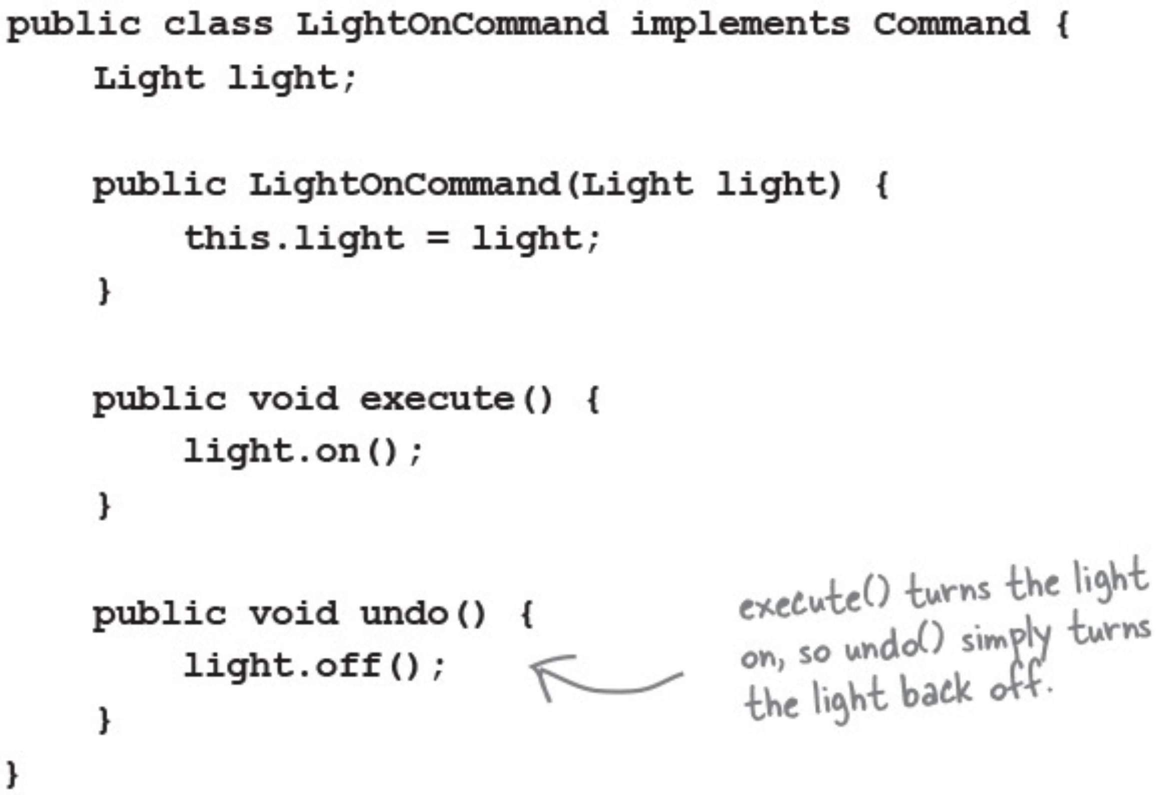
Adding Undo to the Command



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

Here's the new undo() method.

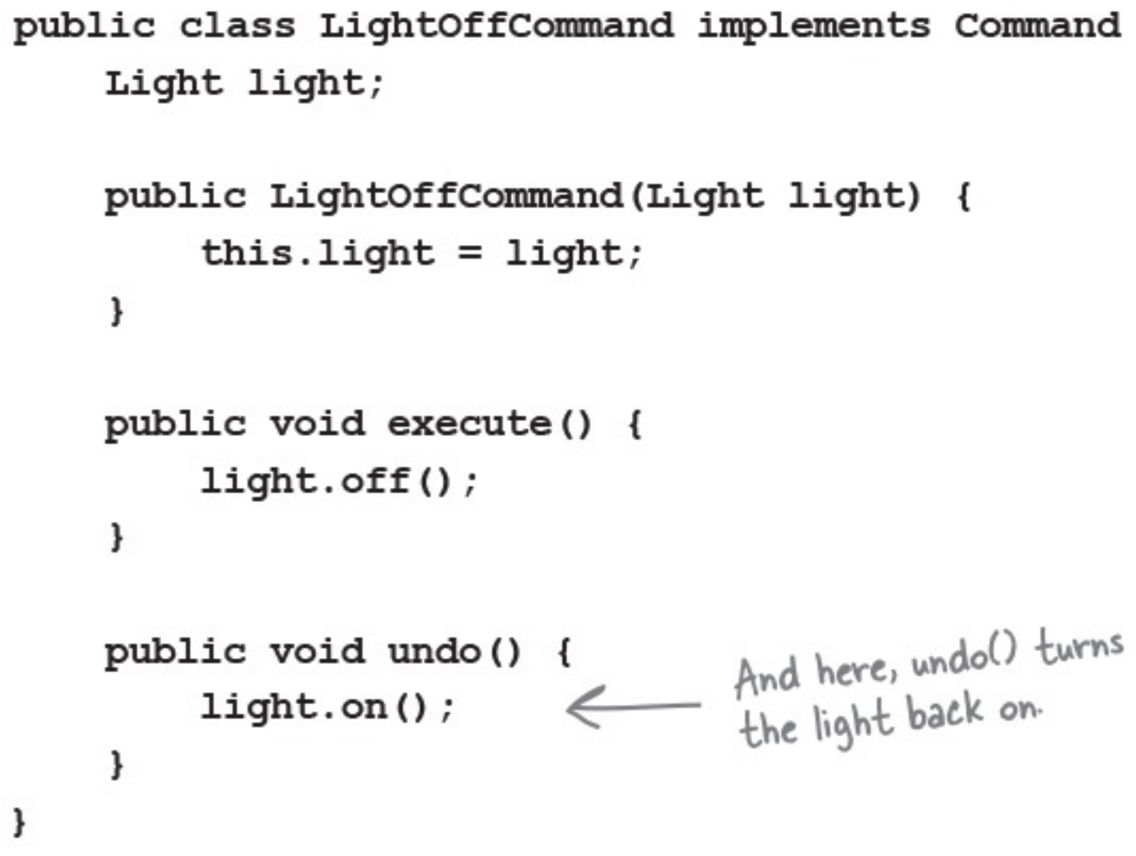
Applying Undo to LightOn



```
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.

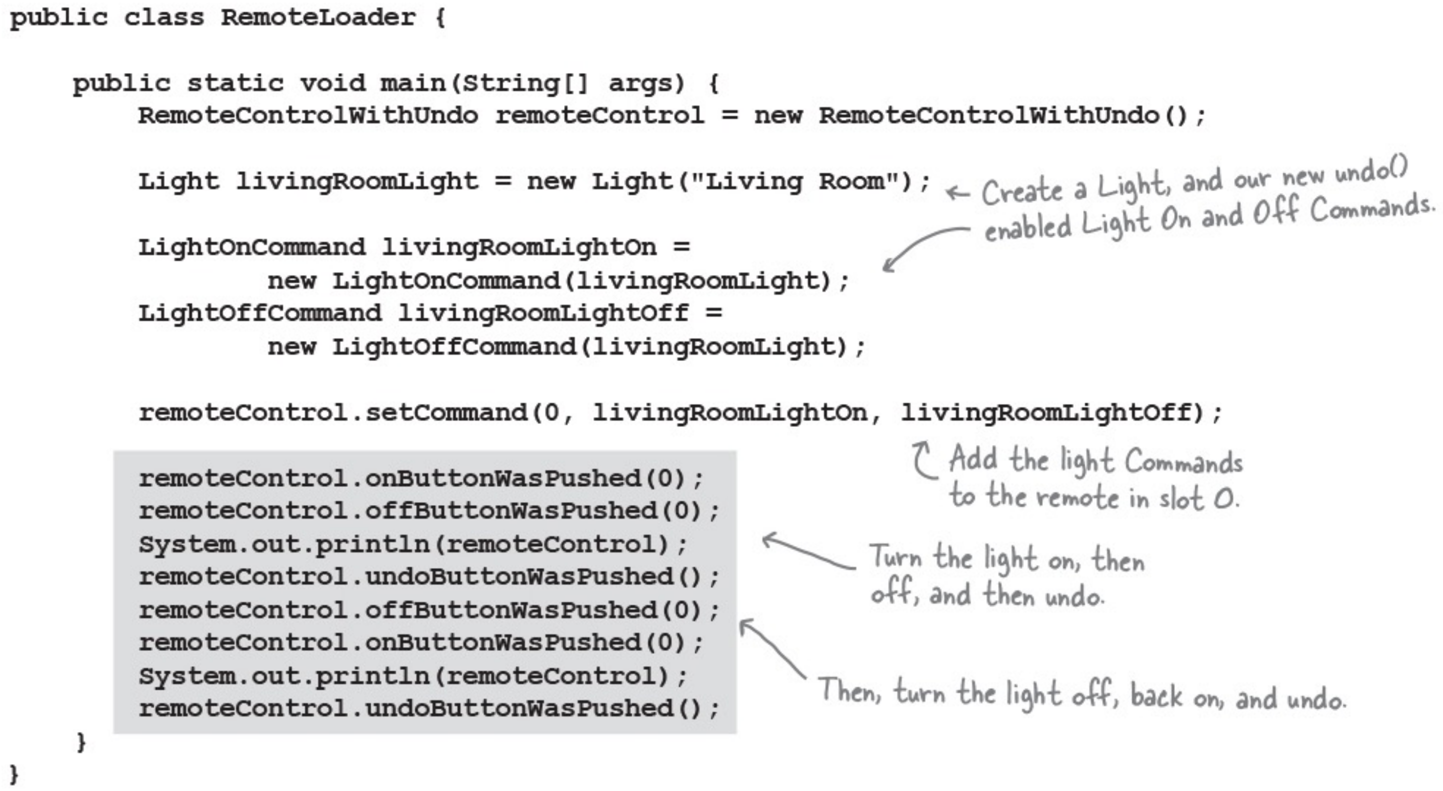
Applying Undo to LightOff



```
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.

The Remote Loader



```
public class RemoteLoader {

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

        Light livingRoomLight = new Light("Living Room");

        LightOnCommand livingRoomLightOn =
                new LightOnCommand(livingRoomLight);
        LightOffCommand livingRoomLightOff =
                new LightOffCommand(livingRoomLight);

        remoteControl.setCommand(0, livingRoomLightOn, livingRoomLightOff);

        remoteControl.onButtonWasPushed(0);
        remoteControl.offButtonWasPushed(0);
        System.out.println(remoteControl);
        remoteControl.undoButtonWasPushed();
        remoteControl.offButtonWasPushed(0);
        remoteControl.onButtonWasPushed(0);
        System.out.println(remoteControl);
        remoteControl.undoButtonWasPushed();
    }
}
```

Create a Light, and our new undo() enabled Light On and Off Commands.

Add the light Commands to the remote in slot 0.

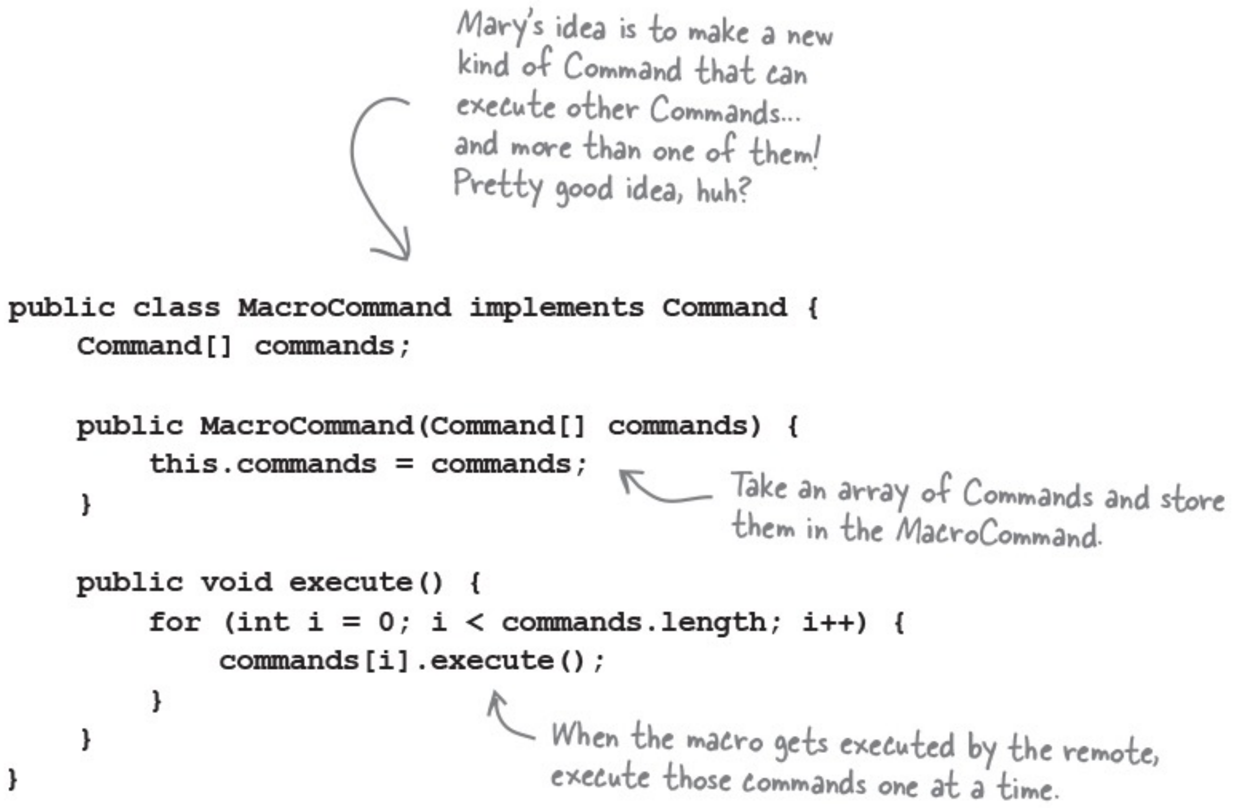
Turn the light on, then off, and then undo.

Then, turn the light off, back on, and undo.

The Party Mode

What’s the point of having a remote if you can’t push one button and have the lights dimmed, the stereo and TV turned on, and the hot tub fired up?

Macro Command



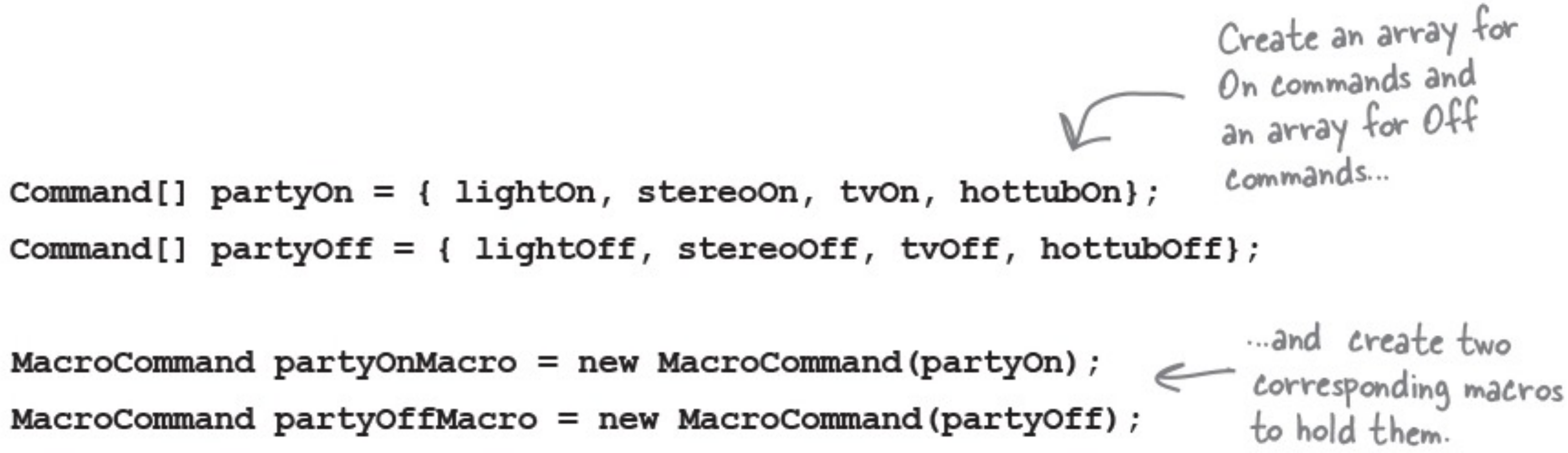
Mary's idea is to make a new kind of Command that can execute other Commands... and more than one of them! Pretty good idea, huh?

```
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.

Macro Command



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

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

Create an array for On commands and an array for Off commands...

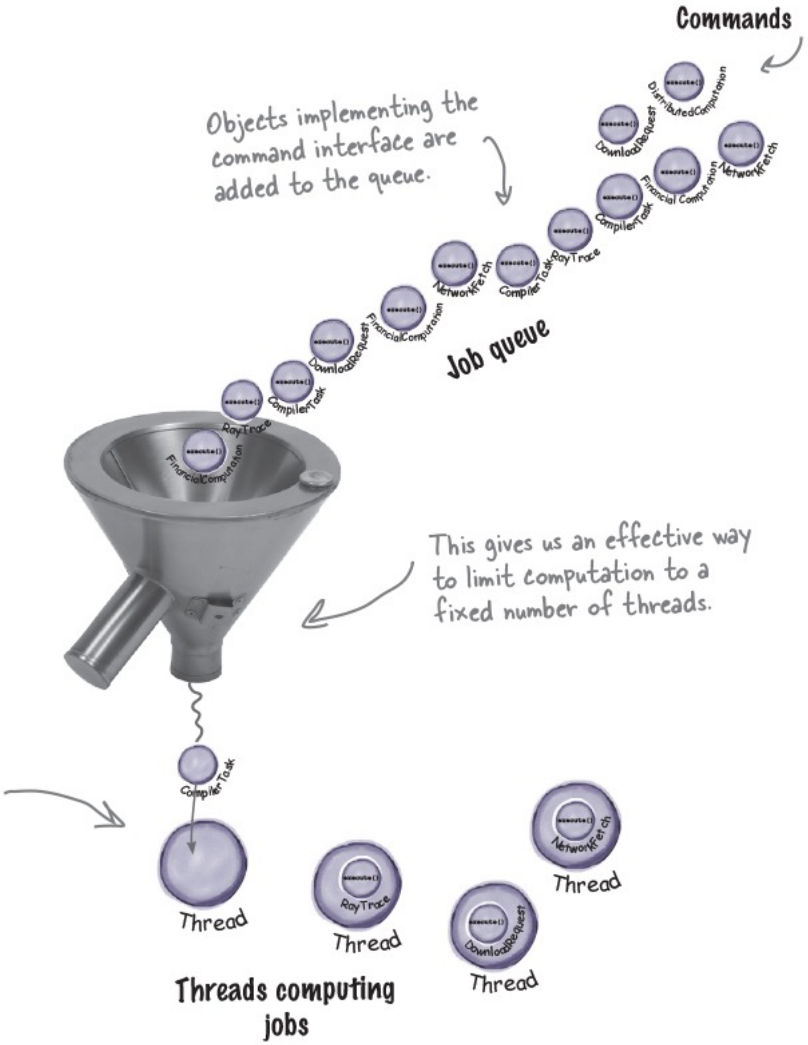
...and create two corresponding macros to hold them.

More uses of the Command

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. We can take this scenario and apply it to many useful applications, such as schedulers, thread pools, and job queues, to name a few.

The Job Queue

2/27/22



Objects implementing the command interface are added to the queue.

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.

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. The Command Pattern can support these semantics with the addition of two methods: store() and load().

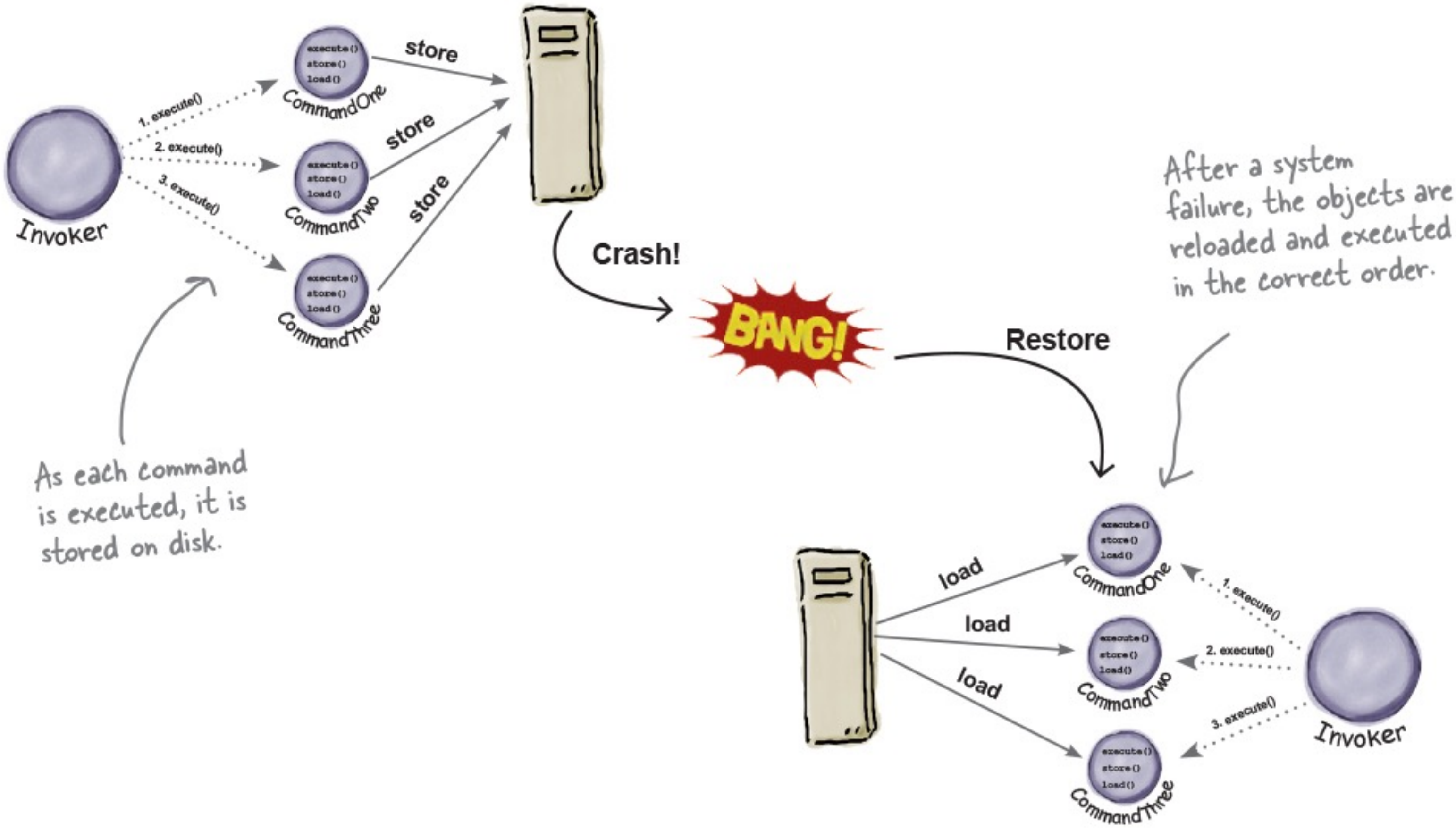
Logging Requests

By using logging, we can save all the operations since the last checkpoint, and if there is a system failure, apply those operations to our checkpoint. Take, for example, a spreadsheet application: 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.

How does it work?

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.

Recover from a crash



As each command is executed, it is stored on disk.

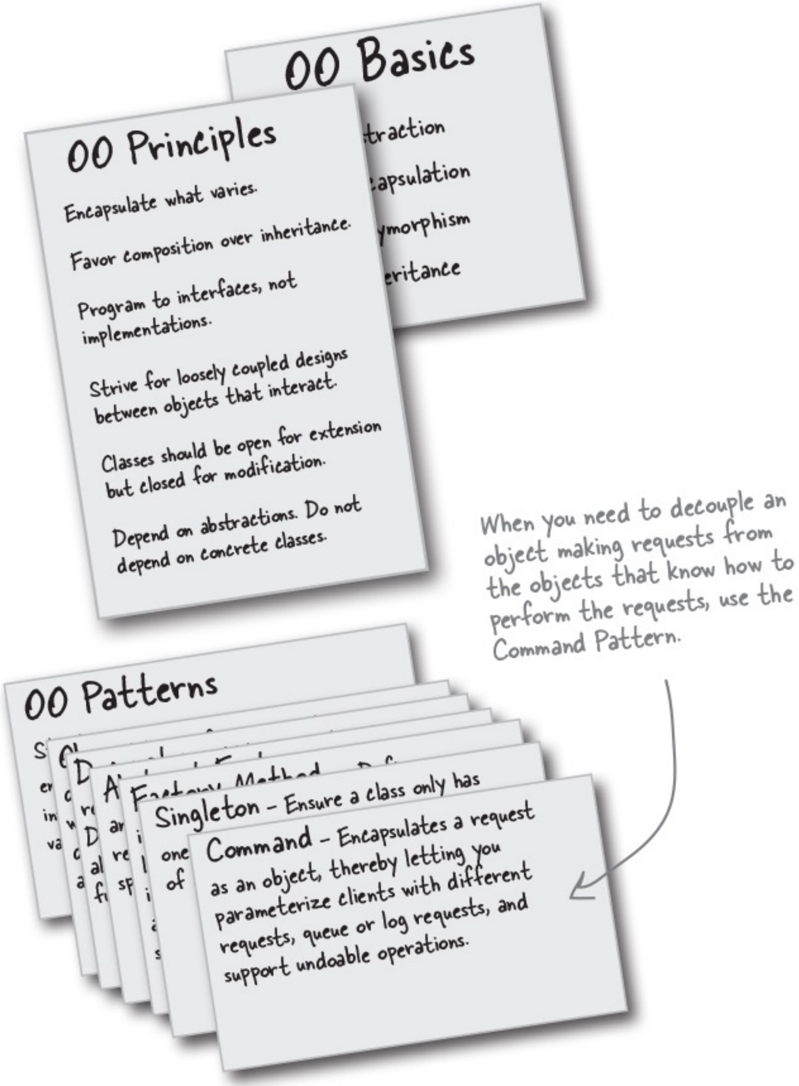
After a system failure, the objects are reloaded and executed in the correct order.

The Command Pattern

Encapsulates a request as an object, thereby letting you parametrize clients with different requests, queue or log requests, and support undoable operations.

Summary

2/27/22



OO Basics

traction

apsulation

ymorphism

eritance

OO Principles

Encapsulate what varies.

Favor composition over inheritance.

Program to interfaces, not implementations.

Strive for loosely coupled designs between objects that interact.

Classes should be open for extension but closed for modification.

Depend on abstractions. Do not depend on concrete classes.

OO Patterns

Singleton – Ensure a class only has one

Command – Encapsulates a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.

When you need to decouple an object making requests from the objects that know how to perform the requests, use the Command Pattern.