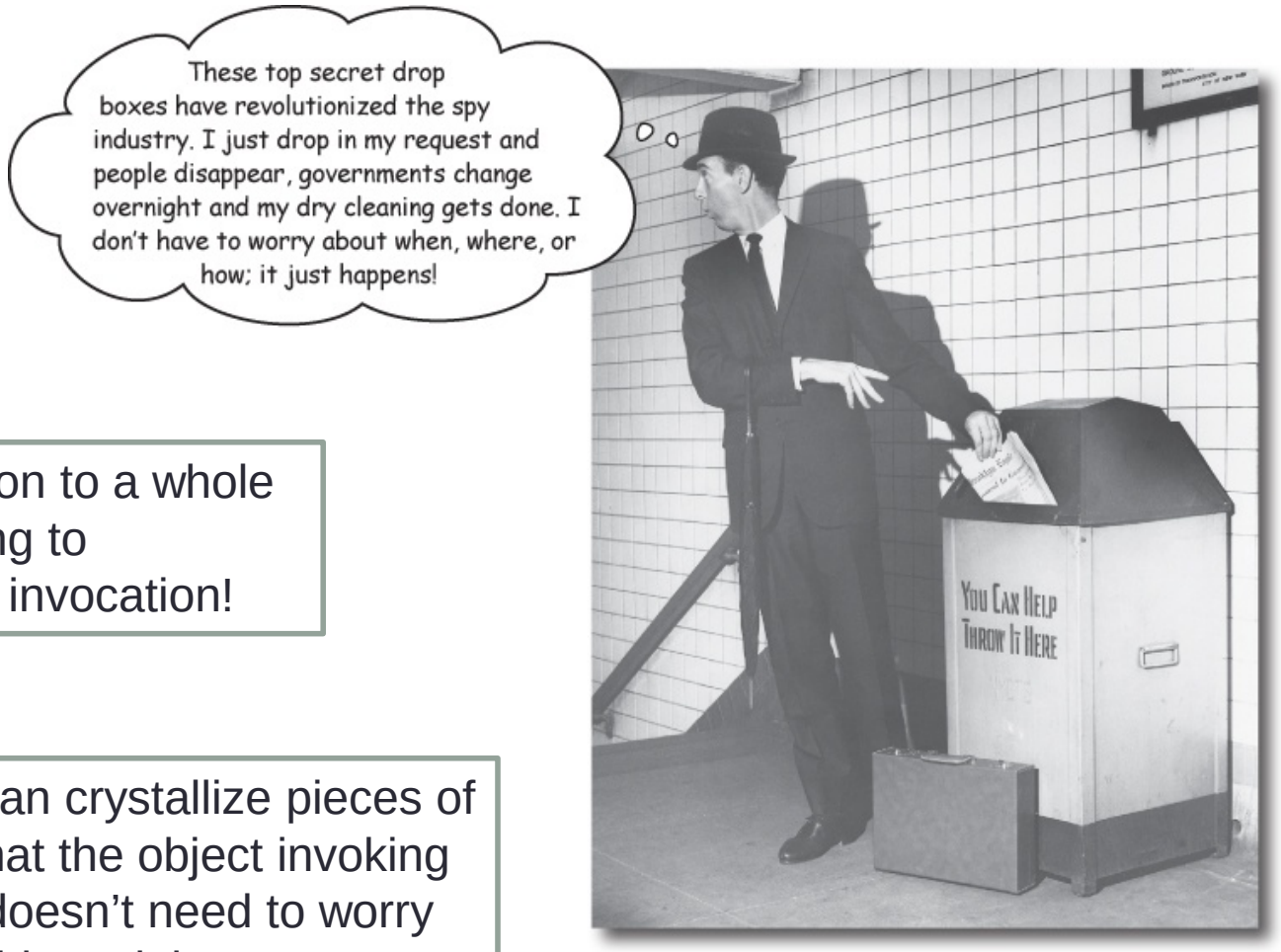
THE COMMAND PATTERN

Chandan R. Rupakheti

Week 4-2

Today



We take encapsulation to a whole new level: we're going to encapsulate method invocation!

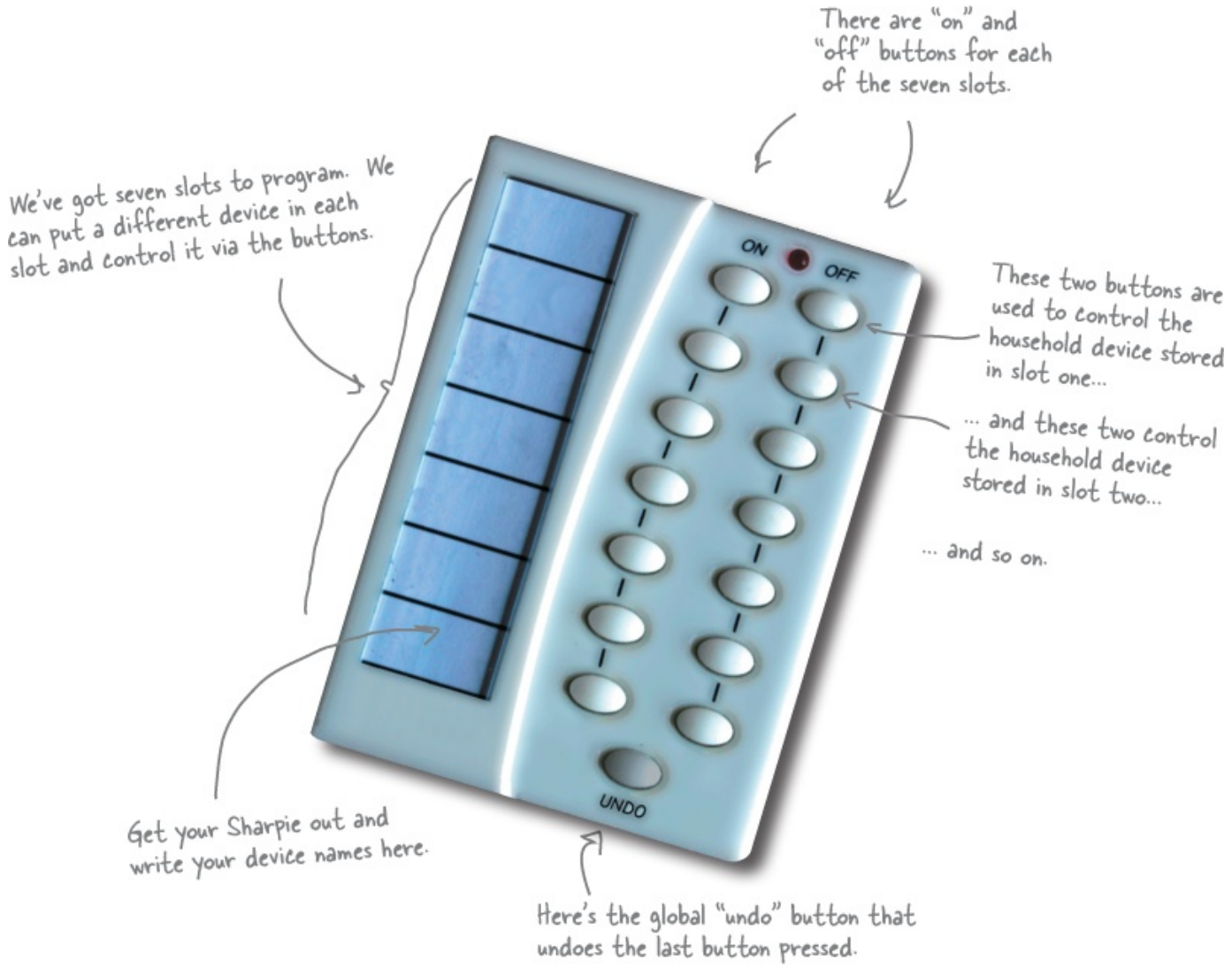
That's right! We can crystallize pieces of computation so that the object invoking the computation doesn't need to worry about how to do things, it just uses our crystallized method to get it done

But first, let's prepare for the client meeting …

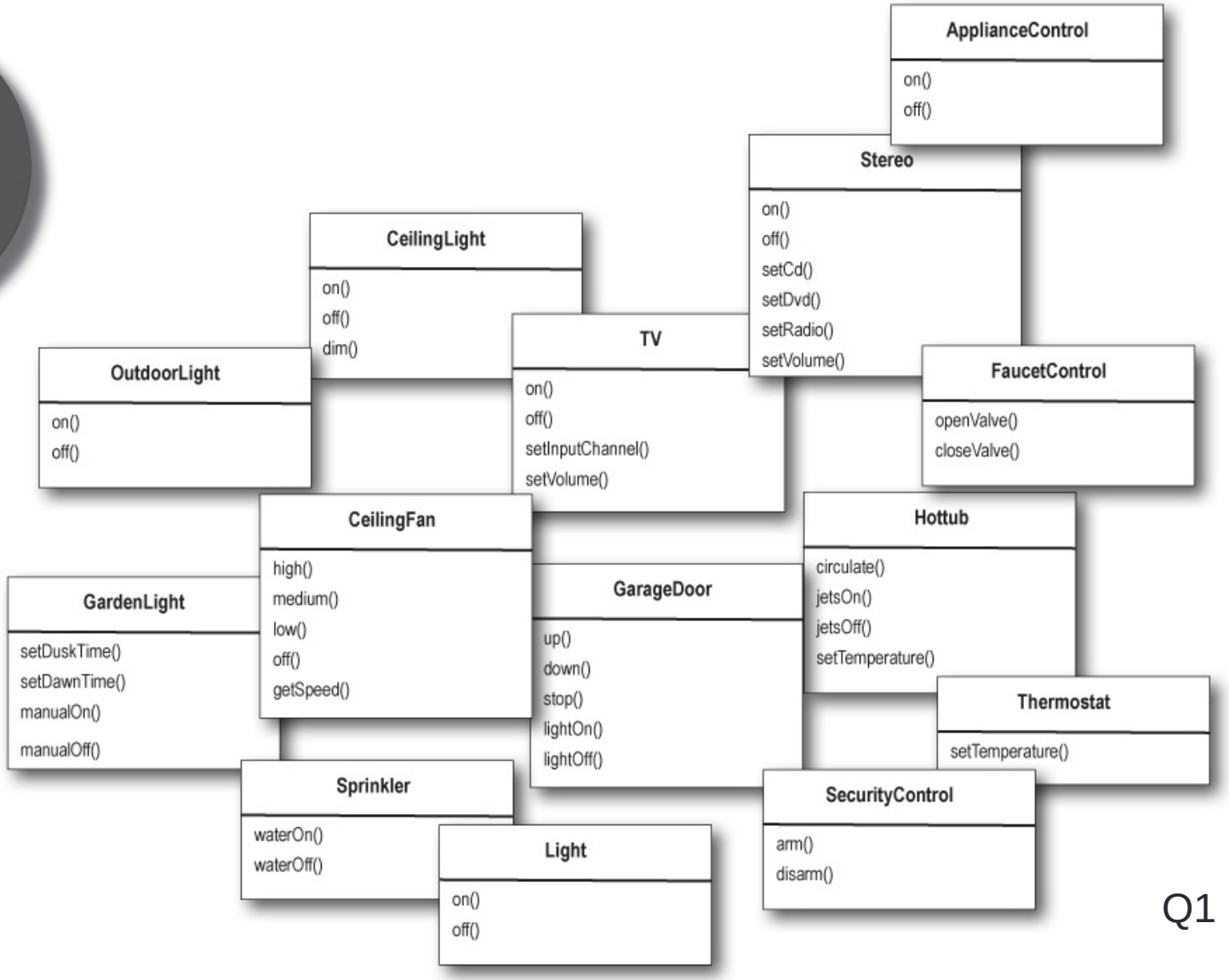
Home Automation or Bust, Inc.

Home Automation or Bust, Inc. **1221 Industrial Avenue, Suite 2000** **Future City, IL 62914**
Greetings!
I recently received a demo and briefing from Johnny Hurricane, CEO of Weather-O-Rama, on their new expandable weather station. I have to say, I was so impressed with the software architecture that **I'd like to ask you to design the API for our new Home Automation Remote Control**. In return for your services we'd be happy to handsomely reward you with stock options in Home Automation or Bust, Inc.
I'm enclosing a prototype of our ground-breaking remote control for your perusal. The **remote control features seven programmable slots** (each can be assigned to a different household device) along with **corresponding on/off buttons for each**. The remote also has a **global undo button**.
I'm also enclosing **a set of Java classes** on CD-R that were **created by various vendors to control** home automation **devices** such as lights, fans, hot tubs, audio equipment, and other similar controllable appliances.
We'd like you to create an API for programming the remote so that **each slot can be assigned to control a device or set of devices**. Note that it is important that we be able to **control the current devices on the disc, and also any future devices** that the vendors may supply.
Given the work you did on the Weather-O-Rama weather station, we know you'll do a great job on our remote control! We look forward to seeing your design.
Sincerely, Bill "X-10" Thompson, CEO

Free hardware! Let's check them out …



Taking a look at the vendor classes



Q1

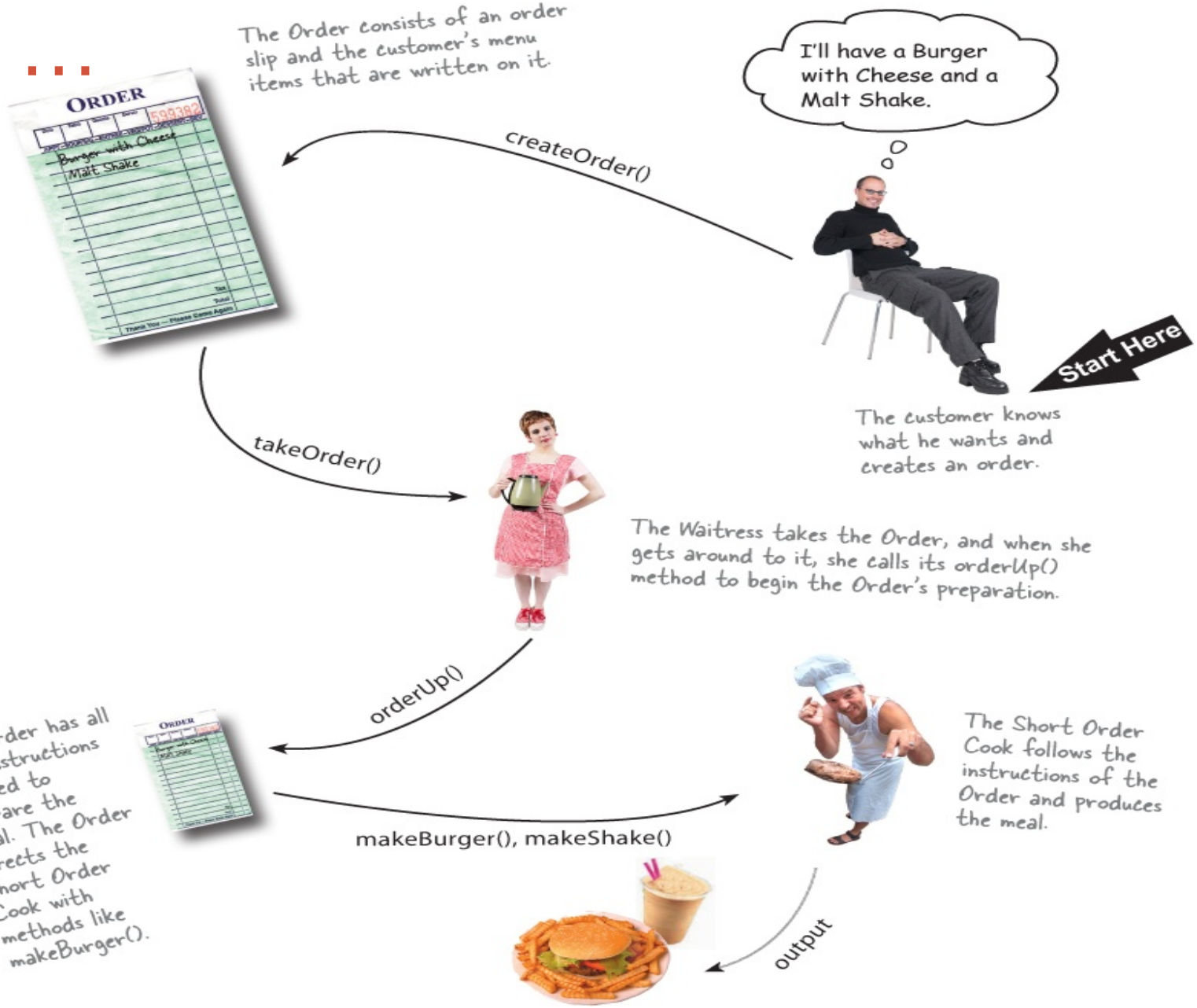
Lets go to a diner to find a solution …

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

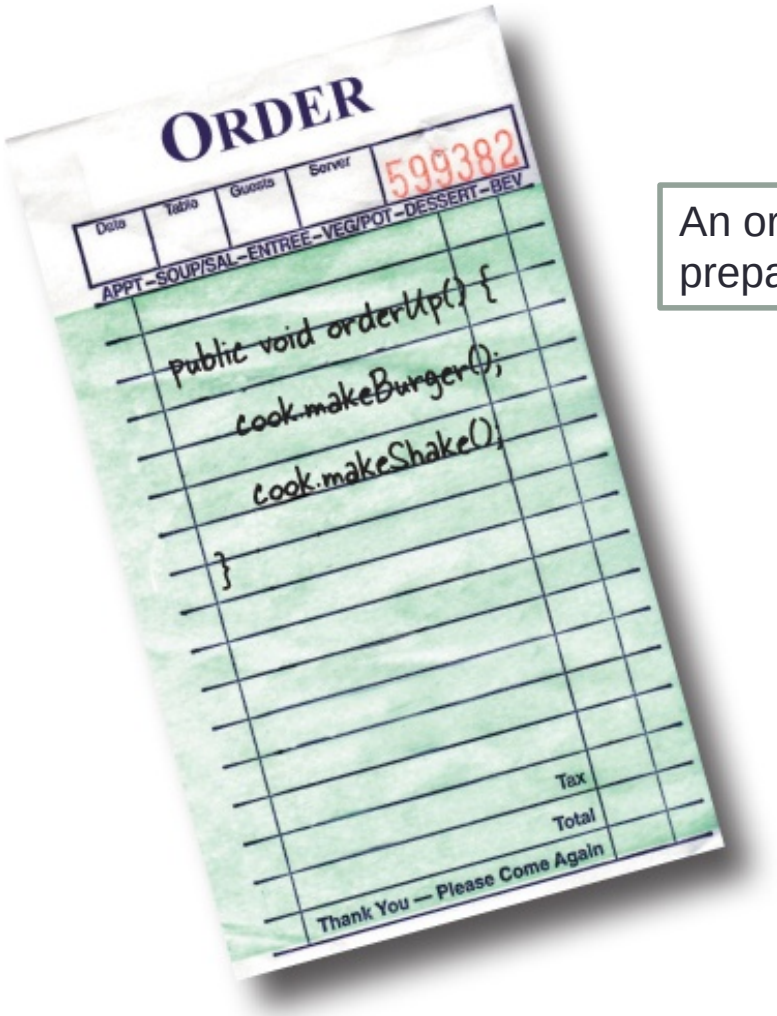
❷ The Waitress takes the Order, places it on the order counter, and says "Order up!"

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

Details …


The Order consists of an order slip and the customer's menu items that are written on it.
ORDER
Burger with Cheese
Malt Shake
I'll have a Burger with Cheese and a Malt Shake.
createOrder()
Start Here
The customer knows what he wants and creates an order.
takeOrder()
The Waitress takes the Order, and when she gets around to it, she calls its orderUp() method to begin the Order's preparation.
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

Role of an Order Slip



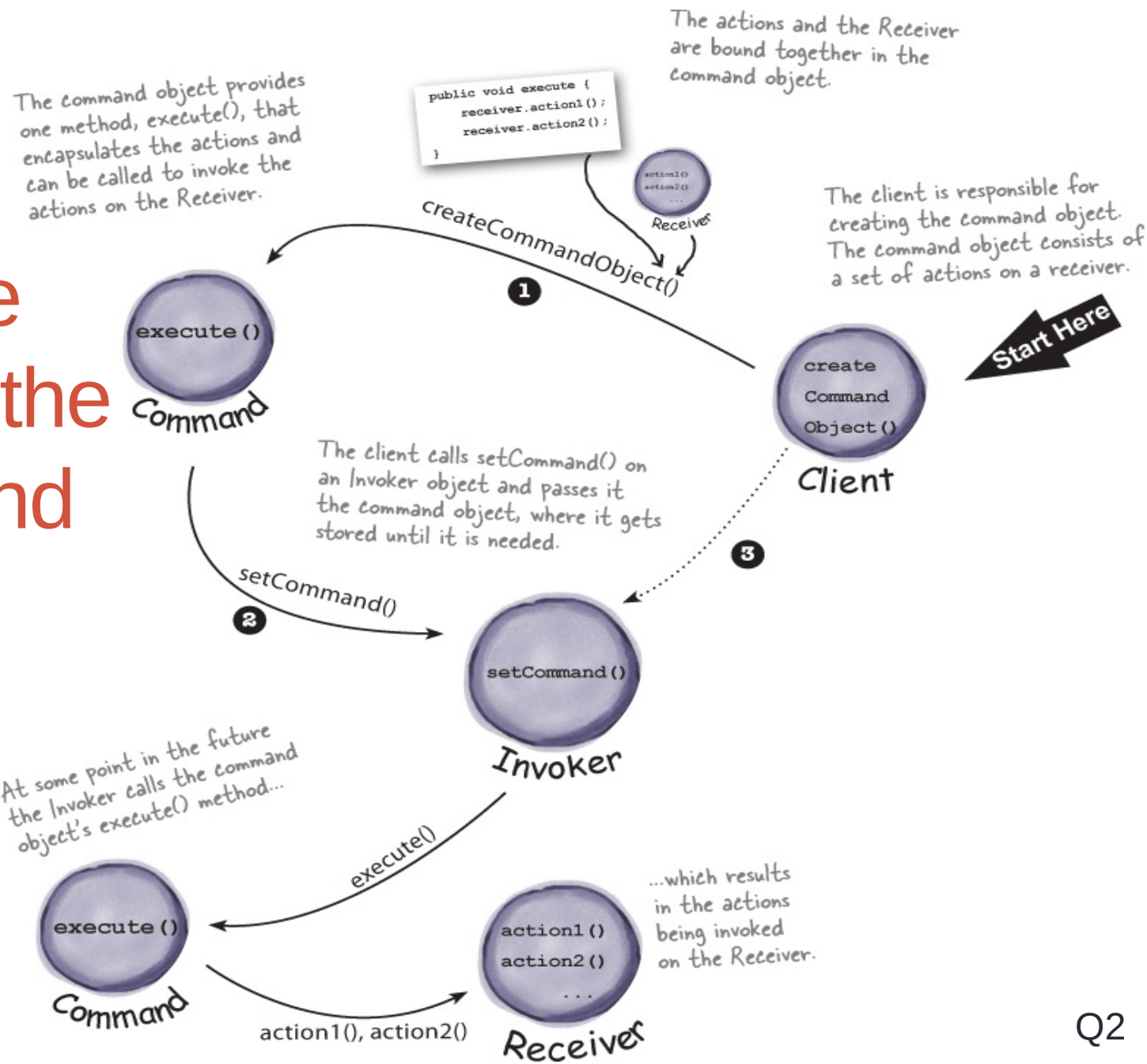
An order slip encapsulates a request to prepare a meal

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!"

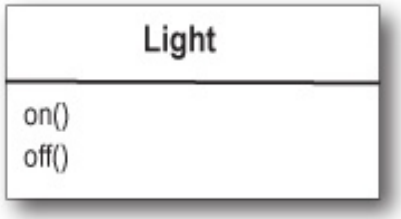
The Cook object knows how to cook

From the Diner to the Command Pattern



Q2

Our first command object




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

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

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

```
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

```
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.

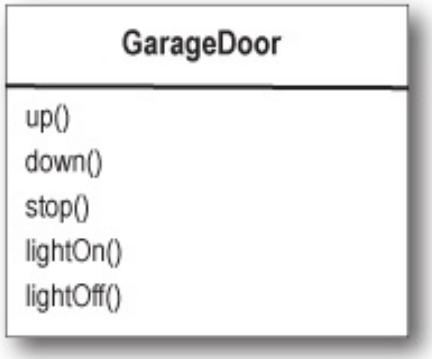
Here, pass the command to the Invoker.

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

%
```

Implementing more commands




```
public class GarageDoorOpenCommand
        implements Command {

}
```

Your code here

```
public class RemoteControlTest {
    public static void main(String[] args) {
        SimpleRemoteControl remote = new SimpleRemoteControl();
        Light light = new Light();
        GarageDoor garageDoor = new GarageDoor();
        LightOnCommand lightOn = new LightOnCommand(light);
        GarageDoorOpenCommand garageOpen =
            new GarageDoorOpenCommand(garageDoor);


        remote.setCommand(lightOn);
        remote.buttonWasPressed();
        remote.setCommand(garageOpen);
        remote.buttonWasPressed();
    }
}
```

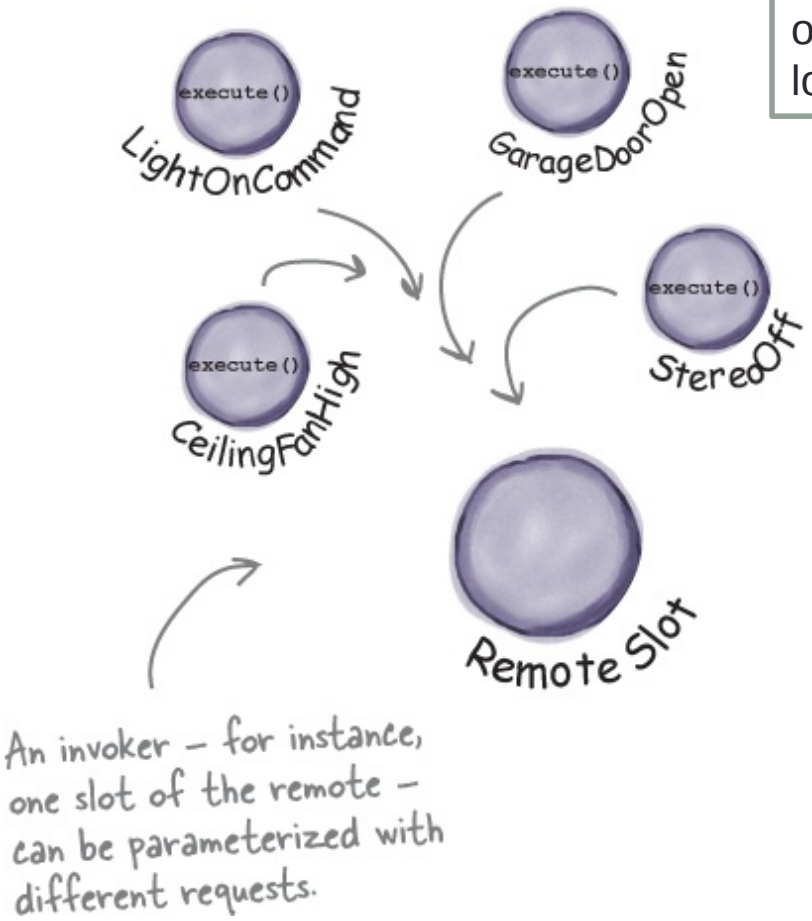
Your output here.

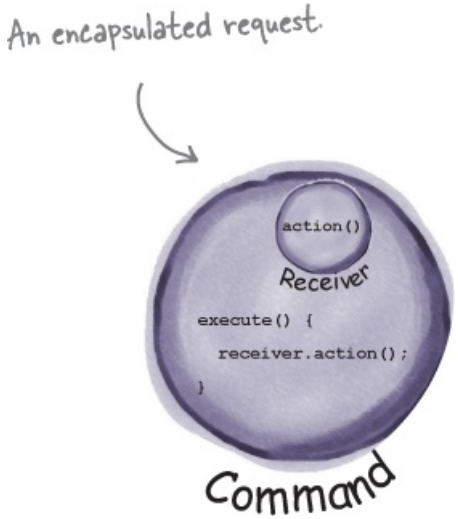
File Edit Window Help GreenEggs&Ham

```
%java RemoteControlTest
```

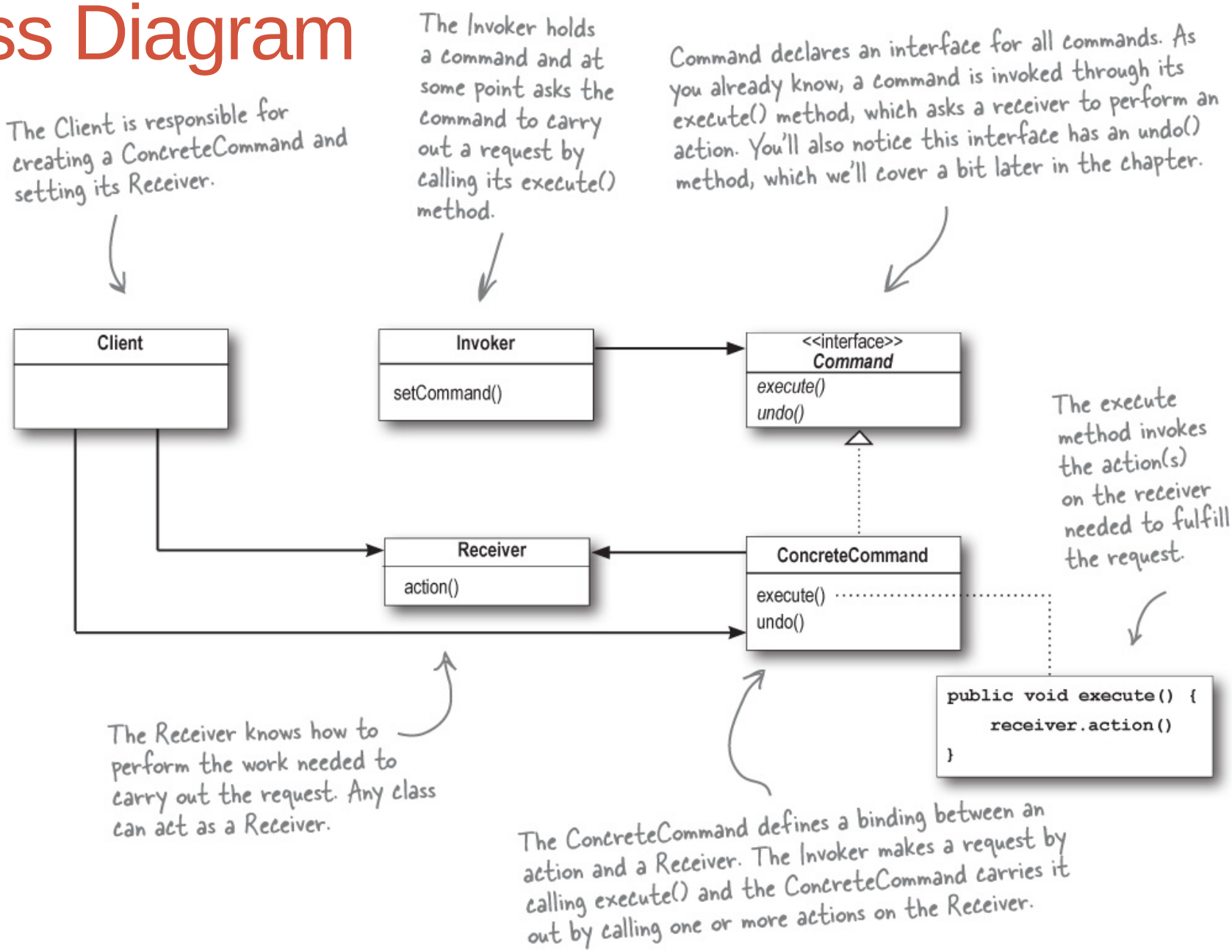
The Command Pattern defined

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.

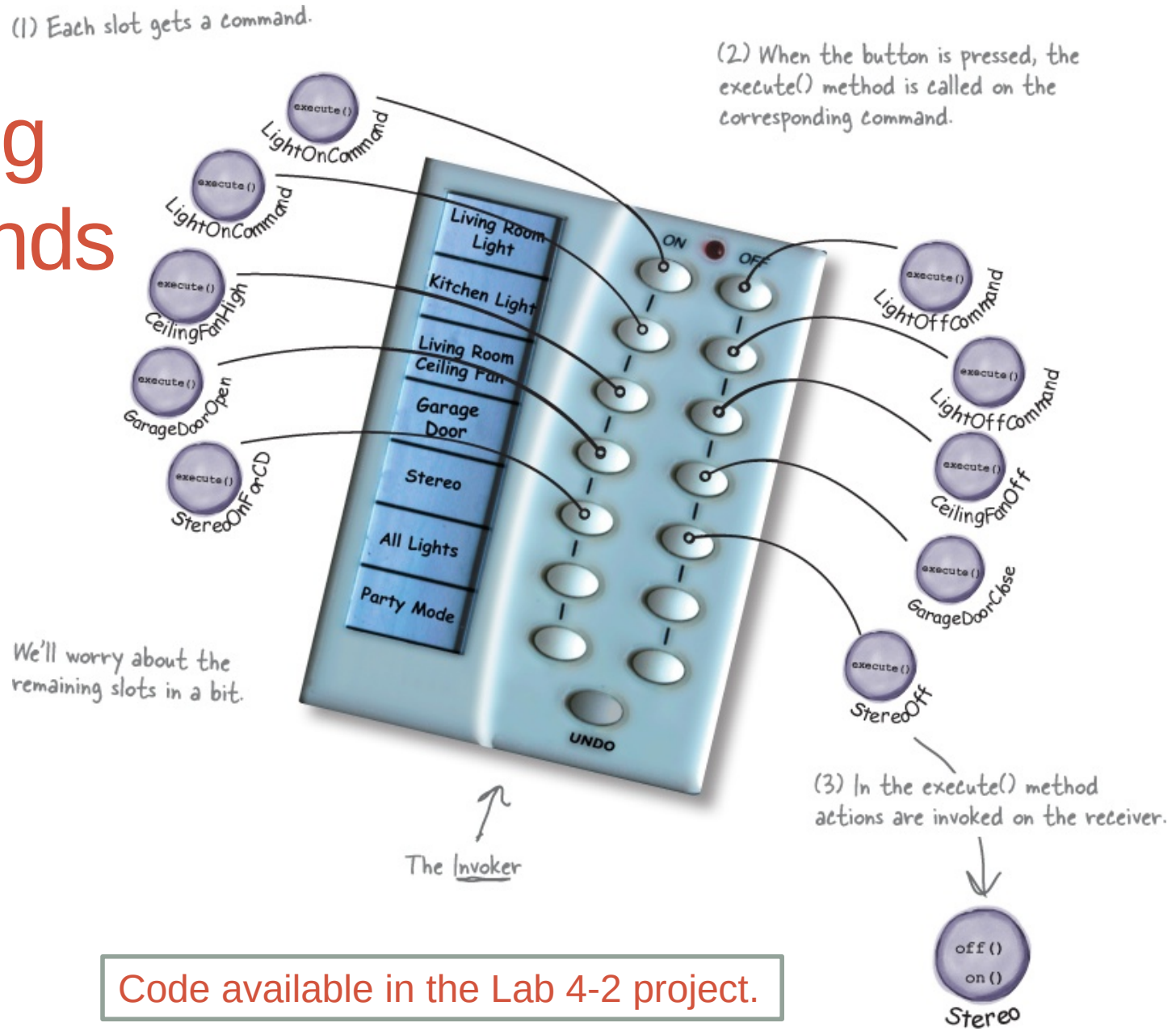




Class Diagram



Assigning Commands to slots



Code available in the Lab 4-2 project.

Coding 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;
        }
    }

    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();
    }
```

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

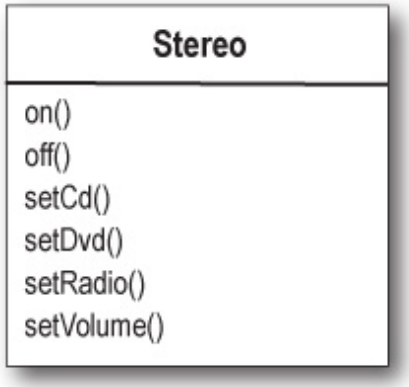
In the constructor all we need to do is instantiate and initialize the on and off arrays.

The 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().

A little more challenging command

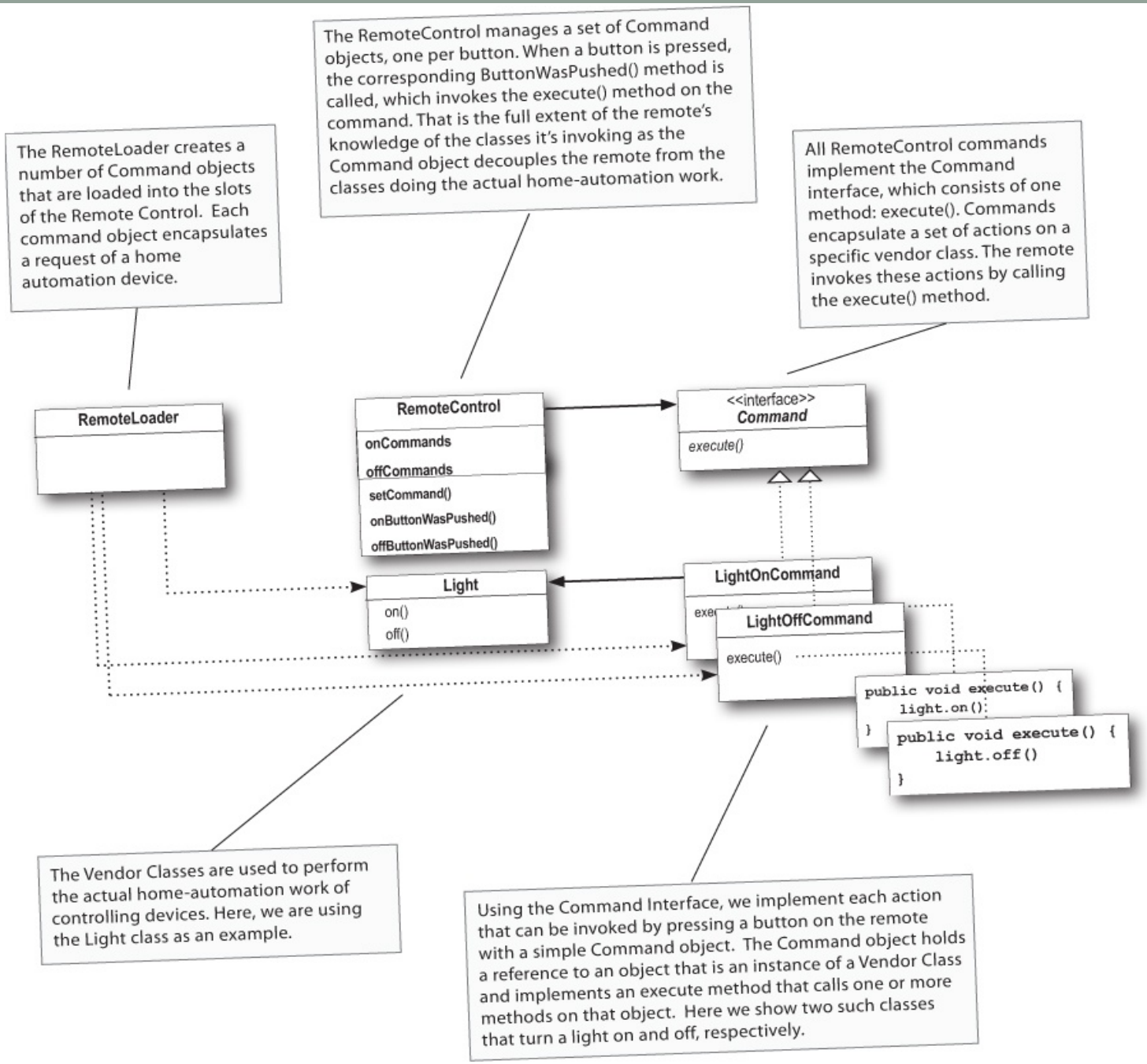



```
public class StereoOnWithCDCommand implements Command {
    Stereo stereo;

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

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

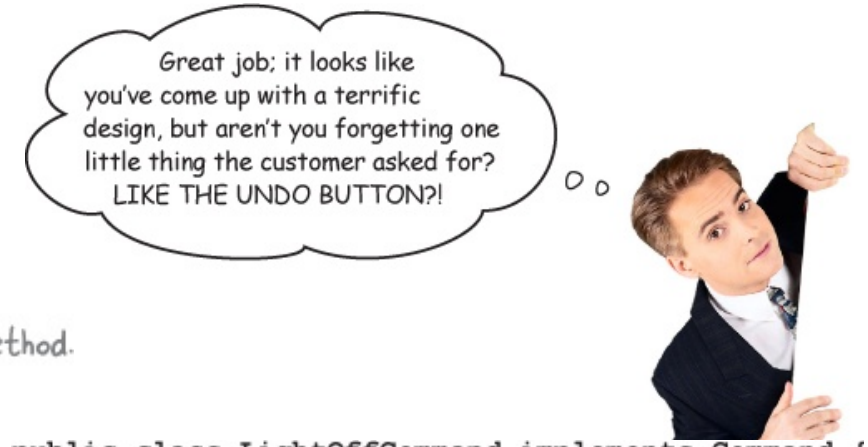
Just like the LightOnCommand, we get passed the instance of the stereo we are going to be controlling and we store it in a local instance variable.

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?

Final Doc for the Client



What about the undo feature?




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

Here's the new undo() method.

```
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.

```
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;
    }

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

    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...
    }
}
```

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.

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.

Recap

The Command Pattern decouples an object making a request from the one that knows how to perform it

Invokers can be parameterized with Commands, even dynamically at runtime.

Commands may support undo by implementing an undo method that restores the object to its previous state before the execute() method was last called.