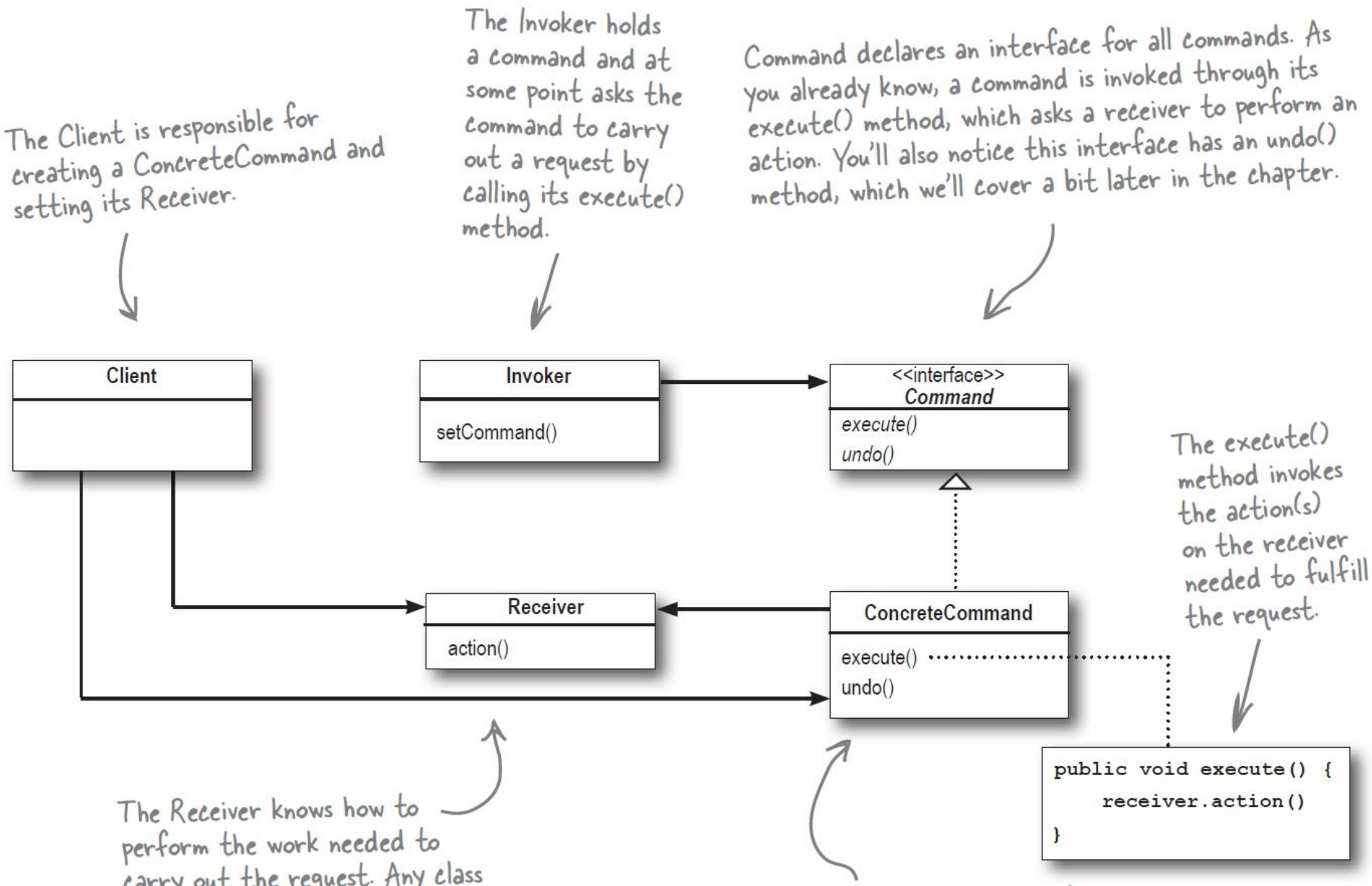
Command Pattern

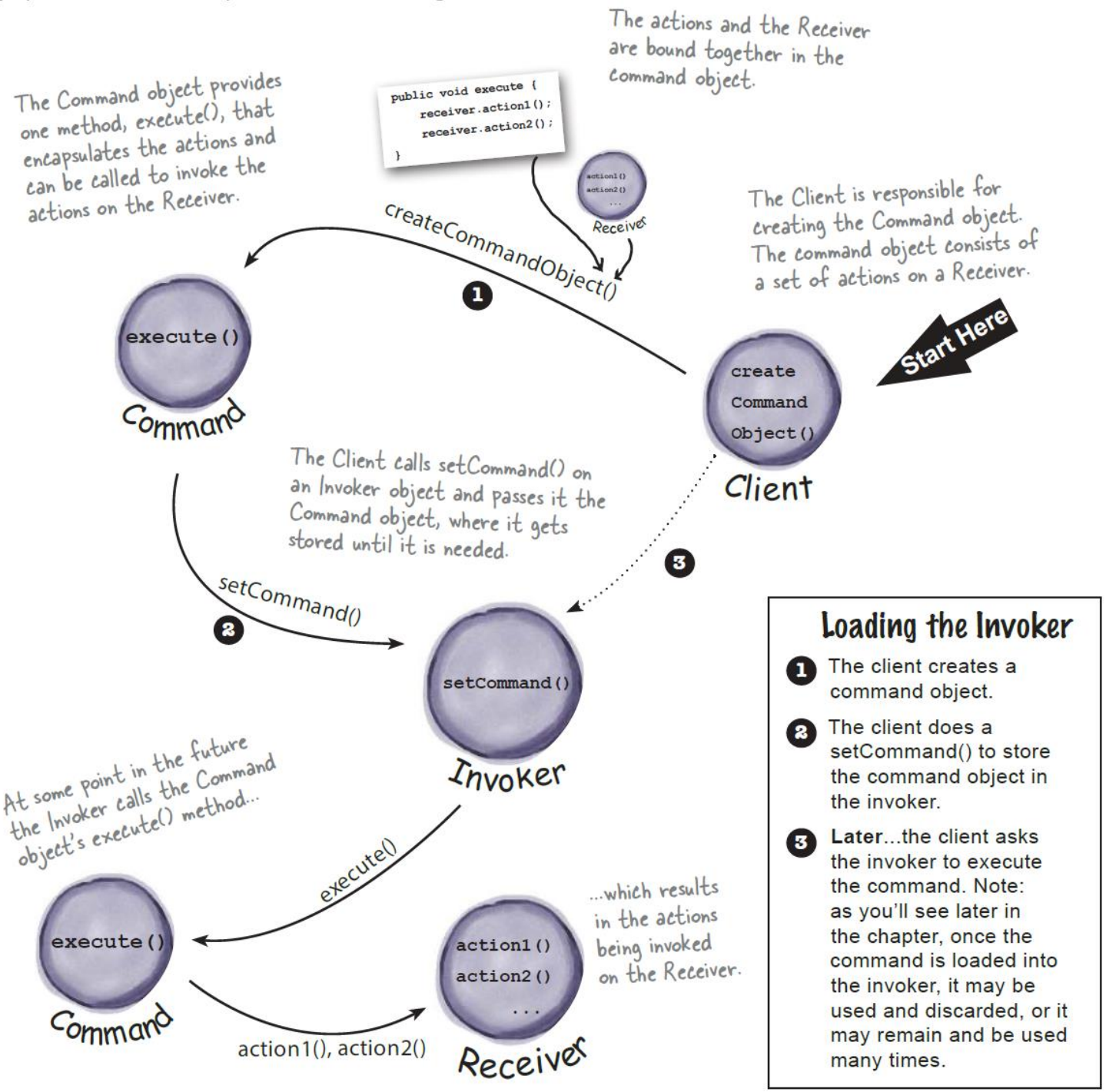
For the Complete Code, See the "Official" Head-First Design Patterns GitHub Repo:

https://github.com/bethrobson/Head-First-Design-Patterns/tree/master/src/headfirst/designpatterns/

And the course SVN repo:

svn://cosc436.net:65436/Examples/trunk


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



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.

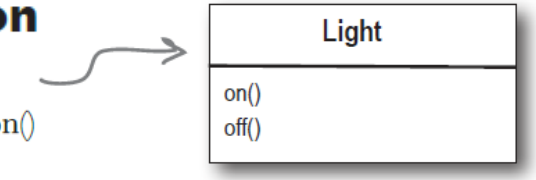
Implementing the Command interface

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

Simple. All we need is one method called 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(). Here's how you can implement this as a command:



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.

Creating a 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.

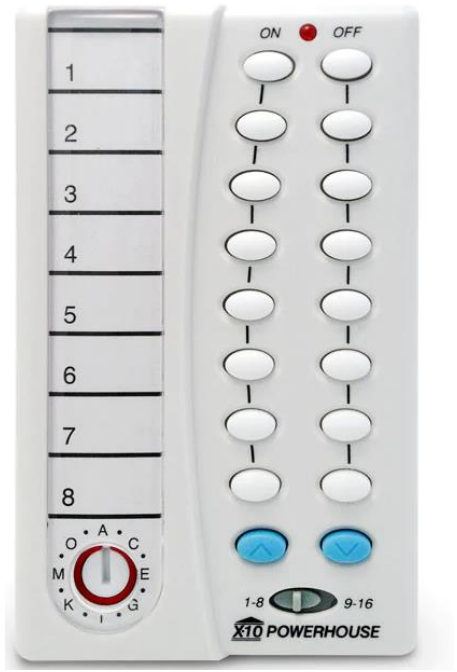
And then we simulate the button being pressed.

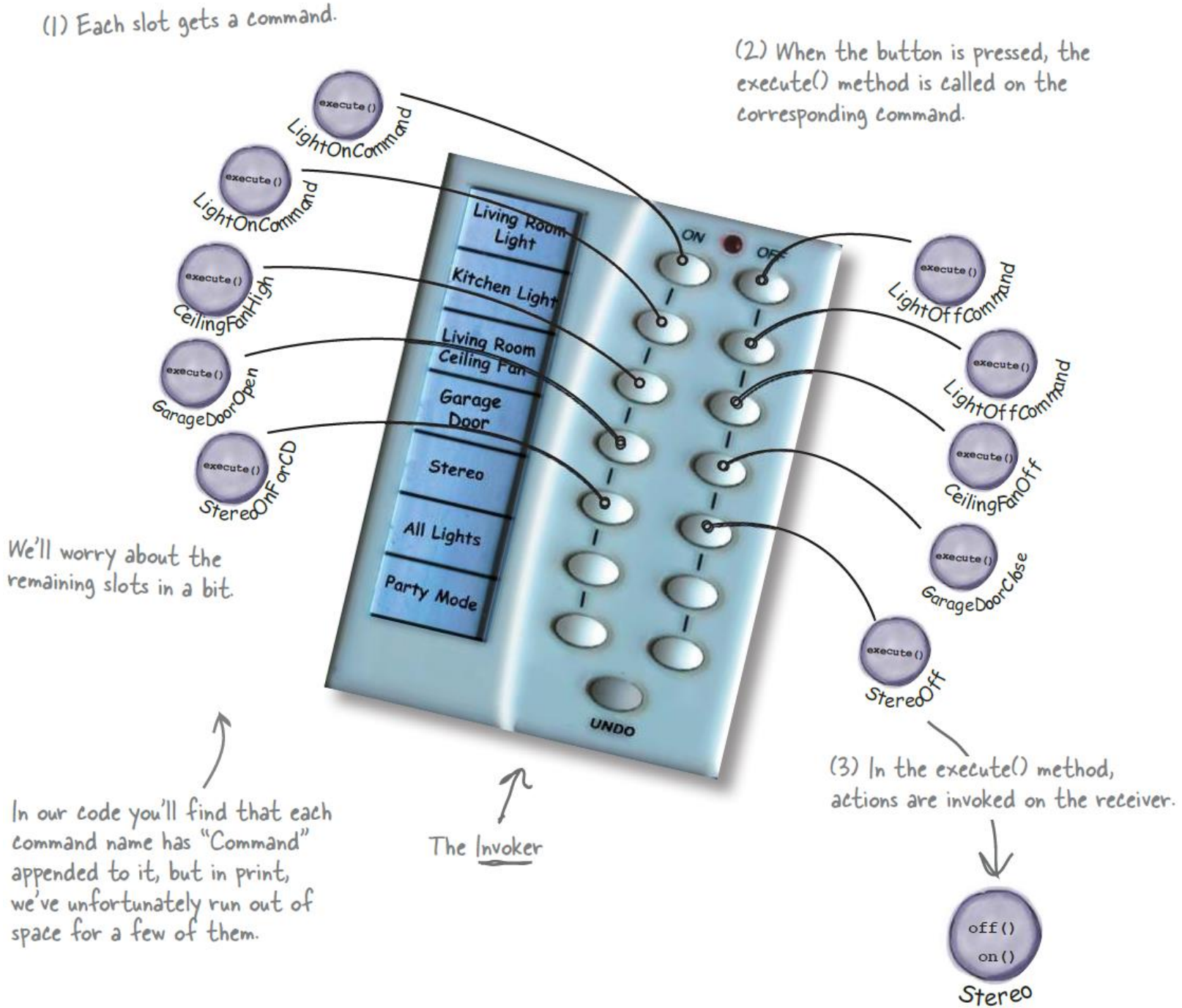
Here's the output of running this test code.

```
File Edit  Window  Help  DinerFoodYum
%java RemoteControlTest

Light is On

%
```

Yes, this is [1970's technology](#) that still exists today!

https://www.x10.com/





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;
        }
    }

    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.

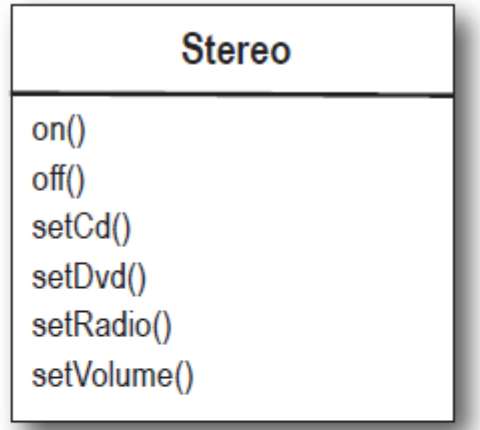
The setCommand() method takes a slot position and an On and Off command to be stored in that slot.

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().

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'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?

```
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("Garage");
        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);

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

Create all the devices in their proper locations.

Create all the Light Command objects.

Create the On and Off for the ceiling fan.

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

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

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 assigned to it. (Note that toString() gets called automatically here, so we don't have to call toString() explicitly.)

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

See the code samples for expanding this idea to:

- Accommodate the “empty” slots (hint, a “`EmptyCommand`” class).
- Creating an “Undo” Command. (Via very abbreviated “Memento Pattern”)
- Use Lambda functions to greatly reduce the “class clutter”.

What are the most obvious patterns that could be combined with the Command Pattern to make for a more useful, or more flexible remote-control implementation?

Builder? State? Iterator? Observer?