

Encapsulating invocation

ELSYS 2014/2015

Vasil Kostov

Georgi Yosifov

In this lecture

- Encapsulating method invocation
- Implementing Undo button

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.

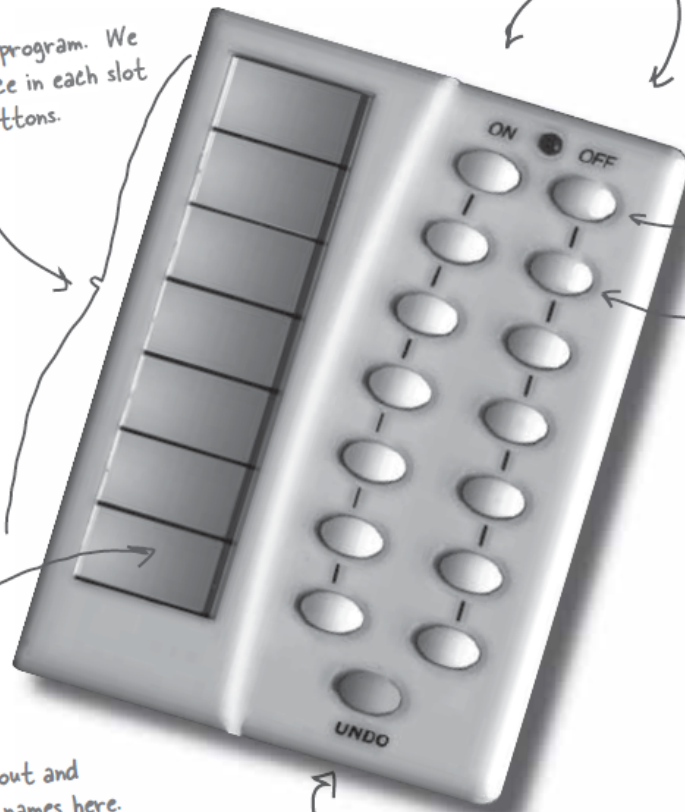
These two buttons are used to control the household device stored in slot one...

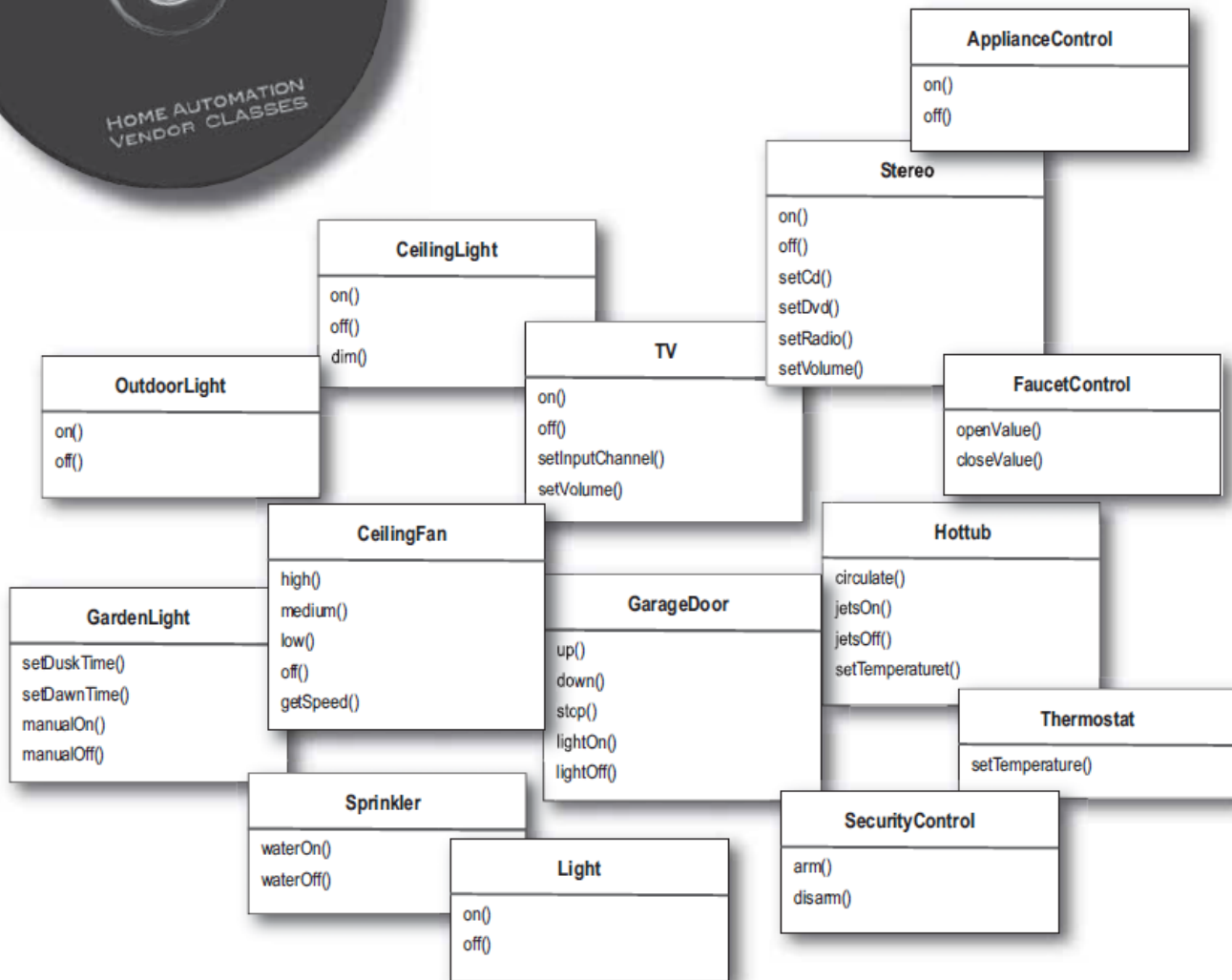
... and these two control the household device stored in slot two...

... and so on.

Get your Sharpie out and write your device names here.

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





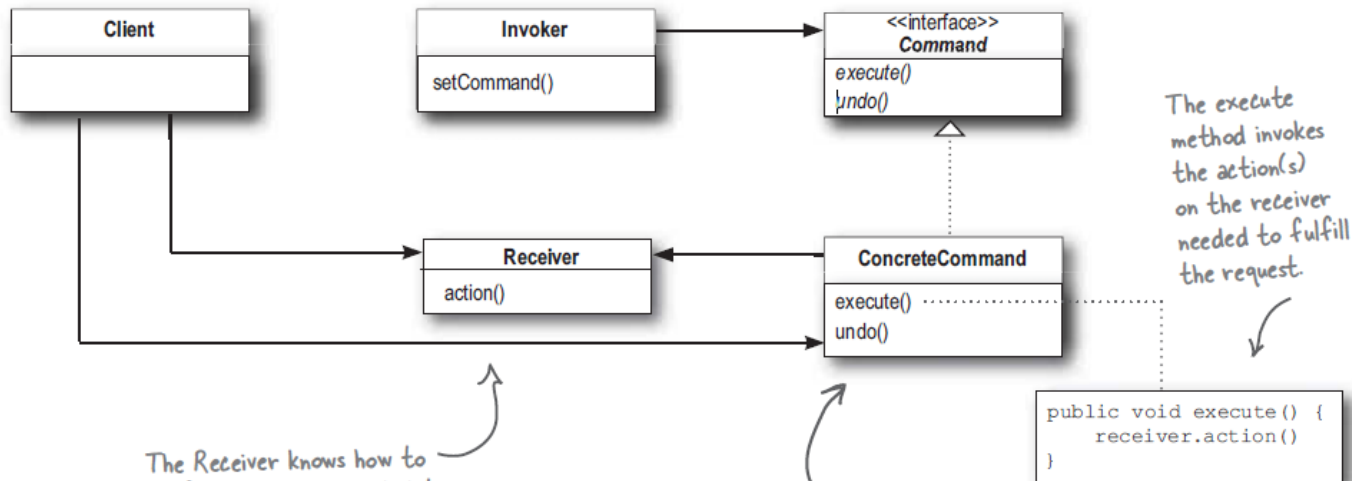
- Not only we have “On()” and “Off()” methods, but also “SetTemperature()” and “setVolume()”
- More vendors in the future
- The remote shouldn't know about the vendor specific

Command Pattern?

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 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.

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

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.


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

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

```
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.

```
public class SimpleRemoteControl {  
    Command slot;
```

← We have one slot to hold our command,
which will control one device.

```
    public SimpleRemoteControl() {}
```

```
    public void setCommand(Command command) {  
        slot = command;  
    }
```

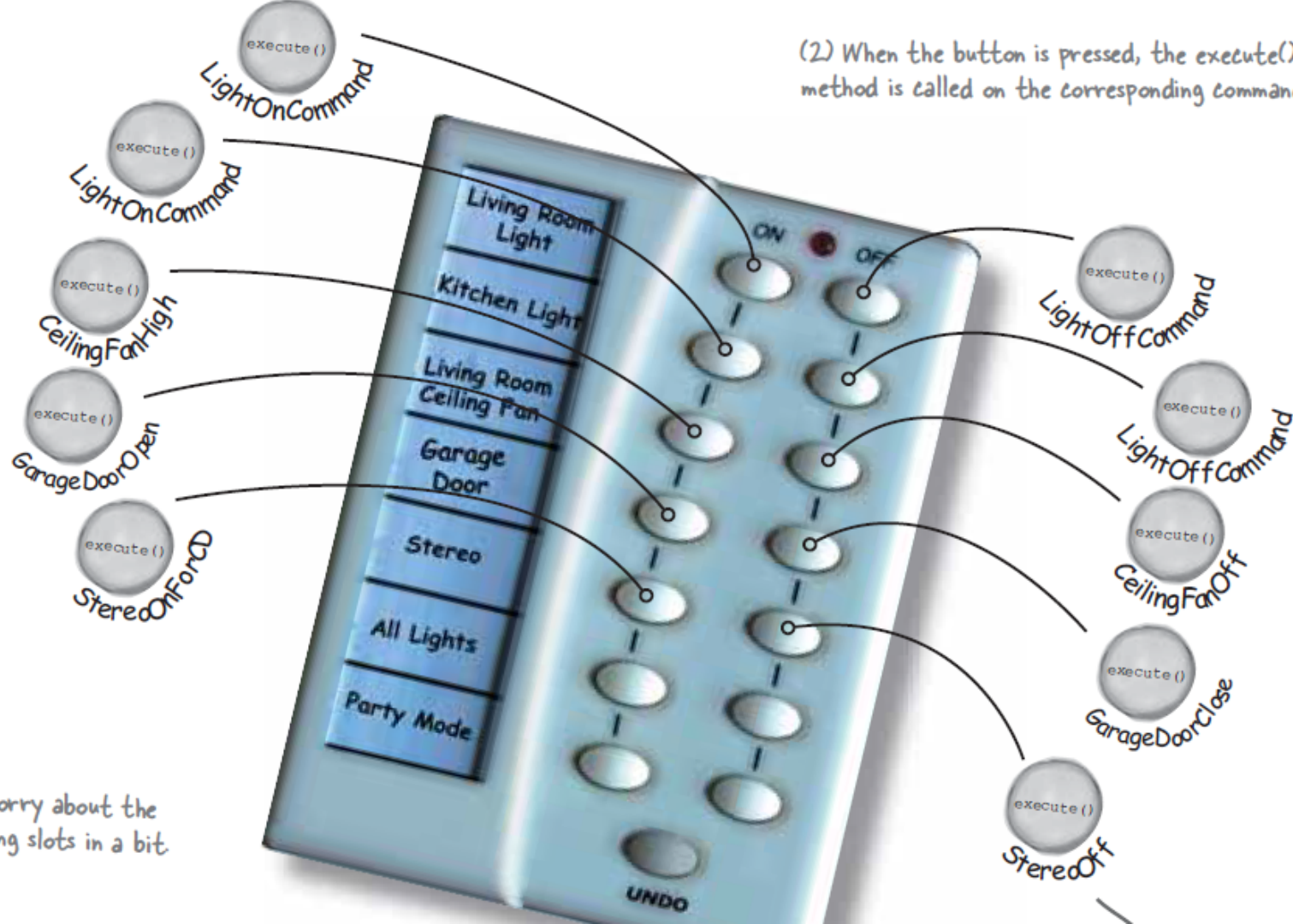
← We have a method for setting
the command the slot is going
to control. This could be called
multiple times if the client of
this code wanted to change the
behavior of the remote button.

```
    public void buttonWasPressed() {  
        slot.execute();  
    }
```

← 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.

```
}
```

(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.

```
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;  
    }  
}
```



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

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

```
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?

NoCommand? (Null Object)

In object-oriented computer programming, a Null Object is an **object with defined neutral ("null") behavior**. The Null Object design pattern describes the uses of such objects and their behavior (or **lack** thereof). It was first published in the Pattern Languages of Program Design book series.


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

NoCommand class

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

Undo button

```
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.

Party Mode

```
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.