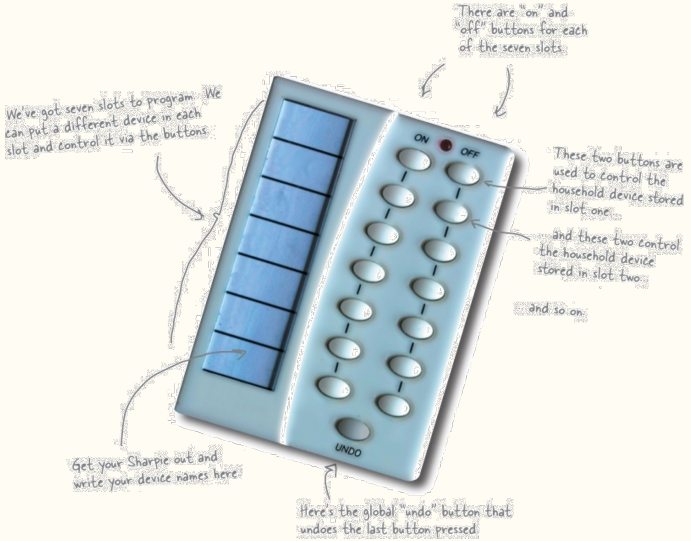
Command Pattern

Encapsulation of method invocation

Case: Home Automation Remote Control

- Universal remote control
- Seven Slots to program
  → different device control via the the buttons
- On and Off buttons for each of the seven slots
- The global undo button

- Utilization of encapsulation

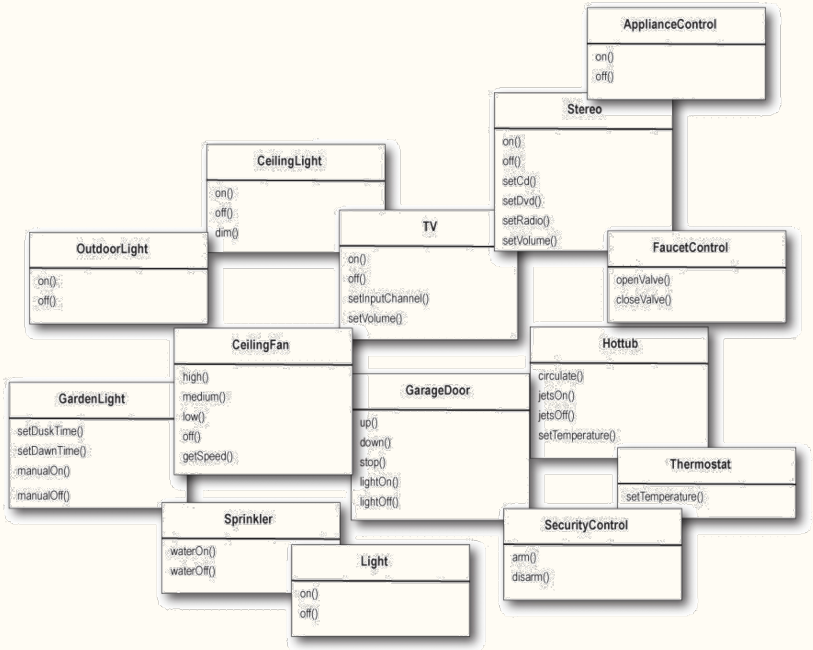


Proposed Classes

Independent classes for all possible devices!!!

Looking carefully there is multiple operation besides on and off.

What is the disadvantage of this approach???



Design Issue

- Decoupling of the requester of an action from the object performing the action
  - the requester -> remote control
  - The object -> an instance of one of your vendor classes
- By using “command objects” into design
  - Encapsulate a request to do something

in the diner, ordering process

Scenario

Customer

CreateOrder()

Order

takeOrder()

Waitress

The order = menu items

orderUP()

Decoupling of Waitress and Cook

Order

The order = menu items + instructions to cook

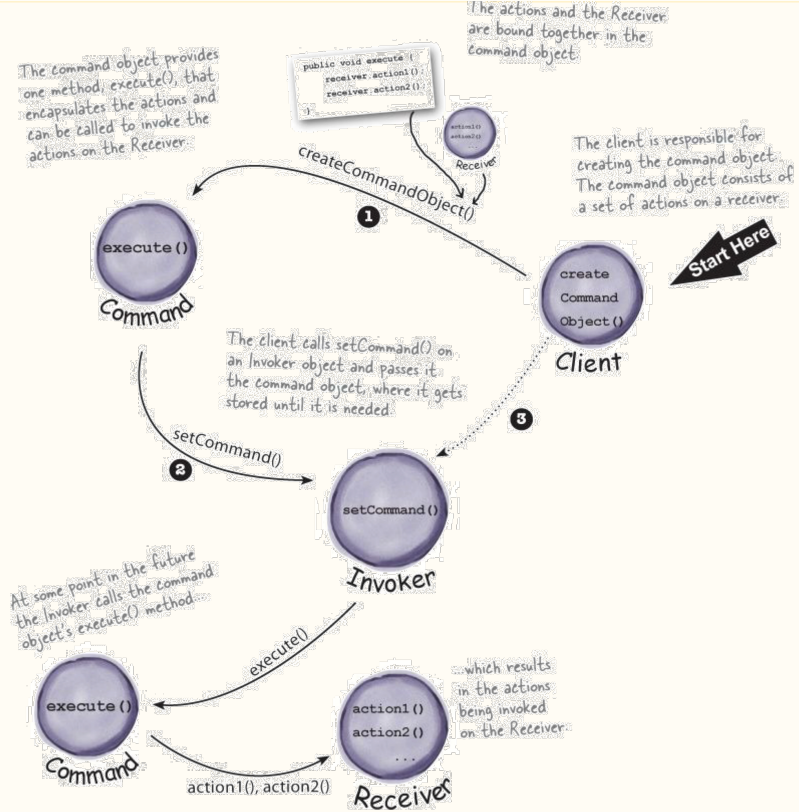
makeBurger()
makeShake()

output

ShortOrderCook

This ShortOrderCook follows the given instruction

1. client creates a command object
    1. Receiver + actions => in command obj.
2. Client does a setCommand() to store the command object
3. Client asks the invoker to execute the command

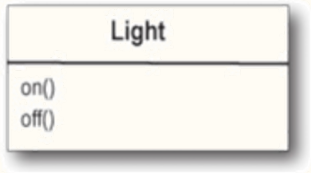


Command Interface

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

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

Light object is the => the receiver of the request



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.

Simplified Example: One button control

```
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.

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.

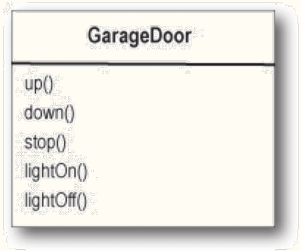
Here, pass the command to the Invoker.

And then we simulate the button being pressed.

Here's the output of running this test code.

```
File  Edit  Window  Help  DinerFoodYum
%java RemoteControlTest

Light is On

%
```

Practice: GarageDoor Control




```
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();
    }
}
```

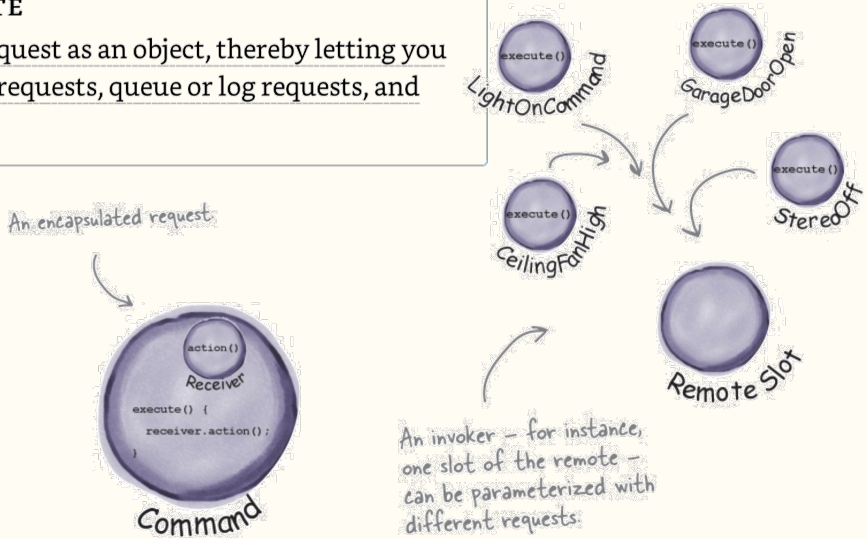
Try this code! What is the result?

Definition of Command Pattern

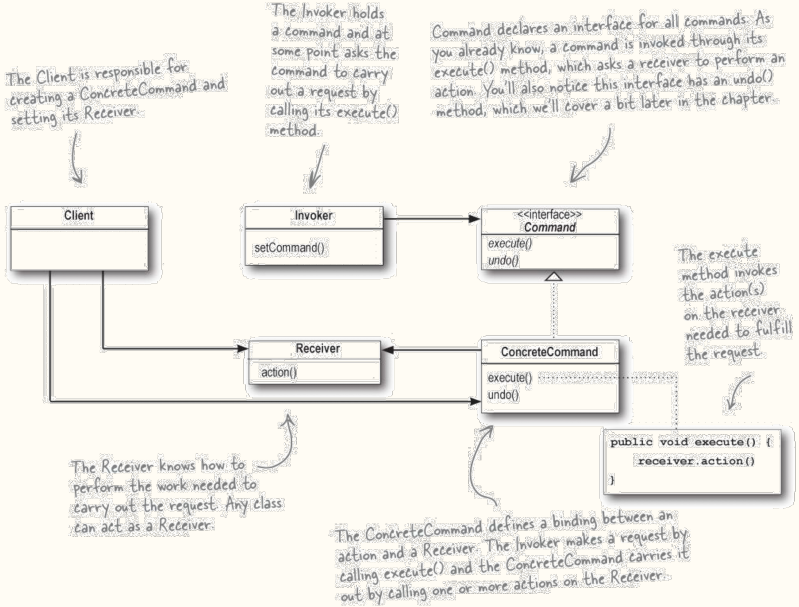
NOTE

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.

120 highlighters

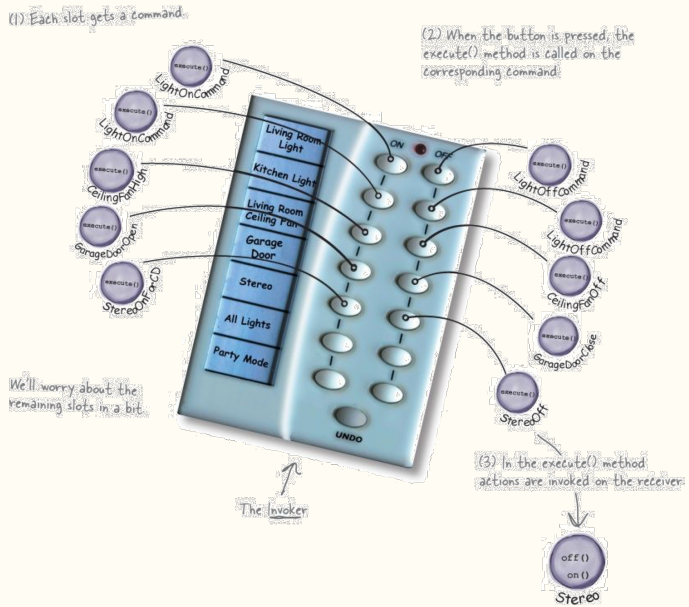


Command Pattern class diagram



Assigning Commands to Slots

- Assign each slot to a command in the remote control
  - Remote control => invoker
  - When button is pressed -> execute() on the corresponding command



```
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();
    }

    public String toString() {
        StringBuffer stringBuff = new StringBuffer();
        stringBuff.append("\n------ Remote Control -------\n");
        for (int i = 0; i < onCommands.length; i++) {
            stringBuff.append("[slot " + i + "] " + onCommands[i].getClass().getName()
                + "    " + offCommands[i].getClass().getName() + "\n");
        }
        return stringBuff.toString();
    }
}
```

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.

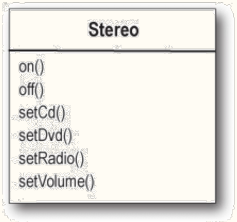
The setCommand() method takes a slot position and an On and Off command to be stored in that slot.

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().

We've overwridden toString() to print out each slot and its corresponding command. You'll see us use this when we test the remote control.

Implementation of Commands




```
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?

```
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);

        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);

        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);
    }
}
```

Create the Up and Down commands for the Garage.

Create the stereo On and Off commands.

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.

File Edit Window Help CommandsGetThingsDone

```
% java RemoteLoader
------ Remote Control -------
[slot 0] LightOnCommand             LightOffCommand
[slot 1] LightOnCommand             LightOffCommand
[slot 2] CeilingFanOnCommand        CeilingFanOffCommand
[slot 3] StereoOnWithCDCommand      StereoOffCommand
[slot 4] NoCommand                  NoCommand
[slot 5] NoCommand                  NoCommand
[slot 6] NoCommand                  NoCommand
```


```
Living Room light is on
Living Room light is off
Kitchen light is on
Kitchen light is off
Living Room ceiling fan is on high
Living Room ceiling fan is off
Living Room stereo is on
Living Room stereo is set for CD input
Living Room Stereo volume set to 11
Living Room stereo is off
%
```

Our commands in action! Remember, the output from each device comes from the vendor classes. For instance, when a light object is turned on it prints "Living Room light is on."

Handling Not allocated Buttons

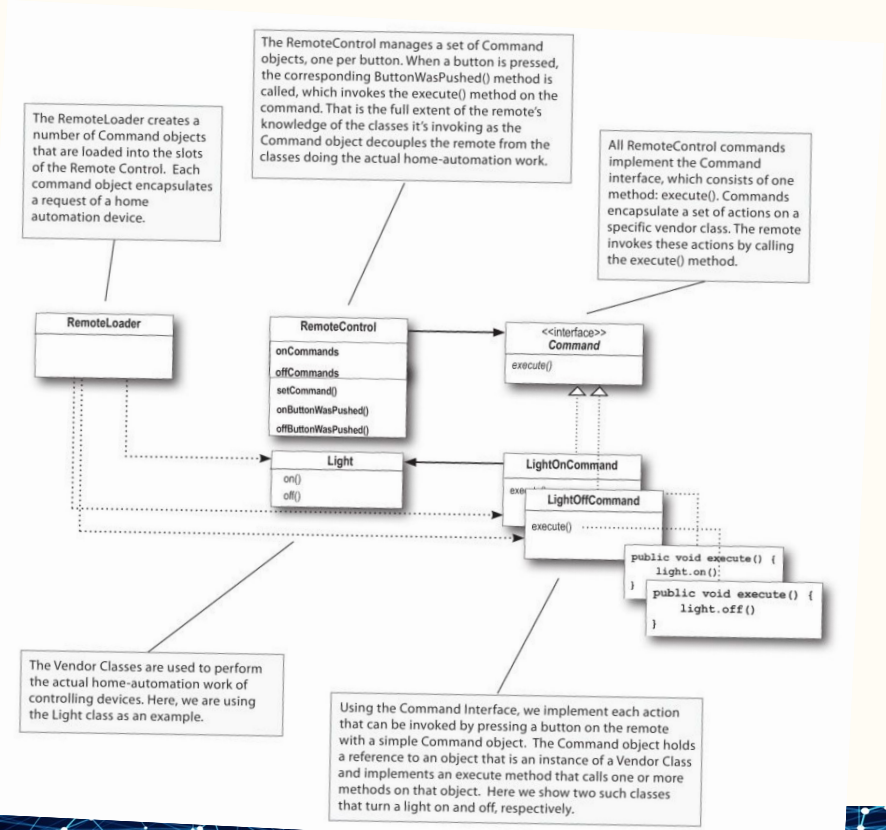
```
public void onButtonWasPushed(int slot) {
    if (onCommands[slot] != null) {
        onCommands[slot].execute();
    }
}
```

So, how do we get around that? Implement a command that does nothing!

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

Then, in our RemoteControl constructor, we assign every slot a NoCommand object by default and we know we'll always have some command to call in each slot.

```
Command noCommand = new NoCommand();
for (int i = 0; i < 7; i++) {
    onCommands[i] = noCommand;
    offCommands[i] = noCommand;
}
```


The RemoteControl manages a set of Command objects, one per button. When a button is pressed, the corresponding ButtonWasPushed() method is called, which invokes the execute() method on the command. That is the full extent of the remote's knowledge of the classes it's invoking as the Command object decouples the remote from the classes doing the actual home-automation work.
The RemoteLoader creates a number of Command objects that are loaded into the slots of the Remote Control. Each command object encapsulates a request of a home automation device.
All RemoteControl commands implement the Command interface, which consists of one method: execute(). Commands encapsulate a set of actions on a specific vendor class. The remote invokes these actions by calling the execute() method.
RemoteLoader
RemoteControl
onCommands
offCommands
setCommand()
onButtonWasPushed()
offButtonWasPushed()
<<interface>>
Command
execute()
Light
on()
off()
LightOnCommand
LightOffCommand
execute()
public void execute() {
light.on()
}
public void execute() {
light.off()
}
The Vendor Classes are used to perform the actual home-automation work of controlling devices. Here, we are using the Light class as an example.
Using the Command Interface, we implement each action that can be invoked by pressing a button on the remote with a simple Command object. The Command object holds a reference to an object that is an instance of a Vendor Class and implements an execute method that calls one or more methods on that object. Here we show two such classes that turn a light on and off, respectively.

Undo action

① When commands support undo, they have an undo() method that mirrors the execute() method. Whatever execute() last did, undo() reverses. So, before we can add undo to our commands, we need to add an undo() method to the Command interface:

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

Here's the new undo() method.

② Let's start with the LightOnCommand: if the LightOnCommand's execute() method was called, then the on() method was last called. We know that undo() needs to do the opposite of this by calling the off() 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

Piece of cake! Now for the LightOffCommand. Here the undo() method just needs to call the Light's on() method.

```
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

Tracking of the last command invoked

```
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.

Test

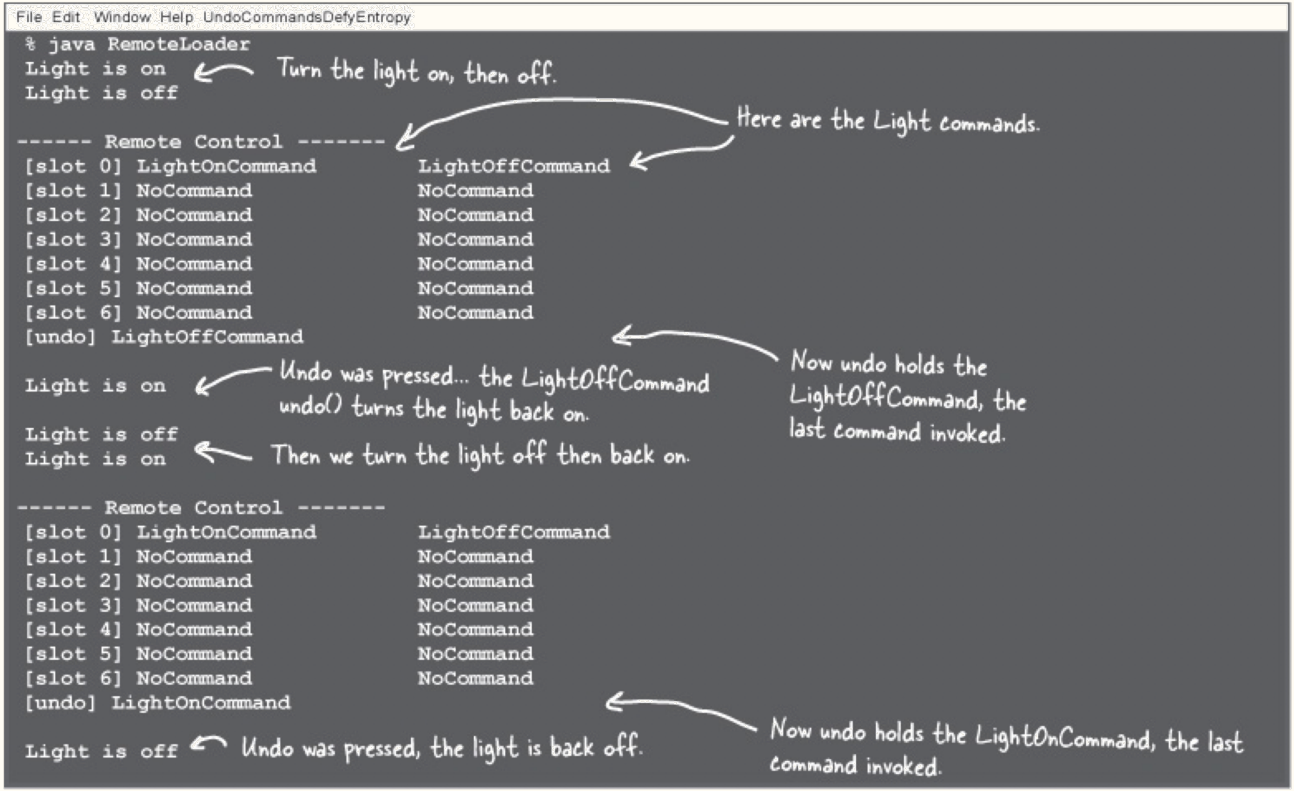
```
public class RemoteLoader {

    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.

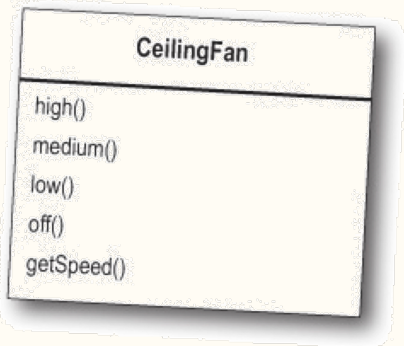
Add the light Commands to the remote in slot 0.

Turn the light on, then off and then undo.

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


File Edit Window Help UndoCommandsDefyEntropy
% java RemoteLoader
Light is on
Light is off
Turn the light on, then off.
Here are the Light commands.
------ Remote Control -------
[slot 0] LightOnCommand LightOffCommand
[slot 1] NoCommand NoCommand
[slot 2] NoCommand NoCommand
[slot 3] NoCommand NoCommand
[slot 4] NoCommand NoCommand
[slot 5] NoCommand NoCommand
[slot 6] NoCommand NoCommand
[undo] LightOffCommand
Now undo holds the LightOffCommand, the last command invoked.
Light is on
Undo was pressed... the LightOffCommand undo() turns the light back on.
Light is off
Light is on
Then we turn the light off then back on.
------ Remote Control -------
[slot 0] LightOnCommand LightOffCommand
[slot 1] NoCommand NoCommand
[slot 2] NoCommand NoCommand
[slot 3] NoCommand NoCommand
[slot 4] NoCommand NoCommand
[slot 5] NoCommand NoCommand
[slot 6] NoCommand NoCommand
[undo] LightOnCommand
Now undo holds the LightOnCommand, the last command invoked.
Light is off
Undo was pressed, the light is back off.

State information for implementation of Undo




```
public class CeilingFan {
    public static final int HIGH = 3;
    public static final int MEDIUM = 2;
    public static final int LOW = 1;
    public static final int OFF = 0;
    String location;
    int speed;

    public CeilingFan(String location) {
        this.location = location;
        speed = OFF;
    }

    public void high() {
        speed = HIGH;
        // code to set fan to high
    }

    public void medium() {
        speed = MEDIUM;
        // code to set fan to medium
    }

    public void low() {
        speed = LOW;
        // code to set fan to low
    }

    public void off() {
        speed = OFF;
        // code to turn fan off
    }

    public int getSpeed() {
        return speed;
    }
}
```

Notice that the CeilingFan class holds local state representing the speed of the ceiling fan.

These methods set the speed of the ceiling fan.

We can get the current speed of the ceiling fan using getSpeed().

```
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.


```
public class RemoteLoader {

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

        CeilingFan ceilingFan = new CeilingFan("Living Room");

        CeilingFanMediumCommand ceilingFanMedium =
                new CeilingFanMediumCommand(ceilingFan);
        CeilingFanHighCommand ceilingFanHigh =
                new CeilingFanHighCommand(ceilingFan);
        CeilingFanOffCommand ceilingFanOff =
                new CeilingFanOffCommand(ceilingFan);

        remoteControl.setCommand(0, ceilingFanMedium, ceilingFanOff);
        remoteControl.setCommand(1, ceilingFanHigh, ceilingFanOff);

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

        remoteControl.onButtonWasPushed(1);
        System.out.println(remoteControl);
        remoteControl.undoButtonWasPushed();
    }
}
```

Here we instantiate three commands: high, medium, and off.

Here we put medium in slot 0, and high in slot 1. We also load up the off command.

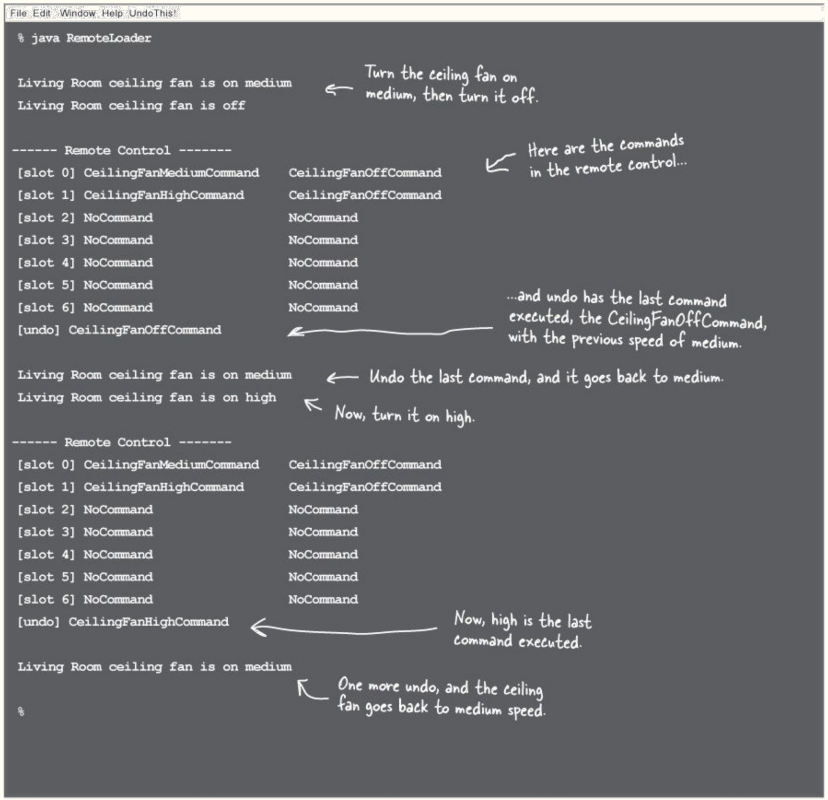
First, turn the fan on medium.

Then turn it off.

Undo! It should go back to medium.

Turn it on to high this time.

And, one more undo; it should go back to medium.


File Edit Window Help UndoThis!
% java RemoteLoader
Living Room ceiling fan is on medium
Living Room ceiling fan is off
Turn the ceiling fan on medium, then turn it off.
------ Remote Control -------
[slot 0] CeilingFanMediumCommand CeilingFanOffCommand
[slot 1] CeilingFanHighCommand CeilingFanOffCommand
[slot 2] NoCommand NoCommand
[slot 3] NoCommand NoCommand
[slot 4] NoCommand NoCommand
[slot 5] NoCommand NoCommand
[slot 6] NoCommand NoCommand
[undo] CeilingFanOffCommand
Here are the commands in the remote control...
...and undo has the last command executed, the CeilingFanOffCommand, with the previous speed of medium.
Living Room ceiling fan is on medium
Living Room ceiling fan is on high
Undo the last command, and it goes back to medium.
Now, turn it on high.
------ Remote Control -------
[slot 0] CeilingFanMediumCommand CeilingFanOffCommand
[slot 1] CeilingFanHighCommand CeilingFanOffCommand
[slot 2] NoCommand NoCommand
[slot 3] NoCommand NoCommand
[slot 4] NoCommand NoCommand
[slot 5] NoCommand NoCommand
[slot 6] NoCommand NoCommand
[undo] CeilingFanHighCommand
Now, high is the last command executed.
Living Room ceiling fan is on medium
One more undo, and the ceiling fan goes back to medium speed.
%

Macro command

① First we create the set of commands we want to go into the macro:

```
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.

② Next we create two arrays, one for the On commands and one for the Off commands, and load them with the corresponding commands:

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

```
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.

③ 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.

④ Finally, we just need to push some buttons and see if this works.

```
System.out.println(remoteControl);
System.out.println("--- Pushing Macro On---");
remoteControl.onButtonWasPushed(0);
System.out.println("--- Pushing Macro Off---");
remoteControl.offButtonWasPushed(0);
```

Here's the output

```
File  Edit  Window  Help  You Can tBeatABabka
% java RemoteLoader
------ Remote Control -------
[slot 0] MacroCommand    MacroCommand
[slot 1] NoCommand       NoCommand
[slot 2] NoCommand       NoCommand
[slot 3] NoCommand       NoCommand
[slot 4] NoCommand       NoCommand
[slot 5] NoCommand       NoCommand
[slot 6] NoCommand       NoCommand
[undo] NoCommand

--- Pushing Macro On---
Light is on
Living Room stereo is on
Living Room TV is on
Living Room TV channel is set for DVD
Hottub is heating to a steaming 104 degrees
Hottub is bubbling!

--- Pushing Macro Off---
Light is off
Living Room stereo is off
Living Room TV is off
Hottub is cooling to 98 degrees
```

Here are the two macro commands.

All the Commands in the macro are executed when we invoke the on macro...

and when we invoke the off macro. Looks like it works.