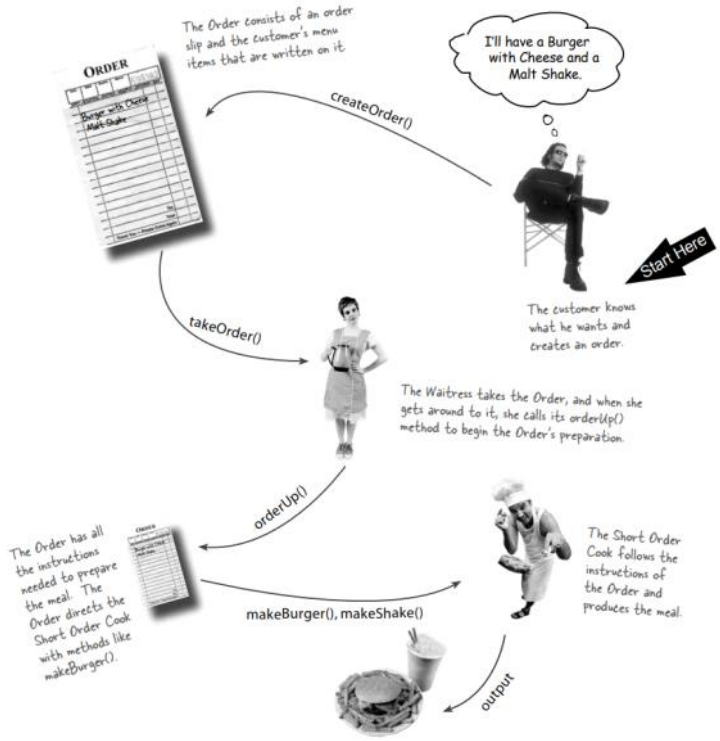
COMMAND DESIGN PATTERN

Tuesday, September 12, 2023 2:28 PM

Need?

→ when we want to design a system in which we 'just order something/command and we doesn't bother about how its being proresses then Command Design pattern would be apt to apply.

Taking an Example of Resturent/outlet



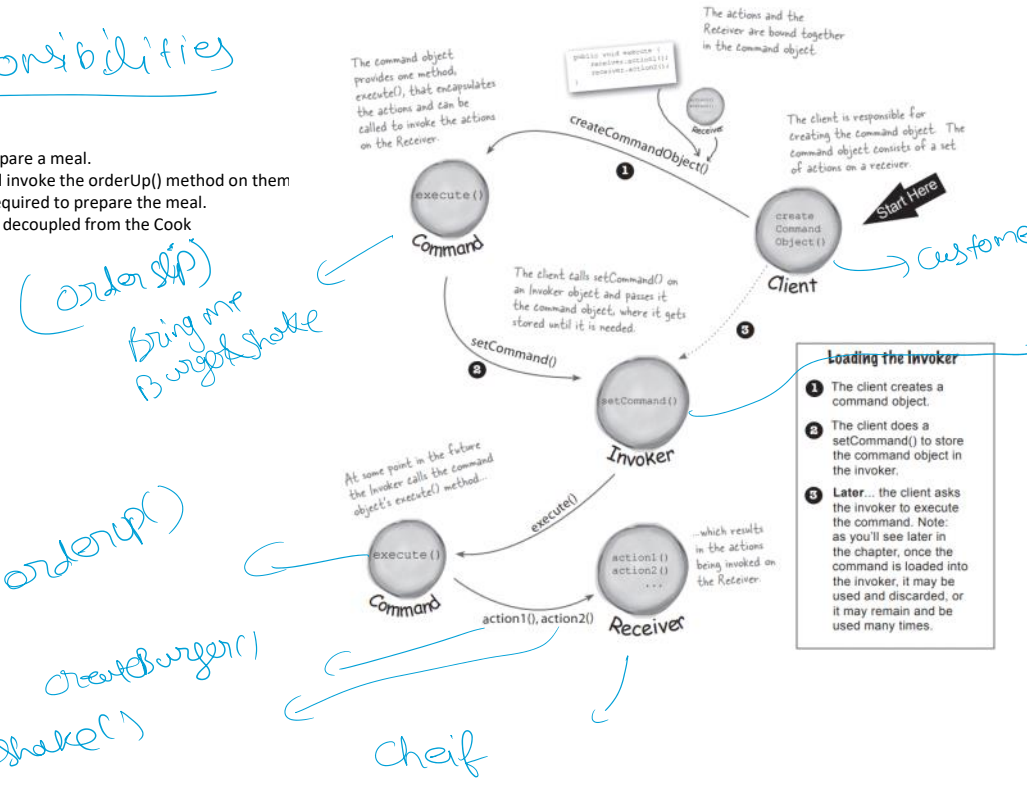
→ Customer order's something in order slip to waitress.

→ waitress pass on order to chelf.

→ chelf prepares order.

Roles & Responsibilities

->An Order Slip encapsulates a request to prepare a meal.
->The Waitress's job is to take Order Slips and invoke the orderUp() method on them
->The Short Order Cook has the knowledge required to prepare the meal.
Note: we have a Diner with a Waitress who is decoupled from the Cook



(order slip) Bring me Burger&shake

customer

waitress

orderUp()

createBurger()

makeshake()

Cheif

Match the diner objects and methods with the corresponding names from the Command Pattern.

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

```
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.

```
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.

```
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.

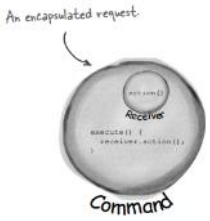
Here's the output of running this test code!

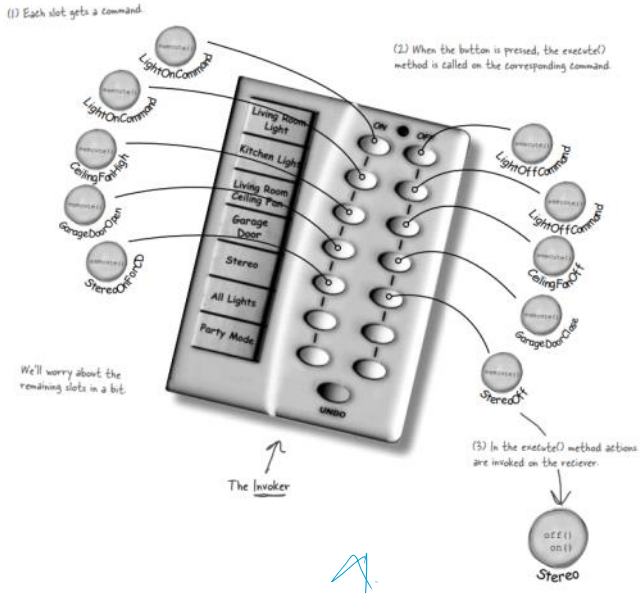
```
File Edit Window Help DinerFoodYum
%java RemoteControlTest
Light is On
%
```

This code is to create a remote to turn on/off lights of home.

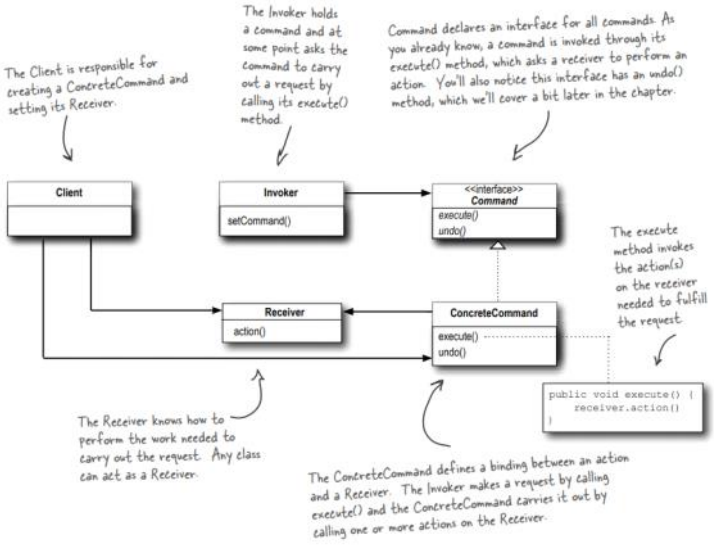
Here we are trying to create a remote ControlHome.

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.



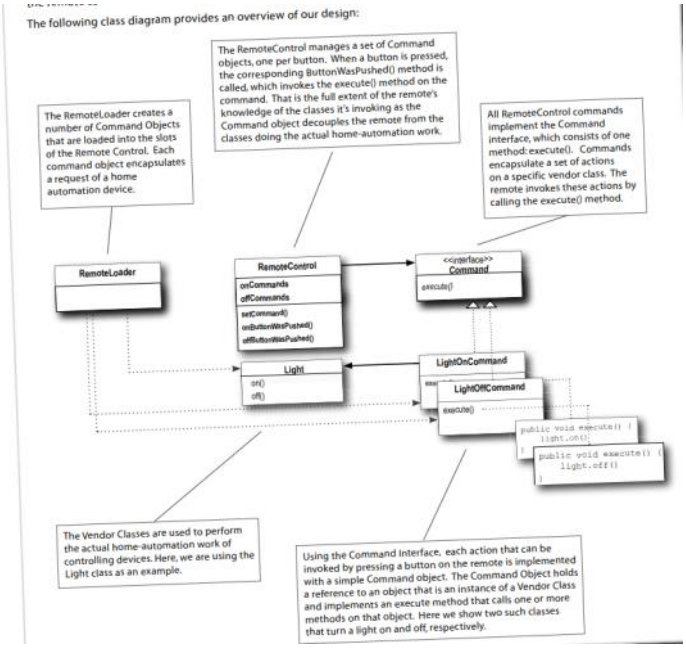


the class diagram



Remote Controll Home diagram.

Generalize class diagram for command pattern.




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

```
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() { }
}
```

using NoCommand nothing to be executed unnecessary

Using NoCommand when nothing to be done in some scenario. It helps to avoid unnecessary null checks!

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();
    }

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

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

On slots

Off Slots

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."

```
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 are binding the receiver to a different action: the off() method.

```
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);

        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 all the devices in their proper locations.

Create all the Light Command objects.

Create the On and Off for the ceiling fan.

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.

Implementing Undo Command

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

Here's the new undo() method.

```
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++) {
```

This is where we'll stash the last command executed for the undo button.

```
    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!

```
$ java RemoteLoader
Light is on
Light is off

------ Remote Control -------
[slot 0] headfirst.command.undo.LightOnCommand    headfirst.command.undo.LightOffCommand
[slot 1] headfirst.command.undo.NoCommand    headfirst.command.undo.NoCommand
[slot 2] headfirst.command.undo.NoCommand    headfirst.command.undo.NoCommand
[slot 3] headfirst.command.undo.NoCommand    headfirst.command.undo.NoCommand
[slot 4] headfirst.command.undo.NoCommand    headfirst.command.undo.NoCommand
[slot 5] headfirst.command.undo.NoCommand    headfirst.command.undo.NoCommand
[slot 6] headfirst.command.undo.NoCommand    headfirst.command.undo.NoCommand
[undo] headfirst.command.undo.LightOffCommand

Light is on
Light is off
Light is on

------ Remote Control -------
[slot 0] headfirst.command.undo.LightOnCommand    headfirst.command.undo.LightOffCommand
[slot 1] headfirst.command.undo.NoCommand    headfirst.command.undo.NoCommand
[slot 2] headfirst.command.undo.NoCommand    headfirst.command.undo.NoCommand
[slot 3] headfirst.command.undo.NoCommand    headfirst.command.undo.NoCommand
[slot 4] headfirst.command.undo.NoCommand    headfirst.command.undo.NoCommand
[slot 5] headfirst.command.undo.NoCommand    headfirst.command.undo.NoCommand
[slot 6] headfirst.command.undo.NoCommand    headfirst.command.undo.NoCommand
[undo] headfirst.command.undo.LightOnCommand

Light is off
```
Turn the light on, then off.
Here's the Light commands.
Undo was pressed... the LightOffCommand undo() turns the light back on.
Now undo holds the LightOffCommand, the last command invoked.
Then we turn the light off then back on.
Undo was pressed, the light is back off.
Now undo holds the LightOnCommand, the last command invoked.

```
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...
}
}
```
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.

```
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.

It was easy to implement undo on light, what about fan?

```
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);
```
Here we instantiate three commands: high, medium, and off.
Here we put medium in

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

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

```
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 zero, and high in slot one. We also load up the off commands.

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.

```
% java RemoteLoader

Living Room ceiling fan is on medium
Living Room ceiling fan is off

------ Remote Control -------
[slot 0] headfirst.command.undo.NoCommand    headfirst.command.undo.NoCommand
[slot 1] headfirst.command.undo.CeilingFanMediumCommand    headfirst.command.undo.CeilingFanOff-
Command
[slot 2] headfirst.command.undo.CeilingFanHighCommand    headfirst.command.undo.CeilingFanOffCom-
mand
[slot 3] headfirst.command.undo.NoCommand    headfirst.command.undo.NoCommand
[slot 4] headfirst.command.undo.NoCommand    headfirst.command.undo.NoCommand
[slot 5] headfirst.command.undo.NoCommand    headfirst.command.undo.NoCommand
[slot 6] headfirst.command.undo.NoCommand    headfirst.command.undo.NoCommand
[undo] headfirst.command.undo.CeilingFanOffCommand

Living Room ceiling fan is on medium
Living Room ceiling fan is on high

------ Remote Control -------
[slot 0] headfirst.command.undo.NoCommand    headfirst.command.undo.NoCommand
[slot 1] headfirst.command.undo.CeilingFanMediumCommand    headfirst.command.undo.CeilingFanOff-
Command
[slot 2] headfirst.command.undo.CeilingFanHighCommand    headfirst.command.undo.CeilingFanOffCom-
mand
[slot 3] headfirst.command.undo.NoCommand    headfirst.command.undo.NoCommand
[slot 4] headfirst.command.undo.NoCommand    headfirst.command.undo.NoCommand
[slot 5] headfirst.command.undo.NoCommand    headfirst.command.undo.NoCommand
[slot 6] headfirst.command.undo.NoCommand    headfirst.command.undo.NoCommand
[undo] headfirst.command.undo.CeilingFanHighCommand

Living Room ceiling fan is on medium
%
```

Turn the ceiling fan on medium, then turn it off.

Here are the commands in the remote control.

...and undo has the last command executed, the CeilingFanOffCommand.

Undo the last command, and it goes back to medium.

Now, turn it on high.

Now, high is the last command executed.

One more undo, and the ceiling fan goes back to medium speed.

Output

Testing Undo

How about creating a party mode?

→ In which a lot of device turns on/off by a single command.

→ Here comes role of MacroCommand.

It basically stands for a Command which execute a couple of minor commands under it.


```
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.

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:

```
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 and an array for Off commands...

...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
```

}

Q: How can I implement a history of undo operations? In other words, I want to be able to press the undo button multiple times.

A: Great question! It's pretty easy actually; instead of keeping just a reference to the last Command executed, you keep a stack of previous commands. Then, whenever undo is pressed, your invoker pops the first item off the stack and calls its undo() method.

```
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] headfirst.command.party.MacroCommand    headfirst.command.party.MacroCommand
[slot 1] headfirst.command.party.NoCommand       headfirst.command.party.NoCommand
[slot 2] headfirst.command.party.NoCommand       headfirst.command.party.NoCommand
[slot 3] headfirst.command.party.NoCommand       headfirst.command.party.NoCommand
[slot 4] headfirst.command.party.NoCommand       headfirst.command.party.NoCommand
[slot 5] headfirst.command.party.NoCommand       headfirst.command.party.NoCommand
[slot 6] headfirst.command.party.NoCommand       headfirst.command.party.NoCommand
[undo] headfirst.command.party.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.

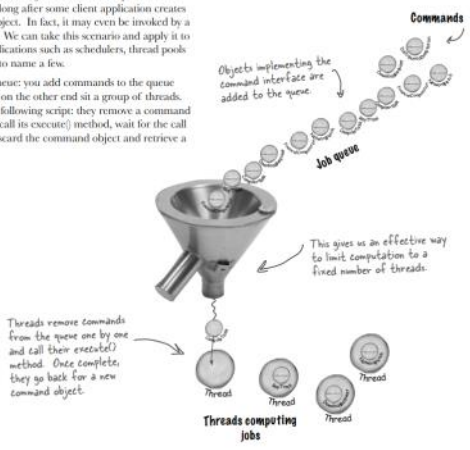
uses of Command pattern in Async execution

Use in logging of systems

More uses of the Command Pattern: queuing requests

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.

Imagine a job queue: you add commands to the queue on one end, and on the other end sit a group of threads. Threads run the following script: they remove a command from the queue, call its execute() method, wait for the call to finish, then discard the command object and retrieve a new one.



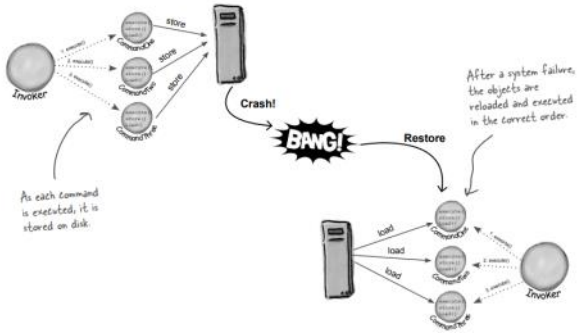
More uses of the Command Pattern: logging requests

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(). In Java we could use object serialization to implement these methods, but the normal caveats for using serialization for persistence apply.

How does this 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.

Now, this kind of logging wouldn't make sense for a remote control; however, there are many applications that invoke actions on large data structures that can't be quickly saved each time a change is made. By using logging, we can save all the operations since the last check point, 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. In more advanced applications, these techniques can be extended to apply to sets of operations in a transactional manner so that all of the operations complete, or none of them do.

We add two methods for logging



YOUTUBE CHANNEL :
https://www.youtube.com/@ShubhamHaritash