Command Design Pattern

> Designed by ZHU Yueming in 2022 Dec. 14th
>
> The the code in this tutorial is based on and modified from "Head First Design Pattern"
>
> The UML diagram in this tutorial is referened from "Head First Design Pattern"

Experimental Objective

Learn how to refactor source code to command design pattern.

1. Introduce of source code

Requirement introduction :

We want to design a remote control to turn on or turn off of lights and air conditioners in different rooms. Here we provide the `Light` and `AirConditioner` class:

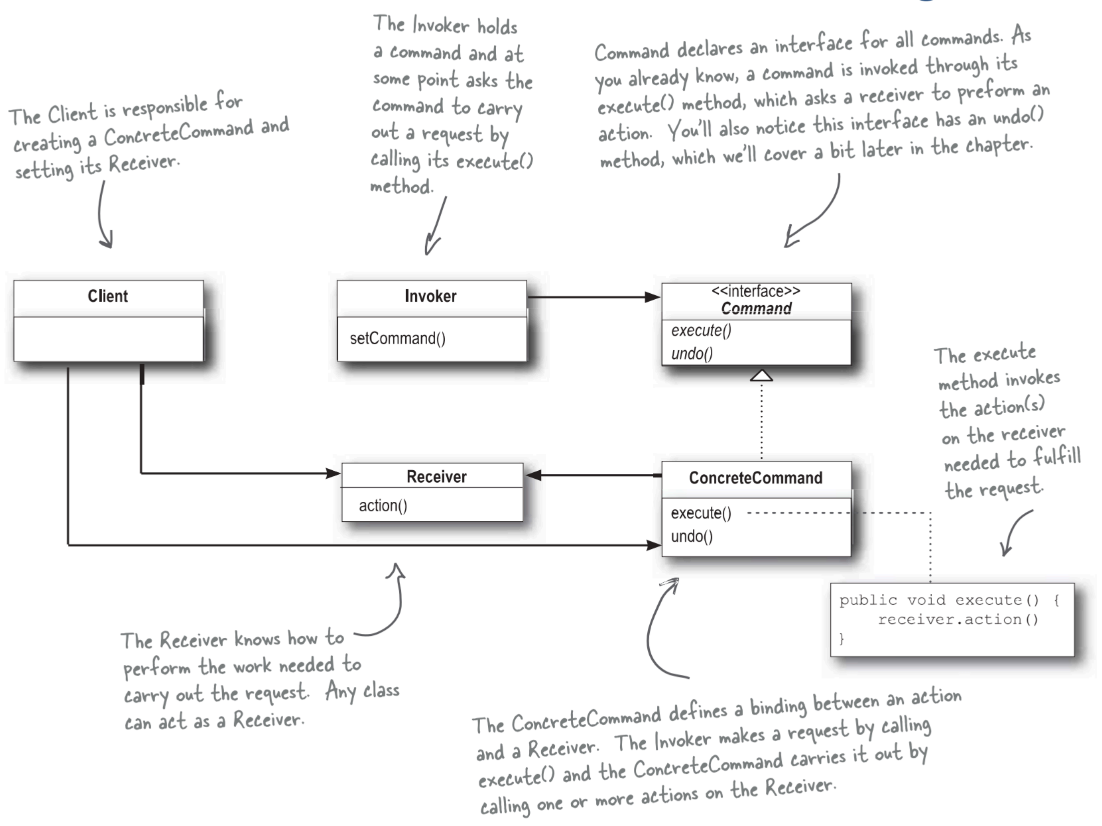
How to use:

- Enter 1 to turn on the air conditioner in bedroom.
- Enter 2 to turn off the air conditioner in bedroom.
- Enter 3 to turn on the air conditioner in living room.
- Enter 4 to turn off the air conditioner in living room.
- Enter 5 to turn on the light in bedroom.
- Enter 6 to turn off the light in bedroom.
- Enter 7 to turn on the light in living room.
- Enter 8 to turn off the light in living room.
- Enter 9 to show the status of all rooms
- Enter 0 to shop the program

```
Please input operation number: 1-9,[1,3,5,7] is on command,[2,4,6,8] is off command, 9 is to show state terminate by 0:
1
bedRoom: Air Conditioner is On
2
bedRoom: Air Conditioner is Off
3
livingRoom: Air Conditioner is On
4
livingRoom: Air Conditioner is Off
5
bedRoom: Light is On
6
bedRoom: Light is Off
7
livingRoom: Light is On
8
livingRoom: Light is Off
9
bedRoom: Air Conditioner is Off
livingRoom: Air Conditioner is Off
bedRoom: Light is Off
livingRoom: Light is Off
```

Command Design pattern

When we encapsulate the command into an object, the **sending** and **execution** of the command can be decoupled, and then we can perform more complex operations on the command. For example: queue, undo, delay, storage, logging, etc.


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 preform 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()
The execute method invokes the action(s) on the receiver needed to fulfill the request.
Receiver
action()
ConcreteCommand
execute()
undo()
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 out by calling one or more actions on the Receiver.

In original code, we only provide two command receiver class: `Lignt` and `AirConditioner`

Task 1. Refactoring to Command Pattern

Our next step is to design a remote control that can separate commands creation from execution. For example: click 1-8 to create 8 different commands, click 9 to display the status, and click 10 to execute the command according to the first-in-first-out principle.

For example:

```
Please input operation number to add a command: 1-9,[1,3,5,7] is on command,
[2,4,6,8] is off command, 9 is to show state. 10 is to execute command.
terminate by 0:
1 3 5 7 2 4
10
bedRoom: Air Conditioner is On
10
livingRoom: Air Conditioner is On
10
bedRoom: Light is On
10
livingRoom: Light is On
10
bedRoom: Air Conditioner is Off
10
livingRoom: Air Conditioner is Off
0
```

The client would be:

```
public class Client_Task1 {
    public static void main(String[] args) {
        AirConditioner roomAirConditioner = new AirConditioner("bedRoom");
        AirConditioner livingAirConditioner = new AirConditioner("livingRoom");
        Light roomLight = new Light("bedRoom");
        Light livingLight = new Light("livingRoom");

        Command[] commands = new Command[8];
        commands[0] = new AirConditionerOnCommand(roomAirConditioner);
        commands[1] = new AirConditionerOffCommand(roomAirConditioner);
        commands[2] = new AirConditionerOnCommand(livingAirConditioner);
        commands[3] = new AirConditionerOffCommand(livingAirConditioner);
        commands[4] = new LightOnCommand(roomLight);
        commands[5] = new LightOnCommand(roomLight);
        commands[6] = new LightOnCommand(livingLight);
        commands[7] = new LightOnCommand(livingLight);

        RemoteCommandQueue remoteCommandQueue = new RemoteCommandQueue();
        Scanner input = new Scanner(System.in);
        System.out.println("Please input operation number to add a command: 1-
9," +
                "[1,3,5,7] is on command,[2,4,6,8] is off command, " +
                "9 is to show state. 10 is to execute command. terminate by
0:");
        int op = 0;
        do {
            try {
                op = input.nextInt();
                switch (op) {
                    case 1: case 2: case 3: case 4: case 5: case 6: case 7:
case 8:
                        remoteCommandQueue.buttonPressed(commands[op-1]);
                        break;
                    case 9:
                        showState(new AirConditioner[]{roomAirConditioner,
livingAirConditioner}
                                , new Light[]{roomLight, livingLight});
                        break;
                    case 10:
                        remoteCommandQueue.commandExecute();
                        break;
                }
            } catch (InputMismatchException e) {
                System.out.println("Exception:" + e);
                input.nextLine();
            }
        } while (op != 0);
        input.close();
    }

    public static void showState(AirConditioner[] airConditioners, Light[]
lights) {
        for (AirConditioner a : airConditioners) {
            System.out.println(a);
        }
        for (Light l : lights) {
            System.out.println(l);
        }
    }
}
```

How to implement?

Step1: design command:

Here we provide an interface named command :

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

Then we will design four concrete command class to implement the interface, including:

```
AirConditionerOnCommand
AirConditionerOffCommand
LightOnCommand
LightOffCommand
```

AirConditionerOnCommand

```
public class AirConditionerOnCommand implements Command {
    AirConditioner airConditioner;

    public AirConditionerOnCommand(AirConditioner airConditioner) {
        this.airConditioner = airConditioner;
    }

    @Override
    public void execute() {
        airConditioner.on();
    }
}
```

Step2: design Invoker

create a Class named `RemoteCommandQueue` and complete the following methods:

```
public class RemoteCommandQueue {
    Queue<Command> commands;

    public RemoteCommandQueue() {
        commands = new ArrayDeque<>();
    }

    /**
     * only add command but not execute
     * @param command
     */
    public void buttonPressed(Command command) {
        //todo: complete

    }

    /**
     * execute the command in the queue by first-in-first-out principle
     * if there is no command in the queue display "no command"
     */
    public void commandExecute() {
     // todo: compelte
    }
}
```

Task 2. Add undo command

After executing a command, we want to execute the undo command again, and the command just executed is considered invalid.

Add another operation 11, which means undo previous command:

For example:

```
Please input operation number to add a command: 1-9,[1,3,5,7] is on command,
[2,4,6,8] is off command, 9 is to show state. 10 is to execute command. 11 is
undo previous command. terminate by 0:
1 3 5
10
bedRoom: Air Conditioner is On
10
livingRoom: Air Conditioner is On
11
livingRoom: Air Conditioner is Off
9
bedRoom: Air Conditioner is On
livingRoom: Air Conditioner is Off
bedRoom: Light is Off
livingRoom: Light is Off
```

How to implement?

Step1: Modify Command interface

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

- Modify those four concrete Command classes. Class `AirConditionerOnCommand` servers as an example:

```
public class AirConditionerOnCommand implements Command {
    AirConditioner airConditioner;

    public AirConditionerOnCommand(AirConditioner airConditioner) {
        this.airConditioner = airConditioner;
    }

    @Override
    public void execute() {
        airConditioner.on();
    }

    @Override
    public void undo() {
        airConditioner.off();
    }
}
```

Step2: Modify Invoker class

```
public class RemoteCommandQueue {
    Queue<Command> commands;
    Command undoCommand;//record the previous command

    public RemoteCommandQueue() {
        commands = new ArrayDeque<>();
    }

    /**
     * only add command but not execute
     *
     * @param command
     */
    public void buttonPressed(Command command) {

    }


    /**
     * execute the command in the queue by first-in-first-out principle
     * if there is no command in the queue display "no command"
     */
    public void commandExecute() {

    }

    /**
     * undo the previous command
     */
    public void commandUndo() {

    }
}
```

Step 3: Modify Client Class

Add 11 operation number to undo the command just executed.