Home Automation System with Command Pattern

public class CommandPatternExample {

public static void main(String[] args) {

// Create receiver objects

Light livingRoomLight = new Light("Living Room");

Light kitchenLight = new Light("Kitchen");

Fan ceilingFan = new Fan("Ceiling Fan");

// Create commands

Command livingRoomLightOn = new LightOnCommand(livingRoomLight);

Command livingRoomLightOff = new LightOffCommand(livingRoomLight);

Command kitchenLightOn = new LightOnCommand(kitchenLight);

Command kitchenLightOff = new LightOffCommand(kitchenLight);

Command fanOn = new FanOnCommand(ceilingFan);

Command fanOff = new FanOffCommand(ceilingFan);

Command fanHigh = new FanHighCommand(ceilingFan);

Command fanMedium = new FanMediumCommand(ceilingFan);

// Create invoker

RemoteControl remote = new RemoteControl();

System.out.println("=== Testing Simple Commands ===");

remote.setCommand(livingRoomLightOn);

remote.pressButton();

remote.setCommand(kitchenLightOn);

remote.pressButton();

remote.setCommand(fanOn);

remote.pressButton();

System.out.println("\n=== Testing Undo ===");

remote.setCommand(livingRoomLightOff);

remote.pressButton();

remote.undo();

remote.setCommand(fanHigh);

remote.pressButton();

remote.setCommand(fanMedium);

remote.pressButton();

remote.undo();

System.out.println("\n=== Testing Macro Command ===");

Command[] partyOn = {livingRoomLightOn, kitchenLightOn, fanOn};

Command[] partyOff = {livingRoomLightOff, kitchenLightOff, fanOff};

MacroCommand partyModeOn = new MacroCommand(partyOn);

MacroCommand partyModeOff = new MacroCommand(partyOff);

remote.setCommand(partyModeOn);

remote.pressButton();

remote.setCommand(partyModeOff);

remote.pressButton();

remote.undo();

}

// Command Interface

interface Command {

void execute();

void undo();

}

// Receiver - Light

static class Light {

private String location;

private boolean isOn;

public Light(String location) {

this.location = location;

}

public void on() {

isOn = true;

System.out.println(location + " light is ON");

}

public void off() {

isOn = false;

System.out.println(location + " light is OFF");

}

public boolean isOn() {

return isOn;

}

}

// Receiver - Fan

static class Fan {

private String location;

private int speed; // 0 = off, 1 = low, 2 = medium, 3 = high

private int previousSpeed;

public Fan(String location) {

this.location = location;

speed = 0;

}

public void high() {

previousSpeed = speed;

speed = 3;

System.out.println(location + " fan is on HIGH");

}

public void medium() {

previousSpeed = speed;

speed = 2;

System.out.println(location + " fan is on MEDIUM");

}

public void on() {

previousSpeed = speed;

speed = 1;

System.out.println(location + " fan is on LOW");

}

public void off() {

previousSpeed = speed;

speed = 0;

System.out.println(location + " fan is OFF");

}

public int getSpeed() {

return speed;

}

public void setSpeed(int speed) {

previousSpeed = this.speed;

this.speed = speed;

}

}

// Concrete Commands for Light

static class LightOnCommand implements Command {

private Light light;

public LightOnCommand(Light light) {

this.light = light;

}

@Override

public void execute() {

light.on();

}

@Override

public void undo() {

light.off();

}

}

static class LightOffCommand implements Command {

private Light light;

public LightOffCommand(Light light) {

this.light = light;

}

@Override

public void execute() {

light.off();

}

@Override

public void undo() {

light.on();

}

}

// Concrete Commands for Fan

static class FanOnCommand implements Command {

private Fan fan;

private int prevSpeed;

public FanOnCommand(Fan fan) {

this.fan = fan;

}

@Override

public void execute() {

prevSpeed = fan.getSpeed();

fan.on();

}

@Override

public void undo() {

if (prevSpeed == 0) {

fan.off();

} else if (prevSpeed == 1) {

fan.on();

} else if (prevSpeed == 2) {

fan.medium();

} else if (prevSpeed == 3) {

fan.high();

}

}

}

static class FanOffCommand implements Command {

private Fan fan;

private int prevSpeed;

public FanOffCommand(Fan fan) {

this.fan = fan;

}

@Override

public void execute() {

prevSpeed = fan.getSpeed();

fan.off();

}

@Override

public void undo() {

if (prevSpeed == 0) {

fan.off();

} else if (prevSpeed == 1) {

fan.on();

} else if (prevSpeed == 2) {

fan.medium();

} else if (prevSpeed == 3) {

fan.high();

}

}

}

static class FanHighCommand implements Command {

private Fan fan;

private int prevSpeed;

public FanHighCommand(Fan fan) {

this.fan = fan;

}

@Override

public void execute() {

prevSpeed = fan.getSpeed();

fan.high();

}

@Override

public void undo() {

if (prevSpeed == 0) {

fan.off();

} else if (prevSpeed == 1) {

fan.on();

} else if (prevSpeed == 2) {

fan.medium();

} else if (prevSpeed == 3) {

fan.high();

}

}

}

static class FanMediumCommand implements Command {

private Fan fan;

private int prevSpeed;

public FanMediumCommand(Fan fan) {

this.fan = fan;

}

@Override

public void execute() {

prevSpeed = fan.getSpeed();

fan.medium();

}

@Override

public void undo() {

if (prevSpeed == 0) {

fan.off();

} else if (prevSpeed == 1) {

fan.on();

} else if (prevSpeed == 2) {

fan.medium();

} else if (prevSpeed == 3) {

fan.high();

}

}

}

// Macro Command

static class MacroCommand implements Command {

private Command[] commands;

public MacroCommand(Command[] commands) {

this.commands = commands;

}

@Override

public void execute() {

for (Command command : commands) {

command.execute();

}

}

@Override

public void undo() {

for (int i = commands.length - 1; i >= 0; i--) {

commands[i].undo();

}

}

}

// Invoker

static class RemoteControl {

private Command command;

private Command lastCommand;

public void setCommand(Command command) {

this.command = command;

}

public void pressButton() {

command.execute();

lastCommand = command;

}

public void undo() {

System.out.println("--- Undo last command ---");

if (lastCommand != null) {

lastCommand.undo();

} else {

System.out.println("No command to undo");

}

}

}

}

Key Features:

1. **Command Objects**: Each action is encapsulated in its own command object
2. **Undo Functionality**: Commands support undo operations
3. **Macro Commands**: Multiple commands can be combined
4. **Decoupling**: Invoker knows nothing about receivers
5. **Extensibility**: Easy to add new commands

Command Pattern Benefits:

1. **Decoupling**: Invoker and receivers are loosely coupled
2. **Undo/Redo**: Easy to implement undo functionality
3. **Queueing**: Commands can be queued for later execution
4. **Logging**: Commands can be logged for audit trails
5. **Macro Commands**: Complex operations can be composed from simple ones

Output:

