23 Behavioral: Command Pattern — Smart-Home Remote-Control Assignment

**Goal** Create a fully undoable smart-home remote based on the **Command pattern**. Each action (turning a light on/off, changing a fan’s speed, altering an AC set-point) is encapsulated as a Command object. The remote (Invoker) triggers commands, stores history, and can call **undo()**. Every class must carry short JavaDoc. A brief **Reflection** at the end explains design trade-offs.

.

├── analysis

│ └── command\_need.md ← problem statement & reasoning

├── src/main/java

│ └── home

│ ├── command

│ │ ├── Command.java

│ │ ├── LightOnCommand.java

│ │ ├── LightOffCommand.java

│ │ ├── FanIncSpeedCommand.java

│ │ ├── FanDecSpeedCommand.java

│ │ ├── AcOnCommand.java

│ │ ├── AcOffCommand.java

│ │ └── AcSetTempCommand.java

│ ├── invoker

│ │ ├── RemoteSlot.java

│ │ └── RemoteControl.java

│ └── receiver

│ ├── Light.java

│ ├── Fan.java

│ └── Ac.java

│ └── CommandDemo.java

├── src/test/java/home

│ ├── UndoLightTest.java

│ ├── QueueCommandsTest.java

│ └── DynamicAssignTest.java

├── reflection.md

└── README.md

#### **1 Command interface**

package home.command;

/\*\*

\* Command object that can be executed and undone.

\*/

public interface Command {

/\*\* Perform the requested action. \*/

void execute();

/\*\* Revert {@link #execute()}. \*/

void undo();

}

#### **2 Receivers**

package home.receiver;

/\*\* Simple on/off light bulb. \*/

public class Light {

private boolean on;

public void on(){ on=true; System.out.println("Light ON"); }

public void off(){ on=false; System.out.println("Light OFF"); }

public boolean isOn(){ return on; }

}

package home.receiver;

/\*\* Ceiling fan with incremental speed control. \*/

public class Fan {

private int speed;

public void inc(){ System.out.println("Fan speed "+ ++speed); }

public void dec(){ speed=Math.max(0,speed-1); System.out.println("Fan speed "+speed); }

public int speed(){ return speed; }

}

package home.receiver;

/\*\* Air-conditioner with power toggle and temperature set-point. \*/

public class Ac {

private boolean on; private int temp=24;

public void on(){ on=true; System.out.println("AC ON"); }

public void off(){ on=false; System.out.println("AC OFF"); }

public void set(int t){ temp=t; System.out.println("AC temp "+t); }

public boolean isOn(){ return on; } public int temp(){ return temp; }

}

#### **3 Concrete Commands (samples)**

package home.command;

import home.receiver.Light;

/\*\* Turn a {@link Light} ON; undo turns it OFF. \*/

public class LightOnCommand implements Command {

private final Light light;

public LightOnCommand(Light l){ this.light=l; }

public void execute(){ light.on(); }

public void undo(){ light.off(); }

}

package home.command;

import home.receiver.Fan;

/\*\* Increase fan speed; undo decreases it. \*/

public class FanIncSpeedCommand implements Command {

private final Fan fan;

public FanIncSpeedCommand(Fan f){ fan=f; }

public void execute(){ fan.inc(); }

public void undo(){ fan.dec(); }

}

package home.command;

import home.receiver.Ac;

/\*\* Change AC set-point, restoring previous value on undo. \*/

public class AcSetTempCommand implements Command {

private final Ac ac; private final int target; private int prev;

public AcSetTempCommand(Ac ac,int t){ this.ac=ac; this.target=t; }

public void execute(){ prev=ac.temp(); ac.set(target); }

public void undo(){ ac.set(prev); }

}

(Analogous classes: LightOffCommand, FanDecSpeedCommand, AcOnCommand, AcOffCommand.)

#### **4 Invoker**

package home.invoker;

import home.command.Command;

/\*\*

\* One button/slot on a remote-control.

\*/

public class RemoteSlot {

private Command cmd;

public void set(Command c){ cmd=c; }

public void press(){ if(cmd!=null) cmd.execute(); }

public void undo(){ if(cmd!=null) cmd.undo(); }

}

package home.invoker;

import java.util.\*;

/\*\* Remote with configurable slots and built-in undo history. \*/

public class RemoteControl {

private final List<RemoteSlot> slots;

private final Deque<Command> history = new ArrayDeque<>();

public RemoteControl(int n){ slots=new ArrayList<>(n); for(int i=0;i<n;i++) slots.add(new RemoteSlot()); }

public RemoteSlot slot(int i){ return slots.get(i); }

public void press(int idx){ slots.get(idx).press(); history.push(slots.get(idx)::undo); }

public void undoLast(){ if(!history.isEmpty()) history.pop().undo(); }

}

#### **5 Demo client**

package home;

import home.receiver.\*; import home.command.\*; import home.invoker.\*;

public class CommandDemo {

public static void main(String[] args){

Light light = new Light();

Fan fan = new Fan();

Ac ac = new Ac();

RemoteControl rc = new RemoteControl(3);

rc.slot(0).set(new LightOnCommand(light));

rc.slot(1).set(new FanIncSpeedCommand(fan));

rc.slot(2).set(new AcSetTempCommand(ac, 20));

rc.press(0); // Light ON

rc.press(1); // Fan speed 1

rc.press(2); // AC temp 20

rc.undoLast(); // AC temp 24 (undo)

}

}

Console excerpt

Light ON

Fan speed 1

AC temp 20

AC temp 24

#### **6 JUnit-style tests (outline)**

/\* UndoLightTest.java \*/

Light bulb = new Light();

Command on = new LightOnCommand(bulb);

on.execute(); assertTrue(bulb.isOn());

on.undo(); assertFalse(bulb.isOn());

/\* QueueCommandsTest.java \*/

Deque<Command> q = new ArrayDeque<>();

Fan fan = new Fan();

q.add(new FanIncSpeedCommand(fan));

q.add(new FanIncSpeedCommand(fan));

while(!q.isEmpty()) q.pop().execute();

assertEquals(2, fan.speed());

/\* DynamicAssignTest.java \*/

RemoteControl rc = new RemoteControl(1);

rc.slot(0).set(new LightOnCommand(new Light()));

rc.press(0);

rc.slot(0).set(new FanIncSpeedCommand(new Fan())); // re-assign

rc.press(0);

## **reflection.md**

The Command pattern cleanly **decouples** UI controls from device logic.  
 *Benefits*

* Undo/redo via stored commands (history stack).
* Macro or scheduling: commands can be queued for later.
* Adding a new device action requires only another Command class; existing invoker code stays unchanged (Open/Closed Principle).

*Trade-offs*

* **Class proliferation** – many small command classes. Use lambdas in Java 8+ if the action is trivial.
* **Memory** – long histories increase RAM; cap history size or persist to disk.
* **Latency** – negligible here, but heavy command objects could add overhead.

Overall, Command provides maximal flexibility for a smart-home controller where actions must be reconfigurable, undoable, and loggable.