Command Design Pattern

The **Command Design Pattern** is a **behavioral pattern** used to **encapsulate a request as an object**, allowing you to parameterize clients with queues, logs, and undo/redo operations. It's **decoupling the sender of a request from its receiver**.

✅ Real-World Analogy: Remote Control

Imagine a remote control with buttons (`Invoker`) that can control various devices like lights, fans (`Receiver`). The actions (e.g. `turnOnLight`) are encapsulated as **command objects** (`Command`) — allowing you to plug and play actions without knowing device internals.

✅ Key Participants

Role	Description
Command	Declares interface (e.g. `execute()`)
ConcreteCommand	Implements `Command`, defines a binding between Receiver and actions
Receiver	Performs the action
Invoker	Calls `execute()` on the command
Client	Creates and configures commands

✅ Java Implementation

👇 Step-by-step: Smart Home Command Example

1. Command Interface

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

2. Receivers

```
public class Light {
    public void turnOn() {
        System.out.println("Light ON");
    }

    public void turnOff() {
        System.out.println("Light OFF");
    }
}
```

3. Concrete Commands

```
public class LightOnCommand implements Command {
    private Light light;

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

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

public class LightOffCommand implements Command {
    private Light light;

    public LightOffCommand(Light light) {
        this.light = light;
    }

    public void execute() {
        light.turnOff();
    }
}
```

4. Invoker

```
public class RemoteControl {
    private Command command;

    public void setCommand(Command command) {
        this.command = command;
    }

    public void pressButton() {
        command.execute();
    }
}
```

5. Client

```
public class Main {
    public static void main(String[] args) {
        Light light = new Light();
        Command on = new LightOnCommand(light);
        Command off = new LightOffCommand(light);

        RemoteControl remote = new RemoteControl();

        remote.setCommand(on);
        remote.pressButton(); // Light ON

        remote.setCommand(off);
        remote.pressButton(); // Light OFF
    }
}
```

✅ Use Cases of Command Pattern

Use Case	Description
GUI buttons and menu actions	Each button maps to a Command
Task queues / job scheduling	Queue up commands for background processing
Transaction-based systems	Log commands for redo/undo

Use Case	Description
Macro recording	Record command sequence
Remote controls	Abstract interaction from actual action

---

how exactly “what to do” and “who does it” are separated in that code.
Let’s break it down **super simply** 👇

🧩 Step 1 — Who does the *actual work*?

```
public class Light {
    public void turnOn() { System.out.println("Light ON"); }
    public void turnOff() { System.out.println("Light OFF"); }
}
```

👉 The `Light` class is the **Receiver** — it knows *how* to turn on or off.
So this is **“who does the work.”**

🧠 Step 2 — What needs to be done?

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

👉 The `Command` interface just says *“there is some work to do.”*
But it doesn’t care *who* will do it or *how*.

⚙️ Step 3 — Concrete Commands decide “what action to take”

```
public class LightOnCommand implements Command {
    private Light light;
    public LightOnCommand(Light light) { this.light = light; }
    public void execute() { light.turnOn(); }
}

public class LightOffCommand implements Command {
    private Light light;
    public LightOffCommand(Light light) { this.light = light; }
    public void execute() { light.turnOff(); }
}
```

👉 Each of these classes **knows what to do** —
`LightOnCommand` means *“turn on the light.”*
`LightOffCommand` means *“turn off the light.”*

They hold a reference to the **Receiver (`Light`)**, but the **caller** (remote) doesn’t know that.

🎛️ Step 4 — Invoker doesn’t know details

```
public class RemoteControl {
    private Command command;
    public void setCommand(Command command) { this.command = command; }
    public void pressButton() { command.execute(); }
}
```

👉 The **RemoteControl** just says *“Hey command, do your job.”*
It doesn’t know:

- What the command is (LightOn or LightOff)
- Who the receiver is (Light)
- What steps are needed (turnOn or turnOff)

👤 Step 5 — Client wires everything together

```
Light light = new Light();
Command on = new LightOnCommand(light);
Command off = new LightOffCommand(light);
RemoteControl remote = new RemoteControl();
```

👉 The **Client** connects the pieces:

- Makes the `Command`
- Gives the `Receiver` to it
- Gives the `Command` to the `Invoker`

💡 So “How is it happening?”

Concept	Who does it here	Meaning
What to do	`Command` (`LightOnCommand`, `LightOffCommand`)	The action (turn on/off)
Who does it	`Receiver` (`Light`)	The actual worker
Who triggers it	`Invoker` (`RemoteControl`)	The button press
Who connects all	`Client` (`Main`)	Sets up everything

✅ In short:

- The `RemoteControl` (Invoker) never calls `light.turnOn()` directly.
- It just says `command.execute()`.
- The `Command` object decides *what action* to take and calls the `Receiver` (`light.turnOn()`).

Thus, **“what to do” (the action)** and **“who does it” (the light)** are **separated** —
that’s the essence of the **Command Design Pattern** 💡

You said:

so what i think if i have a new device fan it RemoteControl can incoporate very easily or i can also have LinkList of devices in remote controll and any future device it can support?

💯 Exactly, you got it **perfectly right!** 👏👏

That’s the **main power of the Command Design Pattern** —
👉 You can easily plug in new devices (like a Fan, TV, AC, Door Lock, etc.)
👉 The **RemoteControl** doesn’t change at all.

Let’s walk through this slowly and clearly 👇

🧠 You said:

> if i have a new device `Fan`, `RemoteControl` can incorporate it easily

✅ 100% true.
You just create a new `Receiver` (Fan) and its corresponding `Command` classes.

Example:

```
public class Fan {
    public void start() { System.out.println("Fan started"); }
    public void stop()  { System.out.println("Fan stopped"); }
}
```

Now create its commands:

```
public class FanStartCommand implements Command {
    private Fan fan;
    public FanStartCommand(Fan fan) { this.fan = fan; }
    public void execute() { fan.start(); }
}

public class FanStopCommand implements Command {
    private Fan fan;
    public FanStopCommand(Fan fan) { this.fan = fan; }
    public void execute() { fan.stop(); }
}
```

And use it in the same `RemoteControl`:

```
Fan fan = new Fan();
Command fanOn  = new FanStartCommand(fan);
Command fanOff = new FanStopCommand(fan);

RemoteControl remote = new RemoteControl();

remote.setCommand(fanOn);
remote.pressButton();   // Fan started

remote.setCommand(fanOff);
remote.pressButton();   // Fan stopped
```

Did we change RemoteControl?
👉 No! Not even one line.

That’s the beauty of this pattern — **Open for extension, closed for modification** ✅

🧩 You also said:

> I can have a LinkedList of devices in RemoteControl

💡 Brilliant thinking — yes, you can!
Instead of one `Command`, your remote could hold **a list of commands**, e.g. to run all together (a *macro*).

```
public class RemoteControl {
    private List<Command> commands = new LinkedList<>();

    public void addCommand(Command command) {
        commands.add(command);
    }

    public void pressAllButtons() {
        for (Command cmd : commands) {
            cmd.execute();
        }
    }
}
```

Then:

```
Light light = new Light();
Fan fan = new Fan();

Command lightOn = new LightOnCommand(light);
Command fanOn = new FanStartCommand(fan);

RemoteControl remote = new RemoteControl();
remote.addCommand(lightOn);
remote.addCommand(fanOn);

remote.pressAllButtons();
// Output:
// Light ON
// Fan started
```

Now your remote supports **multiple devices at once** 🧠 💡
Still — `RemoteControl` doesn't care what the device is —
only that each one has an `execute()` command.

✅ So the benefits you discovered are:

Advantage	Explanation
🔹 **Add new devices easily**	Just create new Command classes
🔹 **No RemoteControl changes**	It always calls `execute()` blindly
🔹 **Can run multiple commands**	Use a list (MacroCommand)
🔹 **Supports future growth**	Pattern is fully extensible

✅ JDK Internal Example

`java.lang.Runnable` is a command:

```
Runnable command = () -> System.out.println("Hello");
new Thread(command).start(); // Thread is the invoker
```

`javax.swing.Action`

- Each action on a UI component is encapsulated using `Action` interface — a command object.

✅ Spring Boot Example

In Spring Boot, the **Command pattern is often used in task execution**, event handling, and controller-to-service delegation.

🧾 Scenario: Payment Processing (Decoupling Strategies)

```
public interface PaymentCommand {
    void process();
}

@Component
public class UpiPaymentCommand implements PaymentCommand {
    public void process() {
        System.out.println("Processing UPI");
    }
}

@Component
public class CreditCardPaymentCommand implements PaymentCommand {
    public void process() {
        System.out.println("Processing Credit Card");
    }
}
```

Central Command Invoker (e.g., Factory/Registry)

```
java
CopyEdit
@Component
public class PaymentInvoker {
    private final Map<String, PaymentCommand> strategyMap;

    public PaymentInvoker(List<PaymentCommand> commands) {
        strategyMap = new HashMap<>();
        strategyMap.put("upi", commands.stream().filter(c -> c instanceof
UpiPaymentCommand).findFirst().get());
        strategyMap.put("card", commands.stream().filter(c -> c instanceof
CreditCardPaymentCommand).findFirst().get());
    }

    public void execute(String method) {
        strategyMap.get(method).process();
    }
}
```

REST Controller

```
@RestController
public class PaymentController {
    @Autowired
    private PaymentInvoker invoker;

    @PostMapping("/pay/{method}")
    public String pay(@PathVariable String method) {
        invoker.execute(method);
        return "Processed " + method;
    }
}
```

✅ Benefits

- Decouples request sender and receiver
- Easily extendable (new command = new class)
- Supports undo/redo
- Logs/audit trail implementation made simple

✅ Variants & Related Patterns

Pattern	Relation
Strategy	Command returns result; Strategy is used for algorithm switching
Chain of Responsibility	Chain of commands
Memento	Often used with Command to support undo
