Command Pattern

Resources

- 41. All Behavioral Design Patterns | Strategy, Observer, State, Template, Command, Visitor, Memento
- 31. Design Undo, Redo feature with Command Pattern | Command Design Pattern| Low Level System Design

Definition

> ***The Command pattern is a behavioral design pattern that encapsulates a request as an object, allowing you to parameterize and queue them, which in turn helps in decoupling the sender and receiver. By storing the object's state, we can also implement undo/redo operations.***

Let's consider an example of Remote controls for devices, like sending commands from an AC Remote to the actual AC (Air Conditioner) device set up in the room. The remote control doesn't need to know how the AC works internally - it just calls `execute()` upon a button press(command). We can introduce parameterization by having different buttons configured with different commands at runtime. The pattern separates the object that invokes the operation(sender) from the one that performs it(receiver), making the system more flexible and maintainable.

The Problem: Traditional Approach

Let's implement the above example in a naive approach.

```
public class AirConditioner {
    boolean isOn;
    int temperature;

    public void turnOn() {
        isOn = true;
        System.out.println("Air conditioner is on");
    }

    public void turnOff() {
        isOn = false;
        System.out.println("Air conditioner is off");
    }

    public void setTemperature(int temperature) {
        this.temperature = temperature;
        System.out.println("Air conditioner temperature set to " +
temperature);
    }
}
```

```java
public class Bulb {
    boolean isOn;

    public void turnOn() {
        isOn = true;
        System.out.println("Bulb is on");
    }

    public void turnOff() {
        isOn = false;
        System.out.println("Bulb is off");
    }
}
```

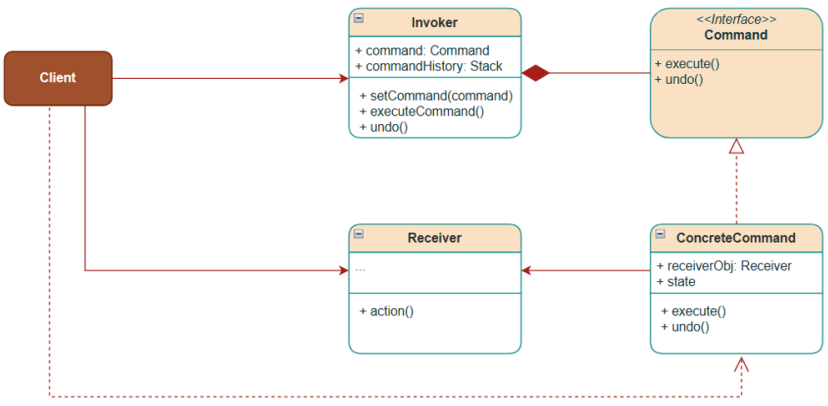
```java
public class Client {
    public static void main(String[] args) {
        System.out.println("Command Pattern: Problem Demo");

        // Device: Air Conditioner Commands
        AirConditioner airConditioner = new AirConditioner();
        airConditioner.turnOn();
        airConditioner.setTemperature(25);
        airConditioner.turnOff();

        // Device: Bulb Commands
        Bulb bulb = new Bulb();
        bulb.turnOn();
        bulb.turnOff();
    }
}
```

The problems with the above implementation are:

1. **Lack of Abstraction:** The remote control is directly dependent on specific device classes. Tomorrow, when we want to scale to a smart controller by adding new devices, we will need to modify the remote control code. This will lead to more redundant code, which is not a good design practice.
2. **Undo/Redo Functionality:** What if we want to add the undo/redo capability? How it will be handled. If we provide the implementation in client code (without command objects storing previous state), implementing undo becomes clumsy, requiring the invoker to track state for all possible operations.
3. **Difficulty in Code Maintenance:** What if in the future, we have to support more commands for more devices example Bulb. Supporting multiple device types leads to complex, monolithic, bloated remote control classes, leading to violation of SOLID Principles and difficulty in Testing.

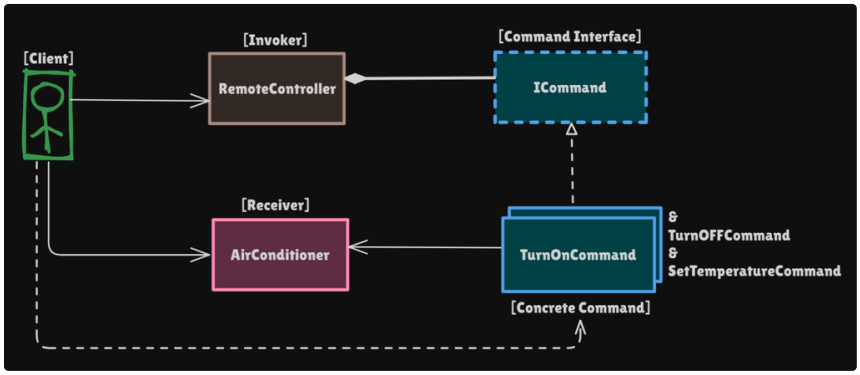
Without this pattern, you end up with tightly coupled, inflexible code that's hard to maintain, test, and extend.

Class Diagram



Structure of the Command Pattern

Understanding the structure of the command design pattern using the AC Remote Control example.



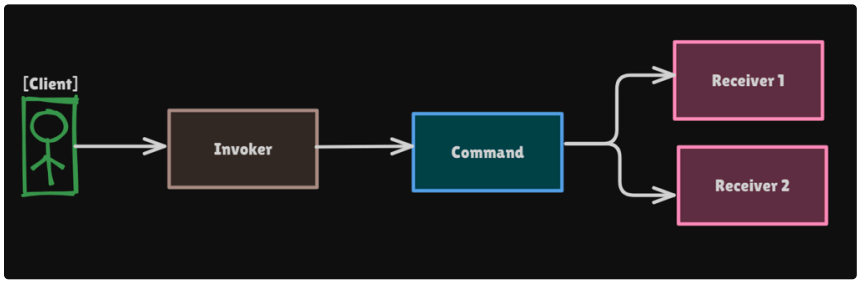
- **Receiver (** `AirConditioner` **)**: The object that performs the actual work. It contains the business logic that will be triggered by a command.
- **Command Interface**: Declares `execute()` and `undo()` methods and introduces a layer of abstraction.
- **Concrete Commands(** `TurnOnCommand` , `TurnOffCommand` , `SetTemperatureCommand` **)**: Implement specific operations like turning on/off, changing temperature. Each receiver operation has a dedicated class.
- **Invoker (** `RemoteController` **)**: Stores a reference to the Command object and executes commands upon receiving a request for an operation. Holds a data structure to track previous states that help implement undo/redo functionality.
- **Client**: Creates and configures commands and associates them with the right receiver, and configures the Invoker to execute the commands.

Implementation (Example: AC Remote with Undo Functionality)

How does the Command design pattern solve the above issue?

It separates the logic of:

- Receiver
- Invoker and
- Command




```
// Receiver - The AC Device that performs actual operations
public class AirConditioner {
    boolean isOn;
    int temperature;

    public void turnOn() {
        isOn = true;
```

```
        System.out.println("Air conditioner is on");
    }

    public void turnOff() {
        isOn = false;
        System.out.println("Air conditioner is off");
    }

    public boolean isOn() {
        return isOn;
    }

    public void setOn(boolean on) {
        isOn = on;
    }

    public int getTemperature() {
        return temperature;
    }

    public void setTemperature(int temperature) {
        this.temperature = temperature;
        System.out.println("Air conditioner temperature set to " +
temperature + "°C");
    }

}
```

```
// Command interface
public interface ICommand {
    void execute();

    void undo();
}
```

```
// Concrete Commands
public class TurnOnCommand implements ICommand {
    private final AirConditioner ac;
    private boolean previousState;

    public TurnOnCommand(AirConditioner ac) {
        this.ac = ac;
    }

    @Override
    public void execute() {
        previousState = ac.isOn();
        ac.turnOn();
    }

    @Override
    public void undo() {
        System.out.print("Undo: Turn On command. ");
        if (!previousState) {
            ac.turnOff();
        }
    }
}
```

```
// Concrete Command
public class TurnOffCommand implements ICommand {
    private final AirConditioner ac;
    private boolean previousState;

    public TurnOffCommand(AirConditioner ac) {
        this.ac = ac;
    }

    @Override
    public void execute() {
```

```
        previousState = ac.isOn();
        ac.turnOff();
    }

    @Override
    public void undo() {
        System.out.print("Undo: Turn Off command. ");
        if (previousState) {
            ac.turnOn();
        }
    }
}
```

```
// Concrete Command
public class SetTemperatureCommand implements ICommand {
    private final AirConditioner ac;
    private final int newTemperature;
    private int previousTemperature;

    public SetTemperatureCommand(AirConditioner ac, int temperature) {
        this.ac = ac;
        this.newTemperature = temperature;
    }

    @Override
    public void execute() {
        previousTemperature = ac.getTemperature();
        ac.setTemperature(newTemperature);
    }

    @Override
    public void undo() {
        System.out.print("Undo: Set Temperature Command. ");
        ac.setTemperature(previousTemperature);
    }
}
```

```
// Invoker - with undo functionality
public class RemoteController {
    ICommand command; // Holds reference to the command
    Stack<ICommand> commandHistory = new Stack<>();

    RemoteController() {
    }

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

    public void pressButton() {
        command.execute();
        commandHistory.push(command);
    }

    public void undo() {
        if (!commandHistory.isEmpty()) {
            ICommand lastCommand = commandHistory.pop();
            lastCommand.undo();
        }
    }
}
```

```
// Client - Demonstration
public class Client {
    public static void main(String[] args) {
        System.out.println("##### Command Pattern: Solution Demo
#####");

        // Create Receiver
        AirConditioner airConditioner = new AirConditioner();
```

```
8
9           // Create Invoker
10          RemoteController remoteObj = new RemoteController();
11
12          // Execute Commands
13          remoteObj.setCommand(new TurnOnCommand(airConditioner));
14          remoteObj.pressButton();
15          remoteObj.setCommand(new SetTemperatureCommand(airConditioner,
    25));
16          remoteObj.pressButton();
17          remoteObj.setCommand(new SetTemperatureCommand(airConditioner,
    18));
18          remoteObj.pressButton();
19          remoteObj.setCommand(new TurnOffCommand(airConditioner));
20          remoteObj.pressButton();
21
22          // Undo Command
23          remoteObj.undo(); // Undo: Turn Off command => AC is now on
24
25          // Undo Command
26          remoteObj.undo(); // Undo: Set Temperature Command. AC
    temperature is now 25°C
27
28          // Undo Command
29          remoteObj.undo(); // Undo: Set Temperature Command. AC
    temperature is now 0°C
30
31          // Undo Command
32          remoteObj.undo(); // Undo: Turn On command => AC is now off
33      }
34  }
```

Output

```
##### Command Pattern: Solution Demo #####
Air conditioner is on
Air conditioner temperature set to 25°C
Air conditioner temperature set to 18°C
Air conditioner is off
Undo: Turn Off command. Air conditioner is on
Undo: Set Temperature Command. Air conditioner temperature set to 25°C
Undo: Set Temperature Command. Air conditioner temperature set to 0°C
Undo: Turn On command. Air conditioner is off

Process finished with exit code 0
```