

**Problem statement:**

1) Suppose we are building a text editor which has buttons like bold, italic and underline for text formatting.

2) So now suppose I have a GUI framework in which which gives me generic button classes.so I have a bold button italic button and a underline button.

3) if I do not follow the command pattern right. So what may happen is that okay I have a class called as bold button.And in this class or in the onClick method,when the user clicks the button What I have to do is I have to hard code that.I have to say okay, make this text as bold.

But it is also possible that this button is also supplied to other applications where it might have a different purpose.

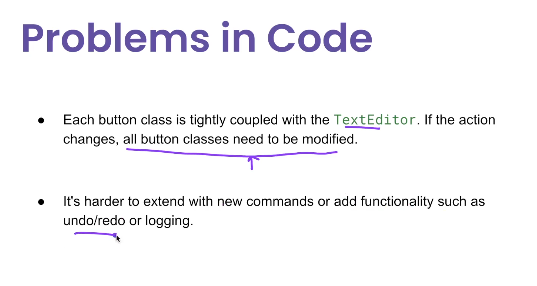
4) I don't want to hard code this functionality inside the onClick method.

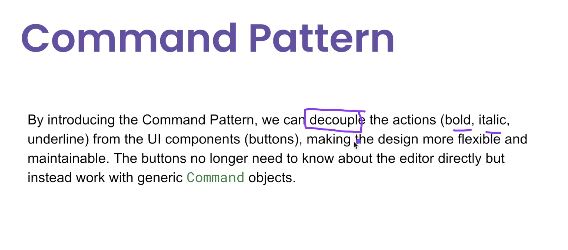
5) but if I hardcode it, it will make it, um, tightly coupled with the text editor. won't be able to use this button with maybe some other application like Photoshop.

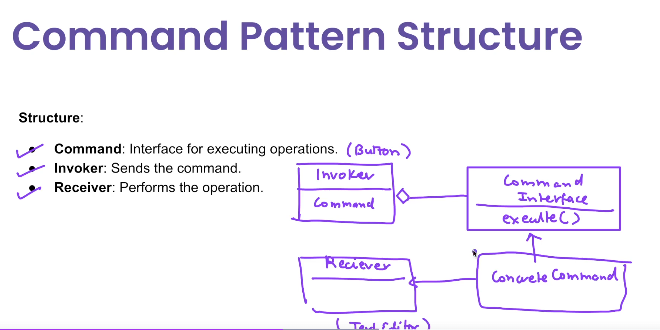
6) So generally the buttons, the other components, they are part of a separate framework.

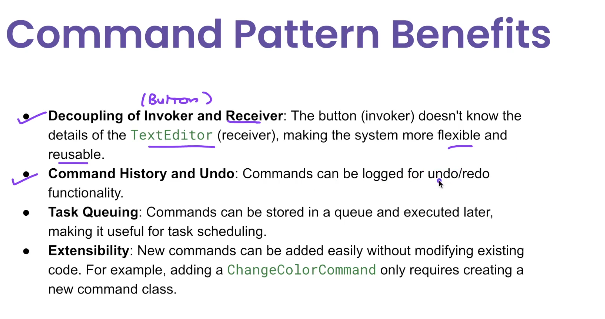
7) Let's say there is a GUI framework which some other developer has written, and text editor is your app that you are building

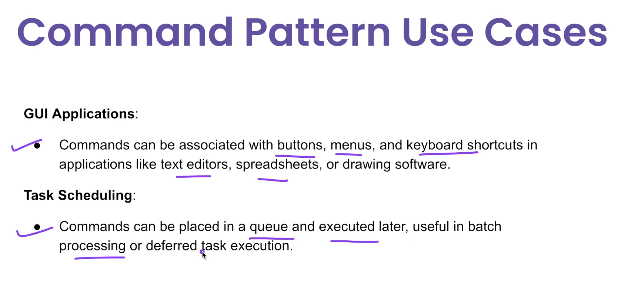
8) So what we want to achieve is we want to achieve loose coupling between the button class and the text editor class

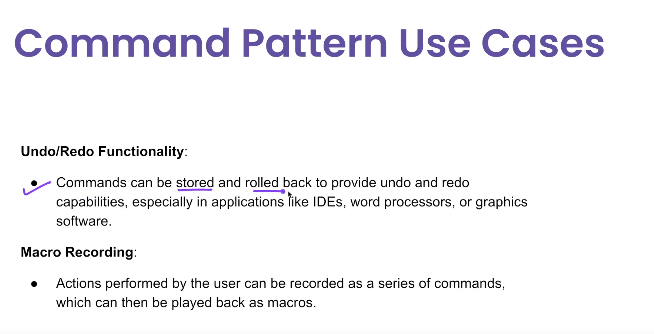












**🧩 What is the Command Pattern?**

The **Command Pattern** is a behavioral design pattern that turns a **request into a standalone object** containing all the information about the request, allowing you to **parameterize, queue, log, undo**, or **delay execution**.

It encapsulates the request as an object (called Command) and decouples the **sender** from the **receiver**.

**✅ Real-World Use Cases of Command Pattern**

| **Use Case** | **Example** |
| --- | --- |
| ✅ GUI actions | Buttons triggering different actions (undo, redo) |
| ✅ Task queues | Asynchronous processing of jobs |
| ✅ Macro recording | Record user actions and replay |
| ✅ Undo/redo functionality | Maintain history of commands to rollback |
| ✅ Remote control / automation | IoT or TV remote – each button triggers a command |
| ✅ Workflow engines | Chain of business tasks executed sequentially |
| ✅ Logging/auditing requests | Store command metadata for future audit |

**🧩 Structure of the Command Pattern**

**👇 Key Participants:**

| **Component** | **Responsibility** |
| --- | --- |
| Command | Declares interface for all commands |
| ConcreteCommand | Implements the command logic |
| Receiver | Actual business logic that executes action |
| Invoker | Triggers the command |
| Client | Configures command and sets receiver |

**✅ Advantages of Command Pattern**

| **Advantage** | **Description** |
| --- | --- |
| ✅ **Decouples sender and receiver** | The invoker doesn’t know the actual business logic |
| ✅ **Easy to extend** | Add new commands without changing existing code |
| ✅ **Supports undo/redo** | Store history of command objects and reverse them |
| ✅ **Supports logging/auditing** | Commands can be persisted or logged |
| ✅ **Macro or batch support** | Execute a series of commands sequentially |

**❌ Disadvantages of Command Pattern**

| **Disadvantage** | **Description** |
| --- | --- |
| ❌ **Increased number of classes** | Each command is a new class, leading to boilerplate |
| ❌ **Overkill for simple logic** | May be too heavy for basic button-click logic |
| ❌ **Difficult to debug in complex chains** | Tracing which command was executed can be hard |
| ❌ **No built-in undo** | You need to implement undo logic yourself if needed |

**✅ When to Use Command Pattern**

| **Use If...** | **Example** |
| --- | --- |
| ✅ You want to queue/delay/undo/redo operations | Job queues, editors |
| ✅ You want to log/audit every action | Financial transactions |
| ✅ You want to decouple UI from logic | Remote controls, GUI apps |
| ✅ You need macro commands | Scriptable systems |

**❌ When NOT to Use**

| **Avoid If...** | **Reason** |
| --- | --- |
| ❌ You only need one simple action | Adds unnecessary complexity |
| ❌ No undo/logging needed | You can directly call receiver methods |
| ❌ Strategy pattern fits better | For choosing between algorithms (e.g., validation) |

**🔁 Command Pattern vs Strategy Pattern**

| **Feature** | **Command Pattern** | **Strategy Pattern** |
| --- | --- | --- |
| Purpose | Encapsulate a request | Encapsulate an algorithm |
| Triggered By | External event or invoker | Client logic or condition |
| Use Case | Undo/Redo, queueing, history | Input validation, sorting, behavior switching |
| Allows Queuing | ✅ Yes | ❌ No |
| Allows Undo | ✅ Yes | ❌ No |
| Runtime Switching | ✅ Yes | ✅ Yes |
| Focus | Action to perform | How to perform |

**✅ Summary Table**

| **Feature** | **Command Pattern** |
| --- | --- |
| Type | Behavioral Pattern |
| Based On | Encapsulation of requests as objects |
| Key Advantage | Decouples caller and logic, supports undo |
| Ideal For | GUI actions, jobs, undo/redo, macro commands |
| Not Ideal For | Simple direct method calls |

**✅ Java Example: Remote Control**

**1. Command Interface**

java

CopyEdit

public interface Command {

void execute();

}

**2. Receiver (Actual Business Logic)**

java

CopyEdit

public class Light {

public void turnOn() {

System.out.println("Light is ON");

}

public void turnOff() {

System.out.println("Light is OFF");

}

}

**3. Concrete Command Classes**

java

CopyEdit

public class LightOnCommand implements Command {

private Light light;

public LightOnCommand(Light light) {

this.light = light;

}

@Override

public void execute() {

light.turnOn();

}

}

public class LightOffCommand implements Command {

private Light light;

public LightOffCommand(Light light) {

this.light = light;

}

@Override

public void execute() {

light.turnOff();

}

}

**4. Invoker (e.g., Remote)**

java

CopyEdit

public class RemoteControl {

private Command command;

public void setCommand(Command command) {

this.command = command;

}

public void pressButton() {

command.execute();

}

}

**5. Client Code**

java

CopyEdit

public class CommandPatternDemo {

public static void main(String[] args) {

Light livingRoomLight = new Light();

Command lightOn = new LightOnCommand(livingRoomLight);

Command lightOff = new LightOffCommand(livingRoomLight);

RemoteControl remote = new RemoteControl();

remote.setCommand(lightOn);

remote.pressButton(); // Output: Light is ON

remote.setCommand(lightOff);

remote.pressButton(); // Output: Light is OFF

}

}