Scenario 6: Customer Request - Command Pattern

Customer Request:

"I need a document editor that can perform various operations like opening, saving, printing, and emailing documents. However, I want to be able to queue these operations, undo them if needed, and possibly redo them later. It would be great if these operations could be encapsulated as individual objects."

Choosing the Right Pattern:

Pattern Recommendation: Command Pattern

Why?

The Command Pattern is ideal for this scenario because:

- Encapsulation of Operations: The customer needs to encapsulate requests (like open, save, print, email) as objects, allowing them to be parameterized, queued, logged, and undone/redone.

- Undo/Redo Functionality: The Command Pattern provides built-in support for undo/redo operations, which are a common requirement in applications like document editors.

- Decoupling the Invoker from the Receiver: The pattern decouples the object that invokes the operation (e.g., a button click) from the object that knows how to perform the operation, making the system more modular and flexible.

Pattern Explanation: Command Pattern

Key Concepts:

- Command: An interface that declares an execute() method, which encapsulates an action (e.g., open, save, print).
- Concrete Command: Implements the Command interface, invoking the corresponding operations on the receiver.
- Receiver: The object that performs the actual work when the command is executed (e.g., the document).
- Invoker: The object that holds and executes the command (e.g., a button in the UI).
- Client: The part of the code that creates concrete command objects and sets up the relationships between the invoker and the receiver.

How It Works:

- Command Interface: Defines a method like execute() that encapsulates the action to be performed.

- Concrete Command: Implements the Command interface and invokes the necessary method(s) on the receiver (e.g., Document class).

- Receiver: The object that actually performs the action (e.g., opening or saving a document).

- Invoker: Calls the execute() method on the command. This could be a button in a user interface that triggers the command.

- Client: The client sets up the invoker, command, and receiver. It creates the concrete commands and associates them with the appropriate receivers.

Implementation Example:

```
// Command Interface
interface Command {
    void execute();
}

// Concrete Commands
class OpenCommand implements Command {
    private Document document;

    public OpenCommand(Document document) {
        this.document = document;
    }

    @Override
    public void execute() {
        document.open();
    }
}

class SaveCommand implements Command {
    private Document document;

    public SaveCommand(Document document) {
        this.document = document;
    }

    @Override
    public void execute() {
        document.save();
    }
}

class PrintCommand implements Command {
    private Document document;

    public PrintCommand(Document document) {
        this.document = document;
    }

    @Override
    public void execute() {
        document.print();
    }
}
```

```
class EmailCommand implements Command {
    private Document document;

    public EmailCommand(Document document) {
        this.document = document;
    }

    @Override
    public void execute() {
        document.email();
    }
}

// Receiver Class
class Document {
    private String name;

    public Document(String name) {
        this.name = name;
    }

    public void open() {
        System.out.println("Opening document: " + name);
    }

    public void save() {
        System.out.println("Saving document: " + name);
    }

    public void print() {
        System.out.println("Printing document: " + name);
    }

    public void email() {
        System.out.println("Emailing document: " + name);
    }
}

// Invoker Class
class DocumentEditor {
    private Command openCommand;
    private Command saveCommand;
    private Command printCommand;
    private Command emailCommand;

    public void setOpenCommand(Command openCommand) {
        this.openCommand = openCommand;
```

```
}

public void setSaveCommand(Command saveCommand) {
    this.saveCommand = saveCommand;
}

public void setPrintCommand(Command printCommand) {
    this.printCommand = printCommand;
}

public void setEmailCommand(Command emailCommand) {
    this.emailCommand = emailCommand;
}

public void clickOpen() {
    openCommand.execute();
}

public void clickSave() {
    saveCommand.execute();
}

public void clickPrint() {
    printCommand.execute();
    }

    public void clickEmail() {
        emailCommand.execute();
    }
}

// Client Code
public class CommandPatternDemo {
    public static void main(String[] args) {
        Document document = new Document("DesignPatterns.docx");

        Command open = new OpenCommand(document);
        Command save = new SaveCommand(document);
        Command print = new PrintCommand(document);
        Command email = new EmailCommand(document);

        DocumentEditor editor = new DocumentEditor();
        editor.setOpenCommand(open);
        editor.setSaveCommand(save);
        editor.setPrintCommand(print);
        editor.setEmailCommand(email);

        // Simulate button clicks
        editor.clickOpen();

        editor.clickSave();

        editor.clickPrint();

        editor.clickEmail();

    }

}
```

Key Points to Remember:

- Command Pattern encapsulates a request as an object, allowing for parameterization of clients with queues, requests, and operations.
- It supports operations like Undo and Redo by keeping a history of commands executed.
- It decouples the object that invokes the operation from the one that knows how to perform it, promoting flexibility and reuse.

When to Use the Command Pattern:

- When you need to parameterize objects by an action to perform.
- When you need to specify, queue, and execute requests at different times.
- When you need to support undo/redo operations in your application.

Advantages:

- Decouples Invoker and Receiver: The invoker doesn?t need to know anything about the concrete command it executes or the receiver that performs the action.
- Supports Undo/Redo: The pattern inherently supports undoable operations by storing the history of commands.

- Flexible and Reusable: Commands can be reused and extended easily, allowing for greater flexibility in the system.

Disadvantages:

- Increased Number of Classes: The pattern can lead to an increase in the number of classes, as you need a separate class for each command.