11. Strategy Pattern

Scenario Recap:

You have a document editor that needs to handle different types of document elements such as text, images, and tables. Users should be able to apply various strategies like spell-checking, grammar-checking, or rendering to these elements. The Strategy Pattern should allow these strategies to be applied uniformly across different types of elements.

Revised Implementation

Here?s how we can modify the implementation to handle different types of document elements while applying various strategies.

```java
// Strategy Interface
interface DocumentProcessingStrategy {
    void execute(DocumentElement element);
}

// Concrete Strategies
class SpellCheckStrategy implements DocumentProcessingStrategy {
    @Override
    public void execute(DocumentElement element) {
        System.out.println("Checking spelling for: " + element.getContent());
    }
}
```

```java
class GrammarCheckStrategy implements DocumentProcessingStrategy {
    @Override
    public void execute(DocumentElement element) {
        System.out.println("Checking grammar for: " + element.getContent());
    }
}

class RenderStrategy implements DocumentProcessingStrategy {
    @Override
    public void execute(DocumentElement element) {
        System.out.println("Rendering element: " + element.getContent());
    }
}

// Abstract Document Element
abstract class DocumentElement {
    protected String content;

    public DocumentElement(String content) {
        this.content = content;
    }

    public String getContent() {
        return content;
    }

    public abstract void process(DocumentProcessingStrategy strategy);
}

// Concrete Document Elements
class TextElement extends DocumentElement {
    public TextElement(String content) {
        super(content);
    }

    @Override
    public void process(DocumentProcessingStrategy strategy) {
        System.out.println("Processing text element...");
        strategy.execute(this);
    }
}

class ImageElement extends DocumentElement {
    public ImageElement(String content) {
        super(content);
    }

    @Override
    public void process(DocumentProcessingStrategy strategy) {
        System.out.println("Processing image element...");
        strategy.execute(this);
    }
}
```

```
class TableElement extends DocumentElement {
    public TableElement(String content) {
        super(content);
    }

    @Override
    public void process(DocumentProcessingStrategy strategy) {
        System.out.println("Processing table element...");
        strategy.execute(this);
    }
}

// Client Code
public class DocumentEditor {
    public static void main(String[] args) {
        DocumentElement text = new TextElement("This is a text paragraph.");
        DocumentElement image = new ImageElement("/images/sample.png");
        DocumentElement table = new TableElement("Table content");

        // Apply spell checking to text
        text.process(new SpellCheckStrategy());

        // Apply rendering to image
        image.process(new RenderStrategy());

        // Apply rendering to table
        table.process(new RenderStrategy());
    }
}
```

Explanation of the Revised Implementation:

DocumentProcessingStrategy Interface: Defines the execute() method that will be implemented by concrete strategies like SpellCheckStrategy, GrammarCheckStrategy, and RenderStrategy.

DocumentElement Abstract Class: This abstract class is extended by different types of document elements (e.g., TextElement, ImageElement, TableElement). Each type of element implements the process() method, which takes a DocumentProcessingStrategy and applies it to the element.

Concrete Document Elements: Each document element type (e.g., TextElement, ImageElement, TableElement) has its own class that inherits from DocumentElement. These classes use the process() method to apply different strategies.

Client Code: The client code creates different document elements and applies various strategies to them, demonstrating how the Strategy Pattern can be used to handle different types of elements uniformly.

How This Solves the Problem:

Uniform Strategy Application: The Strategy Pattern allows you to apply the same strategy to different types of document elements (text, images, tables) uniformly.

Flexibility: You can easily swap out strategies (e.g., switching from spell-checking to rendering) without modifying the document elements themselves.

Extensibility: Adding new document element types or new strategies is straightforward and doesn?t require changes to existing code.