Scenario 12: Customer Request - Visitor Pattern

Customer Request:

"I need a document editor that allows users to handle different types of document elements like text, images, and tables.

Users should be able to apply different operations, such as spell-checking, formatting, and exporting. However, I want to

be able to add new operations in the future without changing the document element classes. The operations should be easily

extendable and applicable to various types of document elements."

Choosing the Right Pattern:

Pattern Recommendation: Visitor Pattern

Why?

The Visitor Pattern is ideal for this scenario because:

1. Extensibility: The customer wants to be able to add new operations (e.g., spell-checking, formatting, exporting) without

modifying the document element classes. The Visitor Pattern allows you to define new operations separately from the elements,

making it easy to extend the system with new functionalities.

2. Separation of Concerns: The pattern separates the algorithms (operations) from the objects on which they operate (document

elements), promoting cleaner and more maintainable code.

3. Uniform Operations on Different Elements: The Visitor Pattern makes it easy to apply the same operation across a collection
of different element types (e.g., text, images, tables) in a consistent way.

Pattern Explanation: Visitor Pattern

Key Concepts:

1. Visitor: An interface that declares a visit method for each type of element in the object structure.
2. Concrete Visitor: Implements the Visitor interface, defining the operations to be performed on each type of element.
3. Element: An interface that declares an accept method, which takes a visitor as an argument.
4. Concrete Element: Implements the Element interface and defines the accept method, which calls the appropriate visit method
on the visitor.

How It Works:

1. Visitor Interface: Defines visit methods for each type of element (e.g., visitTextElement(TextElement element),
visitImageElement(ImageElement element)).

2. Concrete Visitor: Implements the Visitor interface and provides specific operations (e.g., spell-checking, formatting)
for each type of element.

3. Element Interface: Declares an accept method that takes a visitor and calls the appropriate visit method on it.

4. Concrete Element: Implements the Element interface and delegates the operation to the visitor via the accept method.

5. Client: The client creates visitors and applies them to elements by calling the accept method.

Implementation Example

Here's how the implementation might look in Java:

```java
// Visitor Interface
interface DocumentVisitor {
    void visitTextElement(TextElement text);
    void visitImageElement(ImageElement image);
    void visitTableElement(TableElement table);
}

// Concrete Visitor for Spell Checking
class SpellCheckVisitor implements DocumentVisitor {
    @Override
    public void visitTextElement(TextElement text) {
        System.out.println("Checking spelling for text: " + text.getContent());
    }

    @Override
    public void visitImageElement(ImageElement image) {
        System.out.println("No spell-check needed for image.");
    }
    @Override
    public void visitTableElement(TableElement table) {
        System.out.println("Checking spelling for table content: " + table.getContent());
    }
}

// Concrete Visitor for Formatting
class FormattingVisitor implements DocumentVisitor {
    @Override
    public void visitTextElement(TextElement text) {
        System.out.println("Formatting text: " + text.getContent());
    }

    @Override
    public void visitImageElement(ImageElement image) {
        System.out.println("Formatting image: " + image.getContent());
    }

    @Override
    public void visitTableElement(TableElement table) {
        System.out.println("Formatting table: " + table.getContent());
    }
}

// Element Interface
interface DocumentElement {
    void accept(DocumentVisitor visitor);
    String getContent();
}


// Concrete Elements
class TextElement implements DocumentElement {
    private String content;


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


    @Override
    public void accept(DocumentVisitor visitor) {
        visitor.visitTextElement(this);
    }


    @Override
    public String getContent() {
        return content;
    }
}


class ImageElement implements DocumentElement {
    private String content;


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

    @Override
    public void accept(DocumentVisitor visitor) {
        visitor.visitImageElement(this);
    }

    @Override
    public String getContent() {
        return content;
    }
}

class TableElement implements DocumentElement {
    private String content;

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

    @Override
    public void accept(DocumentVisitor visitor) {
        visitor.visitTableElement(this);
    }

    @Override
    public String getContent() {
        return content;
    }
}

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

        // Create visitors
        DocumentVisitor spellCheckVisitor = new SpellCheckVisitor();
        DocumentVisitor formattingVisitor = new FormattingVisitor();

        // Apply visitors to elements
        text.accept(spellCheckVisitor);
        image.accept(spellCheckVisitor);
        table.accept(spellCheckVisitor);

        text.accept(formattingVisitor);
        image.accept(formattingVisitor);
        table.accept(formattingVisitor);
    }
}
```

Key Points to Remember:

1. Visitor Pattern allows you to define new operations on an object structure without changing the classes of the elements on
which it operates.

2. It is particularly useful when you need to perform different types of operations on a set of elements and want to keep these
operations separate from the elements themselves.

When to Use the Visitor Pattern:

1. When you need to perform operations on objects of a complex object structure and want to keep these operations separate from
the object classes.

2. When the object structure rarely changes, but you often need to define new operations.

Advantages:

1. Easy to Add New Operations: New operations can be added without modifying the element classes.

2. Single Responsibility Principle: Operations are kept separate from the object classes, promoting cleaner design.

3. Extensibility: The pattern makes it easy to extend the system with new operations.

Disadvantages:

1. Adding New Elements Can Be Difficult: If the object structure changes frequently, adding new element types requires
changes to the visitor interface and all its concrete implementations.

2. Complexity: The pattern can introduce additional complexity by adding a layer of visitors and visit methods.