Scenario 7: Customer Request

Customer Request:

"I need a document editor that allows users to work with various document elements like text, images, tables, and other objects. Users should be able to group these elements together, apply operations on groups as well as individual elements, and ensure that the entire structure is treated uniformly. For example, users should be able to format a single paragraph or an entire section containing multiple paragraphs, images, and tables."

Choosing the Right Pattern:

Pattern Recommendation: Composite Pattern

Why?

The Composite Pattern is ideal for this scenario because:

- Uniformity: The customer needs to treat individual objects and compositions of objects uniformly. The Composite Pattern allows you to treat individual elements (like text, images) and groups of elements (like sections or entire documents) the same way.
- Hierarchical Structures: The customer wants to manage a hierarchical structure of elements (e.g., a document containing sections, sections containing paragraphs and images). The Composite Pattern makes it easy to work with these structures recursively.
- Operations on Groups: The pattern allows you to perform operations on both individual elements and entire groups, applying formatting, transformations, or other operations consistently.

Pattern Explanation: Composite Pattern

Key Concepts:

- Component: An interface that defines operations that can be applied to both simple and complex objects.
- Leaf: Represents individual objects in the composition (e.g., a text element, an image).
- Composite: Represents a group of objects (e.g., a section containing multiple paragraphs, images). It can contain both Leafs and other Composites.
- Client: Works with the objects through the Component interface, treating individual objects and compositions uniformly.

How It Works:

- Component Interface: Defines methods like add(), remove(), and display() that will be implemented by both Leaf and Composite classes.
- Leaf: Implements the Component interface and represents individual objects (e.g., Text, Image).
- Composite: Implements the Component interface and contains a collection of child components, allowing operations to be applied recursively.
- Client: Interacts with the Component interface, treating individual elements and groups of elements the same way.

Implementation Example:

Here?s how the implementation might look in Java:

```
// Command Interface
interface DocumentElement {
    void display();
}
```

```
// Leaf Class
class TextElement implements DocumentElement {
    private String text;

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

    @Override
    public void display() {
        System.out.println("Text: " + text);
    }
}

class ImageElement implements DocumentElement {
    private String imagePath;

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

    @Override
    public void display() {
        System.out.println("Image: " + imagePath);
    }
}
```

```
// Composite Class
class CompositeElement implements DocumentElement {
    private List<DocumentElement> elements = new ArrayList<>();

    public void add(DocumentElement element) {
        elements.add(element);
    }

    public void remove(DocumentElement element) {
        elements.remove(element);
    }

    @Override
    public void display() {
        for (DocumentElement element : elements) {
            element.display();
        }
    }
}

// Client Code
public class DocumentEditor {
    public static void main(String[] args) {
        // Create individual elements
        DocumentElement paragraph1 = new TextElement("This is the first paragraph.");
```

```
        DocumentElement paragraph2 = new TextElement("This is the second paragraph.");
        DocumentElement image = new ImageElement("/images/sample.png");

        // Create a composite element (e.g., a section)
        CompositeElement section = new CompositeElement();
        section.add(paragraph1);
        section.add(paragraph2);
        section.add(image);

        // Display individual elements
        System.out.println("Displaying individual elements:");
        paragraph1.display();
        image.display();

        // Display the entire section
        System.out.println("\nDisplaying composite element (section):");
        section.display();
    }
}
```

Key Points to Remember:

- Composite Pattern allows you to compose objects into tree structures to represent part-whole hierarchies, and it lets clients treat individual objects and compositions of objects uniformly.
- The pattern is particularly useful in scenarios where you want to apply the same operation to both individual objects and groups of objects.

When to Use the Composite Pattern:

- When you need to represent part-whole hierarchies of objects.
- When you want clients to be able to ignore the difference between compositions of objects and individual objects.

Advantages:

- Uniformity: Treats individual objects and compositions uniformly.
- Simplifies Client Code: The client can work with complex tree structures through a simple interface.
- Extensibility: New types of components can be added easily.

Disadvantages:

- Complexity: Can make the design more complex when the tree structure becomes deep and elaborate.
- Difficult to Restrict Types: It might be difficult to restrict the types of components that can be added to a composite.