Scenario 14: Customer Request

Customer Request:

I need a document editor that allows users to format text in various ways, such as bold, italic, or underline. However, I want to ensure that the formatting options can be applied, removed, or modified independently without affecting the other formatting options. Additionally, the editor should allow combining multiple formatting options.

Choosing the Right Pattern:

Pattern Recommendation: Decorator Pattern

Why?

The Decorator Pattern is ideal for this scenario because:

- Flexible Addition of Functionality: The customer wants to apply multiple formatting options independently and combine them in any order. The Decorator Pattern allows you to 'decorate' text elements with various formatting options without modifying the original text element class.
- Combining Multiple Behaviors: The pattern supports combining multiple decorators to add various behaviors (like bold, italic, underline) to a single text element.
- Open-Closed Principle: The Decorator Pattern adheres to the Open-Closed Principle, meaning you can extend the behavior of objects by adding new decorators without modifying the original object code.

Pattern Explanation: Decorator Pattern

Key Concepts:

- Component: The core interface or abstract class that defines the basic operations (e.g., display method for text elements).

- Concrete Component: The main object that you want to add additional features to (e.g., TextElement).
- Decorator: An abstract class that implements the component interface and contains a reference to a component. Decorators add extra behavior before or after delegating to the component.
- Concrete Decorators: These are specific implementations of decorators that add features like bold, italic, and underline to the text.

Implementation Example

```
// Component Interface
interface TextElement {
    String display();
}

// Concrete Component
class PlainText implements TextElement {
    private String text;

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

    @Override
    public String display() {
        return text;
    }
}
```

```
// Decorator
abstract class TextDecorator implements TextElement {
    protected TextElement textElement;

    public TextDecorator(TextElement textElement) {
        this.textElement = textElement;
    }

    @Override
    public String display() {
        return textElement.display();
    }
}

// Concrete Decorators
class BoldDecorator extends TextDecorator {
    public BoldDecorator(TextElement textElement) {
        super(textElement);
    }

    @Override
    public String display() {
        return "<b>" + super.display() + "</b>";
    }
}
```

```
class ItalicDecorator extends TextDecorator {
    public ItalicDecorator(TextElement textElement) {
        super(textElement);
    }

    @Override
    public String display() {
        return "<i>" + super.display() + "</i>";
    }
}

class UnderlineDecorator extends TextDecorator {
    public UnderlineDecorator(TextElement textElement) {
        super(textElement);
    }

    @Override
    public String display() {
        return "<u>" + super.display() + "</u>";
    }
}

// Client Code
public class DocumentEditor {
    public static void main(String[] args) {
```

```
        // Plain text
        TextElement text = new PlainText("Hello, World!");

        // Apply bold formatting
        text = new BoldDecorator(text);

        // Apply italic formatting
        text = new ItalicDecorator(text);

        // Apply underline formatting
        text = new UnderlineDecorator(text);

        // Display formatted text
        System.out.println(text.display()); // Output: <u><i><b>Hello, World!</b></i></u>
    }
}
```