Decorator Pattern with Document Editor Example

Scenario 3: Customer Request

Customer Request:

"I want a document editor that not only supports different document formats like Word, PDF, and Text but also allows me to add additional features like spell checking, grammar checking, and plagiarism detection. These features should be optional, and I should be able to combine them in any way I want (e.g., spell checking + grammar checking, or grammar checking + plagiarism detection)."

Choosing the Right Pattern:

Pattern Recommendation: Decorator Pattern

Why?

The Decorator Pattern is perfect for this scenario because:

- Dynamic Feature Addition: The customer wants to add features like spell checking, grammar checking, and plagiarism detection dynamically and in any combination. The Decorator Pattern allows you to add responsibilities to individual objects dynamically without affecting others.

- Flexible Combinations: The customer wants the flexibility to combine features in various ways (e.g., spell checking + grammar checking). The Decorator Pattern lets you "decorate" objects with these additional features, allowing you to stack multiple decorators on top of each other.

- Avoids Subclass Explosion: Without the Decorator Pattern, you'd have to create a new subclass for every possible combination of features, leading to a combinatorial explosion of subclasses. The

Decorator Pattern allows you to avoid this by using composition rather than inheritance.

Pattern Explanation: Decorator Pattern

Key Concepts:

- Component: The core interface or abstract class that defines the basic operations (e.g., Document operations like open(), save(), and close()).
- Concrete Component: The main object that you want to add additional features to (e.g., WordDocument, PDFDocument).
- 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 spell checking, grammar checking, etc.

How It Works:

- Component Interface: Defines the operations that can be dynamically extended (e.g., Document with open(), save(), close()).

- Concrete Component: A specific implementation of the component (e.g., WordDocument, PDFDocument).

- Decorator: An abstract class that holds a reference to a component and implements the component interface.

- Concrete Decorators: Implement the decorator class and add additional behavior (e.g., SpellCheckDecorator, GrammarCheckDecorator).

Implementation Example:

Here?s how the implementation might look in Java:

```
// Component Interface
interface Document {
    void open();
    void save();
    void close();
}

// Concrete Components
class WordDocument implements Document {
    @Override
    public void open() {
        System.out.println("Opening Word document.");
    }
    @Override
    public void save() {
        System.out.println("Saving Word document.");
    }
    @Override
    public void close() {
        System.out.println("Closing Word document.");
    }
}

class PDFDocument implements Document {
    @Override
```

```
    public void open() {
        System.out.println("Opening PDF document.");
    }
    @Override
    public void save() {
        System.out.println("Saving PDF document.");
    }
    @Override
    public void close() {
        System.out.println("Closing PDF document.");
    }
}

// Decorator
abstract class DocumentDecorator implements Document {
    protected Document decoratedDocument;

    public DocumentDecorator(Document decoratedDocument) {
        this.decoratedDocument = decoratedDocument;
    }

    @Override
    public void open() {
        decoratedDocument.open();
    }

    @Override
```

```
    public void save() {
        decoratedDocument.save();
    }

    @Override
    public void close() {
        decoratedDocument.close();
    }
}

// Concrete Decorators
class SpellCheckDecorator extends DocumentDecorator {
    public SpellCheckDecorator(Document decoratedDocument) {
        super(decoratedDocument);
    }

    @Override
    public void save() {
        System.out.println("Checking spelling before saving...");
        super.save();
    }
}

class GrammarCheckDecorator extends DocumentDecorator {
    public GrammarCheckDecorator(Document decoratedDocument) {
        super(decoratedDocument);
    }
```

```
    @Override
    public void save() {
        System.out.println("Checking grammar before saving...");
        super.save();
    }
}

class PlagiarismCheckDecorator extends DocumentDecorator {
    public PlagiarismCheckDecorator(Document decoratedDocument) {
        super(decoratedDocument);
    }

    @Override
    public void save() {
        System.out.println("Checking for plagiarism before saving...");
        super.save();
    }
}

// Client Code
public class DocumentEditor {
    public static void main(String[] args) {
        Document doc = new WordDocument();

        // Add spell check and grammar check
        doc = new SpellCheckDecorator(doc);
```

```
        doc = new GrammarCheckDecorator(doc);

        // Optionally add plagiarism check
        doc = new PlagiarismCheckDecorator(doc);

        doc.open();
        doc.save();
        doc.close();
    }
}
```

Key Points to Remember:

- Decorator Pattern allows you to add additional behavior to objects dynamically and flexibly.
- You can combine multiple decorators in different orders to achieve the desired behavior.
- The pattern avoids subclass explosion by allowing you to compose features rather than inheriting them.