Facade Pattern

Scenario:

Imagine you're working on a document editor that offers a wide range of features such as spell-checking, grammar-checking, formatting, exporting to various file formats, and printing. Each of these features may involve multiple classes and complex interactions. You want to provide a simplified interface for users and client code, making it easy to access these features without dealing with the underlying complexity.

Purpose:

The Facade Pattern provides a unified and simplified interface to a complex subsystem. It hides the complexity of the subsystem by exposing a single, easy-to-use interface, making it easier for clients to interact with the system.

When to Use:

- When you want to simplify the interface of a complex subsystem.
- When you want to decouple a client from the subsystem, reducing the number of dependencies and simplifying the client's interaction with the subsystem.
- When you need to provide a higher-level interface that makes the subsystem easier to use.

Key Concepts:

- Facade: The class that provides a simplified interface to the subsystem. It delegates client requests to the appropriate subsystem classes.
- Subsystem Classes: The complex classes that perform the actual work. These classes are usually part of a larger, more complex system.
- Client: The code that interacts with the facade rather than directly with the subsystem classes.

How It Works:

Facade Class: Provides a simplified interface that the client uses to interact with the subsystem. It hides the complexity by delegating tasks to the appropriate subsystem classes.

Subsystem Classes: The classes that perform the actual work. They contain the complex logic and detailed operations that the facade simplifies.

Client: Uses the facade to interact with the subsystem, unaware of the internal complexity.

Implementation Example:

Here's how the implementation might look in Java:

```
// Subsystem Classes
class SpellChecker {
    public void checkSpelling(String document) {
        System.out.println("Checking spelling for document: " + document);
    }
}

class GrammarChecker {
    public void checkGrammar(String document) {
        System.out.println("Checking grammar for document: " + document);
    }
}
```

```
class Formatter {
    public void formatDocument(String document) {
        System.out.println("Formatting document: " + document);
    }
}

class PDFExporter {
    public void exportToPDF(String document) {
        System.out.println("Exporting document to PDF: " + document);
    }
}

class Printer {
    public void printDocument(String document) {
        System.out.println("Printing document: " + document);
    }
}

// Facade Class
class DocumentFacade {
    private SpellChecker spellChecker;
    private GrammarChecker grammarChecker;
    private Formatter formatter;
    private PDFExporter pdfExporter;
    private Printer printer;
```

```
    public DocumentFacade() {
        spellChecker = new SpellChecker();
        grammarChecker = new GrammarChecker();
        formatter = new Formatter();
        pdfExporter = new PDFExporter();
        printer = new Printer();
    }

    public void prepareAndPrintDocument(String document) {
        spellChecker.checkSpelling(document);
        grammarChecker.checkGrammar(document);
        formatter.formatDocument(document);
        printer.printDocument(document);
    }

    public void prepareAndExportDocument(String document) {
        spellChecker.checkSpelling(document);
        grammarChecker.checkGrammar(document);
        formatter.formatDocument(document);
        pdfExporter.exportToPDF(document);
    }
}

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

```
        DocumentFacade documentFacade = new DocumentFacade();

        String document = "Design Patterns in Java";

        System.out.println("Printing Document:");
        documentFacade.prepareAndPrintDocument(document);

        System.out.println("\nExporting Document:");
        documentFacade.prepareAndExportDocument(document);
    }
}
```

Key Points to Remember:

Facade Pattern provides a simplified interface to a complex subsystem, making it easier for clients to use the subsystem without needing to understand its complexity.

Decoupling: The pattern reduces the dependencies between the client and the subsystem by introducing a facade layer, making the code easier to maintain and extend.

Advantages:

- Simplifies Interface: The facade simplifies the interface for the client, making the system easier to use.
- Decouples Client and Subsystem: The pattern decouples the client from the subsystem, reducing the number of dependencies and making the system more maintainable.
- Promotes Reusability: The facade can be reused by different clients, providing a consistent interface to the subsystem.

Disadvantages:

- Limited Flexibility: The facade provides a simplified interface, but it might not expose all the features of the subsystem, limiting flexibility.

- Potential Overhead: The facade introduces an additional layer of abstraction, which can add some overhead.