Adapter Pattern with Document Editor Example

Customer Request:

"I need a document editor that can handle different document formats like Word, PDF, and Text, but I also need to ensure that the editor can support different versions of documents. For example, I might have a Word document with a simple format, and later, I might upgrade to a more complex format with additional features. I want the system to be flexible enough to allow these upgrades without rewriting everything."

Choosing the Right Pattern:

Pattern Recommendation: Adapter Pattern

Why?

The Adapter Pattern is ideal for this scenario because:

- Compatibility: The customer needs to support different versions or formats of documents that may not have a compatible interface. The Adapter Pattern allows you to adapt an interface to be compatible with another without changing the underlying code.

- Flexibility for Upgrades: As the customer upgrades or changes document formats, the Adapter Pattern can help integrate these new formats into the existing system by providing an adapter that translates one interface into another.

- Seamless Integration: The Adapter Pattern allows new versions or different formats to be used in a system without modifying the existing client code, making the transition smooth and less risky.

Pattern Explanation: Adapter Pattern

Key Concepts:

- Target: The interface that the client expects to work with.

- Adaptee: The existing interface or class that needs to be adapted.

- Adapter: A class that implements the target interface and translates requests to the adaptee.

How It Works:

- Target Interface: Defines the interface that the client interacts with. In this case, it might be the basic document operations like open(), save(), and close().

- Adaptee: Represents the existing or legacy document class that doesn't match the new interface.

- Adapter: Implements the target interface and internally uses the adaptee to perform the necessary operations, translating the calls from the target to the adaptee.

Implementation Example:

Here?s how the implementation might look in Java:

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

// Existing Class (Adaptee) with a Different Interface
class LegacyWordDocument {
```

```
    public void start() {
        System.out.println("Starting Legacy Word document.");
    }
    public void write() {
        System.out.println("Writing Legacy Word document.");
    }
    public void shutdown() {
        System.out.println("Shutting down Legacy Word document.");
    }
}

// Adapter Class that Adapts LegacyWordDocument to the Document Interface
class WordDocumentAdapter implements Document {
    private LegacyWordDocument legacyDoc;

    public WordDocumentAdapter(LegacyWordDocument legacyDoc) {
        this.legacyDoc = legacyDoc;
    }

    @Override
    public void open() {
        legacyDoc.start();
    }

    @Override
    public void save() {
        legacyDoc.write();
```

```
    }

    @Override
    public void close() {
        legacyDoc.shutdown();
    }
}

// Client Code
public class DocumentEditor {
    public static void main(String[] args) {
        // Client expects a Document interface
        Document doc = new WordDocumentAdapter(new LegacyWordDocument());
        doc.open();
        doc.save();
        doc.close();
    }
}
```

Key Points to Remember:

- Adapter Pattern allows you to integrate incompatible interfaces or classes into your system by providing a wrapper that adapts one interface to another.
- It?s particularly useful when dealing with legacy code or third-party libraries where modifying the original source is not feasible.

When to Use the Adapter Pattern:

- When you need to use an existing class but its interface doesn?t match the one your client

expects.

- When you want to create a reusable class that cooperates with unrelated or unforeseen classes that don?t have compatible interfaces.

Advantages:

- Seamless Integration: Allows you to integrate new features or versions without changing the existing code.
- Reusability: Promotes reusability of existing classes by adapting them to new interfaces.

Disadvantages:

- Increased Complexity: Introduces additional classes, which can make the system more complex.