Scenario 17: Customer Request

Customer Request:

"I need a document editor that supports different formats like Word, PDF, and Text. Each format has its own set of operations, and I want to ensure that the document editor can handle these different formats while keeping the codebase flexible and extendable. Additionally, the editor should support adding new formats in the future without altering the existing code."

Choosing the Right Pattern:

Pattern Recommendation: Abstract Factory Pattern

Why?

The Abstract Factory Pattern is ideal for this scenario because:

Product Families: The customer needs to support different document formats (like Word, PDF, Text) with their own set of operations. The Abstract Factory Pattern provides an interface for creating families of related or dependent objects without specifying their concrete classes.

Scalability and Extensibility: The pattern allows you to add new formats in the future (e.g., HTML, Markdown) without changing the existing code, promoting a flexible and scalable design.

Consistency: The Abstract Factory Pattern ensures that the created objects are consistent with each other, which is essential when dealing with related products like document formats.

Pattern Explanation: Abstract Factory Pattern

Key Concepts:

Abstract Factory: An interface that declares a set of methods for creating abstract product objects (e.g., createWordDocument(), createPDFDocument(), createTextDocument()).

Concrete Factory: A class that implements the Abstract Factory interface to create specific types of products (e.g., WordDocumentFactory, PDFDocumentFactory).

Abstract Product: An interface that declares methods for products that belong to the same family (e.g., Document).

Concrete Product: A class that implements the Abstract Product interface to define a specific product (e.g., WordDocument, PDFDocument).

How It Works:

Abstract Factory Interface: Defines methods for creating different abstract products (e.g., createWordDocument(), createPDFDocument()).

Concrete Factory: Implements the Abstract Factory interface to create specific variations of the products (e.g., WordDocumentFactory for Word documents, PDFDocumentFactory for PDF documents).

Abstract Product Interface: Represents the common interface for each type of product in the family (e.g., Document for Word, PDF, Text documents).

Concrete Product: Represents a specific implementation of the product interface (e.g., WordDocument, PDFDocument).

Client: Uses the factory to create objects without needing to know the exact class of the object being created.

Implementation Example:

Here's how the implementation might look in Java:

```
// Abstract Product
interface Document {
    void open();
    void save();
    void close();
}

// Concrete Products
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.");
    }
}

class TextDocument implements Document {
    @Override
    public void open() {
        System.out.println("Opening Text document.");
    }
```

```
    @Override
    public void save() {
        System.out.println("Saving Text document.");
    }

    @Override
    public void close() {
        System.out.println("Closing Text document.");
    }
}

// Abstract Factory
interface DocumentFactory {
    Document createDocument();
}

// Concrete Factories
class WordDocumentFactory implements DocumentFactory {
    @Override
    public Document createDocument() {
        return new WordDocument();
    }
}

class PDFDocumentFactory implements DocumentFactory {
    @Override
```

```
    public Document createDocument() {
        return new PDFDocument();
    }
}

class TextDocumentFactory implements DocumentFactory {
    @Override
    public Document createDocument() {
        return new TextDocument();
    }
}

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

        // Create Word document
        factory = new WordDocumentFactory();
        Document doc = factory.createDocument();
        doc.open();
        doc.save();
        doc.close();

        // Create PDF document
        factory = new PDFDocumentFactory();
```

```
        doc = factory.createDocument();
        doc.open();
        doc.save();
        doc.close();

        // Create Text document
        factory = new TextDocumentFactory();
        doc = factory.createDocument();
        doc.open();
        doc.save();
        doc.close();
    }
```

Key Points to Remember:

Abstract Factory Pattern allows you to create families of related products without specifying their concrete classes.

It ensures that the created objects are consistent and promotes scalability and flexibility in the system.

When to Use the Abstract Factory Pattern:

When you need to create families of related products (like different document formats) and ensure they are compatible.

When you want to add new product families (like new document formats) in the future without changing existing code.

Advantages:

- Scalability: New product families can be added easily without modifying existing code.
- Consistency: Ensures that products within a family are compatible with each other.
- Decoupling: Decouples the creation of objects from their usage, promoting flexible and maintainable code.

Disadvantages:

- Complexity: The pattern can introduce additional complexity by requiring multiple factory classes and interfaces.
- Overhead: It may add unnecessary overhead if the system doesn't require the creation of product families.