Abstract Factory Pattern with Document Editor Example

Customer Request:

"I need a document editor that can handle different document formats like Word, PDF, and Text. However, the document editor should be able to create different variations of these documents based on certain configurations (e.g., a standard Word document vs. a Word document with a specific template). I want to ensure that the editor can easily switch between these variations without significant changes to the 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 create different variations of documents, where each variation belongs to a "family" of products (e.g., Word documents with different templates). The Abstract Factory Pattern allows you to create families of related or dependent objects without specifying their concrete classes.

- Switching Between Variations: The customer wants the ability to switch between different variations easily. The Abstract Factory Pattern enables the creation of different document variations based on configurations, while keeping the client code independent of the specific classes.

- Consistency Across Products: The pattern ensures that all products created by a particular factory are compatible, maintaining consistency across different variations of documents.

Pattern Explanation: Abstract Factory Pattern

Key Concepts:

- Abstract Factory: An interface that declares a set of methods for creating abstract product objects.

- Concrete Factory: A class that implements the Abstract Factory interface to create specific types of products.

- Abstract Product: An interface that declares methods for products that belong to the same family.

- Concrete Product: A class that implements the Abstract Product interface to define a specific product.

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., StandardDocumentFactory for standard documents, TemplateDocumentFactory for documents with specific templates).

- 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., StandardWordDocument, TemplateWordDocument).

Implementation Example:

Here?s how the implementation might look in Java:

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

// Concrete Products for Standard Documents
class StandardWordDocument implements Document {
    @Override
    public void open() {
        System.out.println("Opening standard Word document.");
    }
    @Override
    public void save() {
        System.out.println("Saving standard Word document.");
    }
    @Override
    public void close() {
        System.out.println("Closing standard Word document.");
    }
}

class StandardPDFDocument implements Document {
    @Override
    public void open() {
        System.out.println("Opening standard PDF document.");
```

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

// Concrete Products for Template-Based Documents
class TemplateWordDocument implements Document {
    @Override
    public void open() {
        System.out.println("Opening Word document with template.");
    }
    @Override
    public void save() {
        System.out.println("Saving Word document with template.");
    }
    @Override
    public void close() {
        System.out.println("Closing Word document with template.");
    }
}
```

```
class TemplatePDFDocument implements Document {
    @Override
    public void open() {
        System.out.println("Opening PDF document with template.");
    }
    @Override
    public void save() {
        System.out.println("Saving PDF document with template.");
    }
    @Override
    public void close() {
        System.out.println("Closing PDF document with template.");
    }
}

// Abstract Factory Interface
interface DocumentFactory {
    Document createWordDocument();
    Document createPDFDocument();
}

// Concrete Factory for Standard Documents
class StandardDocumentFactory implements DocumentFactory {
    @Override
    public Document createWordDocument() {
        return new StandardWordDocument();
    }
```

```
    @Override
    public Document createPDFDocument() {
        return new StandardPDFDocument();
    }
}

// Concrete Factory for Template-Based Documents
class TemplateDocumentFactory implements DocumentFactory {
    @Override
    public Document createWordDocument() {
        return new TemplateWordDocument();
    }
    @Override
    public Document createPDFDocument() {
        return new TemplatePDFDocument();
    }
}

// Client Code
public class DocumentEditor {
    private DocumentFactory factory;

    public DocumentEditor(DocumentFactory factory) {
        this.factory = factory;
    }

    public void createDocuments() {
```

```
        Document wordDoc = factory.createWordDocument();
        wordDoc.open();
        wordDoc.save();
        wordDoc.close();

        Document pdfDoc = factory.createPDFDocument();
        pdfDoc.open();
        pdfDoc.save();
        pdfDoc.close();
    }

    public static void main(String[] args) {
        DocumentEditor editor = new DocumentEditor(new StandardDocumentFactory());
        editor.createDocuments();

        System.out.println("--- Switching to template-based documents ---");

        editor = new DocumentEditor(new TemplateDocumentFactory());
        editor.createDocuments();
    }
}
```

Key Points to Remember:

- Abstract Factory Pattern allows you to create families of related or dependent objects without specifying their concrete classes.
- It?s particularly useful when you need to ensure consistency across related products (e.g., documents and their templates) and want to switch between product families easily.

When to Use the Abstract Factory Pattern:

- When you need to create families of related products (e.g., standard vs. template-based documents) and ensure they are compatible.
- When you want to switch between different product families easily without modifying existing client code.

Advantages:

- Consistency Across Products: Ensures that all products created by a factory are compatible.
- Scalability: Makes it easy to add new product families without changing existing code.

Disadvantages:

- Complexity: Can introduce additional complexity by adding multiple classes and interfaces.