8 Creational: Assignment — Factory Method Pattern for a Document Processor

**Objective** Show you can decouple object creation from client code with a Factory Method, extend the system for new formats without touching existing clients, and explain the architecture choice.

#### **Starter code (src/main/java/legacy/)**

package legacy;

public interface Document { void open(); }

package legacy;

public class PDFDocument implements Document {

public void open(){ System.out.println("Opening legacy PDF"); }

}

package legacy;

public class WordDocument implements Document {

public void open(){ System.out.println("Opening legacy Word"); }

}

package legacy;

public class Client {

public static void main(String[] args){

new PDFDocument().open();

new WordDocument().open();

}

}

#### **Tasks**

1 analysis/fm\_problems.md — explain why direct new violates Open / Closed and increases coupling.  
 2 Create abstract creator DocumentFactory with createDocument() and helper openDocument().  
 3 Add concrete factories PDFDocumentFactory, WordDocumentFactory.  
 4 Refactor client into clean.Application that receives a DocumentFactory, calls openDocument(), and knows nothing about concrete classes.  
 5 Introduce a new format MarkdownDocument plus MarkdownDocumentFactory; prove Application opens it without modification.  
 6 reflection.md — What makes this Factory Method (not Simple Factory)? Which pattern family does it belong to and why is it more extensible?

#### **Deliverables**

analysis/fm\_problems.md

src/main/java/clean/document/\*\* ← Document, concrete docs

src/main/java/clean/factory/\*\* ← abstract + concrete factories

src/main/java/clean/Application.java

src/main/java/clean/Main.java ← chooses factory at runtime

reflection.md

README.md

#### **Solution reference**

// clean/document/Document.java

package clean.document;

public interface Document { void open(); }

// clean/document/PDFDocument.java

package clean.document;

public class PDFDocument implements Document {

public void open(){ System.out.println("Opening a PDF document."); }

}

// clean/document/WordDocument.java

package clean.document;

public class WordDocument implements Document {

public void open(){ System.out.println("Opening a Word document."); }

}

// clean/document/MarkdownDocument.java

package clean.document;

public class MarkdownDocument implements Document {

public void open(){ System.out.println("Opening a Markdown document."); }

}

// clean/factory/DocumentFactory.java

package clean.factory;

import clean.document.Document;

public abstract class DocumentFactory {

public abstract Document createDocument();

public void openDocument(){ createDocument().open(); }

}

// clean/factory/PDFDocumentFactory.java

package clean.factory;

import clean.document.\*;

public class PDFDocumentFactory extends DocumentFactory {

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

}

// clean/factory/WordDocumentFactory.java

package clean.factory;

import clean.document.\*;

public class WordDocumentFactory extends DocumentFactory {

public Document createDocument(){ return new WordDocument(); }

}

// clean/factory/MarkdownDocumentFactory.java

package clean.factory;

import clean.document.\*;

public class MarkdownDocumentFactory extends DocumentFactory {

public Document createDocument(){ return new MarkdownDocument(); }

}

// clean/Application.java

package clean;

import clean.factory.DocumentFactory;

public class Application {

private final DocumentFactory factory;

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

public void run(){ factory.openDocument(); }

}

// clean/Main.java

package clean;

import clean.factory.\*;

public class Main {

public static void main(String[] args){

String type = "md"; // could be CLI input

DocumentFactory factory =

"pdf".equalsIgnoreCase(type) ? new PDFDocumentFactory() :

"doc".equalsIgnoreCase(type) ? new WordDocumentFactory() :

new MarkdownDocumentFactory();

new Application(factory).run();

}

}

*Architecture pattern used*: **Factory Method** — each concrete creator overrides createDocument() to return its own product, letting the high-level Application depend solely on the abstract DocumentFactory while supporting future formats by subclassing instead of editing existing code.