**Factory Method Pattern Example Documentation**

**1. Overview**

This Java program demonstrates the implementation of the Factory Method design pattern. It allows users to create different types of document objects (Word, PDF, Excel) without explicitly specifying their classes. The program showcases how the Factory Method pattern can be used to create objects based on user input.

**2. Design Pattern**

**Factory Method Pattern**

The Factory Method is a creational design pattern that provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created.

**3. classes**

**Document Interface: The code starts by defining a 'Document' interface. This interface declares three methods: open(), close(), and save(). These methods represent common operations that can be performed on any type of document.**

**Concrete Document Classes: Three classes (WordDocument, PdfDocument, and ExcelDocument) implement the Document interface. Each class provides its own implementation of the open(), close(), and save() methods, printing messages specific to that document type.**

**Abstract Factory Class: An abstract class called DocumentFactory is defined. This class declares an abstract method createDocument() which returns a Document object. This is the core of the Factory Method pattern.**

**Concrete Factory Classes: For each document type, there's a corresponding factory class (WordDocumentFactory, PdfDocumentFactory, ExcelDocumentFactory). Each of these extends DocumentFactory and overrides the createDocument() method to return an instance of its specific document type.**

**Main Class (FactoryMethodPatternExample): The main class demonstrates the use of the Factory Method pattern. It does the following:**

* + **Prompts the user to enter a document type.**
  + **Uses a switch statement to create the appropriate factory based on user input.**
  + **Uses the factory to create a document object.**
  + **Calls the open(), save(), and close() methods on the created document.**

**User Interaction: The program uses a Scanner to get user input for the document type. It then uses this input to determine which factory to use.**

**Object Creation and Usage: Once the appropriate factory is selected, it's used to create a Document object. The type of document created depends on the factory used, but the main code doesn't need to know the specific type - it just uses the Document interface methods.**

**This implementation demonstrates the key principle of the Factory Method pattern: it allows the program to create objects without specifying the exact class of object to be created. The decision about which class to instantiate is deferred to the subclasses (the concrete factories).**

**The benefit of this approach is that it makes the code more flexible and extensible. If you want to add a new document type, you can do so by adding a new concrete Document class and a corresponding Factory class, without needing to modify the existing code that uses documents.**

**Implementation:**

**Link :**[**Click here for the code**](https://github.com/Akashmondal55/Akash_5016855/tree/main/Week-1/Design%20patern%20and%20princple/Exercise-2)

**Conclusion:**

**This implementation of the Factory Method pattern demonstrates a flexible and extensible approach to object creation in software design. The key strengths and benefits of this pattern, as showcased in this example, are:**

1. **Abstraction: The pattern abstracts the object creation process, allowing the client code to work with generic Document interfaces rather than specific implementations.**
2. **Flexibility: New document types can be added easily by creating new concrete Document classes and corresponding Factory classes, without modifying existing code.**
3. **Separation of Concerns: The object creation logic is separated from the object usage, promoting cleaner and more maintainable code.**
4. **Open/Closed Principle: The design adheres to the Open/Closed Principle, as it's open for extension (new document types) but closed for modification of existing code.**
5. **Polymorphism: The pattern leverages polymorphism, allowing different document types to be treated uniformly through the Document interface.**
6. **Scalability: As the application grows, new document types can be integrated seamlessly without affecting the core logic.**
7. **Testability: The factory method pattern facilitates easier unit testing by allowing mock objects to be injected for testing purposes.**

**While this example is relatively simple, it illustrates how the Factory Method pattern can be applied to real-world scenarios where object creation needs to be flexible and decoupled from the client code. This pattern is particularly useful in larger systems where the types of objects to be created may vary or expand over time.**

**However, it's worth noting that this pattern does introduce additional complexity through the creation of multiple classes and interfaces. Therefore, it's most beneficial in scenarios where the flexibility and extensibility it provides outweigh the added complexity.**