1. **Problem Trying To Solve**

- To develop a Java application capable of reading, processing, and outputting currency transactions from various file formats (JSON, CSV, XML).

- It addresses the below challenges :
    1. Multi-Format Support: The application can handle multiple data formats (JSON, CSV, XML), each requiring specific parsing logic to extract transaction details.

    2. Currency Code Validation: It validates currency codes in each transaction against a recognized list to ensure accuracy and prevent processing errors.

    3. Accurate Currency Conversion: The system converts currency values based on predefined exchange rates, including handling multi-step conversions if direct rates are not available.

    4. Robust Error Handling: Errors encountered during parsing, validation, or conversion are managed effectively, ensuring the application's stability and providing clear feedback on transaction statuses.

    5. Efficient Data Processing: Optimized for performance to handle large volumes of transactions quickly and efficiently.

    6. Consistent and Clear Output: Outputs are formatted in the same file type as inputs, with detailed information on transaction outcomes, enhancing usability for further processing or audits.

2. **Design Patterns:**

   Strategy Pattern: Useful for defining a family of algorithms (parsers for different file formats) and making them interchangeable. To implement a common interface for parsing that can be extended for each specific format.

   - **Usage**: This pattern is employed in the handling of different file formats. The IFileHandler interface defines a common set of operations (readData and writeData), and each specific file type (CSV, JSON, XML) has a corresponding class (CSVFileHandler, JSONFileHandler, XMLFileHandler) that implements this interface.

   Factory Pattern: Helps in creating objects without specifying the exact class of object that will be created. For the file type as it could vary, and the corresponding parser and writer need to be instantiated.

- **Usage:** The FileHandlerFactory class is used to instantiate objects of IFileHandler. It decides which file handler to return based on the file type determined by FileTypeUtil.

3. **The consequences of using design pattern(s).**

Consequences of Using the Strategy Pattern
Advantages:

- Flexibility: It allows the application to switch between different file handling strategies easily, making it robust to changes such as the addition of new file formats without modifying existing code.
- Maintainability: Each file handler can be maintained independently, promoting cleaner, more organized code that adheres to the Single Responsibility Principle.

Disadvantages:

- Complexity: Introducing multiple strategies can increase the complexity of the codebase, as each strategy must be implemented and maintained separately.
- Overhead: There might be a runtime overhead associated with determining the appropriate strategy, though minimal, and increased memory usage due to multiple handler objects if not managed properly.

Consequences of Using the Factory Pattern
Advantages:

- Centralized Object Creation: This pattern centralizes the creation of file handler objects, reducing duplications in object creation logic and decreasing the risk of errors.
- Decoupling: Clients remain decoupled from the specific classes that implement IFileHandler, which simplifies the dependencies within the application.

Disadvantages:

- Dependency on Factory Logic: The application becomes dependent on the factory logic for creating instances, which can become complex if the creation logic involves many conditions.
- Modification Requirements: Adding new file formats requires modifications to the factory logic, which can lead to errors if not done carefully.

4. **UML Diagram**
   https://github.com/gopinathsjsu/individual-project-SnehaOdugoudar/blob/branch_2/PART%20-%20A/UML_Diagram.png