Part 1: Credit Card Validation Design

Primary Problem

How can a system read credit card records from a CSV file, identify the correct card type from the card number, validate the card number, and instantiate the appropriate credit card object (VisaCC, MasterCC, AmExCC, etc.) while maintaining flexibility for future card types?

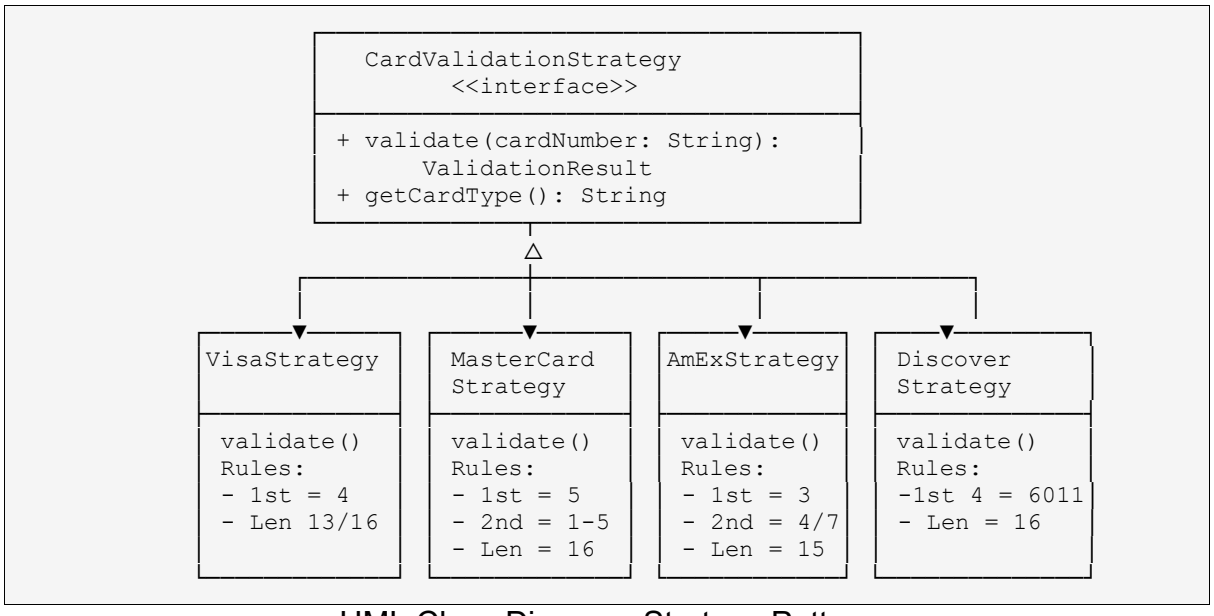
Secondary Problems

1. **Card Type Identification:** Determining which credit card class to instantiate based on specific rules for each card issuer (first digit, second digit, length).
2. **Validation Logic Encapsulation:** Implementing card-specific validation rules without creating a monolithic class with multiple if-else statements.
3. **Object Creation:** Creating the correct card object type based on validated information.

Design Patterns Used

1. Strategy Pattern (Behavioral)

Different credit card types use different validation algorithms, yet the client code should not need to know about these differences. By using the Strategy Pattern, each card type's validation logic is isolated in its own independent class, making the system easier to test and maintain. The Factory Pattern determines the card type and selects the appropriate strategy, allowing the strategy to validate the specific rules without the factory needing to understand the details. This design not only simplifies extensibility, since new card types can be added by creating new strategy classes but also follows the Open/Closed Principle by enabling the system to grow without modifying existing components.

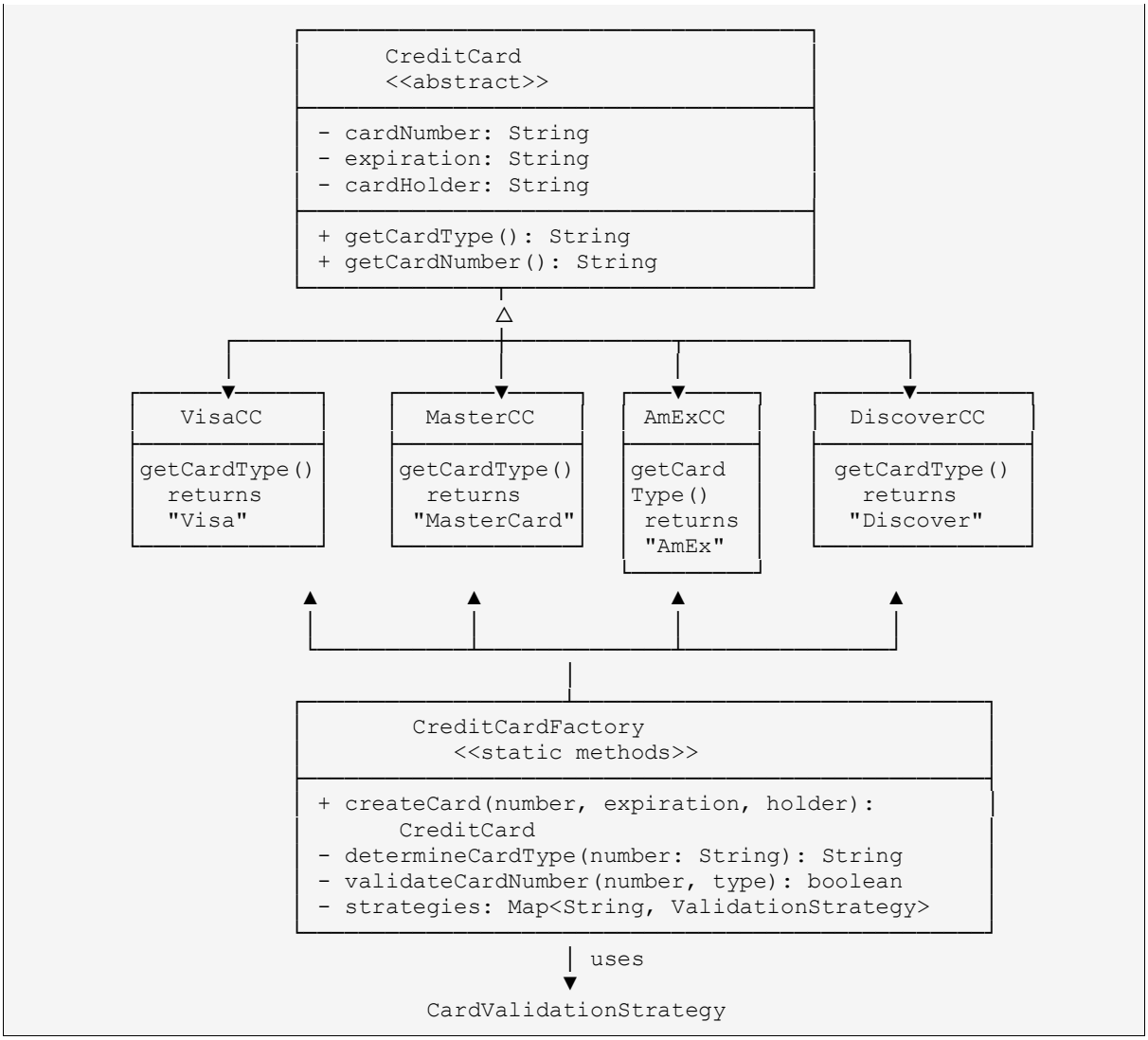


UML Class Diagram: Strategy Pattern

2. Factory Pattern (Creational)

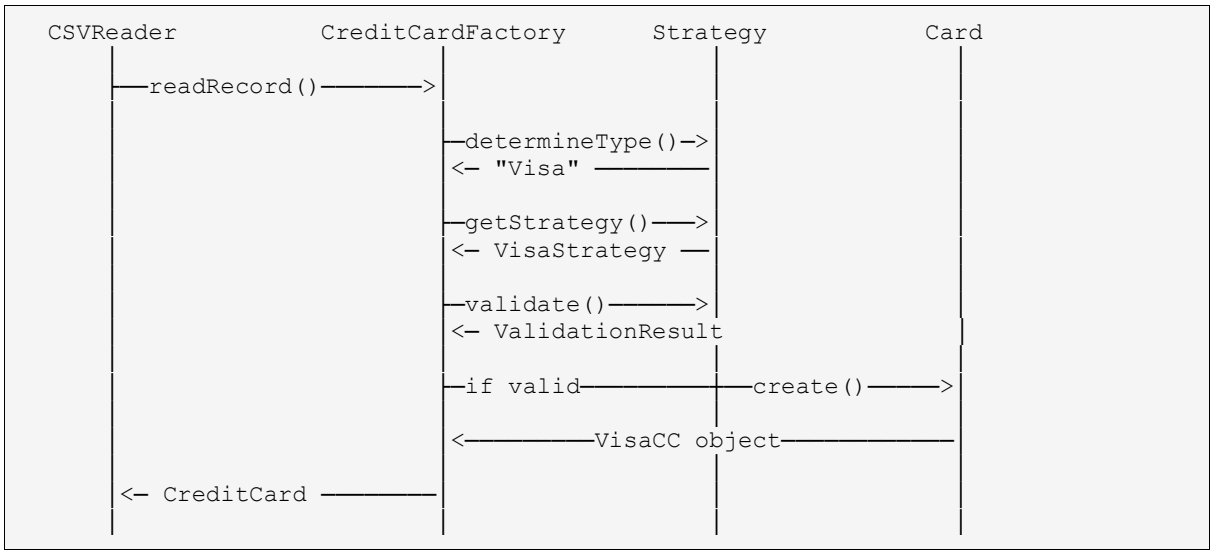
Object creation logic needs to be centralized and kept separate from the client code, which is achieved through the Factory Pattern. The factory receives the card data and first examines the card number to determine its type. It then selects the appropriate validation strategy and

uses that strategy to validate the number. If the card information is valid, the factory instantiates and returns the correct card class; if it is not valid, the factory instead returns an error. This approach keeps the creation process consistent, organized, and hidden from the client.



UML Class Diagram: Factory Pattern

The system follows a straightforward process flow to validate and create credit card objects. It begins by determining the card type through examination of the card number's prefix and length. Once the type is identified, the system retrieves the appropriate validation strategy and uses it to validate the card according to its specific rules. If the card is valid, the corresponding object such as VisaCC, MasterCC, or AmExCC is instantiated. Finally, the system returns either the created CreditCard object or an error if validation fails.



Sequence Diagram: Credit Card Validation Flow

Design Pattern Consequences:

The design patterns used in the system introduce several positive and negative consequences that balance one another. On the positive side, the system gains strong extensibility because adding new card types requires only minimal changes, but this benefit is paired with increased complexity, as the approach introduces more classes than a simple if-else structure. The use of single responsibility ensures that each class has one clear purpose, yet this clarity comes with a degree of indirection, requiring developers to understand interfaces and polymorphism. Testability is also improved since each strategy can be tested independently; however, this modularity brings a small performance overhead due to interface dispatch. Finally, maintainability is enhanced because validation logic is neatly organized by card type, though the factory must be properly configured to maintain accurate mappings between types and their strategies.

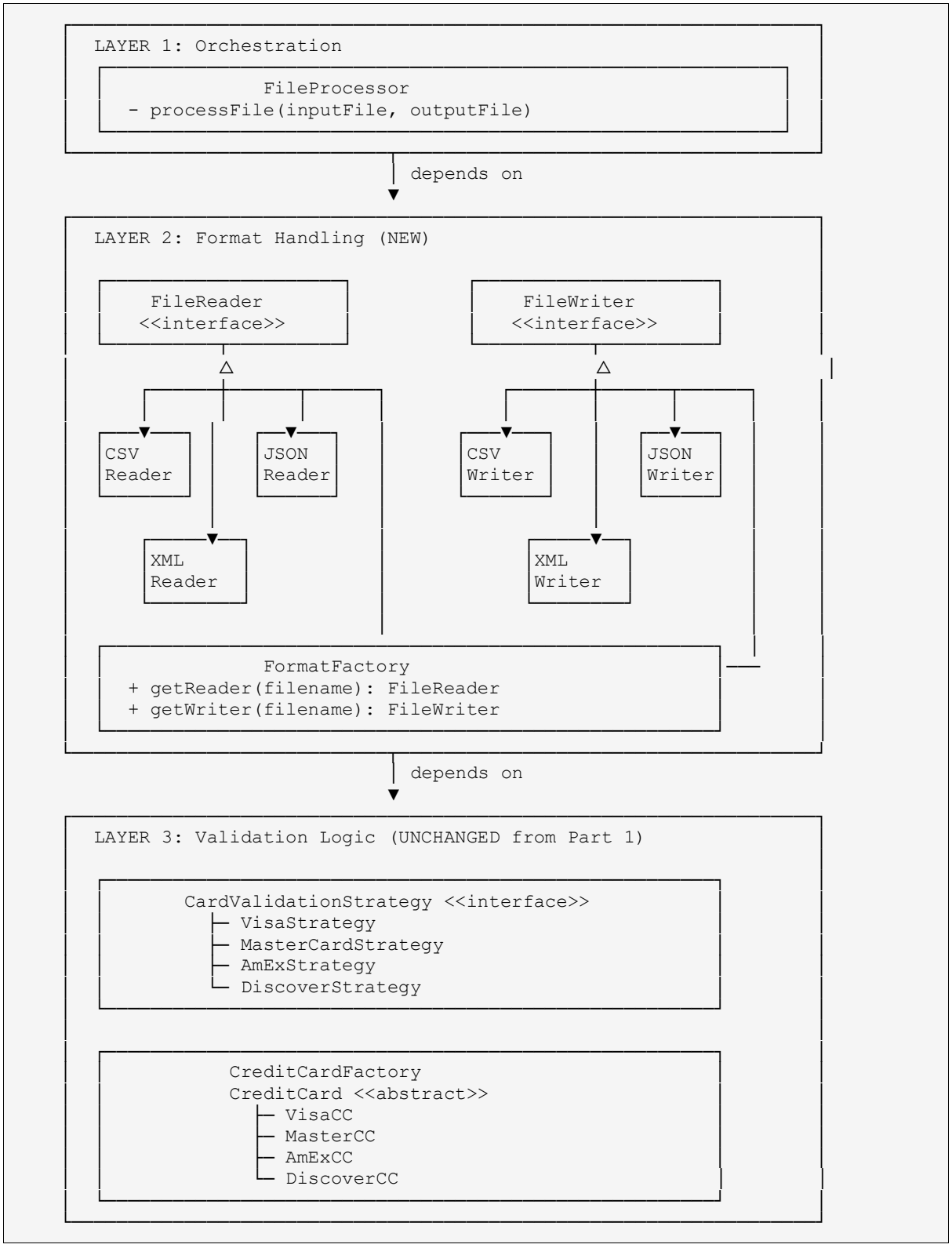
PART 2: Multi-Format File Processing Design

This extension enhances the Part 1 validation system by enabling it to read credit card data from CSV, JSON, and XML files. The processor automatically detects the input format, applies the same validation rules to each record, and writes the results back in the original format. The design also supports future formats without requiring changes to existing code, preserving scalability and the Open/Closed Principle.

Three-Layer Architecture:

The system is organized into three layers to maintain clear separation of responsibilities. The Orchestration Layer contains the FileProcessor, which manages the overall workflow. The Format Handling Layer introduces FileReader and FileWriter interfaces with concrete implementations for CSV, JSON, and XML, coordinated by a FormatFactory. The Validation

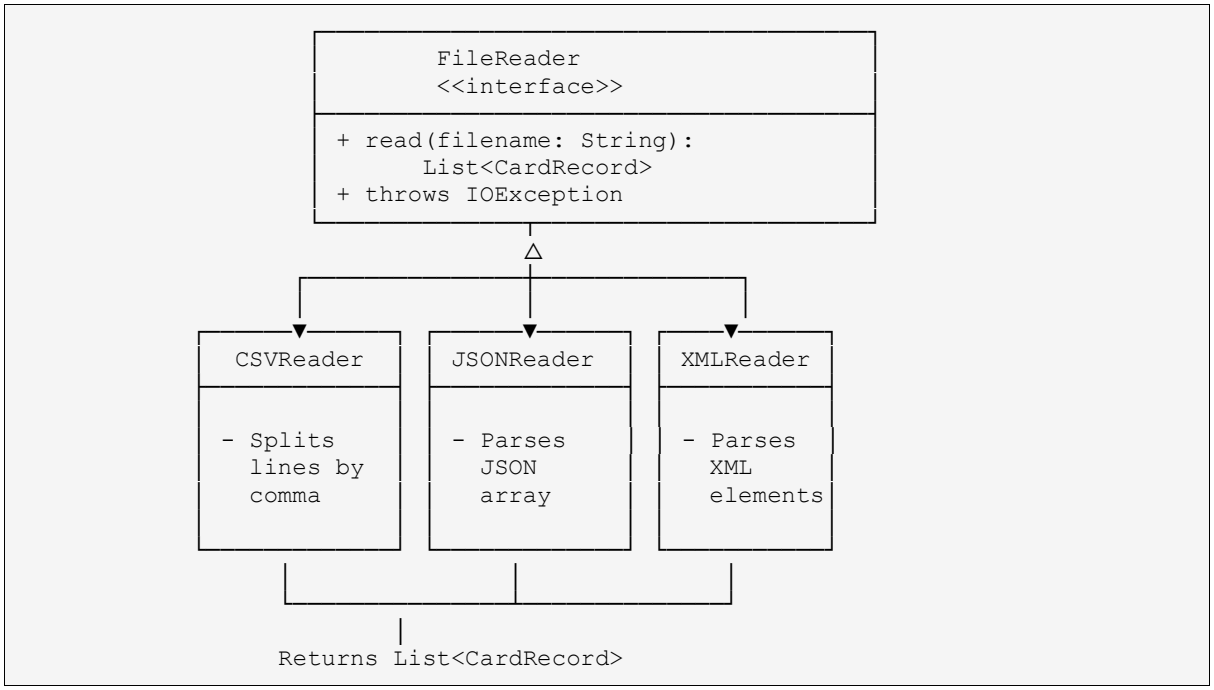
Layer reuses the strategies and factory from Part 1. This structure isolates format-specific logic from validation logic and improves maintainability.



Layered Architecture Diagram

Strategy Pattern for Reading Files:

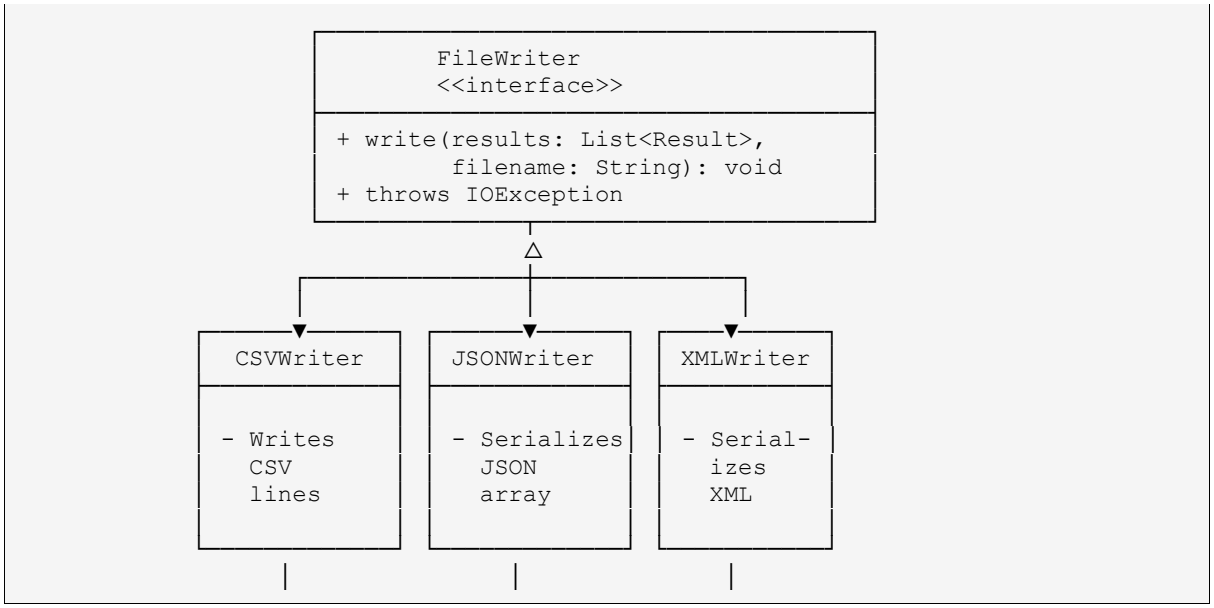
The system uses the Strategy Pattern to handle multiple file formats by defining a FileReader interface and implementing separate readers for CSV, JSON, and XML. Each reader parses its format and returns a list of card records. This keeps parsing logic independent and allows new formats to be added easily.

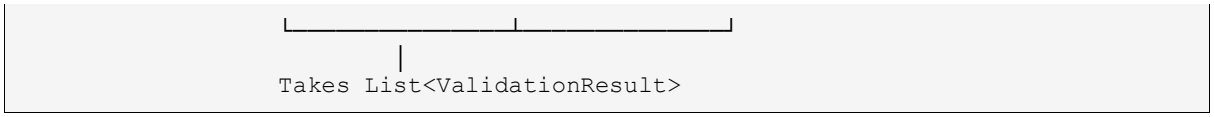


UML Class Diagram: FileReader Strategy

Strategy Pattern for Writing Results:

Writing results also relies on the Strategy Pattern. A FileWriter interface is implemented by CSVWriter, JSONWriter, and XMLWriter, each responsible for serializing results into its respective format. This ensures consistent output handling and simplifies extension to new formats.

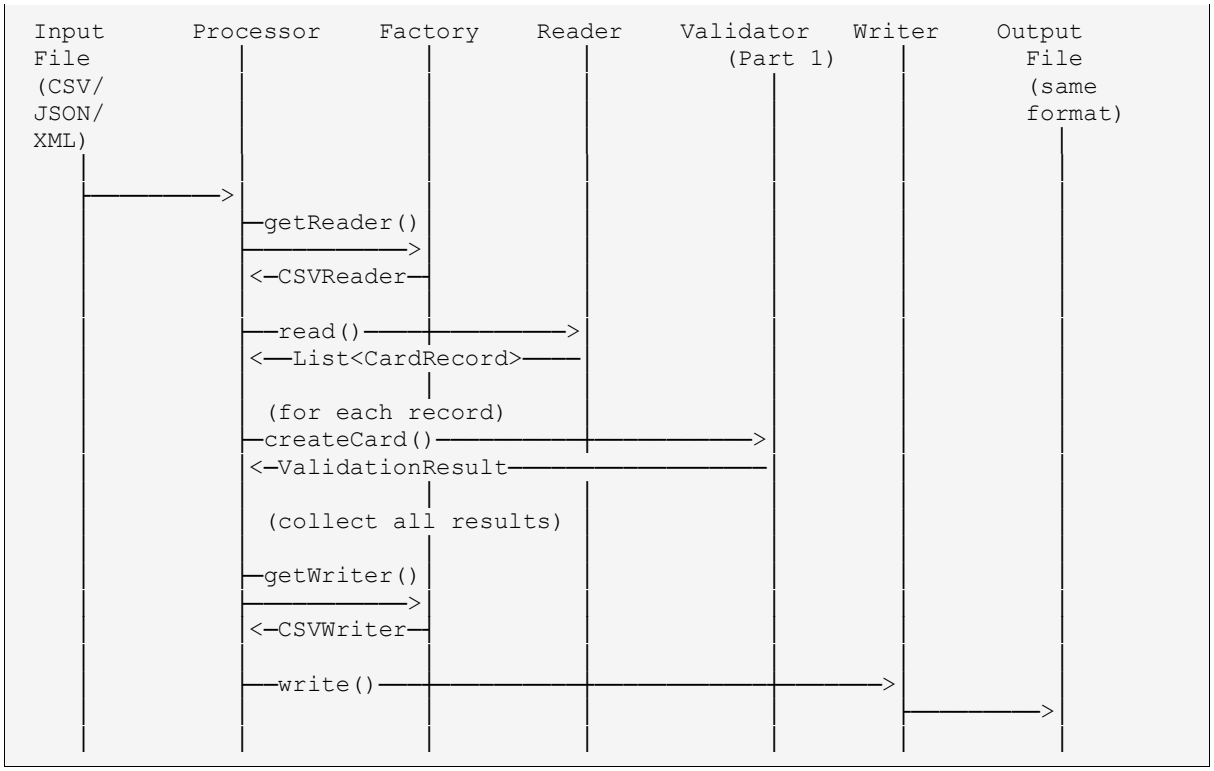




UML Class Diagram: FileWriter Strategy

Complete Processing Workflow:

The workflow begins with the processor selecting the correct reader based on file type, retrieving card records, and validating each using the Part 1 factory and strategies. After collecting results, the processor obtains the appropriate writer and outputs the results in the same format as the input. This unified process works for all supported formats because reading, validation, and writing are abstracted behind interfaces.



Sequence Diagram: Complete Multi-Format Processing

Extensibility: Supporting New Formats

Extending the system to support new formats, such as YAML, requires only adding new reader and writer classes and updating the FormatFactory. No other code changes are needed, and existing logic remains intact. This demonstrates strong adherence to the Open/Closed Principle.