Student Name: Sirisha Polisetty
Student ID: 016012477

Individual Project

Start Assignment

Due Friday by 11:59pm **Points** 100 **Submitting** a file upload

There are 3 parts to be submitted for Individual project.

Credit card problem:

Part 1: (20 points)

You have a **CSV** file that contains credit card records. Each record is on a line. It contains a field for the card number, the expiration date, and the name of the card holder. The fields are comma-separated. In your system you have the following class structure for the credit cards:

a class CreditCard,

classes VisaCC, MasterCC, AmExCC that are all subclasses of CreditCard,

you can assume more subclasses for other credit card types will be added later on.

You now have to design the method(s) (and maybe additional classes) that reads a record from the file, verifies that the credit card number is a possible account number, and creates an instance of the appropriate credit card class. What design patterns could you use for that?

Important details: Credit card numbers cannot exceed 19 digits, including a single check digit in the rightmost position. The exact algorithm for calculating the check digit is not important for this assignment. You can also determine the card issuer based on the credit card number:

MasterCard	First digit is a 5, second digit is in range 1 through 5 inclusive. Only valid length of number is 16 digits.
Visa	First digit is a 4. Length is either 13 or 16 digits.
AmericanExpress	First digit is a 3 and second digit a 4 or 7. Length is 15 digits.
Discover	First four digits are 6011. Length is 16 digits.

Deliverables:

Upload a PDF document containing the text and diagrams in your Git repo. You can use Astah tool for the diagrams.

- Describe what is the primary problem you try to solve.

- Describe what are the secondary problems you try to solve (if there are any).

- Describe what design pattern(s) you use how (use plain text and diagrams).

- Describe the consequences of using this/these pattern(s).

Hint: you face here (at least) two problems, one has to do with how you figure out what kind of card a specific record is about, the other one with how you create the appropriate objects. Look at behavioural patterns and at creational patterns.

Part2: 15 points (Design only)

Continue with the design from Part 1 and extend it to parse different input file formats (json, xml, csv) and detect the type of credit card and then output to a file (in the **same** format as the input - json or xml or csv) - with each line showing the card number, type of card (if a valid card number) and an error (if the card number is not valid). The design should accommodate newer file formats for the future.

Part 3: 65 points

Implement an application (Java code and JUnit tests) for Part 1 and Part2 - that accepts input file name and output file name and writes an output file in the same format as the input (CSV or JSON or XML). Output should contain the details specified in Part 2.

Submit design and code in the github repo (classroom invite will be sent by grader)

Grader will provide sample JUnit formats and input file formats

input_file-1.csv

output_file.csv

input_file.json

output_file.json

input_file.xml

output_file.xml

Part 1:

To address the primary problem To read credit card records from a file, check the validity of the credit card numbers, and create instances of the relevant credit card classes based on the card type, we can apply the Factory Method and Strategy design patterns.
Factory Method Pattern:

- The Factory Method pattern enables us to create an interface for building objects, while giving subclasses the power to choose which class to instantiate. For instance, we can use a CreditCardFactory as the fundamental factory class that generates instances of the relevant credit card subclasses.
- The CreditCardFactory class includes a method called createCreditCard(). This method takes the credit card record as input and returns an instance of the corresponding credit card subclass.
- The CreditCardFactory's subclasses will use this factory method to generate distinct credit card objects like VisaCC, MasterCC, AmExCC, or DiscoverCC.

Strategy Pattern:

- The Strategy pattern enables us to group various methods for checking the validity of credit card numbers and identifying the card type. We can create a CreditCardValidator interface that specifies these methods, and each subclass of credit cards can implement this interface with their own unique validation and card type identification logic.
- The CreditCardValidator interface will define methods like isValidCardNumber() and getCardType().
- Each credit card subclass (VisaCC, MasterCC, AmExCC, etc.) will implement the CreditCardValidator interface and provide their specific implementation for these methods.

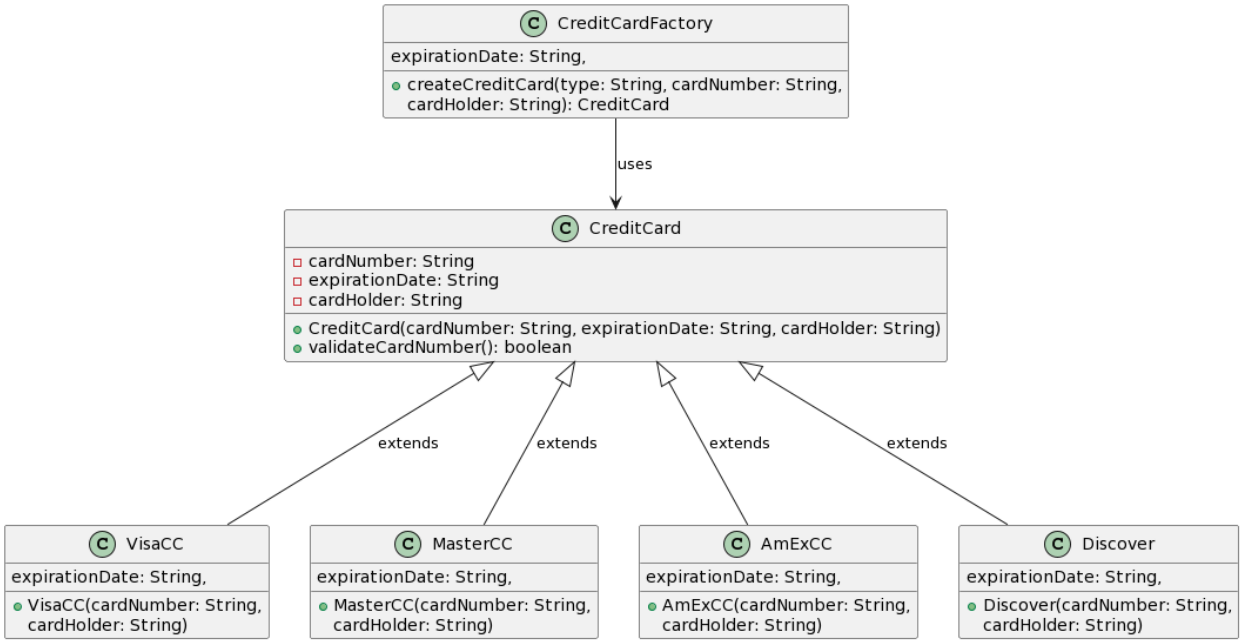
With these design patterns, we can achieve the following benefits:

- The Factory Method pattern allows us to encapsulate the object creation logic and provides a way to create instances of the appropriate credit card subclasses based on the card type. It promotes loose coupling between the client code and the credit card subclasses.

- The Strategy pattern allows us to encapsulate different algorithms for verifying the validity of credit card numbers and determining the card type. It promotes code reusability and flexibility, as we can easily add new credit card types or modify the validation logic without impacting the existing codebase.

In summary, this design ensures that credit card objects are created and verified accurately based on the information provided. It also prioritizes extensibility, maintainability, and separation of concerns for better management of the credit card records.

Using Factory Method for Credit Card Identification



In addition to the primary problem, there are **secondary** problems to address:

1. Identifying the Card Type: One of the secondary problems is determining the card type based on the credit card number. This involves examining specific patterns or rules associated with each card type, such as the first digit or a range of digits, to identify the card type (e.g., Visa, MasterCard, AmEx, Discover). This identification is essential for creating the appropriate credit card class instance.
2. Dynamic Creation of Credit Card Instances: Another secondary problem is dynamically creating the appropriate credit card class instance based on the identified card type. The design should allow for the dynamic creation of the specific subclass of CreditCard based on the determined card type. This ensures that the correct credit card object is created and associated with the corresponding card type.
3. Extensibility for New Credit Card Types: It is also important to design the system to be extensible, allowing for the addition of new credit card types in the future. As mentioned in the problem statement, there may be new subclasses of CreditCard added later on. Therefore, the design should be flexible enough to accommodate the inclusion of new credit card types without significant modifications to the existing codebase.

The consequences of using the Factory Method pattern are:

- The code becomes more flexible and extensible, allowing the addition of new credit card classes without modifying the existing code.
- The client code that creates credit card objects becomes decoupled from the specific credit card class implementations, making it easier to switch between different credit card types.
- The Factory Method encapsulates the complex object creation logic, improving code maintainability and readability.

The consequences of using the Strategy pattern are:

- The validation logic is encapsulated within each credit card class, allowing for different validation strategies for different card types.
- The code follows the Open-Closed Principle, as new validation strategies can be added without modifying the existing code.
- The Strategy pattern improves code reusability and maintainability by separating the validation algorithm from the credit card class itself.

We use the Factory Method pattern to identify the type of credit card and create instances of it dynamically. With this pattern, concrete factory subclasses are responsible for creating the appropriate credit card subclass based on the type of card identified.

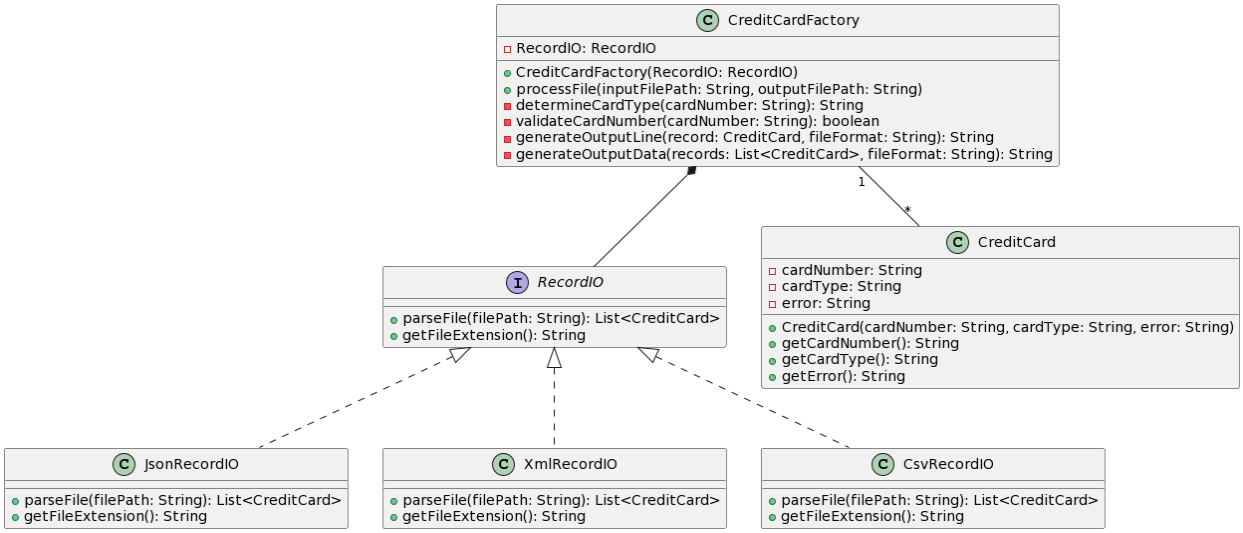
By using the Factory Method pattern, we can ensure that the client code and specific credit card subclasses are loosely coupled. This makes it easier to add new credit card types by creating a new subclass and corresponding factory method. The design is also extendable, enabling the system to accommodate new credit card types without requiring extensive changes to the existing codebase.

Overall, the combination of Factory Method pattern for dynamic creation and identification of card types and designing for extensibility helps address the secondary problems of identifying card types, creating appropriate credit card instances, and accommodating new credit card types in the future.

Part 2:
To extend the design to parse different input file formats (json, xml, csv) and detect the type of credit card, we can use the following approach:

1. File Identification Classes: Create separate classes for each file format (e.g., JSONRecordIO, XMLRecordIO, CSVRecordIO) responsible for parsing the input files and extracting credit card records.
2. Credit Card Detector: Implement a CreditCardDetector class that takes the parsed credit card records and identifies the type of credit card based on the card number using the same validation logic from Part 1.
3. Output Record Classes: Output Record classes for each file format (e.g., JSONRecordIO, XMLRecordIO, CSVRecordIO) responsible for generating the output files with the detected card type and any validation errors.

Extension of the Part 1 for RecordIO Factory.

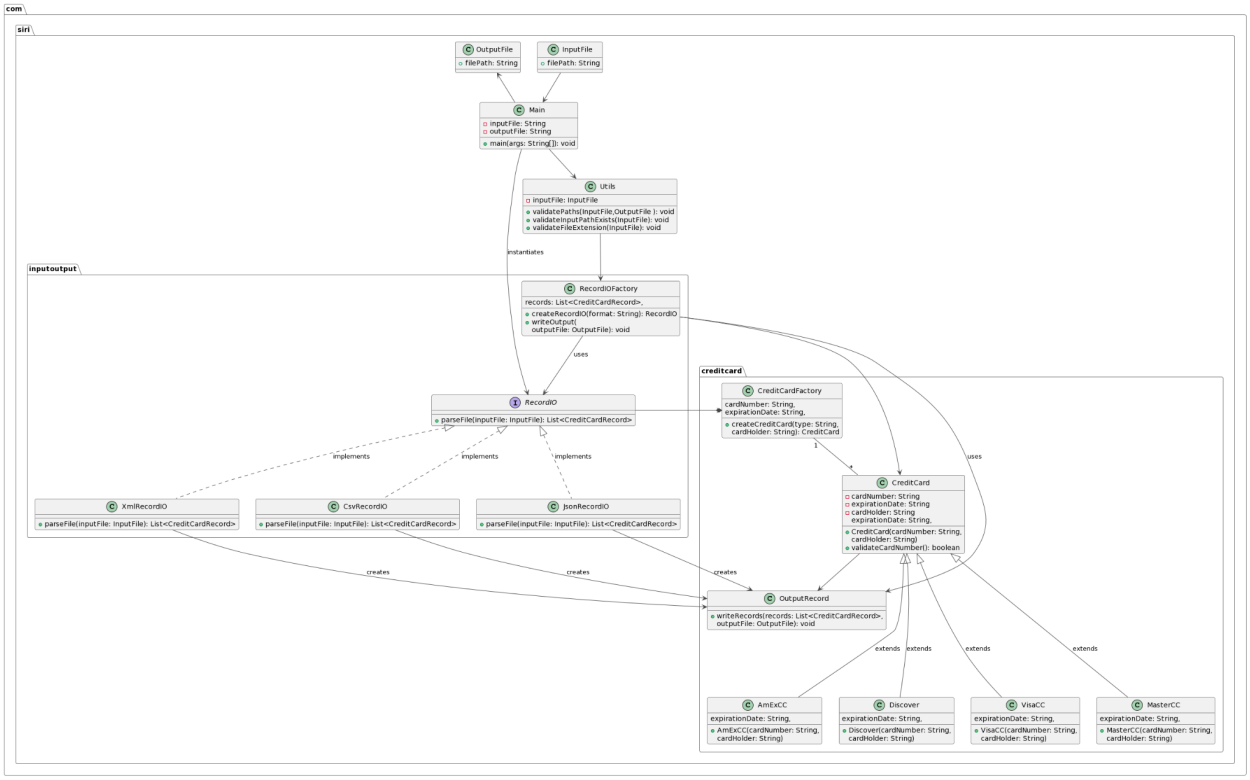


The CreditCardDetector class checks a list of CreditCardRecord objects from the file parser and uses the same validation process as in Part 1 to determine the card type for each record. It then creates a list of CreditCardInfo objects that includes the card number, card type (if valid), and any errors (if invalid).

The Output records classes use the list of CreditCardInfo objects to create an output file in the corresponding format (json, xml, csv) that includes the card number, card type, and error information (if applicable).

Our design enables seamless integration of new file formats and their corresponding RecordIO and output record in the future. Each file format has its own parser and output generator class, making the system more modular and easily extensible. Moreover, the RecordIOFactory class can use the validation logic from Part 1 to identify the card type of different file formats.

Class Diagram for the Design of the Credit Card Processor:



The UML class diagram depicts the structure of the design for the given problem. Here is an explanation of the classes and their relationships:

- Main: This class represents the main entry point of the application. It has the responsibility to handle command-line arguments, such as input and output file paths, and execute the necessary logic.
- Utils: This class contains utility methods that validate the input and output file paths. It ensures that the paths exist and have the correct file extensions.
- InputFile and OutputFile: These classes represent the input and output file paths, respectively.
- RecordIOFactory: This class is responsible for creating the appropriate RecordIO object based on the file format specified. It utilizes the Factory Method pattern to create the desired object.
- RecordIO: This interface defines the contract for parsing files of different formats and returning a list of CreditCardRecord objects.
- XmlRecordIO, CsvRecordIO, JsonRecordIO: These classes implement the RecordIO interface and provide specific implementations for parsing XML, CSV, and JSON files, respectively.
- CreditCard: This class represents a generic credit card and contains common properties like cardNumber, expirationDate, and cardHolder. It also has a method validateCardNumber() that verifies the validity of the card number.

- VisaCC, MasterCC, AmExCC, Discover: These classes are subclasses of CreditCard and represent specific types of credit cards. They inherit the properties and methods from the base class and may provide additional functionality specific to their card type.
- OutputRecord: This class is responsible for writing a list of CreditCardRecord objects to the output file.
- CreditCardFactory: This class implements the Factory Method pattern and is responsible for creating instances of the appropriate credit card subclasses based on the card type specified.

The diagram also shows the relationships between the classes, such as associations, generalizations, and dependencies.
The design is intended to be flexible and easy to expand using design patterns such as Factory Method, inheritance, and interface implementations. It divides tasks, such as handling input/output, validating credit cards, and creating objects, into separate classes, which helps ensure the code is modular and easy to maintain.