CMPE202 Credit Card Problem

Name: ANANTH UPADHYA

SJSU ID :- 015234726

Part-1:

1. Describe what is the primary problem you try to solve?

   * The main issue is figuring out how to build a workflow that uses design patterns to recognize common communication patterns among objects so that our app can correctly identify credit card types or classify them as invalid.

   Solution :- Strategy design pattern and Chain of Responsibility pattern are good fit to this problem

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

   The secondary issue was how to deal with object creation mechanisms, specifically how to construct objects in a way that would allow the credit card to be validated. The most basic type of object development may lead to design issues or add to the design's complexity.

   Solution :- Factory Design Pattern would help in solving this issue

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

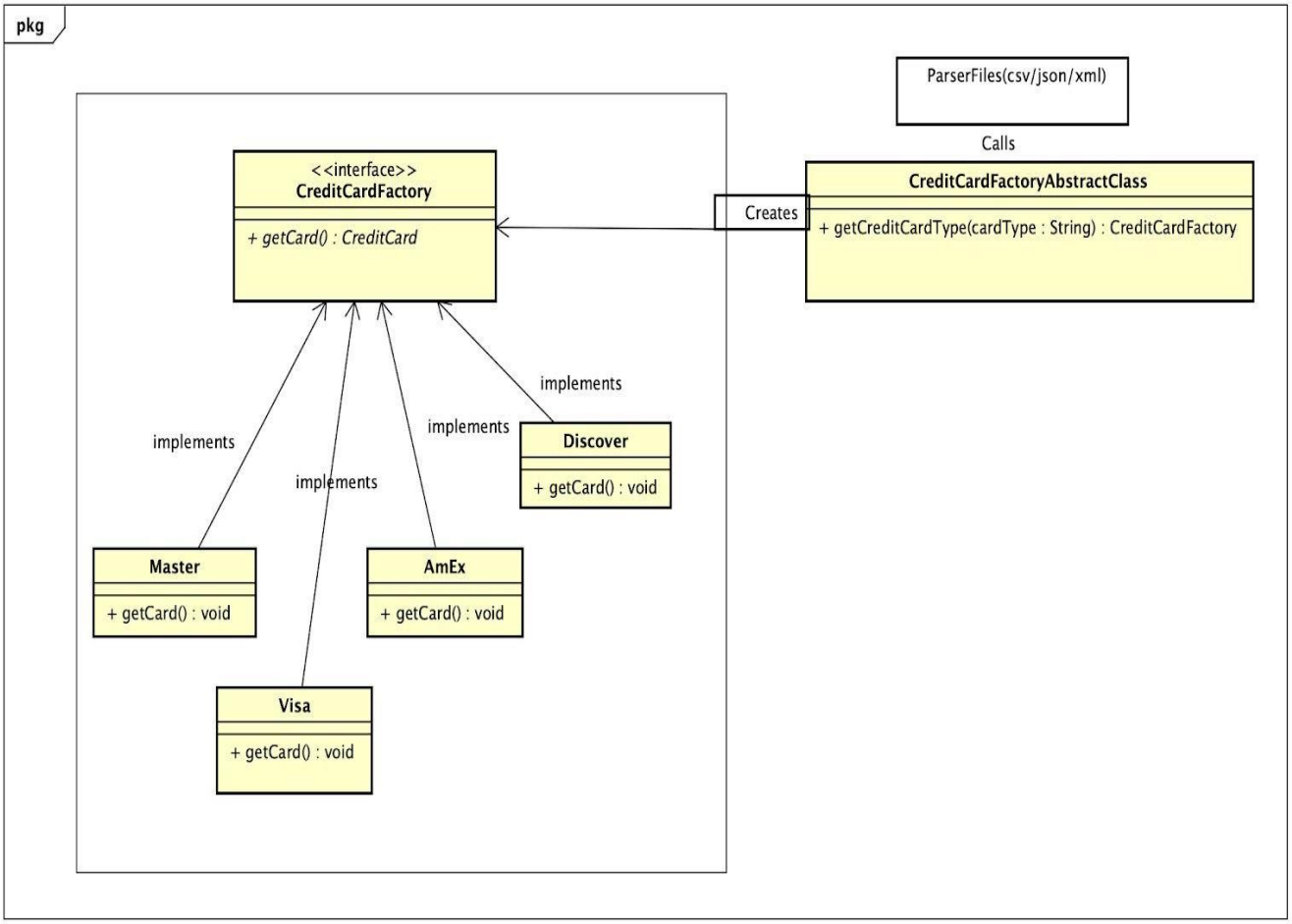
   Design patterns: Factory design pattern and Chain of responsibility design pattern.

   1. Factory design pattern:

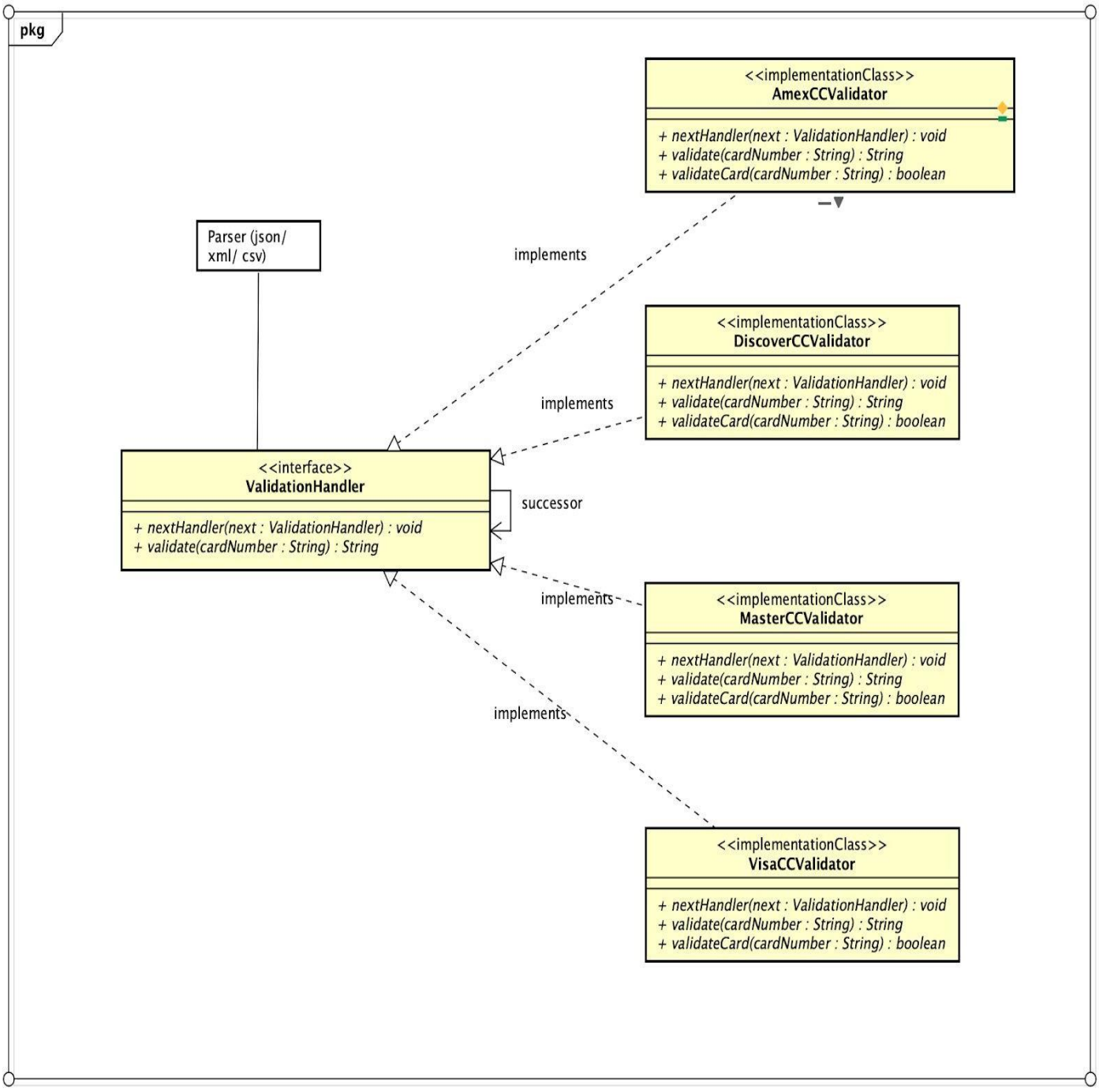
      * After parsing the csv file, we get the credit card number from each entry of the csvfile, now we have to validate that number for each of the four credit card categories and if it matches with any of the given regular expression patterns for each of the four categories, we have to someway communicate to the parser that this credit card belong to one of the following category.

      * For this, factory pattern can be used, using an abstract class(CreditCardFactoryAbstractClass) it abstracts the credit card classes, we create an object of these credit cards using a factory(CreditCardFactory).

      * Based on the string we return from the parser, it creates a suitable object of corresponding Credit Card



2. Chain of responsibility design pattern:
   - Following the parsing of the CSV file, a procedure should be in place to verify the credit card number in order to determine which of the four categories it belongs to.
   - The chain of responsibility pattern can be used to verify if a specific card number belongs to the current category first, and then move on to the next category if it does not. The ValidationHandler interface controls the entire process by passing the credit card number to the four validators: AmExCCValidator, MasterCCValidator, VisaCCValidator, and DiscoverCCValidator.
   - All the four validators implement the interface and pass – on the card number to the next when it does not match with itself



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

Consequences of Factory Design Pattern:

Pros:

1) It gives scope to add new credit card types in future. The creator is not tightly coupled to any ConcreteProduct.
2) It decouples the business logic of creation of a card creation classes from the actuallogic of finding the credit card type in the parsers
3) Provides abstraction between implementation and client classes through inheritance.
4) Allows one to change the design of our application more readily. Makes our code more robust, less coupled and easy to extend.

Cons:

1) Makes code more difficult to read as all of your code is behind an abstraction that may in turn hide abstractions.
2) Parsers might have to subclass the CreditCardFactory interface just to create aparticular credit card Object. The client now must deal with another point of evolution.
3) Too many objects often can decrease performance.

Consequences of Chain of responsibility Design Pattern:

Pros:

1) Simplifies our object
   - It does not have to know the chain's structure to keep direct references to its members
   - Keeps a single reference to their successor

2) It decouples the file parsers and its corresponding credit card validators
   - frees an object from knowing which other object handles a request
   - both the receiver (file parsers) and the sender(validators) have no explicit knowledge of each other
3) Gives us added flexibility in distributing responsibilities among objects. It allows you to add or remove responsibilities dynamically by changing the members or order of the chain.

Cons:

1) One drawback of this pattern is that the execution of the request is not guaranteed. It may fall off the end of the chain if no object handles it.
2) Another drawback is that it can be hard to observe and debug at runtime.

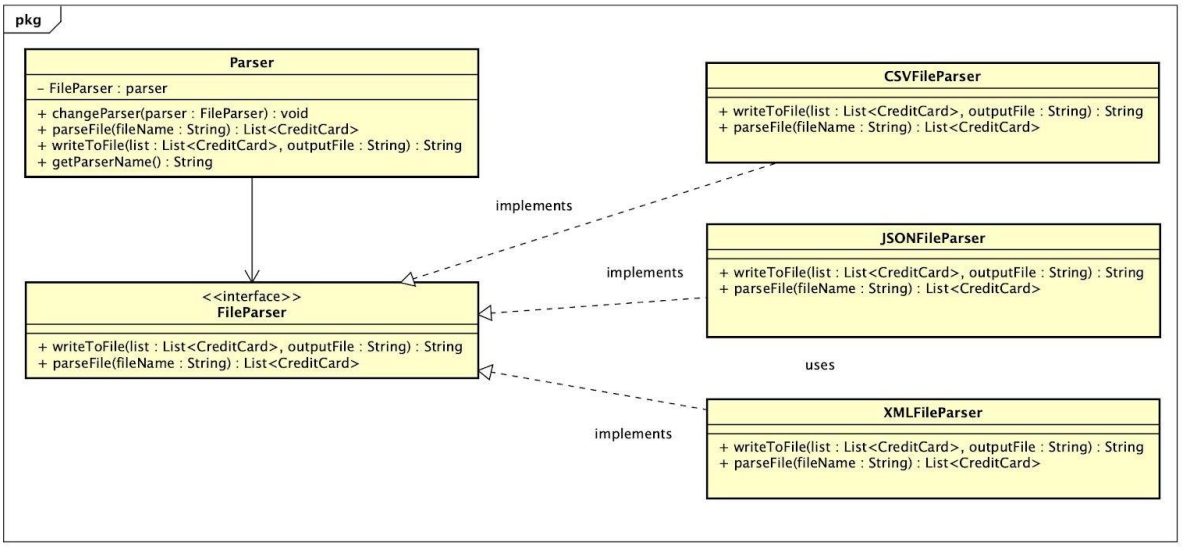
Part -2:

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. I used the Strategy design pattern to extend this file formatting to xml and json. It alsoallows us to implement this functionality to new file formats in the future.

Solution

Using this pattern, we can encapsulate interface(FileParser) details in a base class, and Implementation details in derived classes(CSVFileParser, JSONFileParser, XMLFileParser)

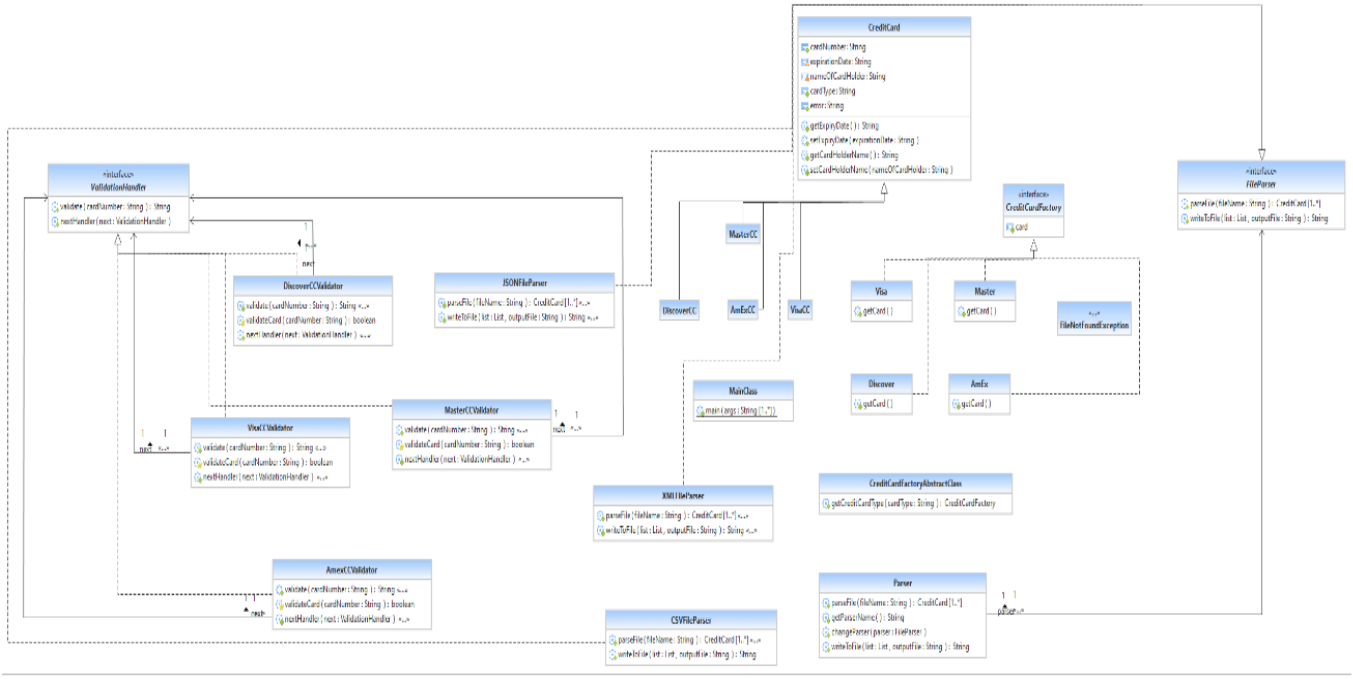


So to build the entire system which finds the credit card type, I used Strategy, Chain of Responsibility and Factory design patterns. Check below the entire system design of the creditcard type finder.

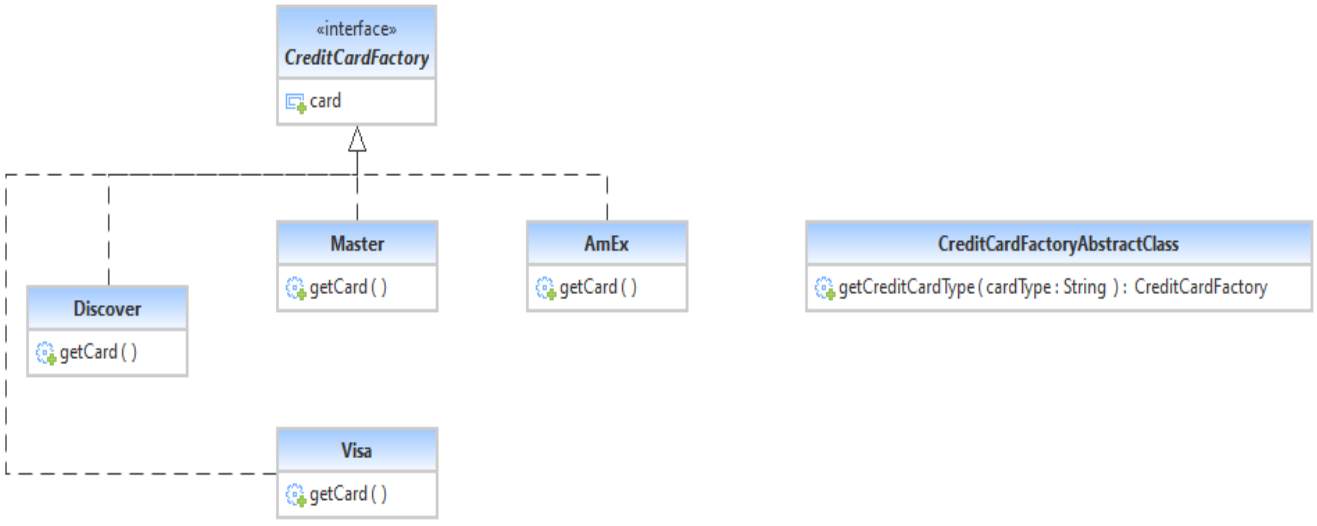
This entire design will allow to add a new credit card type/ new file format with ease without changing much of the code, just by creating classes and their corresponding validators.

This design makes an abstraction between these modules and gives flexibility to the creators.

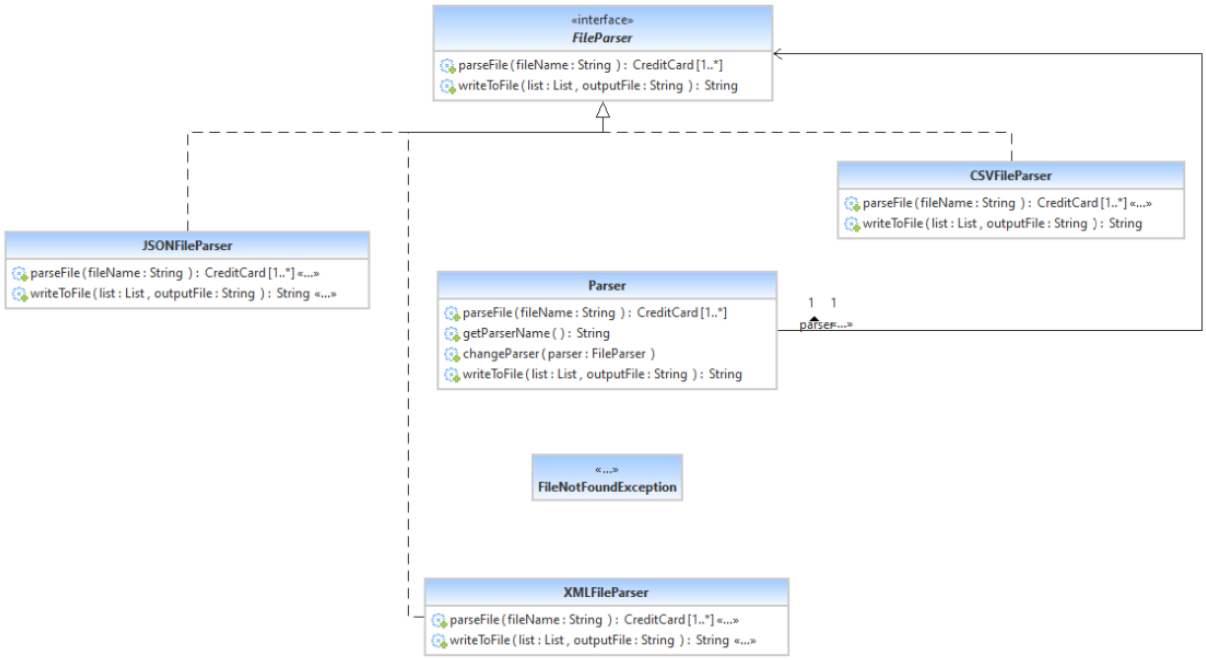
UML Diagrams



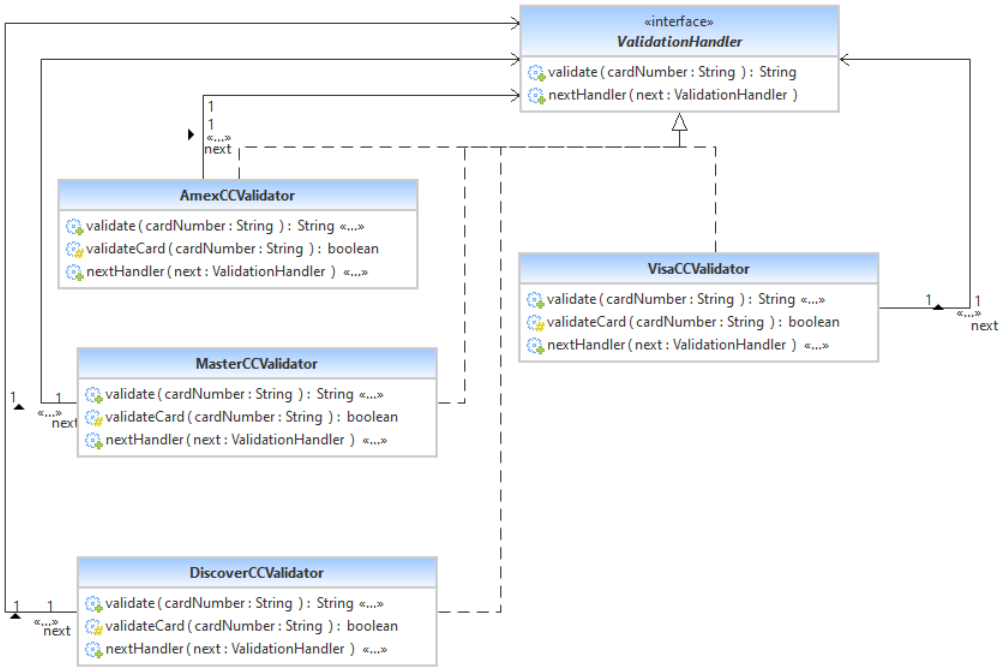
Factory UML Diagram



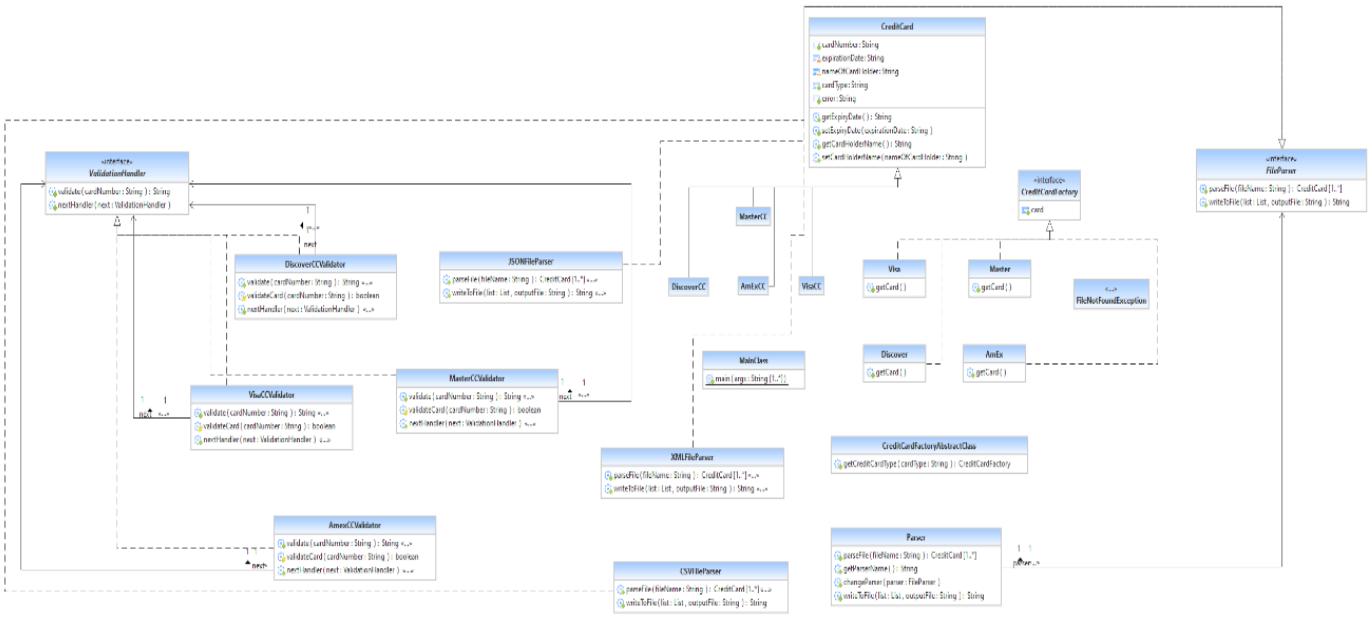
UML File Parser



UML Validator



Complete UML Diagram


CreditCard
cardNumber : String
expirationDate : String
nameOfCardHolder : String
cardType : String
error : String
getExpiryDate( ) : String
setExpiryDate(expirationDate : String)
getCardHolderName( ) : String
setCardHolderName(nameOfCardHolder : String)
«interface»
ValidationHandler
validate(cardNumber : String) : String
nextHandler(next : ValidationHandler)
«interface»
FileParser
parseFile(fileName : String) : CreditCard [1..*]
writeToFile(list : List, outputFile : String) : String
«virtual class»
CreditCardFactory
card
DiscoverCCValidator
validate(cardNumber : String) : String
validateCard(cardNumber : String) : boolean
nextHandler(next : ValidationHandler)
JSONFileParser
parseFile(fileName : String) : CreditCard [1..*]
writeToFile(list : List, outputFile : String) : String
MasterCC
DiscoverCC
AmExCC
VisaCC
Visa
getCard( )
Master
getCard( )
FileNotFoundException
MainClass
main(args : String [1..*])
Discover
getCard( )
AmEx
getCard( )
VisaCCValidator
validate(cardNumber : String) : String
validateCard(cardNumber : String) : boolean
nextHandler(next : ValidationHandler)
MasterCCValidator
validate(cardNumber : String) : String
validateCard(cardNumber : String) : boolean
nextHandler(next : ValidationHandler)
XMLFileParser
parseFile(fileName : String) : CreditCard [1..*]
writeToFile(list : List, outputFile : String) : String
CreditCardFactoryAbstractClass
getCreditCardType(cardType : String) : CreditCardFactory
AmexCCValidator
validate(cardNumber : String) : String
validateCard(cardNumber : String) : boolean
nextHandler(next : ValidationHandler)
CSVFileParser
parseFile(fileName : String) : CreditCard [1..*]
writeToFile(list : List, outputFile : String) : String
Parser
parseFile(fileName : String) : CreditCard [1..*]
getNameName( ) : String
changeParser(parser : FileParser)
writeToFile(list : List, outputFile : String) : String
next
1
1..*

Thank you