CMPE 202 - Individual Project

Submitted By - Girish Bisane
[016650348]

Part 1

Q1. Describe what is the primary problem you try to solve.

The primary problem that I am trying to solve is to check whether a card is valid or not by checking the different types of credit cards i.e.,(Master card, Visa, American Express and Discover) and using the credit card number to identify the card issuer. To find the relevant objects based on card type.

Q2. Describe what are the secondary problems you try to solve (if there are any).

The first was the need to identify the credit card type based on the credit card number and verify that the credit card number was a possible account number.

Another secondary problem was how to maintain extensibility for future credit card types that might be added later. It was important to ensure that the design of the system was flexible enough to accommodate new credit card types without requiring significant modifications to the existing code.

Finally, the introduction of the feature that allowed for the parsing of multiple input files brought in a range of expectations that needed to be handled. The system needed to be able to handle different file formats, and the input files needed to be processed efficiently without compromising the accuracy of the credit card number verification process.

Q3. Describe what design pattern(s) you use?

To solve the primary and secondary problems, we used the following design patterns:

1.Factory Method Pattern:

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

suitable for our primary problem because we need to create an instance of the appropriate credit card class based on the credit card type.

To implement the Factory Method Pattern, we can create an abstract CreditCardFactory class that provides an abstract method for creating CreditCard objects. We can then create concrete factories such as VisaCCFactory, MasterCCFactory, and AmExCCFactory that implement the CreditCardFactory interface and override the createCreditCard() method to create the appropriate CreditCard subclass.

2. Strategy Pattern:

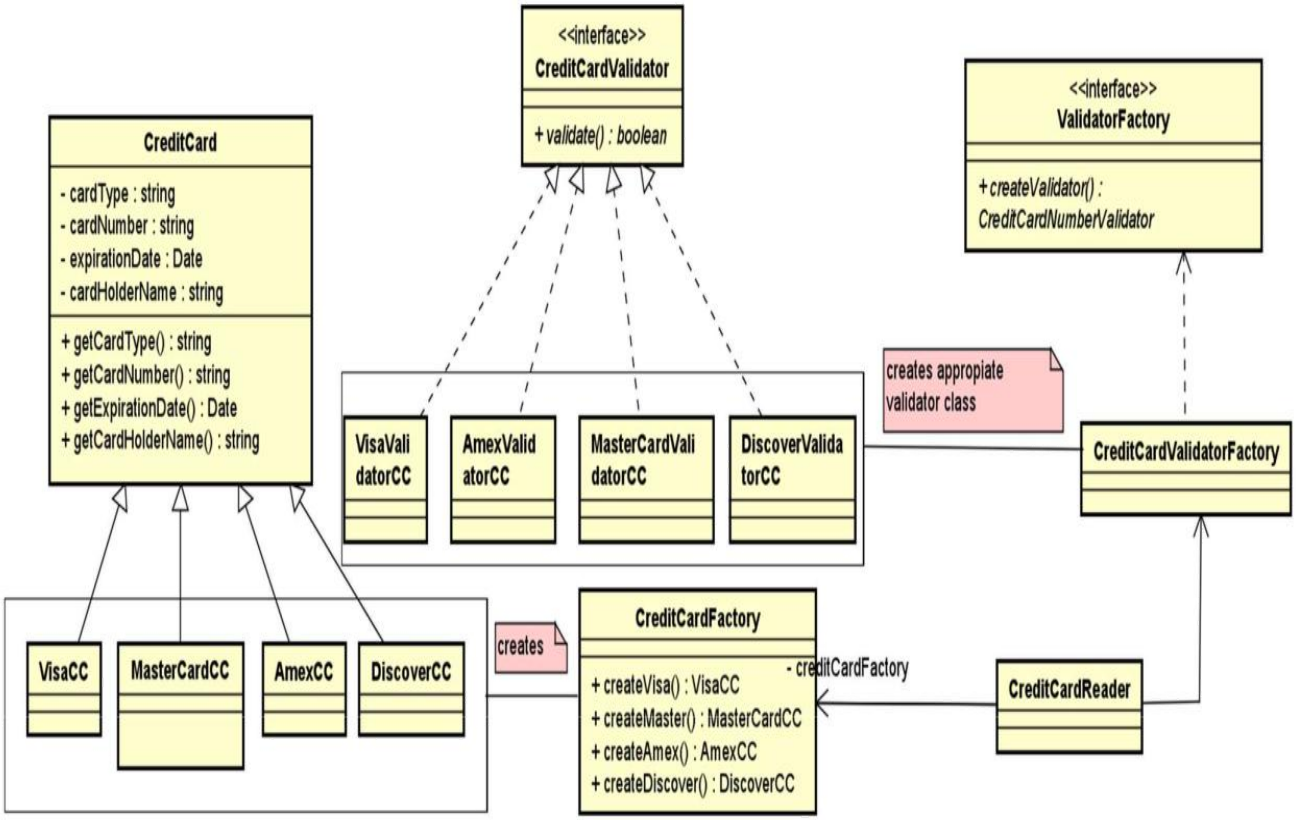
The Strategy Pattern is a behavioral pattern that defines a family of algorithms, encapsulates each one, and makes them interchangeable. This pattern is suitable for our secondary problem because we need to verify the credit card number to ensure that it is a possible account number.

To implement the Strategy Pattern, we can create an interface called Credit Card Number Valid onStrategy that provides a method for validating credit card numbers.
We can then create concrete strategies such as VisaCCNumberValida, onStrategy, MasterCCNumberValidationStrategy, and AmExCCNumberValidation onStrategy that implement the CreditCardNumberValida onStrategy interface and override the validate() method to perform the appropriate credit card number validation on algorithm.

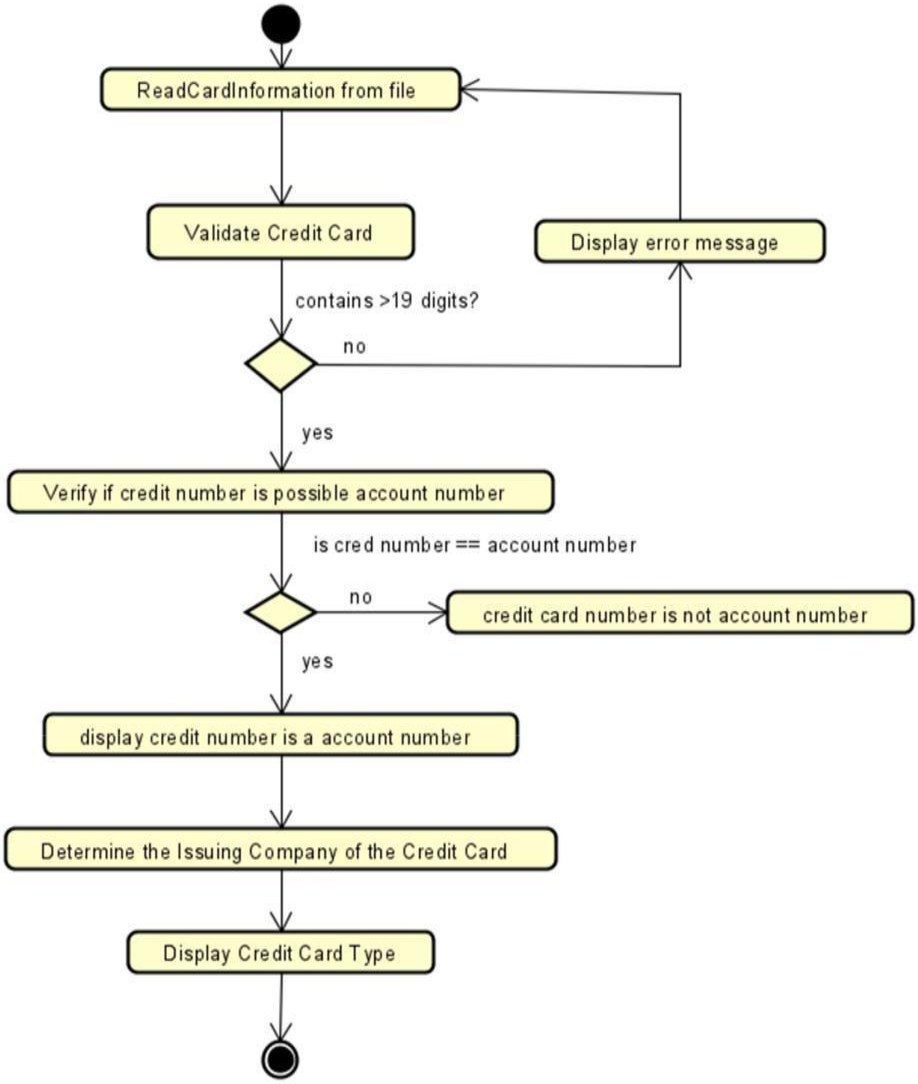
3. Iterator Pattern:

The input files (csv , xml or json) contains a list of nodes that the system need to iterate the list so to do that we will be using iterator pattern, because it will provide us a way to access objects without being concerned about the underlying representation on of the objects.

Class Diagram -



Activity Diagram -



Explanation

The CreditCard class and its subclasses (VisaCC, MasterCC, AmExCC, and DiscoverCC) all have private fields for cardType, cardNumber, expirationDate, and cardholderName, as well as public getters for those fields. The CreditCard class also has a public abstract method getCardType() for getting the type of the credit card.

The CreditCardNumberValidator interface defines the validate() method for validating credit card numbers, and is implemented by the concrete Validator classes.

The CreditCardFactory interface defines the create methods for creating the different types of CreditCard objects, with VisaCCFactory, MasterCCFactory, AmExCCFactory, and DiscoverFactory as its concrete implementation, each creating the appropriate CreditCard subclass.

The ValidatorFactory interface defines the createValidator() method for creating Validator objects, with its concrete implementation on CreditCardValidatorFactory, creating the appropriate Validator subclass.

This design pattern allows for the creation of families of related objects (credit cards and validators) without specifying their concrete classes. It also ensures that the objects are comparable with each other, as the credit card objects created by a factory will always use the validator created by the same factory .

Relationships

The relationship between the classes in the class diagram is as follows:

- The abstract class CreditCard is the parent class for all credit cards.
- The concrete classes VisaCC, MasterCC, and AmExCC are the child classes of CreditCard and inherit its a ributes and methods.
- The CreditCardFactory class is a factory class that creates instances of the appropriate credit card subclass based on the credit card number.

The CreditCardFactory class uses the Singleton pattern to ensure that only one instance of the factory class exists throughout the application.
The CreditCardValidator class contains a static method that checks whether a given credit card number is valid or not. This method is used by the CreditCardFactory class to verify the credit card number before creating an instance of the appropriate credit card subclass.

How the patterns are implemented

The Strategy pattern is not explicitly used in the updated class diagram. However, the CreditCardValidator class can be considered an example of the Strategy pattern.

In the class diagram, the CreditCardValidator class is a separate class that contains a static method for validating credit card numbers. This class can be seen as encapsulating an algorithm for validating credit card numbers. By using a separate class for this algorithm, we can easily swap out different validation on algorithms without affecting the rest of the application.

This is similar to the Strategy pattern, which defines a family of interchangeable algorithms and encapsulates each one as a separate object. In the Strategy pattern, a context object can use different strategies interchangeably by delegating the work to the strategy object. Similarly, in our case, the CreditCardReader class delegates the work of credit card validation on to the CreditCardValidator class.

Therefore, while the Strategy pattern is not explicitly used in the updated class diagram, the CreditCardValidator class can be seen as an example of the Strategy pattern.

Q4. Describe the consequences of using this/these pattern(s).

The consequences of using the Factory Method Pattern are that it provides a way to decouple the creation of objects from their use and allows for flexibility in the creation of objects. The Factory Method Pattern also makes the code easier to extend by adding new subclasses without modifying existing code. However, it can increase the complexity of the code by adding more classes.

The consequences of using the Strategy Pattern are that it allows for flexibility in selecting algorithms at run time and promotes code reuse by encapsulating algorithms in separate classes. The Strategy Pattern also makes the code easier to extend by adding new validation on strategies without modifying existing code. However, it can increase the complexity of the code by adding more classes and interfaces.

Overall, using the Factory Method and Strategy Patterns can provide a flexible and extensible solution to the primary and secondary problems described above.

PART 2 -

Q1. How will incorporate new file types in the system?

This project was extended to parse different input file formats (json, xml, csv). Strategy pattern was used to decide the input file format and therefore which method was going to be used to parse the file. However, since each file contains lines of credit card records, iterator pattern was incorporated with strategy pattern, so that each line of records was read and processed. Below is the class diagram that represents the final classes and methods that will be implemented in the system.