# CMPE 202 - Individual Project

**Aditya Pandey - 017461873**

**Git Repository Link**: Github

**The Primary Problem**

The primary issue I'm addressing is identifying the type of card represented by a record that includes a credit card number, expiration date, and cardholder's name. The main challenge is to read these records, validate the credit card number, determine who issued the card, and create an object of the appropriate credit card class.

## Secondary Problems

**Allowing various file extensions:** The system should be able to handle different input file formats (CSV, JSON, XML) and also accept new formats in the future.

**Credit Card Validation:** Each credit card number must be validated to match if it's a legitimate number and identify the card issuer company.

**Design Pattern:**

**1)Strategy Pattern**

In this implementation, I employed the Strategy Pattern. The method isValid() is established in the foundational class CreditCard, with the responsibility for its execution passed on to the specific subclasses (AmEx, Discover, Master, Visa).

This approach empowers each type of credit card to implement its unique validation strategy, thereby facilitating the expansion of the system to incorporate new card types without necessitating changes to the pre-existing code. Additionally, the InvalidCreditCard class adheres to this pattern as well, offering a distinct strategy to manage invalid credit cards.

**Consequences of the Strategy method**

- As the variety of credit card types expands, there could be an increase in the number of subclasses (like Amex, Visa etc), which might complicate the management of the system.
- Employing the Strategy Pattern can add complexity, particularly when handling a multitude of interchangeable algorithms or strategies.
- Dynamically changing between various strategies during runtime could lead to a minor increase in runtime overhead.

**Secondary Problem**

The secondary problem handled here involves creating the required objects for accommodating different types of input files (e.g., CSV, XML, or JSON).

**2) Factory Method**

The design pattern I implemented is the Factory Method Pattern, as demonstrated in my CreditCardFactory class. This class is responsible for the creation of CreditCard object instances. It utilizes a method that receives inputs such as cardNumber, expirationDate, and cardHolderName and determines which specific CreditCard subclass to create based on certain criteria. This method employs conditional checks on the cardNumber to identify the credit card type. Depending on these checks, it either instantiates and returns an object of a relevant subclass (such as Visa, Master, etc.) or returns null if the card number does not align with any recognized patterns. In this setup, the client code acquires a CreditCard instance through the factory method (getCreditCard), with the actual class of the resulting object varying based on the criteria defined within the factory method.

**Consequences of the Factory method**

- The complexity of the system might escalate with the introduction of new types of credit cards or the addition of further conditions to the factory method.
- The direct instantiation of specific classes (Master, Visa etc) results in a close interdependence between the Credi class and these particular implementations.
- There's a potential breach of the open-closed principle should modifications become necessary in the current factory method logic to integrate new card types.