Primary problem

- Primary problem is to validate if the given card number is valid and determine the type of the card. (i.e, Visa, Master, Amex..)
- **Approach**
    - Read each line from .csv
    - Identify and validate the card number using the rules described in the problem.

Secondary problems

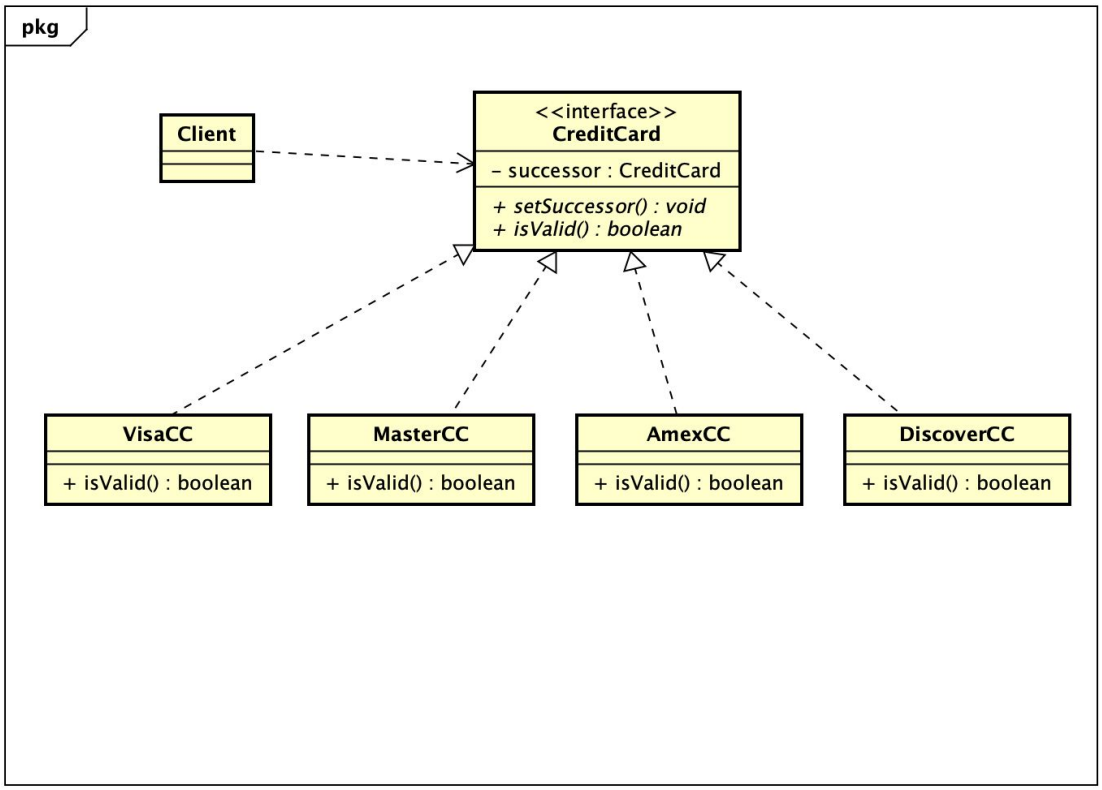
- Use appropriate design patterns to design the solution in such a way that the code can be easily extended to other card types and different input file formats
- This problem can be tackled by choosing the right creational and behavioral patterns.

Design patterns and consequences

Creational Patterns:

1. **Chain of Responsibility** can be used to include the custom logic to validate the card details in its own class. This helps in isolating the responsibilities.
   client can be configured to call one concrete subclass which then propagates the call through the chain if needed. This gives the multiple credit card subclass objects to handle the request

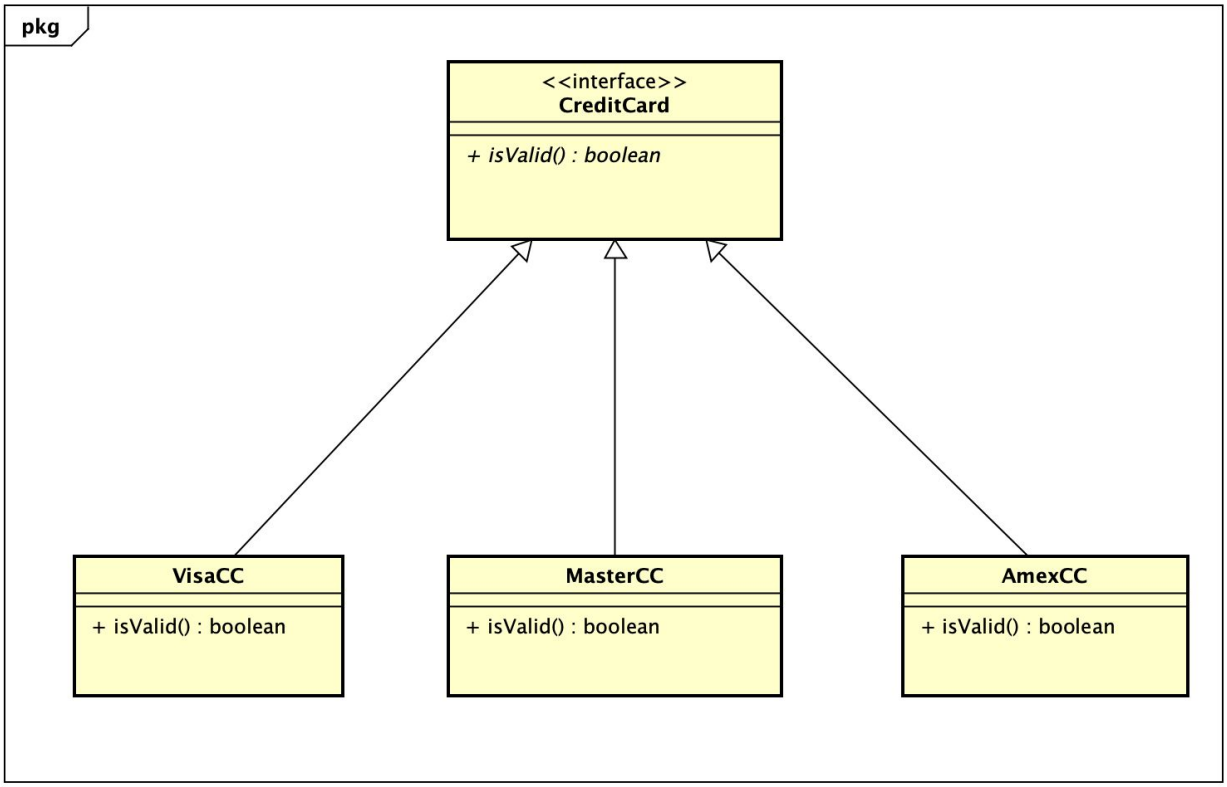
   Consequences:
    - Helps in reducing coupling and simplifies interconnection between objects.
    - Provides flexible design in assigning object responsibilities in their own classes.
    - Main disadvantage of this pattern occurs when any of the chained models is not able to handle the request.



2. **Factory method** pattern can be used to get the different concrete FileFactory objects for CsvFactory, JsonFactory, etc dynamically based on the input file extension type. This allows classes to defer instantiation to its sub classes. We might also need to use the Abstract Factory pattern in addition to Factory method if there is a need to use composition to pass factory instances into Credit card subclasses.

   Consequences:

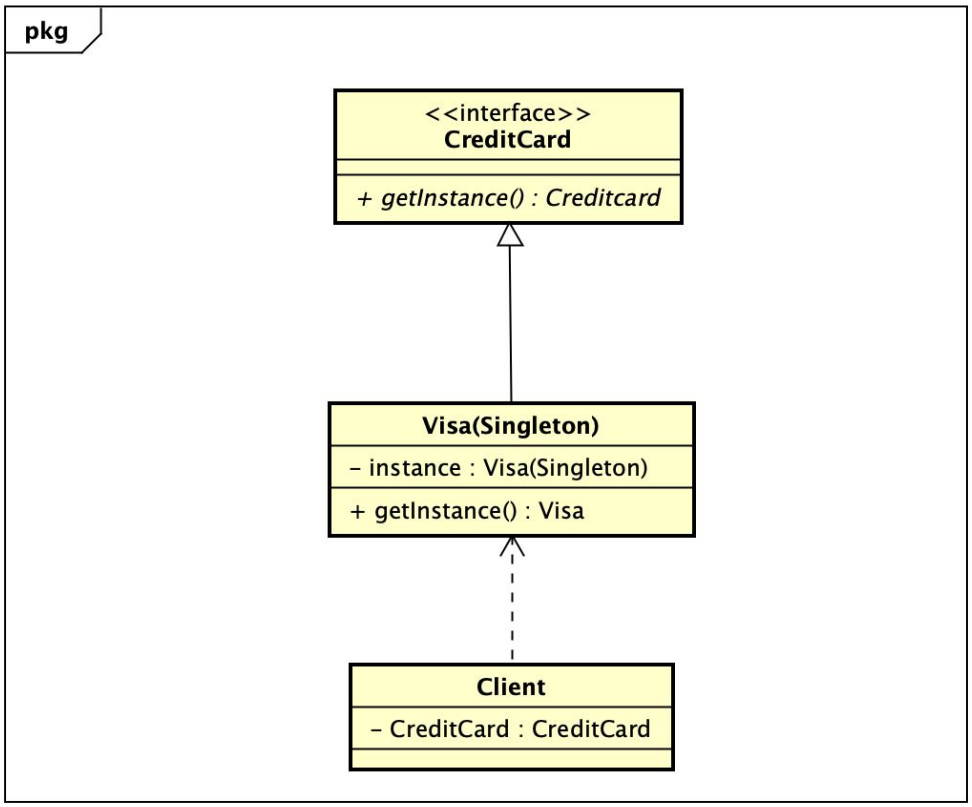
   - Helps in eliminating the need to bind concrete classes into the users of the class, as the upstream writes code for interfaces.
   - The major disadvantage of this pattern is that every product class has to either implement or subclass the creator class.



3. **Builder pattern** can be used to set the fields using chaining for the credit card object, if additional details need to be set.
   Consequences:
   - Main advantage of using a builder pattern is that parameters can be passed in any order the user wants which makes the parameters exchangeable. So this gives finer control of object creation.
   - It gives the flexibility to change the internal implementation and how the objects parameters are assembled.
   - It isolates and gives clear separation of code creation and representation.

4. **Optionally, Singleton pattern** can also be used if there is no need to create multiple instances of credit card subclasses. For example, if a credit card subclass has only one method to validate the credit card type, then creating instances multiple times is not necessary, however we can call the method on the same instance to validate the card number.
   Consequences:

- Has ability to strictly control its instance and can control clients requesting the instance.
- Better design solution to access the instance than global variables.
- Can be modified in such a way, that multiple limited instances can be created if needed
- In a multithreaded environment, creation of a singleton pattern can be a bit expensive for the first time when the object is being created, because of the use of double checked locking.



Behavioral Patterns:

1. **Strategy Pattern** can be used to have a Credit Card Interface, which will be implemented by Credit Card type subclasses (ie., visa, master, amex etc.).
   Each of these concrete classes contains a validation method to validate if the credit card number is valid.

   Consequences:

- Strategy pattern helps in reducing the conditional statements. Having the conditional logic in their own subclass implementation helps to avoid adding the conditional statements in upstreams.
- Different implementations can be written as different strategies and gives the client to choose the implementation it needs. In our case, each credit card has a different strategy of credit card number validation. So different card types can invoke different implementations based on card type. However from clients perspective, they are just using the same method without knowing the internal implementation.