Subarna Chowdhury Soma
SJSU ID: 014549587

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

Answer: The primary focus of my solution is to read a csv file with different credit card entry and to create an instance of the appropriate credit card class if the credit card number is valid. So the main problem part has several sub problems here.

1. Validating credit card number
2. Finding the type of card from a specific record
3. Creating the appropriate objects based on card type

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

Answer: The secondary problem here is designing the solution in such a way that other credit card types can be easily added in future without hampering current application solution. So I have to develop the program making it extension friendly for different credit card types.

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

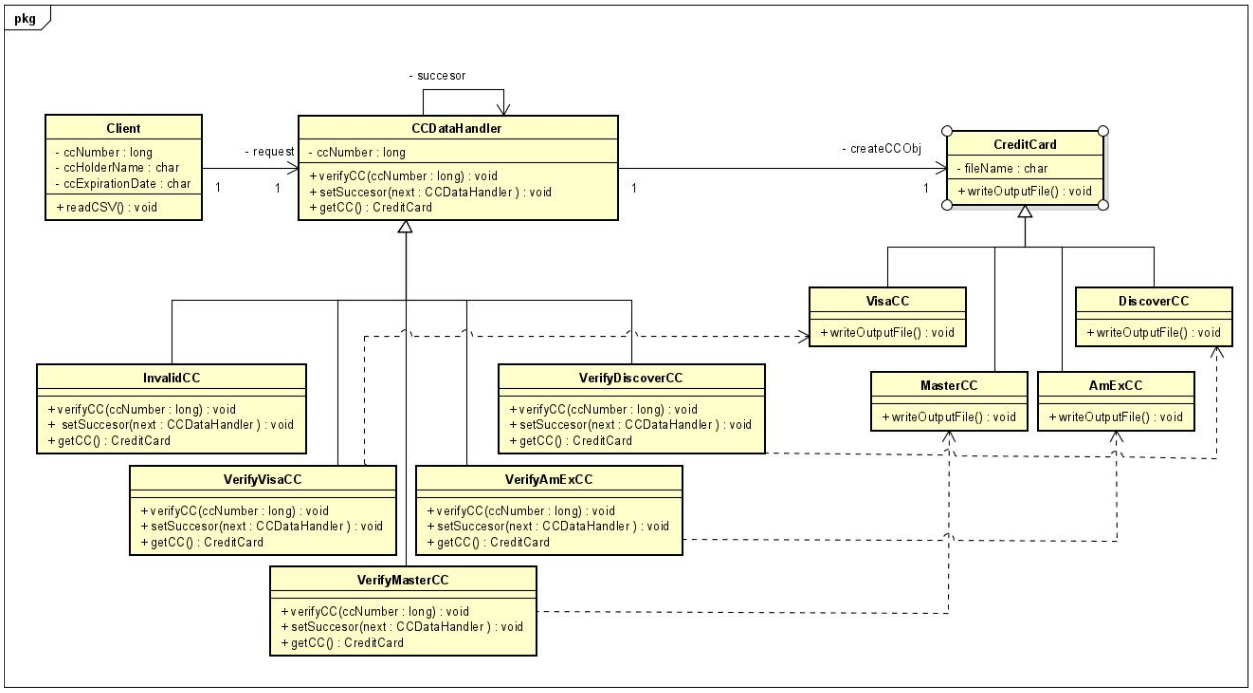
Answer:

Selected Design Patterns: I have selected two design patterns considering my primary and secondary problems.

1. Chain of Responsibility (Behavioural pattern) : To figure out what kind of a card a specific csv record is about.
2. Factory Method (Creational Pattern): To create the appropriate class object dynamically. To be specific I am going to use the parameterized Factory Method.

I shall use these two design patterns combinedly to develop the application for implementing the requirements.

Class Diagram:


pkg
- succesor
Client
- ccNumber : long
- ccHolderName : char
- ccExpirationDate : char
+ readCSV() : void
- request
1
1
CCDataHandler
- ccNumber : long
+ verifyCC(ccNumber : long) : void
+ setSuccesor(next : CCDataHandler) : void
+ getCC() : CreditCard
- createCCObj
1
1
CreditCard
- fileName : char
+ writeOutputFile() : void
VisaCC
+ writeOutputFile() : void
DiscoverCC
+ writeOutputFile() : void
MasterCC
+ writeOutputFile() : void
AmExCC
+ writeOutputFile() : void
InvalidCC
+ verifyCC(ccNumber : long) : void
+ setSuccesor(next : CCDataHandler) : void
+ getCC() : CreditCard
VerifyDiscoverCC
+ verifyCC(ccNumber : long) : void
+ setSuccesor(next : CCDataHandler) : void
+ getCC() : CreditCard
VerifyVisaCC
+ verifyCC(ccNumber : long) : void
+ setSuccesor(next : CCDataHandler) : void
+ getCC() : CreditCard
VerifyAmExCC
+ verifyCC(ccNumber : long) : void
+ setSuccesor(next : CCDataHandler) : void
+ getCC() : CreditCard
VerifyMasterCC
+ verifyCC(ccNumber : long) : void
+ setSuccesor(next : CCDataHandler) : void
+ getCC() : CreditCard

Description: The following table contains all the details of classes from class diagram:

Class Name	Operation(s)	Associated Class(es)	Class Responsibility Details
Client	readCSV()	CCDataHandler	1. Starting point of application 2. Reads the csv file 3. Sends file object to the first object of chain of responsibility, InvalidCC here, for credit number validation and type identification.

CCDataHandler	verifyCC()
	setSuccesecor()
	getCC()
		Client
		InvalidCC
		VerifyVisaCC
		VerifyMasterCC
		VerifyAmExCC
		VerifyDiscoverCC
		CreditCard
			1. An abstract class with factory/abstract method ‘getCC’
			
			2. Super class for its subclasses
			3. Plays the role of a

			Handler interface of chain of responsibility pattern 4. Plays the role of Creator of Factory Method Pattern. It has the declaration of factory method ‘getCC’ which creates the appropriate object 5. Contains declaration of operation verifyCC

InvalidCC	verifyCC()
	setSuccesecor()
	getCC()
		CCDataHandler
		InvalidCC
		VerifyVisaCC
			1. First ConcreteHandler of the chain of responsibility pattern
			
			2. Checks the validity of a credit card number by checking if the number is greater than 19 digits. Implements the logic inside verifyCC().
			
			3. If the credit card number is valid, it forwards the request to its successor

VerifyVisaCC	verifyCC()
	setSuccesecor()
	getCC()
		CCDataHandler
		InvalidCC
		VerifyMasterCC
		VisaCC
			1. Second ConcreteHandler of the chain of responsibility pattern
			
			2. Inside verifyCC(), checks whether **“First digit is a 4. Length is either 13 or 16 digits**” to identify Visa card
			
			3. If not Visa card, it forwards the request to its successor
			
			4. One of the ConcreteCreators of Factory method and

			overrides the factory method getCC() to return an instance of a ConcreteProduct-VisaCC()

VerifyMasterCC	verifyCC()
	setSuccesecor()
	getCC()
		CCDataHandler
		VerifyVisaCC
		VerifyAmExCC
		MasterCC
			1. Third ConcreteHandler of the chain of responsibility pattern
			
			2. Inside verifyCC(), checks whether **“ First digit is a 5, second digit is in range 1 through 5 inclusive. Only valid length of number is 16 digits”** to identify Master card
			
			3. If not Master card, it forwards the request to its successor
			
			4. One of the ConcreteCreators of Factory method and overrides the factory method getCC() to return an instance of a ConcreteProduct-MasterCC()

VerifyAmExCC	verifyCC()
	setSuccesecor()
	getCC()
		CCDataHandler
		VerifyMasterCC
		VerifyDiscoverCC
		AmExCC
			1. Fourth ConcreteHandler of the chain of responsibility pattern
			
			2. Inside verifyCC(), checks whether **“ First digit is a 3 and second digit a 4 or 7. Length is 15 digits”** to identify AmEx card
			
			3. If not AmEx card, it forwards the request to its successor
			
			4. One of the ConcreteCreators of Factory method and

			overrides the factory method getCC() to return an instance of a ConcreteProduct-AmExCC()

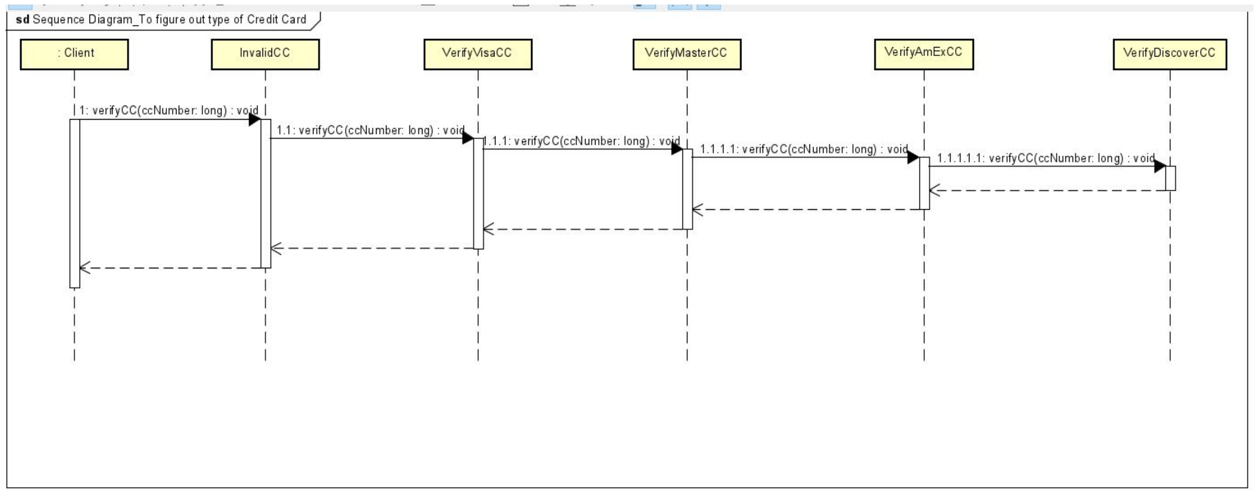
VerifyDiscoverCC	verifyCC()
	setSuccesecor()
	getCC()
		CCDataHandler
		VerifyAmExCC
		DiscoverCC
			1. Last ConcreteHandler of the chain of responsibility pattern
			
			2. Inside verifyCC(), checks whether **" First four digits are 6011. Length is 16 digits"** to identify Discover card
			
			3. If not Discover card, request ends here with a message of 'new card type'
			
			4. One of the ConcreteCreators of Factory method and overrides the factory method getCC() to return an instance of a ConcreteProduct-DiscoverCC()

CreditCard	writeOutputFile()	CCDataHandler VisaCC MasterCC AmExCC ObserverCC	1. Plays the role of Product interface in Factory method pattern 2. Super class for the ConcreteProduct classes and contains operation writeOutputFile() to write to output file
VisaCC	writeOutputFile()	CCDataHandler VerifyVisaCC	1. One of the ConcreteProducts in Factory method pattern 2. Implements operation writeOutputFile() to write to output file
MasterCC	writeOutputFile()	CCDataHandler	1. One of the

		VerifyMasterCC	ConcreteProducts in Factory method pattern 2. Implements operation writeOutputFile() to write to output file
AmExCC	writeOutputFile()	CCDataHandler VerifyAmExCC	1. One of the ConcreteProducts in Factory method pattern 2. Implements operation writeOutputFile() to write to output file
ObserverCC	writeOutputFile()	CCDataHandler VerifyObserverCC	1. One of the ConcreteProducts in Factory method pattern 2. Implements operation writeOutputFile() to write to output file

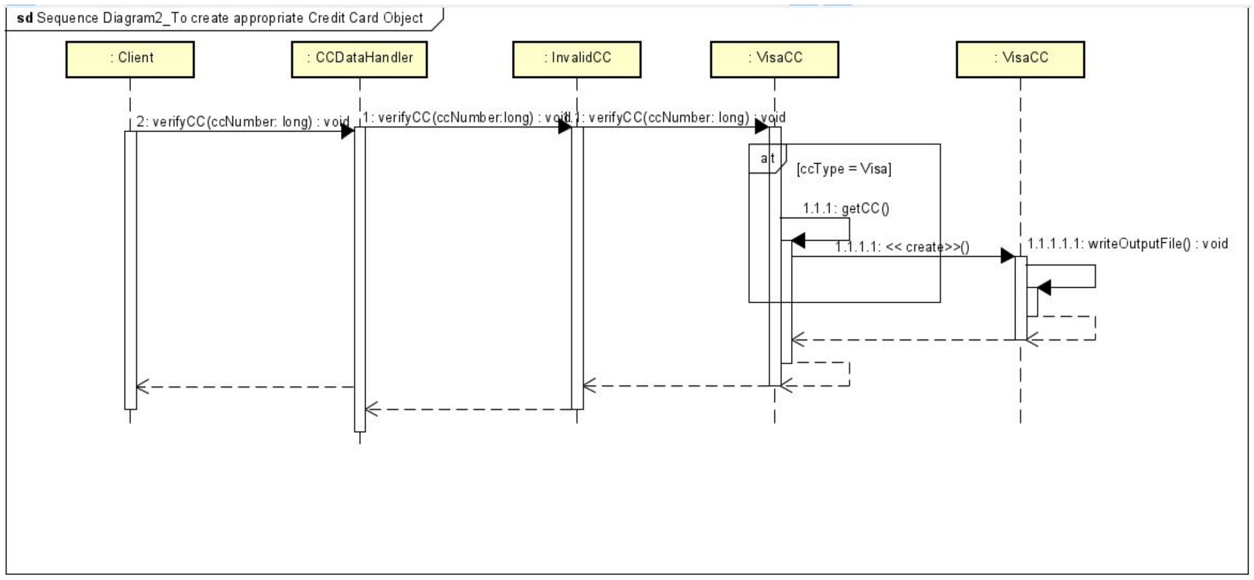
Sequence Diagram:

1. Sequence of actions to figure out what kind of credit card a specific record is about:



sd Sequence Diagram_To figure out type of Credit Card
: Client
InvalidCC
VerifyVisaCC
VerifyMasterCC
VerifyAmExCC
VerifyDiscoverCC
1: verifyCC(ccNumber: long) : void
1.1: verifyCC(ccNumber: long) : void
1.1.1: verifyCC(ccNumber: long) : void
1.1.1.1: verifyCC(ccNumber: long) : void
1.1.1.1.1: verifyCC(ccNumber: long) : void

2. Sequence of actions to create the appropriate credit card objects: for simplicity showing the case when credit card number is valid and card type is ‘Visa’


sd Sequence Diagram2_To create appropriate Credit Card Object
: Client
: CCDataHandler
: InvalidCC
: VisaCC
: VisaCC
2: verifyCC(ccNumber: long) : void
1: verifyCC(ccNumber:long) : void
1.1: verifyCC(ccNumber: long) : void
alt
[ccType = Visa]
1.1.1: getCC()
1.1.1.1: << create>>()
1.1.1.1.1: writeOutputFile() : void

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

<u>**Chain of Responsibility**</u>**:**

Intent:

The intent of this design pattern is to decouple a request from its receiver by providing a chance to multiple objects to handle a request. Each processing object is connected in a chain and responsible for handling a certain type of task. When the processing is done, it forwards the command to the next processor in the chain.

Applicability: In the following scenarios-

- ❏ Multiple objects can process a request and the processor does not have to be a specific object
- ❏ When a request is needed to be send to one of the several objects without knowing or specifying a receiver explicitly
- ❏ When multiple objects, determined at runtime, are required to handle a request
- ❏ When we have to loosely couple the request sender and the receiver
- ❏ When we don’t want to handler explicitly.

- Like this application, where I am using to identify the type of credit card and validate the credit card number also. Here the credit card number will be sent to the first subclass, say ‘InvalidCC”, which will deal with invalid credit number number. If the number is valid, it will be passed to the next object to determine the credit card type.

Consequence:

- This design pattern simplifies object interconnection and encapsulates the processing request inside a pipeline. Instead of using complex and clumsy multiple ‘if’ conditions and different actions, it gives a simpler and well structured code for sender and receiver.
- Sender only keeps a single reference of the base class and passes the request to the first object in the chain. Each object in the chain keeps a reference of its successor and passes the request to the next one if the request is not handled by it.
- Chain of responsibility design pattern ensures encapsulation and loose coupling but it comes with the trade-off of having multiple implementation classes and some maintenance problems if most of the request handling codes are the same in all the extended classes.

Factory Method:

Intent:

The intent of the factory method design pattern is to allow the sub classes to decide which object to create which means subclasses are responsible to create the instance of the class. It is also known as ‘virtual constructor’.

Applicability: In the following scenarios-

- When a class doesn’t know the class of object it has to create ahead of time
- When a class wants its sub-classes to decide the objects to be created
- When a class defer responsibility to sub class to create an object and localize the knowledge
- Like this application, where I am using this design pattern to create the appropriate object of credit card class. As I am using the ‘parameterized factory method’ here, so the ‘CCFactory’ class from the above class diagram will play the role of ‘creator’ here. A creator class can call the factory method to decide the appropriate credit card object creation

Consequence:

- This design pattern allows sub classes to decide which class to instantiate
- Factory method patterns makes it possible to remove all the knowledge about the details of subclass from the client class

- This design pattern makes the program perform and decide dynamically at run time. New concrete classes can be added without hampering the existing problem and recompiling the existing code.
- Factory methods can be used in applications where parallel class hierarchies are required.
- As the factory method depends on subclasses and therefore inheritance, so the all the usual trade-offs of choosing inheritance are also applicable here
- The benefits of using the factory method reduces if the number of concrete classes increases. Because for every concrete class there will be one concrete creator class also. The ‘parameterized factory method’ removes this downside.