Credit Card Validation 1.0 Component Specification

1. Design

The Credit Card Validation Component provides the algorithms to validate the accuracy and validity of a credit card number. However, this component only validates formatting and does not guarantee that the bank will authorize the credit card number. The algorithm provides a simple check digit calculation and pattern match algorithm to verify the number is formatted properly.

1.1 Approach

When doing the research on credit card validation, it quickly becomes apparent that this component needs to provide the framework to define, as easily and as quickly as possible, new credit card type validations (enRoute and/or Australian Bankcard to name a few) and new validation algorithms (to enforce specific formatting or to extend the component to actually communicate to a validation server).

To address creating new credit card types, this design defines an interface (CreditCardValidator) that credit card validation types must adhere to and then provides an abstract class (AbstractCreditCardValidator) that implements, not only the required functionality as defined by the interface, but also provides other handy features to manipulate credit card types (like combining them into a ‘mega’ type using the ‘or’ operation). The application can even define and/or assemble new credit card validations at runtime using the CustomValidator class. All of these validators can be used standalone or can be added to a registry (using an application specific identifier or a standard default identifier) for later use by the application. A singleton instance of the registry has been included but is not required for use. An enumeration pattern is NOT used for the credit card types because the application will likely have it’s own implementations and unique identifiers. Please note: see Section 5 below for an overview of the included types.

To address creating new validation algorithms, this design defines an interface (ValidationAlgorithm) the validation algorithms must adhere to and provides an abstract class (AbstractValidationAlgorithm) that defines operators in order to put an algorithm together in an expressive way. The component defines most of the basic validators (length, digits, LUHN and prefix) and is easily expandable by either the application or by future versions of this component.

To address grouping of related algorithms, this design provides an ‘or’ relationship to group the various credit card validators into a singular validator. Note: only an ‘or’ is required since a grouping, by definition, will match **any** one validator. Let’s provide and example of when a customer will group the MasterCard, Visa and Discover card validators into a singular validator:

```
CreditCardValidator acceptedCards = new MasterCardValidator().or(new
VisaCardValidator()).or(new DiscoverCardValidator());
```

Then the accepted cards can be registered in the registry for use by the application:

```
CreditCardValidatorRegistry.getInstance().addCreditCardValidator(“accepte
d”, acceptedCards);
```

The best way to demonstrate how this component solves the above two issues easily – lets define a credit card not included (enRoute) and define some very weird extreme dummy card:

The enRoute card (prefix 2014 or prefix 2149 with a length of 15 – all digits, LUHN) :
```
ValidationAlgorithm enRouteValidation =
       new ValidationLength(15).and(new ValidationPrefix(2014).or(new
       ValidationPrefix("2149")).and(new
       ValidationFormatAllDigits()).and(new ValidationLUHN());

CreditCardValidator enRouteValidator =
       new CustomValidator("enRoute", enRouteValidation);
```

The TCS card (prefix 7000-7599 except for xx7x range with a length of 10 which has all digits and doesn't validate using the LUHN algorithm):

```
ValidationAlgorithm TCSValidation =
       New ValidationLength(10).and(new
       ValidationRange("7000","7599",1)).and(new
       ValidationRange("7","7",3).not())

CreditCardValidator TCSValidator =
       new CustomValidator("TCS", TCSValidation);
```

1.2 Design Patterns

- Singleton Pattern – allows the application to register types used and shared by the application.
- Strategy – allows the component to interchangeably use algorithms and credit card validation types.
- Composite Pattern – allows the combination of algorithms using operators

1.3 Industry Standards

Implements the industry standard LUHN algorithm and provides helper constants for the industry standard prefixes.

1.4 Required Algorithms

This component will implement the industry standard LUHN algorithm (in ValidatorLUHN). The LUHN algorithm (also known as mod-10) was developed in the 1960's as a way to validate unique numbers such as credit card number, social insurance numbers, insurance numbers and other number forms. The algorithm is in the public domain and is in wide use today. A good reference can be found at http://www.merriampark.com/anatomycc.htm. Here is a quick overview of how the LUHN algorithm works with an example:

The LUHN algorithm works on digits exclusively any other formatting should be ignored:

1. Starting with the second to last digit and moving left, double the value of all the alternating digits.

2. If any doubled value is greater than or equal to 10, we add the two digits (making up the double value) together – otherwise we use the doubled value itself. We sum up all of these values together.
3. Next, we will take every first, third, fifth etc. number and do nothing but add them to our amount from step 2.
4. If the sum of these numbers is divisible by 10, the number is valid; otherwise, it's not.

Example 1:

Visa Test Credit Card Number:
4111-1111-1111-1111

```
 4 1 1 1 - 1 1 1 1 - 1 1 1 1 - 1 1 1 1
x2  x2    x2  x2    x2  x2    x2  x2
--------------------------------------
 8 1 2 1   2 1 2 1   2 1 2 1   2 1 2 1  ← Step 1 (dashes ignored)
```

Step 2 – since none of the doubled values are >=10, we simply add them together: 22

Step 3 – simply add the odd digits together: 8

Step 4 – add (22 + 8) mod 10 = 0. So this credit card is valid.

Example 2:

Visa Test Credit Card Number:
4111-1111-1111-1191

```
 4 1 1 1 - 1 1 1 1 - 1 1 1 1 - 1 1 9  1
x2  x2    x2  x2    x2  x2    x2  x2
---------------------------------------
 8 1 2 1   2 1 2 1   2 1 2 1   2 1 18 1  ← Step 1
```

Step 2 – 8+2+2+2+2+2+2+(1+8) = 29

Step 3 – simply add the odd digits together: 8

Step 4 – add (29 + 8) mod 10 = 7. So this credit card is **not** valid.

1.5 Thread Safety

Since this component implements a singleton pattern – thread safety needs to be addressed. Thread safety is addressed in different ways. The ValidationAlgorithm implementations are immutable and contain no state information – making them safe in a multithreaded application. The AbstractCreditCardValidator (which all CreditCardValidator implementations extend) will use an internal lock or other (depending on the developer implementation) to prevent multiple thread access to the ValidationAlgorithm. The DefaultIdentifier is immutable in this class and does not need to be address. Likewise, the CreditCardValidationRegistry will need to use an internal lock or other (again, depending on the developer implementation) to control access to the registry.

1.6 Component Class Overview

Below is a **very** short overview of the classes in this component. Please refer to the class diagram's documentation tab for a more complete overview of each class

CreditCardValidatorRegistry:
Acts as a registry for all the *CreditCardValidators* that are defined by this component and/or the application and can be initialized from a configuration file. As a helper for the application – also implements a singleton pattern.

CreditCardValidator:
Defines the contract a *CreditCardValidator* must implement for this component to use it.

AbstractCreditCardValidator (implements CreditCardValidator):
An abstract implementation of a CreditCardValidator that provides most of the base functionality of the CreditCardValidator interface and provides some new functionality: the ability to 'and' validators together and the ability to mutate the algorithm.

CreditCardValidatorOr (extends AbstractCreditCardValidator):
An implementation of a *CreditCardValidator* that will combine the validation of two other *CreditCardValidators* using an 'or' pattern.

JCBValidator (extends AbstractCreditCardValidator):
An implementation of a *CreditCardValidator* that will validate JCB (issued credit cards.

MasterCardValidator (extends AbstractCreditCardValidator):
An implementation of a *CreditCardValidator* that will validate MasterCard (issued credit cards.

AmericanExpressValidator (extends AbstractCreditCardValidator):
An implementation of a *CreditCardValidator* that will validate American Express issued credit cards.

DinersClubValidator (extends AbstractCreditCardValidator):
An implementation of a *CreditCardValidator* that will validate DinersClub/Carte Blanche (issued credit cards.

DiscoverValidator (extends AbstractCreditCardValidator):
An implementation of a *CreditCardValidator* that will validate Discover (issued credit cards.

CustomValidator (extends AbstractCreditCardValidator):
An implementation of a CreditCardValidator that will allow the application to setup it's own validator.

ValidationAlgorithm:
Defines the contract a ValidationAlgorithm must implement for this component to use it.

AbstractValidationAlgorithm (implements ValidationAlgorithm):
An abstract implementation of ValidationAlgorithm that provides operator type functionality.

ValidationAlgorithmAnd (extends AbstractValidationAlgorithm):
An implementation of a *ValidationAlgorithm* that will perform an 'and' validation between two other *ValidationAlgorithm*.

ValidationAlgorithmOr (extends AbstractValidationAlgorithm):
An implementation of a *ValidationAlgorithm* that will perform an 'or' validation between two other *ValidationAlgorithm*.

ValidationAlgorithmNot (extends AbstractValidationAlgorithm):
An implementation of a *ValidationAlgorithm* that will perform a 'not' validation on another *ValidationAlgorithm*.

ValidationFormatLength (extends AbstractValidationAlgorithm):
An implementation of a ValidationAlgorithm that validates that passed credit card is a specific length

ValidationFormatAllDigits (extends AbstractValidationAlgorithm):
An implementation of a ValidationAlgorithm that validates that passed credit card is all digits ('0'-'9').

ValidationLUHN (extends AbstractValidationAlgorithm):
An implementation of a ValidationAlgorithm that validate using the LUHN algorithm (as described above).

ValidationRange (extends AbstractValidationAlgorithm):
An implementation of a ValidationAlgorithm that validates that the digits at a specific position are within a specified range.

ValidationPrefix (extends ValidationRange):
A subclass of ValidationRange that validates the passed credit card text starts with a given prefix. This is a utility/helper class to more easily define prefixes.

1.7 Component Exception Definitions

NullPointerException:
This represents some nullable field (String, object, etc) was passed to a function that cannot handle null values.

Almost (there are some exceptions) any class that deals with a String or Object will throw this exception when a null value is encountered. The methods that throw this are clearly marked in the tags section of the documentation tab.

IllegalArgumentException:
This represents some class was passed a string that is empty or an integer out of range.

Almost (there are some exceptions) any class that deals with a String will throw this exception when the string is empty. Additionally, some methods that deal with integer will throw this if the integer is out of range (less than 0 typically). The methods that throw this are clearly marked in the tags section of the documentation tab.

2. Environment Requirements

2.1 Environment

- At minimum, Java1.3 is required for compilation and executing test cases.

2.2 TopCoder Software Components

- Configuration Manager version 2.1

2.3 Third Party Components

None.

3. Installation and Configuration

3.1 Namespace

com.topcoder.ecommerce.creditcard.validation

3.2 Configuration Parameters

The configuration manager is used by the CreditCardValidatorRegistry to populate itself with a standard set of validators if specified. The registry does this by using the ConfigurationManager component to retrieve all the properties for the namespace 'com.topcoder.ecommerce.creditcard.validation.CreditCardValidatorRegistry'. The property name is then assumed to be the identifier for the validator and the property value is the fully-qualified class name to instantiate.

The following would be a (partial) example XML config file:

```
<CMConfig>
  <Config name="
com.topcoder.ecommerce.creditcard.validation.CreditCardValidatorRegistry
">
     <Property name="VISA">
       <Value>
          com.topcoder.ecommerce.creditcard.validation.VisaValidator
       </Value>
     </Property>
     <Property name="OurCard">
       <Value>com.acme.financial.bankingapps.CardValidator </Value>
     </Property>
  </Config>
</CMConfig>
```

3.3 Dependencies Configuration

None

4. Usage Notes

4.1 Required steps to test the component

- Extract the component distribution.
- Follow Dependencies Configuration.
- Execute ‘ant test’ within the directory that the distribution was extracted to.

4.2 Required steps to use the component

This component assumes the configuration manager has been instantiated and loaded with the configuration file containing (if necessary) the namespace above.

4.3 Demo

This component allows the application to be very flexible in how it uses this component:

```
String cc = "1234-1234-1234";

// Use the registry to see which cards it matches
List cards = CreditCardValidatorRegistry.getInstance().validate(cc);
If(cards.size() > 0) … // is valid

// Validate against a specific card using the registry
boolean valid =
CreditCardValidatorRegistry.getInstance().getCreditCardValidator(Discover
Validator.IDENTIFIER).isValid(cc);

// Or validate against the instance specifically
boolean valid = new DiscoverValidator().isValid(cc);

// Or validate against an algorithm specifically
boolean valid = new ValidatorLUHN().isValid(cc);
```

The application can put together ‘profiles’ of credit card validations:

```
// Create a profile for MasterCard/Visa
CreditCardValidator MV =
       new MasterCardValidator().or(new VisaCardValidator());

// Validate against it directly
boolean valid = MV.isValid(cc);

// Or add it to the registry for the application to use under
// an application specific identifier
CreditCardValidatorRegistry.getInstance().addCreditCardValidation("mv",
MV);
```

The application can create new validations using the expressive class construction for algorithms:

```
// Create the enRoute validation
ValidationAlgorithm enRouteValidation =
       new ValidationLength(15).and(
       new ValidationPrefix("2014").or(
       new ValidationPrefix("2149"))).and(
       new ValidationFormatAllDigits().and(
       new ValidationLUHN());


// Create the validator
```

```
CreditCardValidator enRouteValidator =
       new CustomValidator("enRoute", enRouteValidation);

// Add it to the registry using default identifier
CreditCardValidatorRegistry.getInstance().addCreditCardValidation(enRoute
Validator);
```

The application can also validate against all industry specific cards:

```
// Create a gas industry validation algorithm using
// the major industry identifiers (MII)
ValidationAlgorithm gasAlgo =
       new ValidationPrefix(ValidationPrefix.MII_PETROLEUM).and(
       new ValidationFormatAllDigits()).and(
       new ValidationLUHN());

// Create the custom validation
CreditCardValidator gasValidator =
       new CustomValidator("gas", gasAlgo);

// Add it to the registry
CreditCardValidatorRegistry.getInstance().addCreditCardValidation
(gasValidator);
```

5. Implementations

This component comes with a number of credit card validators built in. Please refer to the constructor documentation for each class for specifics in how the ValidationAlgorithm is put together. The following overview of validations is provided as an overview:

Credit Card Type	**Prefix**	**Length of CC Number**	**Implementing Class**
MasterCard	51-55	16	MasterCardValidator
Visa	4	13 or 16	VisaValidator
American Express	34 or 37	15	AmericanExpressValidator
Diners Club/ Carte Blanche	300-305 or 36 or 38	14	DinersClubValidator
Discover	6011	16	DiscoverValidator
JCB	3	16	JCBValidator
JCB	2131,1800	15	JCBValidator

6. Future Enhancements

- Implement additional credit card types – enRoute, Australian Bank Credit and gas industry cards to mention a few
- Implement validators that communicate to the services to do true validation on the number.