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. An ASP.NET Validation Control interface is also supported in order to add the credit card validation directly to an ASP.NET page

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 Austrialian 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’ operator). 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 it’s use. An enumeration pattern is NOT used for the credit card types because they 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) but also includes a powerful regex validator for validating formats.

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 ValidationPrefix(2014) | new ValidationPrefix(2149)) &
       new ValidationLength(15) &
       new ValidationFormatAllDigits() &
       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 except for the last character – which is either an ‘A’ or a ‘D’ and doesn’t validate using the LUHN algorithm):

```
ValidationAlgorithm TCSValidation =
       (new ValidationRange(7000,7599,1) & !new ValidationRange(7,7,3)) &
       new ValidationLength(10) &
       new ValidationFormat(new Regex("\d{9}[AD]");

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.

1.3 Industry Standards

This component provides a Validator (ASPNETCreditCardValidator) for ASP.NET pages. This validator extends the BaseValidator class that can be embedded into ASP.NET pages like:

```
<%@ Register TagPrefix="custom"
Namespace="TopCoder.Validation.CreditCard"
Assembly="CreditCardValidator" %>

    <html>
    <body>
    <form runat="server">
        <asp:TextBox ID="CCNumber" Runat="server" />
        <custom:ASPNETCreditCardValidator
            ControlToValidate="CCNumber"
            Runat="server"
            ErrorMessage="Credit Card Number Invalid!" />
        <asp:Button Text="Validate" Runat="server" />
    </form>
    </body>
</html>
```

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 and may be used from an IIS server – so thread safety needs to be addressed. Thread safety is addressed in a few different ways. The ValidationAlgorithm implementation 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

ASPNETCreditCardValidator:
This class provides an ASP.NET validator for Credit Card entry that will validate using the singleton instance of the CreditCardValidatorRegistry to see if the credit card matches any of the entries.

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 an 'not' validation on another *ValidationAlgorithm*.

ValidationFormat (extends AbstractValidationAlgorithm):
An implementation of a ValidationAlgorithm that will validate credit card text against a regex pattern and return true if it matches.

ValidationFormatLength (extends ValidationFormat):
An implementation of a *ValidationAlgorithm* that will perform validate the length of the credit card text.

ValidationFormatAllDigits (extends ValidationFormat):
An implementation of a *ValidationAlgorithm* that will validate that the passed string contains all digits.

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

ArgumentNullException:
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.

ArgumentOutOfRangeException:

This represents integer field that was passed to a class that was out of a defined range for that class.

Almost (there are some exceptions) any class that deals with an integer will throw this exception when that integer violates some minimum or maximum range. The methods that throw this are clearly marked in the tags section of the documentation tab.

ArgumentException:

This represents some class was passed a string that is empty.

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

1.8 Component Benchmark and Stress Tests

This component should be very lightweight in it's implementation. The areas of concern (where any real processing is done) are in the ValidatorAlgorithms. All but one of the algorithms use .NET provided classes and will be mainly dependent upon the execution speed of the .NET classes. The one area where TCS and the developer has control over is in the LUHN algorithm implementation – this can be implement in linear time and tests should confirm the effectiveness of the implementation.

Memory size should be low in this application. Adding new CreditCardValidations to the registry should only cause this component to consume the memory required for the validation (which should be negligible) and whatever overhead the chosen IDictionary implementation uses.

The developer should choose an IDictionary and IList implementation that are efficient and have low memory footprints under small size conditions. Ideally this component will hold about 15 CreditCardValidations in the registry and at most < 100 (there are not that many unique credit card vendors [there are many more branded cards – but they all come down to the same issuer validation]). The IList implementation returned from the Registry will contain anywhere from 1 to 100 members (never greater than what the IDictionary will contain) and will generally see it's size < 5 elements.

This component is thread safe and should have tests to confirm this (in the registry – one thread getting while another deleting the same CreditCardValidator or in the AbstractCreditCardValidator – one getting the algorithm and another setting). This component will probably be used in a read-only type of mode except for when it is created. Given that, ideally the developer would use a Multi-Read, Single Write type of locking system where multiple readers are allowed simultaneously.

2. Environment Requirements

2.1 Environment

- Windows 95/98/NT/2000/XP
- .Net Framework 1.0 or later.

2.2 TopCoder Software Components

- Configuration Manager version 1.0

- LinkedList version 1.0 may be required if the developer chooses to use it as the IList implementation.

2.3 Third Party Components

None.

3. Installation and Configuration

3.1 Namespace

TopCoder.Validation.CreditCard

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 'TopCoder.Validation.CreditCard.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:

```
…
<TopCoder.Validation.CreditCard.CreditCardRegistry>

<Discover>TopCoder.Validation.CreditCard.DiscoverValidator</Disco
ver>
   <VISA>TopCoder.Validation.CreditCard.VisaValidator</VISA>
   <OurCard>ACME.Financial.BankingApps.CardValidator</OurCard>
</TopCoder.Validation.CreditCard.CreditCardRegistry>
…
```

3.3 Dependencies Configuration

None

4. Usage Notes

4.1 Required steps to test the component

- Extract the component distribution.
- Follow Dependencies Configuration.
- Execute 'nant 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
Ilist cards = CreditCardValidatorRegistry.Validate(cc);
If(cards.Count > 0) … // is valid

// Validate against a specific card using the registry
bool valid =
CreditCardValidatorRegistry.GetCreditCardValidator(DiscoverValidator.IDEN
TIFIER).IsVAlid(cc);

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

// Or validate against an algorithm specifically
bool 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() | new VisaCardValidator();

// Validate against it direcly
bool valid = MV.IsValid(cc);

// Or add it to the registry for the application to use under
// an application specific identifier
CreditCardValidatorRegistry.AddCreditCardValidation(“mv”, MV);
```

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

```
// Create the enRoute validation
ValidationAlgorithm enRouteValidation =
       (new ValidationPrefix(2014) | new ValidationPrefix(2149)) &
       new ValidationLength(15) &
       new ValidationFormatAllDigits() &
       new ValidationLUHN();

// Create the validator
CreditCardValidator enRouteValidator =
       new CustomValidator(“enRoute”, enRouteValidation);

// Add it to the registry using default identifier
CreditCardValidatorRegistry.AddCreditCardValidation(enRouteValidator);
```

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) &
      new ValidationFormatAllDigits() &
      new ValidationLUHN();

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

// Add it to the registry
CreditCardValidatorRegistry.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

- When TCS implements an Object Factory, the registry can then use that (in addition to the configuration manager) to create the validation classes in a more flexible manner (allowing the application to potentially pass in setup information such as formatting).
- 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.