Complex Boolean Expression Evaluator Requirements Specification

1. Scope

1.1 Overview

This component provides complex Boolean expression evaluation and pluggable expression statements evaluators. The Complex Boolean Expression Evaluator provides an expression evaluation system that allows for pluggable statement evaluations. This component will take an incoming string, use the internal evaluators to evaluate each expression found and return the Boolean result.

1.2 Logic Requirements

1.2.1 String Eval Method

The component will provide a single method for simple string based evaluation. The example below is suggested but not required. The designer will modify this signature as needed to provide simple but accurate functionality:

```
public bool eval(string expression) throws EvaluationException;
```

1.2.2 Parse Method

The component will provide a method to parse an expression without evaluating it. This will allow the component user to cache the statement for later use. The statement values (but not its structure) will be modifiable by the component user. The example below is suggested but not required. The designer will modify this signature as needed to provide simple but accurate functionality:

```
public Statement parse(string expression) throws
EvaluationParseException;
```

1.2.3 Statement Eval Method

The component will provide a single method for the evaluation of previously compiled statements. The example below is suggested but not required. The designer will modify this signature as needed to provide simple but accurate functionality:

```
public bool eval(Statement statement) throws EvaluationException;
```

1.2.4 Pluggable Evaluators

- The expression will be built out of many expressions. Each individual expression could be a function of some kind. See example below. These functions will need to have plug-ins defined to actually evaluate the expression to determine whether or not it is true or false.

1.2.5 Initial Evaluators Supported

- Four initial evaluators must be supported:
  - Math Functions – This expression evaluator will need to support all the functionality in our Math Evaluation component.
  - Regular Expressions – This expression evaluator will check to make sure that the given string matches the given pattern. C#'s built in regular expression evaluator may be used for this component. The function will appear in the statement as follows:
    - regexp(value,expression)
  - Count – This will count the number of elements in a list. Examples:

            - count([1,2,3,4]): 4
            - count([]): 0
            - count([2]): 1
            - count([2,1]): 2

        The function will appear in the statement as follows:

            - count([list values])

    - Contains – This method will be used to determine whether or not a value is part of a given list. Examples:
        - contains(1,[1,2,3]): true
        - contains(1,[2,3]): false
        - contains(1,[]): false
        - contains(1,[1,1,1]): true

      The function will appear in the statement as follows:

        - contains(value,[list values])

1.2.6 Compound Statements

- The component must be able to support compound statements with the following operators supported:
    - &&
    - ||
    - !
    - !=
    - ==
- These expressions will have the same meaning as they do in Java.

1.2.7 Order of Operations

- Use the same order of operations found in the Math Expression Evaluator.

1.2.8 Invalid Expressions

- If an invalid expression is found, such as
    - true == != false
    - 53 < 7 - [1,2,3]

  An exception should be thrown.

1.2.9 Exceptions

- Any exception that the designer creates in the overall design of this component must extend the Base Exception found in the Base Exception component.

1.2.10 Logging

- Logging should be implemented using the Logging Wrapper component. The following information should be logged:

- DEBUG:
  - The entrance and exit of each method call. This includes both public and private methods.
- INFO:
  - Information about each of the public method calls. For example, eval should log something like: “evaluating expression …”
- ERROR:
  - Log any checked exception before throwing it to the calling application.

1.2.11 Example

- Here is an example of an incoming statement:

```
((100 > 10) && (contains(1,[1,3])) && ((((10 * 10) / 90) > 10) || (count([1,2,3])
> 3)) && (regexp('THIS IS THE 23rd TEST.','*23*')))
```

This component would use the evaluators to evaluate each of the statements into:

```
((true) && (true) && ((false) || (false) && (true))
```

And return `false` back to the calling application.

1.3 Required Algorithms

None required.

1.4 Example of the Software Usage

An application might use this component to create custom expression statements and combine these custom expressions into a complex statement. A stock analysis application might create some custom expressions to analyze the stocks history and then use this component to determine whether or not a stock should be bought or sold.

1.5 Future Component Direction

More expression evaluators will be developed and added to this list of initially supported evaluators.

2. Interface Requirements

2.1.1 Graphical User Interface Requirements

None required.

2.1.2 Internal Interfaces

None required.

2.1.3 External Interfaces

None required.

2.1.4 Environment Requirements

- Development language: C#
- Compile target: Microsoft .net Framework v1.1

2.1.5 Package Structure

```
Topcoder.Util.ComplexBooleanEval
```

3. Software Requirements

3.1 Administration Requirements

3.1.1 What elements of the application need to be configurable?

- Each type of expression type will need to have an evaluator defined for it. The defaults will be defined in configuration files; however, they should be able to be set programmatically.

3.2 Technical Constraints

3.2.1 Are there particular frameworks or standards that are required?

None required.

3.2.2 TopCoder Software Component Dependencies:

- Configuration Manager: 2.0
- Math Expression Evaluator: 1.0.1
- Logging Wrapper: 2.0

**Please review the TopCoder Software component catalog for existing components that can be used in the design.

3.2.3 Third Party Component, Library, or Product Dependencies:

No third party dependencies.

3.2.4 QA Environment:

- Windows 2000
- Windows 2003

3.3 Design Constraints

The component design and development solutions must adhere to the guidelines as outlined in the TopCoder Software Component Guidelines. Modifications to these guidelines for this component should be detailed below.

3.4 Required Documentation

3.4.1 Design Documentation

- Use-Case Diagram
- Class Diagram
- Sequence Diagram
- Component Specification

3.4.2 Help / User Documentation

- Design documents must clearly define intended component usage in the 'Documentation' tab of Poseidon.