Enhanced String Handling

What is it all about

The motivation behind the ***Enhanced String Handling*** System is to enable programmatic handling of substring transformation. For example:

Input string:
*Today, **{DateTime::dd/MM/yyyy}**, is the beginning of the rest of your life.*

Intended output string:
*Today, 2**1/11/2010**, is the beginning of the rest of your life.*

Assuming that today's date is 21/11/2010 (formatted as: dd/MM/yyyy).

I have encountered the need for such ***Enhanced String Handling*** when dealing with user input or input specifications that needed consistency between program invocations or between separate string evaluations. An example of such a need is a formula that needs to specify a previous business day for a database query specification. Another example where such processing is handy, is the need to have a configuration file (key, value) referring to other (key, value) string pairs as this article's predecessor deals with http://www.codeproject.com/KB/files/Enhanced-Configuration3.aspx. In general the ***Enhanced String Handling*** should be able to take any construct in the format of ***{Identifier::Value}*** and according to the rules set forth for the ***Identifier***, transform the ***Value*** part of the construct.

Where will you go for help if need be

I assume that you are familiar enough with regular expression handling to be able to navigate through this article. If there are aspects of regular expression handling that you need to figure out, then you have a lot of good places to visit and learn from. For example the within the MSDN site: http://msdn.micorosoft.com, search for: "regular expression language elements c#". Also, a Good Samaritan, Jan Goyvaerts, wrote an outstanding regular expression tutorial: http://www.regular-expressions.info/tutorial.html.

To try out your regular expression before you bake it into code you can pass your regular expression through a Regulator or a Tester. VS2010 has a regular expression evaluator in an extension. I have employed Derek Slager's tester: http://derekslager.com/blog/posts/2007/09/a-better-dotnet-regular-expression-tester.ashx for a few years now.

You may always pose questions at the end of the article as well.

Simple and complex

A substring to be processed, like in the earlier example of ***{DateTime::dd/MM/yyyy}***, is called a simple expression. A substring may be complex where it is composed of nested simple expressions. A simple expression is an expression that cannot be broken down into simpler expressions.

Format of a simple substring to be transformed

A simple expression is composed of:

Open delimiter: In the above example it is an open brace ("**{**"), though you may specify a different open delimiter choice. An open delimiter is not restricted to a single character; "***<delimiter>***" could serve as an open delimiter too.

An identifier: In the above example it is: ***DateTime***. An Identifier is not confined to a C# identifier; it may contain spaces and other characters.

A separator: By convention it is a double colon: "***::***", a convention that you may choose differently. A separator may not be equal to a delimiter.

Value: In the above example it is the date format: ***dd/MM/yyyy***. The Value string may have multiple parts. Like in the construct ***{ForeignKey::Value}***, where the ***Value*** has two parts: a **FilePath** and a **Key** for a complete construct of: ***{ForeignKey::C:\dir1\dir2\filename.exe::Key1}***. I recommend that you use the same separator string within the ***Value***, as the one you use to separate the ***Value*** from the **Identifier**. The ***Value***, on the other hand, may also be empty; in which case you should not forget the separator. So for example: ***{CurrentDir::}*** is the correct format for an empty ***Value***, as opposed to: ***{CurrentDir}***.

Closing delimiter: In the above example, it is the close brace ("**}**") and you may specify any close delimiter neither equals the open delimiter nor equal to the separator. Similar to the open delimiter, the close delimiter is not restricted to a single character; "***</delimiter>***" is a dandy close delimiter to "***<delimiter>***".

Special construct that does not fit into the above format

There is a potential need for a special construct, a construct that contains no identifier and no separator. We reserve such a construct to the ***ProcessLiteral*** class. This construct takes a string of the format ***{literal}*** and returns the ***literal*** itself. The class ***ProcessLiteral*** will not operate on a construct containing a separator.

Introduction

Historically, this adaptation started as an ***Enhanced Configuration File Handling***, see: http://www.codeproject.com/KB/files/Enhanced-Configuration3.aspx, and soon enough I found myself in a need of a more generic string handling. Something that will work not only as confined to a configuration file, but also in driving other specification requirements, mostly UI user-specifications.

Goals for our Enhanced String Evaluation

- Ability to externally specify the delimiters and separator.
- Expandability: allows the developer to define any new constructs.
- The system must be flexible enough to allow multiple constructs to be handled simultaneously.
- The system should be able to deal with nested constructs.

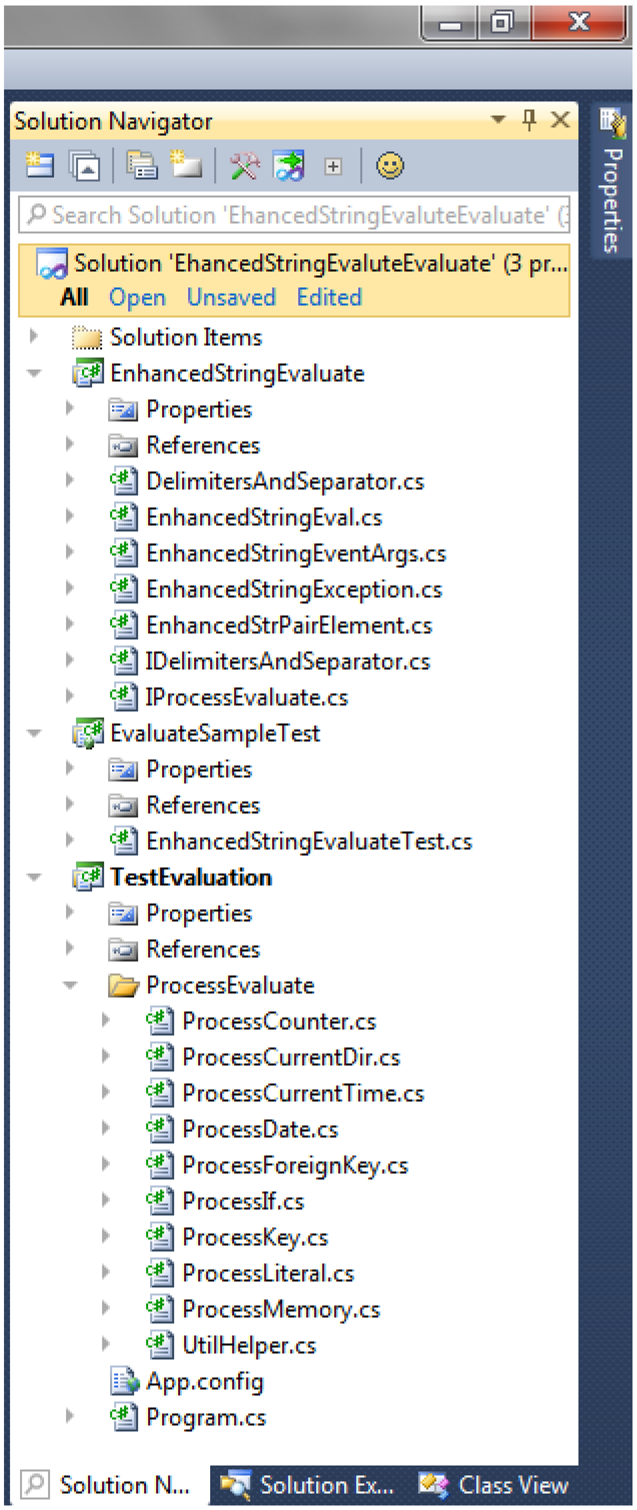
- The system must be easy to use and expand. In this case I confine "easy" to the use of regular-expression constructs.
- The system should be able to leave an unhandled construct intact without throwing exceptions.

Transformations as examples

ProcessCounter:	Returns a running counter value.
ProcessCurrentDir:	Returns the current directory.
ProcessCurrentTime:	Returns current Date/time.
ProcessDate:	Returns the current date (a subset of ***ProcessCurrentTime***, but since ***ProcessDate*** was written first, I include it here as a simpler and good example).
ProcessForeignKey:	Returns the value from within a file populated with (Key, value) pairs.
ProcessIf:	Allows for "if" logic
ProcessKey:	Returns a value from a collection of (key, value) pairs.
ProcessLiteral:	Returns the construct without the delimiters. The special-case construct.
ProcessMemory:	Stores and returns a value, similar to a calculator memory.

Each one of these constructs is further exemplified in ***EnhancedStringEvaluateTest*** class (file: ***EnhancedStringEvaluateTest.cs***) part of the ***EvaluateSampleTest*** assembly, in the accompanying code (see Figure 1—Code Layout).

Throughout the article I will refer to the above Process classes (and to the Process classes that you will build) as ***ProcessXxx*** classes. These classes are in the ***TestEvaluation*** assembly within the folder ***ProcessEvaluate*** (see Figure 1—Code Layout).



Solution Navigator

Search Solution 'EhancedStringEvaluteEvaluate' (

Solution 'EhancedStringEvaluteEvaluate' (3 pr...

All Open Unsaved Edited

```
Solution Items
EnhancedStringEvaluate
    Properties
    References
    DelimitersAndSeparator.cs
    EnhancedStringEval.cs
    EnhancedStringEventArgs.cs
    EnhancedStringException.cs
    EnhancedStrPairElement.cs
    IDelimitersAndSeparator.cs
    IProcessEvaluate.cs
EvaluateSampleTest
    Properties
    References
    EnhancedStringEvaluateTest.cs
TestEvaluation
    Properties
    References
    ProcessEvaluate
        ProcessCounter.cs
        ProcessCurrentDir.cs
        ProcessCurrentTime.cs
        ProcessDate.cs
        ProcessForeignKey.cs
        ProcessIf.cs
        ProcessKey.cs
        ProcessLiteral.cs
        ProcessMemory.cs
        UtilHelper.cs
    App.config
    Program.cs
```

Figure 1. Code Layout.

High level architecture overview

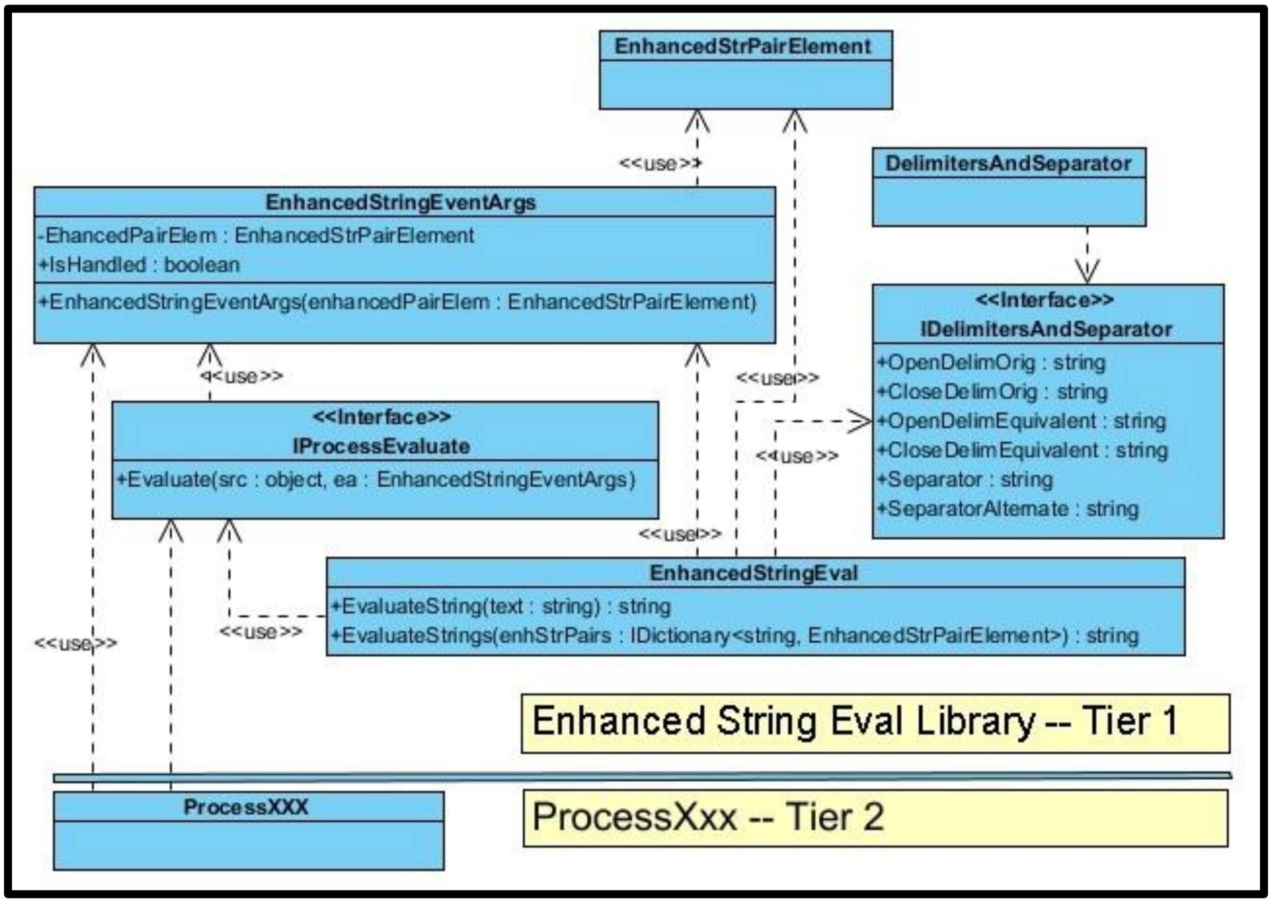

EnhancedStrPairElement
<<use>>
DelimitersAndSeparator
EnhancedStringEventArgs
-EhancedPairElem : EnhancedStrPairElement
+IsHandled : boolean
+EnhancedStringEventArgs(enhancedPairElem : EnhancedStrPairElement)
<<Interface>>
IDelimitersAndSeparator
+OpenDelimOrig : string
+CloseDelimOrig : string
+OpenDelimEquivalent : string
+CloseDelimEquivalent : string
+Separator : string
+SeparatorAlternate : string
<<use>>
<<use>>
<<use>>
<<Interface>>
IProcessEvaluate
+Evaluate(src : object, ea : EnhancedStringEventArgs)
<<use>>
EnhancedStringEval
+EvaluateString(text : string) : string
+EvaluateStrings(enhStrPairs : IDictionary<string, EnhancedStrPairElement>) : string
<<use>>
<<use>>
Enhanced String Eval Library -- Tier 1
ProcessXXX
ProcessXxx -- Tier 2

Figure 2. Architecture Overview

The solution presented in the accompanying code, a two tier solution, has a library driven by the ***EnhancedStringEval*** class is in the one tier and ***ProcessXxx*** classes in the other tier. The ***EnhancedStringEval*** class has two methods, that we will concentrate on, an ***EvaluateString()*** and an ***EvaluateStrings()***, singular and plural method names. These methods are public in scope. The ***EnhancedStringEval*** class "knows" of the ***ProcessXxx***'s ***IProcessEvaluate*** interface (see Figure 2—Architecture Overview).

The ***IProcessEvaluate*** interface is implemented by each of the ***ProcessXxx*** classes and provides the communication transport between the Library and each of the ***ProcessXxx*** classes; following the Strategy design pattern.

This article is dedicated to exploring this interaction, the library in tier 1 and more importantly how to build a ***ProcessXxx*** class. Building ***ProcessXxx*** classes will be your task in creating new ***{Identifier::Value}*** constructs.

Getting started

If what you need to do is solve an immediate problem at hand, then:

- Include, in your solution, the ***EnhancedStringEvaluate*** library.

- Build a specific ***ProcessXxx*** evaluation class. The ***EnhancedStringEval*** library will call it to evaluate your specific string construct.
- Access the string construct from your client code, similar to the examples in ***EnhancedStringEvaluateTest*** class in the ***EvaluateSampleTest*** assembly. And you are done!

A ProcessXxx class

In order to build a ***ProcessXxx*** evaluation class, let's go through an existing example, like the ***ProcessCounter*** class as presented in the ***TestEvaluation*** assembly as part of the ***ProcessEvaluate*** folder, that will process constructs like: ***{Counter::Name}***.

So ***{Counter::page}***, for example, will return 0 on it first invocation, 1 on its second invocation, 2 on its third and so on. And ***{Counter::Footnote}*** will be independent of ***{Counter::page}***. ***{Counter::Footnote}*** will return 0, 1, 2, … on its first, second, third invocation and so forth.

Any of the ***ProcessXxx*** classes is responsible for 2 tasks:

ProcessXxx Task 1: Recognizing the pattern it needs to transform. In general you will set this recognition regular expression pattern in the constructor.

ProcessXxx Task 2: Support the ***IProcessEvaluate*** interface. This interface has a single method:

```
        void Evaluate(object src, EnhancedStringEventArgs ea);
```

and as its name suggests it is responsible for evaluation of the substring that it supports.

The `Evaluate(object src, EnhancedStringEventArgs ea)` method, which is an event delegate callback method and it is responsible for 3 tasks:

Evaluate Task 1: Read from the string construct passed in through the ***ea*** parameter. The string like "Today, {Date::mm/dd/yyyy}, in the market …" is passed in ***ea.EhancedPairElem.Value***.

Evaluate Task 2: Setting the ***ea.IsHandled*** to **true** if the ***Evaluate()*** method evaluated the ***ea.EhancedPairElem.Value*** to a different value from its original value and to **false** otherwise.

Evaluate Task 3: In case that the ***Evaluate()*** method evaluated the original text to a different value from its original ***ea.EhancedPairElem.Value***, then update the value pointed to by ***ea.EhancedPairElem.Value*** to the new value.

ProcessCounter

The ***ProcessCounter*** class that is provided as part of the accompanying code can accept as valid the following formats:

- ***{Counter::Name}*** Will yield 0 upon first invocation, 1 upon second invocation and in general it will yield previous value + 1 after initial value of 0.
- ***{Counter::Name::Init}*** Will yield 0 always and reinitialize the named counter.
- ***{Counter::Name::Next}*** Will yield previous value + 1. Will throw exception if used as first invocation.
- ***{Counter::Name::Previous}*** Yields previous value - 1. Will throw exception if used as first invocation.

- ***{Counter::Name::=n}*** Yields n. Initialize the named counter to n (where n is an integer, may be a negative integer). So ***{Counter::Name::Init}*** and ***{Counter::Name::=0}*** represent the same behavior.
- ***{Counter::Name::+n}*** Increments the named counter by n (n is an integer, may be a negative integer). Will throw exception if used as first invocation. So ***{Counter::Name::Next}*** and ***{Counter::Name::+1}*** represent the same behavior.
- ***{Counter::Name::-n}*** Decrements the named counter by n (n is an integer, may be a negative integer). Will throw exception if used as first invocation. In a sense, ***{Counter::Name::-n}*** is redundant as "n" may assume a negative number. So ***{Counter::page count::+-5}*** is equivalent to ***{Counter::page count::-5}***.

Any other format will be rejected as not part of ***ProcessCount*** and therefore will not be evaluated. So, for example, ***{Counter::Example Count::Init}*** is a valid format and will return 0, but ***{Counter::Example Count::The best example in the world}*** will return itself back and the processing will not throw an exception.

We will refer to the "Init", "Previous", "Next", "=n", "+n" and "-n" optional sub-patterns as the **"Extras"**.

A regular expression capturing the above possibilities

In an attempt to take care of both ***ProcessXxx*** task responsibilities, we will start by first: Recognizing the pattern.

In building the recognition regular expression let's consider the following:

```
string pattern = @"({)\s*Counter\s*::(?<name>[^{}:]+)" +
      @"(::\s*(?<extras>(init)|(next)|(previous)|" +
      @"((?<op>[=+-])\s*(?<direction>[+-])?\s*(?<val>[0-9]+)))?)?\s*(})";
```

A digression into colon handling

Now, we may care to allow a single colon within the name but not a double colon. We are looking only at the name part of the pattern: "`(?<name>[^{}:]+)`". So the name of the counter may be a simple name like "page" as in ***{Counter::page}***, or a more complicated name like "Dr. Seuss: pages of fun" as in ***{Counter::Dr. Seuss: pages of fun}***. When it comes to ***Counter*** processing, allowing a colon within a name is not critical, however, when it comes to **DateTime** format specification a single colon is more significant, so let's discuss it.

In order to accommodate allowing a single-colon and not a double-colon in the ***Name***, we need to write a pattern that will allow us to codify, in a regular expression language: "anything but this string"? Where in our case "anything but this string" is "anything but a double colon". The regular expression language can handle "anything but this character" OOTB (out of the box), but it does not handle "anything but this string" OOTB.

One possible solution is to note that we can reduce the problem of "anything but a double colon" to "any string that does not have a colon precedes a colon". Let's consider the following construct:

```
:?[^:]+(:[^:]+)*:?
```

This regular expression string pattern matches on a single colon and on any colon that is followed or preceded by a non-colon character.

Allowing for the fact that the name should not contain the brace delimiters, we get the pattern for the name as:

```
        (?<name>:?[^{}:]+(:[^{}:]+)*:?)
```

Now putting the ***Counter*** regular expression construct together, we have:

```
string pattern = @"({)\s*Counter\s*::(?<name>:?[^{}:]+(:[^{}:]+)*:?)" +
         @"(::\s*(?<extras>(init)|(next)|(previous)|" +
         @"((?<op>[=+-])\s*(?<direction>[+-])?\s*(?<val>[0-9]+)))?)?\s*(})";
```

Adding Delimiters that span multiple character strings

You are writing your own patterns, so if your delimiters are the open/close braces and your separator is a double colon (all default values) then you are all done with the ***ProcessCounter*** recognition pattern—it is provided above.

If your delimiters are any other single charactered delimiters and your separator is the default double colon value, then you are done with the pattern after a simple delimiter substitution.

However, if your delimiters are multi-charactered strings, then we are back to where we were before: the ***Name*** needs to handle "anything but these strings" where now "these strings" now are the open/close delimiters.

We now need a new way to deal with the issue: how do we codify, in a regular expression language, "anything but this multi-charactered string", or simply put "anything but this string".

A possible solution is to substitute the delimiters with a single-character equivalent and now we have reduced the "anything but this string" to "anything but this character" which the regular expression language can handle—OOTB. The substitution is a three step process:

- Substitute each of the open/close multi-charactered delimiters with a single charactered delimiter; a character that is very, very unlikely to occur within the text. So for the sake of our discussion let's say that the open delimiter is "***<delimiter>***" and the close delimiter is "***</delimiter>***" and we will substitute them with the unicode character 1 ('\u0001') and character 2 ('\u0002') respectively.
- Call the ***Evaluate()*** method in the ***EnahancedStringEval*** class library.
- Replace back any remaining Unicode \u0001 and \u0002 with their equivalent open and close delimiters.

The pattern-magic (pattern of recognition) now looks like so:

```
string pattern = string.Format(
   @"({0})\s*Counter\s*::(?<name>:?[^{0}{1}:]+(:[^{0}{1}:]+)*:?)" +
   @"(::\s*(?<extras>(init)|(next)|(previous)|" +
   @"((?<op>[=+-])\s*(?<direction>[+-])?\s*(?<val>[0-9]+)))?)?\s*({1})",
      _delim.OpenDelimEquivalent, _delim.CloseDelimEquivalent);
```

Where ***_delim.OpenDelimEquivalent*** and ***_delim.CloseDelimEquivalent***, are either the delimiters themselves, in the case that they are single charactered, or they are a single-charactered substitution.

Important: The ***EnhancedStringEval*** library expects single-charactered delimiters, either originally single charactered or substituted equivalent single-charactered.

Axiom of choice

We have discussed two possibilities to construct the pattern of recognition for "anything but this string". The one method is where we did not allow a colon to precede a colon and the second method was substitution of a whole string with a single character then using the regular expression OOTB language capabilities for recognizing "anything but this character". This second choice is used for the delimiters' handling; but, we can employ this same substitution idea to the separator string (the double colon string). The choice of which possibility to use for the separator will affect the second task of ***ProcessXxx***, supporting the ***IProcessEvaluate***.

Putting the ProcessCounter together

The second task we need to accomplish in order to codify the ***ProcessXxx*** class is implementing the ***IProcessEvaluate*** interface. We will go through the two possibilities of doing so. (The code in the ***ProcessCounter*** class has a property, ***PossibilityOption***, keeping track of the possibilities.)

Possibility 1

Treating a colon in the ***Name***, in the manor of a colon that cannot precede a colon; we need a class variable that will understand the pattern to transform, ***_reCounter***:

```
// Class instance variables
private readonly Regex _reCounter;
private readonly IDelimitersAndSeparator _delim;

// Constructor
public ProcessCounter(IDelimitersAndSeparator delim)
{
   _delim = delim;

   // ProcessXxx Task 1:
   RegexOptions reo = RegexOptions.Singleline | RegexOptions.IgnoreCase;
   string pattern = string.Format(
      @"({0})\s*Counter\s*::(?<name>:?[^{0}{1}:]+(:[^{0}{1}:]+)*:?)" +
      @"(::\s*(?<extras>(init)|(next)|(previous)|" +
      @"((?<op>[=+-])\s*(?<direction>[+-])?\s*(?<val>[0-9]+)))?)?\s*({1})",
         _delim.OpenDelimEquivalent, _delim.CloseDelimEquivalent);
   _reCounter = new Regex(pattern, reo);
}
```

In order to complete the ***ProcessCounter*** class coding we need to handle the ***IProcessEvaluate*** interface—***ProcessXxx*** Task 2. The callback will look like:

```
public void Evaluate(object src, EnhancedStringEventArgs ea)
{
   ea.IsHandled = false;
   string text = ea.EhancedPairElem.Value;
```

```
        if (string.IsNullOrWhiteSpace(text)) return;                                5
        bool rc = _reCounter.IsMatch(text);                                         6
        if (!rc) return;                                                            7
        string replacement = _reCounter.Replace(text, CounterReplace);              8
        if (replacement == text) return;                                            9
        ea.IsHandled = true;                                                       10
        ea.EhancedPairElem.Value = replacement;                                    11
    }                                                                              12
```

The ***Evaluate()*** method needs to adhere to the three tasks:

Evaluate Task 1: Read from the string construct passed in through the ***ea*** parameter. Line 4 above sets ***text*** from the ***ea.EhancedPairElem.Value*** and thereafter we treat and transform ***text***.

Evaluate Task 2: Setting the ***ea.IsHandled*** to **true** if the ***Evaluate()*** method evaluated the ***ea.EhancedPairElem.Value*** to a different value from its original value and to **false** otherwise. Line 3 initializes the ***ea.IsHandled*** value to **false** and line 10 sets the ***ea.IsHandled*** to **true** if transformation, to a different value, took place.

Evaluate Task 3: In case that the ***Evaluate()*** method evaluated the original text to a different value from its original ***ea.EhancedPairElem.Value***, then update the value pointed to by ***ea.EhancedPairElem.Value*** to the new value. See line 11.

Possibility 2

This possibility uses the substitution method for the double colon separator, just like we did when handling random length delimiters. In which case, our constructor will look just a tad bit different (differences between the two possibilities are yellow highlighted):

```
    // Class instance variables
    private readonly Regex _reCounter;
    private readonly IDelimitersAndSeparator _delim;

    // Constructor
    public ProcessCounter(IDelimitersAndSeparator delim)
    {
        _delim = delim;

        // ProcessXxx Task 1:
        RegexOptions reo = RegexOptions.Singleline | RegexOptions.IgnoreCase;
        string pattern = string.Format(
            @"({0})\s*Counter\s*{2}(?<name>[^{0}{1}{2}]+)" +
            @"({2}\s*(?<extras>(init)|(next)|(previous)|" +
            @"((?<op>[=+-])\s*(?<direction>[+-])?\s*(?<val>[0-9]+)))?)?\s*({1})",
                _delim.OpenDelimEquivalent, _delim.CloseDelimEquivalent,
                _delim.SeparatorAlternate);
        _reCounter = new Regex(pattern, reo);
    }
```

and the implementation of the ***IProcessEvaluate*** interface—***ProcessXxx*** Task 2, will be like so:

```
    public void Evaluate(object src, EnhancedStringEventArgs ea)               1
    {                                                                          2
        ea.IsHandled = false;                                                  3
        string text = ea.EhancedPairElem.Value;                                4
        if (string.IsNullOrWhiteSpace(text)) return;                           5
        string preText = text.Replace(_delim.Separator, _delim.SeparatorAlternate); 6
        bool rc = _reCounter.IsMatch(preText);                                 7
        if (!rc) return;                                                       8
        string replacement = _reCounter.Replace(preText, CounterReplace);      9
        if (replacement == preText) return;                                   10
        ea.IsHandled = true;                                                  11
        string postText =
            replacement.Replace(_delim.SeparatorAlternate, _delim.Separator);      12
        ea.EhancedPairElem.Value = postText;                                  13
        return;                                                               14
    }                                                                         15
```

The three tasks of the ***Evaluate()*** method:

Evaluate Task 1: Read from the string construct passed in through the ***ea*** parameter. Line 4 above sets ***text*** to the ***ea.EhancedPairElem.Value*** and thereafter we treat and transform ***text***.

Evaluate Task 2: Setting the ***ea.IsHandled*** to **true** if the ***Evaluate()*** method evaluated the ***ea.EhancedPairElem.Value*** to a different value from its original value and to **false** otherwise. See line 3 where we initialize the ***ea.IsHandled*** value to **false** and line 11 where we set the ***ea.IsHandled*** to **true** if translation took place.

Evaluate Task 3: In case that the ***Evaluate()*** method evaluated the original text to a different value from its original ***ea.EhancedPairElem.Value***, then update the value pointed to by ***ea.EhancedPairElem.Value*** to the new value. See line 13.

Transformation of the string-construct

The transformation of ***{Counter::Name}*** to 0, 1, 2, … happens within the following line of code:

```
        string replacement = _reCounter.Replace(preText, CounterReplace);
```

(see line 8 within possibility 1 and line 9 within possibility 2). The transformation happens within the second parameter of the ***_reCounter.Replace*** method. This second parameter, ***CounterReplace***, is an instance of a delegate with a signature of ***MatchEvaluator***. The ***MatchEvaluator*** delegate is defined in the ***System.Text.RegularExpression*** namespace and it looks as follows:

```
    [Serializable]
    public delegate string MatchEvaluator(Match match);
```

The ***CounterReplace*** method represents the actual workhorse of the transformation. I do not delve into it, as it is straight forward and harbors no clever pitfalls. **This is what you will need to write for your own *{Identifier::Value}* construct.** Everything else is just about boiler-plate code.

Colon problems revisited—a second opinion

When matching, for our ***Name*** substring, our regular expression pattern for “a string that does not contain a double colon separator” our first choice was to match on a string where “a colon cannot precede a colon”; for which we came up with a solution: “`(?<name>:?[^{}:]+(:[^{}:]+)*:?)`”. This solution had no problem in dealing with a trailing colon in the name. The substitution method, on the other hand, does not handle a trailing-colon so nicely.

Say we have ***{Counter::Page:::Next}*** as our construct to be evaluated. When the system replaces the double colon to a single character (say Unicode character 3) then our string will be transformed to: ***{Counter\u003Page\u0003:Next}***. The transformation will take the first occurrence of a double colon and transform it to a \u003 leaving a name of the counter as “***Page***” rather than “***Page:***” and the operation as “***:Next***” rather than “***Next***”.

Now let’s think about this issue before rushing to a solution. If we succeed the name with a space then the name on longer ends with a colon and all is well. You may decide to set a rule that: “it is not allowed to end the name with a colon”. Lastly, if for whatever business reason you must have a name end with a colon and you are hell-bent on using the substitution way of dealing with a double colon in the name, then the following is a possible solution.

Employ the ***PreEvaluate()*** and ***PostEvaluate()*** methods of the ***EnhancedStringEval*** library class to overcome the problem. (I will explain the ***PreEvaluate()*** and ***PostEvaluate()*** methods when I discuss the library.) For now keep in mind that the library’s ***Evaluate()*** method has a ***Pre/PostEvaluate()*** methods surrounding it. The way around the problem is to detect when the name ends with trailing colon, substitute the trailing colon only with another character—within the ***PreEvaluate()*** method, say \u0004. Then perform the evaluation—***EvaluateString()***. Finally, replace the remaining \u0004, if exists, with a colon—in the ***PoseEvaluate()*** method. See TestMethod ***TestCounter2*** in the ***EnhancedStringEvaluateTest*** class.

The freedom of choice

Another issue that begs to be addressed is which of the two possibilities should you employ in your ***ProcessXxx***? Possibility 1 using the string that knows internally how to differentiate between a single and double colon like so “`:?[^:]+(:[^:]+)*:?`” or possibility 2 using substitution of a single character for a string and employing the regular expression language capabilities to exclude that single character.

My take is that when the separator string is as simple as a double colon then the added complexity, of handling a single colon within the name, is manageable and is confined to the regular expression pattern only. This complexity is confined to the constructor only and the rest of the class is not aware of changes to the match pattern.

On the other hand when we employ the substitution solution, we need to be vigilant over the fact that we need to transform the string, in the ***Evaluate()*** method back and forth. See the yellow highlights within the code of possibility 2.

To summarize: We should not ignore the fact that we may not need to handle a single colon within the name—restricting a colon within the name part of the recognition pattern may be a good option. If we do need to recognize a single colon, in the name part, and if we deal with a separator that is as simple as a double colon then I will be happy to confine the complexity to the regular expression pattern (in the constructor). On the other hand, if we deal with a separator that is more complex than a double colon then I will not hesitate and employ the substitution solution.

I believe that you are ready to develop a ***ProcessXxx*** of your own. If you feel that you need to see more examples then see the various ***ProcessXxx*** classes in the ***TestEvaluation*** Assembly under the ***ProcessEvaluate*** folder.

Transition

We are done with the ***ProcessXxx*** explanation by looking through the ***ProcessCounter*** class. If you needed to get up and running "quickly" then you have read and studied all you need.

Hereafter, we will look at the program as a whole; see how the library operates and how the "outside" world, the client, interacts with the library.

Meet the Players

ProcessXxx: We have just seen it implementing the ***IProcessEvaluate*** interface. The ***ProcessXxx*** "knows" how to transform a particular ***{Identifier::Value}***.

EnhancedStringEval library: The set of classes in the ***EnhancedStringEvaluate*** assembly, driven by the ***EnhancedStringEval*** class. The ***EnhancedStringEval*** class is responsible for calling the various ***ProcessXxx*** callback methods until no further transformations can occur. The connectivity here is through the interface ***IProcessEvalute*** that promises an implementation of the ***Evalute()*** method.

Client: The routine that calls the library's ***EnhancedStringEval***'s ***EvaluateString()*** method. The Client in the accompanying program example is in the file ***Program.cs*** within the ***TestEvaluation*** assembly part of the ***Main*** static method. Other examples can be found in the ***EvaluateSampleTest*** assembly, in the ***EnhancedStringEvaluateTest*** class. To exemplify the client's call:

```
//
// Client code
//

// First ProcessXxx class (definition)
var currDir = new ProcessCurrentDir();
currDir.CurrentDir = @"C:\Accounting";

// Define a container for ProcessXxx's
var context = new List<IProcessEvaluate>();

// Add 2 ProcessXxx classes
context.Add(currDir);
context.Add(new ProcessCounter());

// Client instantiates the EnhancedStringEval library and then calls it.
var eval = new EnhancedStringEval(context);
```

```
// Evaluate a string using the ProcessXxx's passed in the constructor
string dir1 = eval.EvaluateString("{Counter::Dir}. {CurrentDir::}");
// dir1 == "0. C:\Accounting"
```

The EnhancedStringEval Library Class

Its purpose is to provide a method signature for the client routine to pass a string containing ***{Indetifier::Value}***s. Its signature is:

```
public virtual string EvaluateString(string text)
```

The library will defer processing the various ***{Identifier::Value}***s to the various ***ProcessXxx***s' ***Evaluate()*** methods passed through to it, then return back to the Client the evaluated string.

Dependency Injection (DI) is a natural choice for our case. We will pass the individual ***ProcessXxx*** classes, in a bundle, into the ***EnhancedStringEval*** constructor—"constructor injection". DI (Dependency Injection) is also referred to in literature as IOC (Inversion of Control). Both names refer to the same design pattern. DI allows us to handle such situations where we know the method's interface, but do not know the details of the implementation. DI is a pattern where we "inject" an unknown class implementation into our ***EnhancedStringEval*** class library. Where "inject" refers to passing a class instance to our ***EnhancedStringEval*** class library where the library does not know its implementation and only knows its interface signature.

This is a skimpy description of DI. I assume that you either know of the DI design pattern already or you can read about it elsewhere. (One favorite book of mine for design patterns is "C# 3.0 Design Patterns" by Judith Bishop an O'Reilly publication.) In the grand scheme of things you need not know of the DI design pattern in order to understand the code or the article.

OnEvaluateContext

The class, ***EnhancedStringEval***, relies on the callback method promised by the ***IProcessEvaluate***—***Evaluate()***. The event callback handler is ***OnEvaluateContext*** that lives within the ***EnhancedStringEval*** class and is populated in the constructor:

```
// Constructor
public EnhancedStringEval(
   IEnumerable<IProcessEvaluate> context, IDelimitersAndSeparator delim)
{
   _delim = delim;

   if (context != null)
      foreach (IProcessEvaluate processXxx in context)
         if (processXxx != null)
            OnEvaluateContext += processXxx.Evaluate;
}
```

Or else you may use the built in event add/remove operators ("+=" and "-=") to add/remove an ***Evaluate*** callback methods. Therefore the Client has two options now.

Choice 1:

```
var context = new List<IProcessEvaluate>();
context.Add(new ProcessCounter());
var eval = new EnhancedStringEval(context);
string c1 = eval.EvaluateString("Counter1: {Counter::CounterName1}");
```

The variable ***context*** is the class that is injected into the ***EnhancedStringEval*** class.

Choice 2: You may also add the ***IProcessEvaluate***'s ***Evaluate()*** methods one at a time, like so:

```
var eval = new EnhancedStringEval();
eval.OnEvaluateContext += new ProcessCounter().Evaluate;
string c1 = eval.EvaluateString("Counter1: {Counter::CounterName1}");
```

Again we are faced with a choice and my take is that if you can use choice 1, constructor injection, then use it. However, if for whatever reason, you need the flexibility to add/remove ***IProcessEvaluate***'s ***Evaluate()*** midstream then augment choice 2 to choice 1.

The Client will call one of the two routines of the ***EnhancedStringEval*** class library: ***EvaluateString()*** or ***EvaluateStrings()***, singular or plural routine names. These two methods are similar but serve different purposes and hence the difference in names. We will start our discussion by looking at the ***EvaluateString()*** method.

EvaluateString—singular name

The ***EvaluateString()*** method itself looks like so:

```
Public virtual string EvaluateString(string text)
{
    string preText = PreEvaluate(text);
    string balanceText = BalancePreEvaluate(preText);

    string evalText = EvaluateStringPure(balanceText);

    string postText = PostEvaluate(evalText);
    return postText;
}
```

The ***PreEvaluate()*** method allows the Client to perform some actions on the input text, before the evaluation takes place. The ***PreEvaluate()*** method has a balancing: ***PostEvaluate()*** method; both have a pass-through default implementation. Both of these methods are virtual methods and as such in order to make use of these methods you will need to derive a class from ***EnhancedStringEval*** and override the ***Pre/PostEvaluate()*** methods. We have already seen an example of such a ***Pre/PoseEvaluate()*** methods use in the processing the trailing colon, see ***TestCounter2()*** method in the ***EnhancedStringEvaluateTest*** class. I provided an additional example of such overridden ***PreEvaluate(..)*** method in the ***EvaluateSampleTest*** assembly, see the implementation and use of class ***EnhancedStringEvalNoyyyMMdd*** in the ***TestSpecialHandlingOfDate()*** method.

The ***BalancePreEvaluate(..)*** method is designed to protect the transformation from an imbalanced open/close delimiters. The default implementation of the ***BalancePreEvaluate(..)*** method is to throw an exception, ***EnhancedStringException***, if the open/close delimiters are not balanced. The ***BalancePreEvaluate(..)*** method is not virtual.

The heart of the ***EvaluateString(..)*** method is the ***EvaluateStringPure(..)*** method call, which looks as follows:

```
private string EvaluateStringPure(string text)                              1
{                                                                           2
    if (OnEvaluateContext == null) return text;                             3

    var textElem = new EnhancedStrPairElement(TEMPKEY, _delim.PreMatch(text));   4
    var ea = new EnhancedStringEventArgs(textElem);                         5

    bool bHandled = false;                                                  6
    Delegate[] context = OnEvaluateContext.GetInvocationList();             7
    for (int i = 0; i < PassThroughUpperLimit; ++i)                         8
    {                                                                       9
        bHandled = false;                                                   10
        if (context != null)                                                11
        {                                                                   12
            foreach (Delegate processXxx in context)                        13
            {                                                               14
                try { processXxx.DyncdamicInvoke(new object[] { this, ea }); }   15
                catch { /* Error logging is appropriate here */ }           16
                if (ea.IsHandled)                                           17
                {                                                           18
                    bHandled = true;                                        19
                    string val = _delim.PreMatch(ea.EhancedPairElem.Value);  20
                    string preText = PreEvaluate(val);                      21
                    string balanceText = BalancePreEvaluate(preText);       22
                    textElem = new EnhancedStrPairElement(TEMPKEY, balanceText);   23
                    ea = new EnhancedStringEventArgs(textElem);             24
                }                                                           25
            }                                                               26
        }                                                                   27

        if (!bHandled) break;                                               28
    }                                                                       29

    return _delim.PostMatch(ea.EhancedPairElem.Value);                      30
}                                                                           31
```

Line 15 is a call to one of the ***ProcessXxx***'s ***Evaluate()*** methods. The ***OnEvaluateContext***, line 7, contains all the ***ProcessXxx***'s ***Evaluate()*** methods (passed through in the constructor) and assigns all of them to the variable ***context*** which the loop in line 13 cycles through.

Lines 4-5 set up the ***ea*** an ***EnhancedStringEventArgs*** variable, this variable will be the one passed through to the ***ProcessXxx***'s ***Evaluate*** method, in line 15. This same ***ea*** variable will be updated again, in line 24, if the ***Evaluate()*** delegate changed its original value when evaluated in line 15.

Keeping in mind that a transformed ***{Identifier::Value}*** construct may contain new ***{Identifier::Value}*** constructs, the method goes through a double **for** loop. An outer for loop, in line 8, makes sure that we are done processing all the ***{Identifier::Value}*** constructs that we possibly can. While an inner **for** loop, in line 13, cycles through the ***ProcessXxx***'s ***Evaluate()*** delegates, attempting to resolve a single ***{Identifier::Value}*** constructs within the ***string text***, passed in as a parameter.

The, ***EvaluateStringPure***, routine ends when either nothing was handled in processing within the inner loop, or the number of iterations through the outer loop reaches a maximum upper limit. Reaching a maximum upper limit in the outer loop is likely to be an infinite loop or infinite recursion.

PreEvaluate before every transformation and PostEvaluate after every transformation

At times you will need to repeat one or both of the ***Pre/PostEvaluate*** methods. These are the same steps we called prior to calling the ***EvaluateStringPure(..)*** method, see lines 18 – 25. To exemplify the need to repeat the ***Pre/PostEvaluate(..)*** methods call within the ***if (ea.IsHandled)*** body (lines 18 – 25), consider the scenario in the ***TestSpcialHandlingOfDate()*** method of the ***EnhancedStringEvaluateTest*** class within the ***EvaluateSampleTest*** assembly. This class takes a nested set constructs delimited with (“***#{***”, “***}#***”) and transforms the delimiters to (“***${***”, “***}$***”). Since the constructs are nested there is a need to run the ***Pre/PostEvaluate*** method after every transformation.

PreEvaluate and PostEvaluate evaluated once

At times you will need to run through the ***Pre/PostEvaluate*** methods once rather than after every transformation. Say, for example, that you need to run a construct like: ***${c}*** for calendar date allowing for addition/subtraction of days; so yesterday will be like so: ***${c-1}***. What we need is to transform the ***${c}*** construct to ***{CurrentTime::MM/dd/yyyy}*** and ***${c-1}*** to ***{CurrentTime::MM/dd/yyyy::-1d}*** only then pass the text to the ***EvaluateString(..)*** method. See the test method ***PrePostEvaluate()*** part of the ***EnhancedStringEvaluateTest*** class.

In order to accomplish such a feat we derive a class from the ***EvaluateStringEval*** class as follows:

```
   sealed internal class PrePostEvaluteOnce : EnhancedStringEval
   {
      public PrePostEvaluateOnce(IEnumerable<IProcessEvaluate> context)
         : base(context)
      {
         _reSpeicial = new Regex(
            @"\$\{c((?<direction>[-+])(?<amount>\d+))?\}",
            RegexOptions.Singleline | RegexOptions.IgnoreCase);
      }

      public override string EvaluateString(string text)
      {
         string preText = MyPreEvaluate(text);
         return base.EvaluateString(preText);
      }

      private string MyPreEvaluate(string text)
      {
         Match m = _reSpeicial.Match(text);
         if (!m.Success) return text;

         return _reSpeicial.Replace(text, me => CalendarReplace(me));
      }
      ...
}
```

We override the ***EvaluateString()*** method and perform the transformation before we call the ***EvaluateStringEval***'s ***EvaluateString()***: ***base.EvaluateString(preText)*** method.

Then using it as follows, from the client:

```
var context = new List<IProcessEvaluate>();
context.Add(new ProcessCurrentTime());
EnhancedStringEval proxy = new PrePostEvaluateOnce(context);
string evaluatedString = proxy.EvaluateString(
          "Today, ${c}, every man should exceed his grasp");
```

We instantiate the derived ***PrePostEvaluateOnce*** class and assign it to a base instance ***EnhancedStringEval*** then use it "as usual".

EvaluateStrings—plural name

This routine is used for preprocessing a collection of values, an optimization routine. See for example the ***ProcessKey*** class that handles a construct like: ***{Key::Value}***. The ***ProcessKey*** class handles a collection of (key, value) elements. A good example is the handling of values in a configuration file, like: ***app.config***. In a configuration file we have a list of Key=Value lines. So if we have entries like:

```
<add key="Flat" value="testing" />
<add key="Static flat" value="Static evaluation of flat: {key::flat}" />
```

Then the ***{key::flat}*** construct is intended to produce: "testing". Therefore, ***{key::Static flat}*** is intended to produce: "Static evaluation of flat: testing". The ***EvaluateStrings*** comes to resolve constructs like ***{key::flat}*** once, as opposed to resolving those keys at every invocation of ***{key::Static flat}***.

ProcessKey, a ***ProcessXxx*** class, has a constructor like so:

```
public ProcessKey(IDictionary<string, string> pairs, IDelimitersAndSeparator delim)
{
  _delim = delim;
  _pairEntries = new Dictionary<string, EnhancedStrPairElement>();
  EnahancedPairs = pairs;
  RegexOptions reo = RegexOptions.Singleline | RegexOptions.IgnoreCase;
  string pattern = string.Format(
    @"({0})\s*Key\s*::(?<Name>([^{0}{1}])*?)({1})",
    delim.OpenDelimEquivalent, delim.CloseDelimEquivalent);
  _reKey = new Regex(pattern, reo);
  ResolveKeys();
}
```

Where ***_delim***, ***_pairEntries***, and ***_reKey*** are class instance variables. The last method call in the constructor, ***ResolveKeys()***, is as follows:

```
private void ResolveKeys()
{
    var eval = new EnhancedStringEval(
                 new List<IProcessEvaluate> { this }, _delim);
    eval.EvaluateStrings(_pairEntries);
}
```

Where the last method call is: ***eval.EvaluateStrings(_pairEntries)***–plural .

EvaluateStrings:

```
Public virtual void EvaluateStrings(
                IDictionary<string, EnhancedStrPairElement> enhStrPairs)
{
    PreEvaluate(enhStrPairs);
    BalancePreEvaluate(enhStrPairs);

    EvaluateStringsPure(enhStrPairs);

    PostEvaluate(enhStrPairs);
}
```

Notice the parallelism between the ***EvaluateString()*** method and the ***EvaluateStrings()*** method. This parallelism is inevitable, both routines perform a very similar task; the one operates on a single string while the other operates on a dictionary collection. The heart of the ***EvaluateStrings()*** method is ***EvaluateStringsPure()***:

```
private void EvaluateStringsPure(
        IDictionary<string, EnhancedStrPairElement> enhStrPairs)
{
    if (OnEvaluateContext == null) return;
    var links = new LinkedList<EnhancedStrPairElement>();
    var pairNodes = from elem in enhStrPairs
        where _delim.IsSimpleExpression(elem.Value.Value) select elem.Value;
    foreach (EnhancedStrPairElement pairNode in pairNodes)
        links.AddLast(pairNode);
    if (links.Count == 0) return;

    for (int i = 0; i < PassThroughUpperLimit; ++i)
    {
        LinkedListNode<EnhancedStrPairElement> linkNode = links.First;
        while (linkNode != null)
        {
            bool bEval = EvalSimpleExpression(linkNode.Value);
            if (!bEval)
            {
                LinkedListNode<EnhancedStrPairElement> p = linkNode.Next;
                links.Remove(linkNode);
                linkNode = p;
            }
            else
            {
                PreEvaluate(enhStrPairs);
                BalancePreEvaluate(enhStrPairs);
                linkNode = linkNode.Next;
            }
        }

        if (links.Count == 0) break;
    }
}
```

The routine starts by cycling through constructs that contain simple expressions and sets those simple expressions into a linked list:

```
        var links = new LinkedList<EnhancedStrPairElement>();
        var pairNodes = from elem in enhStrPairs
           where _delim.IsSimpleExpression(elem.Value.Value) select elem.Value;
        foreach (EnhancedStrPairElement pairNode in pairNodes)
           links.AddLast(pairNode);
```

The routine relies on the ***IDelimitersAndSeparator***'s routine ***IsSimpleExpression()*** in order to determine if an expression contains a simple-expression or not.

Then, when a link was transformed, that link will have a new value. Ultimately when we pass the Value through the ***EvalSimpleExpression(linkNode.Value)*** call and it yields no change to the original Value, then the link is removed.

Where the ***EvaluateStringPure*** (singular name) stops when the inner loop evaluated nothing, this ***EvaluateStringsPure*** (plural name) keeps track of simple expressions in a linked list, a link is taken out when it cannot be transformed, so the routine as a whole stops when there are no links to keep on processing.

Other important classes/interfaces

EnhancedStringEventArgs

Going back to the ***IProcessEvaluate*** interface, we see that the second argument of the ***Evaluate()*** method is of type ***EnhancedStringEventArgs*** . This type wraps the ***EnhancedStrPairElement*** type into an ***EventArgs*** derived object. The ***EnhancedStrPairElement*** type comes in order to match the identifier like ***"Counter"*** in a case-insensitive manner. Making ***{Counter::value}***, ***{counter::value}*** and ***{COUNTER::value}*** equivalent constructs.

Writing code of your own

Examining the numerous examples that accompany the article you will find that the examples follow the following steps before evaluation:

- Instantiate and populate a context (a ***List<IProcessEvaluate>*** construct).
- Instantiate an ***EnhancedStringEval*** class using the previously constructed context.
- Lastly using the instance of the ***EnhancedStringEval*** to evaluate a string.

In your own project, more than likely, you will have a small number of contexts for your solution that you use over and over. In order not to allocate a new ***List<IprocessEvaluate>*** and not to instantiate an ***EnhancedStringEval*** for every evaluation, you may care to create a small number of singleton classes, one singleton class for each context that you will use, like so:

```
  public sealed class TransformConfiguration
  {
     private readonly EnhancedStringEval _eval;
     public static readonly Inst = new TransformConfiguration();

     private TransformConfiguration()
     {
         var context = new List<IProcessEvaluate>();
         context.Add(new ProcessDate());
         context.Add(new ProcessKey(config));
         context.Add(..);
         context...
         _eval = new EnhancedStringEval(context);
     }

     public string EvaluateString(string text)
     {
         return _eval.EvaluateString(text);
     }
  }
```

Hereafter, when it comes to evaluating a string using the above "configuration" context, you will evaluate a string like so: ***TransformConfiguration.Inst.EvaluateString(..)***, no need to instantiate the ***EnhancedStringEval*** class or to rebuild the ***context***.

Where do we go from here

The process can be enhanced in a few ways:

- A few of the ***ProcessXxx***, like ***ProcessIf***, can benefit from a grammar handling. An example of a promising grammar can be found in http://www.codeproject.com/KB/recipes/grammar_support_1.aspx "**Parsing Expression Grammar Support for C# 3.0 Part 1 - PEG Lib and Parser Generator"** By Martin Holzherr.
- The specifications as they stand now will not allow for a delimiter as part of the value. So for example: ***{identifier::value containing an open or close brace}*** will not pass the brace matching check. This is not of theoretical interest only. For example if we would like to write a ***{Decrypt::encrypted value}*** decryption construct—if we cannot guarantee that the encrypted value has neither an open nor close braces then we cannot write such a **ProcessDecrypt** class. This problem is not necessarily confined to the case of a single charactered delimiter because the multi charactered delimiters are transformed into single charactered delimiters.

Enjoy!

Avi

The class diagram was done using “Visual Diagram for UML Community Edition”