LinksPlatform's Platform.RegularExpressions.Transformer Class Library

1.1 ./csharp/Platform.RegularExpressions.Transformer/Context.cs

```
#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.RegularExpressions.Transformer
{
    public class Context : IContext
    {
        public string Path { get; }

        public Context(string path) => Path = path;
    }
}
```

1.2 ./csharp/Platform.RegularExpressions.Transformer/IContext.cs

```
#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.RegularExpressions.Transformer
{
    public interface IContext
    {
        public string Path { get; }
    }
}
```

1.3 ./csharp/Platform.RegularExpressions.Transformer/ISubstitutionRule.cs

```
using System.Runtime.CompilerServices;
using System.Text.RegularExpressions;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.RegularExpressions.Transformer
{
    public interface ISubstitutionRule
    {
        Regex MatchPattern
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
        }

        string SubstitutionPattern
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
        }

        Regex PathPattern
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
        }

        int MaximumRepeatCount
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
        }
    }
}
```

1.4 ./csharp/Platform.RegularExpressions.Transformer/ITransformer.cs

```
#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.RegularExpressions.Transformer
{
    public interface ITransformer
    {
        string Transform(string source, IContext context);
    }
}
```

1.5 ./csharp/Platform.RegularExpressions.Transformer/RegexExtensions.cs

```
using System;
using System.Text.RegularExpressions;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.RegularExpressions.Transformer
```

```
{
    public static class RegexExtensions
    {
        public static Regex OverrideOptions(this Regex regex, RegexOptions options, TimeSpan matchTimeout)
        {
            if (regex == null)
            {
                return null;
            }
            return new Regex(regex.ToString(), options, matchTimeout);
        }
    }
}
```

1.6 ./csharp/Platform.RegularExpressions.Transformer/SubstitutionRule.cs

```
using System;
using System.Text;
using System.Text.RegularExpressions;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.RegularExpressions.Transformer
{
    public class SubstitutionRule : ISubstitutionRule
    {
        public static readonly TimeSpan DefaultMatchTimeout = TimeSpan.FromMinutes(5);
        public static readonly RegexOptions DefaultMatchPatternRegexOptions = RegexOptions.Compiled | RegexOptions.Multiline;
        public static readonly RegexOptions DefaultPathPatternRegexOptions = RegexOptions.Compiled | RegexOptions.Singleline;

        public Regex MatchPattern { get; set; }

        public string SubstitutionPattern { get; set; }

        public Regex PathPattern { get; set; }

        public int MaximumRepeatCount { get; set; }

        public SubstitutionRule(Regex matchPattern, string substitutionPattern, Regex pathPattern, int maximumRepeatCount, RegexOptions? matchPatternOptions, RegexOptions? pathPatternOptions, TimeSpan? matchTimeout)
        {
            MatchPattern = matchPattern;
            SubstitutionPattern = substitutionPattern;
            PathPattern = pathPattern;
            MaximumRepeatCount = maximumRepeatCount;
            OverrideMatchPatternOptions(matchPatternOptions ?? matchPattern.Options, matchTimeout ?? matchPattern.MatchTimeout);
            OverridePathPatternOptions(pathPatternOptions ?? pathPattern.Options, matchTimeout ?? pathPattern.MatchTimeout);
        }

        public SubstitutionRule(Regex matchPattern, string substitutionPattern, Regex pathPattern, int maximumRepeatCount, bool useDefaultOptions) : this(matchPattern, substitutionPattern, pathPattern, maximumRepeatCount, useDefaultOptions ? DefaultMatchPatternRegexOptions : (RegexOptions?)null, useDefaultOptions ? DefaultPathPatternRegexOptions : (RegexOptions?)null, useDefaultOptions ? DefaultMatchTimeout : (TimeSpan?)null) { }

        public SubstitutionRule(Regex matchPattern, string substitutionPattern, Regex pathPattern, int maximumRepeatCount) : this(matchPattern, substitutionPattern, pathPattern, maximumRepeatCount, true) { }

        public SubstitutionRule(Regex matchPattern, string substitutionPattern, int maximumRepeatCount) : this(matchPattern, substitutionPattern, null, maximumRepeatCount) { }

        public SubstitutionRule(Regex matchPattern, string substitutionPattern) : this(matchPattern, substitutionPattern, null, 0) { }

        public static implicit operator SubstitutionRule(ValueTuple<string, string> tuple) => new SubstitutionRule(new Regex(tuple.Item1), tuple.Item2);

        public static implicit operator SubstitutionRule(ValueTuple<Regex, string> tuple) => new SubstitutionRule(tuple.Item1, tuple.Item2);

```

```
        public static implicit operator SubstitutionRule(ValueTuple<string, string, int> tuple)
        ↪  => new SubstitutionRule(new Regex(tuple.Item1), tuple.Item2, tuple.Item3);

        public static implicit operator SubstitutionRule(ValueTuple<Regex, string, int> tuple)
        ↪  => new SubstitutionRule(tuple.Item1, tuple.Item2, tuple.Item3);

        public static implicit operator SubstitutionRule(ValueTuple<string, string, Regex, int>
        ↪  tuple) => new SubstitutionRule(new Regex(tuple.Item1), tuple.Item2, tuple.Item3,
        ↪  tuple.Item4);

        public static implicit operator SubstitutionRule(ValueTuple<Regex, string, Regex, int>
        ↪  tuple) => new SubstitutionRule(tuple.Item1, tuple.Item2, tuple.Item3, tuple.Item4);

        public void OverrideMatchPatternOptions(RegexOptions options, TimeSpan matchTimeout) =>
        ↪  MatchPattern = MatchPattern.OverrideOptions(options, matchTimeout);

        public void OverridePathPatternOptions(RegexOptions options, TimeSpan matchTimeout) =>
        ↪  PathPattern = PathPattern.OverrideOptions(options, matchTimeout);

        public override string ToString()
        {
            var sb = new StringBuilder();
            sb.Append('"');
            sb.Append(MatchPattern.ToString());
            sb.Append('"');
            sb.Append(" -> ");
            sb.Append('"');
            sb.Append(SubstitutionPattern);
            sb.Append('"');
            if (PathPattern != null)
            {
                sb.Append(" on files ");
                sb.Append('"');
                sb.Append(PathPattern.ToString());
                sb.Append('"');
            }
            if (MaximumRepeatCount > 0)
            {
                if (MaximumRepeatCount >= int.MaxValue)
                {
                    sb.Append(" repeated forever");
                }
                else
                {
                    sb.Append(" repeated up to ");
                    sb.Append(MaximumRepeatCount);
                    sb.Append(" times");
                }
            }
            return sb.ToString();
        }
    }
}
```

1.7 ./csharp/Platform.RegularExpressions.Transformer/Transformer.cs

```
using System.Collections.Generic;
using System.Linq;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.RegularExpressions.Transformer
{
    public class Transformer : ITransformer
    {
        private readonly IList<ISubstitutionRule> _substitutionRules;

        public Transformer(IList<ISubstitutionRule> substitutionRules) => _substitutionRules =
        ↪  substitutionRules;

        public string Transform(string source, IContext context)
        {
            var current = source;
            var currentFilePath = context?.Path ?? "";
            for (var i = 0; i < _substitutionRules.Count; i++)
            {
                var rule = _substitutionRules[i];
                var matchPattern = rule.MatchPattern;
                var substitutionPattern = rule.SubstitutionPattern;
                var pathPattern = rule.PathPattern;
```

```
                var maximumRepeatCount = rule.MaximumRepeatCount;
                if (pathPattern == null || pathPattern.IsMatch(currentFilePath))
                {
                    var replaceCount = 0;
                    do
                    {
                        current = matchPattern.Replace(current, substitutionPattern);
                        replaceCount++;
                        if (maximumRepeatCount < int.MaxValue && replaceCount > maximumRepeatCount)
                        {
                            break;
                        }
                    }
                    while (matchPattern.IsMatch(current));
                }
            }
            return current;
        }

        public IList<ITransformer> GenerateTransformersForEachRulesStep()
        {
            var transformers = new List<ITransformer>();
            for (int i = 1; i <= _substitutionRules.Count; i++)
            {
                transformers.Add(new Transformer(_substitutionRules.Take(i).ToList()));
            }
            return transformers;
        }
    }
}
```

1.8 ./csharp/Platform.RegularExpressions.Transformer/TransformerCLI.cs

```
using System.Diagnostics;
using System.IO;
using System.Text;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.RegularExpressions.Transformer
{
    public class TransformerCLI
    {
        private readonly ITransformer _transformer;

        public TransformerCLI(ITransformer transformer) => _transformer = transformer;

        public bool Run(string[] args, out string message)
        {
            message = "";
            var sourcePath = GetArgOrDefault(args, 0);
            if (!File.Exists(sourcePath))
            {
                message = $"{sourcePath} file does not exist.";
                return false;
            }
            var targetPath = GetArgOrDefault(args, 1);
            if (string.IsNullOrWhiteSpace(targetPath))
            {
                targetPath = ChangeToTargetExtension(sourcePath);
            }
            else if (Directory.Exists(targetPath) && File.GetAttributes(targetPath).HasFlag(FileAttributes.Directory))
            {
                targetPath = Path.Combine(targetPath, GetTargetFileName(sourcePath));
            }
            else if (LooksLikeDirectoryPath(targetPath))
            {
                Directory.CreateDirectory(targetPath);
                targetPath = Path.Combine(targetPath, GetTargetFileName(sourcePath));
            }
            if (File.Exists(targetPath))
            {
                var applicationPath = Process.GetCurrentProcess().MainModule.FileName;
                var targetFileLastUpdateDateTime = new FileInfo(targetPath).LastWriteTimeUtc;
                if (new FileInfo(sourcePath).LastWriteTimeUtc < targetFileLastUpdateDateTime &&
                    new FileInfo(applicationPath).LastWriteTimeUtc < targetFileLastUpdateDateTime)
                {
```

```
                    return true;
                }
            }
            File.WriteAllText(targetPath, _transformer.Transform(File.ReadAllText(sourcePath,
            ↪  Encoding.UTF8), new Context(sourcePath)), Encoding.UTF8);
            message = $"{targetPath} file written.";
            return true;
        }

        private static string GetTargetFileName(string sourcePath) =>
        ↪  ChangeToTargetExtension(Path.GetFileName(sourcePath));

        private static string ChangeToTargetExtension(string path) => Path.ChangeExtension(path,
        ↪  ".cpp");

        private static bool LooksLikeDirectoryPath(string targetPath) =>
        ↪  targetPath.EndsWith(Path.DirectorySeparatorChar.ToString()) ||
        ↪  targetPath.EndsWith(Path.AltDirectorySeparatorChar.ToString());

        private static string GetArgOrDefault(string[] args, int index) => args.Length > index ?
        ↪  args[index] : null;
    }
}
```

1.9 ./csharp/Platform.RegularExpressions.Transformer/TransformerExtensions.cs

```
using System.Collections.Generic;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.RegularExpressions.Transformer
{
    public static class TransformerExtensions
    {
        public static List<string> GetSteps(this Transformer transformer, string source) =>
        ↪  transformer.GenerateTransformersForEachRulesStep().TransformWithAll(source);

        public static void WriteStepsToFiles(this Transformer transformer, string sourcePath,
        ↪  string targetFilename, string targetExtension, bool skipFilesWithNoChanges) => trans
        ↪  former.GenerateTransformersForEachRulesStep().TransformWithAllToFiles(sourcePath,
        ↪  targetFilename, targetExtension, skipFilesWithNoChanges);
    }
}
```

1.10 ./csharp/Platform.RegularExpressions.Transformer/TransformersListExtensions.cs

```
using System.IO;
using System.Collections.Generic;
using System.Text;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.RegularExpressions.Transformer
{
    public static class TransformersListExtensions
    {
        public static List<string> TransformWithAll(this IList<ITransformer> transformers,
        ↪  string source)
        {
            var strings = new List<string>();
            if (transformers.Count > 0)
            {
                for (int i = 0; i < transformers.Count; i++)
                {
                    strings.Add(transformers[i].Transform(source, null));
                }
            }
            return strings;
        }

        public static void TransformWithAllToFiles(this IList<ITransformer> transformers, string
        ↪  sourcePath, string targetFilename, string targetExtension, bool
        ↪  skipFilesWithNoChanges)
        {
            if (transformers.Count > 0)
            {
                var lastText = "";
                var sourceText = File.ReadAllText(sourcePath, Encoding.UTF8);
                var transformerContext = new Context(sourcePath);
                for (int i = 0; i < transformers.Count; i++)
```

```
                {
                    var transformationOutput = transformers[i].Transform(sourceText,
                    ↪  transformerContext);
                    if (!(skipFilesWithNoChanges && string.Equals(lastText,
                    ↪  transformationOutput)))
                    {
                        lastText = transformationOutput;
                        File.WriteAllText($"{targetFilename}.{i}{targetExtension}",
                        ↪  transformationOutput, Encoding.UTF8);
                    }
                }
            }
        }
    }
}
```

1.11 ./csharp/Platform.RegularExpressions.Transformer.Tests/MarkovAlgorithmsTests.cs

```
using System.Text.RegularExpressions;
using Xunit;

namespace Platform.RegularExpressions.Transformer.Tests
{
    public class MarkovAlgorithmsTests
    {
        /// <remarks>
        /// Example is from https://en.wikipedia.org/wiki/Markov_algorithm.
        /// </remarks>
        [Fact]
        public void BinaryToUnaryNumbersTest()
        {
            var rules = new SubstitutionRule[]
            {
                ("1", "0|", int.MaxValue),      // "1" -> "0|" repeated forever
                // | symbol should be escaped for regular expression pattern, but not in the
                ↪  substitution pattern
                (@"\|0", "0||", int.MaxValue), // "\|0" -> "0||" repeated forever
                ("0", "", int.MaxValue),        // "0" -> "" repeated forever
            };
            var transformer = new Transformer(rules);
            var input = "101";
            var expectedOutput = "|||||";
            var output = transformer.Transform(input, null);
            Assert.Equal(expectedOutput, output);
        }
    }
}
```

1.12 ./csharp/Platform.RegularExpressions.Transformer.Tests/SubstitutionRuleTests.cs

```
using System.Text.RegularExpressions;
using Xunit;

namespace Platform.RegularExpressions.Transformer.Tests
{
    public class SubstitutionRuleTests
    {
        [Fact]
        public void OptionsOverrideTest()
        {
            SubstitutionRule rule = (new Regex(@"^\s*?\#pragma[\sa-zA-Z0-9\/]+$"), "", null, 0);
            Assert.Equal(RegexOptions.Compiled | RegexOptions.Multiline,
            ↪  rule.MatchPattern.Options);
        }
    }
}
```

1.13 ./csharp/Platform.RegularExpressions.Transformer.Tests/TransformersTests.cs

```
using System.IO;
using System.Text;
using System.Text.RegularExpressions;
using Xunit;

namespace Platform.RegularExpressions.Transformer.Tests
{
    public class TransformersTests
    {
        [Fact]
        public void DebugOutputTest()
        {
```

```
            var sourceText = "aaaa";
            var firstStepReferenceText = "bbbb";
            var secondStepReferenceText = "cccc";

            var transformer = new Transformer(new SubstitutionRule[] {
                (new Regex("a"), "b"),
                (new Regex("b"), "c")
            });

            var steps = transformer.GetSteps(sourceText);

            Assert.Equal(2, steps.Count);
            Assert.Equal(firstStepReferenceText, steps[0]);
            Assert.Equal(secondStepReferenceText, steps[1]);
        }

        [Fact]
        public void DebugFilesOutputTest()
        {
            var sourceText = "aaaa";
            var firstStepReferenceText = "bbbb";
            var secondStepReferenceText = "cccc";

            var sourceFilename = Path.GetTempFileName();
            File.WriteAllText(sourceFilename, sourceText, Encoding.UTF8);

            var transformer = new Transformer(new SubstitutionRule[] {
                (new Regex("a"), "b"),
                (new Regex("b"), "c")
            });

            var targetFilename = Path.GetTempFileName();

            transformer.WriteStepsToFiles(sourceFilename, targetFilename, ".txt",
            ↪  skipFilesWithNoChanges: false);

            var firstStepReferenceFilename = $"{targetFilename}.0.txt";
            var secondStepReferenceFilename = $"{targetFilename}.1.txt";

            Assert.True(File.Exists(firstStepReferenceFilename));
            Assert.True(File.Exists(secondStepReferenceFilename));

            Assert.Equal(firstStepReferenceText, File.ReadAllText(firstStepReferenceFilename,
            ↪  Encoding.UTF8));
            Assert.Equal(secondStepReferenceText, File.ReadAllText(secondStepReferenceFilename,
            ↪  Encoding.UTF8));

            File.Delete(sourceFilename);
            File.Delete(firstStepReferenceFilename);
            File.Delete(secondStepReferenceFilename);
        }

        [Fact]
        public void FilesWithNoChangesSkipedTest()
        {
            var sourceText = "aaaa";
            var firstStepReferenceText = "bbbb";
            var thirdStepReferenceText = "cccc";

            var sourceFilename = Path.GetTempFileName();
            File.WriteAllText(sourceFilename, sourceText, Encoding.UTF8);

            var transformer = new Transformer(new SubstitutionRule[] {
                (new Regex("a"), "b"),
                (new Regex("x"), "y"),
                (new Regex("b"), "c")
            });

            var targetFilename = Path.GetTempFileName();

            transformer.WriteStepsToFiles(sourceFilename, targetFilename, ".txt",
            ↪  skipFilesWithNoChanges: true);

            var firstStepReferenceFilename = $"{targetFilename}.0.txt";
            var secondStepReferenceFilename = $"{targetFilename}.1.txt";
            var thirdStepReferenceFilename = $"{targetFilename}.2.txt";

            Assert.True(File.Exists(firstStepReferenceFilename));
            Assert.False(File.Exists(secondStepReferenceFilename));
```

```
88              Assert.True(File.Exists(thirdStepReferenceFilename));
89
90              Assert.Equal(firstStepReferenceText, File.ReadAllText(firstStepReferenceFilename,
                ↪  Encoding.UTF8));
91              Assert.Equal(thirdStepReferenceText, File.ReadAllText(thirdStepReferenceFilename,
                ↪  Encoding.UTF8));
92
93              File.Delete(sourceFilename);
94              File.Delete(firstStepReferenceFilename);
95              File.Delete(secondStepReferenceFilename);
96              File.Delete(thirdStepReferenceFilename);
97          }
98      }
99  }
```

Index