

LinksPlatform's Platform.RegularExpressions.Transformer Class Library

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

```
1  using System;
2  using System.Collections.Generic;
3  using System.Diagnostics;
4  using System.IO;
5  using System.Runtime.CompilerServices;
6  using System.Text;
7  using System.Threading.Tasks;
8
9  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
10
11 namespace Platform.RegularExpressions.Transformer
12 {
13     public class FileTransformer : IFileTransformer
14     {
15         protected readonly ITextTransformer _textTransformer;
16
17         public string SourceFileExtension
18         {
19             [MethodImpl(MethodImplOptions.AggressiveInlining)]
20             get;
21             [MethodImpl(MethodImplOptions.AggressiveInlining)]
22             private set;
23         }
24
25         public string TargetFileExtension
26         {
27             [MethodImpl(MethodImplOptions.AggressiveInlining)]
28             get;
29             [MethodImpl(MethodImplOptions.AggressiveInlining)]
30             private set;
31         }
32
33         public IList<ISubstitutionRule> Rules
34         {
35             [MethodImpl(MethodImplOptions.AggressiveInlining)]
36             get => _textTransformer.Rules;
37         }
38
39         [MethodImpl(MethodImplOptions.AggressiveInlining)]
40         public FileTransformer(ITextTransformer textTransformer, string sourceFileExtension,
41             → string targetFileExtension)
42         {
43             _textTransformer = textTransformer;
44             SourceFileExtension = sourceFileExtension;
45             TargetFileExtension = targetFileExtension;
46         }
47
48         [MethodImpl(MethodImplOptions.AggressiveInlining)]
49         public void Transform(string sourcePath, string targetPath)
50         {
51             var sourceDirectoryExists = DirectoryExists(sourcePath);
52             var sourceDirectoryPath = LooksLikeDirectoryPath(sourcePath);
53             var sourceIsDirectory = sourceDirectoryExists || sourceDirectoryPath;
54             var targetDirectoryExists = DirectoryExists(targetPath);
55             var targetDirectoryPath = LooksLikeDirectoryPath(targetPath);
56             var targetIsDirectory = targetDirectoryExists || targetDirectoryPath;
57             if (sourceIsDirectory && targetIsDirectory)
58             {
59                 // Folder -> Folder
60                 if (!sourceDirectoryExists)
61                 {
62                     return;
63                 }
64                 TransformFolder(sourcePath, targetPath);
65             }
66             else if (!(sourceIsDirectory || targetIsDirectory))
67             {
68                 // File -> File
69                 EnsureSourceFileExists(sourcePath);
70                 EnsureTargetFileDirectoryExists(targetPath);
71                 TransformFile(sourcePath, targetPath);
72             }
73             else if (targetIsDirectory)
74             {
75                 // File -> Folder
76                 EnsureSourceFileExists(sourcePath);
77                 EnsureTargetDirectoryExists(targetPath, targetDirectoryExists);
78                 TransformFile(sourcePath, GetTargetFileName(sourcePath, targetPath));
79             }
80         }
81     }
82 }
```

```

78     }
79     else
80     {
81         // Folder -> File
82         throw new NotSupportedException();
83     }
84 }
85
86 [MethodImpl(MethodImplOptions.AggressiveInlining)]
87 protected virtual void TransformFolder(string sourcePath, string targetPath)
88 {
89     if (CountFilesRecursively(sourcePath, SourceFileExtension) == 0)
90     {
91         return;
92     }
93     EnsureTargetDirectoryExists(targetPath);
94     var directories = Directory.GetDirectories(sourcePath);
95     for (var i = 0; i < directories.Length; i++)
96     {
97         #if NETSTANDARD2_1
98             var relativePath = Path.GetRelativePath(sourcePath, directories[i]);
99         #else
100             var relativePath = directories[i].Replace(sourcePath.TrimEnd('\\') + "\\ ", "");
101         #endif
102         var newTargetPath = Path.Combine(targetPath, relativePath);
103         TransformFolder(directories[i], newTargetPath);
104     }
105     var files = Directory.GetFiles(sourcePath);
106     Parallel.For(0, files.Length, i =>
107     {
108         var file = files[i];
109         if (FileExtensionMatches(file, SourceFileExtension))
110         {
111             TransformFile(file, GetTargetFileName(file, targetPath));
112         }
113     });
114 }
115
116 [MethodImpl(MethodImplOptions.AggressiveInlining)]
117 protected virtual void TransformFile(string sourcePath, string targetPath)
118 {
119     if (File.Exists(targetPath))
120     {
121         var applicationPath = Process.GetCurrentProcess().MainModule.FileName;
122         var targetFileLastUpdateDateTime = new FileInfo(targetPath).LastWriteTimeUtc;
123         if (new FileInfo(sourcePath).LastWriteTimeUtc < targetFileLastUpdateDateTime &&
124             ↪ new FileInfo(applicationPath).LastWriteTimeUtc <
125             ↪ targetFileLastUpdateDateTime)
126         {
127             return;
128         }
129     }
130     var sourceText = File.ReadAllText(sourcePath, Encoding.UTF8);
131     var targetText = _textTransformer.Transform(sourceText);
132     File.WriteAllText(targetPath, targetText, Encoding.UTF8);
133 }
134
135 [MethodImpl(MethodImplOptions.AggressiveInlining)]
136 protected string GetTargetFileName(string sourcePath, string targetDirectory) =>
137     ↪ Path.ChangeExtension(Path.Combine(targetDirectory, Path.GetFileName(sourcePath)),
138     ↪ TargetFileExtension);
139
140 [MethodImpl(MethodImplOptions.AggressiveInlining)]
141 private static long CountFilesRecursively(string path, string extension)
142 {
143     var files = Directory.GetFiles(path);
144     var directories = Directory.GetDirectories(path);
145     var result = 0L;
146     for (var i = 0; i < directories.Length; i++)
147     {
148         result += CountFilesRecursively(directories[i], extension);
149     }
150     for (var i = 0; i < files.Length; i++)
151     {
152         if (FileExtensionMatches(files[i], extension))
153         {
154             result++;
155         }
156     }
157 }

```

```

152     }
153     return result;
154 }
155
156 [MethodImpl(MethodImplOptions.AggressiveInlining)]
157 private static bool FileExtensionMatches(string file, string extension) =>
158     ↪ file.EndsWith(extension, StringComparison.OrdinalIgnoreCase);
159
160 [MethodImpl(MethodImplOptions.AggressiveInlining)]
161 private static void EnsureTargetFileDirectoryExists(string targetPath)
162 {
163     if (!File.Exists(targetPath))
164     {
165         EnsureDirectoryIsCreated(targetPath);
166     }
167 }
168
169 [MethodImpl(MethodImplOptions.AggressiveInlining)]
170 private static void EnsureTargetDirectoryExists(string targetPath) =>
171     ↪ EnsureTargetDirectoryExists(targetPath, DirectoryExists(targetPath));
172
173 [MethodImpl(MethodImplOptions.AggressiveInlining)]
174 private static void EnsureTargetDirectoryExists(string targetPath, bool
175     ↪ targetDirectoryExists)
176 {
177     if (!targetDirectoryExists)
178     {
179         Directory.CreateDirectory(targetPath);
180     }
181 }
182
183 [MethodImpl(MethodImplOptions.AggressiveInlining)]
184 private static void EnsureSourceFileExists(string sourcePath)
185 {
186     if (!File.Exists(sourcePath))
187     {
188         throw new FileNotFoundException("Source file does not exists.", sourcePath);
189     }
190 }
191
192 [MethodImpl(MethodImplOptions.AggressiveInlining)]
193 private static void EnsureDirectoryIsCreated(string targetPath) =>
194     ↪ Directory.CreateDirectory(Path.GetDirectoryName(targetPath));
195
196 [MethodImpl(MethodImplOptions.AggressiveInlining)]
197 private static bool DirectoryExists(string path) => Directory.Exists(path) &&
198     ↪ File.GetAttributes(path).HasFlag(FileAttributes.Directory);
199
200 [MethodImpl(MethodImplOptions.AggressiveInlining)]
201 private static bool LooksLikeDirectoryPath(string path) =>
202     ↪ path.EndsWith(Path.DirectorySeparatorChar.ToString()) ||
203     ↪ path.EndsWith(Path.AltDirectorySeparatorChar.ToString());
204 }
205 }

```

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

```

1 using System.Runtime.CompilerServices;
2
3 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5 namespace Platform.RegularExpressions.Transformer
6 {
7     public interface IFileTransformer : ITransformer
8     {
9         string SourceFileExtension
10         {
11             [MethodImpl(MethodImplOptions.AggressiveInlining)]
12             get;
13         }
14
15         string TargetFileExtension
16         {
17             [MethodImpl(MethodImplOptions.AggressiveInlining)]
18             get;
19         }
20
21         [MethodImpl(MethodImplOptions.AggressiveInlining)]
22         void Transform(string sourcePath, string targetPath);
23     }
24 }

```

```
24 }
```

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

```
1 using System.Runtime.CompilerServices;
2 using System.Text.RegularExpressions;
3
4 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6 namespace Platform.RegularExpressions.Transformer
7 {
8     public interface ISubstitutionRule
9     {
10         Regex MatchPattern
11         {
12             [MethodImpl(MethodImplOptions.AggressiveInlining)]
13             get;
14         }
15
16         string SubstitutionPattern
17         {
18             [MethodImpl(MethodImplOptions.AggressiveInlining)]
19             get;
20         }
21
22         int MaximumRepeatCount
23         {
24             [MethodImpl(MethodImplOptions.AggressiveInlining)]
25             get;
26         }
27     }
28 }
```

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

```
1 using System.Runtime.CompilerServices;
2
3 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5 namespace Platform.RegularExpressions.Transformer
6 {
7     public interface ITextTransformer : ITransformer
8     {
9         [MethodImpl(MethodImplOptions.AggressiveInlining)]
10         string Transform(string sourceText);
11     }
12 }
```

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

```
1 using System;
2 using System.Collections.Generic;
3 using System.Linq;
4 using System.Runtime.CompilerServices;
5 using Platform.Collections;
6
7 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
8
9 namespace Platform.RegularExpressions.Transformer
10 {
11     public static class ITextTransformerExtensions
12     {
13         [MethodImpl(MethodImplOptions.AggressiveInlining)]
14         public static IList<ITextTransformer> GenerateTransformersForEachRule(this
15             ↪ ITextTransformer transformer)
16         {
17             var transformers = new List<ITextTransformer>();
18             for (int i = 1; i <= transformer.Rules.Count; i++)
19             {
20                 transformers.Add(new TextTransformer(transformer.Rules.Take(i).ToList()));
21             }
22             return transformers;
23         }
24
25         [MethodImpl(MethodImplOptions.AggressiveInlining)]
26         public static IList<string> GetSteps(this ITextTransformer transformer, string
27             ↪ sourceText)
28         {
29             if (transformer != null && !transformer.Rules.IsNullOrEmpty())
30             {
31                 var steps = new List<string>();
32                 var steppedTransformer = new TextSteppedTransformer(transformer.Rules,
33                     ↪ sourceText);
```

```

31         while (steppedTransformer.Next())
32         {
33             steps.Add(steppedTransformer.Text);
34         }
35         return steps;
36     }
37     else
38     {
39         return Array.Empty<string>();
40     }
41 }
42
43 [MethodImpl(MethodImplOptions.AggressiveInlining)]
44 public static void WriteStepsToFiles(this ITextTransformer transformer, string
    ↳ sourceText, string targetPath, bool skipFilesWithNoChanges)
45 {
46     if (transformer != null && !transformer.Rules.IsNullOrEmpty())
47     {
48         targetPath.GetPathParts(out var directoryName, out var targetFilename, out var
    ↳ targetExtension);
49         Steps.DeleteAllSteps(directoryName, targetFilename, targetExtension);
50         var lastText = "";
51         var steppedTransformer = new TextSteppedTransformer(transformer.Rules,
    ↳ sourceText);
52         while (steppedTransformer.Next())
53         {
54             var newText = steppedTransformer.Text;
55             Steps.WriteStep(transformer, directoryName, targetFilename, targetExtension,
    ↳ steppedTransformer.Current, ref lastText, newText,
    ↳ skipFilesWithNoChanges);
56         }
57     }
58 }
59 }
60 }

```

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

```

1  using System;
2  using System.Collections.Generic;
3  using System.Runtime.CompilerServices;
4  using Platform.Collections;
5
6  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8  namespace Platform.RegularExpressions.Transformer
9  {
10     public static class ITextTransformersListExtensions
11     {
12         [MethodImpl(MethodImplOptions.AggressiveInlining)]
13         public static IList<string> TransformWithAll(this IList<ITextTransformer> transformers,
    ↳ string source)
14         {
15             if (!transformers.IsNullOrEmpty())
16             {
17                 var steps = new List<string>();
18                 for (int i = 0; i < transformers.Count; i++)
19                 {
20                     steps.Add(transformers[i].Transform(source));
21                 }
22                 return steps;
23             }
24             else
25             {
26                 return Array.Empty<string>();
27             }
28         }
29
30         [MethodImpl(MethodImplOptions.AggressiveInlining)]
31         public static void TransformWithAllToFiles(this IList<ITextTransformer> transformers,
    ↳ string sourceText, string targetPath, bool skipFilesWithNoChanges)
32         {
33             if (!transformers.IsNullOrEmpty())
34             {
35                 targetPath.GetPathParts(out var directoryName, out var targetFilename, out var
    ↳ targetExtension);
36                 Steps.DeleteAllSteps(directoryName, targetFilename, targetExtension);
37                 var lastText = "";
38                 for (int i = 0; i < transformers.Count; i++)

```

```

39         {
40             var transformer = transformers[i];
41             var newText = transformer.Transform(sourceText);
42             Steps.WriteStep(transformer, directoryName, targetFilename, targetExtension,
43                 ↪ i, ref lastText, newText, skipFilesWithNoChanges);
44         }
45     }
46 }
47 }

```

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

```

1 using System.Collections.Generic;
2 using System.Runtime.CompilerServices;
3
4 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6 namespace Platform.RegularExpressions.Transformer
7 {
8     public interface ITransformer
9     {
10         IList<ISubstitutionRule> Rules
11         {
12             [MethodImpl(MethodImplOptions.AggressiveInlining)]
13             get;
14         }
15     }
16 }

```

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

```

1 using System.IO;
2 using System.Runtime.CompilerServices;
3 using System.Text;
4
5 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7 namespace Platform.RegularExpressions.Transformer
8 {
9     public class LogFileTransformer : FileTransformer
10     {
11         [MethodImpl(MethodImplOptions.AggressiveInlining)]
12         public LogFileTransformer(ITextTransformer textTransformer, string
13             ↪ sourceFileExtension, string targetFileExtension) : base(textTransformer,
14             ↪ sourceFileExtension, targetFileExtension) { }
15
16         [MethodImpl(MethodImplOptions.AggressiveInlining)]
17         protected override void TransformFile(string sourcePath, string targetPath)
18         {
19             base.TransformFile(sourcePath, targetPath);
20             // Logging
21             var sourceText = File.ReadAllText(sourcePath, Encoding.UTF8);
22             _textTransformer.WriteStepsToFiles(sourceText, targetPath, skipFilesWithNoChanges:
23                 ↪ true);
24         }
25     }
26 }

```

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

```

1 using System;
2 using System.Runtime.CompilerServices;
3 using System.Text.RegularExpressions;
4
5 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7 namespace Platform.RegularExpressions.Transformer
8 {
9     public static class RegexExtensions
10     {
11         [MethodImpl(MethodImplOptions.AggressiveInlining)]
12         public static Regex OverrideOptions(this Regex regex, RegexOptions options, TimeSpan
13             ↪ matchTimeout)
14         {
15             if (regex == null)
16             {
17                 return null;
18             }
19             return new Regex(regex.ToString(), options, matchTimeout);
20         }
21     }
22 }

```

```

20     }
21 }

```

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

```

1  using Platform.IO;
2  using System.Runtime.CompilerServices;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.RegularExpressions.Transformer
7  {
8      public static class Steps
9      {
10         [MethodImpl(MethodImplOptions.AggressiveInlining)]
11         public static void DeleteAllSteps(string directoryName, string targetFilename, string
            ↪ targetExtension)
12         {
13             FileHelpers.DeleteAll(directoryName, $"{targetFilename}.*.rule.txt");
14             FileHelpers.DeleteAll(directoryName, $"{targetFilename}.{targetExtension}");
15         }
16
17         [MethodImpl(MethodImplOptions.AggressiveInlining)]
18         public static void WriteStep(ITransformer transformer, string directoryName, string
            ↪ targetFilename, string targetExtension, int currentStep, ref string lastText, string
            ↪ newText, bool skipFilesWithNoChanges)
19         {
20             if (!(skipFilesWithNoChanges && string.Equals(lastText, newText)))
21             {
22                 lastText = newText;
23                 newText.WriteToFile(directoryName,
24                     ↪ $"{targetFilename}.{currentStep}{targetExtension}");
25                 var ruleString = transformer.Rules[currentStep].ToString();
26                 ruleString.WriteToFile(directoryName,
27                     ↪ $"{targetFilename}.{currentStep}.rule.txt");
28             }
29         }
30     }
31 }

```

1.11 ./csharp/Platform.RegularExpressions.Transformer/StringExtensions.cs

```

1  using System.IO;
2  using System.Runtime.CompilerServices;
3  using System.Text;
4
5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7  namespace Platform.RegularExpressions.Transformer
8  {
9      internal static class StringExtensions
10     {
11         [MethodImpl(MethodImplOptions.AggressiveInlining)]
12         public static void GetPathParts(this string path, out string directoryName, out string
            ↪ targetFilename, out string targetExtension) => (directoryName, targetFilename,
            ↪ targetExtension) = (Path.GetDirectoryName(path),
            ↪ Path.GetFileNameWithoutExtension(path), Path.GetExtension(path));
13
14         [MethodImpl(MethodImplOptions.AggressiveInlining)]
15         public static void WriteToFile(this string text, string directoryName, string
            ↪ targetFilename) => File.WriteAllText(Path.Combine(directoryName, targetFilename),
            ↪ text, Encoding.UTF8);
16     }
17 }

```

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

```

1  using System;
2  using System.Runtime.CompilerServices;
3  using System.Text;
4  using System.Text.RegularExpressions;
5
6  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8  namespace Platform.RegularExpressions.Transformer
9  {
10     public class SubstitutionRule : ISubstitutionRule
11     {
12         public static readonly TimeSpan DefaultMatchTimeout = TimeSpan.FromMinutes(5);
13         public static readonly RegexOptions DefaultMatchPatternRegexOptions =
            ↪ RegexOptions.Compiled | RegexOptions.Multiline;

```

```

14
15 public Regex MatchPattern
16 {
17     [MethodImpl(MethodImplOptions.AggressiveInlining)]
18     get;
19     [MethodImpl(MethodImplOptions.AggressiveInlining)]
20     set;
21 }
22
23 public string SubstitutionPattern
24 {
25     [MethodImpl(MethodImplOptions.AggressiveInlining)]
26     get;
27     [MethodImpl(MethodImplOptions.AggressiveInlining)]
28     set;
29 }
30
31 public Regex PathPattern
32 {
33     [MethodImpl(MethodImplOptions.AggressiveInlining)]
34     get;
35     [MethodImpl(MethodImplOptions.AggressiveInlining)]
36     set;
37 }
38
39 public int MaximumRepeatCount
40 {
41     [MethodImpl(MethodImplOptions.AggressiveInlining)]
42     get;
43     [MethodImpl(MethodImplOptions.AggressiveInlining)]
44     set;
45 }
46
47 [MethodImpl(MethodImplOptions.AggressiveInlining)]
48 public SubstitutionRule(Regex matchPattern, string substitutionPattern, int
↳ maximumRepeatCount, RegexOptions? matchPatternOptions, TimeSpan? matchTimeout)
49 {
50     MatchPattern = matchPattern;
51     SubstitutionPattern = substitutionPattern;
52     MaximumRepeatCount = maximumRepeatCount;
53     OverrideMatchPatternOptions(matchPatternOptions ?? matchPattern.Options,
↳ matchTimeout ?? matchPattern.MatchTimeout);
54 }
55
56 [MethodImpl(MethodImplOptions.AggressiveInlining)]
57 public SubstitutionRule(Regex matchPattern, string substitutionPattern, int
↳ maximumRepeatCount, bool useDefaultOptions) : this(matchPattern,
↳ substitutionPattern, maximumRepeatCount, useDefaultOptions ?
↳ DefaultMatchPatternRegexOptions : (RegexOptions?)null, useDefaultOptions ?
↳ DefaultMatchTimeout : (TimeSpan?)null) { }
58
59 [MethodImpl(MethodImplOptions.AggressiveInlining)]
60 public SubstitutionRule(Regex matchPattern, string substitutionPattern, int
↳ maximumRepeatCount) : this(matchPattern, substitutionPattern, maximumRepeatCount,
↳ true) { }
61
62 [MethodImpl(MethodImplOptions.AggressiveInlining)]
63 public SubstitutionRule(Regex matchPattern, string substitutionPattern) :
↳ this(matchPattern, substitutionPattern, 0) { }
64
65 [MethodImpl(MethodImplOptions.AggressiveInlining)]
66 public static implicit operator SubstitutionRule(ValueTuple<string, string> tuple) =>
↳ new SubstitutionRule(new Regex(tuple.Item1), tuple.Item2);
67
68 [MethodImpl(MethodImplOptions.AggressiveInlining)]
69 public static implicit operator SubstitutionRule(ValueTuple<Regex, string> tuple) => new
↳ SubstitutionRule(tuple.Item1, tuple.Item2);
70
71 [MethodImpl(MethodImplOptions.AggressiveInlining)]
72 public static implicit operator SubstitutionRule(ValueTuple<string, string, int> tuple)
↳ => new SubstitutionRule(new Regex(tuple.Item1), tuple.Item2, tuple.Item3);
73
74 [MethodImpl(MethodImplOptions.AggressiveInlining)]
75 public static implicit operator SubstitutionRule(ValueTuple<Regex, string, int> tuple)
↳ => new SubstitutionRule(tuple.Item1, tuple.Item2, tuple.Item3);
76
77 [MethodImpl(MethodImplOptions.AggressiveInlining)]
78 public void OverrideMatchPatternOptions(RegexOptions options, TimeSpan matchTimeout) =>
↳ MatchPattern = MatchPattern.OverrideOptions(options, matchTimeout);

```



```

79     [MethodImpl(MethodImplOptions.AggressiveInlining)]
80     public void OverridePathPatternOptions(RegexOptions options, TimeSpan matchTimeout) =>
81     ↪ PathPattern = PathPattern.OverrideOptions(options, matchTimeout);
82
83     [MethodImpl(MethodImplOptions.AggressiveInlining)]
84     public override string ToString()
85     {
86         var sb = new StringBuilder();
87         sb.Append('');
88         sb.Append(MatchPattern.ToString());
89         sb.Append('');
90         sb.Append(" -> ");
91         sb.Append('');
92         sb.Append(SubstitutionPattern);
93         sb.Append('');
94         if (PathPattern != null)
95         {
96             sb.Append(" on files ");
97             sb.Append('');
98             sb.Append(PathPattern.ToString());
99             sb.Append('');
100         }
101         if (MaximumRepeatCount > 0)
102         {
103             if (MaximumRepeatCount >= int.MaxValue)
104             {
105                 sb.Append(" repeated forever");
106             }
107             else
108             {
109                 sb.Append(" repeated up to ");
110                 sb.Append(MaximumRepeatCount);
111                 sb.Append(" times");
112             }
113         }
114         return sb.ToString();
115     }
116 }
117 }

```

1.13 ./csharp/Platform.RegularExpressions.Transformer/TextSteppedTransformer.cs

```

1  using System;
2  using System.Collections.Generic;
3  using System.Runtime.CompilerServices;
4
5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7  namespace Platform.RegularExpressions.Transformer
8  {
9      public class TextSteppedTransformer : ITransformer
10     {
11         public IList<ISubstitutionRule> Rules
12         {
13             [MethodImpl(MethodImplOptions.AggressiveInlining)]
14             get;
15             [MethodImpl(MethodImplOptions.AggressiveInlining)]
16             set;
17         }
18
19         public string Text
20         {
21             [MethodImpl(MethodImplOptions.AggressiveInlining)]
22             get;
23             [MethodImpl(MethodImplOptions.AggressiveInlining)]
24             set;
25         }
26
27         public int Current
28         {
29             [MethodImpl(MethodImplOptions.AggressiveInlining)]
30             get;
31             [MethodImpl(MethodImplOptions.AggressiveInlining)]
32             set;
33         }
34
35         [MethodImpl(MethodImplOptions.AggressiveInlining)]
36         public TextSteppedTransformer(IList<ISubstitutionRule> rules, string text, int current)
37         ↪ => Reset(rules, text, current);

```

```

37     [MethodImpl(MethodImplOptions.AggressiveInlining)]
38     public TextSteppedTransformer(ICollection<ISubstitutionRule> rules, string text) =>
39         ↪ Reset(rules, text);
40
41     [MethodImpl(MethodImplOptions.AggressiveInlining)]
42     public TextSteppedTransformer(ICollection<ISubstitutionRule> rules) => Reset(rules);
43
44     [MethodImpl(MethodImplOptions.AggressiveInlining)]
45     public TextSteppedTransformer() => Reset();
46
47     [MethodImpl(MethodImplOptions.AggressiveInlining)]
48     public void Reset(ICollection<ISubstitutionRule> rules, string text, int current)
49     {
50         Rules = rules;
51         Text = text;
52         Current = current;
53     }
54
55     [MethodImpl(MethodImplOptions.AggressiveInlining)]
56     public void Reset(ICollection<ISubstitutionRule> rules, string text) => Reset(rules, text, -1);
57
58     [MethodImpl(MethodImplOptions.AggressiveInlining)]
59     public void Reset(ICollection<ISubstitutionRule> rules) => Reset(rules, "", -1);
60
61     [MethodImpl(MethodImplOptions.AggressiveInlining)]
62     public void Reset(string text) => Reset(Rules, text, -1);
63
64     [MethodImpl(MethodImplOptions.AggressiveInlining)]
65     public void Reset() => Reset(Array.Empty<ISubstitutionRule>(), "", -1);
66
67     [MethodImpl(MethodImplOptions.AggressiveInlining)]
68     public bool Next()
69     {
70         var current = Current + 1;
71         if (current >= Rules.Count)
72         {
73             return false;
74         }
75         var rule = Rules[current];
76         var matchPattern = rule.MatchPattern;
77         var substitutionPattern = rule.SubstitutionPattern;
78         var maximumRepeatCount = rule.MaximumRepeatCount;
79         var replaceCount = 0;
80         var text = Text;
81         do
82         {
83             text = matchPattern.Replace(text, substitutionPattern);
84             replaceCount++;
85         }
86         while ((maximumRepeatCount == int.MaxValue || replaceCount <= maximumRepeatCount) &&
87             ↪ matchPattern.IsMatch(text));
88         Text = text;
89         Current = current;
90         return true;
91     }
92 }

```

1.14 ./csharp/Platform.RegularExpressions.Transformer/TextTransformer.cs

```

1  using System.Collections.Generic;
2  using System.Runtime.CompilerServices;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.RegularExpressions.Transformer
7  {
8      public class TextTransformer : ITextTransformer
9      {
10         public ICollection<ISubstitutionRule> Rules
11         {
12             [MethodImpl(MethodImplOptions.AggressiveInlining)]
13             get;
14             [MethodImpl(MethodImplOptions.AggressiveInlining)]
15             private set;
16         }
17
18         [MethodImpl(MethodImplOptions.AggressiveInlining)]
19         public TextTransformer(ICollection<ISubstitutionRule> substitutionRules)
20         {

```

```

21     Rules = substitutionRules;
22 }
23
24 [MethodImpl(MethodImplOptions.AggressiveInlining)]
25 public string Transform(string source)
26 {
27     var baseTrasformer = new TextSteppedTransformer(Rules);
28     baseTrasformer.Reset(source);
29     while (baseTrasformer.Next());
30     return baseTrasformer.Text;
31 }
32 }
33 }

```

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

```

1  using System.Runtime.CompilerServices;
2  using Platform.Collections.Arrays;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.RegularExpressions.Transformer
7  {
8      public class TransformerCLI
9      {
10         private readonly IFileTransformer _transformer;
11
12         [MethodImpl(MethodImplOptions.AggressiveInlining)]
13         public TransformerCLI(IFileTransformer transformer) => _transformer = transformer;
14
15         [MethodImpl(MethodImplOptions.AggressiveInlining)]
16         public void Run(string[] args)
17         {
18             var sourcePath = args.GetElementOrDefault(0);
19             var targetPath = args.GetElementOrDefault(1);
20             _transformer.Transform(sourcePath, targetPath);
21         }
22     }
23 }

```

1.16 ./csharp/Platform.RegularExpressions.Transformer.Tests/FileTransformerTests.cs

```

1  using System.IO;
2  using Xunit;
3
4  namespace Platform.RegularExpressions.Transformer.Tests
5  {
6      public class FileTransformerTests
7      {
8          [Fact]
9          public void FolderToFolderTransformationTest()
10         {
11             var tempPath = Path.GetTempPath();
12             var sourceFolderPath = Path.Combine(tempPath,
13                 ↪ "FileTransformerTestsFolderToFolderTransformationTestSourceFolder");
14             var targetFolderPath = Path.Combine(tempPath,
15                 ↪ "FileTransformerTestsFolderToFolderTransformationTestTargetFolder");
16
17             var baseTransformer = new TextTransformer(new SubstitutionRule[]
18             {
19                 ("a", "b"),
20                 ("b", "c")
21             });
22             var fileTransformer = new FileTransformer(baseTransformer, ".cs", ".cpp");
23
24             // Delete before creation (if previous test failed)
25             if (Directory.Exists(sourceFolderPath))
26             {
27                 Directory.Delete(sourceFolderPath, true);
28             }
29             if (Directory.Exists(targetFolderPath))
30             {
31                 Directory.Delete(targetFolderPath, true);
32             }
33
34             Directory.CreateDirectory(sourceFolderPath);
35             Directory.CreateDirectory(targetFolderPath);
36
37             File.WriteAllText(Path.Combine(sourceFolderPath, "a.cs"), "a a a");
38             var aFolderPath = Path.Combine(sourceFolderPath, "A");
39             Directory.CreateDirectory(aFolderPath);

```

```

38     Directory.CreateDirectory(Path.Combine(sourceFolderPath, "B"));
39     File.WriteAllText(Path.Combine(aFolderPath, "b.cs"), "b b b");
40     File.WriteAllText(Path.Combine(sourceFolderPath, "x.txt"), "should not be
    ↪ translated");
41
42     fileTransformer.Transform(sourceFolderPath,
    ↪ $"{targetFolderPath}{Path.DirectorySeparatorChar}");
43
44     var aCppFile = Path.Combine(targetFolderPath, "a.cpp");
45     Assert.True(File.Exists(aCppFile));
46     Assert.Equal("c c c", File.ReadAllText(aCppFile));
47     Assert.True(Directory.Exists(Path.Combine(targetFolderPath, "A")));
48     Assert.False(Directory.Exists(Path.Combine(targetFolderPath, "B")));
49     var bCppFile = Path.Combine(targetFolderPath, "A", "b.cpp");
50     Assert.True(File.Exists(bCppFile));
51     Assert.Equal("c c c", File.ReadAllText(bCppFile));
52     Assert.False(File.Exists(Path.Combine(targetFolderPath, "x.txt")));
53     Assert.False(File.Exists(Path.Combine(targetFolderPath, "x.cpp")));
54
55     Directory.Delete(sourceFolderPath, true);
56     Directory.Delete(targetFolderPath, true);
57 }
58 }
59 }

```

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

```

1  using System.Text.RegularExpressions;
2  using Xunit;
3
4  namespace Platform.RegularExpressions.Transformer.Tests
5  {
6      public class MarkovAlgorithmsTests
7      {
8          /// <remarks>
9          /// Example is from https://en.wikipedia.org/wiki/Markov\_algorithm.
10         /// </remarks>
11         [Fact]
12         public void BinaryToUnaryNumbersTest()
13         {
14             var rules = new SubstitutionRule[]
15             {
16                 ("1", "0|", int.MaxValue), // "1" -> "0|" repeated forever
17                 // | symbol should be escaped for regular expression pattern, but not in the
18                 ↪ substitution pattern
19                 ("@" + "\\0", "0||", int.MaxValue), // "\\0" -> "0||" repeated forever
20                 ("0", "", int.MaxValue), // "0" -> "" repeated forever
21             };
22             var transformer = new TextTransformer(rules);
23             var input = "101";
24             var expectedOutput = "|||||";
25             var output = transformer.Transform(input);
26             Assert.Equal(expectedOutput, output);
27         }
28     }
29 }

```

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

```

1  using System.Text.RegularExpressions;
2  using Xunit;
3
4  namespace Platform.RegularExpressions.Transformer.Tests
5  {
6      public class SubstitutionRuleTests
7      {
8          [Fact]
9          public void OptionsOverrideTest()
10         {
11             SubstitutionRule rule = (new Regex(@"^s*?\#pragma[sa-zA-Z0-9\./]+$$"), "", 0);
12             Assert.Equal(RegexOptions.Compiled | RegexOptions.Multiline,
13                 ↪ rule.MatchPattern.Options);
14         }
15     }
16 }

```

1.19 ./csharp/Platform.RegularExpressions.Transformer.Tests/TextTransformerTests.cs

```

1  using System.IO;
2  using System.Text;
3  using System.Text.RegularExpressions;

```

```

4 using Xunit;
5
6 namespace Platform.RegularExpressions.Transformer.Tests
7 {
8     public class TextTransformerTests
9     {
10         [Fact]
11         public void DebugOutputTest()
12         {
13             var sourceText = "aaaa";
14             var firstStepReferenceText = "bbbb";
15             var secondStepReferenceText = "cccc";
16
17             var transformer = new TextTransformer(new SubstitutionRule[] {
18                 (new Regex("a"), "b"),
19                 (new Regex("b"), "c")
20             });
21
22             var steps = transformer.GetSteps(sourceText);
23
24             Assert.Equal(2, steps.Count);
25             Assert.Equal(firstStepReferenceText, steps[0]);
26             Assert.Equal(secondStepReferenceText, steps[1]);
27         }
28
29         [Fact]
30         public void DebugFilesOutputTest()
31         {
32             var sourceText = "aaaa";
33             var firstStepReferenceText = "bbbb";
34             var secondStepReferenceText = "cccc";
35
36             var transformer = new TextTransformer(new SubstitutionRule[] {
37                 (new Regex("a"), "b"),
38                 (new Regex("b"), "c")
39             });
40
41             var targetFilename = Path.GetTempFileName();
42
43             transformer.WriteStepsToFiles(sourceText, $"{targetFilename}.txt",
44                 ↪ skipFilesWithNoChanges: false);
45
46             CheckAndCleanUpTwoRulesFiles(firstStepReferenceText, secondStepReferenceText,
47                 ↪ transformer, targetFilename);
48
49             private static void CheckAndCleanUpTwoRulesFiles(string firstStepReferenceText, string
50                 ↪ secondStepReferenceText, TextTransformer transformer, string targetFilename)
51             {
52                 var firstStepReferenceFilename = $"{targetFilename}.0.txt";
53                 var firstStepRuleFilename = $"{targetFilename}.0.rule.txt";
54                 var secondStepReferenceFilename = $"{targetFilename}.1.txt";
55                 var secondStepRuleFilename = $"{targetFilename}.1.rule.txt";
56
57                 Assert.True(File.Exists(firstStepReferenceFilename));
58                 Assert.True(File.Exists(firstStepRuleFilename));
59                 Assert.True(File.Exists(secondStepReferenceFilename));
60                 Assert.True(File.Exists(secondStepRuleFilename));
61
62                 Assert.Equal(firstStepReferenceText, File.ReadAllText(firstStepReferenceFilename,
63                     ↪ Encoding.UTF8));
64                 Assert.Equal(transformer.Rules[0].ToString(),
65                     ↪ File.ReadAllText(firstStepRuleFilename, Encoding.UTF8));
66                 Assert.Equal(secondStepReferenceText, File.ReadAllText(secondStepReferenceFilename,
67                     ↪ Encoding.UTF8));
68                 Assert.Equal(transformer.Rules[1].ToString(),
69                     ↪ File.ReadAllText(secondStepRuleFilename, Encoding.UTF8));
70
71                 File.Delete(firstStepReferenceFilename);
72                 File.Delete(firstStepRuleFilename);
73                 File.Delete(secondStepReferenceFilename);
74                 File.Delete(secondStepRuleFilename);
75             }
76
77         [Fact]
78         public void FilesWithNoChangesSkippedTest()
79         {
80             var sourceText = "aaaa";
81             var firstStepReferenceText = "bbbb";

```

```

76     var thirdStepReferenceText = "cccc";
77
78     var transformer = new TextTransformer(new SubstitutionRule[] {
79         (new Regex("a"), "b"),
80         (new Regex("x"), "y"),
81         (new Regex("b"), "c")
82     });
83
84     var targetFilename = Path.GetTempFileName();
85
86     transformer.WriteStepsToFiles(sourceText, $"{targetFilename}.txt",
87         ↪ skipFilesWithNoChanges: true);
88
89     CheckAndCleanUpThreeRulesFiles(firstStepReferenceText, thirdStepReferenceText,
90         ↪ transformer, targetFilename);
91 }
92
93 private static void CheckAndCleanUpThreeRulesFiles(string firstStepReferenceText, string
94     ↪ thirdStepReferenceText, TextTransformer transformer, string targetFilename)
95 {
96     var firstStepReferenceFilename = $"{targetFilename}.0.txt";
97     var firstStepRuleFilename = $"{targetFilename}.0.rule.txt";
98     var secondStepReferenceFilename = $"{targetFilename}.1.txt";
99     var secondStepRuleFilename = $"{targetFilename}.1.rule.txt";
100     var thirdStepReferenceFilename = $"{targetFilename}.2.txt";
101     var thirdStepRuleFilename = $"{targetFilename}.2.rule.txt";
102
103     Assert.True(File.Exists(firstStepReferenceFilename));
104     Assert.True(File.Exists(firstStepRuleFilename));
105     Assert.False(File.Exists(secondStepReferenceFilename));
106     Assert.False(File.Exists(secondStepRuleFilename));
107     Assert.True(File.Exists(thirdStepReferenceFilename));
108     Assert.True(File.Exists(thirdStepRuleFilename));
109
110     Assert.Equal(firstStepReferenceText, File.ReadAllText(firstStepReferenceFilename,
111         ↪ Encoding.UTF8));
112     Assert.Equal(transformer.Rules[0].ToString(),
113         ↪ File.ReadAllText(firstStepRuleFilename, Encoding.UTF8));
114     Assert.Equal(thirdStepReferenceText, File.ReadAllText(thirdStepReferenceFilename,
115         ↪ Encoding.UTF8));
116     Assert.Equal(transformer.Rules[2].ToString(),
117         ↪ File.ReadAllText(thirdStepRuleFilename, Encoding.UTF8));
118
119     File.Delete(firstStepReferenceFilename);
120     File.Delete(firstStepRuleFilename);
121     File.Delete(secondStepReferenceFilename);
122     File.Delete(secondStepRuleFilename);
123     File.Delete(thirdStepReferenceFilename);
124     File.Delete(thirdStepRuleFilename);
125 }
126
127 [Fact]
128 public void DebugOutputUsingTransformersGenerationTest()
129 {
130     var sourceText = "aaaa";
131     var firstStepReferenceText = "bbbb";
132     var secondStepReferenceText = "cccc";
133
134     var transformer = new TextTransformer(new SubstitutionRule[] {
135         (new Regex("a"), "b"),
136         (new Regex("b"), "c")
137     });
138
139     var steps =
140         ↪ transformer.GenerateTransformersForEachRule().TransformWithAll(sourceText);
141
142     Assert.Equal(2, steps.Count);
143     Assert.Equal(firstStepReferenceText, steps[0]);
144     Assert.Equal(secondStepReferenceText, steps[1]);
145 }
146
147 [Fact]
148 public void DebugFilesOutputUsingTransformersGenerationTest()
149 {
150     var sourceText = "aaaa";
151     var firstStepReferenceText = "bbbb";
152     var secondStepReferenceText = "cccc";

```

```

146     var transformer = new TextTransformer(new SubstitutionRule[] {
147         (new Regex("a"), "b"),
148         (new Regex("b"), "c")
149     });
150
151     var targetFilename = Path.GetTempFileName();
152
153     transformer.GenerateTransformersForEachRule().TransformWithAllToFiles(sourceText,
154         ↪ $"{targetFilename}.txt", skipFilesWithNoChanges: false);
155
156     CheckAndCleanUpTwoRulesFiles(firstStepReferenceText, secondStepReferenceText,
157         ↪ transformer, targetFilename);
158 }
159
160 [Fact]
161 public void FilesWithNoChangesSkippedWhenUsingTransformersGenerationTest()
162 {
163     var sourceText = "aaaa";
164     var firstStepReferenceText = "bbbb";
165     var thirdStepReferenceText = "cccc";
166
167     var transformer = new TextTransformer(new SubstitutionRule[] {
168         (new Regex("a"), "b"),
169         (new Regex("x"), "y"),
170         (new Regex("b"), "c")
171     });
172
173     var targetFilename = Path.GetTempFileName();
174
175     transformer.GenerateTransformersForEachRule().TransformWithAllToFiles(sourceText,
176         ↪ $"{targetFilename}.txt", skipFilesWithNoChanges: true);
177
178     CheckAndCleanUpThreeRulesFiles(firstStepReferenceText, thirdStepReferenceText,
179         ↪ transformer, targetFilename);
180 }
181 }
182 }

```

Index

- ./csharp/Platform.RegularExpressions.Transformer.Tests/FileTransformerTests.cs, 11
- ./csharp/Platform.RegularExpressions.Transformer.Tests/MarkovAlgorithmsTests.cs, 12
- ./csharp/Platform.RegularExpressions.Transformer.Tests/SubstitutionRuleTests.cs, 12
- ./csharp/Platform.RegularExpressions.Transformer.Tests/TextTransformerTests.cs, 12
- ./csharp/Platform.RegularExpressions.Transformer/FileTransformer.cs, 1
- ./csharp/Platform.RegularExpressions.Transformer/IFileTransformer.cs, 3
- ./csharp/Platform.RegularExpressions.Transformer/ISubstitutionRule.cs, 4
- ./csharp/Platform.RegularExpressions.Transformer/ITextTransformer.cs, 4
- ./csharp/Platform.RegularExpressions.Transformer/ITextTransformerExtensions.cs, 4
- ./csharp/Platform.RegularExpressions.Transformer/ITextTransformersListExtensions.cs, 5
- ./csharp/Platform.RegularExpressions.Transformer/ITransformer.cs, 6
- ./csharp/Platform.RegularExpressions.Transformer/LoggingFileTransformer.cs, 6
- ./csharp/Platform.RegularExpressions.Transformer/RegexExtensions.cs, 6
- ./csharp/Platform.RegularExpressions.Transformer/Steps.cs, 7
- ./csharp/Platform.RegularExpressions.Transformer/StringExtensions.cs, 7
- ./csharp/Platform.RegularExpressions.Transformer/SubstitutionRule.cs, 7
- ./csharp/Platform.RegularExpressions.Transformer/TextSteppedTransformer.cs, 9
- ./csharp/Platform.RegularExpressions.Transformer/TextTransformer.cs, 10
- ./csharp/Platform.RegularExpressions.Transformer/TransformerCLI.cs, 11