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

```csharp
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.IO;
using System.Runtime.CompilerServices;
using System.Text;
using System.Threading.Tasks;

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

namespace Platform.RegularExpressions.Transformer
{
    /// <summary>
    /// <para>
    /// Represents the file transformer.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="IFileTransformer"/>
    public class FileTransformer : IFileTransformer
    {
        /// <summary>
        /// <para>
        /// The text transformer.
        /// </para>
        /// <para></para>
        /// </summary>
        protected readonly ITextTransformer _textTransformer;

        /// <summary>
        /// <para>
        /// Gets or sets the source file extension value.
        /// </para>
        /// <para></para>
        /// </summary>
        public string SourceFileExtension
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            private set;
        }

        /// <summary>
        /// <para>
        /// Gets or sets the target file extension value.
        /// </para>
        /// <para></para>
        /// </summary>
        public string TargetFileExtension
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            private set;
        }

        /// <summary>
        /// <para>
        /// Gets the rules value.
        /// </para>
        /// <para></para>
        /// </summary>
        public IList<ISubstitutionRule> Rules
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get => _textTransformer.Rules;
        }

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="FileTransformer"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="textTransformer">
        /// <para>A text transformer.</para>
```

```
 78          /// <para></para>
 79          /// </param>
 80          /// <param name="sourceFileExtension">
 81          /// <para>A source file extension.</para>
 82          /// <para></para>
 83          /// </param>
 84          /// <param name="targetFileExtension">
 85          /// <para>A target file extension.</para>
 86          /// <para></para>
 87          /// </param>
 88          [MethodImpl(MethodImplOptions.AggressiveInlining)]
 89          public FileTransformer(ITextTransformer textTransformer, string sourceFileExtension,
         ↪  string targetFileExtension)
 90          {
 91              _textTransformer = textTransformer;
 92              SourceFileExtension = sourceFileExtension;
 93              TargetFileExtension = targetFileExtension;
 94          }
 95
 96          /// <summary>
 97          /// <para>
 98          /// Transforms the source path.
 99          /// </para>
100          /// <para></para>
101          /// </summary>
102          /// <param name="sourcePath">
103          /// <para>The source path.</para>
104          /// <para></para>
105          /// </param>
106          /// <param name="targetPath">
107          /// <para>The target path.</para>
108          /// <para></para>
109          /// </param>
110          /// <exception cref="NotSupportedException">
111          /// <para></para>
112          /// <para></para>
113          /// </exception>
114          [MethodImpl(MethodImplOptions.AggressiveInlining)]
115          public void Transform(string sourcePath, string targetPath)
116          {
117              var defaultPath = Path.GetFullPath(".");
118              if (string.IsNullOrEmpty(sourcePath))
119              {
120                  sourcePath = defaultPath;
121              }
122              if (string.IsNullOrEmpty(targetPath))
123              {
124                  targetPath = defaultPath;
125              }
126              var sourceDirectoryExists = DirectoryExists(sourcePath);
127              var sourceDirectoryPath = LooksLikeDirectoryPath(sourcePath);
128              var sourceIsDirectory = sourceDirectoryExists || sourceDirectoryPath;
129              var targetDirectoryExists = DirectoryExists(targetPath);
130              var targetDirectoryPath = LooksLikeDirectoryPath(targetPath);
131              var targetIsDirectory = targetDirectoryExists || targetDirectoryPath;
132              if (sourceIsDirectory && targetIsDirectory)
133              {
134                  // Folder -> Folder
135                  if (!sourceDirectoryExists)
136                  {
137                      return;
138                  }
139                  TransformFolder(sourcePath, targetPath);
140              }
141              else if (!(sourceIsDirectory || targetIsDirectory))
142              {
143                  // File -> File
144                  EnsureSourceFileExists(sourcePath);
145                  EnsureTargetFileDirectoryExists(targetPath);
146                  TransformFile(sourcePath, targetPath);
147              }
148              else if (targetIsDirectory)
149              {
150                  // File -> Folder
151                  EnsureSourceFileExists(sourcePath);
152                  EnsureTargetDirectoryExists(targetPath, targetDirectoryExists);
153                  TransformFile(sourcePath, GetTargetFileName(sourcePath, targetPath));
154              }
```

```csharp
            else
            {
                // Folder -> File
                throw new NotSupportedException();
            }
        }

        /// <summary>
        /// <para>
        /// Transforms the folder using the specified source path.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="sourcePath">
        /// <para>The source path.</para>
        /// <para></para>
        /// </param>
        /// <param name="targetPath">
        /// <para>The target path.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual void TransformFolder(string sourcePath, string targetPath)
        {
            if (CountFilesRecursively(sourcePath, SourceFileExtension) == 0)
            {
                return;
            }
            EnsureTargetDirectoryExists(targetPath);
            var directories = Directory.GetDirectories(sourcePath);
            for (var i = 0; i < directories.Length; i++)
            {
#if NETSTANDARD2_1
                var relativePath = Path.GetRelativePath(sourcePath, directories[i]);
#else
                var relativePath =
                ↪  directories[i].Replace(sourcePath.TrimEnd(Path.DirectorySeparatorChar) +
                ↪  Path.DirectorySeparatorChar, "");
#endif
                var newTargetPath = Path.Combine(targetPath, relativePath);
                TransformFolder(directories[i], newTargetPath);
            }
            var files = Directory.GetFiles(sourcePath);
            Parallel.For(0, files.Length, i =>
            {
                var file = files[i];
                if (FileExtensionMatches(file, SourceFileExtension))
                {
                    TransformFile(file, GetTargetFileName(file, targetPath));
                }
            });
        }

        /// <summary>
        /// <para>
        /// Transforms the file using the specified source path.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="sourcePath">
        /// <para>The source path.</para>
        /// <para></para>
        /// </param>
        /// <param name="targetPath">
        /// <para>The target path.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual void TransformFile(string sourcePath, string targetPath)
        {
            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)
                {
```

```csharp
229                    return;
230                }
231            }
232            var sourceText = File.ReadAllText(sourcePath, Encoding.UTF8);
233            var targetText = _textTransformer.Transform(sourceText);
234            File.WriteAllText(targetPath, targetText, Encoding.UTF8);
235        }

236
237        /// <summary>
238        /// <para>
239        /// Gets the target file name using the specified source path.
240        /// </para>
241        /// <para></para>
242        /// </summary>
243        /// <param name="sourcePath">
244        /// <para>The source path.</para>
245        /// <para></para>
246        /// </param>
247        /// <param name="targetDirectory">
248        /// <para>The target directory.</para>
249        /// <para></para>
250        /// </param>
251        /// <returns>
252        /// <para>The string</para>
253        /// <para></para>
254        /// </returns>
255        [MethodImpl(MethodImplOptions.AggressiveInlining)]
256        protected string GetTargetFileName(string sourcePath, string targetDirectory) =>
           ↪  Path.ChangeExtension(Path.Combine(targetDirectory, Path.GetFileName(sourcePath)),
           ↪  TargetFileExtension);

257
258        /// <summary>
259        /// <para>
260        /// Counts the files recursively using the specified path.
261        /// </para>
262        /// <para></para>
263        /// </summary>
264        /// <param name="path">
265        /// <para>The path.</para>
266        /// <para></para>
267        /// </param>
268        /// <param name="extension">
269        /// <para>The extension.</para>
270        /// <para></para>
271        /// </param>
272        /// <returns>
273        /// <para>The result.</para>
274        /// <para></para>
275        /// </returns>
276        [MethodImpl(MethodImplOptions.AggressiveInlining)]
277        private static long CountFilesRecursively(string path, string extension)
278        {
279            var files = Directory.GetFiles(path);
280            var directories = Directory.GetDirectories(path);
281            var result = 0L;
282            for (var i = 0; i < directories.Length; i++)
283            {
284                result += CountFilesRecursively(directories[i], extension);
285            }
286            for (var i = 0; i < files.Length; i++)
287            {
288                if (FileExtensionMatches(files[i], extension))
289                {
290                    result++;
291                }
292            }
293            return result;
294        }

295
296        /// <summary>
297        /// <para>
298        /// Determines whether file extension matches.
299        /// </para>
300        /// <para></para>
301        /// </summary>
302        /// <param name="file">
303        /// <para>The file.</para>
304        /// <para></para>
```

```csharp
        /// </param>
        /// <param name="extension">
        /// <para>The extension.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static bool FileExtensionMatches(string file, string extension) =>
            file.EndsWith(extension, StringComparison.OrdinalIgnoreCase);

        /// <summary>
        /// <para>
        /// Ensures the target file directory exists using the specified target path.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="targetPath">
        /// <para>The target path.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static void EnsureTargetFileDirectoryExists(string targetPath)
        {
            if (!File.Exists(targetPath))
            {
                EnsureDirectoryIsCreated(targetPath);
            }
        }

        /// <summary>
        /// <para>
        /// Ensures the target directory exists using the specified target path.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="targetPath">
        /// <para>The target path.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static void EnsureTargetDirectoryExists(string targetPath) =>
            EnsureTargetDirectoryExists(targetPath, DirectoryExists(targetPath));

        /// <summary>
        /// <para>
        /// Ensures the target directory exists using the specified target path.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="targetPath">
        /// <para>The target path.</para>
        /// <para></para>
        /// </param>
        /// <param name="targetDirectoryExists">
        /// <para>The target directory exists.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static void EnsureTargetDirectoryExists(string targetPath, bool
            targetDirectoryExists)
        {
            if (!targetDirectoryExists)
            {
                Directory.CreateDirectory(targetPath);
            }
        }

        /// <summary>
        /// <para>
        /// Ensures the source file exists using the specified source path.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="sourcePath">
        /// <para>The source path.</para>
```

```
380     /// <para></para>
381     /// </param>
382     /// <exception cref="FileNotFoundException">
383     /// <para>Source file does not exists. </para>
384     /// <para></para>
385     /// </exception>
386     [MethodImpl(MethodImplOptions.AggressiveInlining)]
387     private static void EnsureSourceFileExists(string sourcePath)
388     {
389         if (!File.Exists(sourcePath))
390         {
391             throw new FileNotFoundException("Source file does not exists.", sourcePath);
392         }
393     }
394
395     /// <summary>
396     /// <para>
397     /// Ensures the directory is created using the specified target path.
398     /// </para>
399     /// <para></para>
400     /// </summary>
401     /// <param name="targetPath">
402     /// <para>The target path.</para>
403     /// <para></para>
404     /// </param>
405     [MethodImpl(MethodImplOptions.AggressiveInlining)]
406     private static void EnsureDirectoryIsCreated(string targetPath) =>
        ↪   Directory.CreateDirectory(Path.GetDirectoryName(targetPath));
407
408     /// <summary>
409     /// <para>
410     /// Determines whether directory exists.
411     /// </para>
412     /// <para></para>
413     /// </summary>
414     /// <param name="path">
415     /// <para>The path.</para>
416     /// <para></para>
417     /// </param>
418     /// <returns>
419     /// <para>The bool</para>
420     /// <para></para>
421     /// </returns>
422     [MethodImpl(MethodImplOptions.AggressiveInlining)]
423     private static bool DirectoryExists(string path) => Directory.Exists(path) &&
        ↪   File.GetAttributes(path).HasFlag(FileAttributes.Directory);
424
425     /// <summary>
426     /// <para>
427     /// Determines whether looks like directory path.
428     /// </para>
429     /// <para></para>
430     /// </summary>
431     /// <param name="path">
432     /// <para>The path.</para>
433     /// <para></para>
434     /// </param>
435     /// <returns>
436     /// <para>The bool</para>
437     /// <para></para>
438     /// </returns>
439     [MethodImpl(MethodImplOptions.AggressiveInlining)]
440     private static bool LooksLikeDirectoryPath(string path) =>
        ↪   path.EndsWith(Path.DirectorySeparatorChar.ToString()) ||
        ↪   path.EndsWith(Path.AltDirectorySeparatorChar.ToString());
441     }
442 }
```

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       /// <summary>
8       /// <para>
9       /// Defines the file transformer.
```

```csharp
        /// </para>
        /// <para></para>
        /// </summary>
        /// <seealso cref="ITransformer"/>
        public interface IFileTransformer : ITransformer
        {
            /// <summary>
            /// <para>
            /// Gets the source file extension value.
            /// </para>
            /// <para></para>
            /// </summary>
            string SourceFileExtension
            {
                [MethodImpl(MethodImplOptions.AggressiveInlining)]
                get;
            }

            /// <summary>
            /// <para>
            /// Gets the target file extension value.
            /// </para>
            /// <para></para>
            /// </summary>
            string TargetFileExtension
            {
                [MethodImpl(MethodImplOptions.AggressiveInlining)]
                get;
            }

            /// <summary>
            /// <para>
            /// Transforms the source path.
            /// </para>
            /// <para></para>
            /// </summary>
            /// <param name="sourcePath">
            /// <para>The source path.</para>
            /// <para></para>
            /// </param>
            /// <param name="targetPath">
            /// <para>The target path.</para>
            /// <para></para>
            /// </param>
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            void Transform(string sourcePath, string targetPath);
        }
}
```

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

```csharp
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
{
    /// <summary>
    /// <para>
    /// Defines the substitution rule.
    /// </para>
    /// <para></para>
    /// </summary>
    public interface ISubstitutionRule
    {
        /// <summary>
        /// <para>
        /// Gets the match pattern value.
        /// </para>
        /// <para></para>
        /// </summary>
        Regex MatchPattern
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
        }

        /// <summary>
        /// <para>
```

```
30        /// Gets the substitution pattern value.
31        /// </para>
32        /// <para></para>
33        /// </summary>
34        string SubstitutionPattern
35        {
36            [MethodImpl(MethodImplOptions.AggressiveInlining)]
37            get;
38        }
39
40        /// <summary>
41        /// <para>
42        /// Gets the maximum repeat count value.
43        /// </para>
44        /// <para></para>
45        /// </summary>
46        int MaximumRepeatCount
47        {
48            [MethodImpl(MethodImplOptions.AggressiveInlining)]
49            get;
50        }
51    }
52 }
```

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      /// <summary>
8      /// <para>
9      /// Defines the text transformer.
10     /// </para>
11     /// <para></para>
12     /// </summary>
13     /// <seealso cref="ITransformer"/>
14     public interface ITextTransformer : ITransformer
15     {
16         /// <summary>
17         /// <para>
18         /// Transforms the source text.
19         /// </para>
20         /// <para></para>
21         /// </summary>
22         /// <param name="sourceText">
23         /// <para>The source text.</para>
24         /// <para></para>
25         /// </param>
26         /// <returns>
27         /// <para>The string</para>
28         /// <para></para>
29         /// </returns>
30         [MethodImpl(MethodImplOptions.AggressiveInlining)]
31         string Transform(string sourceText);
32     }
33 }
```

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     /// <summary>
12     /// <para>
13     /// Represents the text transformer extensions.
14     /// </para>
15     /// <para></para>
16     /// </summary>
17     public static class ITextTransformerExtensions
18     {
19         /// <summary>
```

```csharp
/// <para>
/// Generates the transformers for each rule using the specified transformer.
/// </para>
/// <para></para>
/// </summary>
/// <param name="transformer">
/// <para>The transformer.</para>
/// <para></para>
/// </param>
/// <returns>
/// <para>The transformers.</para>
/// <para></para>
/// </returns>
[MethodImpl(MethodImplOptions.AggressiveInlining)]
public static IList<ITextTransformer> GenerateTransformersForEachRule(this
    ITextTransformer transformer)
{
    var transformers = new List<ITextTransformer>();
    for (int i = 1; i <= transformer.Rules.Count; i++)
    {
        transformers.Add(new TextTransformer(transformer.Rules.Take(i).ToList()));
    }
    return transformers;
}

/// <summary>
/// <para>
/// Gets the steps using the specified transformer.
/// </para>
/// <para></para>
/// </summary>
/// <param name="transformer">
/// <para>The transformer.</para>
/// <para></para>
/// </param>
/// <param name="sourceText">
/// <para>The source text.</para>
/// <para></para>
/// </param>
/// <returns>
/// <para>A list of string</para>
/// <para></para>
/// </returns>
[MethodImpl(MethodImplOptions.AggressiveInlining)]
public static IList<string> GetSteps(this ITextTransformer transformer, string
    sourceText)
{
    if (transformer != null && !transformer.Rules.IsNullOrEmpty())
    {
        var steps = new List<string>();
        var steppedTransformer = new TextSteppedTransformer(transformer.Rules,
            sourceText);
        while (steppedTransformer.Next())
        {
            steps.Add(steppedTransformer.Text);
        }
        return steps;
    }
    else
    {
        return Array.Empty<string>();
    }
}

/// <summary>
/// <para>
/// Writes the steps to files using the specified transformer.
/// </para>
/// <para></para>
/// </summary>
/// <param name="transformer">
/// <para>The transformer.</para>
/// <para></para>
/// </param>
/// <param name="sourceText">
/// <para>The source text.</para>
/// <para></para>
/// </param>
```

```
 95          /// <param name="targetPath">
 96          /// <para>The target path.</para>
 97          /// <para></para>
 98          /// </param>
 99          /// <param name="skipFilesWithNoChanges">
100          /// <para>The skip files with no changes.</para>
101          /// <para></para>
102          /// </param>
103          [MethodImpl(MethodImplOptions.AggressiveInlining)]
104          public static void WriteStepsToFiles(this ITextTransformer transformer, string
       ↪   sourceText, string targetPath, bool skipFilesWithNoChanges)
105          {
106              if (transformer != null && !transformer.Rules.IsNullOrEmpty())
107              {
108                  targetPath.GetPathParts(out var directoryName, out var targetFilename, out var
           ↪   targetExtension);
109                  Steps.DeleteAllSteps(directoryName, targetFilename, targetExtension);
110                  var lastText = "";
111                  var steppedTransformer = new TextSteppedTransformer(transformer.Rules,
           ↪   sourceText);
112                  while (steppedTransformer.Next())
113                  {
114                      var newText = steppedTransformer.Text;
115                      Steps.WriteStep(transformer, directoryName, targetFilename, targetExtension,
               ↪   steppedTransformer.Current, ref lastText, newText,
               ↪   skipFilesWithNoChanges);
116                  }
117              }
118          }
119      }
120  }
```

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      /// <summary>
11      /// <para>
12      /// Represents the text transformers list extensions.
13      /// </para>
14      /// <para></para>
15      /// </summary>
16      public static class ITextTransformersListExtensions
17      {
18          /// <summary>
19          /// <para>
20          /// Transforms the with all using the specified transformers.
21          /// </para>
22          /// <para></para>
23          /// </summary>
24          /// <param name="transformers">
25          /// <para>The transformers.</para>
26          /// <para></para>
27          /// </param>
28          /// <param name="source">
29          /// <para>The source.</para>
30          /// <para></para>
31          /// </param>
32          /// <returns>
33          /// <para>A list of string</para>
34          /// <para></para>
35          /// </returns>
36          [MethodImpl(MethodImplOptions.AggressiveInlining)]
37          public static IList<string> TransformWithAll(this IList<ITextTransformer> transformers,
           ↪   string source)
38          {
39              if (!transformers.IsNullOrEmpty())
40              {
41                  var steps = new List<string>();
42                  for (int i = 0; i < transformers.Count; i++)
43                  {
44                      steps.Add(transformers[i].Transform(source));
```

```
45                  }
46                  return steps;
47              }
48              else
49              {
50                  return Array.Empty<string>();
51              }
52          }
53
54          /// <summary>
55          /// <para>
56          /// Transforms the with all to files using the specified transformers.
57          /// </para>
58          /// <para></para>
59          /// </summary>
60          /// <param name="transformers">
61          /// <para>The transformers.</para>
62          /// <para></para>
63          /// </param>
64          /// <param name="sourceText">
65          /// <para>The source text.</para>
66          /// <para></para>
67          /// </param>
68          /// <param name="targetPath">
69          /// <para>The target path.</para>
70          /// <para></para>
71          /// </param>
72          /// <param name="skipFilesWithNoChanges">
73          /// <para>The skip files with no changes.</para>
74          /// <para></para>
75          /// </param>
76          [MethodImpl(MethodImplOptions.AggressiveInlining)]
77          public static void TransformWithAllToFiles(this IList<ITextTransformer> transformers,
     ↪   string sourceText, string targetPath, bool skipFilesWithNoChanges)
78          {
79              if (!transformers.IsNullOrEmpty())
80              {
81                  targetPath.GetPathParts(out var directoryName, out var targetFilename, out var
                     ↪   targetExtension);
82                  Steps.DeleteAllSteps(directoryName, targetFilename, targetExtension);
83                  var lastText = "";
84                  for (int i = 0; i < transformers.Count; i++)
85                  {
86                      var transformer = transformers[i];
87                      var newText = transformer.Transform(sourceText);
88                      Steps.WriteStep(transformer, directoryName, targetFilename, targetExtension,
                         ↪   i, ref lastText, newText, skipFilesWithNoChanges);
89                  }
90              }
91          }
92      }
93  }
```

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       /// <summary>
9       /// <para>
10      /// Defines the transformer.
11      /// </para>
12      /// <para></para>
13      /// </summary>
14      public interface ITransformer
15      {
16          /// <summary>
17          /// <para>
18          /// Gets the rules value.
19          /// </para>
20          /// <para></para>
21          /// </summary>
22          IList<ISubstitutionRule> Rules
23          {
24              [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
25              get;
26           }
27        }
28    }
```

1.8 ./csharp/Platform.RegularExpressions.Transformer/LoggingFileTransformer.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       /// <summary>
10      /// <para>
11      /// Represents the logging file transformer.
12      /// </para>
13      /// <para></para>
14      /// </summary>
15      /// <seealso cref="FileTransformer"/>
16      public class LoggingFileTransformer : FileTransformer
17      {
18          /// <summary>
19          /// <para>
20          /// Initializes a new <see cref="LoggingFileTransformer"/> instance.
21          /// </para>
22          /// <para></para>
23          /// </summary>
24          /// <param name="textTransformer">
25          /// <para>A text transformer.</para>
26          /// <para></para>
27          /// </param>
28          /// <param name="sourceFileExtension">
29          /// <para>A source file extension.</para>
30          /// <para></para>
31          /// </param>
32          /// <param name="targetFileExtension">
33          /// <para>A target file extension.</para>
34          /// <para></para>
35          /// </param>
36          [MethodImpl(MethodImplOptions.AggressiveInlining)]
37          public LoggingFileTransformer(ITextTransformer textTransformer, string
              ↪  sourceFileExtension, string targetFileExtension) : base(textTransformer,
              ↪  sourceFileExtension, targetFileExtension) { }
38
39          /// <summary>
40          /// <para>
41          /// Transforms the file using the specified source path.
42          /// </para>
43          /// <para></para>
44          /// </summary>
45          /// <param name="sourcePath">
46          /// <para>The source path.</para>
47          /// <para></para>
48          /// </param>
49          /// <param name="targetPath">
50          /// <para>The target path.</para>
51          /// <para></para>
52          /// </param>
53          [MethodImpl(MethodImplOptions.AggressiveInlining)]
54          protected override void TransformFile(string sourcePath, string targetPath)
55          {
56              base.TransformFile(sourcePath, targetPath);
57              // Logging
58              var sourceText = File.ReadAllText(sourcePath, Encoding.UTF8);
59              _textTransformer.WriteStepsToFiles(sourceText, targetPath, skipFilesWithNoChanges:
                  ↪  true);
60          }
61      }
62  }
```

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
```

```csharp
namespace Platform.RegularExpressions.Transformer
{
    /// <summary>
    /// <para>
    /// Represents the regex extensions.
    /// </para>
    /// <para></para>
    /// </summary>
    public static class RegexExtensions
    {
        /// <summary>
        /// <para>
        /// Overrides the options using the specified regex.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="regex">
        /// <para>The regex.</para>
        /// <para></para>
        /// </param>
        /// <param name="options">
        /// <para>The options.</para>
        /// <para></para>
        /// </param>
        /// <param name="matchTimeout">
        /// <para>The match timeout.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The regex</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static Regex OverrideOptions(this Regex regex, RegexOptions options, TimeSpan
        ↪ matchTimeout)
        {
            if (regex == null)
            {
                return null;
            }
            return new Regex(regex.ToString(), options, matchTimeout);
        }
    }
}
```

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

```csharp
using Platform.IO;
using System.Runtime.CompilerServices;

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

namespace Platform.RegularExpressions.Transformer
{
    /// <summary>
    /// <para>
    /// Represents the steps.
    /// </para>
    /// <para></para>
    /// </summary>
    public static class Steps
    {
        /// <summary>
        /// <para>
        /// Deletes the all steps using the specified directory name.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="directoryName">
        /// <para>The directory name.</para>
        /// <para></para>
        /// </param>
        /// <param name="targetFilename">
        /// <para>The target filename.</para>
        /// <para></para>
        /// </param>
        /// <param name="targetExtension">
        /// <para>The target extension.</para>
```

```csharp
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void DeleteAllSteps(string directoryName, string targetFilename, string
        ↪  targetExtension)
        {
            FileHelpers.DeleteAll(directoryName, $"{targetFilename}.*.rule.txt");
            FileHelpers.DeleteAll(directoryName, $"{targetFilename}.*{targetExtension}");
        }

        /// <summary>
        /// <para>
        /// Writes the step using the specified transformer.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="transformer">
        /// <para>The transformer.</para>
        /// <para></para>
        /// </param>
        /// <param name="directoryName">
        /// <para>The directory name.</para>
        /// <para></para>
        /// </param>
        /// <param name="targetFilename">
        /// <para>The target filename.</para>
        /// <para></para>
        /// </param>
        /// <param name="targetExtension">
        /// <para>The target extension.</para>
        /// <para></para>
        /// </param>
        /// <param name="currentStep">
        /// <para>The current step.</para>
        /// <para></para>
        /// </param>
        /// <param name="lastText">
        /// <para>The last text.</para>
        /// <para></para>
        /// </param>
        /// <param name="newText">
        /// <para>The new text.</para>
        /// <para></para>
        /// </param>
        /// <param name="skipFilesWithNoChanges">
        /// <para>The skip files with no changes.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void WriteStep(ITransformer transformer, string directoryName, string
        ↪  targetFilename, string targetExtension, int currentStep, ref string lastText, string
        ↪  newText, bool skipFilesWithNoChanges)
        {
            if (!(skipFilesWithNoChanges && string.Equals(lastText, newText)))
            {
                lastText = newText;
                newText.WriteToFile(directoryName,
                ↪  $"{targetFilename}.{currentStep}{targetExtension}");
                var ruleString = transformer.Rules[currentStep].ToString();
                ruleString.WriteToFile(directoryName,
                ↪  $"{targetFilename}.{currentStep}.rule.txt");
            }
        }
    }
}
```

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

```csharp
using System.IO;
using System.Runtime.CompilerServices;
using System.Text;

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

namespace Platform.RegularExpressions.Transformer
{
    /// <summary>
    /// <para>
    /// Represents the string extensions.
```

```
12    /// </para>
13    /// <para></para>
14    /// </summary>
15    internal static class StringExtensions
16    {
17        /// <summary>
18        /// <para>
19        /// Gets the path parts using the specified path.
20        /// </para>
21        /// <para></para>
22        /// </summary>
23        /// <param name="path">
24        /// <para>The path.</para>
25        /// <para></para>
26        /// </param>
27        /// <param name="directoryName">
28        /// <para>The directory name.</para>
29        /// <para></para>
30        /// </param>
31        /// <param name="targetFilename">
32        /// <para>The target filename.</para>
33        /// <para></para>
34        /// </param>
35        /// <param name="targetExtension">
36        /// <para>The target extension.</para>
37        /// <para></para>
38        /// </param>
39        [MethodImpl(MethodImplOptions.AggressiveInlining)]
40        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));
41
42        /// <summary>
43        /// <para>
44        /// Writes the to file using the specified text.
45        /// </para>
46        /// <para></para>
47        /// </summary>
48        /// <param name="text">
49        /// <para>The text.</para>
50        /// <para></para>
51        /// </param>
52        /// <param name="directoryName">
53        /// <para>The directory name.</para>
54        /// <para></para>
55        /// </param>
56        /// <param name="targetFilename">
57        /// <para>The target filename.</para>
58        /// <para></para>
59        /// </param>
60        [MethodImpl(MethodImplOptions.AggressiveInlining)]
61        public static void WriteToFile(this string text, string directoryName, string
           ↪   targetFilename) => File.WriteAllText(Path.Combine(directoryName, targetFilename),
           ↪   text, Encoding.UTF8);
62    }
63 }
```

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     /// <summary>
11     /// <para>
12     /// Represents the substitution rule.
13     /// </para>
14     /// <para></para>
15     /// </summary>
16     /// <seealso cref="ISubstitutionRule"/>
17     public class SubstitutionRule : ISubstitutionRule
18     {
19         /// <summary>
```

```csharp
        /// <para>
        /// The from minutes.
        /// </para>
        /// <para></para>
        /// </summary>
        public static readonly TimeSpan DefaultMatchTimeout = TimeSpan.FromMinutes(5);
        /// <summary>
        /// <para>
        /// The multiline.
        /// </para>
        /// <para></para>
        /// </summary>
        public static readonly RegexOptions DefaultMatchPatternRegexOptions =
        ↪   RegexOptions.Compiled | RegexOptions.Multiline;

        /// <summary>
        /// <para>
        /// Gets or sets the match pattern value.
        /// </para>
        /// <para></para>
        /// </summary>
        public Regex MatchPattern
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            set;
        }

        /// <summary>
        /// <para>
        /// Gets or sets the substitution pattern value.
        /// </para>
        /// <para></para>
        /// </summary>
        public string SubstitutionPattern
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            set;
        }

        /// <summary>
        /// <para>
        /// Gets or sets the path pattern value.
        /// </para>
        /// <para></para>
        /// </summary>
        public Regex PathPattern
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            set;
        }

        /// <summary>
        /// <para>
        /// Gets or sets the maximum repeat count value.
        /// </para>
        /// <para></para>
        /// </summary>
        public int MaximumRepeatCount
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            set;
        }

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="SubstitutionRule"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="matchPattern">
        /// <para>A match pattern.</para>
```

```csharp
 98         /// <para></para>
 99         /// </param>
100         /// <param name="substitutionPattern">
101         /// <para>A substitution pattern.</para>
102         /// <para></para>
103         /// </param>
104         /// <param name="maximumRepeatCount">
105         /// <para>A maximum repeat count.</para>
106         /// <para></para>
107         /// </param>
108         /// <param name="matchPatternOptions">
109         /// <para>A match pattern options.</para>
110         /// <para></para>
111         /// </param>
112         /// <param name="matchTimeout">
113         /// <para>A match timeout.</para>
114         /// <para></para>
115         /// </param>
116         [MethodImpl(MethodImplOptions.AggressiveInlining)]
117         public SubstitutionRule(Regex matchPattern, string substitutionPattern, int
             ↪  maximumRepeatCount, RegexOptions? matchPatternOptions, TimeSpan? matchTimeout)
118         {
119             MatchPattern = matchPattern;
120             SubstitutionPattern = substitutionPattern;
121             MaximumRepeatCount = maximumRepeatCount;
122             OverrideMatchPatternOptions(matchPatternOptions ?? matchPattern.Options,
                 ↪  matchTimeout ?? matchPattern.MatchTimeout);
123         }
124
125         /// <summary>
126         /// <para>
127         /// Initializes a new <see cref="SubstitutionRule"/> instance.
128         /// </para>
129         /// <para></para>
130         /// </summary>
131         /// <param name="matchPattern">
132         /// <para>A match pattern.</para>
133         /// <para></para>
134         /// </param>
135         /// <param name="substitutionPattern">
136         /// <para>A substitution pattern.</para>
137         /// <para></para>
138         /// </param>
139         /// <param name="maximumRepeatCount">
140         /// <para>A maximum repeat count.</para>
141         /// <para></para>
142         /// </param>
143         /// <param name="useDefaultOptions">
144         /// <para>A use default options.</para>
145         /// <para></para>
146         /// </param>
147         [MethodImpl(MethodImplOptions.AggressiveInlining)]
148         public SubstitutionRule(Regex matchPattern, string substitutionPattern, int
             ↪  maximumRepeatCount, bool useDefaultOptions) : this(matchPattern,
             ↪  substitutionPattern, maximumRepeatCount, useDefaultOptions ?
             ↪  DefaultMatchPatternRegexOptions : (RegexOptions?)null, useDefaultOptions ?
             ↪  DefaultMatchTimeout : (TimeSpan?)null) { }
149
150         /// <summary>
151         /// <para>
152         /// Initializes a new <see cref="SubstitutionRule"/> instance.
153         /// </para>
154         /// <para></para>
155         /// </summary>
156         /// <param name="matchPattern">
157         /// <para>A match pattern.</para>
158         /// <para></para>
159         /// </param>
160         /// <param name="substitutionPattern">
161         /// <para>A substitution pattern.</para>
162         /// <para></para>
163         /// </param>
164         /// <param name="maximumRepeatCount">
165         /// <para>A maximum repeat count.</para>
166         /// <para></para>
167         /// </param>
168         [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```csharp
169        public SubstitutionRule(Regex matchPattern, string substitutionPattern, int
     ↪   maximumRepeatCount) : this(matchPattern, substitutionPattern, maximumRepeatCount,
     ↪   true) { }
170
171        /// <summary>
172        /// <para>
173        /// Initializes a new <see cref="SubstitutionRule"/> instance.
174        /// </para>
175        /// <para></para>
176        /// </summary>
177        /// <param name="matchPattern">
178        /// <para>A match pattern.</para>
179        /// <para></para>
180        /// </param>
181        /// <param name="substitutionPattern">
182        /// <para>A substitution pattern.</para>
183        /// <para></para>
184        /// </param>
185        [MethodImpl(MethodImplOptions.AggressiveInlining)]
186        public SubstitutionRule(Regex matchPattern, string substitutionPattern) :
     ↪   this(matchPattern, substitutionPattern, 0) { }
187
188        [MethodImpl(MethodImplOptions.AggressiveInlining)]
189        public static implicit operator SubstitutionRule(ValueTuple<string, string> tuple) =>
     ↪   new SubstitutionRule(new Regex(tuple.Item1), tuple.Item2);
190
191        [MethodImpl(MethodImplOptions.AggressiveInlining)]
192        public static implicit operator SubstitutionRule(ValueTuple<Regex, string> tuple) => new
     ↪   SubstitutionRule(tuple.Item1, tuple.Item2);
193
194        [MethodImpl(MethodImplOptions.AggressiveInlining)]
195        public static implicit operator SubstitutionRule(ValueTuple<string, string, int> tuple)
     ↪   => new SubstitutionRule(new Regex(tuple.Item1), tuple.Item2, tuple.Item3);
196
197        [MethodImpl(MethodImplOptions.AggressiveInlining)]
198        public static implicit operator SubstitutionRule(ValueTuple<Regex, string, int> tuple)
     ↪   => new SubstitutionRule(tuple.Item1, tuple.Item2, tuple.Item3);
199
200        /// <summary>
201        /// <para>
202        /// Overrides the match pattern options using the specified options.
203        /// </para>
204        /// <para></para>
205        /// </summary>
206        /// <param name="options">
207        /// <para>The options.</para>
208        /// <para></para>
209        /// </param>
210        /// <param name="matchTimeout">
211        /// <para>The match timeout.</para>
212        /// <para></para>
213        /// </param>
214        [MethodImpl(MethodImplOptions.AggressiveInlining)]
215        public void OverrideMatchPatternOptions(RegexOptions options, TimeSpan matchTimeout) =>
     ↪   MatchPattern = MatchPattern.OverrideOptions(options, matchTimeout);
216
217        /// <summary>
218        /// <para>
219        /// Overrides the path pattern options using the specified options.
220        /// </para>
221        /// <para></para>
222        /// </summary>
223        /// <param name="options">
224        /// <para>The options.</para>
225        /// <para></para>
226        /// </param>
227        /// <param name="matchTimeout">
228        /// <para>The match timeout.</para>
229        /// <para></para>
230        /// </param>
231        [MethodImpl(MethodImplOptions.AggressiveInlining)]
232        public void OverridePathPatternOptions(RegexOptions options, TimeSpan matchTimeout) =>
     ↪   PathPattern = PathPattern.OverrideOptions(options, matchTimeout);
233
234        /// <summary>
235        /// <para>
236        /// Returns the string.
```

```csharp
237         /// </para>
238         /// <para></para>
239         /// </summary>
240         /// <returns>
241         /// <para>The string</para>
242         /// <para></para>
243         /// </returns>
244         [MethodImpl(MethodImplOptions.AggressiveInlining)]
245         public override string ToString()
246         {
247             var sb = new StringBuilder();
248             sb.Append('"');
249             sb.Append(MatchPattern.ToString());
250             sb.Append('"');
251             sb.Append(" -> ");
252             sb.Append('"');
253             sb.Append(SubstitutionPattern);
254             sb.Append('"');
255             if (PathPattern != null)
256             {
257                 sb.Append(" on files ");
258                 sb.Append('"');
259                 sb.Append(PathPattern.ToString());
260                 sb.Append('"');
261             }
262             if (MaximumRepeatCount > 0)
263             {
264                 if (MaximumRepeatCount >= int.MaxValue)
265                 {
266                     sb.Append(" repeated forever");
267                 }
268                 else
269                 {
270                     sb.Append(" repeated up to ");
271                     sb.Append(MaximumRepeatCount);
272                     sb.Append(" times");
273                 }
274             }
275             return sb.ToString();
276         }
277     }
278 }
```

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

```csharp
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      /// <summary>
10     /// <para>
11     /// Represents the text stepped transformer.
12     /// </para>
13     /// <para></para>
14     /// </summary>
15     /// <seealso cref="ITransformer"/>
16     public class TextSteppedTransformer : ITransformer
17     {
18         /// <summary>
19         /// <para>
20         /// Gets or sets the rules value.
21         /// </para>
22         /// <para></para>
23         /// </summary>
24         public IList<ISubstitutionRule> Rules
25         {
26             [MethodImpl(MethodImplOptions.AggressiveInlining)]
27             get;
28             [MethodImpl(MethodImplOptions.AggressiveInlining)]
29             set;
30         }
31
32         /// <summary>
33         /// <para>
34         /// Gets or sets the text value.
35         /// </para>
```

```csharp
36          /// <para></para>
37          /// </summary>
38          public string Text
39          {
40              [MethodImpl(MethodImplOptions.AggressiveInlining)]
41              get;
42              [MethodImpl(MethodImplOptions.AggressiveInlining)]
43              set;
44          }
45
46          /// <summary>
47          /// <para>
48          /// Gets or sets the current value.
49          /// </para>
50          /// <para></para>
51          /// </summary>
52          public int Current
53          {
54              [MethodImpl(MethodImplOptions.AggressiveInlining)]
55              get;
56              [MethodImpl(MethodImplOptions.AggressiveInlining)]
57              set;
58          }
59
60          /// <summary>
61          /// <para>
62          /// Initializes a new <see cref="TextSteppedTransformer"/> instance.
63          /// </para>
64          /// <para></para>
65          /// </summary>
66          /// <param name="rules">
67          /// <para>A rules.</para>
68          /// <para></para>
69          /// </param>
70          /// <param name="text">
71          /// <para>A text.</para>
72          /// <para></para>
73          /// </param>
74          /// <param name="current">
75          /// <para>A current.</para>
76          /// <para></para>
77          /// </param>
78          [MethodImpl(MethodImplOptions.AggressiveInlining)]
79          public TextSteppedTransformer(IList<ISubstitutionRule> rules, string text, int current)
                => Reset(rules, text, current);
80
81          /// <summary>
82          /// <para>
83          /// Initializes a new <see cref="TextSteppedTransformer"/> instance.
84          /// </para>
85          /// <para></para>
86          /// </summary>
87          /// <param name="rules">
88          /// <para>A rules.</para>
89          /// <para></para>
90          /// </param>
91          /// <param name="text">
92          /// <para>A text.</para>
93          /// <para></para>
94          /// </param>
95          [MethodImpl(MethodImplOptions.AggressiveInlining)]
96          public TextSteppedTransformer(IList<ISubstitutionRule> rules, string text) =>
                Reset(rules, text);
97
98          /// <summary>
99          /// <para>
100         /// Initializes a new <see cref="TextSteppedTransformer"/> instance.
101         /// </para>
102         /// <para></para>
103         /// </summary>
104         /// <param name="rules">
105         /// <para>A rules.</para>
106         /// <para></para>
107         /// </param>
108         [MethodImpl(MethodImplOptions.AggressiveInlining)]
109         public TextSteppedTransformer(IList<ISubstitutionRule> rules) => Reset(rules);
110
111         /// <summary>
```

```csharp
        /// <para>
        /// Initializes a new <see cref="TextSteppedTransformer"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TextSteppedTransformer() => Reset();

        /// <summary>
        /// <para>
        /// Resets the rules.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="rules">
        /// <para>The rules.</para>
        /// <para></para>
        /// </param>
        /// <param name="text">
        /// <para>The text.</para>
        /// <para></para>
        /// </param>
        /// <param name="current">
        /// <para>The current.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void Reset(IList<ISubstitutionRule> rules, string text, int current)
        {
            Rules = rules;
            Text = text;
            Current = current;
        }

        /// <summary>
        /// <para>
        /// Resets the rules.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="rules">
        /// <para>The rules.</para>
        /// <para></para>
        /// </param>
        /// <param name="text">
        /// <para>The text.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void Reset(IList<ISubstitutionRule> rules, string text) => Reset(rules, text, -1);

        /// <summary>
        /// <para>
        /// Resets the rules.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="rules">
        /// <para>The rules.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void Reset(IList<ISubstitutionRule> rules) => Reset(rules, "", -1);

        /// <summary>
        /// <para>
        /// Resets the text.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="text">
        /// <para>The text.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void Reset(string text) => Reset(Rules, text, -1);

        /// <summary>
```

```csharp
        /// <para>
        /// Resets this instance.
        /// </para>
        /// <para></para>
        /// </summary>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void Reset() => Reset(Array.Empty<ISubstitutionRule>(), "", -1);

        /// <summary>
        /// <para>
        /// Determines whether this instance next.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public bool Next()
        {
            var current = Current + 1;
            if (current >= Rules.Count)
            {
                return false;
            }
            var rule = Rules[current];
            var matchPattern = rule.MatchPattern;
            var substitutionPattern = rule.SubstitutionPattern;
            var maximumRepeatCount = rule.MaximumRepeatCount;
            var replaceCount = 0;
            var text = Text;
            do
            {
                text = matchPattern.Replace(text, substitutionPattern);
                replaceCount++;
            }
            while ((maximumRepeatCount == int.MaxValue || replaceCount <= maximumRepeatCount) &&
            ↪  matchPattern.IsMatch(text));
            Text = text;
            Current = current;
            return true;
        }
    }
}
```

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

```csharp
using System.Collections.Generic;
using System.Runtime.CompilerServices;

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

namespace Platform.RegularExpressions.Transformer
{
    /// <summary>
    /// <para>
    /// Represents the text transformer.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="ITextTransformer"/>
    public class TextTransformer : ITextTransformer
    {
        /// <summary>
        /// <para>
        /// Gets or sets the rules value.
        /// </para>
        /// <para></para>
        /// </summary>
        public IList<ISubstitutionRule> Rules
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            private set;
        }

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="TextTransformer"/> instance.
```

```
34      /// </para>
35      /// <para></para>
36      /// </summary>
37      /// <param name="substitutionRules">
38      /// <para>A substitution rules.</para>
39      /// <para></para>
40      /// </param>
41      [MethodImpl(MethodImplOptions.AggressiveInlining)]
42      public TextTransformer(IList<ISubstitutionRule> substitutionRules)
43      {
44          Rules = substitutionRules;
45      }
46
47      /// <summary>
48      /// <para>
49      /// Transforms the source.
50      /// </para>
51      /// <para></para>
52      /// </summary>
53      /// <param name="source">
54      /// <para>The source.</para>
55      /// <para></para>
56      /// </param>
57      /// <returns>
58      /// <para>The string</para>
59      /// <para></para>
60      /// </returns>
61      [MethodImpl(MethodImplOptions.AggressiveInlining)]
62      public string Transform(string source)
63      {
64          var baseTrasformer = new TextSteppedTransformer(Rules);
65          baseTrasformer.Reset(source);
66          while (baseTrasformer.Next());
67          return baseTrasformer.Text;
68      }
69  }
70 }
```

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      /// <summary>
9      /// <para>
10     /// Represents the transformer cli.
11     /// </para>
12     /// <para></para>
13     /// </summary>
14     public class TransformerCLI
15     {
16         /// <summary>
17         /// <para>
18         /// The transformer.
19         /// </para>
20         /// <para></para>
21         /// </summary>
22         private readonly IFileTransformer _transformer;
23
24         /// <summary>
25         /// <para>
26         /// Initializes a new <see cref="TransformerCLI"/> instance.
27         /// </para>
28         /// <para></para>
29         /// </summary>
30         /// <param name="transformer">
31         /// <para>A transformer.</para>
32         /// <para></para>
33         /// </param>
34         [MethodImpl(MethodImplOptions.AggressiveInlining)]
35         public TransformerCLI(IFileTransformer transformer) => _transformer = transformer;
36
37         /// <summary>
38         /// <para>
39         /// Runs the args.
40         /// </para>
```

```
41        /// <para></para>
42        /// </summary>
43        /// <param name="args">
44        /// <para>The args.</para>
45        /// <para></para>
46        /// </param>
47        [MethodImpl(MethodImplOptions.AggressiveInlining)]
48        public void Run(string[] args)
49        {
50            var sourcePath = args.GetElementOrDefault(0);
51            var targetPath = args.GetElementOrDefault(1);
52            _transformer.Transform(sourcePath, targetPath);
53        }
54    }
55 }
```

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      /// <summary>
7      /// <para>
8      /// Represents the file transformer tests.
9      /// </para>
10     /// <para></para>
11     /// </summary>
12     public class FileTransformerTests
13     {
14         /// <summary>
15         /// <para>
16         /// Tests that folder to folder transfomation test.
17         /// </para>
18         /// <para></para>
19         /// </summary>
20         [Fact]
21         public void FolderToFolderTransfomationTest()
22         {
23             var tempPath = Path.GetTempPath();
24             var sourceFolderPath = Path.Combine(tempPath,
                 ↪  "FileTransformerTestsFolderToFolderTransfomationTestSourceFolder");
25             var targetFolderPath = Path.Combine(tempPath,
                 ↪  "FileTransformerTestsFolderToFolderTransfomationTestTargetFolder");
26
27             var baseTransformer = new TextTransformer(new SubstitutionRule[]
28             {
29                 ("a", "b"),
30                 ("b", "c")
31             });
32             var fileTransformer = new FileTransformer(baseTransformer, ".cs", ".cpp");
33
34             // Delete before creation (if previous test failed)
35             if (Directory.Exists(sourceFolderPath))
36             {
37                 Directory.Delete(sourceFolderPath, true);
38             }
39             if (Directory.Exists(targetFolderPath))
40             {
41                 Directory.Delete(targetFolderPath, true);
42             }
43
44             Directory.CreateDirectory(sourceFolderPath);
45             Directory.CreateDirectory(targetFolderPath);
46
47             File.WriteAllText(Path.Combine(sourceFolderPath, "a.cs"), "a a a");
48             var aFolderPath = Path.Combine(sourceFolderPath, "A");
49             Directory.CreateDirectory(aFolderPath);
50             Directory.CreateDirectory(Path.Combine(sourceFolderPath, "B"));
51             File.WriteAllText(Path.Combine(aFolderPath, "b.cs"), "b b b");
52             File.WriteAllText(Path.Combine(sourceFolderPath, "x.txt"), "should not be
                 ↪  translated");
53
54             fileTransformer.Transform(sourceFolderPath,
                 ↪  $"{targetFolderPath}{Path.DirectorySeparatorChar}");
55
56             var aCppFile = Path.Combine(targetFolderPath, "a.cpp");
57             Assert.True(File.Exists(aCppFile));
```

```
58          Assert.Equal("c c c", File.ReadAllText(aCppFile));
59          Assert.True(Directory.Exists(Path.Combine(targetFolderPath, "A")));
60          Assert.False(Directory.Exists(Path.Combine(targetFolderPath, "B")));
61          var bCppFile = Path.Combine(targetFolderPath, "A", "b.cpp");
62          Assert.True(File.Exists(bCppFile));
63          Assert.Equal("c c c", File.ReadAllText(bCppFile));
64          Assert.False(File.Exists(Path.Combine(targetFolderPath, "x.txt")));
65          Assert.False(File.Exists(Path.Combine(targetFolderPath, "x.cpp")));
66
67          Directory.Delete(sourceFolderPath, true);
68          Directory.Delete(targetFolderPath, true);
69      }
70    }
71  }
```

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       /// <summary>
7       /// <para>
8       /// Represents the markov algorithms tests.
9       /// </para>
10      /// <para></para>
11      /// </summary>
12      public class MarkovAlgorithmsTests
13      {
14          /// <remarks>
15          /// Example is from https://en.wikipedia.org/wiki/Markov_algorithm.
16          /// </remarks>
17          [Fact]
18          public void BinaryToUnaryNumbersTest()
19          {
20              var rules = new SubstitutionRule[]
21              {
22                  ("1", "0|", int.MaxValue),     // "1" -> "0|" repeated forever
23                  // | symbol should be escaped for regular expression pattern, but not in the
                  ↪   substitution pattern
24                  (@"\|0", "0||", int.MaxValue), // "\|0" -> "0||" repeated forever
25                  ("0", "", int.MaxValue),       // "0" -> "" repeated forever
26              };
27              var transformer = new TextTransformer(rules);
28              var input = "101";
29              var expectedOutput = "|||||";
30              var output = transformer.Transform(input);
31              Assert.Equal(expectedOutput, output);
32          }
33      }
34  }
```

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       /// <summary>
7       /// <para>
8       /// Represents the substitution rule tests.
9       /// </para>
10      /// <para></para>
11      /// </summary>
12      public class SubstitutionRuleTests
13      {
14          /// <summary>
15          /// <para>
16          /// Tests that options override test.
17          /// </para>
18          /// <para></para>
19          /// </summary>
20          [Fact]
21          public void OptionsOverrideTest()
22          {
23              SubstitutionRule rule = ((new Regex(@"^\s*?\#pragma[\sa-zA-Z0-9\/]+$"), "", 0);
24              Assert.Equal(RegexOptions.Compiled | RegexOptions.Multiline,
                  ↪   rule.MatchPattern.Options);
```

```
25              }
26          }
27      }
```

```csharp
 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      /// <summary>
 9      /// <para>
10      /// Represents the text transformer tests.
11      /// </para>
12      /// <para></para>
13      /// </summary>
14      public class TextTransformerTests
15      {
16          /// <summary>
17          /// <para>
18          /// Tests that debug output test.
19          /// </para>
20          /// <para></para>
21          /// </summary>
22          [Fact]
23          public void DebugOutputTest()
24          {
25              var sourceText = "aaaa";
26              var firstStepReferenceText = "bbbb";
27              var secondStepReferenceText = "cccc";
28
29              var transformer = new TextTransformer(new SubstitutionRule[] {
30                  (new Regex("a"), "b"),
31                  (new Regex("b"), "c")
32              });
33
34              var steps = transformer.GetSteps(sourceText);
35
36              Assert.Equal(2, steps.Count);
37              Assert.Equal(firstStepReferenceText, steps[0]);
38              Assert.Equal(secondStepReferenceText, steps[1]);
39          }
40
41          /// <summary>
42          /// <para>
43          /// Tests that debug files output test.
44          /// </para>
45          /// <para></para>
46          /// </summary>
47          [Fact]
48          public void DebugFilesOutputTest()
49          {
50              var sourceText = "aaaa";
51              var firstStepReferenceText = "bbbb";
52              var secondStepReferenceText = "cccc";
53
54              var transformer = new TextTransformer(new SubstitutionRule[] {
55                  (new Regex("a"), "b"),
56                  (new Regex("b"), "c")
57              });
58
59              var targetFilename = Path.GetTempFileName();
60
61              transformer.WriteStepsToFiles(sourceText, $"{targetFilename}.txt",
                  ↪   skipFilesWithNoChanges: false);
62
63              CheckAndCleanUpTwoRulesFiles(firstStepReferenceText, secondStepReferenceText,
                  ↪   transformer, targetFilename);
64          }
65
66          /// <summary>
67          /// <para>
68          /// Checks the and clean up two rules files using the specified first step reference
                  ↪   text.
69          /// </para>
70          /// <para></para>
71          /// </summary>
```

```csharp
72          /// <param name="firstStepReferenceText">
73          /// <para>The first step reference text.</para>
74          /// <para></para>
75          /// </param>
76          /// <param name="secondStepReferenceText">
77          /// <para>The second step reference text.</para>
78          /// <para></para>
79          /// </param>
80          /// <param name="transformer">
81          /// <para>The transformer.</para>
82          /// <para></para>
83          /// </param>
84          /// <param name="targetFilename">
85          /// <para>The target filename.</para>
86          /// <para></para>
87          /// </param>
88          private static void CheckAndCleanUpTwoRulesFiles(string firstStepReferenceText, string
    ↪   secondStepReferenceText, TextTransformer transformer, string targetFilename)
89          {
90              var firstStepReferenceFilename = $"{targetFilename}.0.txt";
91              var firstStepRuleFilename = $"{targetFilename}.0.rule.txt";
92              var secondStepReferenceFilename = $"{targetFilename}.1.txt";
93              var secondStepRuleFilename = $"{targetFilename}.1.rule.txt";
94
95              Assert.True(File.Exists(firstStepReferenceFilename));
96              Assert.True(File.Exists(firstStepRuleFilename));
97              Assert.True(File.Exists(secondStepReferenceFilename));
98              Assert.True(File.Exists(secondStepRuleFilename));
99
100             Assert.Equal(firstStepReferenceText, File.ReadAllText(firstStepReferenceFilename,
    ↪   Encoding.UTF8));
101             Assert.Equal(transformer.Rules[0].ToString(),
    ↪   File.ReadAllText(firstStepRuleFilename, Encoding.UTF8));
102             Assert.Equal(secondStepReferenceText, File.ReadAllText(secondStepReferenceFilename,
    ↪   Encoding.UTF8));
103             Assert.Equal(transformer.Rules[1].ToString(),
    ↪   File.ReadAllText(secondStepRuleFilename, Encoding.UTF8));
104
105             File.Delete(firstStepReferenceFilename);
106             File.Delete(firstStepRuleFilename);
107             File.Delete(secondStepReferenceFilename);
108             File.Delete(secondStepRuleFilename);
109         }
110
111         /// <summary>
112         /// <para>
113         /// Tests that files with no changes skiped test.
114         /// </para>
115         /// <para></para>
116         /// </summary>
117         [Fact]
118         public void FilesWithNoChangesSkipedTest()
119         {
120             var sourceText = "aaaa";
121             var firstStepReferenceText = "bbbb";
122             var thirdStepReferenceText = "cccc";
123
124             var transformer = new TextTransformer(new SubstitutionRule[] {
125                 (new Regex("a"), "b"),
126                 (new Regex("x"), "y"),
127                 (new Regex("b"), "c")
128             });
129
130             var targetFilename = Path.GetTempFileName();
131
132             transformer.WriteStepsToFiles(sourceText, $"{targetFilename}.txt",
    ↪   skipFilesWithNoChanges: true);
133
134             CheckAndCleanUpThreeRulesFiles(firstStepReferenceText, thirdStepReferenceText,
    ↪   transformer, targetFilename);
135         }
136
137         /// <summary>
138         /// <para>
139         /// Checks the and clean up three rules files using the specified first step reference
    ↪   text.
140         /// </para>
141         /// <para></para>
```

```csharp
142        /// </summary>
143        /// <param name="firstStepReferenceText">
144        /// <para>The first step reference text.</para>
145        /// <para></para>
146        /// </param>
147        /// <param name="thirdStepReferenceText">
148        /// <para>The third step reference text.</para>
149        /// <para></para>
150        /// </param>
151        /// <param name="transformer">
152        /// <para>The transformer.</para>
153        /// <para></para>
154        /// </param>
155        /// <param name="targetFilename">
156        /// <para>The target filename.</para>
157        /// <para></para>
158        /// </param>
159        private static void CheckAndCleanUpThreeRulesFiles(string firstStepReferenceText, string
           ↪  thirdStepReferenceText, TextTransformer transformer, string targetFilename)
160        {
161            var firstStepReferenceFilename = $"{targetFilename}.0.txt";
162            var firstStepRuleFilename = $"{targetFilename}.0.rule.txt";
163            var secondStepReferenceFilename = $"{targetFilename}.1.txt";
164            var secondStepRuleFilename = $"{targetFilename}.1.rule.txt";
165            var thirdStepReferenceFilename = $"{targetFilename}.2.txt";
166            var thirdStepRuleFilename = $"{targetFilename}.2.rule.txt";
167
168            Assert.True(File.Exists(firstStepReferenceFilename));
169            Assert.True(File.Exists(firstStepReferenceFilename));
170            Assert.False(File.Exists(secondStepReferenceFilename));
171            Assert.False(File.Exists(secondStepRuleFilename));
172            Assert.True(File.Exists(thirdStepReferenceFilename));
173            Assert.True(File.Exists(thirdStepRuleFilename));
174
175            Assert.Equal(firstStepReferenceText, File.ReadAllText(firstStepReferenceFilename,
               ↪  Encoding.UTF8));
176            Assert.Equal(transformer.Rules[0].ToString(),
               ↪  File.ReadAllText(firstStepRuleFilename, Encoding.UTF8));
177            Assert.Equal(thirdStepReferenceText, File.ReadAllText(thirdStepReferenceFilename,
               ↪  Encoding.UTF8));
178            Assert.Equal(transformer.Rules[2].ToString(),
               ↪  File.ReadAllText(thirdStepRuleFilename, Encoding.UTF8));
179
180            File.Delete(firstStepReferenceFilename);
181            File.Delete(firstStepRuleFilename);
182            File.Delete(secondStepReferenceFilename);
183            File.Delete(secondStepRuleFilename);
184            File.Delete(thirdStepReferenceFilename);
185            File.Delete(thirdStepRuleFilename);
186        }
187
188        /// <summary>
189        /// <para>
190        /// Tests that debug output using transformers generation test.
191        /// </para>
192        /// <para></para>
193        /// </summary>
194        [Fact]
195        public void DebugOutputUsingTransformersGenerationTest()
196        {
197            var sourceText = "aaaa";
198            var firstStepReferenceText = "bbbb";
199            var secondStepReferenceText = "cccc";
200
201            var transformer = new TextTransformer(new SubstitutionRule[] {
202                (new Regex("a"), "b"),
203                (new Regex("b"), "c")
204            });
205
206            var steps =
               ↪  transformer.GenerateTransformersForEachRule().TransformWithAll(sourceText);
207
208            Assert.Equal(2, steps.Count);
209            Assert.Equal(firstStepReferenceText, steps[0]);
210            Assert.Equal(secondStepReferenceText, steps[1]);
211        }
212
213        /// <summary>
```

```csharp
            /// <para>
            /// Tests that debug files output using transformers generation test.
            /// </para>
            /// <para></para>
            /// </summary>
            [Fact]
            public void DebugFilesOutputUsingTransformersGenerationTest()
            {
                var sourceText = "aaaa";
                var firstStepReferenceText = "bbbb";
                var secondStepReferenceText = "cccc";

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

                var targetFilename = Path.GetTempFileName();

                transformer.GenerateTransformersForEachRule().TransformWithAllToFiles(sourceText,
                ↪   $"{targetFilename}.txt", skipFilesWithNoChanges: false);

                CheckAndCleanUpTwoRulesFiles(firstStepReferenceText, secondStepReferenceText,
                ↪   transformer, targetFilename);
            }

            /// <summary>
            /// <para>
            /// Tests that files with no changes skiped when using transformers generation test.
            /// </para>
            /// <para></para>
            /// </summary>
            [Fact]
            public void FilesWithNoChangesSkipedWhenUsingTransformersGenerationTest()
            {
                var sourceText = "aaaa";
                var firstStepReferenceText = "bbbb";
                var thirdStepReferenceText = "cccc";

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

                var targetFilename = Path.GetTempFileName();

                transformer.GenerateTransformersForEachRule().TransformWithAllToFiles(sourceText,
                ↪   $"{targetFilename}.txt", skipFilesWithNoChanges: true);

                CheckAndCleanUpThreeRulesFiles(firstStepReferenceText, thirdStepReferenceText,
                ↪   transformer, targetFilename);
            }
        }
    }
```

Index