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

```
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>
```

```
        /// <para></para>
        /// </param>
        /// <param name="sourceFileExtension">
        /// <para>A source file extension.</para>
        /// <para></para>
        /// </param>
        /// <param name="targetFileExtension">
        /// <para>A target file extension.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public FileTransformer(ITextTransformer textTransformer, string sourceFileExtension,
        ↪  string targetFileExtension)
        {
            _textTransformer = textTransformer;
            SourceFileExtension = sourceFileExtension;
            TargetFileExtension = targetFileExtension;
        }

        /// <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>
        /// <exception cref="NotSupportedException">
        /// <para></para>
        /// <para></para>
        /// </exception>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void Transform(string sourcePath, string targetPath)
        {
            var defaultPath = Path.GetFullPath(".");
            if (string.IsNullOrEmpty(sourcePath))
            {
                sourcePath = defaultPath;
            }
            if (string.IsNullOrEmpty(targetPath))
            {
                targetPath = defaultPath;
            }
            var sourceDirectoryExists = DirectoryExists(sourcePath);
            var sourceDirectoryPath = LooksLikeDirectoryPath(sourcePath);
            var sourceIsDirectory = sourceDirectoryExists || sourceDirectoryPath;
            var targetDirectoryExists = DirectoryExists(targetPath);
            var targetDirectoryPath = LooksLikeDirectoryPath(targetPath);
            var targetIsDirectory = targetDirectoryExists || targetDirectoryPath;
            if (sourceIsDirectory && targetIsDirectory)
            {
                // Folder -> Folder
                if (!sourceDirectoryExists)
                {
                    return;
                }
                TransformFolder(sourcePath, targetPath);
            }
            else if (!(sourceIsDirectory || targetIsDirectory))
            {
                // File -> File
                EnsureSourceFileExists(sourcePath);
                EnsureTargetFileDirectoryExists(targetPath);
                TransformFile(sourcePath, targetPath);
            }
            else if (targetIsDirectory)
            {
                // File -> Folder
                EnsureSourceFileExists(sourcePath);
                EnsureTargetDirectoryExists(targetPath, targetDirectoryExists);
                TransformFile(sourcePath, GetTargetFileName(sourcePath, targetPath));
            }
```

```
155             else
156             {
157                 // Folder -> File
158                 throw new NotSupportedException();
159             }
160         }
161 
162         /// <summary>
163         /// <para>
164         /// Transforms the folder using the specified source path.
165         /// </para>
166         /// <para></para>
167         /// </summary>
168         /// <param name="sourcePath">
169         /// <para>The source path.</para>
170         /// <para></para>
171         /// </param>
172         /// <param name="targetPath">
173         /// <para>The target path.</para>
174         /// <para></para>
175         /// </param>
176         [MethodImpl(MethodImplOptions.AggressiveInlining)]
177         protected virtual void TransformFolder(string sourcePath, string targetPath)
178         {
179             if (CountFilesRecursively(sourcePath, SourceFileExtension) == 0)
180             {
181                 return;
182             }
183             EnsureTargetDirectoryExists(targetPath);
184             var directories = Directory.GetDirectories(sourcePath);
185             for (var i = 0; i < directories.Length; i++)
186             {
187 #if NETSTANDARD2_1
188                 var relativePath = Path.GetRelativePath(sourcePath, directories[i]);
189 #else
190                 var relativePath =
                    ↪  directories[i].Replace(sourcePath.TrimEnd(Path.DirectorySeparatorChar) +
                    ↪  Path.DirectorySeparatorChar, "");
191 #endif
192                 var newTargetPath = Path.Combine(targetPath, relativePath);
193                 TransformFolder(directories[i], newTargetPath);
194             }
195             var files = Directory.GetFiles(sourcePath);
196             Parallel.For(0, files.Length, i =>
197             {
198                 var file = files[i];
199                 if (FileExtensionMatches(file, SourceFileExtension))
200                 {
201                     TransformFile(file, GetTargetFileName(file, targetPath));
202                 }
203             });
204         }
205 
206         /// <summary>
207         /// <para>
208         /// Transforms the file using the specified source path.
209         /// </para>
210         /// <para></para>
211         /// </summary>
212         /// <param name="sourcePath">
213         /// <para>The source path.</para>
214         /// <para></para>
215         /// </param>
216         /// <param name="targetPath">
217         /// <para>The target path.</para>
218         /// <para></para>
219         /// </param>
220         [MethodImpl(MethodImplOptions.AggressiveInlining)]
221         protected virtual void TransformFile(string sourcePath, string targetPath)
222         {
223             if (File.Exists(targetPath))
224             {
225                 var applicationPath = Process.GetCurrentProcess().MainModule.FileName;
226                 var targetFileLastUpdateDateTime = new FileInfo(targetPath).LastWriteTimeUtc;
227                 if (new FileInfo(sourcePath).LastWriteTimeUtc < targetFileLastUpdateDateTime &&
                    ↪  new FileInfo(applicationPath).LastWriteTimeUtc <
                    ↪  targetFileLastUpdateDateTime)
228                 {
```

```
                    return;
                }
            }
            var sourceText = File.ReadAllText(sourcePath, Encoding.UTF8);
            var targetText = _textTransformer.Transform(sourceText);
            File.WriteAllText(targetPath, targetText, Encoding.UTF8);
        }

        /// <summary>
        /// <para>
        /// Gets the target file name using the specified source path.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="sourcePath">
        /// <para>The source path.</para>
        /// <para></para>
        /// </param>
        /// <param name="targetDirectory">
        /// <para>The target directory.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The string</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected string GetTargetFileName(string sourcePath, string targetDirectory) =>
        ↪  Path.ChangeExtension(Path.Combine(targetDirectory, Path.GetFileName(sourcePath)),
        ↪  TargetFileExtension);

        /// <summary>
        /// <para>
        /// Counts the files recursively using the specified path.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="path">
        /// <para>The path.</para>
        /// <para></para>
        /// </param>
        /// <param name="extension">
        /// <para>The extension.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The result.</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static long CountFilesRecursively(string path, string extension)
        {
            var files = Directory.GetFiles(path);
            var directories = Directory.GetDirectories(path);
            var result = 0L;
            for (var i = 0; i < directories.Length; i++)
            {
                result += CountFilesRecursively(directories[i], extension);
            }
            for (var i = 0; i < files.Length; i++)
            {
                if (FileExtensionMatches(files[i], extension))
                {
                    result++;
                }
            }
            return result;
        }

        /// <summary>
        /// <para>
        /// Determines whether file extension matches.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="file">
        /// <para>The file.</para>
        /// <para></para>
```

```
305         /// </param>
306         /// <param name="extension">
307         /// <para>The extension.</para>
308         /// <para></para>
309         /// </param>
310         /// <returns>
311         /// <para>The bool</para>
312         /// <para></para>
313         /// </returns>
314         [MethodImpl(MethodImplOptions.AggressiveInlining)]
315         private static bool FileExtensionMatches(string file, string extension) =>
            ↪ file.EndsWith(extension, StringComparison.OrdinalIgnoreCase);
316
317         /// <summary>
318         /// <para>
319         /// Ensures the target file directory exists using the specified target path.
320         /// </para>
321         /// <para></para>
322         /// </summary>
323         /// <param name="targetPath">
324         /// <para>The target path.</para>
325         /// <para></para>
326         /// </param>
327         [MethodImpl(MethodImplOptions.AggressiveInlining)]
328         private static void EnsureTargetFileDirectoryExists(string targetPath)
329         {
330             if (!File.Exists(targetPath))
331             {
332                 EnsureDirectoryIsCreated(targetPath);
333             }
334         }
335
336         /// <summary>
337         /// <para>
338         /// Ensures the target directory exists using the specified target path.
339         /// </para>
340         /// <para></para>
341         /// </summary>
342         /// <param name="targetPath">
343         /// <para>The target path.</para>
344         /// <para></para>
345         /// </param>
346         [MethodImpl(MethodImplOptions.AggressiveInlining)]
347         private static void EnsureTargetDirectoryExists(string targetPath) =>
            ↪ EnsureTargetDirectoryExists(targetPath, DirectoryExists(targetPath));
348
349         /// <summary>
350         /// <para>
351         /// Ensures the target directory exists using the specified target path.
352         /// </para>
353         /// <para></para>
354         /// </summary>
355         /// <param name="targetPath">
356         /// <para>The target path.</para>
357         /// <para></para>
358         /// </param>
359         /// <param name="targetDirectoryExists">
360         /// <para>The target directory exists.</para>
361         /// <para></para>
362         /// </param>
363         [MethodImpl(MethodImplOptions.AggressiveInlining)]
364         private static void EnsureTargetDirectoryExists(string targetPath, bool
            ↪ targetDirectoryExists)
365         {
366             if (!targetDirectoryExists)
367             {
368                 Directory.CreateDirectory(targetPath);
369             }
370         }
371
372         /// <summary>
373         /// <para>
374         /// Ensures the source file exists using the specified source path.
375         /// </para>
376         /// <para></para>
377         /// </summary>
378         /// <param name="sourcePath">
379         /// <para>The source path.</para>
```

```
        /// <para></para>
        /// </param>
        /// <exception cref="FileNotFoundException">
        /// <para>Source file does not exists. </para>
        /// <para></para>
        /// </exception>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static void EnsureSourceFileExists(string sourcePath)
        {
            if (!File.Exists(sourcePath))
            {
                throw new FileNotFoundException("Source file does not exists.", sourcePath);
            }
        }

        /// <summary>
        /// <para>
        /// Ensures the directory is created 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 EnsureDirectoryIsCreated(string targetPath) =>
        ↪  Directory.CreateDirectory(Path.GetDirectoryName(targetPath));

        /// <summary>
        /// <para>
        /// Determines whether directory exists.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="path">
        /// <para>The path.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static bool DirectoryExists(string path) => Directory.Exists(path) &&
        ↪  File.GetAttributes(path).HasFlag(FileAttributes.Directory);

        /// <summary>
        /// <para>
        /// Determines whether looks like directory path.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="path">
        /// <para>The path.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static bool LooksLikeDirectoryPath(string path) =>
        ↪  path.EndsWith(Path.DirectorySeparatorChar.ToString()) ||
        ↪  path.EndsWith(Path.AltDirectorySeparatorChar.ToString());
    }
}
```

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

```
using System.Runtime.CompilerServices;

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

namespace Platform.RegularExpressions.Transformer
{
    /// <summary>
    /// <para>
    /// Defines the file transformer.
```

```
    /// </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

```
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>
```

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

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

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

```
using System.Runtime.CompilerServices;

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

namespace Platform.RegularExpressions.Transformer
{
    /// <summary>
    /// <para>
    /// Defines the text transformer.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="ITransformer"/>
    public interface ITextTransformer : ITransformer
    {
        /// <summary>
        /// <para>
        /// Transforms the source text.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="sourceText">
        /// <para>The source text.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The string</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        string Transform(string sourceText);
    }
}
```

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

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Runtime.CompilerServices;
using Platform.Collections;

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

namespace Platform.RegularExpressions.Transformer
{
    /// <summary>
    /// <para>
    /// Represents the text transformer extensions.
    /// </para>
    /// <para></para>
    /// </summary>
    public static class ITextTransformerExtensions
    {
        /// <summary>
```

```
        /// <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>
```

```
        /// <param name="targetPath">
        /// <para>The target path.</para>
        /// <para></para>
        /// </param>
        /// <param name="skipFilesWithNoChanges">
        /// <para>The skip files with no changes.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void WriteStepsToFiles(this ITextTransformer transformer, string
        ↪  sourceText, string targetPath, bool skipFilesWithNoChanges)
        {
            if (transformer != null && !transformer.Rules.IsNullOrEmpty())
            {
                targetPath.GetPathParts(out var directoryName, out var targetFilename, out var
                ↪  targetExtension);
                Steps.DeleteAllSteps(directoryName, targetFilename, targetExtension);
                var lastText = "";
                var steppedTransformer = new TextSteppedTransformer(transformer.Rules,
                ↪  sourceText);
                while (steppedTransformer.Next())
                {
                    var newText = steppedTransformer.Text;
                    Steps.WriteStep(transformer, directoryName, targetFilename, targetExtension,
                    ↪  steppedTransformer.Current, ref lastText, newText,
                    ↪  skipFilesWithNoChanges);
                }
            }
        }
    }
}
```

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

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

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

namespace Platform.RegularExpressions.Transformer
{
    /// <summary>
    /// <para>
    /// Represents the text transformers list extensions.
    /// </para>
    /// <para></para>
    /// </summary>
    public static class ITextTransformersListExtensions
    {
        /// <summary>
        /// <para>
        /// Transforms the with all using the specified transformers.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="transformers">
        /// <para>The transformers.</para>
        /// <para></para>
        /// </param>
        /// <param name="source">
        /// <para>The source.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>A list of string</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static IList<string> TransformWithAll(this IList<ITextTransformer> transformers,
        ↪  string source)
        {
            if (!transformers.IsNullOrEmpty())
            {
                var steps = new List<string>();
                for (int i = 0; i < transformers.Count; i++)
                {
                    steps.Add(transformers[i].Transform(source));
```

```
                }
                return steps;
            }
            else
            {
                return Array.Empty<string>();
            }
        }

        /// <summary>
        /// <para>
        /// Transforms the with all to files using the specified transformers.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="transformers">
        /// <para>The transformers.</para>
        /// <para></para>
        /// </param>
        /// <param name="sourceText">
        /// <para>The source text.</para>
        /// <para></para>
        /// </param>
        /// <param name="targetPath">
        /// <para>The target path.</para>
        /// <para></para>
        /// </param>
        /// <param name="skipFilesWithNoChanges">
        /// <para>The skip files with no changes.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void TransformWithAllToFiles(this IList<ITextTransformer> transformers,
        ↪  string sourceText, string targetPath, bool skipFilesWithNoChanges)
        {
            if (!transformers.IsNullOrEmpty())
            {
                targetPath.GetPathParts(out var directoryName, out var targetFilename, out var
                ↪  targetExtension);
                Steps.DeleteAllSteps(directoryName, targetFilename, targetExtension);
                var lastText = "";
                for (int i = 0; i < transformers.Count; i++)
                {
                    var transformer = transformers[i];
                    var newText = transformer.Transform(sourceText);
                    Steps.WriteStep(transformer, directoryName, targetFilename, targetExtension,
                    ↪  i, ref lastText, newText, skipFilesWithNoChanges);
                }
            }
        }
    }
}
```

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

```
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>
    /// Defines the transformer.
    /// </para>
    /// <para></para>
    /// </summary>
    public interface ITransformer
    {
        /// <summary>
        /// <para>
        /// Gets the rules value.
        /// </para>
        /// <para></para>
        /// </summary>
        IList<ISubstitutionRule> Rules
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
            get;
        }
    }
}
```

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

```
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 logging file transformer.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="FileTransformer"/>
    public class LoggingFileTransformer : FileTransformer
    {
        /// <summary>
        /// <para>
        /// Initializes a new <see cref="LoggingFileTransformer"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="textTransformer">
        /// <para>A text transformer.</para>
        /// <para></para>
        /// </param>
        /// <param name="sourceFileExtension">
        /// <para>A source file extension.</para>
        /// <para></para>
        /// </param>
        /// <param name="targetFileExtension">
        /// <para>A target file extension.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LoggingFileTransformer(ITextTransformer textTransformer, string
        ↪  sourceFileExtension, string targetFileExtension) : base(textTransformer,
        ↪  sourceFileExtension, targetFileExtension) { }

        /// <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 override void TransformFile(string sourcePath, string targetPath)
        {
            base.TransformFile(sourcePath, targetPath);
            // Logging
            var sourceText = File.ReadAllText(sourcePath, Encoding.UTF8);
            _textTransformer.WriteStepsToFiles(sourceText, targetPath, skipFilesWithNoChanges:
            ↪  true);
        }
    }
}
```

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

```
using System;
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>
    /// 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

```

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>
```

```
        /// <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

```
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.
```

```
    /// </para>
    /// <para></para>
    /// </summary>
    internal static class StringExtensions
    {
        /// <summary>
        /// <para>
        /// Gets the path parts using the specified path.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="path">
        /// <para>The path.</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>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        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));

        /// <summary>
        /// <para>
        /// Writes the to file using the specified text.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="text">
        /// <para>The text.</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>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void WriteToFile(this string text, string directoryName, string
        ↪  targetFilename) => File.WriteAllText(Path.Combine(directoryName, targetFilename),
        ↪  text, Encoding.UTF8);
    }
}
```

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

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

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

namespace Platform.RegularExpressions.Transformer
{
    /// <summary>
    /// <para>
    /// Represents the substitution rule.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="ISubstitutionRule"/>
    public class SubstitutionRule : ISubstitutionRule
    {
        /// <summary>
```

```
        /// <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>
```

```
        /// <para></para>
        /// </param>
        /// <param name="substitutionPattern">
        /// <para>A substitution pattern.</para>
        /// <para></para>
        /// </param>
        /// <param name="maximumRepeatCount">
        /// <para>A maximum repeat count.</para>
        /// <para></para>
        /// </param>
        /// <param name="matchPatternOptions">
        /// <para>A match pattern options.</para>
        /// <para></para>
        /// </param>
        /// <param name="matchTimeout">
        /// <para>A match timeout.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public SubstitutionRule(Regex matchPattern, string substitutionPattern, int
        ↪  maximumRepeatCount, RegexOptions? matchPatternOptions, TimeSpan? matchTimeout)
        {
            MatchPattern = matchPattern;
            SubstitutionPattern = substitutionPattern;
            MaximumRepeatCount = maximumRepeatCount;
            OverrideMatchPatternOptions(matchPatternOptions ?? matchPattern.Options,
            ↪  matchTimeout ?? matchPattern.MatchTimeout);
        }

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="SubstitutionRule"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="matchPattern">
        /// <para>A match pattern.</para>
        /// <para></para>
        /// </param>
        /// <param name="substitutionPattern">
        /// <para>A substitution pattern.</para>
        /// <para></para>
        /// </param>
        /// <param name="maximumRepeatCount">
        /// <para>A maximum repeat count.</para>
        /// <para></para>
        /// </param>
        /// <param name="useDefaultOptions">
        /// <para>A use default options.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public SubstitutionRule(Regex matchPattern, string substitutionPattern, int
        ↪  maximumRepeatCount, bool useDefaultOptions) : this(matchPattern,
        ↪  substitutionPattern, maximumRepeatCount, useDefaultOptions ?
        ↪  DefaultMatchPatternRegexOptions : (RegexOptions?)null, useDefaultOptions ?
        ↪  DefaultMatchTimeout : (TimeSpan?)null) { }

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="SubstitutionRule"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="matchPattern">
        /// <para>A match pattern.</para>
        /// <para></para>
        /// </param>
        /// <param name="substitutionPattern">
        /// <para>A substitution pattern.</para>
        /// <para></para>
        /// </param>
        /// <param name="maximumRepeatCount">
        /// <para>A maximum repeat count.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
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.
```

```
        /// </para>
        /// <para></para>
        /// </summary>
        /// <returns>
        /// <para>The string</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public override string ToString()
        {
            var sb = new StringBuilder();
            sb.Append('"');
            sb.Append(MatchPattern.ToString());
            sb.Append('"');
            sb.Append(" -> ");
            sb.Append('"');
            sb.Append(SubstitutionPattern);
            sb.Append('"');
            if (PathPattern != null)
            {
                sb.Append(" on files ");
                sb.Append('"');
                sb.Append(PathPattern.ToString());
                sb.Append('"');
            }
            if (MaximumRepeatCount > 0)
            {
                if (MaximumRepeatCount >= int.MaxValue)
                {
                    sb.Append(" repeated forever");
                }
                else
                {
                    sb.Append(" repeated up to ");
                    sb.Append(MaximumRepeatCount);
                    sb.Append(" times");
                }
            }
            return sb.ToString();
        }
    }
}
```

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

```
using System;
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 stepped transformer.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="ITransformer"/>
    public class TextSteppedTransformer : ITransformer
    {
        /// <summary>
        /// <para>
        /// Gets or sets the rules value.
        /// </para>
        /// <para></para>
        /// </summary>
        public IList<ISubstitutionRule> Rules
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            set;
        }

        /// <summary>
        /// <para>
        /// Gets or sets the text value.
        /// </para>
```

```
        /// <para></para>
        /// </summary>
        public string Text
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            set;
        }

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

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="TextSteppedTransformer"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="rules">
        /// <para>A rules.</para>
        /// <para></para>
        /// </param>
        /// <param name="text">
        /// <para>A text.</para>
        /// <para></para>
        /// </param>
        /// <param name="current">
        /// <para>A current.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TextSteppedTransformer(IList<ISubstitutionRule> rules, string text, int current)
        ↪  => Reset(rules, text, current);

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="TextSteppedTransformer"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="rules">
        /// <para>A rules.</para>
        /// <para></para>
        /// </param>
        /// <param name="text">
        /// <para>A text.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TextSteppedTransformer(IList<ISubstitutionRule> rules, string text) =>
        ↪  Reset(rules, text);

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="TextSteppedTransformer"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="rules">
        /// <para>A rules.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TextSteppedTransformer(IList<ISubstitutionRule> rules) => Reset(rules);

        /// <summary>
```

```
        /// <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>
```

```
        /// <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

```
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.
```

```
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="substitutionRules">
        /// <para>A substitution rules.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TextTransformer(IList<ISubstitutionRule> substitutionRules)
        {
            Rules = substitutionRules;
        }

        /// <summary>
        /// <para>
        /// Transforms the source.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="source">
        /// <para>The source.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The string</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public string Transform(string source)
        {
            var baseTrasformer = new TextSteppedTransformer(Rules);
            baseTrasformer.Reset(source);
            while (baseTrasformer.Next());
            return baseTrasformer.Text;
        }
    }
}
```

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

```
using System.Runtime.CompilerServices;
using Platform.Collections.Arrays;

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

namespace Platform.RegularExpressions.Transformer
{
    /// <summary>
    /// <para>
    /// Represents the transformer cli.
    /// </para>
    /// <para></para>
    /// </summary>
    public class TransformerCLI
    {
        /// <summary>
        /// <para>
        /// The transformer.
        /// </para>
        /// <para></para>
        /// </summary>
        private readonly IFileTransformer _transformer;

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="TransformerCLI"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="transformer">
        /// <para>A transformer.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TransformerCLI(IFileTransformer transformer) => _transformer = transformer;

        /// <summary>
        /// <para>
        /// Runs the args.
        /// </para>
```

```
        /// <para></para>
        /// </summary>
        /// <param name="args">
        /// <para>The args.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void Run(string[] args)
        {
            var sourcePath = args.GetElementOrDefault(0);
            var targetPath = args.GetElementOrDefault(1);
            _transformer.Transform(sourcePath, targetPath);
        }
    }
}
```

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

```
using System.IO;
using Xunit;

namespace Platform.RegularExpressions.Transformer.Tests
{
    /// <summary>
    /// <para>
    /// Represents the file transformer tests.
    /// </para>
    /// <para></para>
    /// </summary>
    public class FileTransformerTests
    {
        /// <summary>
        /// <para>
        /// Tests that folder to folder transfomation test.
        /// </para>
        /// <para></para>
        /// </summary>
        [Fact]
        public void FolderToFolderTransfomationTest()
        {
            var tempPath = Path.GetTempPath();
            var sourceFolderPath = Path.Combine(tempPath,
            ↪  "FileTransformerTestsFolderToFolderTransfomationTestSourceFolder");
            var targetFolderPath = Path.Combine(tempPath,
            ↪  "FileTransformerTestsFolderToFolderTransfomationTestTargetFolder");

            var baseTransformer = new TextTransformer(new SubstitutionRule[]
            {
                ("a", "b"),
                ("b", "c")
            });
            var fileTransformer = new FileTransformer(baseTransformer, ".cs", ".cpp");

            // Delete before creation (if previous test failed)
            if (Directory.Exists(sourceFolderPath))
            {
                Directory.Delete(sourceFolderPath, true);
            }
            if (Directory.Exists(targetFolderPath))
            {
                Directory.Delete(targetFolderPath, true);
            }

            Directory.CreateDirectory(sourceFolderPath);
            Directory.CreateDirectory(targetFolderPath);

            File.WriteAllText(Path.Combine(sourceFolderPath, "a.cs"), "a a a");
            var aFolderPath = Path.Combine(sourceFolderPath, "A");
            Directory.CreateDirectory(aFolderPath);
            Directory.CreateDirectory(Path.Combine(sourceFolderPath, "B"));
            File.WriteAllText(Path.Combine(aFolderPath, "b.cs"), "b b b");
            File.WriteAllText(Path.Combine(sourceFolderPath, "x.txt"), "should not be
            ↪  translated");

            fileTransformer.Transform(sourceFolderPath,
            ↪  $"{targetFolderPath}{Path.DirectorySeparatorChar}");

            var aCppFile = Path.Combine(targetFolderPath, "a.cpp");
            Assert.True(File.Exists(aCppFile));
```

```
            Assert.Equal("c c c", File.ReadAllText(aCppFile));
            Assert.True(Directory.Exists(Path.Combine(targetFolderPath, "A")));
            Assert.False(Directory.Exists(Path.Combine(targetFolderPath, "B")));
            var bCppFile = Path.Combine(targetFolderPath, "A", "b.cpp");
            Assert.True(File.Exists(bCppFile));
            Assert.Equal("c c c", File.ReadAllText(bCppFile));
            Assert.False(File.Exists(Path.Combine(targetFolderPath, "x.txt")));
            Assert.False(File.Exists(Path.Combine(targetFolderPath, "x.cpp")));

            Directory.Delete(sourceFolderPath, true);
            Directory.Delete(targetFolderPath, true);
        }
    }
}
```

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

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

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

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

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

namespace Platform.RegularExpressions.Transformer.Tests
{
    /// <summary>
    /// <para>
    /// Represents the substitution rule tests.
    /// </para>
    /// <para></para>
    /// </summary>
    public class SubstitutionRuleTests
    {
        /// <summary>
        /// <para>
        /// Tests that options override test.
        /// </para>
        /// <para></para>
        /// </summary>
        [Fact]
        public void OptionsOverrideTest()
        {
            SubstitutionRule rule = (new Regex(@"^\s*?\#pragma[\sa-zA-Z0-9\/]+$"), "", 0);
            Assert.Equal(RegexOptions.Compiled | RegexOptions.Multiline,
                rule.MatchPattern.Options);
```

```
        }
    }
}
```

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

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

namespace Platform.RegularExpressions.Transformer.Tests
{
    /// <summary>
    /// <para>
    /// Represents the text transformer tests.
    /// </para>
    /// <para></para>
    /// </summary>
    public class TextTransformerTests
    {
        /// <summary>
        /// <para>
        /// Tests that debug output test.
        /// </para>
        /// <para></para>
        /// </summary>
        [Fact]
        public void DebugOutputTest()
        {
            var sourceText = "aaaa";
            var firstStepReferenceText = "bbbb";
            var secondStepReferenceText = "cccc";

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

            var steps = transformer.GetSteps(sourceText);

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

        /// <summary>
        /// <para>
        /// Tests that debug files output test.
        /// </para>
        /// <para></para>
        /// </summary>
        [Fact]
        public void DebugFilesOutputTest()
        {
            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.WriteStepsToFiles(sourceText, $"{targetFilename}.txt",
            ↪  skipFilesWithNoChanges: false);

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

        /// <summary>
        /// <para>
        /// Checks the and clean up two rules files using the specified first step reference
        ↪  text.
        /// </para>
        /// <para></para>
        /// </summary>
```

```
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>
```

```
        /// </summary>
        /// <param name="firstStepReferenceText">
        /// <para>The first step reference text.</para>
        /// <para></para>
        /// </param>
        /// <param name="thirdStepReferenceText">
        /// <para>The third step reference text.</para>
        /// <para></para>
        /// </param>
        /// <param name="transformer">
        /// <para>The transformer.</para>
        /// <para></para>
        /// </param>
        /// <param name="targetFilename">
        /// <para>The target filename.</para>
        /// <para></para>
        /// </param>
        private static void CheckAndCleanUpThreeRulesFiles(string firstStepReferenceText, string thirdStepReferenceText, TextTransformer transformer, string targetFilename)
        {
            var firstStepReferenceFilename = $"{targetFilename}.0.txt";
            var firstStepRuleFilename = $"{targetFilename}.0.rule.txt";
            var secondStepReferenceFilename = $"{targetFilename}.1.txt";
            var secondStepRuleFilename = $"{targetFilename}.1.rule.txt";
            var thirdStepReferenceFilename = $"{targetFilename}.2.txt";
            var thirdStepRuleFilename = $"{targetFilename}.2.rule.txt";

            Assert.True(File.Exists(firstStepReferenceFilename));
            Assert.True(File.Exists(firstStepReferenceFilename));
            Assert.False(File.Exists(secondStepReferenceFilename));
            Assert.False(File.Exists(secondStepRuleFilename));
            Assert.True(File.Exists(thirdStepReferenceFilename));
            Assert.True(File.Exists(thirdStepRuleFilename));

            Assert.Equal(firstStepReferenceText, File.ReadAllText(firstStepReferenceFilename, Encoding.UTF8));
            Assert.Equal(transformer.Rules[0].ToString(), File.ReadAllText(firstStepRuleFilename, Encoding.UTF8));
            Assert.Equal(thirdStepReferenceText, File.ReadAllText(thirdStepReferenceFilename, Encoding.UTF8));
            Assert.Equal(transformer.Rules[2].ToString(), File.ReadAllText(thirdStepRuleFilename, Encoding.UTF8));

            File.Delete(firstStepReferenceFilename);
            File.Delete(firstStepRuleFilename);
            File.Delete(secondStepReferenceFilename);
            File.Delete(secondStepRuleFilename);
            File.Delete(thirdStepReferenceFilename);
            File.Delete(thirdStepRuleFilename);
        }

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

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

            var steps = transformer.GenerateTransformersForEachRule().TransformWithAll(sourceText);

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

        /// <summary>
```

```
        /// <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