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
{
    public class FileTransformer : IFileTransformer
    {
        protected readonly ITextTransformer _textTransformer;

        public string SourceFileExtension
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            private set;
        }

        public string TargetFileExtension
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            private set;
        }

        public IList<ISubstitutionRule> Rules
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get => _textTransformer.Rules;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public FileTransformer(ITextTransformer textTransformer, string sourceFileExtension,
        ↪  string targetFileExtension)
        {
            _textTransformer = textTransformer;
            SourceFileExtension = sourceFileExtension;
            TargetFileExtension = targetFileExtension;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void Transform(string sourcePath, string targetPath)
        {
            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;
                }
                EnsureTargetDirectoryExists(targetDirectoryExists, targetPath);
                TransformFolder(sourcePath, targetPath);
            }
            else if (!(sourceIsDirectory || targetIsDirectory))
            {
                // File -> File
                EnsureSourceFileExists(sourcePath);
                EnsureTargetDirectoryExists(targetPath);
                TransformFile(sourcePath, targetPath);
            }
            else if (targetIsDirectory)
            {
                // File -> Folder
                EnsureSourceFileExists(sourcePath);
                EnsureTargetDirectoryExists(targetDirectoryExists, targetPath);
```

```
                TransformFile(sourcePath, GetTargetFileName(sourcePath, targetPath));
            }
            else
            {
                // Folder -> File
                throw new NotSupportedException();
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual void TransformFolder(string sourcePath, string targetPath)
        {
            var directories = Directory.GetDirectories(sourcePath);
            for (var i = 0; i < directories.Length; i++)
            {
                var relativePath = GetRelativePath(sourcePath, directories[i]);
                var newTargetPath = Path.Combine(targetPath, relativePath);
                TransformFolder(directories[i], newTargetPath);
            }
            var files = Directory.GetFiles(sourcePath);
            Parallel.For(0, files.Length, i =>
            {
                TransformFile(files[i], GetTargetFileName(files[i], targetPath));
            });
        }

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected string GetTargetFileName(string sourcePath, string targetDirectory) =>
            Path.ChangeExtension(Path.Combine(targetDirectory, Path.GetFileName(sourcePath)),
            TargetFileExtension);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static void EnsureTargetDirectoryExists(string targetPath)
        {
            if (!File.Exists(targetPath))
            {
                EnsureDirectoryIsCreated(targetPath);
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static void EnsureTargetDirectoryExists(bool targetDirectoryExists, string
            targetPath)
        {
            if (!targetDirectoryExists)
            {
                Directory.CreateDirectory(targetPath);
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static void EnsureSourceFileExists(string sourcePath)
        {
            if (!File.Exists(sourcePath))
            {
                throw new FileNotFoundException("Source file does not exists.", sourcePath);
            }
        }

```

```
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static string NormalizePath(string path) => Path.GetFullPath(path).TrimEnd(new[]
        ↪  { Path.DirectorySeparatorChar, Path.AltDirectorySeparatorChar });

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static string GetRelativePath(string rootPath, string fullPath)
        {
            rootPath = NormalizePath(rootPath);
            fullPath = NormalizePath(fullPath);
            if (!fullPath.StartsWith(rootPath))
            {
                throw new Exception("Could not find rootPath in fullPath when calculating
                ↪  relative path.");
            }
            return fullPath.Substring(rootPath.Length);
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static void EnsureDirectoryIsCreated(string targetPath) =>
        ↪  Directory.CreateDirectory(Path.GetDirectoryName(targetPath));

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static bool DirectoryExists(string path) => Directory.Exists(path) &&
        ↪  File.GetAttributes(path).HasFlag(FileAttributes.Directory);

        [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
{
    public interface IFileTransformer : ITransformer
    {
        string SourceFileExtension
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
        }

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

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

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

```
21
22          int MaximumRepeatCount
23          {
24              [MethodImpl(MethodImplOptions.AggressiveInlining)]
25              get;
26          }
27      }
28  }
```

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

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

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

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

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

```
1   using System.IO;
2   using System.Collections.Generic;
3   using System.Text;
4   using System.Runtime.CompilerServices;
5
6   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8   namespace Platform.RegularExpressions.Transformer
9   {
10      public static class ITextTransformersListExtensions
11      {
12          [MethodImpl(MethodImplOptions.AggressiveInlining)]
13          public static List<string> TransformWithAll(this IList<ITextTransformer> transformers,
            ↪  string source)
14          {
15              var strings = new List<string>();
16              if (transformers.Count > 0)
17              {
18                  for (int i = 0; i < transformers.Count; i++)
```

```
                {
                    strings.Add(transformers[i].Transform(source));
                }
            }
            return strings;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void TransformWithAllToFiles(this IList<ITextTransformer> transformers,
        ↪  string sourceText, string targetPath, bool skipFilesWithNoChanges)
        {
            if (transformers.Count > 0)
            {
                var directoryName = Path.GetDirectoryName(targetPath);
                var targetFilename = Path.GetFileNameWithoutExtension(targetPath);
                var targetExtension = Path.GetExtension(targetPath);
                var lastText = "";
                for (int i = 0; i < transformers.Count; i++)
                {
                    var transformationOutput = transformers[i].Transform(sourceText);
                    if (!(skipFilesWithNoChanges && string.Equals(lastText,
                    ↪  transformationOutput)))
                    {
                        lastText = transformationOutput;
                        File.WriteAllText(Path.Combine(directoryName,
                        ↪  $"{targetFilename}.{i}{targetExtension}"), transformationOutput,
                        ↪  Encoding.UTF8);
                    }
                }
            }
        }
    }
}
```

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
{
    public interface ITransformer
    {
        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
{
    public class LoggingFileTransformer : FileTransformer
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LoggingFileTransformer(ITextTransformer textTransformer, string
        ↪  sourceFileExtension, string targetFileExtension) : base(textTransformer,
        ↪  sourceFileExtension, targetFileExtension) { }

        [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
{
    public static class RegexExtensions
    {
        [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/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
{
    public class SubstitutionRule : ISubstitutionRule
    {
        public static readonly TimeSpan DefaultMatchTimeout = TimeSpan.FromMinutes(5);
        public static readonly RegexOptions DefaultMatchPatternRegexOptions =
        ↪  RegexOptions.Compiled | RegexOptions.Multiline;

        public Regex MatchPattern
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            set;
        }

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

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

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

        [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);
```

```
54          }
55
56          [MethodImpl(MethodImplOptions.AggressiveInlining)]
57          public SubstitutionRule(Regex matchPattern, string substitutionPattern, int
            ↪  maximumRepeatCount, bool useDefaultOptions) : this(matchPattern,
            ↪  substitutionPattern, maximumRepeatCount, useDefaultOptions ?
            ↪  DefaultMatchPatternRegexOptions : (RegexOptions?)null, useDefaultOptions ?
            ↪  DefaultMatchTimeout : (TimeSpan?)null) { }
58
59          [MethodImpl(MethodImplOptions.AggressiveInlining)]
60          public SubstitutionRule(Regex matchPattern, string substitutionPattern, int
            ↪  maximumRepeatCount) : this(matchPattern, substitutionPattern, maximumRepeatCount,
            ↪  true) { }
61
62          [MethodImpl(MethodImplOptions.AggressiveInlining)]
63          public SubstitutionRule(Regex matchPattern, string substitutionPattern) :
            ↪  this(matchPattern, substitutionPattern, 0) { }
64
65          [MethodImpl(MethodImplOptions.AggressiveInlining)]
66          public static implicit operator SubstitutionRule(ValueTuple<string, string> tuple) =>
            ↪  new SubstitutionRule(new Regex(tuple.Item1), tuple.Item2);
67
68          [MethodImpl(MethodImplOptions.AggressiveInlining)]
69          public static implicit operator SubstitutionRule(ValueTuple<Regex, string> tuple) => new
            ↪  SubstitutionRule(tuple.Item1, tuple.Item2);
70
71          [MethodImpl(MethodImplOptions.AggressiveInlining)]
72          public static implicit operator SubstitutionRule(ValueTuple<string, string, int> tuple)
            ↪  => new SubstitutionRule(new Regex(tuple.Item1), tuple.Item2, tuple.Item3);
73
74          [MethodImpl(MethodImplOptions.AggressiveInlining)]
75          public static implicit operator SubstitutionRule(ValueTuple<Regex, string, int> tuple)
            ↪  => new SubstitutionRule(tuple.Item1, tuple.Item2, tuple.Item3);
76
77          [MethodImpl(MethodImplOptions.AggressiveInlining)]
78          public void OverrideMatchPatternOptions(RegexOptions options, TimeSpan matchTimeout) =>
            ↪  MatchPattern = MatchPattern.OverrideOptions(options, matchTimeout);
79
80          [MethodImpl(MethodImplOptions.AggressiveInlining)]
81          public void OverridePathPatternOptions(RegexOptions options, TimeSpan matchTimeout) =>
            ↪  PathPattern = PathPattern.OverrideOptions(options, matchTimeout);
82
83          [MethodImpl(MethodImplOptions.AggressiveInlining)]
84          public override string ToString()
85          {
86              var sb = new StringBuilder();
87              sb.Append('"');
88              sb.Append(MatchPattern.ToString());
89              sb.Append('"');
90              sb.Append(" -> ");
91              sb.Append('"');
92              sb.Append(SubstitutionPattern);
93              sb.Append('"');
94              if (PathPattern != null)
95              {
96                  sb.Append(" on files ");
97                  sb.Append('"');
98                  sb.Append(PathPattern.ToString());
99                  sb.Append('"');
100             }
101             if (MaximumRepeatCount > 0)
102             {
103                 if (MaximumRepeatCount >= int.MaxValue)
104                 {
105                     sb.Append(" repeated forever");
106                 }
107                 else
108                 {
109                     sb.Append(" repeated up to ");
110                     sb.Append(MaximumRepeatCount);
111                     sb.Append(" times");
112                 }
113             }
114             return sb.ToString();
115         }
116     }
117 }
```

1.11 ./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
{
    public class TextTransformer : ITextTransformer
    {
        public IList<ISubstitutionRule> Rules
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            private set;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TextTransformer(IList<ISubstitutionRule> substitutionRules) => Rules =
        ↪  substitutionRules;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public string Transform(string source)
        {
            var current = source;
            for (var i = 0; i < Rules.Count; i++)
            {
                var rule = Rules[i];
                var matchPattern = rule.MatchPattern;
                var substitutionPattern = rule.SubstitutionPattern;
                var maximumRepeatCount = rule.MaximumRepeatCount;
                var replaceCount = 0;
                do
                {
                    current = matchPattern.Replace(current, substitutionPattern);
                    replaceCount++;
                    if (maximumRepeatCount < int.MaxValue && replaceCount > maximumRepeatCount)
                    {
                        break;
                    }
                }
                while (matchPattern.IsMatch(current));
            }
            return current;
        }
    }
}
```

1.12 ./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
{
    public class TransformerCLI
    {
        private readonly IFileTransformer _transformer;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TransformerCLI(IFileTransformer transformer) => _transformer = transformer;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void Run(string[] args)
        {
            var sourcePath = args.GetElementOrDefault(0);
            var targetPath = args.GetElementOrDefault(1);
            _transformer.Transform(sourcePath, targetPath);
        }
    }
}
```

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

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

namespace Platform.RegularExpressions.Transformer.Tests
{
```

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

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

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

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

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

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

namespace Platform.RegularExpressions.Transformer.Tests
{
    public class TransformersTests
    {
        [Fact]
        public void DebugOutputTest()
        {
            var sourceText = "aaaa";
            var firstStepReferenceText = "bbbb";
            var secondStepReferenceText = "cccc";

            var transformer = new 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]);
        }

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

```

```
36              var transformer = new TextTransformer(new SubstitutionRule[] {
37                  (new Regex("a"), "b"),
38                  (new Regex("b"), "c")
39              });
40
41              var targetFilename = Path.GetTempFileName();
42
43              transformer.WriteStepsToFiles(sourceText, $"{targetFilename}.txt",
                ↪  skipFilesWithNoChanges: false);
44
45              var firstStepReferenceFilename = $"{targetFilename}.0.txt";
46              var secondStepReferenceFilename = $"{targetFilename}.1.txt";
47
48              Assert.True(File.Exists(firstStepReferenceFilename));
49              Assert.True(File.Exists(secondStepReferenceFilename));
50
51              Assert.Equal(firstStepReferenceText, File.ReadAllText(firstStepReferenceFilename,
                ↪  Encoding.UTF8));
52              Assert.Equal(secondStepReferenceText, File.ReadAllText(secondStepReferenceFilename,
                ↪  Encoding.UTF8));
53
54              File.Delete(firstStepReferenceFilename);
55              File.Delete(secondStepReferenceFilename);
56          }
57
58          [Fact]
59          public void FilesWithNoChangesSkipedTest()
60          {
61              var sourceText = "aaaa";
62              var firstStepReferenceText = "bbbb";
63              var thirdStepReferenceText = "cccc";
64
65              var transformer = new TextTransformer(new SubstitutionRule[] {
66                  (new Regex("a"), "b"),
67                  (new Regex("x"), "y"),
68                  (new Regex("b"), "c")
69              });
70
71              var targetFilename = Path.GetTempFileName();
72
73              transformer.WriteStepsToFiles(sourceText, $"{targetFilename}.txt",
                ↪  skipFilesWithNoChanges: true);
74
75              var firstStepReferenceFilename = $"{targetFilename}.0.txt";
76              var secondStepReferenceFilename = $"{targetFilename}.1.txt";
77              var thirdStepReferenceFilename = $"{targetFilename}.2.txt";
78
79              Assert.True(File.Exists(firstStepReferenceFilename));
80              Assert.False(File.Exists(secondStepReferenceFilename));
81              Assert.True(File.Exists(thirdStepReferenceFilename));
82
83              Assert.Equal(firstStepReferenceText, File.ReadAllText(firstStepReferenceFilename,
                ↪  Encoding.UTF8));
84              Assert.Equal(thirdStepReferenceText, File.ReadAllText(thirdStepReferenceFilename,
                ↪  Encoding.UTF8));
85
86              File.Delete(firstStepReferenceFilename);
87              File.Delete(secondStepReferenceFilename);
88              File.Delete(thirdStepReferenceFilename);
89          }
90      }
91  }
```

Index