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;
                }
                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));
```

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual void TransformFolder(string sourcePath, string targetPath)
        {
            if (CountFilesRecursively(sourcePath, SourceFileExtension) == 0)
            {
                return;
            }
            EnsureTargetDirectoryExists(targetPath);
            var directories = Directory.GetDirectories(sourcePath);
            for (var i = 0; i < directories.Length; i++)
            {
                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 =>
            {
                var file = files[i];
                if (FileExtensionMatches(file, SourceFileExtension))
                {
                    TransformFile(file, GetTargetFileName(file, 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 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;
        }

```

```
152         [MethodImpl(MethodImplOptions.AggressiveInlining)]
153         private static bool FileExtensionMatches(string file, string extension) =>
            ↪ file.EndsWith(extension, StringComparison.OrdinalIgnoreCase);
154
155         [MethodImpl(MethodImplOptions.AggressiveInlining)]
156         private static void EnsureTargetFileDirectoryExists(string targetPath)
157         {
158             if (!File.Exists(targetPath))
159             {
160                 EnsureDirectoryIsCreated(targetPath);
161             }
162         }
163
164         [MethodImpl(MethodImplOptions.AggressiveInlining)]
165         private static void EnsureTargetDirectoryExists(string targetPath) =>
            ↪ EnsureTargetDirectoryExists(targetPath, DirectoryExists(targetPath));
166
167         [MethodImpl(MethodImplOptions.AggressiveInlining)]
168         private static void EnsureTargetDirectoryExists(string targetPath, bool
            ↪ targetDirectoryExists)
169         {
170             if (!targetDirectoryExists)
171             {
172                 Directory.CreateDirectory(targetPath);
173             }
174         }
175
176         [MethodImpl(MethodImplOptions.AggressiveInlining)]
177         private static void EnsureSourceFileExists(string sourcePath)
178         {
179             if (!File.Exists(sourcePath))
180             {
181                 throw new FileNotFoundException("Source file does not exists.", sourcePath);
182             }
183         }
184
185         [MethodImpl(MethodImplOptions.AggressiveInlining)]
186         private static string NormalizePath(string path) => Path.GetFullPath(path).TrimEnd(new[]
            ↪ { Path.DirectorySeparatorChar, Path.AltDirectorySeparatorChar });
187
188         [MethodImpl(MethodImplOptions.AggressiveInlining)]
189         private static string GetRelativePath(string rootPath, string fullPath)
190         {
191             rootPath = NormalizePath(rootPath);
192             fullPath = NormalizePath(fullPath);
193             if (!fullPath.StartsWith(rootPath))
194             {
195                 throw new Exception("Could not find rootPath in fullPath when calculating
                    ↪  relative path.");
196             }
197             return fullPath.Substring(rootPath.Length + 1);
198         }
199
200         [MethodImpl(MethodImplOptions.AggressiveInlining)]
201         private static void EnsureDirectoryIsCreated(string targetPath) =>
            ↪ Directory.CreateDirectory(Path.GetDirectoryName(targetPath));
202
203         [MethodImpl(MethodImplOptions.AggressiveInlining)]
204         private static bool DirectoryExists(string path) => Directory.Exists(path) &&
            ↪ File.GetAttributes(path).HasFlag(FileAttributes.Directory);
205
206         [MethodImpl(MethodImplOptions.AggressiveInlining)]
207         private static bool LooksLikeDirectoryPath(string path) =>
            ↪ path.EndsWith(Path.DirectorySeparatorChar.ToString()) ||
            ↪ path.EndsWith(Path.AltDirectorySeparatorChar.ToString());
208     }
209 }
```

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

```
1  using System.Runtime.CompilerServices;
2
3  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5  namespace Platform.RegularExpressions.Transformer
6  {
7      public interface IFileTransformer : ITransformer
8      {
9          string SourceFileExtension
```

```
        {
            [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;
        }

        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
{
    public interface ITextTransformer : ITransformer
    {
        [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
{
    public static class ITextTransformerExtensions
    {
        [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;
        }

        [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>();
            }
        }

        [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);
                var lastText = "";
                var steppedTransformer = new TextSteppedTransformer(transformer.Rules,
                ↪  sourceText);
                while (steppedTransformer.Next())
                {
                    var newText = steppedTransformer.Text;
                    if (!(skipFilesWithNoChanges && string.Equals(lastText, newText)))
                    {
                        lastText = newText;
                        newText.WriteStepToFile(directoryName, targetFilename, targetExtension,
                        ↪  steppedTransformer.Current);
                    }
                }
            }
        }
    }
}
```

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
{
    public static class ITextTransformersListExtensions
    {
        [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>();
            }
        }

        [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);
                var lastText = "";
                for (int i = 0; i < transformers.Count; i++)
                {
                    var transformationOutput = transformers[i].Transform(sourceText);
                    if (!(skipFilesWithNoChanges && string.Equals(lastText,
                    ↪  transformationOutput)))
                    {
                        lastText = transformationOutput;
                        transformationOutput.WriteStepToFile(directoryName, targetFilename,
                        ↪  targetExtension, i);
                    }
                }
            }
        }
    }
}
```

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/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
{
    internal static class StringExtensions
    {
        [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));

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void WriteStepToFile(this string text, string directoryName, string targetFilename, string targetExtension, int currentStep) => File.WriteAllText(Path.Combine(directoryName, $"{targetFilename}.{currentStep}{targetExtension}"), text, Encoding.UTF8);
    }
}
```

1.11 ./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);
        }

        [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) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public SubstitutionRule(Regex matchPattern, string substitutionPattern, int
        ↪  maximumRepeatCount) : this(matchPattern, substitutionPattern, maximumRepeatCount,
        ↪  true) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public SubstitutionRule(Regex matchPattern, string substitutionPattern) :
        ↪  this(matchPattern, substitutionPattern, 0) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static implicit operator SubstitutionRule(ValueTuple<string, string> tuple) =>
        ↪  new SubstitutionRule(new Regex(tuple.Item1), tuple.Item2);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static implicit operator SubstitutionRule(ValueTuple<Regex, string> tuple) => new
        ↪  SubstitutionRule(tuple.Item1, tuple.Item2);

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void OverrideMatchPatternOptions(RegexOptions options, TimeSpan matchTimeout) =>
        ↪  MatchPattern = MatchPattern.OverrideOptions(options, matchTimeout);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void OverridePathPatternOptions(RegexOptions options, TimeSpan matchTimeout) =>
        ↪  PathPattern = PathPattern.OverrideOptions(options, matchTimeout);

        [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();
        }
    }
}
```

1.12 ./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
{
    public class TextSteppedTransformer : ITransformer
    {
        public IList<ISubstitutionRule> Rules
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            set;
        }

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TextSteppedTransformer(IList<ISubstitutionRule> rules, string text, int current)
        ↪  => Reset(rules, text, current);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TextSteppedTransformer(IList<ISubstitutionRule> rules, string text) =>
        ↪  Reset(rules, text);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TextSteppedTransformer(IList<ISubstitutionRule> rules) => Reset(rules);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TextSteppedTransformer() => Reset();

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void Reset(IList<ISubstitutionRule> rules, string text, int current)
        {
            Rules = rules;
            Text = text;
            Current = current;
        }
```

```

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void Reset(IList<ISubstitutionRule> rules, string text) => Reset(rules, text, -1);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void Reset(IList<ISubstitutionRule> rules) => Reset(rules, "", -1);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void Reset(string text) => Reset(Rules, text, -1);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void Reset() => Reset(Array.Empty<ISubstitutionRule>(), "", -1);

        [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;
        }
    }
}
```

1.13 ./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 baseTrasformer = new TextSteppedTransformer(Rules);
            baseTrasformer.Reset(source);
            while (baseTrasformer.Next());
            return baseTrasformer.Text;
        }
    }
}
```

1.14 ./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);
        }
    }
}
```

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

```
using System.IO;
using Xunit;

namespace Platform.RegularExpressions.Transformer.Tests
{
    public class FileTransformerTests
    {
        [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);
        }
    }
}
```

1.16 ./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);
        }
    }
}
```

1.17 ./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);
        }
    }
}
```

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

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

namespace Platform.RegularExpressions.Transformer.Tests
{
    public class TextTransformerTests
    {
        [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);
```

```
23
24              Assert.Equal(2, steps.Count);
25              Assert.Equal(firstStepReferenceText, steps[0]);
26              Assert.Equal(secondStepReferenceText, steps[1]);
27          }
28
29          [Fact]
30          public void DebugFilesOutputTest()
31          {
32              var sourceText = "aaaa";
33              var firstStepReferenceText = "bbbb";
34              var secondStepReferenceText = "cccc";
35
36              var transformer = new TextTransformer(new SubstitutionRule[] {
37                  (new Regex("a"), "b"),
38                  (new Regex("b"), "c")
39              });
40
41              var targetFilename = Path.GetTempFileName();
42
43              transformer.WriteStepsToFiles(sourceText, $"{targetFilename}.txt",
                ↪  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          [Fact]
92          public void DebugOutputUsingTransformersGenerationTest()
93          {
94              var sourceText = "aaaa";
95              var firstStepReferenceText = "bbbb";
```

```
96              var secondStepReferenceText = "cccc";
97
98              var transformer = new TextTransformer(new SubstitutionRule[] {
99                  (new Regex("a"), "b"),
100                 (new Regex("b"), "c")
101             });
102
103             var steps =
                ↪  transformer.GenerateTransformersForEachRule().TransformWithAll(sourceText);
104
105             Assert.Equal(2, steps.Count);
106             Assert.Equal(firstStepReferenceText, steps[0]);
107             Assert.Equal(secondStepReferenceText, steps[1]);
108         }
109
110         [Fact]
111         public void DebugFilesOutputUsingTransformersGenerationTest()
112         {
113             var sourceText = "aaaa";
114             var firstStepReferenceText = "bbbb";
115             var secondStepReferenceText = "cccc";
116
117             var transformer = new TextTransformer(new SubstitutionRule[] {
118                 (new Regex("a"), "b"),
119                 (new Regex("b"), "c")
120             });
121
122             var targetFilename = Path.GetTempFileName();
123
124             transformer.GenerateTransformersForEachRule().TransformWithAllToFiles(sourceText,
                ↪  $"{targetFilename}.txt", skipFilesWithNoChanges: false);
125
126             var firstStepReferenceFilename = $"{targetFilename}.0.txt";
127             var secondStepReferenceFilename = $"{targetFilename}.1.txt";
128
129             Assert.True(File.Exists(firstStepReferenceFilename));
130             Assert.True(File.Exists(secondStepReferenceFilename));
131
132             Assert.Equal(firstStepReferenceText, File.ReadAllText(firstStepReferenceFilename,
                ↪  Encoding.UTF8));
133             Assert.Equal(secondStepReferenceText, File.ReadAllText(secondStepReferenceFilename,
                ↪  Encoding.UTF8));
134
135             File.Delete(firstStepReferenceFilename);
136             File.Delete(secondStepReferenceFilename);
137         }
138
139         [Fact]
140         public void FilesWithNoChangesSkipedWhenUsingTransformersGenerationTest()
141         {
142             var sourceText = "aaaa";
143             var firstStepReferenceText = "bbbb";
144             var thirdStepReferenceText = "cccc";
145
146             var transformer = new TextTransformer(new SubstitutionRule[] {
147                 (new Regex("a"), "b"),
148                 (new Regex("x"), "y"),
149                 (new Regex("b"), "c")
150             });
151
152             var targetFilename = Path.GetTempFileName();
153
154             transformer.GenerateTransformersForEachRule().TransformWithAllToFiles(sourceText,
                ↪  $"{targetFilename}.txt", skipFilesWithNoChanges: true);
155
156             var firstStepReferenceFilename = $"{targetFilename}.0.txt";
157             var secondStepReferenceFilename = $"{targetFilename}.1.txt";
158             var thirdStepReferenceFilename = $"{targetFilename}.2.txt";
159
160             Assert.True(File.Exists(firstStepReferenceFilename));
161             Assert.False(File.Exists(secondStepReferenceFilename));
162             Assert.True(File.Exists(thirdStepReferenceFilename));
163
164             Assert.Equal(firstStepReferenceText, File.ReadAllText(firstStepReferenceFilename,
                ↪  Encoding.UTF8));
165             Assert.Equal(thirdStepReferenceText, File.ReadAllText(thirdStepReferenceFilename,
                ↪  Encoding.UTF8));
166
167             File.Delete(firstStepReferenceFilename);
```

```
168                 File.Delete(secondStepReferenceFilename);
169                 File.Delete(thirdStepReferenceFilename);
170             }
171         }
172     }
```

Index