1.1 ./csharp/Platform.Data.Triplets/CharacterHelpers.cs

```
using System;
using System.Collections.Generic;

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

namespace Platform.Data.Triplets
{
    // TODO: Split logic of Latin and Cyrillic alphabets into different files if possible
    /// <summary>
    /// <para>
    /// Represents the character helpers.
    /// </para>
    /// <para></para>
    /// </summary>
    public static class CharacterHelpers
    {
        /// <summary>
        /// <para>
        /// The character mapping enum.
        /// </para>
        /// <para></para>
        /// </summary>
        public enum CharacterMapping : long
        {
            /// <summary>
            /// <para>
            /// The latin alphabet character mapping.
            /// </para>
            /// <para></para>
            /// </summary>
            LatinAlphabet = 100,
            /// <summary>
            /// <para>
            /// The cyrillic alphabet character mapping.
            /// </para>
            /// <para></para>
            /// </summary>
            CyrillicAlphabet
        }

        /// <summary>
        /// <para>
        /// The first lower case latin letter.
        /// </para>
        /// <para></para>
        /// </summary>
        private const char FirstLowerCaseLatinLetter = 'a';
        /// <summary>
        /// <para>
        /// The last lower case latin letter.
        /// </para>
        /// <para></para>
        /// </summary>
        private const char LastLowerCaseLatinLetter = 'z';
        /// <summary>
        /// <para>
        /// The first upper case latin letter.
        /// </para>
        /// <para></para>
        /// </summary>
        private const char FirstUpperCaseLatinLetter = 'A';
        /// <summary>
        /// <para>
        /// The last upper case latin letter.
        /// </para>
        /// <para></para>
        /// </summary>
        private const char LastUpperCaseLatinLetter = 'Z';
        /// <summary>
        /// <para>
        /// The first lower case cyrillic letter.
        /// </para>
        /// <para></para>
        /// </summary>
        private const char FirstLowerCaseCyrillicLetter = 'a';
        /// <summary>
        /// <para>
```

```
        /// The last lower case cyrillic letter.
        /// </para>
        /// <para></para>
        /// </summary>
        private const char LastLowerCaseCyrillicLetter = 'я';
        /// <summary>
        /// <para>
        /// The first upper case cyrillic letter.
        /// </para>
        /// <para></para>
        /// </summary>
        private const char FirstUpperCaseCyrillicLetter = 'А';
        /// <summary>
        /// <para>
        /// The last upper case cyrillic letter.
        /// </para>
        /// <para></para>
        /// </summary>
        private const char LastUpperCaseCyrillicLetter = 'Я';
        /// <summary>
        /// <para>
        /// The yo lower case cyrillic letter.
        /// </para>
        /// <para></para>
        /// </summary>
        private const char YoLowerCaseCyrillicLetter = 'ё';
        /// <summary>
        /// <para>
        /// The yo upper case cyrillic letter.
        /// </para>
        /// <para></para>
        /// </summary>
        private const char YoUpperCaseCyrillicLetter = 'Ё';

        /// <summary>
        /// <para>
        /// The characters to links.
        /// </para>
        /// <para></para>
        /// </summary>
        private static Link[] _charactersToLinks;
        /// <summary>
        /// <para>
        /// The links to characters.
        /// </para>
        /// <para></para>
        /// </summary>
        private static Dictionary<Link, char> _linksToCharacters;

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="CharacterHelpers"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        static CharacterHelpers() => Create();

        /// <summary>
        /// <para>
        /// Creates.
        /// </para>
        /// <para></para>
        /// </summary>
        private static void Create()
        {
            _charactersToLinks = new Link[char.MaxValue];
            _linksToCharacters = new Dictionary<Link, char>();
            // Create or restore characters
            CreateLatinAlphabet();
            CreateCyrillicAlphabet();
            RegisterExistingCharacters();
        }

        /// <summary>
        /// <para>
        /// Registers the existing characters.
        /// </para>
        /// <para></para>
```

```
        /// </summary>
        private static void RegisterExistingCharacters() =>
        ↪  Net.Character.WalkThroughReferersAsSource(referer =>
        ↪  RegisterExistingCharacter(referer));

        /// <summary>
        /// <para>
        /// Registers the existing character using the specified character.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="character">
        /// <para>The character.</para>
        /// <para></para>
        /// </param>
        private static void RegisterExistingCharacter(Link character)
        {
            if (character.Source == Net.Character && character.Linker == Net.ThatHas)
            {
                var code = character.Target;
                if (code.Source == Net.Code && code.Linker == Net.ThatIsRepresentedBy)
                {
                    var charCode = (char)LinkConverter.ToNumber(code.Target);
                    _charactersToLinks[charCode] = character;
                    _linksToCharacters[character] = charCode;
                }
            }
        }

        /// <summary>
        /// <para>
        /// Recreates.
        /// </para>
        /// <para></para>
        /// </summary>
        public static void Recreate() => Create();

        /// <summary>
        /// <para>
        /// Creates the latin alphabet.
        /// </para>
        /// <para></para>
        /// </summary>
        private static void CreateLatinAlphabet()
        {
            var lettersCharacters = new[]
            {
                'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j',
                'k', 'l', 'm', 'n', 'o', 'p', 'q', 'r', 's', 't',
                'u', 'v', 'w', 'x', 'y', 'z'
            };
            CreateAlphabet(lettersCharacters, "latin alphabet", CharacterMapping.LatinAlphabet);
        }

        /// <summary>
        /// <para>
        /// Creates the cyrillic alphabet.
        /// </para>
        /// <para></para>
        /// </summary>
        private static void CreateCyrillicAlphabet()
        {
            var lettersCharacters = new[]
            {
                'а', 'б', 'в', 'г', 'д', 'е', 'ё', 'ж', 'з', 'и',
                'й', 'к', 'л', 'м', 'н', 'о', 'п', 'р', 'с', 'т',
                'у', 'ф', 'х', 'ц', 'ч', 'ш', 'щ', 'ъ', 'ы', 'ь',
                'э', 'ю', 'я'
            };
            CreateAlphabet(lettersCharacters, "cyrillic alphabet",
            ↪  CharacterMapping.CyrillicAlphabet);
        }

        /// <summary>
        /// <para>
        /// Creates the alphabet using the specified letters characters.
        /// </para>
        /// <para></para>
```

```
231         /// </summary>
232         /// <param name="lettersCharacters">
233         /// <para>The letters characters.</para>
234         /// <para></para>
235         /// </param>
236         /// <param name="alphabetName">
237         /// <para>The alphabet name.</para>
238         /// <para></para>
239         /// </param>
240         /// <param name="mapping">
241         /// <para>The mapping.</para>
242         /// <para></para>
243         /// </param>
244         private static void CreateAlphabet(char[] lettersCharacters, string alphabetName,
            ↪ CharacterMapping mapping)
245         {
246             if (Link.TryGetMapped(mapping, out Link alphabet))
247             {
248                 var letters = alphabet.Target;
249                 letters.WalkThroughSequence(letter =>
250                 {
251                     var lowerCaseLetter = Link.Search(Net.LowerCase, Net.Of, letter);
252                     var upperCaseLetter = Link.Search(Net.UpperCase, Net.Of, letter);
253                     if (lowerCaseLetter != null && upperCaseLetter != null)
254                     {
255                         RegisterExistingLetter(lowerCaseLetter);
256                         RegisterExistingLetter(upperCaseLetter);
257                     }
258                     else
259                     {
260                         RegisterExistingLetter(letter);
261                     }
262                 });
263             }
264             else
265             {
266                 alphabet = Net.CreateMappedThing(mapping);
267                 var letterOfAlphabet = Link.Create(Net.Letter, Net.Of, alphabet);
268                 var lettersLinks = new Link[lettersCharacters.Length];
269                 GenerateAlphabetBasis(ref alphabet, ref letterOfAlphabet, lettersLinks);
270                 for (var i = 0; i < lettersCharacters.Length; i++)
271                 {
272                     var lowerCaseCharacter = lettersCharacters[i];
273                     SetLetterCodes(lettersLinks[i], lowerCaseCharacter, out Link lowerCaseLink,
                        ↪ out Link upperCaseLink);
274                     _charactersToLinks[lowerCaseCharacter] = lowerCaseLink;
275                     _linksToCharacters[lowerCaseLink] = lowerCaseCharacter;
276                     if (upperCaseLink != null)
277                     {
278                         var upperCaseCharacter = char.ToUpper(lowerCaseCharacter);
279                         _charactersToLinks[upperCaseCharacter] = upperCaseLink;
280                         _linksToCharacters[upperCaseLink] = upperCaseCharacter;
281                     }
282                 }
283                 alphabet.SetName(alphabetName);
284                 for (var i = 0; i < lettersCharacters.Length; i++)
285                 {
286                     var lowerCaseCharacter = lettersCharacters[i];
287                     var upperCaseCharacter = char.ToUpper(lowerCaseCharacter);
288                     if (lowerCaseCharacter != upperCaseCharacter)
289                     {
290                         lettersLinks[i].SetName("{" + upperCaseCharacter + " " +
                            ↪ lowerCaseCharacter + "}");
291                     }
292                     else
293                     {
294                         lettersLinks[i].SetName("{" + lowerCaseCharacter + "}");
295                     }
296                 }
297             }
298         }
299
300         /// <summary>
301         /// <para>
302         /// Registers the existing letter using the specified letter.
303         /// </para>
304         /// <para></para>
305         /// </summary>
```

```
306         /// <param name="letter">
307         /// <para>The letter.</para>
308         /// <para></para>
309         /// </param>
310         private static void RegisterExistingLetter(Link letter)
311         {
312             letter.WalkThroughReferersAsSource(referer =>
313                 {
314                     if (referer.Linker == Net.Has)
315                     {
316                         var target = referer.Target;
317                         if (target.Source == Net.Code && target.Linker ==
                            ↪  Net.ThatIsRepresentedBy)
318                         {
319                             var charCode = (char)LinkConverter.ToNumber(target.Target);
320                             _charactersToLinks[charCode] = letter;
321                             _linksToCharacters[letter] = charCode;
322                         }
323                     }
324                 });
325         }
326 
327         /// <summary>
328         /// <para>
329         /// Generates the alphabet basis using the specified alphabet.
330         /// </para>
331         /// <para></para>
332         /// </summary>
333         /// <param name="alphabet">
334         /// <para>The alphabet.</para>
335         /// <para></para>
336         /// </param>
337         /// <param name="letterOfAlphabet">
338         /// <para>The letter of alphabet.</para>
339         /// <para></para>
340         /// </param>
341         /// <param name="letters">
342         /// <para>The letters.</para>
343         /// <para></para>
344         /// </param>
345         private static void GenerateAlphabetBasis(ref Link alphabet, ref Link letterOfAlphabet,
            ↪  Link[] letters)
346         {
347             // Принцип, на примере латинского алфавита.
348             //latin alphabet: alphabet that consists of a and b and c and ... and z.
349             //a: letter of latin alphabet that is before b.
350             //b: letter of latin alphabet that is between (a and c).
351             //c: letter of latin alphabet that is between (b and e).
352             //...
353             //y: letter of latin alphabet that is between (x and z).
354             //z: letter of latin alphabet that is after y.
355             const int firstLetterIndex = 0;
356             for (var i = firstLetterIndex; i < letters.Length; i++)
357             {
358                 letters[i] = Net.CreateThing();
359             }
360             var lastLetterIndex = letters.Length - 1;
361             Link.Update(ref letters[firstLetterIndex], letterOfAlphabet, Net.ThatIsBefore,
                ↪  letters[firstLetterIndex + 1]);
362             Link.Update(ref letters[lastLetterIndex], letterOfAlphabet, Net.ThatIsAfter,
                ↪  letters[lastLetterIndex - 1]);
363             const int secondLetterIndex = firstLetterIndex + 1;
364             for (var i = secondLetterIndex; i < lastLetterIndex; i++)
365             {
366                 Link.Update(ref letters[i], letterOfAlphabet, Net.ThatIsBetween, letters[i - 1]
                    ↪  & letters[i + 1]);
367             }
368             Link.Update(ref alphabet, Net.Alphabet, Net.ThatConsistsOf,
                ↪  LinkConverter.FromList(letters));
369         }
370 
371         /// <summary>
372         /// <para>
373         /// Sets the letter codes using the specified letter.
374         /// </para>
375         /// <para></para>
376         /// </summary>
```

```
        /// <param name="letter">
        /// <para>The letter.</para>
        /// <para></para>
        /// </param>
        /// <param name="lowerCaseCharacter">
        /// <para>The lower case character.</para>
        /// <para></para>
        /// </param>
        /// <param name="lowerCase">
        /// <para>The lower case.</para>
        /// <para></para>
        /// </param>
        /// <param name="upperCase">
        /// <para>The upper case.</para>
        /// <para></para>
        /// </param>
        private static void SetLetterCodes(Link letter, char lowerCaseCharacter, out Link
        ↪  lowerCase, out Link upperCase)
        {
            var upperCaseCharacter = char.ToUpper(lowerCaseCharacter);
            if (upperCaseCharacter != lowerCaseCharacter)
            {
                lowerCase = Link.Create(Net.LowerCase, Net.Of, letter);
                var lowerCaseCharacterCode = Link.Create(Net.Code, Net.ThatIsRepresentedBy,
                ↪  LinkConverter.FromNumber(lowerCaseCharacter));
                Link.Create(lowerCase, Net.Has, lowerCaseCharacterCode);
                upperCase = Link.Create(Net.UpperCase, Net.Of, letter);
                var upperCaseCharacterCode = Link.Create(Net.Code, Net.ThatIsRepresentedBy,
                ↪  LinkConverter.FromNumber(upperCaseCharacter));
                Link.Create(upperCase, Net.Has, upperCaseCharacterCode);
            }
            else
            {
                lowerCase = letter;
                upperCase = null;
                Link.Create(letter, Net.Has, Link.Create(Net.Code, Net.ThatIsRepresentedBy,
                ↪  LinkConverter.FromNumber(lowerCaseCharacter)));
            }
        }

        /// <summary>
        /// <para>
        /// Creates the simple character link using the specified character.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="character">
        /// <para>The character.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        private static Link CreateSimpleCharacterLink(char character) =>
        ↪  Link.Create(Net.Character, Net.ThatHas, Link.Create(Net.Code,
        ↪  Net.ThatIsRepresentedBy, LinkConverter.FromNumber(character)));

        /// <summary>
        /// <para>
        /// Determines whether is letter of latin alphabet.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="character">
        /// <para>The character.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        private static bool IsLetterOfLatinAlphabet(char character)
            => (character >= FirstLowerCaseLatinLetter && character <= LastLowerCaseLatinLetter)
            || (character >= FirstUpperCaseLatinLetter && character <= LastUpperCaseLatinLetter);

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

```
449         /// Determines whether is letter of cyrillic alphabet.
450         /// </para>
451         /// <para></para>
452         /// </summary>
453         /// <param name="character">
454         /// <para>The character.</para>
455         /// <para></para>
456         /// </param>
457         /// <returns>
458         /// <para>The bool</para>
459         /// <para></para>
460         /// </returns>
461         private static bool IsLetterOfCyrillicAlphabet(char character)
462             => (character >= FirstLowerCaseCyrillicLetter && character <=
                ↪  LastLowerCaseCyrillicLetter)
463             || (character >= FirstUpperCaseCyrillicLetter && character <=
                ↪  LastUpperCaseCyrillicLetter)
464             || character == YoLowerCaseCyrillicLetter || character == YoUpperCaseCyrillicLetter;
465 
466         /// <summary>
467         /// <para>
468         /// Creates the char using the specified character.
469         /// </para>
470         /// <para></para>
471         /// </summary>
472         /// <param name="character">
473         /// <para>The character.</para>
474         /// <para></para>
475         /// </param>
476         /// <returns>
477         /// <para>The link</para>
478         /// <para></para>
479         /// </returns>
480         public static Link FromChar(char character)
481         {
482             if (_charactersToLinks[character] == null)
483             {
484                 if (IsLetterOfLatinAlphabet(character))
485                 {
486                     CreateLatinAlphabet();
487                     return _charactersToLinks[character];
488                 }
489                 else if (IsLetterOfCyrillicAlphabet(character))
490                 {
491                     CreateCyrillicAlphabet();
492                     return _charactersToLinks[character];
493                 }
494                 else
495                 {
496                     var simpleCharacter = CreateSimpleCharacterLink(character);
497                     _charactersToLinks[character] = simpleCharacter;
498                     _linksToCharacters[simpleCharacter] = character;
499                     return simpleCharacter;
500                 }
501             }
502             else
503             {
504                 return _charactersToLinks[character];
505             }
506         }
507 
508         /// <summary>
509         /// <para>
510         /// Returns the char using the specified link.
511         /// </para>
512         /// <para></para>
513         /// </summary>
514         /// <param name="link">
515         /// <para>The link.</para>
516         /// <para></para>
517         /// </param>
518         /// <exception cref="ArgumentOutOfRangeException">
519         /// <para>Указанная связь не являяется символом.</para>
520         /// <para></para>
521         /// </exception>
522         /// <returns>
523         /// <para>The char.</para>
524         /// <para></para>
```

```
525             /// </returns>
526             public static char ToChar(Link link)
527             {
528                 if (!_linksToCharacters.TryGetValue(link, out char @char))
529                 {
530                     throw new ArgumentOutOfRangeException(nameof(link), "Указанная связь не
                        ↪  является символом.");
531                 }
532                 return @char;
533             }
534 
535             /// <summary>
536             /// <para>
537             /// Determines whether is char.
538             /// </para>
539             /// <para></para>
540             /// </summary>
541             /// <param name="link">
542             /// <para>The link.</para>
543             /// <para></para>
544             /// </param>
545             /// <returns>
546             /// <para>The bool</para>
547             /// <para></para>
548             /// </returns>
549             public static bool IsChar(Link link) => link != null &&
                ↪  _linksToCharacters.ContainsKey(link);
550         }
551     }
```

1.2 ./csharp/Platform.Data.Triplets/GexfExporter.cs

```
1   using System;
2   using System.Collections.Generic;
3   using System.IO;
4   using System.Text;
5   using System.Xml;
6   using Platform.Collections.Sets;
7   using Platform.Communication.Protocol.Gexf;
8   using GexfNode = Platform.Communication.Protocol.Gexf.Node;
9 
10  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
11 
12  namespace Platform.Data.Triplets
13  {
14      /// <summary>
15      /// <para>
16      /// Represents the gexf exporter.
17      /// </para>
18      /// <para></para>
19      /// </summary>
20      public static class GexfExporter
21      {
22          /// <summary>
23          /// <para>
24          /// The source label.
25          /// </para>
26          /// <para></para>
27          /// </summary>
28          private const string SourceLabel = "Source";
29          /// <summary>
30          /// <para>
31          /// The linker label.
32          /// </para>
33          /// <para></para>
34          /// </summary>
35          private const string LinkerLabel = "Linker";
36          /// <summary>
37          /// <para>
38          /// The target label.
39          /// </para>
40          /// <para></para>
41          /// </summary>
42          private const string TargetLabel = "Target";
43 
44          /// <summary>
45          /// <para>
46          /// Returns the xml.
47          /// </para>
48          /// <para></para>
```

```
        /// </summary>
        /// <returns>
        /// <para>The string</para>
        /// <para></para>
        /// </returns>
        public static string ToXml()
        {
            var sb = new StringBuilder();
            using (var writer = XmlWriter.Create(sb))
            {
                WriteXml(writer, CollectLinks());
            }
            return sb.ToString();
        }

        /// <summary>
        /// <para>
        /// Returns the file using the specified path.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="path">
        /// <para>The path.</para>
        /// <para></para>
        /// </param>
        public static void ToFile(string path)
        {
            using (var file = File.OpenWrite(path))
            using (var writer = XmlWriter.Create(file))
            {
                WriteXml(writer, CollectLinks());
            }
        }

        /// <summary>
        /// <para>
        /// Returns the file using the specified path.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="path">
        /// <para>The path.</para>
        /// <para></para>
        /// </param>
        /// <param name="filter">
        /// <para>The filter.</para>
        /// <para></para>
        /// </param>
        public static void ToFile(string path, Func<Link, bool> filter)
        {
            using (var file = File.OpenWrite(path))
            using (var writer = XmlWriter.Create(file))
            {
                WriteXml(writer, CollectLinks(filter));
            }
        }

        /// <summary>
        /// <para>
        /// Collects the links using the specified link match.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="linkMatch">
        /// <para>The link match.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The matching links.</para>
        /// <para></para>
        /// </returns>
        private static HashSet<Link> CollectLinks(Func<Link, bool> linkMatch)
        {
            var matchingLinks = new HashSet<Link>();
            Link.WalkThroughAllLinks(link =>
            {
                if (linkMatch(link))
                {
```

```
                    matchingLinks.Add(link);
                }
            });
            return matchingLinks;
        }

        /// <summary>
        /// <para>
        /// Collects the links.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <returns>
        /// <para>The matching links.</para>
        /// <para></para>
        /// </returns>
        private static HashSet<Link> CollectLinks()
        {
            var matchingLinks = new HashSet<Link>();
            Link.WalkThroughAllLinks(matchingLinks.AddAndReturnVoid);
            return matchingLinks;
        }

        /// <summary>
        /// <para>
        /// Writes the xml using the specified writer.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="writer">
        /// <para>The writer.</para>
        /// <para></para>
        /// </param>
        /// <param name="matchingLinks">
        /// <para>The matching links.</para>
        /// <para></para>
        /// </param>
        private static void WriteXml(XmlWriter writer, HashSet<Link> matchingLinks)
        {
            var edgesCounter = 0;
            Gexf.WriteXml(writer,
            () => // nodes
            {
                foreach (var matchingLink in matchingLinks)
                {
                    GexfNode.WriteXml(writer, matchingLink.ToInt(), matchingLink.ToString());
                }
            },
            () => // edges
            {
                foreach (var matchingLink in matchingLinks)
                {
                    if (matchingLinks.Contains(matchingLink.Source))
                    {
                        Edge.WriteXml(writer, edgesCounter++, matchingLink.ToInt(),
                        ↪  matchingLink.Source.ToInt(), SourceLabel);
                    }
                    if (matchingLinks.Contains(matchingLink.Linker))
                    {
                        Edge.WriteXml(writer, edgesCounter++, matchingLink.ToInt(),
                        ↪  matchingLink.Linker.ToInt(), LinkerLabel);
                    }
                    if (matchingLinks.Contains(matchingLink.Target))
                    {
                        Edge.WriteXml(writer, edgesCounter++, matchingLink.ToInt(),
                        ↪  matchingLink.Target.ToInt(), TargetLabel);
                    }
                }
            });
        }
    }
}
```

1.3 ./csharp/Platform.Data.Triplets/ILink.cs

```
using System;

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

```

```
5  namespace Platform.Data.Triplets
6  {
7      /// <summary>
8      /// <para>
9      /// Defines the link.
10     /// </para>
11     /// <para></para>
12     /// </summary>
13     internal partial interface ILink<TLink>
14         where TLink : ILink<TLink>
15     {
16         /// <summary>
17         /// <para>
18         /// Gets the source value.
19         /// </para>
20         /// <para></para>
21         /// </summary>
22         TLink Source { get; }
23         /// <summary>
24         /// <para>
25         /// Gets the linker value.
26         /// </para>
27         /// <para></para>
28         /// </summary>
29         TLink Linker { get; }
30         /// <summary>
31         /// <para>
32         /// Gets the target value.
33         /// </para>
34         /// <para></para>
35         /// </summary>
36         TLink Target { get; }
37     }
38
39     /// <summary>
40     /// <para>
41     /// Defines the link.
42     /// </para>
43     /// <para></para>
44     /// </summary>
45     internal partial interface ILink<TLink>
46         where TLink : ILink<TLink>
47     {
48         /// <summary>
49         /// <para>
50         /// Determines whether this instance walk through referers as linker.
51         /// </para>
52         /// <para></para>
53         /// </summary>
54         /// <param name="walker">
55         /// <para>The walker.</para>
56         /// <para></para>
57         /// </param>
58         /// <returns>
59         /// <para>The bool</para>
60         /// <para></para>
61         /// </returns>
62         bool WalkThroughReferersAsLinker(Func<TLink, bool> walker);
63         /// <summary>
64         /// <para>
65         /// Determines whether this instance walk through referers as source.
66         /// </para>
67         /// <para></para>
68         /// </summary>
69         /// <param name="walker">
70         /// <para>The walker.</para>
71         /// <para></para>
72         /// </param>
73         /// <returns>
74         /// <para>The bool</para>
75         /// <para></para>
76         /// </returns>
77         bool WalkThroughReferersAsSource(Func<TLink, bool> walker);
78         /// <summary>
79         /// <para>
80         /// Determines whether this instance walk through referers as target.
81         /// </para>
82         /// <para></para>
```

```
        /// </summary>
        /// <param name="walker">
        /// <para>The walker.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        bool WalkThroughReferersAsTarget(Func<TLink, bool> walker);
        /// <summary>
        /// <para>
        /// Walks the through referers using the specified walker.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="walker">
        /// <para>The walker.</para>
        /// <para></para>
        /// </param>
        void WalkThroughReferers(Func<TLink, bool> walker);
    }

    /// <summary>
    /// <para>
    /// Defines the link.
    /// </para>
    /// <para></para>
    /// </summary>
    internal partial interface ILink<TLink>
        where TLink : ILink<TLink>
    {
        /// <summary>
        /// <para>
        /// Walks the through referers as linker using the specified walker.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="walker">
        /// <para>The walker.</para>
        /// <para></para>
        /// </param>
        void WalkThroughReferersAsLinker(Action<TLink> walker);
        /// <summary>
        /// <para>
        /// Walks the through referers as source using the specified walker.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="walker">
        /// <para>The walker.</para>
        /// <para></para>
        /// </param>
        void WalkThroughReferersAsSource(Action<TLink> walker);
        /// <summary>
        /// <para>
        /// Walks the through referers as target using the specified walker.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="walker">
        /// <para>The walker.</para>
        /// <para></para>
        /// </param>
        void WalkThroughReferersAsTarget(Action<TLink> walker);
        /// <summary>
        /// <para>
        /// Walks the through referers using the specified walker.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="walker">
        /// <para>The walker.</para>
        /// <para></para>
        /// </param>
        void WalkThroughReferers(Action<TLink> walker);
    }
}
```

```
/*
using System;
namespace NetLibrary
{
    interface ILink
    {
        // Статические методы (общие для всех связей)
        public static ILink Create(ILink source, ILink linker, ILink target);
        public static void Update(ref ILink link, ILink newSource, ILink newLinker, ILink
↪  newTarget);
        public static void Delete(ref ILink link);
        public static ILink Search(ILink source, ILink linker, ILink target);
    }
}
*/
/*
Набор функций, который необходим для работы с сущностью Link:

(Работа со значением сущности Link, значение состоит из 3-х частей, также сущностей Link)
1. Получить адрес "начальной" сущности Link. (Получить адрес из поля Source)
2. Получить адрес сущности Link, которая играет роль связки между "начальной" и "конечной"
↪  сущностями Link. (Получить адрес из поля Linker)
3. Получить адрес "конечной" сущности Link. (Получить адрес из поля Target)

4. Пройтись по всем сущностями Link, которые ссылаются на сущность Link с указанным адресом, и у
↪  которых поле Source равно этому адресу.
5. Пройтись по всем сущностями Link, которые ссылаются на сущность Link с указанным адресом, и у
↪  которых поле Linker равно этому адресу.
6. Пройтись по всем сущностями Link, которые ссылаются на сущность Link с указанным адресом, и у
↪  которых поле Target равно этому адресу.

7. Создать сущность Link со значением (смыслом), которым являются адреса на другие 3 сущности
↪  Link (где первая является "начальной", вторая является "связкой", а третья является
↪  "конечной").
8. Обновление сущности Link с указанным адресом новым значением (смыслом), которым являются
↪  адреса на другие 3 сущности Link (где первая является "начальной", вторая является
↪  "связкой", а третья является "конечной").
9. Удаление сущности Link с указаным адресом.
10. Поиск сущности Link со значением (смыслом), которым являются адреса на другие 3 сущности
↪  Link (где первая является "начальной", вторая является "связкой", а третья является
↪  "конечной").
*/
```

1.4 ./csharp/Platform.Data.Triplets/Link.Debug.cs

```
using System;
using System.Diagnostics;

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

namespace Platform.Data.Triplets
{
    /// <summary>
    /// <para>
    /// The link.
    /// </para>
    /// <para></para>
    /// </summary>
    public partial struct Link
    {
        #region Properties

        // ReSharper disable InconsistentNaming
        // ReSharper disable UnusedMember.Local
#pragma warning disable IDE0051 // Remove unused private members

        /// <summary>
        /// <para>
        /// Gets the я a value.
        /// </para>
        /// <para></para>
        /// </summary>
        [DebuggerDisplay(null, Name = "Source")]
        private Link Я_A => this == null ? Itself : Source;

        /// <summary>
        /// <para>
        /// Gets the я b value.
        /// </para>
        /// <para></para>
```

```
        /// </summary>
        [DebuggerDisplay(null, Name = "Linker")]
        private Link Я_B => this == null ? Itself : Linker;

        /// <summary>
        /// <para>
        /// Gets the я c value.
        /// </para>
        /// <para></para>
        /// </summary>
        [DebuggerDisplay(null, Name = "Target")]
        private Link Я_C => this == null ? Itself : Target;

        /// <summary>
        /// <para>
        /// Gets the я d value.
        /// </para>
        /// <para></para>
        /// </summary>
        [DebuggerDisplay("Count = {Я_DC}", Name = "ReferersBySource")]
        private Link[] Я_D => this.GetArrayOfRererersBySource();

        /// <summary>
        /// <para>
        /// Gets the я e value.
        /// </para>
        /// <para></para>
        /// </summary>
        [DebuggerDisplay("Count = {Я_EC}", Name = "ReferersByLinker")]
        private Link[] Я_E => this.GetArrayOfRererersByLinker();

        /// <summary>
        /// <para>
        /// Gets the я f value.
        /// </para>
        /// <para></para>
        /// </summary>
        [DebuggerDisplay("Count = {Я_FC}", Name = "ReferersByTarget")]
        private Link[] Я_F => this.GetArrayOfRererersByTarget();

        /// <summary>
        /// <para>
        /// Gets the я dc value.
        /// </para>
        /// <para></para>
        /// </summary>
        [DebuggerBrowsable(DebuggerBrowsableState.Never)]
        private Int64 Я_DC => this == null ? 0 : ReferersBySourceCount;

        /// <summary>
        /// <para>
        /// Gets the я ec value.
        /// </para>
        /// <para></para>
        /// </summary>
        [DebuggerBrowsable(DebuggerBrowsableState.Never)]
        private Int64 Я_EC => this == null ? 0 : ReferersByLinkerCount;

        /// <summary>
        /// <para>
        /// Gets the я fc value.
        /// </para>
        /// <para></para>
        /// </summary>
        [DebuggerBrowsable(DebuggerBrowsableState.Never)]
        private Int64 Я_FC => this == null ? 0 : ReferersByTargetCount;

        /// <summary>
        /// <para>
        /// Gets the я h value.
        /// </para>
        /// <para></para>
        /// </summary>
        [DebuggerDisplay(null, Name = "Timestamp")]
        private DateTime Я_H => this == null ? DateTime.MinValue : Timestamp;

        // ReSharper restore UnusedMember.Local
        // ReSharper restore InconsistentNaming
#pragma warning restore IDE0051 // Remove unused private members
```

```

        #endregion

        /// <summary>
        /// <para>
        /// Returns the string.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <returns>
        /// <para>The string</para>
        /// <para></para>
        /// </returns>
        public override string ToString()
        {
            const string nullString = "null";
            if (this == null)
            {
                return nullString;
            }
            else
            {
                if (this.TryGetName(out string name))
                {
                    return name;
                }
                else
                {
                    return ((long)_link).ToString();
                }
            }
        }
    }
}
```

1.5 ./csharp/Platform.Data.Triplets/Link.cs

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Runtime.InteropServices;
using System.Threading;
using Int = System.Int64;
using LinkIndex = System.UInt64;

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

namespace Platform.Data.Triplets
{
    /// <summary>
    /// <para>
    /// The link definition.
    /// </para>
    /// <para></para>
    /// </summary>
    public struct LinkDefinition : IEquatable<LinkDefinition>
    {
        /// <summary>
        /// <para>
        /// The source.
        /// </para>
        /// <para></para>
        /// </summary>
        public readonly Link Source;
        /// <summary>
        /// <para>
        /// The linker.
        /// </para>
        /// <para></para>
        /// </summary>
        public readonly Link Linker;
        /// <summary>
        /// <para>
        /// The target.
        /// </para>
        /// <para></para>
        /// </summary>
        public readonly Link Target;

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

```
45          /// Initializes a new <see cref="LinkDefinition"/> instance.
46          /// </para>
47          /// <para></para>
48          /// </summary>
49          /// <param name="source">
50          /// <para>A source.</para>
51          /// <para></para>
52          /// </param>
53          /// <param name="linker">
54          /// <para>A linker.</para>
55          /// <para></para>
56          /// </param>
57          /// <param name="target">
58          /// <para>A target.</para>
59          /// <para></para>
60          /// </param>
61          public LinkDefinition(Link source, Link linker, Link target)
62          {
63              Source = source;
64              Linker = linker;
65              Target = target;
66          }
67
68          /// <summary>
69          /// <para>
70          /// Initializes a new <see cref="LinkDefinition"/> instance.
71          /// </para>
72          /// <para></para>
73          /// </summary>
74          /// <param name="link">
75          /// <para>A link.</para>
76          /// <para></para>
77          /// </param>
78          public LinkDefinition(Link link) : this(link.Source, link.Linker, link.Target) { }
79
80          /// <summary>
81          /// <para>
82          /// Determines whether this instance equals.
83          /// </para>
84          /// <para></para>
85          /// </summary>
86          /// <param name="other">
87          /// <para>The other.</para>
88          /// <para></para>
89          /// </param>
90          /// <returns>
91          /// <para>The bool</para>
92          /// <para></para>
93          /// </returns>
94          public bool Equals(LinkDefinition other) => Source == other.Source && Linker ==
            ↪  other.Linker && Target == other.Target;
95      }
96
97      /// <summary>
98      /// <para>
99      /// The link.
100     /// </para>
101     /// <para></para>
102     /// </summary>
103     public partial struct Link : ILink<Link>, IEquatable<Link>
104     {
105         /// <summary>
106         /// <para>
107         /// The dll name.
108         /// </para>
109         /// <para></para>
110         /// </summary>
111         private const string DllName = "Platform_Data_Triplets_Kernel";
112
113         // TODO: Заменить на очередь событий, по примеру Node.js (+сделать выключаемым)
114         /// <summary>
115         /// <para>
116         /// The created delegate.
117         /// </para>
118         /// <para></para>
119         /// </summary>
120         public delegate void CreatedDelegate(LinkDefinition createdLink);
121         public static event CreatedDelegate CreatedEvent = createdLink => { };
```

```
122
123         /// <summary>
124         /// <para>
125         /// The updated delegate.
126         /// </para>
127         /// <para></para>
128         /// </summary>
129         public delegate void UpdatedDelegate(LinkDefinition linkBeforeUpdate, LinkDefinition
            ↪  linkAfterUpdate);
130         public static event UpdatedDelegate UpdatedEvent = (linkBeforeUpdate, linkAfterUpdate)
            ↪  => { };
131
132         /// <summary>
133         /// <para>
134         /// The deleted delegate.
135         /// </para>
136         /// <para></para>
137         /// </summary>
138         public delegate void DeletedDelegate(LinkDefinition deletedLink);
139         public static event DeletedDelegate DeletedEvent = deletedLink => { };
140
141         #region Low Level
142
143         #region Basic Operations
144
145         /// <summary>
146         /// <para>
147         /// Gets the source index using the specified link.
148         /// </para>
149         /// <para></para>
150         /// </summary>
151         /// <param name="link">
152         /// <para>The link.</para>
153         /// <para></para>
154         /// </param>
155         /// <returns>
156         /// <para>The link index</para>
157         /// <para></para>
158         /// </returns>
159         [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
160         private static extern LinkIndex GetSourceIndex(LinkIndex link);
161
162         /// <summary>
163         /// <para>
164         /// Gets the linker index using the specified link.
165         /// </para>
166         /// <para></para>
167         /// </summary>
168         /// <param name="link">
169         /// <para>The link.</para>
170         /// <para></para>
171         /// </param>
172         /// <returns>
173         /// <para>The link index</para>
174         /// <para></para>
175         /// </returns>
176         [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
177         private static extern LinkIndex GetLinkerIndex(LinkIndex link);
178
179         /// <summary>
180         /// <para>
181         /// Gets the target index using the specified link.
182         /// </para>
183         /// <para></para>
184         /// </summary>
185         /// <param name="link">
186         /// <para>The link.</para>
187         /// <para></para>
188         /// </param>
189         /// <returns>
190         /// <para>The link index</para>
191         /// <para></para>
192         /// </returns>
193         [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
194         private static extern LinkIndex GetTargetIndex(LinkIndex link);
195
196         /// <summary>
197         /// <para>
```

```
        /// Gets the first referer by source index using the specified link.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="link">
        /// <para>The link.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link index</para>
        /// <para></para>
        /// </returns>
        [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
        private static extern LinkIndex GetFirstRefererBySourceIndex(LinkIndex link);

        /// <summary>
        /// <para>
        /// Gets the first referer by linker index using the specified link.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="link">
        /// <para>The link.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link index</para>
        /// <para></para>
        /// </returns>
        [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
        private static extern LinkIndex GetFirstRefererByLinkerIndex(LinkIndex link);

        /// <summary>
        /// <para>
        /// Gets the first referer by target index using the specified link.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="link">
        /// <para>The link.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link index</para>
        /// <para></para>
        /// </returns>
        [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
        private static extern LinkIndex GetFirstRefererByTargetIndex(LinkIndex link);

        /// <summary>
        /// <para>
        /// Gets the time using the specified link.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="link">
        /// <para>The link.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The int</para>
        /// <para></para>
        /// </returns>
        [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
        private static extern Int GetTime(LinkIndex link);

        /// <summary>
        /// <para>
        /// Creates the link using the specified source.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="source">
        /// <para>The source.</para>
        /// <para></para>
        /// </param>
        /// <param name="linker">
        /// <para>The linker.</para>
```

```
        /// <para></para>
        /// </param>
        /// <param name="target">
        /// <para>The target.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link index</para>
        /// <para></para>
        /// </returns>
        [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
        private static extern LinkIndex CreateLink(LinkIndex source, LinkIndex linker, LinkIndex
        ↪  target);

        /// <summary>
        /// <para>
        /// Updates the link using the specified link.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="link">
        /// <para>The link.</para>
        /// <para></para>
        /// </param>
        /// <param name="newSource">
        /// <para>The new source.</para>
        /// <para></para>
        /// </param>
        /// <param name="newLinker">
        /// <para>The new linker.</para>
        /// <para></para>
        /// </param>
        /// <param name="newTarget">
        /// <para>The new target.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link index</para>
        /// <para></para>
        /// </returns>
        [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
        private static extern LinkIndex UpdateLink(LinkIndex link, LinkIndex newSource,
        ↪  LinkIndex newLinker, LinkIndex newTarget);

        /// <summary>
        /// <para>
        /// Deletes the link using the specified link.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="link">
        /// <para>The link.</para>
        /// <para></para>
        /// </param>
        [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
        private static extern void DeleteLink(LinkIndex link);

        /// <summary>
        /// <para>
        /// Replaces the link using the specified link.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="link">
        /// <para>The link.</para>
        /// <para></para>
        /// </param>
        /// <param name="replacement">
        /// <para>The replacement.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link index</para>
        /// <para></para>
        /// </returns>
        [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
        private static extern LinkIndex ReplaceLink(LinkIndex link, LinkIndex replacement);

```

```
        /// <summary>
        /// <para>
        /// Searches the link using the specified source.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="source">
        /// <para>The source.</para>
        /// <para></para>
        /// </param>
        /// <param name="linker">
        /// <para>The linker.</para>
        /// <para></para>
        /// </param>
        /// <param name="target">
        /// <para>The target.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link index</para>
        /// <para></para>
        /// </returns>
        [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
        private static extern LinkIndex SearchLink(LinkIndex source, LinkIndex linker, LinkIndex
        ↪  target);

        /// <summary>
        /// <para>
        /// Gets the mapped link using the specified mapped index.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="mappedIndex">
        /// <para>The mapped index.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link index</para>
        /// <para></para>
        /// </returns>
        [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
        private static extern LinkIndex GetMappedLink(Int mappedIndex);

        /// <summary>
        /// <para>
        /// Sets the mapped link using the specified mapped index.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="mappedIndex">
        /// <para>The mapped index.</para>
        /// <para></para>
        /// </param>
        /// <param name="linkIndex">
        /// <para>The link index.</para>
        /// <para></para>
        /// </param>
        [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
        private static extern void SetMappedLink(Int mappedIndex, LinkIndex linkIndex);

        /// <summary>
        /// <para>
        /// Opens the links using the specified filename.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="filename">
        /// <para>The filename.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The int</para>
        /// <para></para>
        /// </returns>
        [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
        private static extern Int OpenLinks(string filename);

        /// <summary>
```

```
        /// <para>
        /// Closes the links.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <returns>
        /// <para>The int</para>
        /// <para></para>
        /// </returns>
        [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
        private static extern Int CloseLinks();

        #endregion

        #region Referers Count Selectors

        /// <summary>
        /// <para>
        /// Gets the link number of referers by source using the specified link.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="link">
        /// <para>The link.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link index</para>
        /// <para></para>
        /// </returns>
        [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
        private static extern LinkIndex GetLinkNumberOfReferersBySource(LinkIndex link);

        /// <summary>
        /// <para>
        /// Gets the link number of referers by linker using the specified link.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="link">
        /// <para>The link.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link index</para>
        /// <para></para>
        /// </returns>
        [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
        private static extern LinkIndex GetLinkNumberOfReferersByLinker(LinkIndex link);

        /// <summary>
        /// <para>
        /// Gets the link number of referers by target using the specified link.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="link">
        /// <para>The link.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link index</para>
        /// <para></para>
        /// </returns>
        [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
        private static extern LinkIndex GetLinkNumberOfReferersByTarget(LinkIndex link);

        #endregion

        #region Referers Walkers

        /// <summary>
        /// <para>
        /// The visitor.
        /// </para>
        /// <para></para>
        /// </summary>
        private delegate void Visitor(LinkIndex link);
        /// <summary>
```

```
508         /// <para>
509         /// The stopable visitor.
510         /// </para>
511         /// <para></para>
512         /// </summary>
513         private delegate Int StopableVisitor(LinkIndex link);
514 
515         /// <summary>
516         /// <para>
517         /// Walks the through all referers by source using the specified root.
518         /// </para>
519         /// <para></para>
520         /// </summary>
521         /// <param name="root">
522         /// <para>The root.</para>
523         /// <para></para>
524         /// </param>
525         /// <param name="action">
526         /// <para>The action.</para>
527         /// <para></para>
528         /// </param>
529         [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
530         private static extern void WalkThroughAllReferersBySource(LinkIndex root, Visitor
            ↪  action);
531 
532         /// <summary>
533         /// <para>
534         /// Walks the through referers by source using the specified root.
535         /// </para>
536         /// <para></para>
537         /// </summary>
538         /// <param name="root">
539         /// <para>The root.</para>
540         /// <para></para>
541         /// </param>
542         /// <param name="func">
543         /// <para>The func.</para>
544         /// <para></para>
545         /// </param>
546         /// <returns>
547         /// <para>The int</para>
548         /// <para></para>
549         /// </returns>
550         [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
551         private static extern int WalkThroughReferersBySource(LinkIndex root, StopableVisitor
            ↪  func);
552 
553         /// <summary>
554         /// <para>
555         /// Walks the through all referers by linker using the specified root.
556         /// </para>
557         /// <para></para>
558         /// </summary>
559         /// <param name="root">
560         /// <para>The root.</para>
561         /// <para></para>
562         /// </param>
563         /// <param name="action">
564         /// <para>The action.</para>
565         /// <para></para>
566         /// </param>
567         [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
568         private static extern void WalkThroughAllReferersByLinker(LinkIndex root, Visitor
            ↪  action);
569 
570         /// <summary>
571         /// <para>
572         /// Walks the through referers by linker using the specified root.
573         /// </para>
574         /// <para></para>
575         /// </summary>
576         /// <param name="root">
577         /// <para>The root.</para>
578         /// <para></para>
579         /// </param>
580         /// <param name="func">
581         /// <para>The func.</para>
582         /// <para></para>
```

```
        /// </param>
        /// <returns>
        /// <para>The int</para>
        /// <para></para>
        /// </returns>
        [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
        private static extern int WalkThroughReferersByLinker(LinkIndex root, StopableVisitor
        ↪  func);

        /// <summary>
        /// <para>
        /// Walks the through all referers by target using the specified root.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="root">
        /// <para>The root.</para>
        /// <para></para>
        /// </param>
        /// <param name="action">
        /// <para>The action.</para>
        /// <para></para>
        /// </param>
        [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
        private static extern void WalkThroughAllReferersByTarget(LinkIndex root, Visitor
        ↪  action);

        /// <summary>
        /// <para>
        /// Walks the through referers by target using the specified root.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="root">
        /// <para>The root.</para>
        /// <para></para>
        /// </param>
        /// <param name="func">
        /// <para>The func.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The int</para>
        /// <para></para>
        /// </returns>
        [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
        private static extern int WalkThroughReferersByTarget(LinkIndex root, StopableVisitor
        ↪  func);

        /// <summary>
        /// <para>
        /// Walks the through all links using the specified action.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="action">
        /// <para>The action.</para>
        /// <para></para>
        /// </param>
        [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
        private static extern void WalkThroughAllLinks(Visitor action);

        /// <summary>
        /// <para>
        /// Walks the through links using the specified func.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="func">
        /// <para>The func.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The int</para>
        /// <para></para>
        /// </returns>
        [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
        private static extern int WalkThroughLinks(StopableVisitor func);
```

```
658
659         #endregion
660
661         #endregion
662
663         #region Constains
664
665         /// <summary>
666         /// <para>
667         /// The itself.
668         /// </para>
669         /// <para></para>
670         /// </summary>
671         public static readonly Link Itself = null;
672         /// <summary>
673         /// <para>
674         /// The continue.
675         /// </para>
676         /// <para></para>
677         /// </summary>
678         public static readonly bool Continue = true;
679         /// <summary>
680         /// <para>
681         /// The stop.
682         /// </para>
683         /// <para></para>
684         /// </summary>
685         public static readonly bool Stop;
686
687         #endregion
688
689         #region Static Fields
690
691         /// <summary>
692         /// <para>
693         /// The lock object.
694         /// </para>
695         /// <para></para>
696         /// </summary>
697         private static readonly object _lockObject = new object();
698         /// <summary>
699         /// <para>
700         /// The memory manager is ready.
701         /// </para>
702         /// <para></para>
703         /// </summary>
704         private static volatile int _memoryManagerIsReady;
705         /// <summary>
706         /// <para>
707         /// The dictionary.
708         /// </para>
709         /// <para></para>
710         /// </summary>
711         private static readonly Dictionary<ulong, long> _linkToMappingIndex = new
            ↪  Dictionary<ulong, long>();
712
713         #endregion
714
715         #region Fields
716
717         /// <summary>
718         /// <para>
719         /// The link.
720         /// </para>
721         /// <para></para>
722         /// </summary>
723         [DebuggerBrowsable(DebuggerBrowsableState.Never)]
724         private readonly LinkIndex _link;
725
726         #endregion
727
728         #region Properties
729
730         /// <summary>
731         /// <para>
732         /// Gets the source value.
733         /// </para>
734         /// <para></para>
735         /// </summary>
736         [DebuggerBrowsable(DebuggerBrowsableState.Never)]
```

```
737         public Link Source => GetSourceIndex(_link);
738 
739         /// <summary>
740         /// <para>
741         /// Gets the linker value.
742         /// </para>
743         /// <para></para>
744         /// </summary>
745         [DebuggerBrowsable(DebuggerBrowsableState.Never)]
746         public Link Linker => GetLinkerIndex(_link);
747 
748         /// <summary>
749         /// <para>
750         /// Gets the target value.
751         /// </para>
752         /// <para></para>
753         /// </summary>
754         [DebuggerBrowsable(DebuggerBrowsableState.Never)]
755         public Link Target => GetTargetIndex(_link);
756 
757         /// <summary>
758         /// <para>
759         /// Gets the first referer by source value.
760         /// </para>
761         /// <para></para>
762         /// </summary>
763         [DebuggerBrowsable(DebuggerBrowsableState.Never)]
764         public Link FirstRefererBySource => GetFirstRefererBySourceIndex(_link);
765 
766         /// <summary>
767         /// <para>
768         /// Gets the first referer by linker value.
769         /// </para>
770         /// <para></para>
771         /// </summary>
772         [DebuggerBrowsable(DebuggerBrowsableState.Never)]
773         public Link FirstRefererByLinker => GetFirstRefererByLinkerIndex(_link);
774 
775         /// <summary>
776         /// <para>
777         /// Gets the first referer by target value.
778         /// </para>
779         /// <para></para>
780         /// </summary>
781         [DebuggerBrowsable(DebuggerBrowsableState.Never)]
782         public Link FirstRefererByTarget => GetFirstRefererByTargetIndex(_link);
783 
784         /// <summary>
785         /// <para>
786         /// Gets the referers by source count value.
787         /// </para>
788         /// <para></para>
789         /// </summary>
790         [DebuggerBrowsable(DebuggerBrowsableState.Never)]
791         public Int ReferersBySourceCount => (Int)GetLinkNumberOfReferersBySource(_link);
792 
793         /// <summary>
794         /// <para>
795         /// Gets the referers by linker count value.
796         /// </para>
797         /// <para></para>
798         /// </summary>
799         [DebuggerBrowsable(DebuggerBrowsableState.Never)]
800         public Int ReferersByLinkerCount => (Int)GetLinkNumberOfReferersByLinker(_link);
801 
802         /// <summary>
803         /// <para>
804         /// Gets the referers by target count value.
805         /// </para>
806         /// <para></para>
807         /// </summary>
808         [DebuggerBrowsable(DebuggerBrowsableState.Never)]
809         public Int ReferersByTargetCount => (Int)GetLinkNumberOfReferersByTarget(_link);
810 
811         /// <summary>
812         /// <para>
813         /// Gets the total referers value.
814         /// </para>
```

```
        /// <para></para>
        /// </summary>
        [DebuggerBrowsable(DebuggerBrowsableState.Never)]
        public Int TotalReferers => (Int)GetLinkNumberOfReferersBySource(_link) +
        ↪  (Int)GetLinkNumberOfReferersByLinker(_link) +
        ↪  (Int)GetLinkNumberOfReferersByTarget(_link);

        /// <summary>
        /// <para>
        /// Gets the timestamp value.
        /// </para>
        /// <para></para>
        /// </summary>
        [DebuggerBrowsable(DebuggerBrowsableState.Never)]
        public DateTime Timestamp => DateTime.FromFileTimeUtc(GetTime(_link));

        #endregion

        #region Infrastructure

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="Link"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="link">
        /// <para>A link.</para>
        /// <para></para>
        /// </param>
        public Link(LinkIndex link) => _link = link;

        /// <summary>
        /// <para>
        /// Starts the memory manager using the specified storage filename.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="storageFilename">
        /// <para>The storage filename.</para>
        /// <para></para>
        /// </param>
        /// <exception cref="InvalidOperationException">
        /// <para>Файл ({storageFilename}) хранилища не удалось открыть.</para>
        /// <para></para>
        /// </exception>
        public static void StartMemoryManager(string storageFilename)
        {
            lock (_lockObject)
            {
                if (_memoryManagerIsReady == default)
                {
                    if (OpenLinks(storageFilename) == 0)
                    {
                        throw new InvalidOperationException($"Файл ({storageFilename})
                        ↪  хранилища не удалось открыть.");
                    }
                    Interlocked.Exchange(ref _memoryManagerIsReady, 1);
                }
            }
        }

        /// <summary>
        /// <para>
        /// Stops the memory manager.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <exception cref="InvalidOperationException">
        /// <para>Файл хранилища не удалось закрыть. Возможно он был уже закрыт, или не
        ↪  открывался вовсе.</para>
        /// <para></para>
        /// </exception>
        public static void StopMemoryManager()
        {
            lock (_lockObject)
            {
                if (_memoryManagerIsReady != default)
```

```
                {
                    if (CloseLinks() == 0)
                    {
                        throw new InvalidOperationException("Файл хранилища не удалось закрыть. Возможно он был уже закрыт, или не открывался вовсе.");
                    }
                    Interlocked.Exchange(ref _memoryManagerIsReady, 0);
                }
            }
        }

        public static implicit operator LinkIndex?(Link link) => link._link == 0 ? (LinkIndex?)null : link._link;

        public static implicit operator Link(LinkIndex? link) => new Link(link ?? 0);

        public static implicit operator Int(Link link) => (Int)link._link;

        public static implicit operator Link(Int link) => new Link((LinkIndex)link);

        public static implicit operator LinkIndex(Link link) => link._link;

        public static implicit operator Link(LinkIndex link) => new Link(link);

        public static explicit operator Link(List<Link> links) => LinkConverter.FromList(links);

        public static explicit operator Link(Link[] links) => LinkConverter.FromList(links);

        public static explicit operator Link(string @string) => LinkConverter.FromString(@string);

        public static bool operator ==(Link first, Link second) => first.Equals(second);

        public static bool operator !=(Link first, Link second) => !first.Equals(second);

        public static Link operator &(Link first, Link second) => Create(first, Net.And, second);

        /// <summary>
        /// <para>
        /// Determines whether this instance equals.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="obj">
        /// <para>The obj.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        public override bool Equals(object obj) => obj is Link link ? Equals(link) : false;

        /// <summary>
        /// <para>
        /// Determines whether this instance equals.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="other">
        /// <para>The other.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        public bool Equals(Link other) => _link == other._link || (LinkDoesNotExist(_link) && LinkDoesNotExist(other._link));

        /// <summary>
        /// <para>
        /// Gets the hash code.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <returns>
        /// <para>The int</para>
```

```
        /// <para></para>
        /// </returns>
        public override int GetHashCode() => base.GetHashCode();

        /// <summary>
        /// <para>
        /// Determines whether link does not exist.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="link">
        /// <para>The link.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        private static bool LinkDoesNotExist(LinkIndex link) => link == 0 ||
        ↪  GetLinkerIndex(link) == 0;

        /// <summary>
        /// <para>
        /// Determines whether link was deleted.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="link">
        /// <para>The link.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        private static bool LinkWasDeleted(LinkIndex link) => link != 0 && GetLinkerIndex(link)
        ↪  == 0;

        /// <summary>
        /// <para>
        /// Determines whether this instance is matching to.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="source">
        /// <para>The source.</para>
        /// <para></para>
        /// </param>
        /// <param name="linker">
        /// <para>The linker.</para>
        /// <para></para>
        /// </param>
        /// <param name="target">
        /// <para>The target.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        private bool IsMatchingTo(Link source, Link linker, Link target)
            => ((Source == this && source == null) || (Source == source))
            && ((Linker == this && linker == null) || (Linker == linker))
            && ((Target == this && target == null) || (Target == target));

        /// <summary>
        /// <para>
        /// Returns the index.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <returns>
        /// <para>The link index</para>
        /// <para></para>
        /// </returns>
        public LinkIndex ToIndex() => _link;

        /// <summary>
```

```
        /// <para>
        /// Returns the int.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <returns>
        /// <para>The int</para>
        /// <para></para>
        /// </returns>
        public Int ToInt() => (Int)_link;

        #endregion

        #region Basic Operations

        /// <summary>
        /// <para>
        /// Creates the source.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="source">
        /// <para>The source.</para>
        /// <para></para>
        /// </param>
        /// <param name="linker">
        /// <para>The linker.</para>
        /// <para></para>
        /// </param>
        /// <param name="target">
        /// <para>The target.</para>
        /// <para></para>
        /// </param>
        /// <exception cref="InvalidOperationException">
        /// <para>Менеджер памяти ещё не готов.</para>
        /// <para></para>
        /// </exception>
        /// <exception cref="InvalidOperationException">
        /// <para>Невозможно создать связь.</para>
        /// <para></para>
        /// </exception>
        /// <exception cref="ArgumentException">
        /// <para>Удалённая связь не может использоваться в качестве значения. </para>
        /// <para></para>
        /// </exception>
        /// <exception cref="ArgumentException">
        /// <para>Удалённая связь не может использоваться в качестве значения. </para>
        /// <para></para>
        /// </exception>
        /// <exception cref="ArgumentException">
        /// <para>Удалённая связь не может использоваться в качестве значения. </para>
        /// <para></para>
        /// </exception>
        /// <returns>
        /// <para>The link.</para>
        /// <para></para>
        /// </returns>
        public static Link Create(Link source, Link linker, Link target)
        {
            if (_memoryManagerIsReady == default)
            {
                throw new InvalidOperationException("Менеджер памяти ещё не готов.");
            }
            if (LinkWasDeleted(source))
            {
                throw new ArgumentException("Удалённая связь не может использоваться в качестве
                ↪  значения.", nameof(source));
            }
            if (LinkWasDeleted(linker))
            {
                throw new ArgumentException("Удалённая связь не может использоваться в качестве
                ↪  значения.", nameof(linker));
            }
            if (LinkWasDeleted(target))
            {
                throw new ArgumentException("Удалённая связь не может использоваться в качестве
                ↪  значения.", nameof(target));
            }
```

```
            Link link = CreateLink(source, linker, target);
            if (link == null)
            {
                throw new InvalidOperationException("Невозможно создать связь.");
            }
            CreatedEvent.Invoke(new LinkDefinition(link));
            return link;
        }

        /// <summary>
        /// <para>
        /// Restores the index.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="index">
        /// <para>The index.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        public static Link Restore(Int index) => Restore((LinkIndex)index);

        /// <summary>
        /// <para>
        /// Restores the index.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="index">
        /// <para>The index.</para>
        /// <para></para>
        /// </param>
        /// <exception cref="InvalidOperationException">
        /// <para>Менеджер памяти ещё не готов.</para>
        /// <para></para>
        /// </exception>
        /// <exception cref="InvalidOperationException">
        /// <para>Связь с указанным адресом удалена, либо не существовала.</para>
        /// <para></para>
        /// </exception>
        /// <exception cref="ArgumentException">
        /// <para>У связи не может быть нулевого адреса.</para>
        /// <para></para>
        /// </exception>
        /// <exception cref="InvalidOperationException">
        /// <para>Указатель не является корректным. </para>
        /// <para></para>
        /// </exception>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        public static Link Restore(LinkIndex index)
        {
            if (_memoryManagerIsReady == default)
            {
                throw new InvalidOperationException("Менеджер памяти ещё не готов.");
            }
            if (index == 0)
            {
                throw new ArgumentException("У связи не может быть нулевого адреса.");
            }
            try
            {
                Link link = index;
                if (LinkDoesNotExist(link))
                {
                    throw new InvalidOperationException("Связь с указанным адресом удалена, либо
                    ↪  не существовала.");
                }
                return link;
            }
            catch (Exception ex)
            {
                throw new InvalidOperationException("Указатель не является корректным.", ex);
```

```
1191            }
1192        }
1193
1194        /// <summary>
1195        /// <para>
1196        /// Creates the mapped using the specified source.
1197        /// </para>
1198        /// <para></para>
1199        /// </summary>
1200        /// <param name="source">
1201        /// <para>The source.</para>
1202        /// <para></para>
1203        /// </param>
1204        /// <param name="linker">
1205        /// <para>The linker.</para>
1206        /// <para></para>
1207        /// </param>
1208        /// <param name="target">
1209        /// <para>The target.</para>
1210        /// <para></para>
1211        /// </param>
1212        /// <param name="mappingIndex">
1213        /// <para>The mapping index.</para>
1214        /// <para></para>
1215        /// </param>
1216        /// <returns>
1217        /// <para>The link</para>
1218        /// <para></para>
1219        /// </returns>
1220        public static Link CreateMapped(Link source, Link linker, Link target, object
            ↪ mappingIndex) => CreateMapped(source, linker, target, Convert.ToInt64(mappingIndex));
1221
1222        /// <summary>
1223        /// <para>
1224        /// Creates the mapped using the specified source.
1225        /// </para>
1226        /// <para></para>
1227        /// </summary>
1228        /// <param name="source">
1229        /// <para>The source.</para>
1230        /// <para></para>
1231        /// </param>
1232        /// <param name="linker">
1233        /// <para>The linker.</para>
1234        /// <para></para>
1235        /// </param>
1236        /// <param name="target">
1237        /// <para>The target.</para>
1238        /// <para></para>
1239        /// </param>
1240        /// <param name="mappingIndex">
1241        /// <para>The mapping index.</para>
1242        /// <para></para>
1243        /// </param>
1244        /// <exception cref="InvalidOperationException">
1245        /// <para>Менеджер памяти ещё не готов.</para>
1246        /// <para></para>
1247        /// </exception>
1248        /// <exception cref="InvalidOperationException">
1249        /// <para>Существующая привязанная связь не соответствует указанным Source, Linker и
            ↪ Target.</para>
1250        /// <para></para>
1251        /// </exception>
1252        /// <exception cref="InvalidOperationException">
1253        /// <para>Установить привязанную связь не удалось.</para>
1254        /// <para></para>
1255        /// </exception>
1256        /// <returns>
1257        /// <para>The mapped link.</para>
1258        /// <para></para>
1259        /// </returns>
1260        public static Link CreateMapped(Link source, Link linker, Link target, Int mappingIndex)
1261        {
1262            if (_memoryManagerIsReady == default)
1263            {
1264                throw new InvalidOperationException("Менеджер памяти ещё не готов.");
1265            }
1266            Link mappedLink = GetMappedLink(mappingIndex);
```

```
            if (mappedLink == null)
            {
                mappedLink = Create(source, linker, target);
                SetMappedLink(mappingIndex, mappedLink);
                if (GetMappedLink(mappingIndex) != mappedLink)
                {
                    throw new InvalidOperationException("Установить привязанную связь не
                    ↪  удалось.");
                }
            }
            else if (!mappedLink.IsMatchingTo(source, linker, target))
            {
                throw new InvalidOperationException("Существующая привязанная связь не
                ↪  соответствует указанным Source, Linker и Target.");
            }
            _linkToMappingIndex[mappedLink] = mappingIndex;
            return mappedLink;
        }

        /// <summary>
        /// <para>
        /// Determines whether try set mapped.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="link">
        /// <para>The link.</para>
        /// <para></para>
        /// </param>
        /// <param name="mappingIndex">
        /// <para>The mapping index.</para>
        /// <para></para>
        /// </param>
        /// <param name="rewrite">
        /// <para>The rewrite.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        public static bool TrySetMapped(Link link, Int mappingIndex, bool rewrite = false)
        {
            Link mappedLink = GetMappedLink(mappingIndex);

            if (mappedLink == null || rewrite)
            {
                mappedLink = link;
                SetMappedLink(mappingIndex, mappedLink);
                if (GetMappedLink(mappingIndex) != mappedLink)
                {
                    return false;
                }
            }
            else if (!mappedLink.IsMatchingTo(link.Source, link.Linker, link.Target))
            {
                return false;
            }
            _linkToMappingIndex[mappedLink] = mappingIndex;
            return true;
        }

        /// <summary>
        /// <para>
        /// Gets the mapped using the specified mapping index.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="mappingIndex">
        /// <para>The mapping index.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        public static Link GetMapped(object mappingIndex) =>
        ↪  GetMapped(Convert.ToInt64(mappingIndex));
```

```
        /// <summary>
        /// <para>
        /// Gets the mapped using the specified mapping index.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="mappingIndex">
        /// <para>The mapping index.</para>
        /// <para></para>
        /// </param>
        /// <exception cref="InvalidOperationException">
        /// <para>Mapped link with index {mappingIndex} is not set.</para>
        /// <para></para>
        /// </exception>
        /// <returns>
        /// <para>The mapped link.</para>
        /// <para></para>
        /// </returns>
        public static Link GetMapped(Int mappingIndex)
        {
            if (!TryGetMapped(mappingIndex, out Link mappedLink))
            {
                throw new InvalidOperationException($"Mapped link with index {mappingIndex} is
                ↪  not set.");
            }
            return mappedLink;
        }

        /// <summary>
        /// <para>
        /// Gets the mapped or default using the specified mapping index.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="mappingIndex">
        /// <para>The mapping index.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The mapped link.</para>
        /// <para></para>
        /// </returns>
        public static Link GetMappedOrDefault(object mappingIndex)
        {
            TryGetMapped(mappingIndex, out Link mappedLink);
            return mappedLink;
        }

        /// <summary>
        /// <para>
        /// Gets the mapped or default using the specified mapping index.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="mappingIndex">
        /// <para>The mapping index.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The mapped link.</para>
        /// <para></para>
        /// </returns>
        public static Link GetMappedOrDefault(Int mappingIndex)
        {
            TryGetMapped(mappingIndex, out Link mappedLink);
            return mappedLink;
        }

        /// <summary>
        /// <para>
        /// Determines whether try get mapped.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="mappingIndex">
        /// <para>The mapping index.</para>
        /// <para></para>
```

```
        /// </param>
        /// <param name="mappedLink">
        /// <para>The mapped link.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        public static bool TryGetMapped(object mappingIndex, out Link mappedLink) =>
        ↪  TryGetMapped(Convert.ToInt64(mappingIndex), out mappedLink);

        /// <summary>
        /// <para>
        /// Determines whether try get mapped.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="mappingIndex">
        /// <para>The mapping index.</para>
        /// <para></para>
        /// </param>
        /// <param name="mappedLink">
        /// <para>The mapped link.</para>
        /// <para></para>
        /// </param>
        /// <exception cref="InvalidOperationException">
        /// <para>Менеджер памяти ещё не готов.</para>
        /// <para></para>
        /// </exception>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        public static bool TryGetMapped(Int mappingIndex, out Link mappedLink)
        {
            if (_memoryManagerIsReady == default)
            {
                throw new InvalidOperationException("Менеджер памяти ещё не готов.");
            }
            mappedLink = GetMappedLink(mappingIndex);
            if (mappedLink != null)
            {
                _linkToMappingIndex[mappedLink] = mappingIndex;
            }
            return mappedLink != null;
        }

        /// <summary>
        /// <para>
        /// Updates the link.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="link">
        /// <para>The link.</para>
        /// <para></para>
        /// </param>
        /// <param name="newSource">
        /// <para>The new source.</para>
        /// <para></para>
        /// </param>
        /// <param name="newLinker">
        /// <para>The new linker.</para>
        /// <para></para>
        /// </param>
        /// <param name="newTarget">
        /// <para>The new target.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link.</para>
        /// <para></para>
        /// </returns>
        public static Link Update(Link link, Link newSource, Link newLinker, Link newTarget)
        {
            Update(ref link, newSource, newLinker, newTarget);
            return link;
```

```
1496            }
1497
1498            /// <summary>
1499            /// <para>
1500            /// Updates the link.
1501            /// </para>
1502            /// <para></para>
1503            /// </summary>
1504            /// <param name="link">
1505            /// <para>The link.</para>
1506            /// <para></para>
1507            /// </param>
1508            /// <param name="newSource">
1509            /// <para>The new source.</para>
1510            /// <para></para>
1511            /// </param>
1512            /// <param name="newLinker">
1513            /// <para>The new linker.</para>
1514            /// <para></para>
1515            /// </param>
1516            /// <param name="newTarget">
1517            /// <para>The new target.</para>
1518            /// <para></para>
1519            /// </param>
1520            /// <exception cref="InvalidOperationException">
1521            /// <para>Менеджер памяти ещё не готов.</para>
1522            /// <para></para>
1523            /// </exception>
1524            /// <exception cref="ArgumentException">
1525            /// <para>Нельзя обновить несуществующую связь. </para>
1526            /// <para></para>
1527            /// </exception>
1528            /// <exception cref="ArgumentException">
1529            /// <para>Удалённая связь не может использоваться в качестве нового значения. </para>
1530            /// <para></para>
1531            /// </exception>
1532            /// <exception cref="ArgumentException">
1533            /// <para>Удалённая связь не может использоваться в качестве нового значения. </para>
1534            /// <para></para>
1535            /// </exception>
1536            /// <exception cref="ArgumentException">
1537            /// <para>Удалённая связь не может использоваться в качестве нового значения. </para>
1538            /// <para></para>
1539            /// </exception>
1540            public static void Update(ref Link link, Link newSource, Link newLinker, Link newTarget)
1541            {
1542                if (_memoryManagerIsReady == default)
1543                {
1544                    throw new InvalidOperationException("Менеджер памяти ещё не готов.");
1545                }
1546                if (LinkDoesNotExist(link))
1547                {
1548                    throw new ArgumentException("Нельзя обновить несуществующую связь.",
                        ↪ nameof(link));
1549                }
1550                if (LinkWasDeleted(newSource))
1551                {
1552                    throw new ArgumentException("Удалённая связь не может использоваться в качестве
                        ↪ нового значения.", nameof(newSource));
1553                }
1554                if (LinkWasDeleted(newLinker))
1555                {
1556                    throw new ArgumentException("Удалённая связь не может использоваться в качестве
                        ↪ нового значения.", nameof(newLinker));
1557                }
1558                if (LinkWasDeleted(newTarget))
1559                {
1560                    throw new ArgumentException("Удалённая связь не может использоваться в качестве
                        ↪ нового значения.", nameof(newTarget));
1561                }
1562                LinkIndex previousLinkIndex = link;
1563                _linkToMappingIndex.TryGetValue(link, out long mappingIndex);
1564                var previousDefinition = new LinkDefinition(link);
1565                link = UpdateLink(link, newSource, newLinker, newTarget);
1566                if (mappingIndex >= 0 && previousLinkIndex != link)
1567                {
1568                    _linkToMappingIndex.Remove(previousLinkIndex);
```

```
                SetMappedLink(mappingIndex, link);
                _linkToMappingIndex.Add(link, mappingIndex);
            }
            UpdatedEvent(previousDefinition, new LinkDefinition(link));
        }

        /// <summary>
        /// <para>
        /// Deletes the link.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="link">
        /// <para>The link.</para>
        /// <para></para>
        /// </param>
        public static void Delete(Link link) => Delete(ref link);

        /// <summary>
        /// <para>
        /// Deletes the link.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="link">
        /// <para>The link.</para>
        /// <para></para>
        /// </param>
        public static void Delete(ref Link link)
        {
            if (LinkDoesNotExist(link))
            {
                return;
            }
            LinkIndex previousLinkIndex = link;
            _linkToMappingIndex.TryGetValue(link, out long mappingIndex);
            var previousDefinition = new LinkDefinition(link);
            DeleteLink(link);
            link = null;
            if (mappingIndex >= 0)
            {
                _linkToMappingIndex.Remove(previousLinkIndex);
                SetMappedLink(mappingIndex, 0);
            }
            DeletedEvent(previousDefinition);
        }

        //public static void Replace(ref Link link, Link replacement)
        //{
        //    if (!MemoryManagerIsReady)
        //        throw new InvalidOperationException("Менеджер памяти ещё не готов.");
        //    if (LinkDoesNotExist(link))
        //        throw new InvalidOperationException("Если связь не существует, её нельзя
        ↪  заменить.");
        //    if (LinkDoesNotExist(replacement))
        //        throw new ArgumentException("Пустая или удалённая связь не может быть
        ↪  замещаемым значением.", "replacement");
        //    link = ReplaceLink(link, replacement);
        //}

        /// <summary>
        /// <para>
        /// Searches the source.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="source">
        /// <para>The source.</para>
        /// <para></para>
        /// </param>
        /// <param name="linker">
        /// <para>The linker.</para>
        /// <para></para>
        /// </param>
        /// <param name="target">
        /// <para>The target.</para>
        /// <para></para>
        /// </param>
```

```
        /// <exception cref="InvalidOperationException">
        /// <para>Выполнить поиск связи можно только по существующим связям.</para>
        /// <para></para>
        /// </exception>
        /// <exception cref="InvalidOperationException">
        /// <para>Менеджер памяти ещё не готов.</para>
        /// <para></para>
        /// </exception>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        public static Link Search(Link source, Link linker, Link target)
        {
            if (_memoryManagerIsReady == default)
            {
                throw new InvalidOperationException("Менеджер памяти ещё не готов.");
            }
            if (LinkDoesNotExist(source) || LinkDoesNotExist(linker) || LinkDoesNotExist(target))
            {
                throw new InvalidOperationException("Выполнить поиск связи можно только по
                ↪  существующим связям.");
            }
            return SearchLink(source, linker, target);
        }

        /// <summary>
        /// <para>
        /// Determines whether exists.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="source">
        /// <para>The source.</para>
        /// <para></para>
        /// </param>
        /// <param name="linker">
        /// <para>The linker.</para>
        /// <para></para>
        /// </param>
        /// <param name="target">
        /// <para>The target.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        public static bool Exists(Link source, Link linker, Link target) => SearchLink(source,
        ↪  linker, target) != 0;

        #endregion

        #region Referers Walkers

        /// <summary>
        /// <para>
        /// Determines whether this instance walk through referers as source.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="walker">
        /// <para>The walker.</para>
        /// <para></para>
        /// </param>
        /// <exception cref="InvalidOperationException">
        /// <para>С несуществующей связью нельзя производитить операции.</para>
        /// <para></para>
        /// </exception>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        public bool WalkThroughReferersAsSource(Func<Link, bool> walker)
        {
            if (LinkDoesNotExist(this))
            {
```

```
                throw new InvalidOperationException("С несуществующей связью нельзя
                ↪  производитить операции.");
            }
            var referers = ReferersBySourceCount;
            if (referers == 1)
            {
                return walker(FirstRefererBySource);
            }
            else if (referers > 1)
            {
                return WalkThroughReferersBySource(this, x => walker(x) ? 1 : 0) != 0;
            }
            else
            {
                return true;
            }
        }

        /// <summary>
        /// <para>
        /// Walks the through referers as source using the specified walker.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="walker">
        /// <para>The walker.</para>
        /// <para></para>
        /// </param>
        /// <exception cref="InvalidOperationException">
        /// <para>С несуществующей связью нельзя производитить операции.</para>
        /// <para></para>
        /// </exception>
        public void WalkThroughReferersAsSource(Action<Link> walker)
        {
            if (LinkDoesNotExist(this))
            {
                throw new InvalidOperationException("С несуществующей связью нельзя
                ↪  производитить операции.");
            }
            var referers = ReferersBySourceCount;
            if (referers == 1)
            {
                walker(FirstRefererBySource);
            }
            else if (referers > 1)
            {
                WalkThroughAllReferersBySource(this, x => walker(x));
            }
        }

        /// <summary>
        /// <para>
        /// Determines whether this instance walk through referers as linker.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="walker">
        /// <para>The walker.</para>
        /// <para></para>
        /// </param>
        /// <exception cref="InvalidOperationException">
        /// <para>С несуществующей связью нельзя производитить операции.</para>
        /// <para></para>
        /// </exception>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        public bool WalkThroughReferersAsLinker(Func<Link, bool> walker)
        {
            if (LinkDoesNotExist(this))
            {
                throw new InvalidOperationException("С несуществующей связью нельзя
                ↪  производитить операции.");
            }
            var referers = ReferersByLinkerCount;
            if (referers == 1)
            {
```

```
            return walker(FirstRefererByLinker);
        }
        else if (referers > 1)
        {
            return WalkThroughReferersByLinker(this, x => walker(x) ? 1 : 0) != 0;
        }
        else
        {
            return true;
        }
    }

    /// <summary>
    /// <para>
    /// Walks the through referers as linker using the specified walker.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <param name="walker">
    /// <para>The walker.</para>
    /// <para></para>
    /// </param>
    /// <exception cref="InvalidOperationException">
    /// <para>С несуществующей связью нельзя производитить операции.</para>
    /// <para></para>
    /// </exception>
    public void WalkThroughReferersAsLinker(Action<Link> walker)
    {
        if (LinkDoesNotExist(this))
        {
            throw new InvalidOperationException("С несуществующей связью нельзя
            ↪  производитить операции.");
        }
        var referers = ReferersByLinkerCount;
        if (referers == 1)
        {
            walker(FirstRefererByLinker);
        }
        else if (referers > 1)
        {
            WalkThroughAllReferersByLinker(this, x => walker(x));
        }
    }

    /// <summary>
    /// <para>
    /// Determines whether this instance walk through referers as target.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <param name="walker">
    /// <para>The walker.</para>
    /// <para></para>
    /// </param>
    /// <exception cref="InvalidOperationException">
    /// <para>С несуществующей связью нельзя производитить операции.</para>
    /// <para></para>
    /// </exception>
    /// <returns>
    /// <para>The bool</para>
    /// <para></para>
    /// </returns>
    public bool WalkThroughReferersAsTarget(Func<Link, bool> walker)
    {
        if (LinkDoesNotExist(this))
        {
            throw new InvalidOperationException("С несуществующей связью нельзя
            ↪  производитить операции.");
        }
        var referers = ReferersByTargetCount;
        if (referers == 1)
        {
            return walker(FirstRefererByTarget);
        }
        else if (referers > 1)
        {
            return WalkThroughReferersByTarget(this, x => walker(x) ? 1 : 0) != 0;
        }
```

```
            else
            {
                return true;
            }
        }

        /// <summary>
        /// <para>
        /// Walks the through referers as target using the specified walker.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="walker">
        /// <para>The walker.</para>
        /// <para></para>
        /// </param>
        /// <exception cref="InvalidOperationException">
        /// <para>С несуществующей связью нельзя производитить операции.</para>
        /// <para></para>
        /// </exception>
        public void WalkThroughReferersAsTarget(Action<Link> walker)
        {
            if (LinkDoesNotExist(this))
            {
                throw new InvalidOperationException("С несуществующей связью нельзя
                ↪  производитить операции.");
            }
            var referers = ReferersByTargetCount;
            if (referers == 1)
            {
                walker(FirstRefererByTarget);
            }
            else if (referers > 1)
            {
                WalkThroughAllReferersByTarget(this, x => walker(x));
            }
        }

        /// <summary>
        /// <para>
        /// Walks the through referers using the specified walker.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="walker">
        /// <para>The walker.</para>
        /// <para></para>
        /// </param>
        /// <exception cref="InvalidOperationException">
        /// <para>С несуществующей связью нельзя производитить операции.</para>
        /// <para></para>
        /// </exception>
        public void WalkThroughReferers(Action<Link> walker)
        {
            if (LinkDoesNotExist(this))
            {
                throw new InvalidOperationException("С несуществующей связью нельзя
                ↪  производитить операции.");
            }
            void wrapper(ulong x) => walker(x);
            WalkThroughAllReferersBySource(this, wrapper);
            WalkThroughAllReferersByLinker(this, wrapper);
            WalkThroughAllReferersByTarget(this, wrapper);
        }

        /// <summary>
        /// <para>
        /// Walks the through referers using the specified walker.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="walker">
        /// <para>The walker.</para>
        /// <para></para>
        /// </param>
        /// <exception cref="InvalidOperationException">
        /// <para>С несуществующей связью нельзя производитить операции.</para>
        /// <para></para>
```

```
        /// </exception>
        public void WalkThroughReferers(Func<Link, bool> walker)
        {
            if (LinkDoesNotExist(this))
            {
                throw new InvalidOperationException("С несуществующей связью нельзя
                ↪  производитить операции.");
            }
            long wrapper(ulong x) => walker(x) ? 1 : 0;
            WalkThroughReferersBySource(this, wrapper);
            WalkThroughReferersByLinker(this, wrapper);
            WalkThroughReferersByTarget(this, wrapper);
        }

        /// <summary>
        /// <para>
        /// Determines whether walk through all links.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="walker">
        /// <para>The walker.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        public static bool WalkThroughAllLinks(Func<Link, bool> walker) => WalkThroughLinks(x =>
        ↪  walker(x) ? 1 : 0) != 0;

        /// <summary>
        /// <para>
        /// Walks the through all links using the specified walker.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="walker">
        /// <para>The walker.</para>
        /// <para></para>
        /// </param>
        public static void WalkThroughAllLinks(Action<Link> walker) => WalkThroughAllLinks(new
        ↪  Visitor(x => walker(x)));

        #endregion
    }
}
```

1.6 ./csharp/Platform.Data.Triplets/LinkConverter.cs

```
using System;
using System.Collections.Generic;
using Platform.Data.Sequences;

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

namespace Platform.Data.Triplets
{
    /// <summary>
    /// <para>
    /// Represents the link converter.
    /// </para>
    /// <para></para>
    /// </summary>
    public static class LinkConverter
    {
        /// <summary>
        /// <para>
        /// Creates the list using the specified links.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="links">
        /// <para>The links.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The element.</para>
        /// <para></para>
```

```
        /// </returns>
        public static Link FromList(List<Link> links)
        {
            var i = links.Count - 1;
            var element = links[i];
            while (--i >= 0)
            {
                element = links[i] & element;
            }
            return element;
        }

        /// <summary>
        /// <para>
        /// Creates the list using the specified links.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="links">
        /// <para>The links.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The element.</para>
        /// <para></para>
        /// </returns>
        public static Link FromList(Link[] links)
        {
            var i = links.Length - 1;
            var element = links[i];
            while (--i >= 0)
            {
                element = links[i] & element;
            }
            return element;
        }

        /// <summary>
        /// <para>
        /// Returns the list using the specified link.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="link">
        /// <para>The link.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The list.</para>
        /// <para></para>
        /// </returns>
        public static List<Link> ToList(Link link)
        {
            var list = new List<Link>();
            SequenceWalker.WalkRight(link, x => x.Source, x => x.Target, x => x.Linker !=
            ↪  Net.And, list.Add);
            return list;
        }

        /// <summary>
        /// <para>
        /// Creates the number using the specified number.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="number">
        /// <para>The number.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        public static Link FromNumber(long number) => NumberHelpers.FromNumber(number);

        /// <summary>
        /// <para>
        /// Returns the number using the specified number.
```

```
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="number">
        /// <para>The number.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The long</para>
        /// <para></para>
        /// </returns>
        public static long ToNumber(Link number) => NumberHelpers.ToNumber(number);

        /// <summary>
        /// <para>
        /// Creates the char using the specified c.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="c">
        /// <para>The .</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        public static Link FromChar(char c) => CharacterHelpers.FromChar(c);

        /// <summary>
        /// <para>
        /// Returns the char using the specified char link.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="charLink">
        /// <para>The char link.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The char</para>
        /// <para></para>
        /// </returns>
        public static char ToChar(Link charLink) => CharacterHelpers.ToChar(charLink);

        /// <summary>
        /// <para>
        /// Creates the chars using the specified chars.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="chars">
        /// <para>The chars.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        public static Link FromChars(char[] chars) => FromObjectsToSequence(chars, FromChar);

        /// <summary>
        /// <para>
        /// Creates the chars using the specified chars.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="chars">
        /// <para>The chars.</para>
        /// <para></para>
        /// </param>
        /// <param name="takeFrom">
        /// <para>The take from.</para>
        /// <para></para>
        /// </param>
        /// <param name="takeUntil">
        /// <para>The take until.</para>
        /// <para></para>
```

```
185             /// </param>
186             /// <returns>
187             /// <para>The link</para>
188             /// <para></para>
189             /// </returns>
190             public static Link FromChars(char[] chars, int takeFrom, int takeUntil) =>
                ↪  FromObjectsToSequence(chars, takeFrom, takeUntil, FromChar);
191
192             /// <summary>
193             /// <para>
194             /// Creates the numbers using the specified numbers.
195             /// </para>
196             /// <para></para>
197             /// </summary>
198             /// <param name="numbers">
199             /// <para>The numbers.</para>
200             /// <para></para>
201             /// </param>
202             /// <returns>
203             /// <para>The link</para>
204             /// <para></para>
205             /// </returns>
206             public static Link FromNumbers(long[] numbers) => FromObjectsToSequence(numbers,
                ↪  FromNumber);
207
208             /// <summary>
209             /// <para>
210             /// Creates the numbers using the specified numbers.
211             /// </para>
212             /// <para></para>
213             /// </summary>
214             /// <param name="numbers">
215             /// <para>The numbers.</para>
216             /// <para></para>
217             /// </param>
218             /// <param name="takeFrom">
219             /// <para>The take from.</para>
220             /// <para></para>
221             /// </param>
222             /// <param name="takeUntil">
223             /// <para>The take until.</para>
224             /// <para></para>
225             /// </param>
226             /// <returns>
227             /// <para>The link</para>
228             /// <para></para>
229             /// </returns>
230             public static Link FromNumbers(long[] numbers, int takeFrom, int takeUntil) =>
                ↪  FromObjectsToSequence(numbers, takeFrom, takeUntil, FromNumber);
231
232             /// <summary>
233             /// <para>
234             /// Creates the numbers using the specified numbers.
235             /// </para>
236             /// <para></para>
237             /// </summary>
238             /// <param name="numbers">
239             /// <para>The numbers.</para>
240             /// <para></para>
241             /// </param>
242             /// <returns>
243             /// <para>The link</para>
244             /// <para></para>
245             /// </returns>
246             public static Link FromNumbers(ushort[] numbers) => FromObjectsToSequence(numbers, x =>
                ↪  FromNumber(x));
247
248             /// <summary>
249             /// <para>
250             /// Creates the numbers using the specified numbers.
251             /// </para>
252             /// <para></para>
253             /// </summary>
254             /// <param name="numbers">
255             /// <para>The numbers.</para>
256             /// <para></para>
257             /// </param>
258             /// <param name="takeFrom">
```

```
        /// <para>The take from.</para>
        /// <para></para>
        /// </param>
        /// <param name="takeUntil">
        /// <para>The take until.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        public static Link FromNumbers(ushort[] numbers, int takeFrom, int takeUntil) =>
        ↪  FromObjectsToSequence(numbers, takeFrom, takeUntil, x => FromNumber(x));

        /// <summary>
        /// <para>
        /// Creates the numbers using the specified numbers.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="numbers">
        /// <para>The numbers.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        public static Link FromNumbers(uint[] numbers) => FromObjectsToSequence(numbers, x =>
        ↪  FromNumber(x));

        /// <summary>
        /// <para>
        /// Creates the numbers using the specified numbers.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="numbers">
        /// <para>The numbers.</para>
        /// <para></para>
        /// </param>
        /// <param name="takeFrom">
        /// <para>The take from.</para>
        /// <para></para>
        /// </param>
        /// <param name="takeUntil">
        /// <para>The take until.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        public static Link FromNumbers(uint[] numbers, int takeFrom, int takeUntil) =>
        ↪  FromObjectsToSequence(numbers, takeFrom, takeUntil, x => FromNumber(x));

        /// <summary>
        /// <para>
        /// Creates the numbers using the specified numbers.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="numbers">
        /// <para>The numbers.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        public static Link FromNumbers(byte[] numbers) => FromObjectsToSequence(numbers, x =>
        ↪  FromNumber(x));

        /// <summary>
        /// <para>
        /// Creates the numbers using the specified numbers.
        /// </para>
        /// <para></para>
```

```
        /// </summary>
        /// <param name="numbers">
        /// <para>The numbers.</para>
        /// <para></para>
        /// </param>
        /// <param name="takeFrom">
        /// <para>The take from.</para>
        /// <para></para>
        /// </param>
        /// <param name="takeUntil">
        /// <para>The take until.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        public static Link FromNumbers(byte[] numbers, int takeFrom, int takeUntil) =>
        ↪  FromObjectsToSequence(numbers, takeFrom, takeUntil, x => FromNumber(x));

        /// <summary>
        /// <para>
        /// Creates the numbers using the specified numbers.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="numbers">
        /// <para>The numbers.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        public static Link FromNumbers(bool[] numbers) => FromObjectsToSequence(numbers, x =>
        ↪  FromNumber(x ? 1 : 0));

        /// <summary>
        /// <para>
        /// Creates the numbers using the specified numbers.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="numbers">
        /// <para>The numbers.</para>
        /// <para></para>
        /// </param>
        /// <param name="takeFrom">
        /// <para>The take from.</para>
        /// <para></para>
        /// </param>
        /// <param name="takeUntil">
        /// <para>The take until.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        public static Link FromNumbers(bool[] numbers, int takeFrom, int takeUntil) =>
        ↪  FromObjectsToSequence(numbers, takeFrom, takeUntil, x => FromNumber(x ? 1 : 0));

        /// <summary>
        /// <para>
        /// Creates the objects to sequence using the specified objects.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <typeparam name="T">
        /// <para>The .</para>
        /// <para></para>
        /// </typeparam>
        /// <param name="objects">
        /// <para>The objects.</para>
        /// <para></para>
        /// </param>
        /// <param name="converter">
        /// <para>The converter.</para>
```

```
408         /// <para></para>
409         /// </param>
410         /// <returns>
411         /// <para>The link</para>
412         /// <para></para>
413         /// </returns>
414         public static Link FromObjectsToSequence<T>(T[] objects, Func<T, Link> converter) =>
            ↪ FromObjectsToSequence(objects, 0, objects.Length, converter);
415
416         /// <summary>
417         /// <para>
418         /// Creates the objects to sequence using the specified objects.
419         /// </para>
420         /// <para></para>
421         /// </summary>
422         /// <typeparam name="T">
423         /// <para>The .</para>
424         /// <para></para>
425         /// </typeparam>
426         /// <param name="objects">
427         /// <para>The objects.</para>
428         /// <para></para>
429         /// </param>
430         /// <param name="takeFrom">
431         /// <para>The take from.</para>
432         /// <para></para>
433         /// </param>
434         /// <param name="takeUntil">
435         /// <para>The take until.</para>
436         /// <para></para>
437         /// </param>
438         /// <param name="converter">
439         /// <para>The converter.</para>
440         /// <para></para>
441         /// </param>
442         /// <exception cref="ArgumentOutOfRangeException">
443         /// <para>Нельзя преобразовать пустой список к связям.</para>
444         /// <para></para>
445         /// </exception>
446         /// <returns>
447         /// <para>The link</para>
448         /// <para></para>
449         /// </returns>
450         public static Link FromObjectsToSequence<T>(T[] objects, int takeFrom, int takeUntil,
            ↪ Func<T, Link> converter)
451         {
452             var length = takeUntil - takeFrom;
453             if (length <= 0)
454             {
455                 throw new ArgumentOutOfRangeException(nameof(takeUntil), "Нельзя преобразовать
                    ↪ пустой список к связям.");
456             }
457             var copy = new Link[length];
458             for (int i = takeFrom, j = 0; i < takeUntil; i++, j++)
459             {
460                 copy[j] = converter(objects[i]);
461             }
462             return FromList(copy);
463         }
464
465         /// <summary>
466         /// <para>
467         /// Creates the chars using the specified str.
468         /// </para>
469         /// <para></para>
470         /// </summary>
471         /// <param name="str">
472         /// <para>The str.</para>
473         /// <para></para>
474         /// </param>
475         /// <returns>
476         /// <para>The link</para>
477         /// <para></para>
478         /// </returns>
479         public static Link FromChars(string str)
480         {
481             var copy = new Link[str.Length];
482             for (var i = 0; i < copy.Length; i++)
```

```
            {
                copy[i] = FromChar(str[i]);
            }
            return FromList(copy);
        }

        /// <summary>
        /// <para>
        /// Creates the string using the specified str.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="str">
        /// <para>The str.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The str link.</para>
        /// <para></para>
        /// </returns>
        public static Link FromString(string str)
        {
            var copy = new Link[str.Length];
            for (var i = 0; i < copy.Length; i++)
            {
                copy[i] = FromChar(str[i]);
            }
            var strLink = Link.Create(Net.String, Net.ThatConsistsOf, FromList(copy));
            return strLink;
        }

        /// <summary>
        /// <para>
        /// Returns the string using the specified link.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="link">
        /// <para>The link.</para>
        /// <para></para>
        /// </param>
        /// <exception cref="ArgumentOutOfRangeException">
        /// <para>Specified link is not a string.</para>
        /// <para></para>
        /// </exception>
        /// <returns>
        /// <para>The string</para>
        /// <para></para>
        /// </returns>
        public static string ToString(Link link)
        {
            if (link.IsString())
            {
                return ToString(ToList(link.Target));
            }
            throw new ArgumentOutOfRangeException(nameof(link), "Specified link is not a
            ↪  string.");
        }

        /// <summary>
        /// <para>
        /// Returns the string using the specified char links.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="charLinks">
        /// <para>The char links.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The string</para>
        /// <para></para>
        /// </returns>
        public static string ToString(List<Link> charLinks)
        {
            var chars = new char[charLinks.Count];
            for (var i = 0; i < charLinks.Count; i++)
            {
```

```
                chars[i] = ToChar(charLinks[i]);
            }
            return new string(chars);
        }
    }
}
```

1.7 ./csharp/Platform.Data.Triplets/LinkExtensions.cs

```
using System;
using System.Collections.Generic;
using System.Text;
using Platform.Data.Sequences;
using Platform.Data.Triplets.Sequences;

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

namespace Platform.Data.Triplets
{
    /// <summary>
    /// <para>
    /// Represents the link extensions.
    /// </para>
    /// <para></para>
    /// </summary>
    public static class LinkExtensions
    {
        /// <summary>
        /// <para>
        /// Sets the name using the specified link.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="link">
        /// <para>The link.</para>
        /// <para></para>
        /// </param>
        /// <param name="name">
        /// <para>The name.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link.</para>
        /// <para></para>
        /// </returns>
        public static Link SetName(this Link link, string name)
        {
            Link.Create(link, Net.Has, Link.Create(Net.Name, Net.ThatIsRepresentedBy,
            ↪  LinkConverter.FromString(name)));
            return link; // Chaining
        }

        /// <summary>
        /// <para>
        /// The link.
        /// </para>
        /// <para></para>
        /// </summary>
        private static readonly HashSet<Link> _linksWithNamesGatheringProcess = new
        ↪  HashSet<Link>();

        /// <summary>
        /// <para>
        /// Determines whether try get name.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="link">
        /// <para>The link.</para>
        /// <para></para>
        /// </param>
        /// <param name="str">
        /// <para>The str.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
```

```
69          public static bool TryGetName(this Link link, out string str)
70          {
71              // Защита от зацикливания
72              if (!_linksWithNamesGatheringProcess.Add(link))
73              {
74                  str = "...";
75                  return true;
76              }
77              try
78              {
79                  if (link != null)
80                  {
81                      if (link.Linker == Net.And)
82                      {
83                          str = SequenceHelpers.FormatSequence(link);
84                          return true;
85                      }
86                      else if (link.IsGroup())
87                      {
88                          str = LinkConverter.ToString(LinkConverter.ToList(link.Target));
89                          return true;
90                      }
91                      else if (link.IsChar())
92                      {
93                          str = LinkConverter.ToChar(link).ToString();
94                          return true;
95                      }
96                      else if (link.TryGetSpecificName(out str))
97                      {
98                          return true;
99                      }
100
101                     if (link.Source == link || link.Linker == link || link.Target == link)
102                     {
103                         return false;
104                     }
105
106                     if (link.Source.TryGetName(out string sourceName) &&
                        ↪  link.Linker.TryGetName(out string linkerName) &&
                        ↪  link.Target.TryGetName(out string targetName))
107                     {
108                         var sb = new StringBuilder();
109                         sb.Append(sourceName).Append(' ').Append(linkerName).Append('
                            ↪  ').Append(targetName);
110                         str = sb.ToString();
111                         return true;
112                     }
113                 }
114                 str = null;
115                 return false;
116             }
117             finally
118             {
119                 _linksWithNamesGatheringProcess.Remove(link);
120             }
121         }
122
123         /// <summary>
124         /// <para>
125         /// Determines whether try get specific name.
126         /// </para>
127         /// <para></para>
128         /// </summary>
129         /// <param name="link">
130         /// <para>The link.</para>
131         /// <para></para>
132         /// </param>
133         /// <param name="name">
134         /// <para>The name.</para>
135         /// <para></para>
136         /// </param>
137         /// <returns>
138         /// <para>The bool</para>
139         /// <para></para>
140         /// </returns>
141         public static bool TryGetSpecificName(this Link link, out string name)
142         {
143             string nameLocal = null;
144             if (Net.Name.ReferersBySourceCount < link.ReferersBySourceCount)
```

```
145             {
146                 Net.Name.WalkThroughReferersAsSource(referer =>
147                 {
148                     if (referer.Linker == Net.ThatIsRepresentedBy)
149                     {
150                         if (Link.Exists(link, Net.Has, referer))
151                         {
152                             nameLocal = LinkConverter.ToString(referer.Target);
153                             return false; // Останавливаем проход
154                         }
155                     }
156                     return true;
157                 });
158             }
159             else
160             {
161                 link.WalkThroughReferersAsSource(referer =>
162                 {
163                     if (referer.Linker == Net.Has)
164                     {
165                         var nameLink = referer.Target;
166                         if (nameLink.Source == Net.Name && nameLink.Linker ==
                            ↪  Net.ThatIsRepresentedBy)
167                         {
168                             nameLocal = LinkConverter.ToString(nameLink.Target);
169                             return false; // Останавливаем проход
170                         }
171                     }
172                     return true;
173                 });
174             }
175 
176             name = nameLocal;
177             return nameLocal != null;
178         }
179 
180         // Проверка на пренадлежность классу
181         /// <summary>
182         /// <para>
183         /// Determines whether is.
184         /// </para>
185         /// <para></para>
186         /// </summary>
187         /// <param name="link">
188         /// <para>The link.</para>
189         /// <para></para>
190         /// </param>
191         /// <param name="@class">
192         /// <para>The class.</para>
193         /// <para></para>
194         /// </param>
195         /// <returns>
196         /// <para>The bool</para>
197         /// <para></para>
198         /// </returns>
199         public static bool Is(this Link link, Link @class)
200         {
201             if (link.Linker == Net.IsA)
202             {
203                 if (link.Target == @class)
204                 {
205                     return true;
206                 }
207                 else
208                 {
209                     return link.Target.Is(@class);
210                 }
211             }
212             return false;
213         }
214 
215         // Несколько не правильное определение, так выйдет, что любая сумма входящая в диапазон
            ↪  значений char будет символом.
216         // Нужно изменить определение чара, идеально: char consists of sum of [8, 64].
217         /// <summary>
218         /// <para>
219         /// Determines whether is char.
220         /// </para>
```

```
221         /// <para></para>
222         /// </summary>
223         /// <param name="link">
224         /// <para>The link.</para>
225         /// <para></para>
226         /// </param>
227         /// <returns>
228         /// <para>The bool</para>
229         /// <para></para>
230         /// </returns>
231         public static bool IsChar(this Link link) => CharacterHelpers.IsChar(link);
232
233         /// <summary>
234         /// <para>
235         /// Determines whether is group.
236         /// </para>
237         /// <para></para>
238         /// </summary>
239         /// <param name="link">
240         /// <para>The link.</para>
241         /// <para></para>
242         /// </param>
243         /// <returns>
244         /// <para>The bool</para>
245         /// <para></para>
246         /// </returns>
247         public static bool IsGroup(this Link link) => link != null && link.Source == Net.Group
            ↪  && link.Linker == Net.ThatConsistsOf;
248
249         /// <summary>
250         /// <para>
251         /// Determines whether is sum.
252         /// </para>
253         /// <para></para>
254         /// </summary>
255         /// <param name="link">
256         /// <para>The link.</para>
257         /// <para></para>
258         /// </param>
259         /// <returns>
260         /// <para>The bool</para>
261         /// <para></para>
262         /// </returns>
263         public static bool IsSum(this Link link) => link != null && link.Source == Net.Sum &&
            ↪  link.Linker == Net.Of;
264
265         /// <summary>
266         /// <para>
267         /// Determines whether is string.
268         /// </para>
269         /// <para></para>
270         /// </summary>
271         /// <param name="link">
272         /// <para>The link.</para>
273         /// <para></para>
274         /// </param>
275         /// <returns>
276         /// <para>The bool</para>
277         /// <para></para>
278         /// </returns>
279         public static bool IsString(this Link link) => link != null && link.Source == Net.String
            ↪  && link.Linker == Net.ThatConsistsOf;
280
281         /// <summary>
282         /// <para>
283         /// Determines whether is name.
284         /// </para>
285         /// <para></para>
286         /// </summary>
287         /// <param name="link">
288         /// <para>The link.</para>
289         /// <para></para>
290         /// </param>
291         /// <returns>
292         /// <para>The bool</para>
293         /// <para></para>
294         /// </returns>
```

```
295         public static bool IsName(this Link link) => link != null && link.Source == Net.Name &&
            ↪  link.Linker == Net.Of;
296
297         /// <summary>
298         /// <para>
299         /// Gets the array of rererers by source using the specified link.
300         /// </para>
301         /// <para></para>
302         /// </summary>
303         /// <param name="link">
304         /// <para>The link.</para>
305         /// <para></para>
306         /// </param>
307         /// <returns>
308         /// <para>The link array</para>
309         /// <para></para>
310         /// </returns>
311         public static Link[] GetArrayOfRerererersBySource(this Link link)
312         {
313             if (link == null)
314             {
315                 return new Link[0];
316             }
317             else
318             {
319                 var array = new Link[link.ReferersBySourceCount];
320                 var index = 0;
321                 link.WalkThroughReferersAsSource(referer => array[index++] = referer);
322                 return array;
323             }
324         }
325
326         /// <summary>
327         /// <para>
328         /// Gets the array of rererers by linker using the specified link.
329         /// </para>
330         /// <para></para>
331         /// </summary>
332         /// <param name="link">
333         /// <para>The link.</para>
334         /// <para></para>
335         /// </param>
336         /// <returns>
337         /// <para>The link array</para>
338         /// <para></para>
339         /// </returns>
340         public static Link[] GetArrayOfRerererersByLinker(this Link link)
341         {
342             if (link == null)
343             {
344                 return new Link[0];
345             }
346             else
347             {
348                 var array = new Link[link.ReferersByLinkerCount];
349                 var index = 0;
350                 link.WalkThroughReferersAsLinker(referer => array[index++] = referer);
351                 return array;
352             }
353         }
354
355         /// <summary>
356         /// <para>
357         /// Gets the array of rererers by target using the specified link.
358         /// </para>
359         /// <para></para>
360         /// </summary>
361         /// <param name="link">
362         /// <para>The link.</para>
363         /// <para></para>
364         /// </param>
365         /// <returns>
366         /// <para>The link array</para>
367         /// <para></para>
368         /// </returns>
369         public static Link[] GetArrayOfRerererersByTarget(this Link link)
370         {
371             if (link == null)
```

```
            {
                return new Link[0];
            }
            else
            {
                var array = new Link[link.ReferersByTargetCount];
                var index = 0;
                link.WalkThroughReferersAsTarget(referer => array[index++] = referer);
                return array;
            }
        }

        /// <summary>
        /// <para>
        /// Walks the through sequence using the specified link.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="link">
        /// <para>The link.</para>
        /// <para></para>
        /// </param>
        /// <param name="action">
        /// <para>The action.</para>
        /// <para></para>
        /// </param>
        public static void WalkThroughSequence(this Link link, Action<Link> action) =>
        ↪ SequenceWalker.WalkRight(link, x => x.Source, x => x.Target, x => x.Linker !=
        ↪ Net.And, action);
    }
}
```

1.8 ./csharp/Platform.Data.Triplets/Net.cs

```
using Platform.Threading;

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

namespace Platform.Data.Triplets
{
    /// <summary>
    /// <para>
    /// The net mapping enum.
    /// </para>
    /// <para></para>
    /// </summary>
    public enum NetMapping : long
    {
        /// <summary>
        /// <para>
        /// The link net mapping.
        /// </para>
        /// <para></para>
        /// </summary>
        Link,
        /// <summary>
        /// <para>
        /// The thing net mapping.
        /// </para>
        /// <para></para>
        /// </summary>
        Thing,
        /// <summary>
        /// <para>
        /// The is net mapping.
        /// </para>
        /// <para></para>
        /// </summary>
        IsA,
        /// <summary>
        /// <para>
        /// The is not net mapping.
        /// </para>
        /// <para></para>
        /// </summary>
        IsNotA,

        /// <summary>
        /// <para>
        /// The of net mapping.
```

```
        /// </para>
        /// <para></para>
        /// </summary>
        Of,
        /// <summary>
        /// <para>
        /// The and net mapping.
        /// </para>
        /// <para></para>
        /// </summary>
        And,
        /// <summary>
        /// <para>
        /// The that consists of net mapping.
        /// </para>
        /// <para></para>
        /// </summary>
        ThatConsistsOf,
        /// <summary>
        /// <para>
        /// The has net mapping.
        /// </para>
        /// <para></para>
        /// </summary>
        Has,
        /// <summary>
        /// <para>
        /// The contains net mapping.
        /// </para>
        /// <para></para>
        /// </summary>
        Contains,
        /// <summary>
        /// <para>
        /// The contained by net mapping.
        /// </para>
        /// <para></para>
        /// </summary>
        ContainedBy,

        /// <summary>
        /// <para>
        /// The one net mapping.
        /// </para>
        /// <para></para>
        /// </summary>
        One,
        /// <summary>
        /// <para>
        /// The zero net mapping.
        /// </para>
        /// <para></para>
        /// </summary>
        Zero,

        /// <summary>
        /// <para>
        /// The sum net mapping.
        /// </para>
        /// <para></para>
        /// </summary>
        Sum,
        /// <summary>
        /// <para>
        /// The character net mapping.
        /// </para>
        /// <para></para>
        /// </summary>
        Character,
        /// <summary>
        /// <para>
        /// The string net mapping.
        /// </para>
        /// <para></para>
        /// </summary>
        String,
        /// <summary>
        /// <para>
        /// The name net mapping.
```

```
        /// </para>
        /// <para></para>
        /// </summary>
        Name,

        /// <summary>
        /// <para>
        /// The set net mapping.
        /// </para>
        /// <para></para>
        /// </summary>
        Set,
        /// <summary>
        /// <para>
        /// The group net mapping.
        /// </para>
        /// <para></para>
        /// </summary>
        Group,

        /// <summary>
        /// <para>
        /// The parsed from net mapping.
        /// </para>
        /// <para></para>
        /// </summary>
        ParsedFrom,
        /// <summary>
        /// <para>
        /// The that is net mapping.
        /// </para>
        /// <para></para>
        /// </summary>
        ThatIs,
        /// <summary>
        /// <para>
        /// The that is before net mapping.
        /// </para>
        /// <para></para>
        /// </summary>
        ThatIsBefore,
        /// <summary>
        /// <para>
        /// The that is between net mapping.
        /// </para>
        /// <para></para>
        /// </summary>
        ThatIsBetween,
        /// <summary>
        /// <para>
        /// The that is after net mapping.
        /// </para>
        /// <para></para>
        /// </summary>
        ThatIsAfter,
        /// <summary>
        /// <para>
        /// The that is represented by net mapping.
        /// </para>
        /// <para></para>
        /// </summary>
        ThatIsRepresentedBy,
        /// <summary>
        /// <para>
        /// The that has net mapping.
        /// </para>
        /// <para></para>
        /// </summary>
        ThatHas,

        /// <summary>
        /// <para>
        /// The text net mapping.
        /// </para>
        /// <para></para>
        /// </summary>
        Text,
        /// <summary>
        /// <para>
```

```
        /// The path net mapping.
        /// </para>
        /// <para></para>
        /// </summary>
        Path,
        /// <summary>
        /// <para>
        /// The content net mapping.
        /// </para>
        /// <para></para>
        /// </summary>
        Content,
        /// <summary>
        /// <para>
        /// The empty content net mapping.
        /// </para>
        /// <para></para>
        /// </summary>
        EmptyContent,
        /// <summary>
        /// <para>
        /// The empty net mapping.
        /// </para>
        /// <para></para>
        /// </summary>
        Empty,
        /// <summary>
        /// <para>
        /// The alphabet net mapping.
        /// </para>
        /// <para></para>
        /// </summary>
        Alphabet,
        /// <summary>
        /// <para>
        /// The letter net mapping.
        /// </para>
        /// <para></para>
        /// </summary>
        Letter,
        /// <summary>
        /// <para>
        /// The case net mapping.
        /// </para>
        /// <para></para>
        /// </summary>
        Case,
        /// <summary>
        /// <para>
        /// The upper net mapping.
        /// </para>
        /// <para></para>
        /// </summary>
        Upper,
        /// <summary>
        /// <para>
        /// The upper case net mapping.
        /// </para>
        /// <para></para>
        /// </summary>
        UpperCase,
        /// <summary>
        /// <para>
        /// The lower net mapping.
        /// </para>
        /// <para></para>
        /// </summary>
        Lower,
        /// <summary>
        /// <para>
        /// The lower case net mapping.
        /// </para>
        /// <para></para>
        /// </summary>
        LowerCase,
        /// <summary>
        /// <para>
        /// The code net mapping.
```

```
283             /// </para>
284             /// <para></para>
285             /// </summary>
286             Code
287         }
288 
289         /// <summary>
290         /// <para>
291         /// Represents the net.
292         /// </para>
293         /// <para></para>
294         /// </summary>
295         public static class Net
296         {
297             /// <summary>
298             /// <para>
299             /// Gets or sets the link value.
300             /// </para>
301             /// <para></para>
302             /// </summary>
303             public static Link Link { get; private set; }
304             /// <summary>
305             /// <para>
306             /// Gets or sets the thing value.
307             /// </para>
308             /// <para></para>
309             /// </summary>
310             public static Link Thing { get; private set; }
311             /// <summary>
312             /// <para>
313             /// Gets or sets the is a value.
314             /// </para>
315             /// <para></para>
316             /// </summary>
317             public static Link IsA { get; private set; }
318             /// <summary>
319             /// <para>
320             /// Gets or sets the is not a value.
321             /// </para>
322             /// <para></para>
323             /// </summary>
324             public static Link IsNotA { get; private set; }
325 
326             /// <summary>
327             /// <para>
328             /// Gets or sets the of value.
329             /// </para>
330             /// <para></para>
331             /// </summary>
332             public static Link Of { get; private set; }
333             /// <summary>
334             /// <para>
335             /// Gets or sets the and value.
336             /// </para>
337             /// <para></para>
338             /// </summary>
339             public static Link And { get; private set; }
340             /// <summary>
341             /// <para>
342             /// Gets or sets the that consists of value.
343             /// </para>
344             /// <para></para>
345             /// </summary>
346             public static Link ThatConsistsOf { get; private set; }
347             /// <summary>
348             /// <para>
349             /// Gets or sets the has value.
350             /// </para>
351             /// <para></para>
352             /// </summary>
353             public static Link Has { get; private set; }
354             /// <summary>
355             /// <para>
356             /// Gets or sets the contains value.
357             /// </para>
358             /// <para></para>
359             /// </summary>
360             public static Link Contains { get; private set; }
```

```
        /// <summary>
        /// <para>
        /// Gets or sets the contained by value.
        /// </para>
        /// <para></para>
        /// </summary>
        public static Link ContainedBy { get; private set; }

        /// <summary>
        /// <para>
        /// Gets or sets the one value.
        /// </para>
        /// <para></para>
        /// </summary>
        public static Link One { get; private set; }
        /// <summary>
        /// <para>
        /// Gets or sets the zero value.
        /// </para>
        /// <para></para>
        /// </summary>
        public static Link Zero { get; private set; }

        /// <summary>
        /// <para>
        /// Gets or sets the sum value.
        /// </para>
        /// <para></para>
        /// </summary>
        public static Link Sum { get; private set; }
        /// <summary>
        /// <para>
        /// Gets or sets the character value.
        /// </para>
        /// <para></para>
        /// </summary>
        public static Link Character { get; private set; }
        /// <summary>
        /// <para>
        /// Gets or sets the string value.
        /// </para>
        /// <para></para>
        /// </summary>
        public static Link String { get; private set; }
        /// <summary>
        /// <para>
        /// Gets or sets the name value.
        /// </para>
        /// <para></para>
        /// </summary>
        public static Link Name { get; private set; }

        /// <summary>
        /// <para>
        /// Gets or sets the set value.
        /// </para>
        /// <para></para>
        /// </summary>
        public static Link Set { get; private set; }
        /// <summary>
        /// <para>
        /// Gets or sets the group value.
        /// </para>
        /// <para></para>
        /// </summary>
        public static Link Group { get; private set; }

        /// <summary>
        /// <para>
        /// Gets or sets the parsed from value.
        /// </para>
        /// <para></para>
        /// </summary>
        public static Link ParsedFrom { get; private set; }
        /// <summary>
        /// <para>
        /// Gets or sets the that is value.
        /// </para>
```

```
        /// <para></para>
        /// </summary>
        public static Link ThatIs { get; private set; }
        /// <summary>
        /// <para>
        /// Gets or sets the that is before value.
        /// </para>
        /// <para></para>
        /// </summary>
        public static Link ThatIsBefore { get; private set; }
        /// <summary>
        /// <para>
        /// Gets or sets the that is between value.
        /// </para>
        /// <para></para>
        /// </summary>
        public static Link ThatIsBetween { get; private set; }
        /// <summary>
        /// <para>
        /// Gets or sets the that is after value.
        /// </para>
        /// <para></para>
        /// </summary>
        public static Link ThatIsAfter { get; private set; }
        /// <summary>
        /// <para>
        /// Gets or sets the that is represented by value.
        /// </para>
        /// <para></para>
        /// </summary>
        public static Link ThatIsRepresentedBy { get; private set; }
        /// <summary>
        /// <para>
        /// Gets or sets the that has value.
        /// </para>
        /// <para></para>
        /// </summary>
        public static Link ThatHas { get; private set; }

        /// <summary>
        /// <para>
        /// Gets or sets the text value.
        /// </para>
        /// <para></para>
        /// </summary>
        public static Link Text { get; private set; }
        /// <summary>
        /// <para>
        /// Gets or sets the path value.
        /// </para>
        /// <para></para>
        /// </summary>
        public static Link Path { get; private set; }
        /// <summary>
        /// <para>
        /// Gets or sets the content value.
        /// </para>
        /// <para></para>
        /// </summary>
        public static Link Content { get; private set; }
        /// <summary>
        /// <para>
        /// Gets or sets the empty content value.
        /// </para>
        /// <para></para>
        /// </summary>
        public static Link EmptyContent { get; private set; }
        /// <summary>
        /// <para>
        /// Gets or sets the empty value.
        /// </para>
        /// <para></para>
        /// </summary>
        public static Link Empty { get; private set; }
        /// <summary>
        /// <para>
        /// Gets or sets the alphabet value.
        /// </para>
```

```
        /// <para></para>
        /// </summary>
        public static Link Alphabet { get; private set; }
        /// <summary>
        /// <para>
        /// Gets or sets the letter value.
        /// </para>
        /// <para></para>
        /// </summary>
        public static Link Letter { get; private set; }
        /// <summary>
        /// <para>
        /// Gets or sets the case value.
        /// </para>
        /// <para></para>
        /// </summary>
        public static Link Case { get; private set; }
        /// <summary>
        /// <para>
        /// Gets or sets the upper value.
        /// </para>
        /// <para></para>
        /// </summary>
        public static Link Upper { get; private set; }
        /// <summary>
        /// <para>
        /// Gets or sets the upper case value.
        /// </para>
        /// <para></para>
        /// </summary>
        public static Link UpperCase { get; private set; }
        /// <summary>
        /// <para>
        /// Gets or sets the lower value.
        /// </para>
        /// <para></para>
        /// </summary>
        public static Link Lower { get; private set; }
        /// <summary>
        /// <para>
        /// Gets or sets the lower case value.
        /// </para>
        /// <para></para>
        /// </summary>
        public static Link LowerCase { get; private set; }
        /// <summary>
        /// <para>
        /// Gets or sets the code value.
        /// </para>
        /// <para></para>
        /// </summary>
        public static Link Code { get; private set; }

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="Net"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        static Net() => Create();

        /// <summary>
        /// <para>
        /// Creates the thing.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        public static Link CreateThing() => Link.Create(Link.Itself, IsA, Thing);

        /// <summary>
        /// <para>
        /// Creates the mapped thing using the specified mapping.
        /// </para>
        /// <para></para>
```

```
595         /// </summary>
596         /// <param name="mapping">
597         /// <para>The mapping.</para>
598         /// <para></para>
599         /// </param>
600         /// <returns>
601         /// <para>The link</para>
602         /// <para></para>
603         /// </returns>
604         public static Link CreateMappedThing(object mapping) => Link.CreateMapped(Link.Itself,
            ↪ IsA, Thing, mapping);
605
606         /// <summary>
607         /// <para>
608         /// Creates the link.
609         /// </para>
610         /// <para></para>
611         /// </summary>
612         /// <returns>
613         /// <para>The link</para>
614         /// <para></para>
615         /// </returns>
616         public static Link CreateLink() => Link.Create(Link.Itself, IsA, Link);
617
618         /// <summary>
619         /// <para>
620         /// Creates the mapped link using the specified mapping.
621         /// </para>
622         /// <para></para>
623         /// </summary>
624         /// <param name="mapping">
625         /// <para>The mapping.</para>
626         /// <para></para>
627         /// </param>
628         /// <returns>
629         /// <para>The link</para>
630         /// <para></para>
631         /// </returns>
632         public static Link CreateMappedLink(object mapping) => Link.CreateMapped(Link.Itself,
            ↪ IsA, Link, mapping);
633
634         /// <summary>
635         /// <para>
636         /// Creates the set.
637         /// </para>
638         /// <para></para>
639         /// </summary>
640         /// <returns>
641         /// <para>The link</para>
642         /// <para></para>
643         /// </returns>
644         public static Link CreateSet() => Link.Create(Link.Itself, IsA, Set);
645
646         /// <summary>
647         /// <para>
648         /// Creates.
649         /// </para>
650         /// <para></para>
651         /// </summary>
652         private static void Create()
653         {
654             #region Core
655
656             IsA = Link.GetMappedOrDefault(NetMapping.IsA);
657             IsNotA = Link.GetMappedOrDefault(NetMapping.IsNotA);
658             Link = Link.GetMappedOrDefault(NetMapping.Link);
659             Thing = Link.GetMappedOrDefault(NetMapping.Thing);
660
661             if (IsA == null || IsNotA == null || Link == null || Thing == null)
662             {
663                 // Наивная инициализация (Не является корректным объяснением).
664                 IsA = Link.CreateMapped(Link.Itself, Link.Itself, Link.Itself, NetMapping.IsA);
                    ↪ // Стоит переделать в "[x] is a member|instance|element of the class [y]"
665                 IsNotA = Link.CreateMapped(Link.Itself, Link.Itself, IsA, NetMapping.IsNotA);
666                 Link = Link.CreateMapped(Link.Itself, IsA, Link.Itself, NetMapping.Link);
667                 Thing = Link.CreateMapped(Link.Itself, IsNotA, Link, NetMapping.Thing);
668
```

```
669                 IsA = Link.Update(IsA, IsA, IsA, Link); // Исключение, позволяющие завершить
                    ↪  систему
670             }
671
672             #endregion
673
674             Of = CreateMappedLink(NetMapping.Of);
675             And = CreateMappedLink(NetMapping.And);
676             ThatConsistsOf = CreateMappedLink(NetMapping.ThatConsistsOf);
677             Has = CreateMappedLink(NetMapping.Has);
678             Contains = CreateMappedLink(NetMapping.Contains);
679             ContainedBy = CreateMappedLink(NetMapping.ContainedBy);
680
681             One = CreateMappedThing(NetMapping.One);
682             Zero = CreateMappedThing(NetMapping.Zero);
683
684             Sum = CreateMappedThing(NetMapping.Sum);
685             Character = CreateMappedThing(NetMapping.Character);
686             String = CreateMappedThing(NetMapping.String);
687             Name = Link.CreateMapped(Link.Itself, IsA, String, NetMapping.Name);
688
689             Set = CreateMappedThing(NetMapping.Set);
690             Group = CreateMappedThing(NetMapping.Group);
691
692             ParsedFrom = CreateMappedLink(NetMapping.ParsedFrom);
693             ThatIs = CreateMappedLink(NetMapping.ThatIs);
694             ThatIsBefore = CreateMappedLink(NetMapping.ThatIsBefore);
695             ThatIsAfter = CreateMappedLink(NetMapping.ThatIsAfter);
696             ThatIsBetween = CreateMappedLink(NetMapping.ThatIsBetween);
697             ThatIsRepresentedBy = CreateMappedLink(NetMapping.ThatIsRepresentedBy);
698             ThatHas = CreateMappedLink(NetMapping.ThatHas);
699
700             Text = CreateMappedThing(NetMapping.Text);
701             Path = CreateMappedThing(NetMapping.Path);
702             Content = CreateMappedThing(NetMapping.Content);
703             Empty = CreateMappedThing(NetMapping.Empty);
704             EmptyContent = Link.CreateMapped(Content, ThatIs, Empty, NetMapping.EmptyContent);
705             Alphabet = CreateMappedThing(NetMapping.Alphabet);
706             Letter = Link.CreateMapped(Link.Itself, IsA, Character, NetMapping.Letter);
707             Case = CreateMappedThing(NetMapping.Case);
708             Upper = CreateMappedThing(NetMapping.Upper);
709             UpperCase = Link.CreateMapped(Case, ThatIs, Upper, NetMapping.UpperCase);
710             Lower = CreateMappedThing(NetMapping.Lower);
711             LowerCase = Link.CreateMapped(Case, ThatIs, Lower, NetMapping.LowerCase);
712             Code = CreateMappedThing(NetMapping.Code);
713
714             SetNames();
715         }
716
717         /// <summary>
718         /// <para>
719         /// Recreates.
720         /// </para>
721         /// <para></para>
722         /// </summary>
723         public static void Recreate()
724         {
725             ThreadHelpers.InvokeWithExtendedMaxStackSize(() => Link.Delete(IsA));
726             CharacterHelpers.Recreate();
727             Create();
728         }
729
730         /// <summary>
731         /// <para>
732         /// Sets the names.
733         /// </para>
734         /// <para></para>
735         /// </summary>
736         private static void SetNames()
737         {
738             Thing.SetName("thing");
739             Link.SetName("link");
740             IsA.SetName("is a");
741             IsNotA.SetName("is not a");
742
743             Of.SetName("of");
744             And.SetName("and");
745             ThatConsistsOf.SetName("that consists of");
746             Has.SetName("has");
```

```
            Contains.SetName("contains");
            ContainedBy.SetName("contained by");

            One.SetName("one");
            Zero.SetName("zero");

            Character.SetName("character");
            Sum.SetName("sum");
            String.SetName("string");
            Name.SetName("name");

            Set.SetName("set");
            Group.SetName("group");

            ParsedFrom.SetName("parsed from");
            ThatIs.SetName("that is");
            ThatIsBefore.SetName("that is before");
            ThatIsAfter.SetName("that is after");
            ThatIsBetween.SetName("that is between");
            ThatIsRepresentedBy.SetName("that is represented by");
            ThatHas.SetName("that has");

            Text.SetName("text");
            Path.SetName("path");
            Content.SetName("content");
            Empty.SetName("empty");
            EmptyContent.SetName("empty content");
            Alphabet.SetName("alphabet");
            Letter.SetName("letter");
            Case.SetName("case");
            Upper.SetName("upper");
            Lower.SetName("lower");
            Code.SetName("code");
        }
    }
}
```

1.9 ./csharp/Platform.Data.Triplets/NumberHelpers.cs

```
using System;
using System.Collections.Generic;
using System.Globalization;
using Platform.Numbers;

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

namespace Platform.Data.Triplets
{
    /// <summary>
    /// <para>
    /// Represents the number helpers.
    /// </para>
    /// <para></para>
    /// </summary>
    public static class NumberHelpers
    {
        /// <summary>
        /// <para>
        /// Gets or sets the numbers to links value.
        /// </para>
        /// <para></para>
        /// </summary>
        public static Link[] NumbersToLinks { get; private set; }
        /// <summary>
        /// <para>
        /// Gets or sets the links to numbers value.
        /// </para>
        /// <para></para>
        /// </summary>
        public static Dictionary<Link, long> LinksToNumbers { get; private set; }

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="NumberHelpers"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        static NumberHelpers() => Create();

        /// <summary>
```

```
        /// <para>
        /// Creates.
        /// </para>
        /// <para></para>
        /// </summary>
        private static void Create()
        {
            NumbersToLinks = new Link[64];
            LinksToNumbers = new Dictionary<Link, long>();
            NumbersToLinks[0] = Net.One;
            LinksToNumbers[Net.One] = 1;
        }

        /// <summary>
        /// <para>
        /// Recreates.
        /// </para>
        /// <para></para>
        /// </summary>
        public static void Recreate() => Create();

        /// <summary>
        /// <para>
        /// Creates the power of 2 using the specified power of 2.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="powerOf2">
        /// <para>The power of.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The result.</para>
        /// <para></para>
        /// </returns>
        private static Link FromPowerOf2(long powerOf2)
        {
            var result = NumbersToLinks[powerOf2];
            if (result == null)
            {
                var previousPowerOf2Link = NumbersToLinks[powerOf2 - 1];
                if (previousPowerOf2Link == null)
                {
                    previousPowerOf2Link = NumbersToLinks[0];
                    for (var i = 1; i < powerOf2; i++)
                    {
                        if (NumbersToLinks[i] == null)
                        {
                            var numberLink = Link.Create(Net.Sum, Net.Of, previousPowerOf2Link &
                            ↪  previousPowerOf2Link);
                            var num = (long)System.Math.Pow(2, i);
                            NumbersToLinks[i] = numberLink;
                            LinksToNumbers[numberLink] = num;
                            numberLink.SetName(num.ToString(CultureInfo.InvariantCulture));
                        }
                        previousPowerOf2Link = NumbersToLinks[i];
                    }
                }
                result = Link.Create(Net.Sum, Net.Of, previousPowerOf2Link &
                ↪  previousPowerOf2Link);
                var number = (long)System.Math.Pow(2, powerOf2);
                NumbersToLinks[powerOf2] = result;
                LinksToNumbers[result] = number;
                result.SetName(number.ToString(CultureInfo.InvariantCulture));
            }
            return result;
        }

        /// <summary>
        /// <para>
        /// Creates the number using the specified number.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="number">
        /// <para>The number.</para>
        /// <para></para>
        /// </param>
```

```
        /// <exception cref="NotSupportedException">
        /// <para>Negative numbers are not supported yet.</para>
        /// <para></para>
        /// </exception>
        /// <returns>
        /// <para>The sum.</para>
        /// <para></para>
        /// </returns>
        public static Link FromNumber(long number)
        {
            if (number == 0)
            {
                return Net.Zero;
            }
            if (number == 1)
            {
                return Net.One;
            }
            var links = new Link[Bit.Count(number)];
            if (number >= 0)
            {
                for (long key = 1, powerOf2 = 0, i = 0; key <= number; key *= 2, powerOf2++)
                {
                    if ((number & key) == key)
                    {
                        links[i] = FromPowerOf2(powerOf2);
                        i++;
                    }
                }
            }
            else
            {
                throw new NotSupportedException("Negative numbers are not supported yet.");
            }
            var sum = Link.Create(Net.Sum, Net.Of, LinkConverter.FromList(links));
            return sum;
        }

        /// <summary>
        /// <para>
        /// Returns the number using the specified link.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="link">
        /// <para>The link.</para>
        /// <para></para>
        /// </param>
        /// <exception cref="ArgumentOutOfRangeException">
        /// <para>Specified link is not a number.</para>
        /// <para></para>
        /// </exception>
        /// <returns>
        /// <para>The long</para>
        /// <para></para>
        /// </returns>
        public static long ToNumber(Link link)
        {
            if (link == Net.Zero)
            {
                return 0;
            }
            if (link == Net.One)
            {
                return 1;
            }
            if (link.IsSum())
            {
                var numberParts = LinkConverter.ToList(link.Target);
                long number = 0;
                for (var i = 0; i < numberParts.Count; i++)
                {
                    GoDownAndTakeIt(numberParts[i], out long numberPart);
                    number += numberPart;
                }
                return number;
            }
            throw new ArgumentOutOfRangeException(nameof(link), "Specified link is not a
            ↪  number.");
```

```
        }

        /// <summary>
        /// <para>
        /// Goes the down and take it using the specified link.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="link">
        /// <para>The link.</para>
        /// <para></para>
        /// </param>
        /// <param name="number">
        /// <para>The number.</para>
        /// <para></para>
        /// </param>
        private static void GoDownAndTakeIt(Link link, out long number)
        {
            if (!LinksToNumbers.TryGetValue(link, out number))
            {
                var previousNumberLink = link.Target.Source;
                GoDownAndTakeIt(previousNumberLink, out number);
                var previousNumberIndex = (int)System.Math.Log(number, 2);
                var newNumberIndex = previousNumberIndex + 1;
                var newNumberLink = Link.Create(Net.Sum, Net.Of, previousNumberLink &
                ↪  previousNumberLink);
                number += number;
                NumbersToLinks[newNumberIndex] = newNumberLink;
                LinksToNumbers[newNumberLink] = number;
            }
        }
    }
}
```

1.10 ./csharp/Platform.Data.Triplets/Sequences/CompressionExperiments.cs

```
using System;
using System.Collections.Generic;

namespace Platform.Data.Triplets.Sequences
{
    /// <summary>
    /// <para>
    /// Represents the compression experiments.
    /// </para>
    /// <para></para>
    /// </summary>
    internal static class CompressionExperiments
    {
        /// <summary>
        /// <para>
        /// Rights the join using the specified subject.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="subject">
        /// <para>The subject.</para>
        /// <para></para>
        /// </param>
        /// <param name="@object">
        /// <para>The object.</para>
        /// <para></para>
        /// </param>
        public static void RightJoin(ref Link subject, Link @object)
        {
            if (subject.Linker == Net.And && subject.ReferersBySourceCount == 0 &&
            ↪  subject.ReferersByTargetCount == 0)
            {
                var subJoint = Link.Search(subject.Target, Net.And, @object);
                if (subJoint != null && subJoint != subject)
                {
                    Link.Update(ref subject, subject.Source, Net.And, subJoint);
                    return;
                }
            }
            subject = Link.Create(subject, Net.And, @object);
        }

        //public static Link RightJoinUnsafe(Link subject, Link @object)
```

```
43          //{
44          //    if (subject.Linker == Net.And && subject.ReferersBySourceCount == 0 &&
            ↪  subject.ReferersByTargetCount == 0)
45          //    {
46          //        Link subJoint = Link.Search(subject.Target, Net.And, @object);
47          //        if (subJoint != null && subJoint != subject)
48          //        {
49          //            Link.Update(ref subject, subject.Source, Net.And, subJoint);
50          //            return subject;
51          //        }
52          //    }
53          //    return Link.Create(subject, Net.And, @object);
54          //}
55
56          ////public static void LeftJoin(ref Link subject, Link @object)
57          ////{
58          ////    if (subject.Linker == Net.And && subject.ReferersBySourceCount == 0 &&
            ↪  subject.ReferersByTargetCount == 0)
59          ////    {
60          ////        Link subJoint = Link.Search(@object, Net.And, subject.Source);
61          ////        if (subJoint != null && subJoint != subject)
62          ////        {
63          ////            Link.Update(ref subject, subJoint, Net.And, subject.Target);
64          ////            return;
65          ////        }
66          ////    }
67          ////    subject = Link.Create(@object, Net.And, subject);
68          ////}
69
70          /// <summary>
71          /// <para>
72          /// Lefts the join using the specified subject.
73          /// </para>
74          /// <para></para>
75          /// </summary>
76          /// <param name="subject">
77          /// <para>The subject.</para>
78          /// <para></para>
79          /// </param>
80          /// <param name="@object">
81          /// <para>The object.</para>
82          /// <para></para>
83          /// </param>
84          public static void LeftJoin(ref Link subject, Link @object)
85          {
86              if (subject.Linker == Net.And && subject.ReferersBySourceCount == 0 &&
                ↪  subject.ReferersByTargetCount == 0)
87              {
88                  var subJoint = Link.Search(@object, Net.And, subject.Source);
89                  if (subJoint != null && subJoint != subject)
90                  {
91                      Link.Update(ref subject, subJoint, Net.And, subject.Target);
92                      //var prev = Link.Search(@object, Net.And, subject);
93                      //if (prev != null)
94                      //{
95                      //    Link.Update(ref prev, subJoint, Net.And, subject.Target);
96                      //}
97                      return;
98                  }
99              }
100             subject = Link.Create(@object, Net.And, subject);
101         }
102
103         // Сначала сжатие налево, а затем направо (так эффективнее)
104         // Не приятный момент, что обе связи, и первая и вторая могут быть изменены в результате
            ↪  алгоритма.
105         //public static Link CombinedJoin(ref Link first, ref Link second)
106         //{
107         //    Link atomicConnection = Link.Search(first, Net.And, second);
108         //    if (atomicConnection != null)
109         //    {
110         //        return atomicConnection;
111         //    }
112         //    else
113         //    {
114         //        if (second.Linker == Net.And)
115         //        {
116         //            Link subJoint = Link.Search(first, Net.And, second.Source);
```

```
117         //            if (subJoint != null && subJoint != second)// && subJoint.TotalReferers >
            ↪  second.TotalReferers)
118         //            {
119         //                //if (first.Linker == Net.And)
120         //                //{
121         //                //    // TODO: ...
122         //                //}
123         //                if (second.TotalReferers > 0)
124         //                {
125         //                    // В данный момент это никак не влияет, из-за того что добавлено
            ↪  условие по требованию
126         //                    // использования атомарного соедининеия если оно есть
127         //
128         //                    // В целом же приоритет между обходным соединением и атомарным
            ↪  нужно определять по весу.
129         //                    // И если в сети обнаружено сразу два варианта прохода - простой и
            ↪  обходной - нужно перебрасывать
130         //                    // пути с меньшим весом на использование путей с большим весом.
            ↪  (Это и технически эффективнее и более оправдано
131         //                    // с точки зрения смысла).
132         //
133         //                    // Положительный эффект текущей реализации, что она быстро
            ↪  "успокаивается" набирает критическую массу
134         //                    // и перестаёт вести себя не предсказуемо
135         //
136         //                    // Неприятность учёта веса в том, что нужно обрабатывать большое
            ↪  количество комбинаций.
137         //                    // Но вероятно это оправдано.
138         //
139         //                    //var prev = Link.Search(first, Net.And, second);
140         //                    //if (prev != null && subJoint != prev) // && prev.TotalReferers <
            ↪  subJoint.TotalReferers)
141         //                    //{
142         //                    //    Link.Update(ref prev, subJoint, Net.And, second.Target);
143         //                    //    if (second.TotalReferers == 0)
144         //                    //    {
145         //                    //        Link.Delete(ref second);
146         //                    //    }
147         //                    //    return prev;
148         //                    //}
149         //                    //return Link.Create(subJoint, Net.And, second.Target);
150         //                }
151         //                else
152         //                {
153         //                    Link.Update(ref second, subJoint, Net.And, second.Target);
154         //                    return second;
155         //                }
156         //            }
157         //        }
158         //        if (first.Linker == Net.And)
159         //        {
160         //            Link subJoint = Link.Search(first.Target, Net.And, second);
161         //            if (subJoint != null && subJoint != first)// && subJoint.TotalReferers >
            ↪  first.TotalReferers)
162         //            {
163         //                if (first.TotalReferers > 0)
164         //                {
165         //                    //var prev = Link.Search(first, Net.And, second);
166         //                    //if (prev != null && subJoint != prev) // && prev.TotalReferers <
            ↪  subJoint.TotalReferers)
167         //                    //{
168         //                    //    Link.Update(ref prev, first.Source, Net.And, subJoint);
169         //                    //    if (first.TotalReferers == 0)
170         //                    //    {
171         //                    //        Link.Delete(ref first);
172         //                    //    }
173         //                    //    return prev;
174         //                    //}
175         //                    //return Link.Create(first.Source, Net.And, subJoint);
176         //                }
177         //                else
178         //                {
179         //                    Link.Update(ref first, first.Source, Net.And, subJoint);
180         //                    return first;
181         //                }
182         //            }
183         //        }
```

```
184             //        return Link.Create(first, Net.And, second);
185             //    }
186             //}
187
188             /// <summary>
189             /// <para>
190             /// The compressions count.
191             /// </para>
192             /// <para></para>
193             /// </summary>
194             public static int CompressionsCount;
195
196             /// <summary>
197             /// <para>
198             /// Combineds the join using the specified first.
199             /// </para>
200             /// <para></para>
201             /// </summary>
202             /// <param name="first">
203             /// <para>The first.</para>
204             /// <para></para>
205             /// </param>
206             /// <param name="second">
207             /// <para>The second.</para>
208             /// <para></para>
209             /// </param>
210             /// <returns>
211             /// <para>The direct connection.</para>
212             /// <para></para>
213             /// </returns>
214             public static Link CombinedJoin(ref Link first, ref Link second)
215             {
216                 // Перестроение работает хорошо только когда одна из связей является парой и
                    ↪  аккумулятором одновременно
217                 // Когда обе связи - пары - нужно использовать другой алгоритм, иначе сжатие будет
                    ↪  отсутствовать.
218                 //if ((first.Linker == Net.And && second.Linker != Net.And)
219                 // || (second.Linker == Net.And && first.Linker != Net.And))
220                 //{
221                 //    Link connection = TryReconstructConnection(first, second);
222                 //    if (connection != null)
223                 //    {
224                 //        CompressionsCount++;
225                 //        return connection;
226                 //    }
227                 //}
228                 //return first & second;
229                 //long totalDoublets = Net.And.ReferersByLinkerCount;
230                 if (first == null || second == null)
231                 {
232                 }
233                 var directConnection = Link.Search(first, Net.And, second);
234                 if (directConnection == null)
235                 {
236                     directConnection = TryReconstructConnection(first, second);
237                 }
238                 Link rightCrossConnection = null;
239                 if (second.Linker == Net.And)
240                 {
241                     var assumedRightCrossConnection = Link.Search(first, Net.And, second.Source);
242                     if (assumedRightCrossConnection != null && second != assumedRightCrossConnection)
243                     {
244                         rightCrossConnection = assumedRightCrossConnection;
245                     }
246                     else
247                     {
248                         rightCrossConnection = TryReconstructConnection(first, second.Source);
249                     }
250                 }
251                 Link leftCrossConnection = null;
252                 if (first.Linker == Net.And)
253                 {
254                     var assumedLeftCrossConnection = Link.Search(first.Target, Net.And, second);
255                     if (assumedLeftCrossConnection != null && first != assumedLeftCrossConnection)
256                     {
257                         leftCrossConnection = assumedLeftCrossConnection;
258                     }
259                     else
```

```
260                 {
261                     leftCrossConnection = TryReconstructConnection(first.Target, second);
262                 }
263             }
264             // Наверное имеет смысл только в "безвыходной" ситуации
265             //if (directConnection == null && rightCrossConnection == null &&
                ↪  leftCrossConnection == null)
266             //{
267             //    directConnection = TryReconstructConnection(first, second);
268             //    // Может давать более агрессивное сжатие, но теряется стабильность
269             //    //if (directConnection == null)
270             //    //{
271             //    //    //if (second.Linker == Net.And)
272             //    //    //{
273             //    //    //    Link assumedRightCrossConnection = TryReconstructConnection(first,
                ↪  second.Source);
274             //    //    //    if (assumedRightCrossConnection != null && second !=
                ↪  assumedRightCrossConnection)
275             //    //    //    {
276             //    //    //        rightCrossConnection = assumedRightCrossConnection;
277             //    //    //    }
278             //    //    //}
279             //    //    //if (rightCrossConnection == null)
280             //    //    //{
281             //    //    //if (first.Linker == Net.And)
282             //    //    //{
283             //    //    //    Link assumedLeftCrossConnection =
                ↪  TryReconstructConnection(first.Target, second);
284             //    //    //    if (assumedLeftCrossConnection != null && first !=
                ↪  assumedLeftCrossConnection)
285             //    //    //    {
286             //    //    //        leftCrossConnection = assumedLeftCrossConnection;
287             //    //    //    }
288             //    //    //}
289             //    //    //}
290             //    //}
291             //}
292             //Link middleCrossConnection = null;
293             //if (second.Linker == Net.And && first.Linker == Net.And)
294             //{
295             //    Link assumedMiddleCrossConnection = Link.Search(first.Target, Net.And,
                ↪  second.Source);
296             //    if (assumedMiddleCrossConnection != null && first !=
                ↪  assumedMiddleCrossConnection && second != assumedMiddleCrossConnection)
297             //    {
298             //        middleCrossConnection = assumedMiddleCrossConnection;
299             //    }
300             //}
301             //Link rightMiddleCrossConnectinon = null;
302             //if (middleCrossConnection != null)
303             //{
304             //}
305             if (directConnection != null
306             && (rightCrossConnection == null || directConnection.TotalReferers >=
                ↪  rightCrossConnection.TotalReferers)
307             && (leftCrossConnection == null || directConnection.TotalReferers >=
                ↪  leftCrossConnection.TotalReferers))
308             {
309                 if (rightCrossConnection != null)
310                 {
311                     var prev = Link.Search(rightCrossConnection, Net.And, second.Target);
312                     if (prev != null && directConnection != prev)
313                     {
314                         Link.Update(ref prev, first, Net.And, second);
315                     }
316                     if (rightCrossConnection.TotalReferers == 0)
317                     {
318                         Link.Delete(ref rightCrossConnection);
319                     }
320                 }
321                 if (leftCrossConnection != null)
322                 {
323                     var prev = Link.Search(first.Source, Net.And, leftCrossConnection);
324                     if (prev != null && directConnection != prev)
325                     {
326                         Link.Update(ref prev, first, Net.And, second);
327                     }
```

```
328                         if (leftCrossConnection.TotalReferers == 0)
329                         {
330                             Link.Delete(ref leftCrossConnection);
331                         }
332                     }
333                     TryReconstructConnection(first, second);
334                     return directConnection;
335                 }
336                 else if (rightCrossConnection != null
337                      && (directConnection == null || rightCrossConnection.TotalReferers >=
                         ↪  directConnection.TotalReferers)
338                      && (leftCrossConnection == null || rightCrossConnection.TotalReferers >=
                         ↪  leftCrossConnection.TotalReferers))
339                 {
340                     if (directConnection != null)
341                     {
342                         var prev = Link.Search(first, Net.And, second);
343                         if (prev != null && rightCrossConnection != prev)
344                         {
345                             Link.Update(ref prev, rightCrossConnection, Net.And, second.Target);
346                         }
347                     }
348                     if (leftCrossConnection != null)
349                     {
350                         var prev = Link.Search(first.Source, Net.And, leftCrossConnection);
351                         if (prev != null && rightCrossConnection != prev)
352                         {
353                             Link.Update(ref prev, rightCrossConnection, Net.And, second.Target);
354                         }
355                     }
356                     //TryReconstructConnection(first, second.Source);
357                     //TryReconstructConnection(rightCrossConnection, second.Target); // ухудшает
                         ↪  стабильность
358                     var resultConnection = rightCrossConnection & second.Target;
359                     //if (second.TotalReferers == 0)
360                     //    Link.Delete(ref second);
361                     return resultConnection;
362                 }
363                 else if (leftCrossConnection != null
364                      && (directConnection == null || leftCrossConnection.TotalReferers >=
                         ↪  directConnection.TotalReferers)
365                      && (rightCrossConnection == null || leftCrossConnection.TotalReferers >=
                         ↪  rightCrossConnection.TotalReferers))
366                 {
367                     if (directConnection != null)
368                     {
369                         var prev = Link.Search(first, Net.And, second);
370                         if (prev != null && leftCrossConnection != prev)
371                         {
372                             Link.Update(ref prev, first.Source, Net.And, leftCrossConnection);
373                         }
374                     }
375                     if (rightCrossConnection != null)
376                     {
377                         var prev = Link.Search(rightCrossConnection, Net.And, second.Target);
378                         if (prev != null && rightCrossConnection != prev)
379                         {
380                             Link.Update(ref prev, first.Source, Net.And, leftCrossConnection);
381                         }
382                     }
383                     //TryReconstructConnection(first.Target, second);
384                     //TryReconstructConnection(first.Source, leftCrossConnection); // ухудшает
                         ↪  стабильность
385                     var resultConnection = first.Source & leftCrossConnection;
386                     //if (first.TotalReferers == 0)
387                     //    Link.Delete(ref first);
388                     return resultConnection;
389                 }
390                 else
391                 {
392                     if (directConnection != null)
393                     {
394                         return directConnection;
395                     }
396                     if (rightCrossConnection != null)
397                     {
398                         return rightCrossConnection & second.Target;
399                     }
```

```
400                 if (leftCrossConnection != null)
401                 {
402                     return first.Source & leftCrossConnection;
403                 }
404             }
405             // Можно фиксировать по окончанию каждой из веток, какой эффект от неё происходит
                ↪  (на сколько уменьшается/увеличивается количество связей)
406             directConnection = first & second;
407             //long difference = Net.And.ReferersByLinkerCount - totalDoublets;
408             //if (difference != 1)
409             //{
410             //    Console.WriteLine(Net.And.ReferersByLinkerCount - totalDoublets);
411             //}
412             return directConnection;
413         }
414 
415         /// <summary>
416         /// <para>
417         /// Tries the reconstruct connection using the specified first.
418         /// </para>
419         /// <para></para>
420         /// </summary>
421         /// <param name="first">
422         /// <para>The first.</para>
423         /// <para></para>
424         /// </param>
425         /// <param name="second">
426         /// <para>The second.</para>
427         /// <para></para>
428         /// </param>
429         /// <exception cref="NotImplementedException">
430         /// <para></para>
431         /// <para></para>
432         /// </exception>
433         /// <returns>
434         /// <para>The direct connection.</para>
435         /// <para></para>
436         /// </returns>
437         private static Link TryReconstructConnection(Link first, Link second)
438         {
439             Link directConnection = null;
440             if (second.ReferersBySourceCount < first.ReferersBySourceCount)
441             {
442                 //  o_|      x_o ...
443                 // x_|        |___|
444                 //
445                 // <-
446                 second.WalkThroughReferersAsSource(couple =>
447                 {
448                     if (couple.Linker == Net.And && couple.ReferersByTargetCount == 1 &&
                        ↪  couple.ReferersBySourceCount == 0)
449                     {
450                         var neighbour = couple.FirstRefererByTarget;
451                         if (neighbour.Linker == Net.And && neighbour.Source == first)
452                         {
453                             if (directConnection == null)
454                             {
455                                 directConnection = first & second;
456                             }
457                             Link.Update(ref neighbour, directConnection, Net.And, couple.Target);
458                             //Link.Delete(ref couple); // Можно заменить удалением couple
459                         }
460                     }
461                     if (couple.Linker == Net.And)
462                     {
463                         var neighbour = couple.FirstRefererByTarget;
464                         if (neighbour.Linker == Net.And && neighbour.Source == first)
465                         {
466                             throw new NotImplementedException();
467                         }
468                     }
469                 });
470             }
471             else
472             {
473                 //  o_|     x_o ...
474                 // x_|       |___|
475                 //
```

```
476                     // ->
477                     first.WalkThroughReferersAsSource(couple =>
478                     {
479                         if (couple.Linker == Net.And)
480                         {
481                             var neighbour = couple.Target;
482                             if (neighbour.Linker == Net.And && neighbour.Source == second)
483                             {
484                                 if (neighbour.ReferersByTargetCount == 1 &&
                                    ↪  neighbour.ReferersBySourceCount == 0)
485                                 {
486                                     if (directConnection == null)
487                                     {
488                                         directConnection = first & second;
489                                     }
490                                     Link.Update(ref couple, directConnection, Net.And,
                                        ↪  neighbour.Target);
491                                     //Link.Delete(ref neighbour);
492                                 }
493                             }
494                         }
495                     });
496                 }
497 
498                 if (second.ReferersByTargetCount < first.ReferersByTargetCount)
499                 {
500                     // |_x      ... x_o
501                     //  |_o      |___|
502                     //
503                     //   <-
504                     second.WalkThroughReferersAsTarget(couple =>
505                     {
506                         if (couple.Linker == Net.And)
507                         {
508                             var neighbour = couple.Source;
509                             if (neighbour.Linker == Net.And && neighbour.Target == first)
510                             {
511                                 if (neighbour.ReferersByTargetCount == 0 &&
                                    ↪  neighbour.ReferersBySourceCount == 1)
512                                 {
513                                     if (directConnection == null)
514                                     {
515                                         directConnection = first & second;
516                                     }
517                                     Link.Update(ref couple, neighbour.Source, Net.And,
                                        ↪  directConnection);
518                                     //Link.Delete(ref neighbour);
519                                 }
520                             }
521                         }
522                     });
523                 }
524                 else
525                 {
526                     // |_x      ... x_o
527                     //  |_o      |___|
528                     //
529                     //   ->
530                     first.WalkThroughReferersAsTarget((couple) =>
531                     {
532                         if (couple.Linker == Net.And && couple.ReferersByTargetCount == 0 &&
                            ↪  couple.ReferersBySourceCount == 1)
533                         {
534                             var neighbour = couple.FirstRefererBySource;
535                             if (neighbour.Linker == Net.And && neighbour.Target == second)
536                             {
537                                 if (directConnection == null)
538                                 {
539                                     directConnection = first & second;
540                                 }
541                                 Link.Update(ref neighbour, couple.Source, Net.And, directConnection);
542                                 Link.Delete(ref couple);
543                             }
544                         }
545                     });
546                 }
547                 if (directConnection != null)
548                 {
```

```
            CompressionsCount++;
        }
        return directConnection;
    }

    ////public static Link CombinedJoin(Link left, Link right)
    ////{
    ////    Link rightSubJoint = Link.Search(left, Net.And, right.Source);
    ////    if (rightSubJoint != null && rightSubJoint != right)
    ////    {
    ////        long rightSubJointReferers = rightSubJoint.TotalReferers;
    ////        Link leftSubJoint = Link.Search(left.Target, Net.And, right);
    ////        if (leftSubJoint != null && leftSubJoint != left)
    ////        {
    ////            long leftSubJointReferers = leftSubJoint.TotalReferers;
    ////            if (leftSubJointReferers > rightSubJointReferers)
    ////            {
    ////                long leftReferers = left.TotalReferers;
    ////                if (leftReferers > 0)
    ////                {
    ////                    return Link.Create(left.Source, Net.And, leftSubJoint);
    ////                }
    ////                else
    ////                {
    ////                    Link.Update(ref left, left.Source, Net.And, leftSubJoint);
    ////                    return left;
    ////                }
    ////            }
    ////        }
    ////        long rightReferers = right.TotalReferers;
    ////        if (rightReferers > 0)
    ////        {
    ////            return Link.Create(rightSubJoint, Net.And, right.Target);
    ////        }
    ////        else
    ////        {
    ////            Link.Update(ref right, rightSubJoint, Net.And, right.Target);
    ////            return right;
    ////        }
    ////    }
    ////    return Link.Create(left, Net.And, right);
    ////}
    //public static Link CombinedJoin(Link left, Link right)
    //{
    //    long leftReferers = left.TotalReferers;
    //    Link leftSubJoint = Link.Search(left.Target, Net.And, right);
    //    if (leftSubJoint != null && leftSubJoint != left)
    //    {
    //        long leftSubJointReferers = leftSubJoint.TotalReferers;
    //    }
    //    long rightReferers = left.TotalReferers;
    //    Link rightSubJoint = Link.Search(left, Net.And, right.Source);
    //    long rightSubJointReferers = rightSubJoint != null ? rightSubJoint.TotalReferers :
    ↪  long.MinValue;
    //}
    //public static Link LeftJoinUnsafe(Link subject, Link @object)
    //{
    //    if (subject.Linker == Net.And && subject.ReferersBySourceCount == 0 &&
    ↪  subject.ReferersByTargetCount == 0)
    //    {
    //        Link subJoint = Link.Search(@object, Net.And, subject.Source);
    //        if (subJoint != null && subJoint != subject)
    //        {
    //            Link.Update(ref subject, subJoint, Net.And, subject.Target);
    //            return subject;
    //        }
    //    }
    //    return Link.Create(@object, Net.And, subject);
    //}

    /// <summary>
    /// <para>
    /// The chunk size.
    /// </para>
    /// <para></para>
    /// </summary>
    public static int ChunkSize = 2;

```

```
        //public static Link FromList(List<Link> links)
        //{
        //    Link element = links[0];
        //    for (int i = 1; i < links.Count; i += ChunkSize)
        //    {
        //        int j = (i + ChunkSize - 1);
        //        j = j < links.Count ? j : (links.Count - 1);
        //        Link subElement = links[j];
        //        while (--j >= i) LeftJoin(ref subElement, links[j]);
        //        RightJoin(ref element, subElement);
        //    }
        //    return element;
        //}

        //public static Link FromList(Link[] links)
        //{
        //    Link element = links[0];
        //    for (int i = 1; i < links.Length; i += ChunkSize)
        //    {
        //        int j = (i + ChunkSize - 1);
        //        j = j < links.Length ? j : (links.Length - 1);
        //        Link subElement = links[j];
        //        while (--j >= i) LeftJoin(ref subElement, links[j]);
        //        RightJoin(ref element, subElement);
        //    }
        //    return element;
        //}

        //public static Link FromList(IList<Link> links)
        //{
        //    Link element = links[0];
        //    for (int i = 1; i < links.Count; i += ChunkSize)
        //    {
        //        int j = (i + ChunkSize - 1);
        //        j = j < links.Count ? j : (links.Count - 1);
        //        Link subElement = links[j];
        //        while (--j >= i)
        //        {
        //            Link x = links[j];
        //            subElement = CombinedJoin(ref x, ref subElement);
        //        }
        //        element = CombinedJoin(ref element, ref subElement);
        //    }
        //    return element;
        //}
        //public static Link FromList(IList<Link> links)
        //{
        //    int i = 0;
        //    Link element = links[i++];
        //    if (links.Count % 2 == 0)
        //    {
        //        element = CombinedJoin(element, links[i++]);
        //    }
        //    for (; i < links.Count; i += 2)
        //    {
        //        Link doublet = CombinedJoin(links[i], links[i + 1]);
        //        element = CombinedJoin(ref element, ref doublet);
        //    }
        //    return element;
        //}

        // Заглушка, возможно опасная
        /// <summary>
        /// <para>
        /// Combineds the join using the specified element.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="element">
        /// <para>The element.</para>
        /// <para></para>
        /// </param>
        /// <param name="link">
        /// <para>The link.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
```

```
703         /// <para></para>
704         /// </returns>
705         private static Link CombinedJoin(Link element, Link link)
706         {
707             return CombinedJoin(ref element, ref link);
708         }
709 
710         //public static Link FromList(List<Link> links)
711         //{
712         //    int i = links.Count - 1;
713         //    Link element = links[i];
714         //    while (--i >= 0) element = LinkConverterOld.ConnectLinks2(links[i], element,
    ↪  links, ref i);
715         //    return element;
716         //}
717         //public static Link FromList(Link[] links)
718         //{
719         //    int i = links.Length - 1;
720         //    Link element = links[i];
721         //    while (--i >= 0) element = LinkConverterOld.ConnectLinks2(links[i], element,
    ↪  links, ref i);
722         //    return element;
723         //}
724         //public static Link FromList(List<Link> links)
725         //{
726         //    Link element = links[0];
727         //    for (int i = 1; i < links.Count; i += ChunkSize)
728         //    {
729         //        int j = (i + ChunkSize - 1);
730         //        j = j < links.Count ? j : (links.Count - 1);
731         //        Link subElement = links[j];
732         //        while (--j >= i) subElement = CombinedJoin(links[j], subElement);
733         //        element = CombinedJoin(element, subElement);
734         //    }
735         //    return element;
736         //}
737         //public static Link FromList(Link[] links)
738         //{
739         //    Link element = links[0];
740         //    for (int i = 1; i < links.Length; i += ChunkSize)
741         //    {
742         //        int j = (i + ChunkSize - 1);
743         //        j = j < links.Length ? j : (links.Length - 1);
744         //        Link subElement = links[j];
745         //        while (--j >= i) subElement = CombinedJoin(links[j], subElement);
746         //        element = CombinedJoin(element, subElement);
747         //    }
748         //    return element;
749         //}
750         //public static Link FromList(IList<Link> links)
751         //{
752         //    int leftBound = 0;
753         //    int rightBound = links.Count - 1;
754         //    if (leftBound == rightBound)
755         //    {
756         //        return links[0];
757         //    }
758         //    Link left = links[leftBound];
759         //    Link right = links[rightBound];
760         //    long leftReferers = left.ReferersBySourceCount + left.ReferersByTargetCount;
761         //    long rightReferers = right.ReferersBySourceCount + right.ReferersByTargetCount;
762         //    while (true)
763         //    {
764         //        //if (rightBound % 2 != leftBound % 2)
765         //        if (rightReferers >= leftReferers)
766         //        {
767         //            int nextRightBound = --rightBound;
768         //            if (nextRightBound == leftBound)
769         //            {
770         //                var x = CombinedJoin(ref left, ref right);
771         //                return x;
772         //            }
773         //            else
774         //            {
775         //                Link nextRight = links[nextRightBound];
776         //                right = CombinedJoin(ref nextRight, ref right);
777         //                rightReferers = right.ReferersBySourceCount +
    ↪  right.ReferersByTargetCount;
```

```
778         //            }
779         //        }
780         //        else
781         //        {
782         //            int nextLeftBound = ++leftBound;
783         //            if (nextLeftBound == rightBound)
784         //            {
785         //                return CombinedJoin(ref left, ref right);
786         //            }
787         //            else
788         //            {
789         //                Link nextLeft = links[nextLeftBound];
790         //                left = CombinedJoin(ref left, ref nextLeft);
791         //                leftReferers = left.ReferersBySourceCount + left.ReferersByTargetCount;
792         //            }
793         //        }
794         //    }
795         //}
796         //public static Link FromList(IList<Link> links)
797         //{
798         //    int i = links.Count - 1;
799         //    Link element = links[i];
800         //    while (--i >= 0)
801         //    {
802         //        LeftJoin(ref element, links[i]); // LeftJoin(ref element, links[i]);
803         //    }
804         //    return element;
805         //}
806
807         /// <summary>
808         /// <para>
809         /// Creates the list using the specified links.
810         /// </para>
811         /// <para></para>
812         /// </summary>
813         /// <param name="links">
814         /// <para>The links.</para>
815         /// <para></para>
816         /// </param>
817         /// <returns>
818         /// <para>The element.</para>
819         /// <para></para>
820         /// </returns>
821         public static Link FromList(List<Link> links)
822         {
823             var i = links.Count - 1;
824             var element = links[i];
825             while (--i >= 0)
826             {
827                 var x = links[i];
828                 element = CombinedJoin(ref x, ref element); // LeftJoin(ref element, links[i]);
829             }
830             return element;
831         }
832
833         /// <summary>
834         /// <para>
835         /// Creates the list using the specified links.
836         /// </para>
837         /// <para></para>
838         /// </summary>
839         /// <param name="links">
840         /// <para>The links.</para>
841         /// <para></para>
842         /// </param>
843         /// <returns>
844         /// <para>The element.</para>
845         /// <para></para>
846         /// </returns>
847         public static Link FromList(Link[] links)
848         {
849             var i = links.Length - 1;
850             var element = links[i];
851             while (--i >= 0)
852             {
853                 element = CombinedJoin(ref links[i], ref element); // LeftJoin(ref element,
                    ↪  links[i]);
854             }
```

```
            return element;
        }
    }
}
```

1.11 ./csharp/Platform.Data.Triplets/Sequences/SequenceHelpers.cs

```
using System;
using System.Collections.Generic;
using System.Text;
using Platform.Data.Sequences;

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

namespace Platform.Data.Triplets.Sequences
{
    /// <remarks>
    /// TODO: Check that CollectMatchingSequences algorithm is working, if not throw it away.
    /// TODO: Think of the abstraction on Sequences that can be equally usefull for triple links, doublet links and so on.
    /// </remarks>
    public static class SequenceHelpers
    {
        /// <summary>
        /// <para>
        /// The max sequence format size.
        /// </para>
        /// <para></para>
        /// </summary>
        public static readonly int MaxSequenceFormatSize = 20;

        //public static void DeleteSequence(Link sequence)
        //{
        //}

        /// <summary>
        /// <para>
        /// Formats the sequence using the specified sequence.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="sequence">
        /// <para>The sequence.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The string</para>
        /// <para></para>
        /// </returns>
        public static string FormatSequence(Link sequence)
        {
            var visitedElements = 0;
            var sb = new StringBuilder();
            sb.Append('{');
            StopableSequenceWalker.WalkRight(sequence, x => x.Source, x => x.Target, x => x.Linker != Net.And, element =>
            {
                if (visitedElements > 0)
                {
                    sb.Append(',');
                }
                sb.Append(element.ToString());
                visitedElements++;
                if (visitedElements < MaxSequenceFormatSize)
                {
                    return true;
                }
                else
                {
                    sb.Append(", ...");
                    return false;
                }
            });
            sb.Append('}');
            return sb.ToString();
        }

        /// <summary>
        /// <para>
        /// Collects the matching sequences using the specified links.
```

```
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="links">
        /// <para>The links.</para>
        /// <para></para>
        /// </param>
        /// <exception cref="InvalidOperationException">
        /// <para>Подпоследовательности с одним элементом не поддерживаются.</para>
        /// <para></para>
        /// </exception>
        /// <returns>
        /// <para>The results.</para>
        /// <para></para>
        /// </returns>
        public static List<Link> CollectMatchingSequences(Link[] links)
        {
            if (links.Length == 1)
            {
                throw new InvalidOperationException("Подпоследовательности с одним элементом не
                ↪  поддерживаются.");
            }
            var leftBound = 0;
            var rightBound = links.Length - 1;
            var left = links[leftBound++];
            var right = links[rightBound--];
            var results = new List<Link>();
            CollectMatchingSequences(left, leftBound, links, right, rightBound, ref results);
            return results;
        }

        /// <summary>
        /// <para>
        /// Collects the matching sequences using the specified left link.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="leftLink">
        /// <para>The left link.</para>
        /// <para></para>
        /// </param>
        /// <param name="leftBound">
        /// <para>The left bound.</para>
        /// <para></para>
        /// </param>
        /// <param name="middleLinks">
        /// <para>The middle links.</para>
        /// <para></para>
        /// </param>
        /// <param name="rightLink">
        /// <para>The right link.</para>
        /// <para></para>
        /// </param>
        /// <param name="rightBound">
        /// <para>The right bound.</para>
        /// <para></para>
        /// </param>
        /// <param name="results">
        /// <para>The results.</para>
        /// <para></para>
        /// </param>
        private static void CollectMatchingSequences(Link leftLink, int leftBound, Link[]
        ↪  middleLinks, Link rightLink, int rightBound, ref List<Link> results)
        {
            var leftLinkTotalReferers = leftLink.ReferersBySourceCount +
            ↪  leftLink.ReferersByTargetCount;
            var rightLinkTotalReferers = rightLink.ReferersBySourceCount +
            ↪  rightLink.ReferersByTargetCount;
            if (leftLinkTotalReferers <= rightLinkTotalReferers)
            {
                var nextLeftLink = middleLinks[leftBound];
                var elements = GetRightElements(leftLink, nextLeftLink);
                if (leftBound <= rightBound)
                {
                    for (var i = elements.Length - 1; i >= 0; i--)
                    {
                        var element = elements[i];
                        if (element != null)
```

```
                        {
                            CollectMatchingSequences(element, leftBound + 1, middleLinks,
                            ↪  rightLink, rightBound, ref results);
                        }
                    }
                }
                else
                {
                    for (var i = elements.Length - 1; i >= 0; i--)
                    {
                        var element = elements[i];
                        if (element != null)
                        {
                            results.Add(element);
                        }
                    }
                }
            }
            else
            {
                var nextRightLink = middleLinks[rightBound];
                var elements = GetLeftElements(rightLink, nextRightLink);
                if (leftBound <= rightBound)
                {
                    for (var i = elements.Length - 1; i >= 0; i--)
                    {
                        var element = elements[i];
                        if (element != null)
                        {
                            CollectMatchingSequences(leftLink, leftBound, middleLinks,
                            ↪  elements[i], rightBound - 1, ref results);
                        }
                    }
                }
                else
                {
                    for (var i = elements.Length - 1; i >= 0; i--)
                    {
                        var element = elements[i];
                        if (element != null)
                        {
                            results.Add(element);
                        }
                    }
                }
            }
        }

        /// <summary>
        /// <para>
        /// Gets the right elements using the specified start link.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="startLink">
        /// <para>The start link.</para>
        /// <para></para>
        /// </param>
        /// <param name="rightLink">
        /// <para>The right link.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The result.</para>
        /// <para></para>
        /// </returns>
        public static Link[] GetRightElements(Link startLink, Link rightLink)
        {
            var result = new Link[4];
            TryStepRight(startLink, rightLink, result, 0);
            startLink.WalkThroughReferersAsTarget(couple =>
                {
                    if (couple.Linker == Net.And)
                    {
                        if (TryStepRight(couple, rightLink, result, 2))
                        {
                            return Link.Stop;
                        }
```

```
                }
                return Link.Continue;
            });
        return result;
    }

    /// <summary>
    /// <para>
    /// Determines whether try step right.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <param name="startLink">
    /// <para>The start link.</para>
    /// <para></para>
    /// </param>
    /// <param name="rightLink">
    /// <para>The right link.</para>
    /// <para></para>
    /// </param>
    /// <param name="result">
    /// <para>The result.</para>
    /// <para></para>
    /// </param>
    /// <param name="offset">
    /// <para>The offset.</para>
    /// <para></para>
    /// </param>
    /// <returns>
    /// <para>The bool</para>
    /// <para></para>
    /// </returns>
    public static bool TryStepRight(Link startLink, Link rightLink, Link[] result, int
    ↪ offset)
    {
        var added = 0;
        startLink.WalkThroughReferersAsSource(couple =>
            {
                if (couple.Linker == Net.And)
                {
                    var coupleTarget = couple.Target;
                    if (coupleTarget == rightLink)
                    {
                        result[offset] = couple;
                        if (++added == 2)
                        {
                            return Link.Stop;
                        }
                    }
                    else if (coupleTarget.Linker == Net.And && coupleTarget.Source ==
                    ↪ rightLink)
                    {
                        result[offset + 1] = couple;
                        if (++added == 2)
                        {
                            return Link.Stop;
                        }
                    }
                }
                return Link.Continue;
            });
        return added > 0;
    }

    /// <summary>
    /// <para>
    /// Gets the left elements using the specified start link.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <param name="startLink">
    /// <para>The start link.</para>
    /// <para></para>
    /// </param>
    /// <param name="leftLink">
    /// <para>The left link.</para>
    /// <para></para>
    /// </param>
```

```
        /// <returns>
        /// <para>The result.</para>
        /// <para></para>
        /// </returns>
        public static Link[] GetLeftElements(Link startLink, Link leftLink)
        {
            var result = new Link[4];
            TryStepLeft(startLink, leftLink, result, 0);
            startLink.WalkThroughReferersAsSource(couple =>
                {
                    if (couple.Linker == Net.And)
                    {
                        if (TryStepLeft(couple, leftLink, result, 2))
                        {
                            return Link.Stop;
                        }
                    }
                    return Link.Continue;
                });
            return result;
        }

        /// <summary>
        /// <para>
        /// Determines whether try step left.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="startLink">
        /// <para>The start link.</para>
        /// <para></para>
        /// </param>
        /// <param name="leftLink">
        /// <para>The left link.</para>
        /// <para></para>
        /// </param>
        /// <param name="result">
        /// <para>The result.</para>
        /// <para></para>
        /// </param>
        /// <param name="offset">
        /// <para>The offset.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        public static bool TryStepLeft(Link startLink, Link leftLink, Link[] result, int offset)
        {
            var added = 0;
            startLink.WalkThroughReferersAsTarget(couple =>
                {
                    if (couple.Linker == Net.And)
                    {
                        var coupleSource = couple.Source;
                        if (coupleSource == leftLink)
                        {
                            result[offset] = couple;
                            if (++added == 2)
                            {
                                return Link.Stop;
                            }
                        }
                        else if (coupleSource.Linker == Net.And && coupleSource.Target ==
                        ↪  leftLink)
                        {
                            result[offset + 1] = couple;
                            if (++added == 2)
                            {
                                return Link.Stop;
                            }
                        }
                    }
                    return Link.Continue;
                });
            return added > 0;
        }
```

```
    }
}
```

1.12 ./csharp/Platform.Data.Triplets.Tests/LinkTests.cs

```
using System.IO;
using Xunit;
using Platform.Random;
using Platform.Ranges;

namespace Platform.Data.Triplets.Tests
{
    /// <summary>
    /// <para>
    /// Represents the link tests.
    /// </para>
    /// <para></para>
    /// </summary>
    public static class LinkTests
    {
        /// <summary>
        /// <para>
        /// The lock.
        /// </para>
        /// <para></para>
        /// </summary>
        public static object Lock = new object(); //-V3090

        /// <summary>
        /// <para>
        /// The thing visitor counter.
        /// </para>
        /// <para></para>
        /// </summary>
        private static ulong _thingVisitorCounter;
        /// <summary>
        /// <para>
        /// The is visitor counter.
        /// </para>
        /// <para></para>
        /// </summary>
        private static ulong _isAVisitorCounter;
        /// <summary>
        /// <para>
        /// The link visitor counter.
        /// </para>
        /// <para></para>
        /// </summary>
        private static ulong _linkVisitorCounter;

        /// <summary>
        /// <para>
        /// Things the visitor using the specified link index.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="linkIndex">
        /// <para>The link index.</para>
        /// <para></para>
        /// </param>
        static void ThingVisitor(Link linkIndex)
        {
            _thingVisitorCounter += linkIndex;
        }

        /// <summary>
        /// <para>
        /// Ises the a visitor using the specified link index.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="linkIndex">
        /// <para>The link index.</para>
        /// <para></para>
        /// </param>
        static void IsAVisitor(Link linkIndex)
        {
            _isAVisitorCounter += linkIndex;
        }

```

```
        /// <summary>
        /// <para>
        /// Links the visitor using the specified link index.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="linkIndex">
        /// <para>The link index.</para>
        /// <para></para>
        /// </param>
        static void LinkVisitor(Link linkIndex)
        {
            _linkVisitorCounter += linkIndex;
        }

        /// <summary>
        /// <para>
        /// Tests that create delete link test.
        /// </para>
        /// <para></para>
        /// </summary>
        [Fact]
        public static void CreateDeleteLinkTest()
        {
            lock (Lock)
            {
                string filename = "db.links";

                File.Delete(filename);

                Link.StartMemoryManager(filename);

                Link link1 = Link.Create(Link.Itself, Link.Itself, Link.Itself);

                Link.Delete(link1);

                Link.StopMemoryManager();

                File.Delete(filename);
            }
        }

        /// <summary>
        /// <para>
        /// Tests that deep create update delete link test.
        /// </para>
        /// <para></para>
        /// </summary>
        [Fact]
        public static void DeepCreateUpdateDeleteLinkTest()
        {
            lock (Lock)
            {
                string filename = "db.links";

                File.Delete(filename);

                Link.StartMemoryManager(filename);

                Link isA = Link.Create(Link.Itself, Link.Itself, Link.Itself);
                Link isNotA = Link.Create(Link.Itself, Link.Itself, isA);
                Link link = Link.Create(Link.Itself, isA, Link.Itself);
                Link thing = Link.Create(Link.Itself, isNotA, link);

                //Assert::IsTrue(GetLinksCount() == 4);

                Assert.Equal(isA, isA.Target);

                isA = Link.Update(isA, isA, isA, link); // Произведено замыкание

                Assert.Equal(link, isA.Target);

                Link.Delete(isA); // Одна эта операция удалит все 4 связи

                //Assert::IsTrue(GetLinksCount() == 0);

                Link.StopMemoryManager();

                File.Delete(filename);
```

```
155             }
156         }
157 
158         /// <summary>
159         /// <para>
160         /// Tests that link referers walk test.
161         /// </para>
162         /// <para></para>
163         /// </summary>
164         [Fact]
165         public static void LinkReferersWalkTest()
166         {
167             lock (Lock)
168             {
169                 string filename = "db.links";
170 
171                 File.Delete(filename);
172 
173                 Link.StartMemoryManager(filename);
174 
175                 Link isA = Link.Create(Link.Itself, Link.Itself, Link.Itself);
176                 Link isNotA = Link.Create(Link.Itself, Link.Itself, isA);
177                 Link link = Link.Create(Link.Itself, isA, Link.Itself);
178                 Link thing = Link.Create(Link.Itself, isNotA, link);
179                 isA = Link.Update(isA, isA, isA, link);
180 
181                 Assert.Equal(1, thing.ReferersBySourceCount);
182                 Assert.Equal(2, isA.ReferersByLinkerCount);
183                 Assert.Equal(3, link.ReferersByTargetCount);
184 
185                 _thingVisitorCounter = 0;
186                 _isAVisitorCounter = 0;
187                 _linkVisitorCounter = 0;
188 
189                 thing.WalkThroughReferersAsSource(ThingVisitor);
190                 isA.WalkThroughReferersAsLinker(IsAVisitor);
191                 link.WalkThroughReferersAsTarget(LinkVisitor);
192 
193                 Assert.Equal(4UL, _thingVisitorCounter);
194                 Assert.Equal(1UL + 3UL, _isAVisitorCounter);
195                 Assert.Equal(1UL + 3UL + 4UL, _linkVisitorCounter);
196 
197                 Link.StopMemoryManager();
198 
199                 File.Delete(filename);
200             }
201         }
202 
203         /// <summary>
204         /// <para>
205         /// Tests that multiple random creations and deletions test.
206         /// </para>
207         /// <para></para>
208         /// </summary>
209         [Fact]
210         public static void MultipleRandomCreationsAndDeletionsTest()
211         {
212             lock (Lock)
213             {
214                 string filename = "db.links";
215 
216                 File.Delete(filename);
217 
218                 Link.StartMemoryManager(filename);
219 
220                 TestMultipleRandomCreationsAndDeletions(2000);
221 
222                 Link.StopMemoryManager();
223 
224                 File.Delete(filename);
225             }
226         }
227 
228         /// <summary>
229         /// <para>
230         /// Tests the multiple random creations and deletions using the specified maximum
    ↪  operations per cycle.
231         /// </para>
232         /// <para></para>
```

```
233         /// </summary>
234         /// <param name="maximumOperationsPerCycle">
235         /// <para>The maximum operations per cycle.</para>
236         /// <para></para>
237         /// </param>
238         private static void TestMultipleRandomCreationsAndDeletions(int
            ↪ maximumOperationsPerCycle)
239         {
240             var and = Link.Create(Link.Itself, Link.Itself, Link.Itself);
241             //var comparer = Comparer<TLink>.Default;
242             for (var N = 1; N < maximumOperationsPerCycle; N++)
243             {
244                 var random = new System.Random(N);
245                 var linksCount = 1;
246                 for (var i = 0; i < N; i++)
247                 {
248                     var createPoint = random.NextBoolean();
249                     if (linksCount > 2 && createPoint)
250                     {
251                         var linksAddressRange = new Range<ulong>(1, (ulong)linksCount);
252                         Link source = random.NextUInt64(linksAddressRange);
253                         Link target = random.NextUInt64(linksAddressRange); //-V3086
254                         var resultLink = Link.Create(source, and, target);
255                         if (resultLink > linksCount)
256                         {
257                             linksCount++;
258                         }
259                     }
260                     else
261                     {
262                         Link.Create(Link.Itself, Link.Itself, Link.Itself);
263                         linksCount++;
264                     }
265                 }
266                 for (var i = 0; i < N; i++)
267                 {
268                     Link link = i + 2;
269                     if (link.Linker != null)
270                     {
271                         Link.Delete(link);
272                         linksCount--;
273                     }
274                 }
275             }
276         }
277     }
278 }
```

1.13 ./csharp/Platform.Data.Triplets.Tests/PersistentMemoryManagerTests.cs

```
1  using System.IO;
2  using Xunit;
3
4  namespace Platform.Data.Triplets.Tests
5  {
6      /// <summary>
7      /// <para>
8      /// Represents the persistent memory manager tests.
9      /// </para>
10     /// <para></para>
11     /// </summary>
12     public static class PersistentMemoryManagerTests
13     {
14         /// <summary>
15         /// <para>
16         /// Tests that file mapping test.
17         /// </para>
18         /// <para></para>
19         /// </summary>
20         [Fact]
21         public static void FileMappingTest()
22         {
23             lock (LinkTests.Lock)
24             {
25                 string filename = "db.links";
26
27                 File.Delete(filename);
28
29                 Link.StartMemoryManager(filename);
30
```

```
            Link.StopMemoryManager();

            File.Delete(filename);
        }
    }

    /// <summary>
    /// <para>
    /// Tests that allocate and free link test.
    /// </para>
    /// <para></para>
    /// </summary>
    [Fact]
    public static void AllocateAndFreeLinkTest()
    {
        lock (LinkTests.Lock)
        {
            string filename = "db.links";

            File.Delete(filename);

            Link.StartMemoryManager(filename);

            Link link = Link.Create(Link.Itself, Link.Itself, Link.Itself);

            Link.Delete(link);

            Link.StopMemoryManager();

            File.Delete(filename);
        }
    }

    /// <summary>
    /// <para>
    /// Tests that attach to unused link test.
    /// </para>
    /// <para></para>
    /// </summary>
    [Fact]
    public static void AttachToUnusedLinkTest()
    {
        lock (LinkTests.Lock)
        {
            string filename = "db.links";

            File.Delete(filename);

            Link.StartMemoryManager(filename);

            Link link1 = Link.Create(Link.Itself, Link.Itself, Link.Itself);
            Link link2 = Link.Create(Link.Itself, Link.Itself, Link.Itself);

            Link.Delete(link1); // Creates "hole" and forces "Attach" to be executed

            Link.StopMemoryManager();

            File.Delete(filename);
        }
    }

    /// <summary>
    /// <para>
    /// Tests that detach to unused link test.
    /// </para>
    /// <para></para>
    /// </summary>
    [Fact]
    public static void DetachToUnusedLinkTest()
    {
        lock (LinkTests.Lock)
        {
            string filename = "db.links";

            File.Delete(filename);

            Link.StartMemoryManager(filename);

            Link link1 = Link.Create(Link.Itself, Link.Itself, Link.Itself);
```

```
                Link link2 = Link.Create(Link.Itself, Link.Itself, Link.Itself);

                Link.Delete(link1); // Creates "hole" and forces "Attach" to be executed
                Link.Delete(link2); // Removes both links, all "Attached" links forced to be
                ↪  "Detached" here

                Link.StopMemoryManager();

                File.Delete(filename);
            }
        }

        /// <summary>
        /// <para>
        /// Tests that get set mapped link test.
        /// </para>
        /// <para></para>
        /// </summary>
        [Fact]
        public static void GetSetMappedLinkTest()
        {
            lock (LinkTests.Lock)
            {
                string filename = "db.links";

                File.Delete(filename);

                Link.StartMemoryManager(filename);

                var mapped = Link.GetMappedOrDefault(0);

                var mappingSet = Link.TrySetMapped(mapped, 0);

                Assert.True(mappingSet);

                Link.StopMemoryManager();

                File.Delete(filename);
            }
        }
    }
}
```

Index