

# LinksPlatform's Platform.Data.Triplets Class Library

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

```
1  using System;
2  using System.Collections.Generic;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Data.Triplets
7  {
8      // TODO: Split logic of Latin and Cyrillic alphabets into different files if possible
9      /// <summary>
10     /// <para>
11     /// Represents the character helpers.
12     /// </para>
13     /// <para></para>
14     /// </summary>
15     public static class CharacterHelpers
16     {
17         /// <summary>
18         /// <para>
19         /// The character mapping enum.
20         /// </para>
21         /// <para></para>
22         /// </summary>
23         public enum CharacterMapping : long
24         {
25             /// <summary>
26             /// <para>
27             /// The latin alphabet character mapping.
28             /// </para>
29             /// <para></para>
30             /// </summary>
31             LatinAlphabet = 100,
32             /// <summary>
33             /// <para>
34             /// The cyrillic alphabet character mapping.
35             /// </para>
36             /// <para></para>
37             /// </summary>
38             CyrillicAlphabet
39         }
40
41         /// <summary>
42         /// <para>
43         /// The first lower case latin letter.
44         /// </para>
45         /// <para></para>
46         /// </summary>
47         private const char FirstLowerCaseLatinLetter = 'a';
48         /// <summary>
49         /// <para>
50         /// The last lower case latin letter.
51         /// </para>
52         /// <para></para>
53         /// </summary>
54         private const char LastLowerCaseLatinLetter = 'z';
55         /// <summary>
56         /// <para>
57         /// The first upper case latin letter.
58         /// </para>
59         /// <para></para>
60         /// </summary>
61         private const char FirstUpperCaseLatinLetter = 'A';
62         /// <summary>
63         /// <para>
64         /// The last upper case latin letter.
65         /// </para>
66         /// <para></para>
67         /// </summary>
68         private const char LastUpperCaseLatinLetter = 'Z';
69         /// <summary>
70         /// <para>
71         /// The first lower case cyrillic letter.
72         /// </para>
73         /// <para></para>
74         /// </summary>
75         private const char FirstLowerCaseCyrillicLetter = 'a';
76         /// <summary>
77         /// <para>
```

```

78     /// The last lower case cyrillic letter.
79     /// </para>
80     /// <para></para>
81     /// </summary>
82     private const char LastLowerCaseCyrillicLetter = 'я';
83     /// <summary>
84     /// <para>
85     /// The first upper case cyrillic letter.
86     /// </para>
87     /// <para></para>
88     /// </summary>
89     private const char FirstUpperCaseCyrillicLetter = 'А';
90     /// <summary>
91     /// <para>
92     /// The last upper case cyrillic letter.
93     /// </para>
94     /// <para></para>
95     /// </summary>
96     private const char LastUpperCaseCyrillicLetter = 'Я';
97     /// <summary>
98     /// <para>
99     /// The yo lower case cyrillic letter.
100    /// </para>
101    /// <para></para>
102    /// </summary>
103    private const char YoLowerCaseCyrillicLetter = 'ё';
104    /// <summary>
105    /// <para>
106    /// The yo upper case cyrillic letter.
107    /// </para>
108    /// <para></para>
109    /// </summary>
110    private const char YoUpperCaseCyrillicLetter = 'Ё';
111
112    /// <summary>
113    /// <para>
114    /// The characters to links.
115    /// </para>
116    /// <para></para>
117    /// </summary>
118    private static Link[] _charactersToLinks;
119    /// <summary>
120    /// <para>
121    /// The links to characters.
122    /// </para>
123    /// <para></para>
124    /// </summary>
125    private static Dictionary<Link, char> _linksToCharacters;
126
127    /// <summary>
128    /// <para>
129    /// Initializes a new <see cref="CharacterHelpers"/> instance.
130    /// </para>
131    /// <para></para>
132    /// </summary>
133    static CharacterHelpers() => Create();
134
135    /// <summary>
136    /// <para>
137    /// Creates.
138    /// </para>
139    /// <para></para>
140    /// </summary>
141    private static void Create()
142    {
143        _charactersToLinks = new Link[char.MaxValue];
144        _linksToCharacters = new Dictionary<Link, char>();
145        // Create or restore characters
146        CreateLatinAlphabet();
147        CreateCyrillicAlphabet();
148        RegisterExistingCharacters();
149    }
150
151    /// <summary>
152    /// <para>
153    /// Registers the existing characters.
154    /// </para>
155    /// <para></para>

```

```

156 /// </summary>
157 private static void RegisterExistingCharacters() =>
    ↪ Net.Character.WalkThroughReferersAsSource(referer =>
    ↪ RegisterExistingCharacter(referer));
158
159 /// <summary>
160 /// <para>
161 /// Registers the existing character using the specified character.
162 /// </para>
163 /// <para></para>
164 /// </summary>
165 /// <param name="character">
166 /// <para>The character.</para>
167 /// <para></para>
168 /// </param>
169 private static void RegisterExistingCharacter(Link character)
170 {
171     if (character.Source == Net.Character && character.Linker == Net.ThatHas)
172     {
173         var code = character.Target;
174         if (code.Source == Net.Code && code.Linker == Net.ThatIsRepresentedBy)
175         {
176             var charCode = (char)LinkConverter.ToNumber(code.Target);
177             _charactersToLinks[charCode] = character;
178             _linksToCharacters[character] = charCode;
179         }
180     }
181 }
182
183 /// <summary>
184 /// <para>
185 /// Recreates.
186 /// </para>
187 /// <para></para>
188 /// </summary>
189 public static void Recreate() => Create();
190
191 /// <summary>
192 /// <para>
193 /// Creates the latin alphabet.
194 /// </para>
195 /// <para></para>
196 /// </summary>
197 private static void CreateLatinAlphabet()
198 {
199     var lettersCharacters = new[]
200     {
201         'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j',
202         'k', 'l', 'm', 'n', 'o', 'p', 'q', 'r', 's', 't',
203         'u', 'v', 'w', 'x', 'y', 'z',
204     };
205     CreateAlphabet(lettersCharacters, "latin alphabet", CharacterMapping.LatinAlphabet);
206 }
207
208 /// <summary>
209 /// <para>
210 /// Creates the cyrillic alphabet.
211 /// </para>
212 /// <para></para>
213 /// </summary>
214 private static void CreateCyrillicAlphabet()
215 {
216     var lettersCharacters = new[]
217     {
218         'a', 'б', 'в', 'г', 'д', 'е', 'ё', 'ж', 'з', 'и',
219         'й', 'к', 'л', 'м', 'н', 'о', 'п', 'р', 'с', 'т',
220         'у', 'ф', 'х', 'ц', 'ч', 'ш', 'щ', 'ъ', 'ы', 'ь',
221         'э', 'ю', 'я',
222     };
223     CreateAlphabet(lettersCharacters, "cyrillic alphabet",
    ↪ CharacterMapping.CyrillicAlphabet);
224 }
225
226 /// <summary>
227 /// <para>
228 /// Creates the alphabet using the specified letters characters.
229 /// </para>
230 /// <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 ==
318                 ↪ Net.ThatIsRepresentedBy)
319             {
320                 var charCode = (char)LinkConverter.ToNumber(target.Target);
321                 _charactersToLinks[charCode] = letter;
322                 _linksToCharacters[letter] = charCode;
323             }
324         }
325     });
326 }
327
328 /// <summary>
329 /// <para>
330 /// Generates the alphabet basis using the specified alphabet.
331 /// </para>
332 /// <para></para>
333 /// </summary>
334 /// <param name="alphabet">
335 /// <para>The alphabet.</para>
336 /// <para></para>
337 /// </param>
338 /// <param name="letterOfAlphabet">
339 /// <para>The letter of alphabet.</para>
340 /// <para></para>
341 /// </param>
342 /// <param name="letters">
343 /// <para>The letters.</para>
344 /// <para></para>
345 /// </param>
346 private static void GenerateAlphabetBasis(ref Link alphabet, ref Link letterOfAlphabet,
347     ↪ Link[] letters)
348 {
349     // Принцип, на примере латинского алфавита.
350     //latin alphabet: alphabet that consists of a and b and c and ... and z.
351     //a: letter of latin alphabet that is before b.
352     //b: letter of latin alphabet that is between (a and c).
353     //c: letter of latin alphabet that is between (b and e).
354     //...
355     //y: letter of latin alphabet that is between (x and z).
356     //z: letter of latin alphabet that is after y.
357     const int firstLetterIndex = 0;
358     for (var i = firstLetterIndex; i < letters.Length; i++)
359     {
360         letters[i] = Net.CreateThing();
361     }
362     var lastLetterIndex = letters.Length - 1;
363     Link.Update(ref letters[firstLetterIndex], letterOfAlphabet, Net.ThatIsBefore,
364         ↪ letters[firstLetterIndex + 1]);
365     Link.Update(ref letters[lastLetterIndex], letterOfAlphabet, Net.ThatIsAfter,
366         ↪ letters[lastLetterIndex - 1]);
367     const int secondLetterIndex = firstLetterIndex + 1;
368     for (var i = secondLetterIndex; i < lastLetterIndex; i++)
369     {
370         Link.Update(ref letters[i], letterOfAlphabet, Net.ThatIsBetween, letters[i - 1]
371             ↪ & letters[i + 1]);
372     }
373     Link.Update(ref alphabet, Net.Alphabet, Net.ThatConsistsOf,
374         ↪ LinkConverter.FromList(letters));
375 }
376
377 /// <summary>
378 /// <para>
379 /// Sets the letter codes using the specified letter.
380 /// </para>
381 /// <para></para>
382 /// </summary>

```

```

377     /// <param name="letter">
378     /// <para>The letter.</para>
379     /// <para></para>
380     /// </param>
381     /// <param name="lowerCaseCharacter">
382     /// <para>The lower case character.</para>
383     /// <para></para>
384     /// </param>
385     /// <param name="lowerCase">
386     /// <para>The lower case.</para>
387     /// <para></para>
388     /// </param>
389     /// <param name="upperCase">
390     /// <para>The upper case.</para>
391     /// <para></para>
392     /// </param>
393     private static void SetLetterCodes(Link letter, char lowerCaseCharacter, out Link
    ↪ lowerCase, out Link upperCase)
394     {
395         var upperCaseCharacter = char.ToUpper(lowerCaseCharacter);
396         if (upperCaseCharacter != lowerCaseCharacter)
397         {
398             lowerCase = Link.Create(Net.LowerCase, Net.Of, letter);
399             var lowerCaseCharacterCode = Link.Create(Net.Code, Net.ThatIsRepresentedBy,
    ↪ LinkConverter.FromNumber(lowerCaseCharacter));
400             Link.Create(lowerCase, Net.Has, lowerCaseCharacterCode);
401             upperCase = Link.Create(Net.UpperCase, Net.Of, letter);
402             var upperCaseCharacterCode = Link.Create(Net.Code, Net.ThatIsRepresentedBy,
    ↪ LinkConverter.FromNumber(upperCaseCharacter));
403             Link.Create(upperCase, Net.Has, upperCaseCharacterCode);
404         }
405         else
406         {
407             lowerCase = letter;
408             upperCase = null;
409             Link.Create(letter, Net.Has, Link.Create(Net.Code, Net.ThatIsRepresentedBy,
    ↪ LinkConverter.FromNumber(lowerCaseCharacter)));
410         }
411     }
412
413     /// <summary>
414     /// <para>
415     /// Creates the simple character link using the specified character.
416     /// </para>
417     /// <para></para>
418     /// </summary>
419     /// <param name="character">
420     /// <para>The character.</para>
421     /// <para></para>
422     /// </param>
423     /// <returns>
424     /// <para>The link</para>
425     /// <para></para>
426     /// </returns>
427     private static Link CreateSimpleCharacterLink(char character) =>
    ↪ Link.Create(Net.Character, Net.ThatHas, Link.Create(Net.Code,
    ↪ Net.ThatIsRepresentedBy, LinkConverter.FromNumber(character)));
428
429     /// <summary>
430     /// <para>
431     /// Determines whether is letter of latin alphabet.
432     /// </para>
433     /// <para></para>
434     /// </summary>
435     /// <param name="character">
436     /// <para>The character.</para>
437     /// <para></para>
438     /// </param>
439     /// <returns>
440     /// <para>The bool</para>
441     /// <para></para>
442     /// </returns>
443     private static bool IsLetterOfLatinAlphabet(char character)
444         => (character >= FirstLowerCaseLatinLetter && character <= LastLowerCaseLatinLetter)
445         || (character >= FirstUpperCaseLatinLetter && character <= LastUpperCaseLatinLetter);
446
447     /// <summary>
448     /// <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)
         || (character >= FirstUpperCaseCyrillicLetter && character <=
         ↳ LastUpperCaseCyrillicLetter)
         || character == YoLowerCaseCyrillicLetter || character == YoUpperCaseCyrillicLetter;

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     }

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 является 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>

```



```

49     /// </summary>
50     /// <returns>
51     /// <para>The string</para>
52     /// <para></para>
53     /// </returns>
54     public static string ToXml()
55     {
56         var sb = new StringBuilder();
57         using (var writer = XmlWriter.Create(sb))
58         {
59             WriteXml(writer, CollectLinks());
60         }
61         return sb.ToString();
62     }
63
64     /// <summary>
65     /// <para>
66     /// Returns the file using the specified path.
67     /// </para>
68     /// <para></para>
69     /// </summary>
70     /// <param name="path">
71     /// <para>The path.</para>
72     /// <para></para>
73     /// </param>
74     public static void ToFile(string path)
75     {
76         using (var file = File.OpenWrite(path))
77         using (var writer = XmlWriter.Create(file))
78         {
79             WriteXml(writer, CollectLinks());
80         }
81     }
82
83     /// <summary>
84     /// <para>
85     /// Returns the file using the specified path.
86     /// </para>
87     /// <para></para>
88     /// </summary>
89     /// <param name="path">
90     /// <para>The path.</para>
91     /// <para></para>
92     /// </param>
93     /// <param name="filter">
94     /// <para>The filter.</para>
95     /// <para></para>
96     /// </param>
97     public static void ToFile(string path, Func<Link, bool> filter)
98     {
99         using (var file = File.OpenWrite(path))
100         using (var writer = XmlWriter.Create(file))
101         {
102             WriteXml(writer, CollectLinks(filter));
103         }
104     }
105
106     /// <summary>
107     /// <para>
108     /// Collects the links using the specified link match.
109     /// </para>
110     /// <para></para>
111     /// </summary>
112     /// <param name="linkMatch">
113     /// <para>The link match.</para>
114     /// <para></para>
115     /// </param>
116     /// <returns>
117     /// <para>The matching links.</para>
118     /// <para></para>
119     /// </returns>
120     private static HashSet<Link> CollectLinks(Func<Link, bool> linkMatch)
121     {
122         var matchingLinks = new HashSet<Link>();
123         Link.WalkThroughAllLinks(link =>
124         {
125             if (linkMatch(link))
126                 matchingLinks.Add(link);
127         });
128         return matchingLinks;
129     }

```

```

127         matchingLinks.Add(link);
128     }
129 });
130 return matchingLinks;
131 }
132
133 /// <summary>
134 /// <para>
135 /// Collects the links.
136 /// </para>
137 /// <para></para>
138 /// </summary>
139 /// <returns>
140 /// <para>The matching links.</para>
141 /// <para></para>
142 /// </returns>
143 private static HashSet<Link> CollectLinks()
144 {
145     var matchingLinks = new HashSet<Link>();
146     Link.WalkThroughAllLinks(matchingLinks.AddAndReturnVoid);
147     return matchingLinks;
148 }
149
150 /// <summary>
151 /// <para>
152 /// Writes the xml using the specified writer.
153 /// </para>
154 /// <para></para>
155 /// </summary>
156 /// <param name="writer">
157 /// <para>The writer.</para>
158 /// <para></para>
159 /// </param>
160 /// <param name="matchingLinks">
161 /// <para>The matching links.</para>
162 /// <para></para>
163 /// </param>
164 private static void WriteXml(XmlWriter writer, HashSet<Link> matchingLinks)
165 {
166     var edgesCounter = 0;
167     Gexf.WriteXml(writer,
168         () => // nodes
169         {
170             foreach (var matchingLink in matchingLinks)
171             {
172                 GexfNode.WriteXml(writer, matchingLink.ToInt(), matchingLink.ToString());
173             }
174         },
175         () => // edges
176         {
177             foreach (var matchingLink in matchingLinks)
178             {
179                 if (matchingLinks.Contains(matchingLink.Source))
180                 {
181                     Edge.WriteXml(writer, edgesCounter++, matchingLink.ToInt(),
182                         ⇨ matchingLink.Source.ToInt(), SourceLabel);
183                 }
184                 if (matchingLinks.Contains(matchingLink.Linker))
185                 {
186                     Edge.WriteXml(writer, edgesCounter++, matchingLink.ToInt(),
187                         ⇨ matchingLink.Linker.ToInt(), LinkerLabel);
188                 }
189                 if (matchingLinks.Contains(matchingLink.Target))
190                 {
191                     Edge.WriteXml(writer, edgesCounter++, matchingLink.ToInt(),
192                         ⇨ matchingLink.Target.ToInt(), TargetLabel);
193                 }
194             }
195         });
196 }
197 }
198 }
199 }

```

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

```

1 using System;
2
3 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4

```

```

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>

```

```

83     /// </summary>
84     /// <param name="walker">
85     /// <para>The walker.</para>
86     /// <para></para>
87     /// </param>
88     /// <returns>
89     /// <para>The bool</para>
90     /// <para></para>
91     /// </returns>
92     bool WalkThroughReferersAsTarget(Func<TLink, bool> walker);
93     /// <summary>
94     /// <para>
95     /// Walks the through referers using the specified walker.
96     /// </para>
97     /// <para></para>
98     /// </summary>
99     /// <param name="walker">
100    /// <para>The walker.</para>
101    /// <para></para>
102    /// </param>
103    void WalkThroughReferers(Func<TLink, bool> walker);
104 }
105
106 /// <summary>
107 /// <para>
108 /// Defines the link.
109 /// </para>
110 /// <para></para>
111 /// </summary>
112 internal partial interface ILink<TLink>
113     where TLink : ILink<TLink>
114 {
115     /// <summary>
116     /// <para>
117     /// Walks the through referers as linker using the specified walker.
118     /// </para>
119     /// <para></para>
120     /// </summary>
121     /// <param name="walker">
122     /// <para>The walker.</para>
123     /// <para></para>
124     /// </param>
125     void WalkThroughReferersAsLinker(Action<TLink> walker);
126     /// <summary>
127     /// <para>
128     /// Walks the through referers as source using the specified walker.
129     /// </para>
130     /// <para></para>
131     /// </summary>
132     /// <param name="walker">
133     /// <para>The walker.</para>
134     /// <para></para>
135     /// </param>
136     void WalkThroughReferersAsSource(Action<TLink> walker);
137     /// <summary>
138     /// <para>
139     /// Walks the through referers as target using the specified walker.
140     /// </para>
141     /// <para></para>
142     /// </summary>
143     /// <param name="walker">
144     /// <para>The walker.</para>
145     /// <para></para>
146     /// </param>
147     void WalkThroughReferersAsTarget(Action<TLink> walker);
148     /// <summary>
149     /// <para>
150     /// Walks the through referers using the specified walker.
151     /// </para>
152     /// <para></para>
153     /// </summary>
154     /// <param name="walker">
155     /// <para>The walker.</para>
156     /// <para></para>
157     /// </param>
158     void WalkThroughReferers(Action<TLink> walker);
159 }
160 }

```

```

161  /*
162  using System;
163  namespace NetLibrary
164  {
165      interface ILink
166      {
167          // Статические методы (общие для всех связей)
168          public static ILink Create(ILink source, ILink linker, ILink target);
169          public static void Update(ref ILink link, ILink newSource, ILink newLinker, ILink
↪ newTarget);
170          public static void Delete(ref ILink link);
171          public static ILink Search(ILink source, ILink linker, ILink target);
172      }
173  }
174  */
175  /*
176  Набор функций, который необходим для работы с сущностью Link:
177
178  (Работа со значением сущности Link, значение состоит из 3-х частей, также сущностей Link)
179  1. Получить адрес "начальной" сущности Link. (Получить адрес из поля Source)
180  2. Получить адрес сущности Link, которая играет роль связи между "начальной" и "конечной"
↪ сущностями Link. (Получить адрес из поля Linker)
181  3. Получить адрес "конечной" сущности Link. (Получить адрес из поля Target)
182
183  4. Пройтись по всем сущностям Link, которые ссылаются на сущность Link с указанным адресом, и у
↪ которых поле Source равно этому адресу.
184  5. Пройтись по всем сущностям Link, которые ссылаются на сущность Link с указанным адресом, и у
↪ которых поле Linker равно этому адресу.
185  6. Пройтись по всем сущностям Link, которые ссылаются на сущность Link с указанным адресом, и у
↪ которых поле Target равно этому адресу.
186
187  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").
188  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").
189  9. Удаление сущности Link с указанным адресом.
190  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").
191  */

```

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

```

1  using System;
2  using System.Diagnostics;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Data.Triplets
7  {
8      /// <summary>
9      /// <para>
10     /// The link.
11     /// </para>
12     /// <para></para>
13     /// </summary>
14     public partial struct Link
15     {
16         #region Properties
17
18         // ReSharper disable InconsistentNaming
19         // ReSharper disable UnusedMember.Local
20         #pragma warning disable IDE0051 // Remove unused private members
21
22         /// <summary>
23         /// <para>
24         /// Gets the я a value.
25         /// </para>
26         /// <para></para>
27         /// </summary>
28         [DebuggerDisplay(null, Name = "Source")]
29         private Link Я_A => this == null ? Itself : Source;
30
31         /// <summary>
32         /// <para>
33         /// Gets the я b value.
34         /// </para>
35         /// <para></para>

```

```

36     /// </summary>
37     [DebuggerDisplay(null, Name = "Linker")]
38     private Link Я_B => this == null ? Itself : Linker;
39
40     /// <summary>
41     /// <para>
42     /// Gets the я c value.
43     /// </para>
44     /// <para></para>
45     /// </summary>
46     [DebuggerDisplay(null, Name = "Target")]
47     private Link Я_C => this == null ? Itself : Target;
48
49     /// <summary>
50     /// <para>
51     /// Gets the я d value.
52     /// </para>
53     /// <para></para>
54     /// </summary>
55     [DebuggerDisplay("Count = {Я_DC}", Name = "ReferersBySource")]
56     private Link[] Я_D => this.GetArrayOfRererersBySource();
57
58     /// <summary>
59     /// <para>
60     /// Gets the я e value.
61     /// </para>
62     /// <para></para>
63     /// </summary>
64     [DebuggerDisplay("Count = {Я_EC}", Name = "ReferersByLinker")]
65     private Link[] Я_E => this.GetArrayOfRererersByLinker();
66
67     /// <summary>
68     /// <para>
69     /// Gets the я f value.
70     /// </para>
71     /// <para></para>
72     /// </summary>
73     [DebuggerDisplay("Count = {Я_FC}", Name = "ReferersByTarget")]
74     private Link[] Я_F => this.GetArrayOfRererersByTarget();
75
76     /// <summary>
77     /// <para>
78     /// Gets the я dc value.
79     /// </para>
80     /// <para></para>
81     /// </summary>
82     [DebuggerBrowsable(DebuggerBrowsableState.Never)]
83     private Int64 Я_DC => this == null ? 0 : ReferersBySourceCount;
84
85     /// <summary>
86     /// <para>
87     /// Gets the я ec value.
88     /// </para>
89     /// <para></para>
90     /// </summary>
91     [DebuggerBrowsable(DebuggerBrowsableState.Never)]
92     private Int64 Я_EC => this == null ? 0 : ReferersByLinkerCount;
93
94     /// <summary>
95     /// <para>
96     /// Gets the я fc value.
97     /// </para>
98     /// <para></para>
99     /// </summary>
100    [DebuggerBrowsable(DebuggerBrowsableState.Never)]
101    private Int64 Я_FC => this == null ? 0 : ReferersByTargetCount;
102
103    /// <summary>
104    /// <para>
105    /// Gets the я h value.
106    /// </para>
107    /// <para></para>
108    /// </summary>
109    [DebuggerDisplay(null, Name = "Timestamp")]
110    private DateTime Я_H => this == null ? DateTime.MinValue : Timestamp;
111
112    // ReSharper restore UnusedMember.Local
113    // ReSharper restore InconsistentNaming
114    #pragma warning restore IDE0051 // Remove unused private members

```

```

115         #endregion
116
117         /// <summary>
118         /// <para>
119         /// Returns the string.
120         /// </para>
121         /// <para></para>
122         /// </summary>
123         /// <returns>
124         /// <para>The string</para>
125         /// <para></para>
126         /// </returns>
127     public override string ToString()
128     {
129         const string nullString = "null";
130         if (this == null)
131         {
132             return nullString;
133         }
134         else
135         {
136             if (this.TryGetName(out string name))
137             {
138                 return name;
139             }
140             else
141             {
142                 return ((long)_link).ToString();
143             }
144         }
145     }
146 }
147 }
148 }

```

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

```

1  using System;
2  using System.Collections.Generic;
3  using System.Diagnostics;
4  using System.Runtime.InteropServices;
5  using System.Threading;
6  using Int = System.Int64;
7  using LinkIndex = System.UInt64;
8
9  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
10
11 namespace Platform.Data.Triplets
12 {
13     /// <summary>
14     /// <para>
15     /// The link definition.
16     /// </para>
17     /// <para></para>
18     /// </summary>
19     public struct LinkDefinition : IEquatable<LinkDefinition>
20     {
21         /// <summary>
22         /// <para>
23         /// The source.
24         /// </para>
25         /// <para></para>
26         /// </summary>
27         public readonly Link Source;
28         /// <summary>
29         /// <para>
30         /// The linker.
31         /// </para>
32         /// <para></para>
33         /// </summary>
34         public readonly Link Linker;
35         /// <summary>
36         /// <para>
37         /// The target.
38         /// </para>
39         /// <para></para>
40         /// </summary>
41         public readonly Link Target;
42
43         /// <summary>
44         /// <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 ==
95     ↪ other.Linker && Target == other.Target;
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     /// <summary>
123     /// <para>
124     /// The updated delegate.
125     /// </para>
126     /// <para></para>
127     /// </summary>
128     public delegate void UpdatedDelegate(LinkDefinition linkBeforeUpdate, LinkDefinition
129     ↪ linkAfterUpdate);
130     public static event UpdatedDelegate UpdatedEvent = (linkBeforeUpdate, linkAfterUpdate)
131     ↪ => { };
132
133     /// <summary>
134     /// <para>
135     /// The deleted delegate.
136     /// </para>
137     /// <para></para>
138     /// </summary>
139     public delegate void DeletedDelegate(LinkDefinition deletedLink);
140     public static event DeletedDelegate DeletedEvent = deletedLink => { };
141
142     #region Low Level
143
144     #region Basic Operations
145
146     /// <summary>
147     /// <para>
148     /// Gets the source index using the specified link.
149     /// </para>
150     /// <para></para>
151     /// </summary>
152     /// <param name="link">
153     /// <para>The link.</para>
154     /// <para></para>
155     /// </param>
156     /// <returns>
157     /// <para>The link index</para>
158     /// <para></para>
159     /// </returns>
160     [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
161     private static extern LinkIndex GetSourceIndex(LinkIndex link);
162
163     /// <summary>
164     /// <para>
165     /// Gets the linker index using the specified link.
166     /// </para>
167     /// <para></para>
168     /// </summary>
169     /// <param name="link">
170     /// <para>The link.</para>
171     /// <para></para>
172     /// </param>
173     /// <returns>
174     /// <para>The link index</para>
175     /// <para></para>
176     /// </returns>
177     [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
178     private static extern LinkIndex GetLinkerIndex(LinkIndex link);
179
180     /// <summary>
181     /// <para>
182     /// Gets the target index using the specified link.
183     /// </para>
184     /// <para></para>
185     /// </summary>
186     /// <param name="link">
187     /// <para>The link.</para>
188     /// <para></para>
189     /// </param>
190     /// <returns>
191     /// <para>The link index</para>
192     /// <para></para>
193     /// </returns>
194     [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
195     private static extern LinkIndex GetTargetIndex(LinkIndex link);
196
197     /// <summary>
198     /// <para>

```

```

198     /// Gets the first referer by source index using the specified link.
199     /// </para>
200     /// <para></para>
201     /// </summary>
202     /// <param name="link">
203     /// <para>The link.</para>
204     /// <para></para>
205     /// </param>
206     /// <returns>
207     /// <para>The link index</para>
208     /// <para></para>
209     /// </returns>
210     [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
211     private static extern LinkIndex GetFirstRefererBySourceIndex(LinkIndex link);
212
213     /// <summary>
214     /// <para>
215     /// Gets the first referer by linker index using the specified link.
216     /// </para>
217     /// <para></para>
218     /// </summary>
219     /// <param name="link">
220     /// <para>The link.</para>
221     /// <para></para>
222     /// </param>
223     /// <returns>
224     /// <para>The link index</para>
225     /// <para></para>
226     /// </returns>
227     [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
228     private static extern LinkIndex GetFirstRefererByLinkerIndex(LinkIndex link);
229
230     /// <summary>
231     /// <para>
232     /// Gets the first referer by target index using the specified link.
233     /// </para>
234     /// <para></para>
235     /// </summary>
236     /// <param name="link">
237     /// <para>The link.</para>
238     /// <para></para>
239     /// </param>
240     /// <returns>
241     /// <para>The link index</para>
242     /// <para></para>
243     /// </returns>
244     [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
245     private static extern LinkIndex GetFirstRefererByTargetIndex(LinkIndex link);
246
247     /// <summary>
248     /// <para>
249     /// Gets the time using the specified link.
250     /// </para>
251     /// <para></para>
252     /// </summary>
253     /// <param name="link">
254     /// <para>The link.</para>
255     /// <para></para>
256     /// </param>
257     /// <returns>
258     /// <para>The int</para>
259     /// <para></para>
260     /// </returns>
261     [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
262     private static extern Int GetTime(LinkIndex link);
263
264     /// <summary>
265     /// <para>
266     /// Creates the link using the specified source.
267     /// </para>
268     /// <para></para>
269     /// </summary>
270     /// <param name="source">
271     /// <para>The source.</para>
272     /// <para></para>
273     /// </param>
274     /// <param name="linker">
275     /// <para>The linker.</para>

```

```

276     /// <para></para>
277     /// </param>
278     /// <param name="target">
279     /// <para>The target.</para>
280     /// <para></para>
281     /// </param>
282     /// <returns>
283     /// <para>The link index</para>
284     /// <para></para>
285     /// </returns>
286     [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
287     private static extern LinkIndex CreateLink(LinkIndex source, LinkIndex linker, LinkIndex
        ↪ target);
288
289     /// <summary>
290     /// <para>
291     /// Updates the link using the specified link.
292     /// </para>
293     /// <para></para>
294     /// </summary>
295     /// <param name="link">
296     /// <para>The link.</para>
297     /// <para></para>
298     /// </param>
299     /// <param name="newSource">
300     /// <para>The new source.</para>
301     /// <para></para>
302     /// </param>
303     /// <param name="newLinker">
304     /// <para>The new linker.</para>
305     /// <para></para>
306     /// </param>
307     /// <param name="newTarget">
308     /// <para>The new target.</para>
309     /// <para></para>
310     /// </param>
311     /// <returns>
312     /// <para>The link index</para>
313     /// <para></para>
314     /// </returns>
315     [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
316     private static extern LinkIndex UpdateLink(LinkIndex link, LinkIndex newSource,
        ↪ LinkIndex newLinker, LinkIndex newTarget);
317
318     /// <summary>
319     /// <para>
320     /// Deletes the link using the specified link.
321     /// </para>
322     /// <para></para>
323     /// </summary>
324     /// <param name="link">
325     /// <para>The link.</para>
326     /// <para></para>
327     /// </param>
328     [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
329     private static extern void DeleteLink(LinkIndex link);
330
331     /// <summary>
332     /// <para>
333     /// Replaces the link using the specified link.
334     /// </para>
335     /// <para></para>
336     /// </summary>
337     /// <param name="link">
338     /// <para>The link.</para>
339     /// <para></para>
340     /// </param>
341     /// <param name="replacement">
342     /// <para>The replacement.</para>
343     /// <para></para>
344     /// </param>
345     /// <returns>
346     /// <para>The link index</para>
347     /// <para></para>
348     /// </returns>
349     [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
350     private static extern LinkIndex ReplaceLink(LinkIndex link, LinkIndex replacement);
351

```

```

352     /// <summary>
353     /// <para>
354     /// Searches the link using the specified source.
355     /// </para>
356     /// <para></para>
357     /// </summary>
358     /// <param name="source">
359     /// <para>The source.</para>
360     /// <para></para>
361     /// </param>
362     /// <param name="linker">
363     /// <para>The linker.</para>
364     /// <para></para>
365     /// </param>
366     /// <param name="target">
367     /// <para>The target.</para>
368     /// <para></para>
369     /// </param>
370     /// <returns>
371     /// <para>The link index</para>
372     /// <para></para>
373     /// </returns>
374     [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
375     private static extern LinkIndex SearchLink(LinkIndex source, LinkIndex linker, LinkIndex
        ↪ target);
376
377     /// <summary>
378     /// <para>
379     /// Gets the mapped link using the specified mapped index.
380     /// </para>
381     /// <para></para>
382     /// </summary>
383     /// <param name="mappedIndex">
384     /// <para>The mapped index.</para>
385     /// <para></para>
386     /// </param>
387     /// <returns>
388     /// <para>The link index</para>
389     /// <para></para>
390     /// </returns>
391     [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
392     private static extern LinkIndex GetMappedLink(Int mappedIndex);
393
394     /// <summary>
395     /// <para>
396     /// Sets the mapped link using the specified mapped index.
397     /// </para>
398     /// <para></para>
399     /// </summary>
400     /// <param name="mappedIndex">
401     /// <para>The mapped index.</para>
402     /// <para></para>
403     /// </param>
404     /// <param name="linkIndex">
405     /// <para>The link index.</para>
406     /// <para></para>
407     /// </param>
408     [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
409     private static extern void SetMappedLink(Int mappedIndex, LinkIndex linkIndex);
410
411     /// <summary>
412     /// <para>
413     /// Opens the links using the specified filename.
414     /// </para>
415     /// <para></para>
416     /// </summary>
417     /// <param name="filename">
418     /// <para>The filename.</para>
419     /// <para></para>
420     /// </param>
421     /// <returns>
422     /// <para>The int</para>
423     /// <para></para>
424     /// </returns>
425     [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
426     private static extern Int OpenLinks(string filename);
427
428     /// <summary>

```

```

429     /// <para>
430     /// Closes the links.
431     /// </para>
432     /// <para></para>
433     /// </summary>
434     /// <returns>
435     /// <para>The int</para>
436     /// <para></para>
437     /// </returns>
438     [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
439     private static extern Int CloseLinks();
440
441     #endregion
442
443     #region Referers Count Selectors
444
445     /// <summary>
446     /// <para>
447     /// Gets the link number of referers by source using the specified link.
448     /// </para>
449     /// <para></para>
450     /// </summary>
451     /// <param name="link">
452     /// <para>The link.</para>
453     /// <para></para>
454     /// </param>
455     /// <returns>
456     /// <para>The link index</para>
457     /// <para></para>
458     /// </returns>
459     [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
460     private static extern LinkIndex GetLinkNumberOfReferersBySource(LinkIndex link);
461
462     /// <summary>
463     /// <para>
464     /// Gets the link number of referers by linker using the specified link.
465     /// </para>
466     /// <para></para>
467     /// </summary>
468     /// <param name="link">
469     /// <para>The link.</para>
470     /// <para></para>
471     /// </param>
472     /// <returns>
473     /// <para>The link index</para>
474     /// <para></para>
475     /// </returns>
476     [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
477     private static extern LinkIndex GetLinkNumberOfReferersByLinker(LinkIndex link);
478
479     /// <summary>
480     /// <para>
481     /// Gets the link number of referers by target using the specified link.
482     /// </para>
483     /// <para></para>
484     /// </summary>
485     /// <param name="link">
486     /// <para>The link.</para>
487     /// <para></para>
488     /// </param>
489     /// <returns>
490     /// <para>The link index</para>
491     /// <para></para>
492     /// </returns>
493     [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
494     private static extern LinkIndex GetLinkNumberOfReferersByTarget(LinkIndex link);
495
496     #endregion
497
498     #region Referers Walkers
499
500     /// <summary>
501     /// <para>
502     /// The visitor.
503     /// </para>
504     /// <para></para>
505     /// </summary>
506     private delegate void Visitor(LinkIndex link);
507     /// <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>

```

```

583     /// </param>
584     /// <returns>
585     /// <para>The int</para>
586     /// <para></para>
587     /// </returns>
588     [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
589     private static extern int WalkThroughReferersByLinker(LinkIndex root, StopableVisitor
        ↪ func);
590
591     /// <summary>
592     /// <para>
593     /// Walks the through all referers by target using the specified root.
594     /// </para>
595     /// <para></para>
596     /// </summary>
597     /// <param name="root">
598     /// <para>The root.</para>
599     /// <para></para>
600     /// </param>
601     /// <param name="action">
602     /// <para>The action.</para>
603     /// <para></para>
604     /// </param>
605     [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
606     private static extern void WalkThroughAllReferersByTarget(LinkIndex root, Visitor
        ↪ action);
607
608     /// <summary>
609     /// <para>
610     /// Walks the through referers by target using the specified root.
611     /// </para>
612     /// <para></para>
613     /// </summary>
614     /// <param name="root">
615     /// <para>The root.</para>
616     /// <para></para>
617     /// </param>
618     /// <param name="func">
619     /// <para>The func.</para>
620     /// <para></para>
621     /// </param>
622     /// <returns>
623     /// <para>The int</para>
624     /// <para></para>
625     /// </returns>
626     [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
627     private static extern int WalkThroughReferersByTarget(LinkIndex root, StopableVisitor
        ↪ func);
628
629     /// <summary>
630     /// <para>
631     /// Walks the through all links using the specified action.
632     /// </para>
633     /// <para></para>
634     /// </summary>
635     /// <param name="action">
636     /// <para>The action.</para>
637     /// <para></para>
638     /// </param>
639     [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
640     private static extern void WalkThroughAllLinks(Visitor action);
641
642     /// <summary>
643     /// <para>
644     /// Walks the through links using the specified func.
645     /// </para>
646     /// <para></para>
647     /// </summary>
648     /// <param name="func">
649     /// <para>The func.</para>
650     /// <para></para>
651     /// </param>
652     /// <returns>
653     /// <para>The int</para>
654     /// <para></para>
655     /// </returns>
656     [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
657     private static extern int WalkThroughLinks(StopableVisitor func);

```

```

658 #endregion
659
660 #endregion
661
662 #region Constains
663
664     /// <summary>
665     /// <para>
666     /// The itself.
667     /// </para>
668     /// <para></para>
669     /// </summary>
670 public static readonly Link Itself = null;
671     /// <summary>
672     /// <para>
673     /// The continue.
674     /// </para>
675     /// <para></para>
676     /// </summary>
677 public static readonly bool Continue = true;
678     /// <summary>
679     /// <para>
680     /// The stop.
681     /// </para>
682     /// <para></para>
683     /// </summary>
684 public static readonly bool Stop;
685
686 #endregion
687
688 #region Static Fields
689
690     /// <summary>
691     /// <para>
692     /// The lock object.
693     /// </para>
694     /// <para></para>
695     /// </summary>
696 private static readonly object _lockObject = new object();
697     /// <summary>
698     /// <para>
699     /// The memory manager is ready.
700     /// </para>
701     /// <para></para>
702     /// </summary>
703 private static volatile int _memoryManagerIsReady;
704     /// <summary>
705     /// <para>
706     /// The dictionary.
707     /// </para>
708     /// <para></para>
709     /// </summary>
710 private static readonly Dictionary<ulong, long> _linkToMappingIndex = new
711     ↪ 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>

```

```

815     /// <para></para>
816     /// </summary>
817     [DebuggerBrowsable(DebuggerBrowsableState.Never)]
818     public Int TotalReferers => (Int)GetLinkNumberOfReferersBySource(_link) +
        ↳ (Int)GetLinkNumberOfReferersByLinker(_link) +
        ↳ (Int)GetLinkNumberOfReferersByTarget(_link);

819
820     /// <summary>
821     /// <para>
822     /// Gets the timestamp value.
823     /// </para>
824     /// <para></para>
825     /// </summary>
826     [DebuggerBrowsable(DebuggerBrowsableState.Never)]
827     public DateTime Timestamp => DateTime.FromFileTimeUtc(GetTime(_link));
828
829     #endregion
830
831     #region Infrastructure
832
833     /// <summary>
834     /// <para>
835     /// Initializes a new <see cref="Link"/> instance.
836     /// </para>
837     /// <para></para>
838     /// </summary>
839     /// <param name="link">
840     /// <para>A link.</para>
841     /// <para></para>
842     /// </param>
843     public Link(LinkIndex link) => _link = link;
844
845     /// <summary>
846     /// <para>
847     /// Starts the memory manager using the specified storage filename.
848     /// </para>
849     /// <para></para>
850     /// </summary>
851     /// <param name="storageFilename">
852     /// <para>The storage filename.</para>
853     /// <para></para>
854     /// </param>
855     /// <exception cref="InvalidOperationException">
856     /// <para>Файл ({storageFilename}) хранилища не удалось открыть.</para>
857     /// <para></para>
858     /// </exception>
859     public static void StartMemoryManager(string storageFilename)
860     {
861         lock (_lockObject)
862         {
863             if (_memoryManagerIsReady == default)
864             {
865                 if (OpenLinks(storageFilename) == 0)
866                 {
867                     throw new InvalidOperationException($"Файл ({storageFilename})
868                         ↳ хранилища не удалось открыть.");
869                 }
870                 Interlocked.Exchange(ref _memoryManagerIsReady, 1);
871             }
872         }
873
874     /// <summary>
875     /// <para>
876     /// Stops the memory manager.
877     /// </para>
878     /// <para></para>
879     /// </summary>
880     /// <exception cref="InvalidOperationException">
881     /// <para>Файл хранилища не удалось закрыть. Возможно он был уже закрыт, или не
882     ↳ открывался вовсе.</para>
883     /// <para></para>
884     /// </exception>
885     public static void StopMemoryManager()
886     {
887         lock (_lockObject)
888         {
889             if (_memoryManagerIsReady != default)

```

```

889     {
890         if (CloseLinks() == 0)
891         {
892             throw new InvalidOperationException("Файл хранилища не удалось закрыть.
893             ↪ Возможно он был уже закрыт, или не открывался вовсе.");
894         }
895         Interlocked.Exchange(ref _memoryManagerIsReady, 0);
896     }
897 }
898
899 public static implicit operator LinkIndex?(Link link) => link._link == 0 ?
900     ↪ (LinkIndex?)null : link._link;
901
902 public static implicit operator Link(LinkIndex? link) => new Link(link ?? 0);
903
904 public static implicit operator Int(Link link) => (Int)link._link;
905
906 public static implicit operator Link(Int link) => new Link((LinkIndex)link);
907
908 public static implicit operator LinkIndex(Link link) => link._link;
909
910 public static implicit operator Link(LinkIndex link) => new Link(link);
911
912 public static explicit operator Link(List<Link> links) => LinkConverter.FromList(links);
913
914 public static explicit operator Link(Link[] links) => LinkConverter.FromList(links);
915
916 public static explicit operator Link(string @string) =>
917     ↪ LinkConverter.FromString(@string);
918
919 public static bool operator ==(Link first, Link second) => first.Equals(second);
920
921 public static bool operator !=(Link first, Link second) => !first.Equals(second);
922
923 public static Link operator &(amp;Link first, Link second) => Create(first, Net.And, second);
924
925 /// <summary>
926 /// <para>
927 /// Determines whether this instance equals.
928 /// </para>
929 /// </summary>
930 /// <param name="obj">
931 /// <para>The obj.</para>
932 /// </param>
933 /// <returns>
934 /// <para>The bool</para>
935 /// </returns>
936 public override bool Equals(object obj) => obj is Link link ? Equals(link) : false;
937
938 /// <summary>
939 /// <para>
940 /// Determines whether this instance equals.
941 /// </para>
942 /// </summary>
943 /// <param name="other">
944 /// <para>The other.</para>
945 /// </param>
946 /// <returns>
947 /// <para>The bool</para>
948 /// </returns>
949 public bool Equals(Link other) => _link == other._link || (LinkDoesNotExist(_link) &&
950     ↪ LinkDoesNotExist(other._link));
951
952 /// <summary>
953 /// <para>
954 /// Gets the hash code.
955 /// </para>
956 /// </summary>
957 /// <returns>
958 /// <para>The int</para>

```

```

963     /// <para></para>
964     /// </returns>
965     public override int GetHashCode() => base.GetHashCode();
966
967     /// <summary>
968     /// <para>
969     /// Determines whether link does not exist.
970     /// </para>
971     /// <para></para>
972     /// </summary>
973     /// <param name="link">
974     /// <para>The link.</para>
975     /// <para></para>
976     /// </param>
977     /// <returns>
978     /// <para>The bool</para>
979     /// <para></para>
980     /// </returns>
981     private static bool LinkDoesNotExist(LinkIndex link) => link == 0 ||
982     ↪ GetLinkerIndex(link) == 0;
983
984     /// <summary>
985     /// <para>
986     /// Determines whether link was deleted.
987     /// </para>
988     /// <para></para>
989     /// </summary>
990     /// <param name="link">
991     /// <para>The link.</para>
992     /// <para></para>
993     /// </param>
994     /// <returns>
995     /// <para>The bool</para>
996     /// <para></para>
997     /// </returns>
998     private static bool LinkWasDeleted(LinkIndex link) => link != 0 && GetLinkerIndex(link)
999     ↪ == 0;
1000
1001     /// <summary>
1002     /// <para>
1003     /// Determines whether this instance is matching to.
1004     /// </para>
1005     /// <para></para>
1006     /// </summary>
1007     /// <param name="source">
1008     /// <para>The source.</para>
1009     /// <para></para>
1010     /// </param>
1011     /// <param name="linker">
1012     /// <para>The linker.</para>
1013     /// <para></para>
1014     /// </param>
1015     /// <param name="target">
1016     /// <para>The target.</para>
1017     /// <para></para>
1018     /// </param>
1019     /// <returns>
1020     /// <para>The bool</para>
1021     /// <para></para>
1022     /// </returns>
1023     private bool IsMatchingTo(Link source, Link linker, Link target)
1024     => ((Source == this && source == null) || (Source == source))
1025     && ((Linker == this && linker == null) || (Linker == linker))
1026     && ((Target == this && target == null) || (Target == target));
1027
1028     /// <summary>
1029     /// <para>
1030     /// Returns the index.
1031     /// </para>
1032     /// <para></para>
1033     /// </summary>
1034     /// <returns>
1035     /// <para>The link index</para>
1036     /// <para></para>
1037     /// </returns>
1038     public LinkIndex ToIndex() => _link;

```

```

1039    /// <para>
1040    /// Returns the int.
1041    /// </para>
1042    /// <para></para>
1043    /// </summary>
1044    /// <returns>
1045    /// <para>The int</para>
1046    /// <para></para>
1047    /// </returns>
1048    public Int ToInt() => (Int)_link;
1049
1050    #endregion
1051
1052    #region Basic Operations
1053
1054    /// <summary>
1055    /// <para>
1056    /// Creates the source.
1057    /// </para>
1058    /// <para></para>
1059    /// </summary>
1060    /// <param name="source">
1061    /// <para>The source.</para>
1062    /// <para></para>
1063    /// </param>
1064    /// <param name="linker">
1065    /// <para>The linker.</para>
1066    /// <para></para>
1067    /// </param>
1068    /// <param name="target">
1069    /// <para>The target.</para>
1070    /// <para></para>
1071    /// </param>
1072    /// <exception cref="InvalidOperationException">
1073    /// <para>Менеджер памяти ещё не готов.</para>
1074    /// <para></para>
1075    /// </exception>
1076    /// <exception cref="InvalidOperationException">
1077    /// <para>Невозможно создать связь.</para>
1078    /// <para></para>
1079    /// </exception>
1080    /// <exception cref="ArgumentException">
1081    /// <para>Удалённая связь не может использоваться в качестве значения. </para>
1082    /// <para></para>
1083    /// </exception>
1084    /// <exception cref="ArgumentException">
1085    /// <para>Удалённая связь не может использоваться в качестве значения. </para>
1086    /// <para></para>
1087    /// </exception>
1088    /// <exception cref="ArgumentException">
1089    /// <para>Удалённая связь не может использоваться в качестве значения. </para>
1090    /// <para></para>
1091    /// </exception>
1092    /// <returns>
1093    /// <para>The link.</para>
1094    /// <para></para>
1095    /// </returns>
1096    public static Link Create(Link source, Link linker, Link target)
1097    {
1098        if (_memoryManagerIsReady == default)
1099        {
1100            throw new InvalidOperationException("Менеджер памяти ещё не готов.");
1101        }
1102        if (LinkWasDeleted(source))
1103        {
1104            throw new ArgumentException("Удалённая связь не может использоваться в качестве
1105                ↪ значения.", nameof(source));
1106        }
1107        if (LinkWasDeleted(linker))
1108        {
1109            throw new ArgumentException("Удалённая связь не может использоваться в качестве
1110                ↪ значения.", nameof(linker));
1111        }
1112        if (LinkWasDeleted(target))
1113        {
1114            throw new ArgumentException("Удалённая связь не может использоваться в качестве
1115                ↪ значения.", nameof(target));
1116        }
1117    }

```

```

1114     Link link = CreateLink(source, linker, target);
1115     if (link == null)
1116     {
1117         throw new InvalidOperationException("Невозможно создать связь.");
1118     }
1119     CreatedEvent.Invoke(new LinkDefinition(link));
1120     return link;
1121 }
1122
1123 /// <summary>
1124 /// <para>
1125 /// Restores the index.
1126 /// </para>
1127 /// <para></para>
1128 /// </summary>
1129 /// <param name="index">
1130 /// <para>The index.</para>
1131 /// <para></para>
1132 /// </param>
1133 /// <returns>
1134 /// <para>The link</para>
1135 /// <para></para>
1136 /// </returns>
1137 public static Link Restore(Int index) => Restore((LinkIndex)index);
1138
1139 /// <summary>
1140 /// <para>
1141 /// Restores the index.
1142 /// </para>
1143 /// <para></para>
1144 /// </summary>
1145 /// <param name="index">
1146 /// <para>The index.</para>
1147 /// <para></para>
1148 /// </param>
1149 /// <exception cref="InvalidOperationException">
1150 /// <para>Менеджер памяти ещё не готов.</para>
1151 /// <para></para>
1152 /// </exception>
1153 /// <exception cref="InvalidOperationException">
1154 /// <para>Связь с указанным адресом удалена, либо не существовала.</para>
1155 /// <para></para>
1156 /// </exception>
1157 /// <exception cref="ArgumentException">
1158 /// <para>У связи не может быть нулевого адреса.</para>
1159 /// <para></para>
1160 /// </exception>
1161 /// <exception cref="InvalidOperationException">
1162 /// <para>Указатель не является корректным. </para>
1163 /// <para></para>
1164 /// </exception>
1165 /// <returns>
1166 /// <para>The link</para>
1167 /// <para></para>
1168 /// </returns>
1169 public static Link Restore(LinkIndex index)
1170 {
1171     if (_memoryManagerIsReady == default)
1172     {
1173         throw new InvalidOperationException("Менеджер памяти ещё не готов.");
1174     }
1175     if (index == 0)
1176     {
1177         throw new ArgumentException("У связи не может быть нулевого адреса.");
1178     }
1179     try
1180     {
1181         Link link = index;
1182         if (LinkDoesNotExist(link))
1183         {
1184             throw new InvalidOperationException("Связь с указанным адресом удалена, либо
1185                 ↳ не существовала.");
1186         }
1187         return link;
1188     }
1189     catch (Exception ex)
1190     {
1191         throw new InvalidOperationException("Указатель не является корректным.", ex);
1192     }
1193 }

```

```

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);

```

```

1267     if (mappedLink == null)
1268     {
1269         mappedLink = Create(source, linker, target);
1270         SetMappedLink(mappingIndex, mappedLink);
1271         if (GetMappedLink(mappingIndex) != mappedLink)
1272         {
1273             throw new InvalidOperationException("Установить привязанную связь не
1274                 ↳ удалось.");
1275         }
1276     }
1277     else if (!mappedLink.IsMatchingTo(source, linker, target))
1278     {
1279         throw new InvalidOperationException("Существующая привязанная связь не
1280             ↳ соответствует указанным Source, Linker и Target.");
1281     }
1282     _linkToMappingIndex[mappedLink] = mappingIndex;
1283     return mappedLink;
1284 }
1285
1286 /// <summary>
1287 /// <para>
1288 /// Determines whether try set mapped.
1289 /// </para>
1290 /// </summary>
1291 /// <param name="link">
1292 /// <para>The link.</para>
1293 /// </param>
1294 /// <param name="mappingIndex">
1295 /// <para>The mapping index.</para>
1296 /// </param>
1297 /// <param name="rewrite">
1298 /// <para>The rewrite.</para>
1299 /// </param>
1300 /// <returns>
1301 /// <para>The bool</para>
1302 /// </returns>
1303 public static bool TrySetMapped(Link link, Int mappingIndex, bool rewrite = false)
1304 {
1305     Link mappedLink = GetMappedLink(mappingIndex);
1306
1307     if (mappedLink == null || rewrite)
1308     {
1309         mappedLink = link;
1310         SetMappedLink(mappingIndex, mappedLink);
1311         if (GetMappedLink(mappingIndex) != mappedLink)
1312         {
1313             return false;
1314         }
1315     }
1316     else if (!mappedLink.IsMatchingTo(link.Source, link.Linker, link.Target))
1317     {
1318         return false;
1319     }
1320     _linkToMappingIndex[mappedLink] = mappingIndex;
1321     return true;
1322 }
1323
1324 /// <summary>
1325 /// <para>
1326 /// Gets the mapped using the specified mapping index.
1327 /// </para>
1328 /// </summary>
1329 /// <param name="mappingIndex">
1330 /// <para>The mapping index.</para>
1331 /// </param>
1332 /// <returns>
1333 /// <para>The link</para>
1334 /// </returns>
1335 public static Link GetMapped(object mappingIndex) =>
1336     ↳ GetMapped(Convert.ToInt64(mappingIndex));

```



```

1342
1343     /// <summary>
1344     /// <para>
1345     /// Gets the mapped using the specified mapping index.
1346     /// </para>
1347     /// <para></para>
1348     /// </summary>
1349     /// <param name="mappingIndex">
1350     /// <para>The mapping index.</para>
1351     /// <para></para>
1352     /// </param>
1353     /// <exception cref="InvalidOperationException">
1354     /// <para>Mapped link with index {mappingIndex} is not set.</para>
1355     /// <para></para>
1356     /// </exception>
1357     /// <returns>
1358     /// <para>The mapped link.</para>
1359     /// <para></para>
1360     /// </returns>
1361     public static Link GetMapped(Int mappingIndex)
1362     {
1363         if (!TryGetMapped(mappingIndex, out Link mappedLink))
1364         {
1365             throw new InvalidOperationException($"Mapped link with index {mappingIndex} is
1366                 ↪ not set.");
1367         }
1368         return mappedLink;
1369     }
1370
1371     /// <summary>
1372     /// <para>
1373     /// Gets the mapped or default using the specified mapping index.
1374     /// </para>
1375     /// <para></para>
1376     /// </summary>
1377     /// <param name="mappingIndex">
1378     /// <para>The mapping index.</para>
1379     /// <para></para>
1380     /// </param>
1381     /// <returns>
1382     /// <para>The mapped link.</para>
1383     /// <para></para>
1384     /// </returns>
1385     public static Link GetMappedOrDefault(object mappingIndex)
1386     {
1387         TryGetMapped(mappingIndex, out Link mappedLink);
1388         return mappedLink;
1389     }
1390
1391     /// <summary>
1392     /// <para>
1393     /// Gets the mapped or default using the specified mapping index.
1394     /// </para>
1395     /// <para></para>
1396     /// </summary>
1397     /// <param name="mappingIndex">
1398     /// <para>The mapping index.</para>
1399     /// <para></para>
1400     /// </param>
1401     /// <returns>
1402     /// <para>The mapped link.</para>
1403     /// <para></para>
1404     /// </returns>
1405     public static Link GetMappedOrDefault(Int mappingIndex)
1406     {
1407         TryGetMapped(mappingIndex, out Link mappedLink);
1408         return mappedLink;
1409     }
1410
1411     /// <summary>
1412     /// <para>
1413     /// Determines whether try get mapped.
1414     /// </para>
1415     /// <para></para>
1416     /// </summary>
1417     /// <param name="mappingIndex">
1418     /// <para>The mapping index.</para>
1419     /// <para></para>

```

```

1419     /// </param>
1420     /// <param name="mappedLink">
1421     /// <para>The mapped link.</para>
1422     /// <para></para>
1423     /// </param>
1424     /// <returns>
1425     /// <para>The bool</para>
1426     /// <para></para>
1427     /// </returns>
1428     public static bool TryGetMapped(object mappingIndex, out Link mappedLink) =>
        ↪ TryGetMapped(Convert.ToInt64(mappingIndex), out mappedLink);
1429
1430     /// <summary>
1431     /// <para>
1432     /// Determines whether try get mapped.
1433     /// </para>
1434     /// <para></para>
1435     /// </summary>
1436     /// <param name="mappingIndex">
1437     /// <para>The mapping index.</para>
1438     /// <para></para>
1439     /// </param>
1440     /// <param name="mappedLink">
1441     /// <para>The mapped link.</para>
1442     /// <para></para>
1443     /// </param>
1444     /// <exception cref="InvalidOperationException">
1445     /// <para>Менеджер памяти ещё не готов.</para>
1446     /// <para></para>
1447     /// </exception>
1448     /// <returns>
1449     /// <para>The bool</para>
1450     /// <para></para>
1451     /// </returns>
1452     public static bool TryGetMapped(Int mappingIndex, out Link mappedLink)
1453     {
1454         if (_memoryManagerIsReady == default)
1455         {
1456             throw new InvalidOperationException("Менеджер памяти ещё не готов.");
1457         }
1458         mappedLink = GetMappedLink(mappingIndex);
1459         if (mappedLink != null)
1460         {
1461             _linkToMappingIndex[mappedLink] = mappingIndex;
1462         }
1463         return mappedLink != null;
1464     }
1465
1466     /// <summary>
1467     /// <para>
1468     /// Updates the link.
1469     /// </para>
1470     /// <para></para>
1471     /// </summary>
1472     /// <param name="link">
1473     /// <para>The link.</para>
1474     /// <para></para>
1475     /// </param>
1476     /// <param name="newSource">
1477     /// <para>The new source.</para>
1478     /// <para></para>
1479     /// </param>
1480     /// <param name="newLinker">
1481     /// <para>The new linker.</para>
1482     /// <para></para>
1483     /// </param>
1484     /// <param name="newTarget">
1485     /// <para>The new target.</para>
1486     /// <para></para>
1487     /// </param>
1488     /// <returns>
1489     /// <para>The link.</para>
1490     /// <para></para>
1491     /// </returns>
1492     public static Link Update(Link link, Link newSource, Link newLinker, Link newTarget)
1493     {
1494         Update(ref link, newSource, newLinker, newTarget);
1495         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.",
1549                 ↪ nameof(link));
1550         }
1551         if (LinkWasDeleted(newSource))
1552         {
1553             throw new ArgumentException("Удалённая связь не может использоваться в качестве
1554                 ↪ нового значения.", nameof(newSource));
1555         }
1556         if (LinkWasDeleted(newLinker))
1557         {
1558             throw new ArgumentException("Удалённая связь не может использоваться в качестве
1559                 ↪ нового значения.", nameof(newLinker));
1560         }
1561         if (LinkWasDeleted(newTarget))
1562         {
1563             throw new ArgumentException("Удалённая связь не может использоваться в качестве
1564                 ↪ нового значения.", nameof(newTarget));
1565         }
1566         LinkIndex previousLinkIndex = link;
1567         _linkToMappingIndex.TryGetValue(link, out long mappingIndex);
1568         var previousDefinition = new LinkDefinition(link);
1569         link = UpdateLink(link, newSource, newLinker, newTarget);
1570         if (mappingIndex >= 0 && previousLinkIndex != link)
1571         {
1572             _linkToMappingIndex.Remove(previousLinkIndex);

```

```

1569         SetMappedLink(mappingIndex, link);
1570         _linkToMappingIndex.Add(link, mappingIndex);
1571     }
1572     UpdatedEvent(previousDefinition, new LinkDefinition(link));
1573 }
1574
1575 /// <summary>
1576 /// <para>
1577 /// Deletes the link.
1578 /// </para>
1579 /// <para></para>
1580 /// </summary>
1581 /// <param name="link">
1582 /// <para>The link.</para>
1583 /// <para></para>
1584 /// </param>
1585 public static void Delete(Link link) => Delete(ref link);
1586
1587 /// <summary>
1588 /// <para>
1589 /// Deletes the link.
1590 /// </para>
1591 /// <para></para>
1592 /// </summary>
1593 /// <param name="link">
1594 /// <para>The link.</para>
1595 /// <para></para>
1596 /// </param>
1597 public static void Delete(ref Link link)
1598 {
1599     if (LinkDoesNotExist(link))
1600     {
1601         return;
1602     }
1603     LinkIndex previousLinkIndex = link;
1604     _linkToMappingIndex.TryGetValue(link, out long mappingIndex);
1605     var previousDefinition = new LinkDefinition(link);
1606     DeleteLink(link);
1607     link = null;
1608     if (mappingIndex >= 0)
1609     {
1610         _linkToMappingIndex.Remove(previousLinkIndex);
1611         SetMappedLink(mappingIndex, 0);
1612     }
1613     DeletedEvent(previousDefinition);
1614 }
1615
1616 //public static void Replace(ref Link link, Link replacement)
1617 //{
1618 //    if (!MemoryManagerIsReady)
1619 //        throw new InvalidOperationException("Менеджер памяти ещё не готов.");
1620 //    if (LinkDoesNotExist(link))
1621 //        throw new InvalidOperationException("Если связь не существует, её нельзя
1622 //        → заменить.");
1623 //    if (LinkDoesNotExist(replacement))
1624 //        throw new ArgumentException("Пустая или удалённая связь не может быть
1625 //        → замещаемым значением.", "replacement");
1626 //    link = ReplaceLink(link, replacement);
1627 //}
1628
1629 /// <summary>
1630 /// <para>
1631 /// Searches the source.
1632 /// </para>
1633 /// <para></para>
1634 /// </summary>
1635 /// <param name="source">
1636 /// <para>The source.</para>
1637 /// <para></para>
1638 /// </param>
1639 /// <param name="linker">
1640 /// <para>The linker.</para>
1641 /// <para></para>
1642 /// </param>
1643 /// <param name="target">
1644 /// <para>The target.</para>
1645 /// <para></para>
1646 /// </param>

```

```

1645 /// <exception cref="InvalidOperationException">
1646 /// <para>Выполнить поиск связи можно только по существующим связям.</para>
1647 /// <para></para>
1648 /// </exception>
1649 /// <exception cref="InvalidOperationException">
1650 /// <para>Менеджер памяти ещё не готов.</para>
1651 /// <para></para>
1652 /// </exception>
1653 /// <returns>
1654 /// <para>The link</para>
1655 /// <para></para>
1656 /// </returns>
1657 public static Link Search(Link source, Link linker, Link target)
1658 {
1659     if (_memoryManagerIsReady == default)
1660     {
1661         throw new InvalidOperationException("Менеджер памяти ещё не готов.");
1662     }
1663     if (LinkDoesNotExist(source) || LinkDoesNotExist(linker) || LinkDoesNotExist(target))
1664     {
1665         throw new InvalidOperationException("Выполнить поиск связи можно только по
1666             ↳ существующим связям.");
1667     }
1668     return SearchLink(source, linker, target);
1669 }
1670 /// <summary>
1671 /// <para>
1672 /// Determines whether exists.
1673 /// </para>
1674 /// <para></para>
1675 /// </summary>
1676 /// <param name="source">
1677 /// <para>The source.</para>
1678 /// <para></para>
1679 /// </param>
1680 /// <param name="linker">
1681 /// <para>The linker.</para>
1682 /// <para></para>
1683 /// </param>
1684 /// <param name="target">
1685 /// <para>The target.</para>
1686 /// <para></para>
1687 /// </param>
1688 /// <returns>
1689 /// <para>The bool</para>
1690 /// <para></para>
1691 /// </returns>
1692 public static bool Exists(Link source, Link linker, Link target) => SearchLink(source,
1693     ↳ linker, target) != 0;
1694
1695 #endregion
1696 #region Referers Walkers
1697
1698 /// <summary>
1699 /// <para>
1700 /// Determines whether this instance walk through referers as source.
1701 /// </para>
1702 /// <para></para>
1703 /// </summary>
1704 /// <param name="walker">
1705 /// <para>The walker.</para>
1706 /// <para></para>
1707 /// </param>
1708 /// <exception cref="InvalidOperationException">
1709 /// <para>С несуществующей связью нельзя производить операции.</para>
1710 /// <para></para>
1711 /// </exception>
1712 /// <returns>
1713 /// <para>The bool</para>
1714 /// <para></para>
1715 /// </returns>
1716 public bool WalkThroughReferersAsSource(Func<Link, bool> walker)
1717 {
1718     if (LinkDoesNotExist(this))
1719     {

```

```

1720         throw new InvalidOperationException("С несуществующей связью нельзя
1721             ↪ производить операции.");
1722     }
1723     var referers = ReferersBySourceCount;
1724     if (referers == 1)
1725     {
1726         return walker(FirstRefererBySource);
1727     }
1728     else if (referers > 1)
1729     {
1730         return WalkThroughReferersBySource(this, x => walker(x) ? 1 : 0) != 0;
1731     }
1732     else
1733     {
1734         return true;
1735     }
1736 }
1737
1738 /// <summary>
1739 /// <para>
1740 /// Walks the through referers as source using the specified walker.
1741 /// </para>
1742 /// <para></para>
1743 /// </summary>
1744 /// <param name="walker">
1745 /// <para>The walker.</para>
1746 /// <para></para>
1747 /// </param>
1748 /// <exception cref="InvalidOperationException">
1749 /// <para>С несуществующей связью нельзя производить операции.</para>
1750 /// <para></para>
1751 /// </exception>
1752 public void WalkThroughReferersAsSource(Action<Link> walker)
1753 {
1754     if (LinkDoesNotExist(this))
1755     {
1756         throw new InvalidOperationException("С несуществующей связью нельзя
1757             ↪ производить операции.");
1758     }
1759     var referers = ReferersBySourceCount;
1760     if (referers == 1)
1761     {
1762         walker(FirstRefererBySource);
1763     }
1764     else if (referers > 1)
1765     {
1766         WalkThroughAllReferersBySource(this, x => walker(x));
1767     }
1768 }
1769
1770 /// <summary>
1771 /// <para>
1772 /// Determines whether this instance walk through referers as linker.
1773 /// </para>
1774 /// <para></para>
1775 /// </summary>
1776 /// <param name="walker">
1777 /// <para>The walker.</para>
1778 /// <para></para>
1779 /// </param>
1780 /// <exception cref="InvalidOperationException">
1781 /// <para>С несуществующей связью нельзя производить операции.</para>
1782 /// <para></para>
1783 /// </exception>
1784 /// <returns>
1785 /// <para>The bool</para>
1786 /// <para></para>
1787 /// </returns>
1788 public bool WalkThroughReferersAsLinker(Func<Link, bool> walker)
1789 {
1790     if (LinkDoesNotExist(this))
1791     {
1792         throw new InvalidOperationException("С несуществующей связью нельзя
1793             ↪ производить операции.");
1794     }
1795     var referers = ReferersByLinkerCount;
1796     if (referers == 1)
1797     {

```

```

1795         return walker(FirstRefererByLinker);
1796     }
1797     else if (referers > 1)
1798     {
1799         return WalkThroughReferersByLinker(this, x => walker(x) ? 1 : 0) != 0;
1800     }
1801     else
1802     {
1803         return true;
1804     }
1805 }
1806
1807 /// <summary>
1808 /// <para>
1809 /// Walks the through referers as linker using the specified walker.
1810 /// </para>
1811 /// <para></para>
1812 /// </summary>
1813 /// <param name="walker">
1814 /// <para>The walker.</para>
1815 /// <para></para>
1816 /// </param>
1817 /// <exception cref="InvalidOperationException">
1818 /// <para>С несуществующей связью нельзя производить операции.</para>
1819 /// <para></para>
1820 /// </exception>
1821 public void WalkThroughReferersAsLinker(Action<Link> walker)
1822 {
1823     if (LinkDoesNotExist(this))
1824     {
1825         throw new InvalidOperationException("С несуществующей связью нельзя
1826             ↪ производить операции.");
1827     }
1828     var referers = ReferersByLinkerCount;
1829     if (referers == 1)
1830     {
1831         walker(FirstRefererByLinker);
1832     }
1833     else if (referers > 1)
1834     {
1835         WalkThroughAllReferersByLinker(this, x => walker(x));
1836     }
1837 }
1838
1839 /// <summary>
1840 /// <para>
1841 /// Determines whether this instance walk through referers as target.
1842 /// </para>
1843 /// <para></para>
1844 /// </summary>
1845 /// <param name="walker">
1846 /// <para>The walker.</para>
1847 /// <para></para>
1848 /// </param>
1849 /// <exception cref="InvalidOperationException">
1850 /// <para>С несуществующей связью нельзя производить операции.</para>
1851 /// <para></para>
1852 /// </exception>
1853 /// <returns>
1854 /// <para>The bool</para>
1855 /// <para></para>
1856 /// </returns>
1857 public bool WalkThroughReferersAsTarget(Func<Link, bool> walker)
1858 {
1859     if (LinkDoesNotExist(this))
1860     {
1861         throw new InvalidOperationException("С несуществующей связью нельзя
1862             ↪ производить операции.");
1863     }
1864     var referers = ReferersByTargetCount;
1865     if (referers == 1)
1866     {
1867         return walker(FirstRefererByTarget);
1868     }
1869     else if (referers > 1)
1870     {
1871         return WalkThroughReferersByTarget(this, x => walker(x) ? 1 : 0) != 0;
1872     }

```

```

1871         else
1872         {
1873             return true;
1874         }
1875     }
1876
1877     /// <summary>
1878     /// <para>
1879     /// Walks the through referers as target using the specified walker.
1880     /// </para>
1881     /// <para></para>
1882     /// </summary>
1883     /// <param name="walker">
1884     /// <para>The walker.</para>
1885     /// <para></para>
1886     /// </param>
1887     /// <exception cref="InvalidOperationException">
1888     /// <para>С несуществующей связью нельзя производить операции.</para>
1889     /// <para></para>
1890     /// </exception>
1891     public void WalkThroughReferersAsTarget(Action<Link> walker)
1892     {
1893         if (LinkDoesNotExist(this))
1894         {
1895             throw new InvalidOperationException("С несуществующей связью нельзя
1896                 ↪ производить операции.");
1897         }
1898         var referers = ReferersByTargetCount;
1899         if (referers == 1)
1900         {
1901             walker(FirstRefererByTarget);
1902         }
1903         else if (referers > 1)
1904         {
1905             WalkThroughAllReferersByTarget(this, x => walker(x));
1906         }
1907
1908     /// <summary>
1909     /// <para>
1910     /// Walks the through referers using the specified walker.
1911     /// </para>
1912     /// <para></para>
1913     /// </summary>
1914     /// <param name="walker">
1915     /// <para>The walker.</para>
1916     /// <para></para>
1917     /// </param>
1918     /// <exception cref="InvalidOperationException">
1919     /// <para>С несуществующей связью нельзя производить операции.</para>
1920     /// <para></para>
1921     /// </exception>
1922     public void WalkThroughReferers(Action<Link> walker)
1923     {
1924         if (LinkDoesNotExist(this))
1925         {
1926             throw new InvalidOperationException("С несуществующей связью нельзя
1927                 ↪ производить операции.");
1928         }
1929         void wrapper(ulong x) => walker(x);
1930         WalkThroughAllReferersBySource(this, wrapper);
1931         WalkThroughAllReferersByLinker(this, wrapper);
1932         WalkThroughAllReferersByTarget(this, wrapper);
1933     }
1934
1935     /// <summary>
1936     /// <para>
1937     /// Walks the through referers using the specified walker.
1938     /// </para>
1939     /// <para></para>
1940     /// </summary>
1941     /// <param name="walker">
1942     /// <para>The walker.</para>
1943     /// <para></para>
1944     /// </param>
1945     /// <exception cref="InvalidOperationException">
1946     /// <para>С несуществующей связью нельзя производить операции.</para>
1947     /// <para></para>

```



```

1947     /// </exception>
1948     public void WalkThroughReferers(Func<Link, bool> walker)
1949     {
1950         if (LinkDoesNotExist(this))
1951         {
1952             throw new InvalidOperationException("С несуществующей связью нельзя
1953                 ↳ производить операции.");
1954         }
1955         long wrapper(ulong x) => walker(x) ? 1 : 0;
1956         WalkThroughReferersBySource(this, wrapper);
1957         WalkThroughReferersByLinker(this, wrapper);
1958         WalkThroughReferersByTarget(this, wrapper);
1959     }
1960     /// <summary>
1961     /// <para>
1962     /// Determines whether walk through all links.
1963     /// </para>
1964     /// <para></para>
1965     /// </summary>
1966     /// <param name="walker">
1967     /// <para>The walker.</para>
1968     /// <para></para>
1969     /// </param>
1970     /// <returns>
1971     /// <para>The bool</para>
1972     /// <para></para>
1973     /// </returns>
1974     public static bool WalkThroughAllLinks(Func<Link, bool> walker) => WalkThroughLinks(x =>
1975         ↳ walker(x) ? 1 : 0) != 0;
1976     /// <summary>
1977     /// <para>
1978     /// Walks the through all links using the specified walker.
1979     /// </para>
1980     /// <para></para>
1981     /// </summary>
1982     /// <param name="walker">
1983     /// <para>The walker.</para>
1984     /// <para></para>
1985     /// </param>
1986     public static void WalkThroughAllLinks(Action<Link> walker) => WalkThroughAllLinks(new
1987         ↳ Visitor(x => walker(x)));
1988     #endregion
1989 }
1990 }

```

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

```

1  using System;
2  using System.Collections.Generic;
3  using Platform.Data.Sequences;
4
5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7  namespace Platform.Data.Triplets
8  {
9      /// <summary>
10     /// <para>
11     /// Represents the link converter.
12     /// </para>
13     /// <para></para>
14     /// </summary>
15     public static class LinkConverter
16     {
17         /// <summary>
18         /// <para>
19         /// Creates the list using the specified links.
20         /// </para>
21         /// <para></para>
22         /// </summary>
23         /// <param name="links">
24         /// <para>The links.</para>
25         /// <para></para>
26         /// </param>
27         /// <returns>
28         /// <para>The element.</para>
29         /// <para></para>

```

```

30     /// </returns>
31 public static Link FromList(List<Link> links)
32 {
33     var i = links.Count - 1;
34     var element = links[i];
35     while (--i >= 0)
36     {
37         element = links[i] & element;
38     }
39     return element;
40 }
41
42 /// <summary>
43 /// <para>
44 /// Creates the list using the specified links.
45 /// </para>
46 /// <para></para>
47 /// </summary>
48 /// <param name="links">
49 /// <para>The links.</para>
50 /// <para></para>
51 /// </param>
52 /// <returns>
53 /// <para>The element.</para>
54 /// <para></para>
55 /// </returns>
56 public static Link FromList(Link[] links)
57 {
58     var i = links.Length - 1;
59     var element = links[i];
60     while (--i >= 0)
61     {
62         element = links[i] & element;
63     }
64     return element;
65 }
66
67 /// <summary>
68 /// <para>
69 /// Returns the list using the specified link.
70 /// </para>
71 /// <para></para>
72 /// </summary>
73 /// <param name="link">
74 /// <para>The link.</para>
75 /// <para></para>
76 /// </param>
77 /// <returns>
78 /// <para>The list.</para>
79 /// <para></para>
80 /// </returns>
81 public static List<Link> ToList(Link link)
82 {
83     var list = new List<Link>();
84     SequenceWalker.WalkRight(link, x => x.Source, x => x.Target, x => x.Linker !=
85         ↪ Net.And, list.Add);
86     return list;
87 }
88
89 /// <summary>
90 /// <para>
91 /// Creates the number using the specified number.
92 /// </para>
93 /// <para></para>
94 /// </summary>
95 /// <param name="number">
96 /// <para>The number.</para>
97 /// <para></para>
98 /// </param>
99 /// <returns>
100 /// <para>The link</para>
101 /// <para></para>
102 /// </returns>
103 public static Link FromNumber(long number) => NumberHelpers.FromNumber(number);
104
105 /// <summary>
106 /// <para>
107 /// Returns the number using the specified number.

```

```

107     /// </para>
108     /// <para></para>
109     /// </summary>
110     /// <param name="number">
111     /// <para>The number.</para>
112     /// <para></para>
113     /// </param>
114     /// <returns>
115     /// <para>The long</para>
116     /// <para></para>
117     /// </returns>
118     public static long ToNumber(Link number) => NumberHelpers.ToNumber(number);
119
120     /// <summary>
121     /// <para>
122     /// Creates the char using the specified c.
123     /// </para>
124     /// <para></para>
125     /// </summary>
126     /// <param name="c">
127     /// <para>The .</para>
128     /// <para></para>
129     /// </param>
130     /// <returns>
131     /// <para>The link</para>
132     /// <para></para>
133     /// </returns>
134     public static Link FromChar(char c) => CharacterHelpers.FromChar(c);
135
136     /// <summary>
137     /// <para>
138     /// Returns the char using the specified char link.
139     /// </para>
140     /// <para></para>
141     /// </summary>
142     /// <param name="charLink">
143     /// <para>The char link.</para>
144     /// <para></para>
145     /// </param>
146     /// <returns>
147     /// <para>The char</para>
148     /// <para></para>
149     /// </returns>
150     public static char ToChar(Link charLink) => CharacterHelpers.ToChar(charLink);
151
152     /// <summary>
153     /// <para>
154     /// Creates the chars using the specified chars.
155     /// </para>
156     /// <para></para>
157     /// </summary>
158     /// <param name="chars">
159     /// <para>The chars.</para>
160     /// <para></para>
161     /// </param>
162     /// <returns>
163     /// <para>The link</para>
164     /// <para></para>
165     /// </returns>
166     public static Link FromChars(char[] chars) => FromObjectsToSequence(chars, FromChar);
167
168     /// <summary>
169     /// <para>
170     /// Creates the chars using the specified chars.
171     /// </para>
172     /// <para></para>
173     /// </summary>
174     /// <param name="chars">
175     /// <para>The chars.</para>
176     /// <para></para>
177     /// </param>
178     /// <param name="takeFrom">
179     /// <para>The take from.</para>
180     /// <para></para>
181     /// </param>
182     /// <param name="takeUntil">
183     /// <para>The take until.</para>
184     /// <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) =>
191         ↪ FromObjectsToSequence(chars, takeFrom, takeUntil, FromChar);
192
193     /// <summary>
194     /// <para>
195     /// Creates the numbers using the specified numbers.
196     /// </para>
197     /// <para></para>
198     /// </summary>
199     /// <param name="numbers">
200     /// <para>The numbers.</para>
201     /// <para></para>
202     /// </param>
203     /// <returns>
204     /// <para>The link</para>
205     /// <para></para>
206     /// </returns>
207     public static Link FromNumbers(long[] numbers) => FromObjectsToSequence(numbers,
208         ↪ FromNumber);
209
210     /// <summary>
211     /// <para>
212     /// Creates the numbers using the specified numbers.
213     /// </para>
214     /// <para></para>
215     /// </summary>
216     /// <param name="numbers">
217     /// <para>The numbers.</para>
218     /// <para></para>
219     /// </param>
220     /// <param name="takeFrom">
221     /// <para>The take from.</para>
222     /// <para></para>
223     /// </param>
224     /// <param name="takeUntil">
225     /// <para>The take until.</para>
226     /// <para></para>
227     /// </param>
228     /// <returns>
229     /// <para>The link</para>
230     /// <para></para>
231     /// </returns>
232     public static Link FromNumbers(long[] numbers, int takeFrom, int takeUntil) =>
233         ↪ FromObjectsToSequence(numbers, takeFrom, takeUntil, FromNumber);
234
235     /// <summary>
236     /// <para>
237     /// Creates the numbers using the specified numbers.
238     /// </para>
239     /// <para></para>
240     /// </summary>
241     /// <param name="numbers">
242     /// <para>The numbers.</para>
243     /// <para></para>
244     /// </param>
245     /// <returns>
246     /// <para>The link</para>
247     /// <para></para>
248     /// </returns>
249     public static Link FromNumbers(ushort[] numbers) => FromObjectsToSequence(numbers, x =>
250         ↪ FromNumber(x));
251
252     /// <summary>
253     /// <para>
254     /// Creates the numbers using the specified numbers.
255     /// </para>
256     /// <para></para>
257     /// </summary>
258     /// <param name="numbers">
259     /// <para>The numbers.</para>
260     /// <para></para>
261     /// </param>
262     /// <param name="takeFrom">

```

```

259    /// <para>The take from.</para>
260    /// <para></para>
261    /// </param>
262    /// <param name="takeUntil">
263    /// <para>The take until.</para>
264    /// <para></para>
265    /// </param>
266    /// <returns>
267    /// <para>The link</para>
268    /// <para></para>
269    /// </returns>
270    public static Link FromNumbers(ushort[] numbers, int takeFrom, int takeUntil) =>
        ↪ FromObjectsToSequence(numbers, takeFrom, takeUntil, x => FromNumber(x));

271    /// <summary>
272    /// <para>
273    /// Creates the numbers using the specified numbers.
274    /// </para>
275    /// <para></para>
276    /// </summary>
277    /// <param name="numbers">
278    /// <para>The numbers.</para>
279    /// <para></para>
280    /// </param>
281    /// <returns>
282    /// <para>The link</para>
283    /// <para></para>
284    /// </returns>
285    public static Link FromNumbers(uint[] numbers) => FromObjectsToSequence(numbers, x =>
        ↪ FromNumber(x));

287    /// <summary>
288    /// <para>
289    /// Creates the numbers using the specified numbers.
290    /// </para>
291    /// <para></para>
292    /// </summary>
293    /// <param name="numbers">
294    /// <para>The numbers.</para>
295    /// <para></para>
296    /// </param>
297    /// <param name="takeFrom">
298    /// <para>The take from.</para>
299    /// <para></para>
300    /// </param>
301    /// <param name="takeUntil">
302    /// <para>The take until.</para>
303    /// <para></para>
304    /// </param>
305    /// <returns>
306    /// <para>The link</para>
307    /// <para></para>
308    /// </returns>
309    public static Link FromNumbers(uint[] numbers, int takeFrom, int takeUntil) =>
        ↪ FromObjectsToSequence(numbers, takeFrom, takeUntil, x => FromNumber(x));

311    /// <summary>
312    /// <para>
313    /// Creates the numbers using the specified numbers.
314    /// </para>
315    /// <para></para>
316    /// </summary>
317    /// <param name="numbers">
318    /// <para>The numbers.</para>
319    /// <para></para>
320    /// </param>
321    /// <returns>
322    /// <para>The link</para>
323    /// <para></para>
324    /// </returns>
325    public static Link FromNumbers(byte[] numbers) => FromObjectsToSequence(numbers, x =>
        ↪ FromNumber(x));

327    /// <summary>
328    /// <para>
329    /// Creates the numbers using the specified numbers.
330    /// </para>
331    /// <para></para>
332    /// </summary>

```

```

333     /// </summary>
334     /// <param name="numbers">
335     /// <para>The numbers.</para>
336     /// <para></para>
337     /// </param>
338     /// <param name="takeFrom">
339     /// <para>The take from.</para>
340     /// <para></para>
341     /// </param>
342     /// <param name="takeUntil">
343     /// <para>The take until.</para>
344     /// <para></para>
345     /// </param>
346     /// <returns>
347     /// <para>The link</para>
348     /// <para></para>
349     /// </returns>
350     public static Link FromNumbers(byte[] numbers, int takeFrom, int takeUntil) =>
        ↪ FromObjectsToSequence(numbers, takeFrom, takeUntil, x => FromNumber(x));
351
352     /// <summary>
353     /// <para>
354     /// Creates the numbers using the specified numbers.
355     /// </para>
356     /// <para></para>
357     /// </summary>
358     /// <param name="numbers">
359     /// <para>The numbers.</para>
360     /// <para></para>
361     /// </param>
362     /// <returns>
363     /// <para>The link</para>
364     /// <para></para>
365     /// </returns>
366     public static Link FromNumbers(bool[] numbers) => FromObjectsToSequence(numbers, x =>
        ↪ FromNumber(x ? 1 : 0));
367
368     /// <summary>
369     /// <para>
370     /// Creates the numbers using the specified numbers.
371     /// </para>
372     /// <para></para>
373     /// </summary>
374     /// <param name="numbers">
375     /// <para>The numbers.</para>
376     /// <para></para>
377     /// </param>
378     /// <param name="takeFrom">
379     /// <para>The take from.</para>
380     /// <para></para>
381     /// </param>
382     /// <param name="takeUntil">
383     /// <para>The take until.</para>
384     /// <para></para>
385     /// </param>
386     /// <returns>
387     /// <para>The link</para>
388     /// <para></para>
389     /// </returns>
390     public static Link FromNumbers(bool[] numbers, int takeFrom, int takeUntil) =>
        ↪ FromObjectsToSequence(numbers, takeFrom, takeUntil, x => FromNumber(x ? 1 : 0));
391
392     /// <summary>
393     /// <para>
394     /// Creates the objects to sequence using the specified objects.
395     /// </para>
396     /// <para></para>
397     /// </summary>
398     /// <typeparam name="T">
399     /// <para>The .</para>
400     /// <para></para>
401     /// </typeparam>
402     /// <param name="objects">
403     /// <para>The objects.</para>
404     /// <para></para>
405     /// </param>
406     /// <param name="converter">
407     /// <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) =>
415         ↪ FromObjectsToSequence(objects, 0, objects.Length, converter);
416
417     /// <summary>
418     /// <para>
419     /// Creates the objects to sequence using the specified objects.
420     /// </para>
421     /// <para></para>
422     /// </summary>
423     /// <typeparam name="T">
424     /// <para>The .</para>
425     /// <para></para>
426     /// </typeparam>
427     /// <param name="objects">
428     /// <para>The objects.</para>
429     /// <para></para>
430     /// </param>
431     /// <param name="takeFrom">
432     /// <para>The take from.</para>
433     /// <para></para>
434     /// </param>
435     /// <param name="takeUntil">
436     /// <para>The take until.</para>
437     /// <para></para>
438     /// </param>
439     /// <param name="converter">
440     /// <para>The converter.</para>
441     /// <para></para>
442     /// </param>
443     /// <exception cref="ArgumentOutOfRangeException">
444     /// <para>Нельзя преобразовать пустой список к связям.</para>
445     /// <para></para>
446     /// </exception>
447     /// <returns>
448     /// <para>The link</para>
449     /// <para></para>
450     /// </returns>
451     public static Link FromObjectsToSequence<T>(T[] objects, int takeFrom, int takeUntil,
452         ↪ Func<T, Link> converter)
453     {
454         var length = takeUntil - takeFrom;
455         if (length <= 0)
456         {
457             throw new ArgumentOutOfRangeException(nameof(takeUntil), "Нельзя преобразовать
458                 ↪ пустой список к связям.");
459         }
460         var copy = new Link[length];
461         for (int i = takeFrom, j = 0; i < takeUntil; i++, j++)
462         {
463             copy[j] = converter(objects[i]);
464         }
465         return FromList(copy);
466     }
467
468     /// <summary>
469     /// <para>
470     /// Creates the chars using the specified str.
471     /// </para>
472     /// <para></para>
473     /// </summary>
474     /// <param name="str">
475     /// <para>The str.</para>
476     /// <para></para>
477     /// </param>
478     /// <returns>
479     /// <para>The link</para>
480     /// <para></para>
481     /// </returns>
482     public static Link FromChars(string str)
483     {
484         var copy = new Link[str.Length];
485         for (var i = 0; i < copy.Length; i++)

```

```

483     {
484         copy[i] = FromChar(str[i]);
485     }
486     return FromList(copy);
487 }
488
489 /// <summary>
490 /// <para>
491 /// Creates the string using the specified str.
492 /// </para>
493 /// <para></para>
494 /// </summary>
495 /// <param name="str">
496 /// <para>The str.</para>
497 /// <para></para>
498 /// </param>
499 /// <returns>
500 /// <para>The str link.</para>
501 /// <para></para>
502 /// </returns>
503 public static Link FromString(string str)
504 {
505     var copy = new Link[str.Length];
506     for (var i = 0; i < copy.Length; i++)
507     {
508         copy[i] = FromChar(str[i]);
509     }
510     var strLink = Link.Create(Net.String, Net.ThatConsistsOf, FromList(copy));
511     return strLink;
512 }
513
514 /// <summary>
515 /// <para>
516 /// Returns the string using the specified link.
517 /// </para>
518 /// <para></para>
519 /// </summary>
520 /// <param name="link">
521 /// <para>The link.</para>
522 /// <para></para>
523 /// </param>
524 /// <exception cref="ArgumentOutOfRangeException">
525 /// <para>Specified link is not a string.</para>
526 /// <para></para>
527 /// </exception>
528 /// <returns>
529 /// <para>The string</para>
530 /// <para></para>
531 /// </returns>
532 public static string ToString(Link link)
533 {
534     if (link.IsString())
535     {
536         return ToString(ToList(link.Target));
537     }
538     throw new ArgumentOutOfRangeException(nameof(link), "Specified link is not a
539         ↪ string.");
540 }
541
542 /// <summary>
543 /// <para>
544 /// Returns the string using the specified char links.
545 /// </para>
546 /// <para></para>
547 /// </summary>
548 /// <param name="charLinks">
549 /// <para>The char links.</para>
550 /// <para></para>
551 /// </param>
552 /// <returns>
553 /// <para>The string</para>
554 /// <para></para>
555 /// </returns>
556 public static string ToString(List<Link> charLinks)
557 {
558     var chars = new char[charLinks.Count];
559     for (var i = 0; i < charLinks.Count; i++)
560     {

```



```

560         chars[i] = ToChar(charLinks[i]);
561     }
562     return new string(chars);
563 }
564 }
565 }

```

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

```

1  using System;
2  using System.Collections.Generic;
3  using System.Text;
4  using Platform.Data.Sequences;
5  using Platform.Data.Triplets.Sequences;
6
7  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
8
9  namespace Platform.Data.Triplets
10 {
11     /// <summary>
12     /// <para>
13     /// Represents the link extensions.
14     /// </para>
15     /// <para></para>
16     /// </summary>
17     public static class LinkExtensions
18     {
19         /// <summary>
20         /// <para>
21         /// Sets the name using the specified link.
22         /// </para>
23         /// <para></para>
24         /// </summary>
25         /// <param name="link">
26         /// <para>The link.</para>
27         /// <para></para>
28         /// </param>
29         /// <param name="name">
30         /// <para>The name.</para>
31         /// <para></para>
32         /// </param>
33         /// <returns>
34         /// <para>The link.</para>
35         /// <para></para>
36         /// </returns>
37         public static Link SetName(this Link link, string name)
38         {
39             Link.Create(link, Net.Has, Link.Create(Net.Name, Net.ThatIsRepresentedBy,
40             ↪ LinkConverter.FromString(name)));
41             return link; // Chaining
42         }
43
44         /// <summary>
45         /// <para>
46         /// The link.
47         /// </para>
48         /// <para></para>
49         /// </summary>
50         private static readonly HashSet<Link> _linksWithNamesGatheringProcess = new
51         ↪ HashSet<Link>();
52
53         /// <summary>
54         /// <para>
55         /// Determines whether try get name.
56         /// </para>
57         /// <para></para>
58         /// </summary>
59         /// <param name="link">
60         /// <para>The link.</para>
61         /// <para></para>
62         /// </param>
63         /// <param name="str">
64         /// <para>The str.</para>
65         /// <para></para>
66         /// </param>
67         /// <returns>
68         /// <para>The bool</para>
69         /// <para></para>
70         /// </returns>

```

```

69 public static bool TryGetName(this Link link, out string str)
70 {
71     // Защита от заикливания
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) &&
107                 ↪ link.Linker.TryGetName(out string linkerName) &&
108                 ↪ link.Target.TryGetName(out string targetName))
109             {
110                 var sb = new StringBuilder();
111                 sb.Append(sourceName).Append(' ').Append(linkerName).Append(' ')
112                 ↪ .Append(targetName);
113                 str = sb.ToString();
114                 return true;
115             }
116             str = null;
117             return false;
118         }
119     }
120     finally
121     {
122         _linksWithNamesGatheringProcess.Remove(link);
123     }
124 }
125
126 /// <summary>
127 /// <para>
128 /// Determines whether try get specific name.
129 /// </para>
130 /// <para></para>
131 /// </summary>
132 /// <param name="link">
133 /// <para>The link.</para>
134 /// <para></para>
135 /// </param>
136 /// <param name="name">
137 /// <para>The name.</para>
138 /// <para></para>
139 /// </param>
140 /// <returns>
141 /// <para>The bool</para>
142 /// <para></para>
143 /// </returns>
144 public static bool TryGetSpecificName(this Link link, out string name)
145 {
146     string nameLocal = null;
147     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 ==
167                 ↪ Net.ThatIsRepresentedBy)
168             {
169                 nameLocal = LinkConverter.ToString(nameLink.Target);
170                 return false; // Останавливаем проход
171             }
172             return true;
173         });
174     }
175
176     name = nameLocal;
177     return nameLocal != null;
178 }

```

```

180 // Проверка на принадлежность 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
216 ↪ значений char будет символом.
217 // Нужно изменить определение чара, идеально: char consists of sum of [8, 64].
218 /// <summary>
219 /// <para>
220 /// Determines whether is char.
221 /// </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
248     ↪ && link.Linker == Net.ThatConsistsOf;
249
250     /// <summary>
251     /// <para>
252     /// Determines whether is sum.
253     /// </para>
254     /// <para></para>
255     /// </summary>
256     /// <param name="link">
257     /// <para>The link.</para>
258     /// <para></para>
259     /// </param>
260     /// <returns>
261     /// <para>The bool</para>
262     /// <para></para>
263     /// </returns>
264     public static bool IsSum(this Link link) => link != null && link.Source == Net.Sum &&
265     ↪ link.Linker == Net.Of;
266
267     /// <summary>
268     /// <para>
269     /// Determines whether is string.
270     /// </para>
271     /// <para></para>
272     /// </summary>
273     /// <param name="link">
274     /// <para>The link.</para>
275     /// <para></para>
276     /// </param>
277     /// <returns>
278     /// <para>The bool</para>
279     /// <para></para>
280     /// </returns>
281     public static bool IsString(this Link link) => link != null && link.Source == Net.String
282     ↪ && link.Linker == Net.ThatConsistsOf;
283
284     /// <summary>
285     /// <para>
286     /// Determines whether is name.
287     /// </para>
288     /// <para></para>
289     /// </summary>
290     /// <param name="link">
291     /// <para>The link.</para>
292     /// <para></para>
293     /// </param>
294     /// <returns>
295     /// <para>The bool</para>
296     /// <para></para>
297     /// </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[] GetArrayOfRererersBySource(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[] GetArrayOfRererersByLinker(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[] GetArrayOfRererersByTarget(this Link link)
370 {
371     if (link == null)

```

```

372     {
373         return new Link[0];
374     }
375     else
376     {
377         var array = new Link[link.ReferersByTargetCount];
378         var index = 0;
379         link.WalkThroughReferersAsTarget(referer => array[index++] = referer);
380         return array;
381     }
382 }
383
384 /// <summary>
385 /// <para>
386 /// Walks the through sequence using the specified link.
387 /// </para>
388 /// <para></para>
389 /// </summary>
390 /// <param name="link">
391 /// <para>The link.</para>
392 /// <para></para>
393 /// </param>
394 /// <param name="action">
395 /// <para>The action.</para>
396 /// <para></para>
397 /// </param>
398 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);
399 }
400 }

```

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

```

1 using Platform.Threading;
2
3 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5 namespace Platform.Data.Triplets
6 {
7     /// <summary>
8     /// <para>
9     /// The net mapping enum.
10    /// </para>
11    /// <para></para>
12    /// </summary>
13    public enum NetMapping : long
14    {
15        /// <summary>
16        /// <para>
17        /// The link net mapping.
18        /// </para>
19        /// <para></para>
20        /// </summary>
21        Link,
22        /// <summary>
23        /// <para>
24        /// The thing net mapping.
25        /// </para>
26        /// <para></para>
27        /// </summary>
28        Thing,
29        /// <summary>
30        /// <para>
31        /// The is net mapping.
32        /// </para>
33        /// <para></para>
34        /// </summary>
35        IsA,
36        /// <summary>
37        /// <para>
38        /// The is not net mapping.
39        /// </para>
40        /// <para></para>
41        /// </summary>
42        IsNotA,
43
44        /// <summary>
45        /// <para>
46        /// The of net mapping.

```

```

47     /// </para>
48     /// <para></para>
49     /// </summary>
50     Of,
51     /// <summary>
52     /// <para>
53     /// The and net mapping.
54     /// </para>
55     /// <para></para>
56     /// </summary>
57     And,
58     /// <summary>
59     /// <para>
60     /// The that consists of net mapping.
61     /// </para>
62     /// <para></para>
63     /// </summary>
64     ThatConsistsOf,
65     /// <summary>
66     /// <para>
67     /// The has net mapping.
68     /// </para>
69     /// <para></para>
70     /// </summary>
71     Has,
72     /// <summary>
73     /// <para>
74     /// The contains net mapping.
75     /// </para>
76     /// <para></para>
77     /// </summary>
78     Contains,
79     /// <summary>
80     /// <para>
81     /// The contained by net mapping.
82     /// </para>
83     /// <para></para>
84     /// </summary>
85     ContainedBy,
86
87     /// <summary>
88     /// <para>
89     /// The one net mapping.
90     /// </para>
91     /// <para></para>
92     /// </summary>
93     One,
94     /// <summary>
95     /// <para>
96     /// The zero net mapping.
97     /// </para>
98     /// <para></para>
99     /// </summary>
100    Zero,
101
102    /// <summary>
103    /// <para>
104    /// The sum net mapping.
105    /// </para>
106    /// <para></para>
107    /// </summary>
108    Sum,
109    /// <summary>
110    /// <para>
111    /// The character net mapping.
112    /// </para>
113    /// <para></para>
114    /// </summary>
115    Character,
116    /// <summary>
117    /// <para>
118    /// The string net mapping.
119    /// </para>
120    /// <para></para>
121    /// </summary>
122    String,
123    /// <summary>
124    /// <para>
125    /// The name net mapping.

```

```

126     /// </para>
127     /// <para></para>
128     /// </summary>
129     Name,
130
131     /// <summary>
132     /// <para>
133     /// The set net mapping.
134     /// </para>
135     /// <para></para>
136     /// </summary>
137     Set,
138     /// <summary>
139     /// <para>
140     /// The group net mapping.
141     /// </para>
142     /// <para></para>
143     /// </summary>
144     Group,
145
146     /// <summary>
147     /// <para>
148     /// The parsed from net mapping.
149     /// </para>
150     /// <para></para>
151     /// </summary>
152     ParsedFrom,
153     /// <summary>
154     /// <para>
155     /// The that is net mapping.
156     /// </para>
157     /// <para></para>
158     /// </summary>
159     ThatIs,
160     /// <summary>
161     /// <para>
162     /// The that is before net mapping.
163     /// </para>
164     /// <para></para>
165     /// </summary>
166     ThatIsBefore,
167     /// <summary>
168     /// <para>
169     /// The that is between net mapping.
170     /// </para>
171     /// <para></para>
172     /// </summary>
173     ThatIsBetween,
174     /// <summary>
175     /// <para>
176     /// The that is after net mapping.
177     /// </para>
178     /// <para></para>
179     /// </summary>
180     ThatIsAfter,
181     /// <summary>
182     /// <para>
183     /// The that is represented by net mapping.
184     /// </para>
185     /// <para></para>
186     /// </summary>
187     ThatIsRepresentedBy,
188     /// <summary>
189     /// <para>
190     /// The that has net mapping.
191     /// </para>
192     /// <para></para>
193     /// </summary>
194     ThatHas,
195
196     /// <summary>
197     /// <para>
198     /// The text net mapping.
199     /// </para>
200     /// <para></para>
201     /// </summary>
202     Text,
203     /// <summary>
204     /// <para>

```



```
205     /// The path net mapping.
206     /// </para>
207     /// <para></para>
208     /// </summary>
209     Path,
210     /// <summary>
211     /// <para>
212     /// The content net mapping.
213     /// </para>
214     /// <para></para>
215     /// </summary>
216     Content,
217     /// <summary>
218     /// <para>
219     /// The empty content net mapping.
220     /// </para>
221     /// <para></para>
222     /// </summary>
223     EmptyContent,
224     /// <summary>
225     /// <para>
226     /// The empty net mapping.
227     /// </para>
228     /// <para></para>
229     /// </summary>
230     Empty,
231     /// <summary>
232     /// <para>
233     /// The alphabet net mapping.
234     /// </para>
235     /// <para></para>
236     /// </summary>
237     Alphabet,
238     /// <summary>
239     /// <para>
240     /// The letter net mapping.
241     /// </para>
242     /// <para></para>
243     /// </summary>
244     Letter,
245     /// <summary>
246     /// <para>
247     /// The case net mapping.
248     /// </para>
249     /// <para></para>
250     /// </summary>
251     Case,
252     /// <summary>
253     /// <para>
254     /// The upper net mapping.
255     /// </para>
256     /// <para></para>
257     /// </summary>
258     Upper,
259     /// <summary>
260     /// <para>
261     /// The upper case net mapping.
262     /// </para>
263     /// <para></para>
264     /// </summary>
265     UpperCase,
266     /// <summary>
267     /// <para>
268     /// The lower net mapping.
269     /// </para>
270     /// <para></para>
271     /// </summary>
272     Lower,
273     /// <summary>
274     /// <para>
275     /// The lower case net mapping.
276     /// </para>
277     /// <para></para>
278     /// </summary>
279     LowerCase,
280     /// <summary>
281     /// <para>
282     /// 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; }

```

```

361     /// <summary>
362     /// <para>
363     /// Gets or sets the contained by value.
364     /// </para>
365     /// <para></para>
366     /// </summary>
367     public static Link ContainedBy { get; private set; }
368
369     /// <summary>
370     /// <para>
371     /// Gets or sets the one value.
372     /// </para>
373     /// <para></para>
374     /// </summary>
375     public static Link One { get; private set; }
376     /// <summary>
377     /// <para>
378     /// Gets or sets the zero value.
379     /// </para>
380     /// <para></para>
381     /// </summary>
382     public static Link Zero { get; private set; }
383
384     /// <summary>
385     /// <para>
386     /// Gets or sets the sum value.
387     /// </para>
388     /// <para></para>
389     /// </summary>
390     public static Link Sum { get; private set; }
391     /// <summary>
392     /// <para>
393     /// Gets or sets the character value.
394     /// </para>
395     /// <para></para>
396     /// </summary>
397     public static Link Character { get; private set; }
398     /// <summary>
399     /// <para>
400     /// Gets or sets the string value.
401     /// </para>
402     /// <para></para>
403     /// </summary>
404     public static Link String { get; private set; }
405     /// <summary>
406     /// <para>
407     /// Gets or sets the name value.
408     /// </para>
409     /// <para></para>
410     /// </summary>
411     public static Link Name { get; private set; }
412
413     /// <summary>
414     /// <para>
415     /// Gets or sets the set value.
416     /// </para>
417     /// <para></para>
418     /// </summary>
419     public static Link Set { get; private set; }
420     /// <summary>
421     /// <para>
422     /// Gets or sets the group value.
423     /// </para>
424     /// <para></para>
425     /// </summary>
426     public static Link Group { get; private set; }
427
428     /// <summary>
429     /// <para>
430     /// Gets or sets the parsed from value.
431     /// </para>
432     /// <para></para>
433     /// </summary>
434     public static Link ParsedFrom { get; private set; }
435     /// <summary>
436     /// <para>
437     /// Gets or sets the that is value.
438     /// </para>

```

```

439     /// <para></para>
440     /// </summary>
441     public static Link ThatIs { get; private set; }
442     /// <summary>
443     /// <para>
444     /// Gets or sets the that is before value.
445     /// </para>
446     /// <para></para>
447     /// </summary>
448     public static Link ThatIsBefore { get; private set; }
449     /// <summary>
450     /// <para>
451     /// Gets or sets the that is between value.
452     /// </para>
453     /// <para></para>
454     /// </summary>
455     public static Link ThatIsBetween { get; private set; }
456     /// <summary>
457     /// <para>
458     /// Gets or sets the that is after value.
459     /// </para>
460     /// <para></para>
461     /// </summary>
462     public static Link ThatIsAfter { get; private set; }
463     /// <summary>
464     /// <para>
465     /// Gets or sets the that is represented by value.
466     /// </para>
467     /// <para></para>
468     /// </summary>
469     public static Link ThatIsRepresentedBy { get; private set; }
470     /// <summary>
471     /// <para>
472     /// Gets or sets the that has value.
473     /// </para>
474     /// <para></para>
475     /// </summary>
476     public static Link ThatHas { get; private set; }
477
478     /// <summary>
479     /// <para>
480     /// Gets or sets the text value.
481     /// </para>
482     /// <para></para>
483     /// </summary>
484     public static Link Text { get; private set; }
485     /// <summary>
486     /// <para>
487     /// Gets or sets the path value.
488     /// </para>
489     /// <para></para>
490     /// </summary>
491     public static Link Path { get; private set; }
492     /// <summary>
493     /// <para>
494     /// Gets or sets the content value.
495     /// </para>
496     /// <para></para>
497     /// </summary>
498     public static Link Content { get; private set; }
499     /// <summary>
500     /// <para>
501     /// Gets or sets the empty content value.
502     /// </para>
503     /// <para></para>
504     /// </summary>
505     public static Link EmptyContent { get; private set; }
506     /// <summary>
507     /// <para>
508     /// Gets or sets the empty value.
509     /// </para>
510     /// <para></para>
511     /// </summary>
512     public static Link Empty { get; private set; }
513     /// <summary>
514     /// <para>
515     /// Gets or sets the alphabet value.
516     /// </para>

```

```

517     /// <para></para>
518     /// </summary>
519     public static Link Alphabet { get; private set; }
520     /// <summary>
521     /// <para>
522     /// Gets or sets the letter value.
523     /// </para>
524     /// <para></para>
525     /// </summary>
526     public static Link Letter { get; private set; }
527     /// <summary>
528     /// <para>
529     /// Gets or sets the case value.
530     /// </para>
531     /// <para></para>
532     /// </summary>
533     public static Link Case { get; private set; }
534     /// <summary>
535     /// <para>
536     /// Gets or sets the upper value.
537     /// </para>
538     /// <para></para>
539     /// </summary>
540     public static Link Upper { get; private set; }
541     /// <summary>
542     /// <para>
543     /// Gets or sets the upper case value.
544     /// </para>
545     /// <para></para>
546     /// </summary>
547     public static Link UpperCase { get; private set; }
548     /// <summary>
549     /// <para>
550     /// Gets or sets the lower value.
551     /// </para>
552     /// <para></para>
553     /// </summary>
554     public static Link Lower { get; private set; }
555     /// <summary>
556     /// <para>
557     /// Gets or sets the lower case value.
558     /// </para>
559     /// <para></para>
560     /// </summary>
561     public static Link LowerCase { get; private set; }
562     /// <summary>
563     /// <para>
564     /// Gets or sets the code value.
565     /// </para>
566     /// <para></para>
567     /// </summary>
568     public static Link Code { get; private set; }
569
570     /// <summary>
571     /// <para>
572     /// Initializes a new <see cref="Net"/> instance.
573     /// </para>
574     /// <para></para>
575     /// </summary>
576     static Net() => Create();
577
578     /// <summary>
579     /// <para>
580     /// Creates the thing.
581     /// </para>
582     /// <para></para>
583     /// </summary>
584     /// <returns>
585     /// <para>The link</para>
586     /// <para></para>
587     /// </returns>
588     public static Link CreateThing() => Link.Create(Link.Itself, IsA, Thing);
589
590     /// <summary>
591     /// <para>
592     /// Creates the mapped thing using the specified mapping.
593     /// </para>
594     /// <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);
665             ↪ // Стоит переделать в "[x] is a member|instance|element of the class [y]"
666             IsNotA = Link.CreateMapped(Link.Itself, Link.Itself, Isa, NetMapping.IsNotA);
667             Link = Link.CreateMapped(Link.Itself, Isa, Link.Itself, NetMapping.Link);
668             Thing = Link.CreateMapped(Link.Itself, IsNotA, Link, NetMapping.Thing);

```

```

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");

```

```

747         Contains.SetName("contains");
748         ContainedBy.SetName("contained by");
749
750         One.SetName("one");
751         Zero.SetName("zero");
752
753         Character.SetName("character");
754         Sum.SetName("sum");
755         String.SetName("string");
756         Name.SetName("name");
757
758         Set.SetName("set");
759         Group.SetName("group");
760
761         ParsedFrom.SetName("parsed from");
762         ThatIs.SetName("that is");
763         ThatIsBefore.SetName("that is before");
764         ThatIsAfter.SetName("that is after");
765         ThatIsBetween.SetName("that is between");
766         ThatIsRepresentedBy.SetName("that is represented by");
767         ThatHas.SetName("that has");
768
769         Text.SetName("text");
770         Path.SetName("path");
771         Content.SetName("content");
772         Empty.SetName("empty");
773         EmptyContent.SetName("empty content");
774         Alphabet.SetName("alphabet");
775         Letter.SetName("letter");
776         Case.SetName("case");
777         Upper.SetName("upper");
778         Lower.SetName("lower");
779         Code.SetName("code");
780     }
781 }
782

```

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

```

1  using System;
2  using System.Collections.Generic;
3  using System.Globalization;
4  using Platform.Numbers;
5
6  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8  namespace Platform.Data.Triplets
9  {
10     /// <summary>
11     /// <para>
12     /// Represents the number helpers.
13     /// </para>
14     /// <para></para>
15     /// </summary>
16     public static class NumberHelpers
17     {
18         /// <summary>
19         /// <para>
20         /// Gets or sets the numbers to links value.
21         /// </para>
22         /// <para></para>
23         /// </summary>
24         public static Link[] NumbersToLinks { get; private set; }
25         /// <summary>
26         /// <para>
27         /// Gets or sets the links to numbers value.
28         /// </para>
29         /// <para></para>
30         /// </summary>
31         public static Dictionary<Link, long> LinksToNumbers { get; private set; }
32
33         /// <summary>
34         /// <para>
35         /// Initializes a new <see cref="NumberHelpers"/> instance.
36         /// </para>
37         /// <para></para>
38         /// </summary>
39         static NumberHelpers() => Create();
40
41         /// <summary>

```



```

42     /// <para>
43     /// Creates.
44     /// </para>
45     /// <para></para>
46     /// </summary>
47     private static void Create()
48     {
49         NumbersToLinks = new Link[64];
50         LinksToNumbers = new Dictionary<Link, long>();
51         NumbersToLinks[0] = Net.One;
52         LinksToNumbers[Net.One] = 1;
53     }
54
55     /// <summary>
56     /// <para>
57     /// Recreates.
58     /// </para>
59     /// <para></para>
60     /// </summary>
61     public static void Recreate() => Create();
62
63     /// <summary>
64     /// <para>
65     /// Creates the power of 2 using the specified power of 2.
66     /// </para>
67     /// <para></para>
68     /// </summary>
69     /// <param name="powerOf2">
70     /// <para>The power of.</para>
71     /// <para></para>
72     /// </param>
73     /// <returns>
74     /// <para>The result.</para>
75     /// <para></para>
76     /// </returns>
77     private static Link FromPowerOf2(long powerOf2)
78     {
79         var result = NumbersToLinks[powerOf2];
80         if (result == null)
81         {
82             var previousPowerOf2Link = NumbersToLinks[powerOf2 - 1];
83             if (previousPowerOf2Link == null)
84             {
85                 previousPowerOf2Link = NumbersToLinks[0];
86                 for (var i = 1; i < powerOf2; i++)
87                 {
88                     if (NumbersToLinks[i] == null)
89                     {
90                         var numberLink = Link.Create(Net.Sum, Net.Of, previousPowerOf2Link &
91                             ↪ previousPowerOf2Link);
92                         var num = (long)System.Math.Pow(2, i);
93                         NumbersToLinks[i] = numberLink;
94                         LinksToNumbers[numberLink] = num;
95                         numberLink.SetName(num.ToString(CultureInfo.InvariantCulture));
96                     }
97                     previousPowerOf2Link = NumbersToLinks[i];
98                 }
99                 result = Link.Create(Net.Sum, Net.Of, previousPowerOf2Link &
100                     ↪ previousPowerOf2Link);
101                 var number = (long)System.Math.Pow(2, powerOf2);
102                 NumbersToLinks[powerOf2] = result;
103                 LinksToNumbers[result] = number;
104                 result.SetName(number.ToString(CultureInfo.InvariantCulture));
105             }
106             return result;
107         }
108
109     /// <summary>
110     /// <para>
111     /// Creates the number using the specified number.
112     /// </para>
113     /// <para></para>
114     /// </summary>
115     /// <param name="number">
116     /// <para>The number.</para>
117     /// <para></para>
118     /// </param>

```

```

118 /// <exception cref="NotSupportedException">
119 /// <para>Negative numbers are not supported yet.</para>
120 /// <para></para>
121 /// </exception>
122 /// <returns>
123 /// <para>The sum.</para>
124 /// <para></para>
125 /// </returns>
126 public static Link FromNumber(long number)
127 {
128     if (number == 0)
129     {
130         return Net.Zero;
131     }
132     if (number == 1)
133     {
134         return Net.One;
135     }
136     var links = new Link[Bit.Count(number)];
137     if (number >= 0)
138     {
139         for (long key = 1, powerOf2 = 0, i = 0; key <= number; key *= 2, powerOf2++)
140         {
141             if ((number & key) == key)
142             {
143                 links[i] = FromPowerOf2(powerOf2);
144                 i++;
145             }
146         }
147     }
148     else
149     {
150         throw new NotSupportedException("Negative numbers are not supported yet.");
151     }
152     var sum = Link.Create(Net.Sum, Net.Of, LinkConverter.FromList(links));
153     return sum;
154 }
155
156 /// <summary>
157 /// <para>
158 /// Returns the number using the specified link.
159 /// </para>
160 /// <para></para>
161 /// </summary>
162 /// <param name="link">
163 /// <para>The link.</para>
164 /// <para></para>
165 /// </param>
166 /// <exception cref="ArgumentOutOfRangeException">
167 /// <para>Specified link is not a number.</para>
168 /// <para></para>
169 /// </exception>
170 /// <returns>
171 /// <para>The long</para>
172 /// <para></para>
173 /// </returns>
174 public static long ToNumber(Link link)
175 {
176     if (link == Net.Zero)
177     {
178         return 0;
179     }
180     if (link == Net.One)
181     {
182         return 1;
183     }
184     if (link.IsSum())
185     {
186         var numberParts = LinkConverter.ToList(link.Target);
187         long number = 0;
188         for (var i = 0; i < numberParts.Count; i++)
189         {
190             GoDownAndTakeIt(numberParts[i], out long numberPart);
191             number += numberPart;
192         }
193         return number;
194     }
195     throw new ArgumentOutOfRangeException(nameof(link), "Specified link is not a
↪ number.");

```

```

196     }
197
198     /// <summary>
199     /// <para>
200     /// Goes the down and take it using the specified link.
201     /// </para>
202     /// <para></para>
203     /// </summary>
204     /// <param name="link">
205     /// <para>The link.</para>
206     /// <para></para>
207     /// </param>
208     /// <param name="number">
209     /// <para>The number.</para>
210     /// <para></para>
211     /// </param>
212     private static void GoDownAndTakeIt(Link link, out long number)
213     {
214         if (!LinksToNumbers.TryGetValue(link, out number))
215         {
216             var previousNumberLink = link.Target.Source;
217             GoDownAndTakeIt(previousNumberLink, out number);
218             var previousNumberIndex = (int)System.Math.Log(number, 2);
219             var newNumberIndex = previousNumberIndex + 1;
220             var newNumberLink = Link.Create(Net.Sum, Net.Of, previousNumberLink &
221                 ↪ previousNumberLink);
222             number += number;
223             NumbersToLinks[newNumberIndex] = newNumberLink;
224             LinksToNumbers[newNumberLink] = number;
225         }
226     }
227 }

```

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

```

1  using System;
2  using System.Collections.Generic;
3
4  namespace Platform.Data.Triplets.Sequences
5  {
6      /// <summary>
7      /// <para>
8      /// Represents the compression experiments.
9      /// </para>
10     /// <para></para>
11     /// </summary>
12     internal static class CompressionExperiments
13     {
14         /// <summary>
15         /// <para>
16         /// Rights the join using the specified subject.
17         /// </para>
18         /// <para></para>
19         /// </summary>
20         /// <param name="subject">
21         /// <para>The subject.</para>
22         /// <para></para>
23         /// </param>
24         /// <param name="@object">
25         /// <para>The object.</para>
26         /// <para></para>
27         /// </param>
28         public static void RightJoin(ref Link subject, Link @object)
29         {
30             if (subject.Linker == Net.And && subject.ReferersBySourceCount == 0 &&
31                 ↪ subject.ReferersByTargetCount == 0)
32             {
33                 var subJoint = Link.Search(subject.Target, Net.And, @object);
34                 if (subJoint != null && subJoint != subject)
35                 {
36                     Link.Update(ref subject, subject.Source, Net.And, subJoint);
37                     return;
38                 }
39                 subject = Link.Create(subject, Net.And, @object);
40             }
41
42             //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 >
118 //         second.TotalReferers)
119 //         {
120 //             //if (first.Linker == Net.And)
121 //             //{
122 //                 // TODO: ...
123 //             }
124 //             if (second.TotalReferers > 0)
125 //             {
126 //                 // В данный момент это никак не влияет, из-за того что добавлено
127 //                 // условие по требованию
128 //                 // использования атомарного соединения если оно есть
129 //                 // В целом же приоритет между обходным соединением и атомарным
130 //                 // нужно определять по весу.
131 //                 // И если в сети обнаружено сразу два варианта прохода - простой и
132 //                 // обходной - нужно перебрасывать
133 //                 // пути с меньшим весом на использование путей с большим весом.
134 //                 // (Это и технически эффективнее и более оправдано
135 //                 // с точки зрения смысла).
136 //                 // Положительный эффект текущей реализации, что она быстро
137 //                 // "успокаивается" набирает критическую массу
138 //                 // и перестаёт вести себя не предсказуемо
139 //                 // Неприятность учёта веса в том, что нужно обрабатывать большое
140 //                 // количество комбинаций.
141 //                 // Но вероятно это оправдано.
142 //                 //var prev = Link.Search(first, Net.And, second);
143 //                 //if (prev != null && subJoint != prev) // && prev.TotalReferers <
144 //                 //subJoint.TotalReferers)
145 //                 //{
146 //                     // Link.Update(ref prev, subJoint, Net.And, second.Target);
147 //                     // if (second.TotalReferers == 0)
148 //                     // {
149 //                         // Link.Delete(ref second);
150 //                     // }
151 //                     // return prev;
152 //                 //}
153 //                 //return Link.Create(subJoint, Net.And, second.Target);
154 //             }
155 //             else
156 //             {
157 //                 Link.Update(ref second, subJoint, Net.And, second.Target);
158 //                 return second;
159 //             }
160 //         }
161 //         if (first.Linker == Net.And)
162 //         {
163 //             Link subJoint = Link.Search(first.Target, Net.And, second);
164 //             if (subJoint != null && subJoint != first)// && subJoint.TotalReferers >
165 //             first.TotalReferers)
166 //             {
167 //                 if (first.TotalReferers > 0)
168 //                 {
169 //                     //var prev = Link.Search(first, Net.And, second);
170 //                     //if (prev != null && subJoint != prev) // && prev.TotalReferers <
171 //                     //subJoint.TotalReferers)
172 //                     //{
173 //                         // Link.Update(ref prev, first.Source, Net.And, subJoint);
174 //                         // if (first.TotalReferers == 0)
175 //                         // {
176 //                             // Link.Delete(ref first);
177 //                         // }
178 //                         // return prev;
179 //                     //}
180 //                     //return Link.Create(first.Source, Net.And, subJoint);
181 //                 }
182 //                 else
183 //                 {
184 //                     Link.Update(ref first, first.Source, Net.And, subJoint);
185 //                     return first;
186 //                 }
187 //             }
188 //         }
189 //     }
190 // }

```

```

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
217     // ↳ аккумулятором одновременно
218     // Когда обе связи - пары - нужно использовать другой алгоритм, иначе сжатие будет
219     // ↳ отсутствовать.
220     //if ((first.Linker == Net.And && second.Linker != Net.And)
221     // || (second.Linker == Net.And && first.Linker != Net.And))
222     //{
223     //     Link connection = TryReconstructConnection(first, second);
224     //     if (connection != null)
225     //     {
226     //         CompressionsCount++;
227     //         return connection;
228     //     }
229     //}
230     //return first & second;
231     //long totalDoublets = Net.And.ReferersByLinkerCount;
232     if (first == null || second == null)
233     {
234     }
235     var directConnection = Link.Search(first, Net.And, second);
236     if (directConnection == null)
237     {
238         directConnection = TryReconstructConnection(first, second);
239     }
240     Link rightCrossConnection = null;
241     if (second.Linker == Net.And)
242     {
243         var assumedRightCrossConnection = Link.Search(first, Net.And, second.Source);
244         if (assumedRightCrossConnection != null && second != assumedRightCrossConnection)
245         {
246             rightCrossConnection = assumedRightCrossConnection;
247         }
248         else
249         {
250             rightCrossConnection = TryReconstructConnection(first, second.Source);
251         }
252     }
253     Link leftCrossConnection = null;
254     if (first.Linker == Net.And)
255     {
256         var assumedLeftCrossConnection = Link.Search(first.Target, Net.And, second);
257         if (assumedLeftCrossConnection != null && first != assumedLeftCrossConnection)
258         {
259             leftCrossConnection = assumedLeftCrossConnection;
260         }
261         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 rightMiddleCrossConnection = 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     }
329 }

```

```

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 >=
338         ↪ directConnection.TotalReferers)
339     && (leftCrossConnection == null || rightCrossConnection.TotalReferers >=
340         ↪ leftCrossConnection.TotalReferers))
341 {
342     if (directConnection != null)
343     {
344         var prev = Link.Search(first, Net.And, second);
345         if (prev != null && rightCrossConnection != prev)
346         {
347             Link.Update(ref prev, rightCrossConnection, Net.And, second.Target);
348         }
349     }
350     if (leftCrossConnection != null)
351     {
352         var prev = Link.Search(first.Source, Net.And, leftCrossConnection);
353         if (prev != null && rightCrossConnection != prev)
354         {
355             Link.Update(ref prev, rightCrossConnection, Net.And, second.Target);
356         }
357     }
358     //TryReconstructConnection(first, second.Source);
359     //TryReconstructConnection(rightCrossConnection, second.Target); // ухудшает
360     ↪ стабильность
361     var resultConnection = rightCrossConnection & second.Target;
362     //if (second.TotalReferers == 0)
363     //    Link.Delete(ref second);
364     return resultConnection;
365 }
366 else if (leftCrossConnection != null
367     && (directConnection == null || leftCrossConnection.TotalReferers >=
368         ↪ directConnection.TotalReferers)
369     && (rightCrossConnection == null || leftCrossConnection.TotalReferers >=
370         ↪ rightCrossConnection.TotalReferers))
371 {
372     if (directConnection != null)
373     {
374         var prev = Link.Search(first, Net.And, second);
375         if (prev != null && leftCrossConnection != prev)
376         {
377             Link.Update(ref prev, first.Source, Net.And, leftCrossConnection);
378         }
379     }
380     if (rightCrossConnection != null)
381     {
382         var prev = Link.Search(rightCrossConnection, Net.And, second.Target);
383         if (prev != null && rightCrossConnection != prev)
384         {
385             Link.Update(ref prev, first.Source, Net.And, leftCrossConnection);
386         }
387     }
388     //TryReconstructConnection(first.Target, second);
389     //TryReconstructConnection(first.Source, leftCrossConnection); // ухудшает
390     ↪ стабильность
391     var resultConnection = first.Source & leftCrossConnection;
392     //if (first.TotalReferers == 0)
393     //    Link.Delete(ref first);
394     return resultConnection;
395 }
396 else
397 {
398     if (directConnection != null)
399     {
400         return directConnection;
401     }
402     if (rightCrossConnection != null)
403     {
404         return rightCrossConnection & second.Target;
405     }
406 }

```



```

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

```

```

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 &&
485                 ↪ neighbour.ReferersBySourceCount == 0)
486             {
487                 if (directConnection == null)
488                 {
489                     directConnection = first & second;
490                 }
491                 Link.Update(ref couple, directConnection, Net.And,
492                     ↪ neighbour.Target);
493                 //Link.Delete(ref neighbour);
494             }
495         }
496     }
497 });
498 }
499 if (second.ReferersByTargetCount < first.ReferersByTargetCount)
500 {
501     // |_x      ... x_o
502     // |_o      |___|
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 &&
512                     ↪ neighbour.ReferersBySourceCount == 1)
513                 {
514                     if (directConnection == null)
515                     {
516                         directConnection = first & second;
517                     }
518                     Link.Update(ref couple, neighbour.Source, Net.And,
519                         ↪ directConnection);
520                     //Link.Delete(ref neighbour);
521                 }
522             }
523         }
524     });
525 }
526 else
527 {
528     // |_x      ... x_o
529     // |_o      |___|
530     // ->
531     first.WalkThroughReferersAsTarget((couple) =>
532     {
533         if (couple.Linker == Net.And && couple.ReferersByTargetCount == 0 &&
534             ↪ couple.ReferersBySourceCount == 1)
535         {
536             var neighbour = couple.FirstRefererBySource;
537             if (neighbour.Linker == Net.And && neighbour.Target == second)
538             {
539                 if (directConnection == null)
540                 {
541                     directConnection = first & second;
542                 }
543                 Link.Update(ref neighbour, couple.Source, Net.And, directConnection);
544                 Link.Delete(ref couple);
545             }
546         }
547     });
548 }
549 if (directConnection != null)
550 {

```

```

549         CompressionsCount++;
550     }
551     return directConnection;
552 }
553
554 ///public static Link CombinedJoin(Link left, Link right)
555 ///{
556     Link rightSubJoint = Link.Search(left, Net.And, right.Source);
557     if (rightSubJoint != null && rightSubJoint != right)
558     {
559         long rightSubJointReferers = rightSubJoint.TotalReferers;
560         Link leftSubJoint = Link.Search(left.Target, Net.And, right);
561         if (leftSubJoint != null && leftSubJoint != left)
562         {
563             long leftSubJointReferers = leftSubJoint.TotalReferers;
564             if (leftSubJointReferers > rightSubJointReferers)
565             {
566                 long leftReferers = left.TotalReferers;
567                 if (leftReferers > 0)
568                 {
569                     return Link.Create(left.Source, Net.And, leftSubJoint);
570                 }
571                 else
572                 {
573                     Link.Update(ref left, left.Source, Net.And, leftSubJoint);
574                     return left;
575                 }
576             }
577             long rightReferers = right.TotalReferers;
578             if (rightReferers > 0)
579             {
580                 return Link.Create(rightSubJoint, Net.And, right.Target);
581             }
582             else
583             {
584                 Link.Update(ref right, rightSubJoint, Net.And, right.Target);
585                 return right;
586             }
587         }
588     }
589     return Link.Create(left, Net.And, right);
590 }
591 //public static Link CombinedJoin(Link left, Link right)
592 //{
593     long leftReferers = left.TotalReferers;
594     Link leftSubJoint = Link.Search(left.Target, Net.And, right);
595     if (leftSubJoint != null && leftSubJoint != left)
596     {
597         long leftSubJointReferers = leftSubJoint.TotalReferers;
598     }
599     long rightReferers = left.TotalReferers;
600     Link rightSubJoint = Link.Search(left, Net.And, right.Source);
601     long rightSubJointReferers = rightSubJoint != null ? rightSubJoint.TotalReferers :
602     ↪ long.MinValue;
603 }
604 //public static Link LeftJoinUnsafe(Link subject, Link @object)
605 //{
606     if (subject.Linker == Net.And && subject.ReferersBySourceCount == 0 &&
607     ↪ subject.ReferersByTargetCount == 0)
608     {
609         Link subJoint = Link.Search(@object, Net.And, subject.Source);
610         if (subJoint != null && subJoint != subject)
611         {
612             Link.Update(ref subject, subJoint, Net.And, subject.Target);
613             return subject;
614         }
615     }
616     return Link.Create(@object, Net.And, subject);
617 }
618
619 /// <summary>
620 /// <para>
621 /// The chunk size.
622 /// </para>
623 /// <para></para>
624 /// </summary>
625 public static int ChunkSize = 2;

```

```

625 //public static Link FromList(List<Link> links)
626 //{
627 //    Link element = links[0];
628 //    for (int i = 1; i < links.Count; i += ChunkSize)
629 //    {
630 //        int j = (i + ChunkSize - 1);
631 //        j = j < links.Count ? j : (links.Count - 1);
632 //        Link subElement = links[j];
633 //        while (--j >= i) LeftJoin(ref subElement, links[j]);
634 //        RightJoin(ref element, subElement);
635 //    }
636 //    return element;
637 //}
638
639 //public static Link FromList(Link[] links)
640 //{
641 //    Link element = links[0];
642 //    for (int i = 1; i < links.Length; i += ChunkSize)
643 //    {
644 //        int j = (i + ChunkSize - 1);
645 //        j = j < links.Length ? j : (links.Length - 1);
646 //        Link subElement = links[j];
647 //        while (--j >= i) LeftJoin(ref subElement, links[j]);
648 //        RightJoin(ref element, subElement);
649 //    }
650 //    return element;
651 //}
652
653 //public static Link FromList(ICollection<Link> links)
654 //{
655 //    Link element = links[0];
656 //    for (int i = 1; i < links.Count; i += ChunkSize)
657 //    {
658 //        int j = (i + ChunkSize - 1);
659 //        j = j < links.Count ? j : (links.Count - 1);
660 //        Link subElement = links[j];
661 //        while (--j >= i)
662 //        {
663 //            Link x = links[j];
664 //            subElement = CombinedJoin(ref x, ref subElement);
665 //        }
666 //        element = CombinedJoin(ref element, ref subElement);
667 //    }
668 //    return element;
669 //}
670 //public static Link FromList(ICollection<Link> links)
671 //{
672 //    int i = 0;
673 //    Link element = links[i++];
674 //    if (links.Count % 2 == 0)
675 //    {
676 //        element = CombinedJoin(element, links[i++]);
677 //    }
678 //    for (; i < links.Count; i += 2)
679 //    {
680 //        Link doublet = CombinedJoin(links[i], links[i + 1]);
681 //        element = CombinedJoin(ref element, ref doublet);
682 //    }
683 //    return element;
684 //}
685
686 // Заглушка, возможно опасная
687 /// <summary>
688 /// <para>
689 /// Combineds the join using the specified element.
690 /// </para>
691 /// <para></para>
692 /// </summary>
693 /// <param name="element">
694 /// <para>The element.</para>
695 /// <para></para>
696 /// </param>
697 /// <param name="link">
698 /// <para>The link.</para>
699 /// <para></para>
700 /// </param>
701 /// <returns>
702 /// <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,
715     //         ↪ links, ref i);
716     //     return element;
717     ///}
718     ///public static Link FromList(Link[] links)
719     ///{
720     //     int i = links.Length - 1;
721     //     Link element = links[i];
722     //     while (--i >= 0) element = LinkConverterOld.ConnectLinks2(links[i], element,
723     //         ↪ links, ref i);
724     //     return element;
725     ///}
726     ///public static Link FromList(List<Link> links)
727     ///{
728     //     Link element = links[0];
729     //     for (int i = 1; i < links.Count; i += ChunkSize)
730     //     {
731     //         int j = (i + ChunkSize - 1);
732     //         j = j < links.Count ? j : (links.Count - 1);
733     //         Link subElement = links[j];
734     //         while (--j >= i) subElement = CombinedJoin(links[j], subElement);
735     //         element = CombinedJoin(element, subElement);
736     //     }
737     //     return element;
738     ///}
739     ///public static Link FromList(Link[] links)
740     ///{
741     //     Link element = links[0];
742     //     for (int i = 1; i < links.Length; i += ChunkSize)
743     //     {
744     //         int j = (i + ChunkSize - 1);
745     //         j = j < links.Length ? j : (links.Length - 1);
746     //         Link subElement = links[j];
747     //         while (--j >= i) subElement = CombinedJoin(links[j], subElement);
748     //         element = CombinedJoin(element, subElement);
749     //     }
750     //     return element;
751     ///}
752     ///public static Link FromList(ICollection<Link> links)
753     ///{
754     //     int leftBound = 0;
755     //     int rightBound = links.Count - 1;
756     //     if (leftBound == rightBound)
757     //     {
758     //         return links[0];
759     //     }
760     //     Link left = links[leftBound];
761     //     Link right = links[rightBound];
762     //     long leftReferers = left.ReferersBySourceCount + left.ReferersByTargetCount;
763     //     long rightReferers = right.ReferersBySourceCount + right.ReferersByTargetCount;
764     //     while (true)
765     //     {
766     //         //if (rightBound % 2 != leftBound % 2)
767     //         if (rightReferers >= leftReferers)
768     //         {
769     //             int nextRightBound = --rightBound;
770     //             if (nextRightBound == leftBound)
771     //             {
772     //                 var x = CombinedJoin(ref left, ref right);
773     //                 return x;
774     //             }
775     //             else
776     //             {
777     //                 Link nextRight = links[nextRightBound];
778     //                 right = CombinedJoin(ref nextRight, ref right);
779     //                 rightReferers = right.ReferersBySourceCount +
780     //                 ↪ 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(ICollection<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,
854             ↪ links[i]);

```

```

855         return element;
856     }
857 }
858 }

```

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

```

1  using System;
2  using System.Collections.Generic;
3  using System.Text;
4  using Platform.Data.Sequences;
5
6  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8  namespace Platform.Data.Triplets.Sequences
9  {
10     /// <remarks>
11     /// TODO: Check that CollectMatchingSequences algorithm is working, if not throw it away.
12     /// TODO: Think of the abstraction on Sequences that can be equally usefull for triple
13     /// ↪ links, doublet links and so on.
14     /// </remarks>
15     public static class SequenceHelpers
16     {
17         /// <summary>
18         /// <para>
19         /// The max sequence format size.
20         /// </para>
21         /// <para></para>
22         /// </summary>
23         public static readonly int MaxSequenceFormatSize = 20;
24
25         //public static void DeleteSequence(Link sequence)
26         //{
27         //}
28
29         /// <summary>
30         /// <para>
31         /// Formats the sequence using the specified sequence.
32         /// </para>
33         /// <para></para>
34         /// </summary>
35         /// <param name="sequence">
36         /// <para>The sequence.</para>
37         /// <para></para>
38         /// </param>
39         /// <returns>
40         /// <para>The string</para>
41         /// <para></para>
42         /// </returns>
43         public static string FormatSequence(Link sequence)
44         {
45             var visitedElements = 0;
46             var sb = new StringBuilder();
47             sb.Append('{');
48             StopableSequenceWalker.WalkRight(sequence, x => x.Source, x => x.Target, x =>
49                 ↪ x.Linker != Net.And, element =>
50                 {
51                     if (visitedElements > 0)
52                     {
53                         sb.Append(',');
54                     }
55                     sb.Append(element.ToString());
56                     visitedElements++;
57                     if (visitedElements < MaxSequenceFormatSize)
58                     {
59                         return true;
60                     }
61                     else
62                     {
63                         sb.Append(", ...");
64                         return false;
65                     }
66                 }
67             );
68             sb.Append('}');
69             return sb.ToString();
70         }
71
72         /// <summary>
73         /// <para>
74         /// Collects the matching sequences using the specified links.

```

```

72     /// </para>
73     /// <para></para>
74     /// </summary>
75     /// <param name="links">
76     /// <para>The links.</para>
77     /// <para></para>
78     /// </param>
79     /// <exception cref="InvalidOperationException">
80     /// <para>Подпоследовательности с одним элементом не поддерживаются.</para>
81     /// <para></para>
82     /// </exception>
83     /// <returns>
84     /// <para>The results.</para>
85     /// <para></para>
86     /// </returns>
87     public static List<Link> CollectMatchingSequences(Link[] links)
88     {
89         if (links.Length == 1)
90         {
91             throw new InvalidOperationException("Подпоследовательности с одним элементом не
92                 ↳ поддерживаются.");
93         }
94         var leftBound = 0;
95         var rightBound = links.Length - 1;
96         var left = links[leftBound++];
97         var right = links[rightBound--];
98         var results = new List<Link>();
99         CollectMatchingSequences(left, leftBound, links, right, rightBound, ref results);
100         return results;
101     }
102     /// <summary>
103     /// <para>
104     /// Collects the matching sequences using the specified left link.
105     /// </para>
106     /// <para></para>
107     /// </summary>
108     /// <param name="leftLink">
109     /// <para>The left link.</para>
110     /// <para></para>
111     /// </param>
112     /// <param name="leftBound">
113     /// <para>The left bound.</para>
114     /// <para></para>
115     /// </param>
116     /// <param name="middleLinks">
117     /// <para>The middle links.</para>
118     /// <para></para>
119     /// </param>
120     /// <param name="rightLink">
121     /// <para>The right link.</para>
122     /// <para></para>
123     /// </param>
124     /// <param name="rightBound">
125     /// <para>The right bound.</para>
126     /// <para></para>
127     /// </param>
128     /// <param name="results">
129     /// <para>The results.</para>
130     /// <para></para>
131     /// </param>
132     private static void CollectMatchingSequences(Link leftLink, int leftBound, Link[]
133         ↳ middleLinks, Link rightLink, int rightBound, ref List<Link> results)
134     {
135         var leftLinkTotalReferers = leftLink.ReferersBySourceCount +
136         ↳ leftLink.ReferersByTargetCount;
137         var rightLinkTotalReferers = rightLink.ReferersBySourceCount +
138         ↳ rightLink.ReferersByTargetCount;
139         if (leftLinkTotalReferers <= rightLinkTotalReferers)
140         {
141             var nextLeftLink = middleLinks[leftBound];
142             var elements = GetRightElements(leftLink, nextLeftLink);
143             if (leftBound <= rightBound)
144             {
145                 for (var i = elements.Length - 1; i >= 0; i--)
146                 {
147                     var element = elements[i];
148                     if (element != null)

```



```

146         {
147             CollectMatchingSequences(element, leftBound + 1, middleLinks,
148                                     ↪ rightLink, rightBound, ref results);
149         }
150     }
151     else
152     {
153         for (var i = elements.Length - 1; i >= 0; i--)
154         {
155             var element = elements[i];
156             if (element != null)
157             {
158                 results.Add(element);
159             }
160         }
161     }
162 }
163 else
164 {
165     var nextRightLink = middleLinks[rightBound];
166     var elements = GetLeftElements(rightLink, nextRightLink);
167     if (leftBound <= rightBound)
168     {
169         for (var i = elements.Length - 1; i >= 0; i--)
170         {
171             var element = elements[i];
172             if (element != null)
173             {
174                 CollectMatchingSequences(leftLink, leftBound, middleLinks,
175                                         ↪ elements[i], rightBound - 1, ref results);
176             }
177         }
178     }
179     else
180     {
181         for (var i = elements.Length - 1; i >= 0; i--)
182         {
183             var element = elements[i];
184             if (element != null)
185             {
186                 results.Add(element);
187             }
188         }
189     }
190 }
191
192 /// <summary>
193 /// <para>
194 /// Gets the right elements using the specified start link.
195 /// </para>
196 /// <para></para>
197 /// </summary>
198 /// <param name="startLink">
199 /// <para>The start link.</para>
200 /// <para></para>
201 /// </param>
202 /// <param name="rightLink">
203 /// <para>The right link.</para>
204 /// <para></para>
205 /// </param>
206 /// <returns>
207 /// <para>The result.</para>
208 /// <para></para>
209 /// </returns>
210 public static Link[] GetRightElements(Link startLink, Link rightLink)
211 {
212     var result = new Link[4];
213     TryStepRight(startLink, rightLink, result, 0);
214     startLink.WalkThroughReferersAsTarget(couple =>
215     {
216         if (couple.Linker == Net.And)
217         {
218             if (TryStepRight(couple, rightLink, result, 2))
219             {
220                 return Link.Stop;
221             }

```

```

222     }
223     return Link.Continue;
224 });
225 return result;
226 }
227
228 /// <summary>
229 /// <para>
230 /// Determines whether try step right.
231 /// </para>
232 /// <para></para>
233 /// </summary>
234 /// <param name="startLink">
235 /// <para>The start link.</para>
236 /// <para></para>
237 /// </param>
238 /// <param name="rightLink">
239 /// <para>The right link.</para>
240 /// <para></para>
241 /// </param>
242 /// <param name="result">
243 /// <para>The result.</para>
244 /// <para></para>
245 /// </param>
246 /// <param name="offset">
247 /// <para>The offset.</para>
248 /// <para></para>
249 /// </param>
250 /// <returns>
251 /// <para>The bool</para>
252 /// <para></para>
253 /// </returns>
254 public static bool TryStepRight(Link startLink, Link rightLink, Link[] result, int
    ↪ offset)
255 {
256     var added = 0;
257     startLink.WalkThroughReferersAsSource(couple =>
258     {
259         if (couple.Linker == Net.And)
260         {
261             var coupleTarget = couple.Target;
262             if (coupleTarget == rightLink)
263             {
264                 result[offset] = couple;
265                 if (++added == 2)
266                 {
267                     return Link.Stop;
268                 }
269             }
270             else if (coupleTarget.Linker == Net.And && coupleTarget.Source ==
    ↪ rightLink)
271             {
272                 result[offset + 1] = couple;
273                 if (++added == 2)
274                 {
275                     return Link.Stop;
276                 }
277             }
278         }
279         return Link.Continue;
280     });
281     return added > 0;
282 }
283
284 /// <summary>
285 /// <para>
286 /// Gets the left elements using the specified start link.
287 /// </para>
288 /// <para></para>
289 /// </summary>
290 /// <param name="startLink">
291 /// <para>The start link.</para>
292 /// <para></para>
293 /// </param>
294 /// <param name="leftLink">
295 /// <para>The left link.</para>
296 /// <para></para>
297 /// </param>

```

```

298 /// <returns>
299 /// <para>The result.</para>
300 /// <para></para>
301 /// </returns>
302 public static Link[] GetLeftElements(Link startLink, Link leftLink)
303 {
304     var result = new Link[4];
305     TryStepLeft(startLink, leftLink, result, 0);
306     startLink.WalkThroughReferersAsSource(couple =>
307     {
308         if (couple.Linker == Net.And)
309         {
310             if (TryStepLeft(couple, leftLink, result, 2))
311             {
312                 return Link.Stop;
313             }
314         }
315         return Link.Continue;
316     });
317     return result;
318 }
319
320 /// <summary>
321 /// <para>
322 /// Determines whether try step left.
323 /// </para>
324 /// <para></para>
325 /// </summary>
326 /// <param name="startLink">
327 /// <para>The start link.</para>
328 /// <para></para>
329 /// </param>
330 /// <param name="leftLink">
331 /// <para>The left link.</para>
332 /// <para></para>
333 /// </param>
334 /// <param name="result">
335 /// <para>The result.</para>
336 /// <para></para>
337 /// </param>
338 /// <param name="offset">
339 /// <para>The offset.</para>
340 /// <para></para>
341 /// </param>
342 /// <returns>
343 /// <para>The bool</para>
344 /// <para></para>
345 /// </returns>
346 public static bool TryStepLeft(Link startLink, Link leftLink, Link[] result, int offset)
347 {
348     var added = 0;
349     startLink.WalkThroughReferersAsTarget(couple =>
350     {
351         if (couple.Linker == Net.And)
352         {
353             var coupleSource = couple.Source;
354             if (coupleSource == leftLink)
355             {
356                 result[offset] = couple;
357                 if (++added == 2)
358                 {
359                     return Link.Stop;
360                 }
361             }
362             else if (coupleSource.Linker == Net.And && coupleSource.Target ==
363                 ↪ leftLink)
364             {
365                 result[offset + 1] = couple;
366                 if (++added == 2)
367                 {
368                     return Link.Stop;
369                 }
370             }
371             return Link.Continue;
372         });
373     return added > 0;
374 }

```

```
375 }
376 }
```

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

```
1 using System.IO;
2 using Xunit;
3 using Platform.Random;
4 using Platform.Ranges;
5
6 namespace Platform.Data.Triplets.Tests
7 {
8     /// <summary>
9     /// <para>
10    /// Represents the link tests.
11    /// </para>
12    /// <para></para>
13    /// </summary>
14    public static class LinkTests
15    {
16        /// <summary>
17        /// <para>
18        /// The lock.
19        /// </para>
20        /// <para></para>
21        /// </summary>
22        public static object Lock = new object(); //-V3090
23
24        /// <summary>
25        /// <para>
26        /// The thing visitor counter.
27        /// </para>
28        /// <para></para>
29        /// </summary>
30        private static ulong _thingVisitorCounter;
31        /// <summary>
32        /// <para>
33        /// The is visitor counter.
34        /// </para>
35        /// <para></para>
36        /// </summary>
37        private static ulong _isAVisitorCounter;
38        /// <summary>
39        /// <para>
40        /// The link visitor counter.
41        /// </para>
42        /// <para></para>
43        /// </summary>
44        private static ulong _linkVisitorCounter;
45
46        /// <summary>
47        /// <para>
48        /// Things the visitor using the specified link index.
49        /// </para>
50        /// <para></para>
51        /// </summary>
52        /// <param name="linkIndex">
53        /// <para>The link index.</para>
54        /// <para></para>
55        /// </param>
56        static void ThingVisitor(Link linkIndex)
57        {
58            _thingVisitorCounter += linkIndex;
59        }
60
61        /// <summary>
62        /// <para>
63        /// Ises the a visitor using the specified link index.
64        /// </para>
65        /// <para></para>
66        /// </summary>
67        /// <param name="linkIndex">
68        /// <para>The link index.</para>
69        /// <para></para>
70        /// </param>
71        static void IsAVisitor(Link linkIndex)
72        {
73            _isAVisitorCounter += linkIndex;
74        }
75    }
```

```

76     /// <summary>
77     /// <para>
78     /// Links the visitor using the specified link index.
79     /// </para>
80     /// <para></para>
81     /// </summary>
82     /// <param name="linkIndex">
83     /// <para>The link index.</para>
84     /// <para></para>
85     /// </param>
86     static void LinkVisitor(Link linkIndex)
87     {
88         _linkVisitorCounter += linkIndex;
89     }
90
91     /// <summary>
92     /// <para>
93     /// Tests that create delete link test.
94     /// </para>
95     /// <para></para>
96     /// </summary>
97     [Fact]
98     public static void CreateDeleteLinkTest()
99     {
100         lock (Lock)
101         {
102             string filename = "db.links";
103
104             File.Delete(filename);
105
106             Link.StartMemoryManager(filename);
107
108             Link link1 = Link.Create(Link.Itself, Link.Itself, Link.Itself);
109
110             Link.Delete(link1);
111
112             Link.StopMemoryManager();
113
114             File.Delete(filename);
115         }
116     }
117
118     /// <summary>
119     /// <para>
120     /// Tests that deep create update delete link test.
121     /// </para>
122     /// <para></para>
123     /// </summary>
124     [Fact]
125     public static void DeepCreateUpdateDeleteLinkTest()
126     {
127         lock (Lock)
128         {
129             string filename = "db.links";
130
131             File.Delete(filename);
132
133             Link.StartMemoryManager(filename);
134
135             Link isA = Link.Create(Link.Itself, Link.Itself, Link.Itself);
136             Link isNotA = Link.Create(Link.Itself, Link.Itself, isA);
137             Link link = Link.Create(Link.Itself, isA, Link.Itself);
138             Link thing = Link.Create(Link.Itself, isNotA, link);
139
140             //Assert::IsTrue(GetLinksCount() == 4);
141
142             Assert.Equal(isA, isA.Target);
143
144             isA = Link.Update(isA, isA, isA, link); // Произведено замыкание
145
146             Assert.Equal(link, isA.Target);
147
148             Link.Delete(isA); // Одна эта операция удалит все 4 связи
149
150             //Assert::IsTrue(GetLinksCount() == 0);
151
152             Link.StopMemoryManager();
153
154             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
231     ↪ operations per cycle.
232 /// </para>
233 /// <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

```

```

31         Link.StopMemoryManager();
32
33         File.Delete(filename);
34     }
35 }
36
37 /// <summary>
38 /// <para>
39 /// Tests that allocate and free link test.
40 /// </para>
41 /// <para></para>
42 /// </summary>
43 [Fact]
44 public static void AllocateAndFreeLinkTest()
45 {
46     lock (LinkTests.Lock)
47     {
48         string filename = "db.links";
49
50         File.Delete(filename);
51
52         Link.StartMemoryManager(filename);
53
54         Link link = Link.Create(Link.Itself, Link.Itself, Link.Itself);
55
56         Link.Delete(link);
57
58         Link.StopMemoryManager();
59
60         File.Delete(filename);
61     }
62 }
63
64 /// <summary>
65 /// <para>
66 /// Tests that attach to unused link test.
67 /// </para>
68 /// <para></para>
69 /// </summary>
70 [Fact]
71 public static void AttachToUnusedLinkTest()
72 {
73     lock (LinkTests.Lock)
74     {
75         string filename = "db.links";
76
77         File.Delete(filename);
78
79         Link.StartMemoryManager(filename);
80
81         Link link1 = Link.Create(Link.Itself, Link.Itself, Link.Itself);
82         Link link2 = Link.Create(Link.Itself, Link.Itself, Link.Itself);
83
84         Link.Delete(link1); // Creates "hole" and forces "Attach" to be executed
85
86         Link.StopMemoryManager();
87
88         File.Delete(filename);
89     }
90 }
91
92 /// <summary>
93 /// <para>
94 /// Tests that detach to unused link test.
95 /// </para>
96 /// <para></para>
97 /// </summary>
98 [Fact]
99 public static void DetachToUnusedLinkTest()
100 {
101     lock (LinkTests.Lock)
102     {
103         string filename = "db.links";
104
105         File.Delete(filename);
106
107         Link.StartMemoryManager(filename);
108
109         Link link1 = Link.Create(Link.Itself, Link.Itself, Link.Itself);

```



```

110         Link link2 = Link.Create(Link.Itself, Link.Itself, Link.Itself);
111
112         Link.Delete(link1); // Creates "hole" and forces "Attach" to be executed
113         Link.Delete(link2); // Removes both links, all "Attached" links forced to be
            ↳ "Detached" here
114
115         Link.StopMemoryManager();
116
117         File.Delete(filename);
118     }
119 }
120
121 /// <summary>
122 /// <para>
123 /// Tests that get set mapped link test.
124 /// </para>
125 /// <para></para>
126 /// </summary>
127 [Fact]
128 public static void GetSetMappedLinkTest()
129 {
130     lock (LinkTests.Lock)
131     {
132         string filename = "db.links";
133
134         File.Delete(filename);
135
136         Link.StartMemoryManager(filename);
137
138         var mapped = Link.GetMappedOrDefault(0);
139
140         var mappingSet = Link.TrySetMapped(mapped, 0);
141
142         Assert.True(mappingSet);
143
144         Link.StopMemoryManager();
145
146         File.Delete(filename);
147     }
148 }
149 }
150 }

```

## Index

- ./csharp/Platform.Data.Triplets.Tests/LinkTests.cs, 84
- ./csharp/Platform.Data.Triplets.Tests/PersistentMemoryManagerTests.cs, 87
- ./csharp/Platform.Data.Triplets/CharacterHelpers.cs, 1
- ./csharp/Platform.Data.Triplets/GexfExporter.cs, 8
- ./csharp/Platform.Data.Triplets/ILink.cs, 10
- ./csharp/Platform.Data.Triplets/Link.Debug.cs, 13
- ./csharp/Platform.Data.Triplets/Link.cs, 15
- ./csharp/Platform.Data.Triplets/LinkConverter.cs, 41
- ./csharp/Platform.Data.Triplets/LinkExtensions.cs, 49
- ./csharp/Platform.Data.Triplets/Net.cs, 54
- ./csharp/Platform.Data.Triplets/NumberHelpers.cs, 64
- ./csharp/Platform.Data.Triplets/Sequences/CompressionExperiments.cs, 67
- ./csharp/Platform.Data.Triplets/Sequences/SequenceHelpers.cs, 79