

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        private const char FirstLowerCaseLatinLetter = 'a';
41        private const char LastLowerCaseLatinLetter = 'z';
42        private const char FirstUpperCaseLatinLetter = 'A';
43        private const char LastUpperCaseLatinLetter = 'Z';
44        private const char FirstLowerCaseCyrillicLetter = 'а';
45        private const char LastLowerCaseCyrillicLetter = 'я';
46        private const char FirstUpperCaseCyrillicLetter = 'А';
47        private const char LastUpperCaseCyrillicLetter = 'Я';
48        private const char YoLowerCaseCyrillicLetter = 'ё';
49        private const char YoUpperCaseCyrillicLetter = 'Ё';
50        private static Link[] _charactersToLinks;
51        private static Dictionary<Link, char> _linksToCharacters;
52
53        /// <summary>
54        /// <para>
55        /// Initializes a new <see cref="CharacterHelpers"/> instance.
56        /// </para>
57        /// <para></para>
58        /// </summary>
59        static CharacterHelpers() => Create();
60        private static void Create()
61        {
62            _charactersToLinks = new Link[char.MaxValue];
63            _linksToCharacters = new Dictionary<Link, char>();
64            // Create or restore characters
65            CreateLatinAlphabet();
66            CreateCyrillicAlphabet();
67            RegisterExistingCharacters();
68        }
69        private static void RegisterExistingCharacters() =>
70        ↪ Net.Character.WalkThroughReferersAsSource(referer =>
71        ↪ RegisterExistingCharacter(referer));
72        private static void RegisterExistingCharacter(Link character)
73        {
74            if (character.Source == Net.Character && character.Linker == Net.ThatHas)
75            {
76                var code = character.Target;
77                if (code.Source == Net.Code && code.Linker == Net.ThatIsRepresentedBy)
```

```

76         {
77             var charCode = (char)LinkConverter.ToNumber(code.Target);
78             _charactersToLinks[charCode] = character;
79             _linksToCharacters[character] = charCode;
80         }
81     }
82 }
83
84 /// <summary>
85 /// <para>
86 /// Recreates.
87 /// </para>
88 /// <para></para>
89 /// </summary>
90 public static void Recreate() => Create();
91 private static void CreateLatinAlphabet()
92 {
93     var lettersCharacters = new[]
94     {
95         'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j',
96         'k', 'l', 'm', 'n', 'o', 'p', 'q', 'r', 's', 't',
97         'u', 'v', 'w', 'x', 'y', 'z',
98     };
99     CreateAlphabet(lettersCharacters, "latin alphabet", CharacterMapping.LatinAlphabet);
100 }
101 private static void CreateCyrillicAlphabet()
102 {
103     var lettersCharacters = new[]
104     {
105         'a', 'б', 'в', 'г', 'д', 'e', 'ё', 'ж', 'з', 'и',
106         'й', 'к', 'л', 'м', 'н', 'o', 'п', 'р', 'с', 'т',
107         'y', 'ф', 'х', 'ц', 'ч', 'ш', 'щ', 'ъ', 'ы', 'ь',
108         'э', 'ю', 'я',
109     };
110     CreateAlphabet(lettersCharacters, "cyrillic alphabet",
111         ↪ CharacterMapping.CyrillicAlphabet);
112 }
113 private static void CreateAlphabet(char[] lettersCharacters, string alphabetName,
114     ↪ CharacterMapping mapping)
115 {
116     if (Link.TryGetMapped(mapping, out Link alphabet))
117     {
118         var letters = alphabet.Target;
119         letters.WalkThroughSequence(letter =>
120         {
121             var lowerCaseLetter = Link.Search(Net.LowerCase, Net.Of, letter);
122             var upperCaseLetter = Link.Search(Net.UpperCase, Net.Of, letter);
123             if (lowerCaseLetter != null && upperCaseLetter != null)
124             {
125                 RegisterExistingLetter(lowerCaseLetter);
126                 RegisterExistingLetter(upperCaseLetter);
127             }
128             else
129             {
130                 RegisterExistingLetter(letter);
131             }
132         });
133     }
134     else
135     {
136         alphabet = Net.CreateMappedThing(mapping);
137         var letterOfAlphabet = Link.Create(Net.Letter, Net.Of, alphabet);
138         var lettersLinks = new Link[lettersCharacters.Length];
139         GenerateAlphabetBasis(ref alphabet, ref letterOfAlphabet, lettersLinks);
140         for (var i = 0; i < lettersCharacters.Length; i++)
141         {
142             var lowerCaseCharacter = lettersCharacters[i];
143             SetLetterCodes(lettersLinks[i], lowerCaseCharacter, out Link lowerCaseLink,
144                 ↪ out Link upperCaseLink);
145             _charactersToLinks[lowerCaseCharacter] = lowerCaseLink;
146             _linksToCharacters[lowerCaseLink] = lowerCaseCharacter;
147             if (upperCaseLink != null)
148             {
149                 var upperCaseCharacter = char.ToUpper(lowerCaseCharacter);
150                 _charactersToLinks[upperCaseCharacter] = upperCaseLink;
151                 _linksToCharacters[upperCaseLink] = upperCaseCharacter;
152             }
153         }
154     }
155 }

```

```

151     alphabet.SetName(alphabetName);
152     for (var i = 0; i < lettersCharacters.Length; i++)
153     {
154         var lowerCaseCharacter = lettersCharacters[i];
155         var upperCaseCharacter = char.ToUpper(lowerCaseCharacter);
156         if (lowerCaseCharacter != upperCaseCharacter)
157         {
158             lettersLinks[i].SetName("{ " + upperCaseCharacter + " " +
159                 ↪ lowerCaseCharacter + "}");
160         }
161         else
162         {
163             lettersLinks[i].SetName("{ " + lowerCaseCharacter + "}");
164         }
165     }
166 }
167 private static void RegisterExistingLetter(Link letter)
168 {
169     letter.WalkThroughReferersAsSource(referer =>
170     {
171         if (referer.Linker == Net.Has)
172         {
173             var target = referer.Target;
174             if (target.Source == Net.Code && target.Linker ==
175                 ↪ Net.ThatIsRepresentedBy)
176             {
177                 var charCode = (char)LinkConverter.ToNumber(target.Target);
178                 _charactersToLinks[charCode] = letter;
179                 _linksToCharacters[letter] = charCode;
180             }
181         }
182     });
183 private static void GenerateAlphabetBasis(ref Link alphabet, ref Link letterOfAlphabet,
184     ↪ Link[] letters)
185 {
186     // Принцип, на примере латинского алфавита.
187     // latin alphabet: alphabet that consists of a and b and c and ... and z.
188     // a: letter of latin alphabet that is before b.
189     // b: letter of latin alphabet that is between (a and c).
190     // c: letter of latin alphabet that is between (b and e).
191     // ...
192     // y: letter of latin alphabet that is between (x and z).
193     // z: letter of latin alphabet that is after y.
194     const int firstLetterIndex = 0;
195     for (var i = firstLetterIndex; i < letters.Length; i++)
196     {
197         letters[i] = Net.CreateThing();
198     }
199     var lastLetterIndex = letters.Length - 1;
200     Link.Update(ref letters[firstLetterIndex], letterOfAlphabet, Net.ThatIsBefore,
201         ↪ letters[firstLetterIndex + 1]);
202     Link.Update(ref letters[lastLetterIndex], letterOfAlphabet, Net.ThatIsAfter,
203         ↪ letters[lastLetterIndex - 1]);
204     const int secondLetterIndex = firstLetterIndex + 1;
205     for (var i = secondLetterIndex; i < lastLetterIndex; i++)
206     {
207         Link.Update(ref letters[i], letterOfAlphabet, Net.ThatIsBetween, letters[i - 1]
208             ↪ & letters[i + 1]);
209     }
210     Link.Update(ref alphabet, Net.Alphabet, Net.ThatConsistsOf,
211         ↪ LinkConverter.FromList(letters));
212 }
213 private static void SetLetterCodes(Link letter, char lowerCaseCharacter, out Link
214     ↪ lowerCase, out Link upperCase)
215 {
216     var upperCaseCharacter = char.ToUpper(lowerCaseCharacter);
217     if (upperCaseCharacter != lowerCaseCharacter)
218     {
219         lowerCase = Link.Create(Net.LowerCase, Net.Of, letter);
220         var lowerCaseCharacterCode = Link.Create(Net.Code, Net.ThatIsRepresentedBy,
221             ↪ LinkConverter.FromNumber(lowerCaseCharacter));
222         Link.Create(lowerCase, Net.Has, lowerCaseCharacterCode);
223         upperCase = Link.Create(Net.UpperCase, Net.Of, letter);
224         var upperCaseCharacterCode = Link.Create(Net.Code, Net.ThatIsRepresentedBy,
225             ↪ LinkConverter.FromNumber(upperCaseCharacter));

```

```

218         Link.Create(upperCase, Net.Has, upperCaseCharacterCode);
219     }
220     else
221     {
222         lowerCase = letter;
223         upperCase = null;
224         Link.Create(letter, Net.Has, Link.Create(Net.Code, Net.ThatIsRepresentedBy,
225             ↪ LinkConverter.FromNumber(lowerCaseCharacter)));
226     }
227 private static Link CreateSimpleCharacterLink(char character) =>
228     ↪ Link.Create(Net.Character, Net.ThatHas, Link.Create(Net.Code,
229     ↪ Net.ThatIsRepresentedBy, LinkConverter.FromNumber(character)));
230 private static bool IsLetterOfLatinAlphabet(char character)
231     => (character >= FirstLowerCaseLatinLetter && character <= LastLowerCaseLatinLetter)
232     || (character >= FirstUpperCaseLatinLetter && character <= LastUpperCaseLatinLetter);
233 private static bool IsLetterOfCyrillicAlphabet(char character)
234     => (character >= FirstLowerCaseCyrillicLetter && character <=
235     ↪ LastLowerCaseCyrillicLetter)
236     || (character >= FirstUpperCaseCyrillicLetter && character <=
237     ↪ LastUpperCaseCyrillicLetter)
238     || character == YoLowerCaseCyrillicLetter || character == YoUpperCaseCyrillicLetter;
239
240 /// <summary>
241 /// <para>
242 /// Creates the char using the specified character.
243 /// </para>
244 /// <para></para>
245 /// </summary>
246 /// <param name="character">
247 /// <para>The character.</para>
248 /// <para></para>
249 /// </param>
250 /// <returns>
251 /// <para>The link</para>
252 /// <para></para>
253 /// </returns>
254 public static Link FromChar(char character)
255 {
256     if (_charactersToLinks[character] == null)
257     {
258         if (IsLetterOfLatinAlphabet(character))
259         {
260             CreateLatinAlphabet();
261             return _charactersToLinks[character];
262         }
263         else if (IsLetterOfCyrillicAlphabet(character))
264         {
265             CreateCyrillicAlphabet();
266             return _charactersToLinks[character];
267         }
268         else
269         {
270             var simpleCharacter = CreateSimpleCharacterLink(character);
271             _charactersToLinks[character] = simpleCharacter;
272             _linksToCharacters[simpleCharacter] = character;
273             return simpleCharacter;
274         }
275     }
276     else
277     {
278         return _charactersToLinks[character];
279     }
280 }
281
282 /// <summary>
283 /// <para>
284 /// Returns the char using the specified link.
285 /// </para>
286 /// <para></para>
287 /// </summary>
288 /// <param name="link">
289 /// <para>The link.</para>
290 /// <para></para>
291 /// </param>
292 /// <exception cref="ArgumentOutOfRangeException">
293 /// <para>Указанная связь не является символом.</para>
294 /// <para></para>

```

```

291     /// </exception>
292     /// <returns>
293     /// <para>The char.</para>
294     /// <para></para>
295     /// </returns>
296     public static char ToChar(Link link)
297     {
298         if (!_linksToCharacters.TryGetValue(link, out char @char))
299         {
300             throw new ArgumentOutOfRangeException(nameof(link), "Указанная связь не
301                 ↳ является символом.");
302         }
303         return @char;
304     }
305     /// <summary>
306     /// <para>
307     /// Determines whether is char.
308     /// </para>
309     /// <para></para>
310     /// </summary>
311     /// <param name="link">
312     /// <para>The link.</para>
313     /// <para></para>
314     /// </param>
315     /// <returns>
316     /// <para>The bool</para>
317     /// <para></para>
318     /// </returns>
319     public static bool IsChar(Link link) => link != null &&
320         ↳ _linksToCharacters.ContainsKey(link);
321 }

```

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         private const string SourceLabel = "Source";
23         private const string LinkerLabel = "Linker";
24         private const string TargetLabel = "Target";
25
26         /// <summary>
27         /// <para>
28         /// Returns the xml.
29         /// </para>
30         /// <para></para>
31         /// </summary>
32         /// <returns>
33         /// <para>The string</para>
34         /// <para></para>
35         /// </returns>
36         public static string ToXml()
37         {
38             var sb = new StringBuilder();
39             using (var writer = XmlWriter.Create(sb))
40             {
41                 WriteXml(writer, CollectLinks());
42             }
43             return sb.ToString();
44         }
45     }
46 }

```

```

45
46 /// <summary>
47 /// <para>
48 /// Returns the file using the specified path.
49 /// </para>
50 /// <para></para>
51 /// </summary>
52 /// <param name="path">
53 /// <para>The path.</para>
54 /// <para></para>
55 /// </param>
56 public static void ToFile(string path)
57 {
58     using (var file = File.OpenWrite(path))
59     using (var writer = XmlWriter.Create(file))
60     {
61         WriteXml(writer, CollectLinks());
62     }
63 }
64
65 /// <summary>
66 /// <para>
67 /// Returns the file using the specified path.
68 /// </para>
69 /// <para></para>
70 /// </summary>
71 /// <param name="path">
72 /// <para>The path.</para>
73 /// <para></para>
74 /// </param>
75 /// <param name="filter">
76 /// <para>The filter.</para>
77 /// <para></para>
78 /// </param>
79 public static void ToFile(string path, Func<Link, bool> filter)
80 {
81     using (var file = File.OpenWrite(path))
82     using (var writer = XmlWriter.Create(file))
83     {
84         WriteXml(writer, CollectLinks(filter));
85     }
86 }
87 private static HashSet<Link> CollectLinks(Func<Link, bool> linkMatch)
88 {
89     var matchingLinks = new HashSet<Link>();
90     Link.WalkThroughAllLinks(link =>
91     {
92         if (linkMatch(link))
93         {
94             matchingLinks.Add(link);
95         }
96     });
97     return matchingLinks;
98 }
99 private static HashSet<Link> CollectLinks()
100 {
101     var matchingLinks = new HashSet<Link>();
102     Link.WalkThroughAllLinks(matchingLinks.AddAndReturnVoid);
103     return matchingLinks;
104 }
105 private static void WriteXml(XmlWriter writer, HashSet<Link> matchingLinks)
106 {
107     var edgesCounter = 0;
108     Gexf.WriteXml(writer,
109     () => // nodes
110     {
111         foreach (var matchingLink in matchingLinks)
112         {
113             GexfNode.WriteXml(writer, matchingLink.ToInt(), matchingLink.ToString());
114         }
115     },
116     () => // edges
117     {
118         foreach (var matchingLink in matchingLinks)
119         {
120             if (matchingLinks.Contains(matchingLink.Source))
121             {

```

```

122         Edge.WriteXml(writer, edgesCounter++, matchingLink.ToInt(),
123             ↳ matchingLink.Source.ToInt(), SourceLabel);
124     }
125     if (matchingLinks.Contains(matchingLink.Linker))
126     {
127         Edge.WriteXml(writer, edgesCounter++, matchingLink.ToInt(),
128             ↳ matchingLink.Linker.ToInt(), LinkerLabel);
129     }
130     if (matchingLinks.Contains(matchingLink.Target))
131     {
132         Edge.WriteXml(writer, edgesCounter++, matchingLink.ToInt(),
133             ↳ matchingLink.Target.ToInt(), TargetLabel);
134     }
135 }
136 }

```

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<TLinkAddress>
14     where TLinkAddress : ILink<TLinkAddress>
15     {
16         /// <summary>
17         /// <para>
18         /// Gets the source value.
19         /// </para>
20         /// <para></para>
21         /// </summary>
22         TLinkAddress Source { get; }
23         /// <summary>
24         /// <para>
25         /// Gets the linker value.
26         /// </para>
27         /// <para></para>
28         /// </summary>
29         TLinkAddress Linker { get; }
30         /// <summary>
31         /// <para>
32         /// Gets the target value.
33         /// </para>
34         /// <para></para>
35         /// </summary>
36         TLinkAddress Target { get; }
37     }
38
39     /// <summary>
40     /// <para>
41     /// Defines the link.
42     /// </para>
43     /// <para></para>
44     /// </summary>
45     internal partial interface ILink<TLinkAddress>
46     where TLinkAddress : ILink<TLinkAddress>
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<TLinkAddress, 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<TLinkAddress, 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<TLinkAddress, 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<TLinkAddress, bool> walker);
104 }
105
106 /// <summary>
107 /// <para>
108 /// Defines the link.
109 /// </para>
110 /// <para></para>
111 /// </summary>
112 internal partial interface ILink<TLinkAddress>
113     where TLinkAddress : ILink<TLinkAddress>
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<TLinkAddress> 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<TLinkAddress> 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<TLinkAddress> 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<TLinkAddress> 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
170         ↪ newTarget);
171         public static void Delete(ref ILink link);
172         public static ILink Search(ILink source, ILink linker, ILink target);
173     }
174 }
175 */
176 /*
177 Набор функций, который необходим для работы с сущностью Link:
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 связки между "начальной" и "конечной"
181   ↪ сущностями Link. (Получить адрес из поля Linker)
182 3. Получить адрес "конечной" сущности Link. (Получить адрес из поля Target)
183 4. Пройтись по всем сущностями Link, которые ссылаются на сущность Link с указанным адресом, и у
184   ↪ которых поле Source равно этому адресу.
185 5. Пройтись по всем сущностями Link, которые ссылаются на сущность Link с указанным адресом, и у
186   ↪ которых поле Linker равно этому адресу.
187 6. Пройтись по всем сущностями Link, которые ссылаются на сущность Link с указанным адресом, и у
188   ↪ которых поле Target равно этому адресу.
189 7. Создать сущность Link со значением (смыслом), которым являются адреса на другие 3 сущности
190   ↪ Link (где первая является "начальной", вторая является "связкой", а третья является
191   ↪ "конечной").
192 8. Обновление сущности Link с указанным адресом новым значением (смыслом), которым являются
193   ↪ адреса на другие 3 сущности Link (где первая является "начальной", вторая является
194   ↪ "связкой", а третья является "конечной").
195 9. Удаление сущности Link с указанным адресом.
196 10. Поиск сущности Link со значением (смыслом), которым являются адреса на другие 3 сущности
197   ↪ Link (где первая является "начальной", вторая является "связкой", а третья является
198   ↪ "конечной").
199 */

```

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
116    #endregion
117
118    /// <summary>
119    /// <para>
120    /// Returns the string.
121    /// </para>
122    /// <para></para>
123    /// </summary>
124    /// <returns>
125    /// <para>The string</para>
126    /// <para></para>
127    /// </returns>
128    public override string ToString()
129    {
130        const string nullString = "null";
131        if (this == null)
132        {
133            return nullString;
134        }
135        else
136        {
137            if (this.TryGetName(out string name))
138            {
139                return name;
140            }
141            else
142            {
143                return ((long)_link).ToString();
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     private const string DllName = "Platform_Data_Triplets_Kernel";
106
107     // TODO: Заменить на очередь событий, по примеру Node.js (+сделать выключаемым)
108     /// <summary>
109     /// <para>
110     /// The created delegate.
111     /// </para>
112     /// <para></para>
113     /// </summary>
114     public delegate void CreatedDelegate(LinkDefinition createdLink);
115     public static event CreatedDelegate CreatedEvent = createdLink => { };
116
117     /// <summary>
118     /// <para>
119     /// The updated delegate.
120     /// </para>
121     /// <para></para>
122     /// </summary>
123     public delegate void UpdatedDelegate(LinkDefinition linkBeforeUpdate, LinkDefinition
124     ↪ linkAfterUpdate);
125     public static event UpdatedDelegate UpdatedEvent = (linkBeforeUpdate, linkAfterUpdate)
126     ↪ => { };
127
128     /// <summary>
129     /// <para>
130     /// The deleted delegate.
131     /// </para>
132     /// <para></para>
133     /// </summary>
134     public delegate void DeletedDelegate(LinkDefinition deletedLink);
135     public static event DeletedDelegate DeletedEvent = deletedLink => { };
136
137     #region Low Level
138
139     #region Basic Operations
140
141     /// <summary>
142     /// <para>
143     /// Gets the source index using the specified link.
144     /// </para>
145     /// <para></para>
146     /// </summary>
147     /// <param name="link">
148     /// <para>The link.</para>
149     /// <para></para>
150     /// </param>
151     /// <returns>
152     /// <para>The link index</para>
153     /// <para></para>
154     /// </returns>
155     [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
156     private static extern LinkIndex GetSourceIndex(LinkIndex link);
157
158     /// <summary>
159     /// <para>
160     /// Gets the linker index using the specified link.
161     /// </para>
162     /// <para></para>
163     /// </summary>
164     /// <param name="link">
165     /// <para>The link.</para>
166     /// <para></para>
167     /// </param>
168     /// <returns>
169     /// <para>The link index</para>
170     /// <para></para>
171     /// </returns>
172     [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
173     private static extern LinkIndex GetLinkerIndex(LinkIndex link);

```

```

173     /// <summary>
174     /// <para>
175     /// Gets the target index using the specified link.
176     /// </para>
177     /// <para></para>
178     /// </summary>
179     /// <param name="link">
180     /// <para>The link.</para>
181     /// <para></para>
182     /// </param>
183     /// <returns>
184     /// <para>The link index</para>
185     /// <para></para>
186     /// </returns>
187     [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
188     private static extern LinkIndex GetTargetIndex(LinkIndex link);
189
190     /// <summary>
191     /// <para>
192     /// Gets the first referer by source index using the specified link.
193     /// </para>
194     /// <para></para>
195     /// </summary>
196     /// <param name="link">
197     /// <para>The link.</para>
198     /// <para></para>
199     /// </param>
200     /// <returns>
201     /// <para>The link index</para>
202     /// <para></para>
203     /// </returns>
204     [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
205     private static extern LinkIndex GetFirstRefererBySourceIndex(LinkIndex link);
206
207     /// <summary>
208     /// <para>
209     /// Gets the first referer by linker index using the specified link.
210     /// </para>
211     /// <para></para>
212     /// </summary>
213     /// <param name="link">
214     /// <para>The link.</para>
215     /// <para></para>
216     /// </param>
217     /// <returns>
218     /// <para>The link index</para>
219     /// <para></para>
220     /// </returns>
221     [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
222     private static extern LinkIndex GetFirstRefererByLinkerIndex(LinkIndex link);
223
224     /// <summary>
225     /// <para>
226     /// Gets the first referer by target index using the specified link.
227     /// </para>
228     /// <para></para>
229     /// </summary>
230     /// <param name="link">
231     /// <para>The link.</para>
232     /// <para></para>
233     /// </param>
234     /// <returns>
235     /// <para>The link index</para>
236     /// <para></para>
237     /// </returns>
238     [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
239     private static extern LinkIndex GetFirstRefererByTargetIndex(LinkIndex link);
240
241     /// <summary>
242     /// <para>
243     /// Gets the time using the specified link.
244     /// </para>
245     /// <para></para>
246     /// </summary>
247     /// <param name="link">
248     /// <para>The link.</para>
249     /// <para></para>
250     /// </param>

```

```

251     /// <returns>
252     /// <para>The int</para>
253     /// <para></para>
254     /// </returns>
255     [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
256     private static extern Int GetTime(LinkIndex link);
257
258     /// <summary>
259     /// <para>
260     /// Creates the link using the specified source.
261     /// </para>
262     /// <para></para>
263     /// </summary>
264     /// <param name="source">
265     /// <para>The source.</para>
266     /// <para></para>
267     /// </param>
268     /// <param name="linker">
269     /// <para>The linker.</para>
270     /// <para></para>
271     /// </param>
272     /// <param name="target">
273     /// <para>The target.</para>
274     /// <para></para>
275     /// </param>
276     /// <returns>
277     /// <para>The link index</para>
278     /// <para></para>
279     /// </returns>
280     [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
281     private static extern LinkIndex CreateLink(LinkIndex source, LinkIndex linker, LinkIndex
        ↪ target);
282
283     /// <summary>
284     /// <para>
285     /// Updates the link using the specified link.
286     /// </para>
287     /// <para></para>
288     /// </summary>
289     /// <param name="link">
290     /// <para>The link.</para>
291     /// <para></para>
292     /// </param>
293     /// <param name="newSource">
294     /// <para>The new source.</para>
295     /// <para></para>
296     /// </param>
297     /// <param name="newLinker">
298     /// <para>The new linker.</para>
299     /// <para></para>
300     /// </param>
301     /// <param name="newTarget">
302     /// <para>The new target.</para>
303     /// <para></para>
304     /// </param>
305     /// <returns>
306     /// <para>The link index</para>
307     /// <para></para>
308     /// </returns>
309     [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
310     private static extern LinkIndex UpdateLink(LinkIndex link, LinkIndex newSource,
        ↪ LinkIndex newLinker, LinkIndex newTarget);
311
312     /// <summary>
313     /// <para>
314     /// Deletes the link using the specified link.
315     /// </para>
316     /// <para></para>
317     /// </summary>
318     /// <param name="link">
319     /// <para>The link.</para>
320     /// <para></para>
321     /// </param>
322     [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
323     private static extern void DeleteLink(LinkIndex link);
324
325     /// <summary>
326     /// <para>

```

```

327     /// Replaces the link using the specified link.
328     /// </para>
329     /// <para></para>
330     /// </summary>
331     /// <param name="link">
332     /// <para>The link.</para>
333     /// <para></para>
334     /// </param>
335     /// <param name="replacement">
336     /// <para>The replacement.</para>
337     /// <para></para>
338     /// </param>
339     /// <returns>
340     /// <para>The link index</para>
341     /// <para></para>
342     /// </returns>
343     [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
344     private static extern LinkIndex ReplaceLink(LinkIndex link, LinkIndex replacement);
345
346     /// <summary>
347     /// <para>
348     /// Searches the link using the specified source.
349     /// </para>
350     /// <para></para>
351     /// </summary>
352     /// <param name="source">
353     /// <para>The source.</para>
354     /// <para></para>
355     /// </param>
356     /// <param name="linker">
357     /// <para>The linker.</para>
358     /// <para></para>
359     /// </param>
360     /// <param name="target">
361     /// <para>The target.</para>
362     /// <para></para>
363     /// </param>
364     /// <returns>
365     /// <para>The link index</para>
366     /// <para></para>
367     /// </returns>
368     [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
369     private static extern LinkIndex SearchLink(LinkIndex source, LinkIndex linker, LinkIndex
        ↪ target);
370
371     /// <summary>
372     /// <para>
373     /// Gets the mapped link using the specified mapped index.
374     /// </para>
375     /// <para></para>
376     /// </summary>
377     /// <param name="mappedIndex">
378     /// <para>The mapped index.</para>
379     /// <para></para>
380     /// </param>
381     /// <returns>
382     /// <para>The link index</para>
383     /// <para></para>
384     /// </returns>
385     [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
386     private static extern LinkIndex GetMappedLink(Int mappedIndex);
387
388     /// <summary>
389     /// <para>
390     /// Sets the mapped link using the specified mapped index.
391     /// </para>
392     /// <para></para>
393     /// </summary>
394     /// <param name="mappedIndex">
395     /// <para>The mapped index.</para>
396     /// <para></para>
397     /// </param>
398     /// <param name="linkIndex">
399     /// <para>The link index.</para>
400     /// <para></para>
401     /// </param>
402     [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
403     private static extern void SetMappedLink(Int mappedIndex, LinkIndex linkIndex);

```



```

404     /// <summary>
405     /// <para>
406     /// Opens the links using the specified filename.
407     /// </para>
408     /// <para></para>
409     /// </summary>
410     /// <param name="filename">
411     /// <para>The filename.</para>
412     /// <para></para>
413     /// </param>
414     /// <returns>
415     /// <para>The int</para>
416     /// <para></para>
417     /// </returns>
418     [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
419     private static extern Int OpenLinks(string filename);
420
421     /// <summary>
422     /// <para>
423     /// Closes the links.
424     /// </para>
425     /// <para></para>
426     /// </summary>
427     /// <returns>
428     /// <para>The int</para>
429     /// <para></para>
430     /// </returns>
431     [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
432     private static extern Int CloseLinks();
433
434     #endregion
435
436     #region Referers Count Selectors
437
438     /// <summary>
439     /// <para>
440     /// Gets the link number of referers by source using the specified link.
441     /// </para>
442     /// <para></para>
443     /// </summary>
444     /// <param name="link">
445     /// <para>The link.</para>
446     /// <para></para>
447     /// </param>
448     /// <returns>
449     /// <para>The link index</para>
450     /// <para></para>
451     /// </returns>
452     [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
453     private static extern LinkIndex GetLinkNumberOfReferersBySource(LinkIndex link);
454
455     /// <summary>
456     /// <para>
457     /// Gets the link number of referers by linker using the specified link.
458     /// </para>
459     /// <para></para>
460     /// </summary>
461     /// <param name="link">
462     /// <para>The link.</para>
463     /// <para></para>
464     /// </param>
465     /// <returns>
466     /// <para>The link index</para>
467     /// <para></para>
468     /// </returns>
469     [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
470     private static extern LinkIndex GetLinkNumberOfReferersByLinker(LinkIndex link);
471
472     /// <summary>
473     /// <para>
474     /// Gets the link number of referers by target using the specified link.
475     /// </para>
476     /// <para></para>
477     /// </summary>
478     /// <param name="link">
479     /// <para>The link.</para>
480     /// <para></para>
481     /// </param>

```

```

482     /// </param>
483     /// <returns>
484     /// <para>The link index</para>
485     /// <para></para>
486     /// </returns>
487     [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
488     private static extern LinkIndex GetLinkNumberOfReferersByTarget(LinkIndex link);
489
490 #endregion
491
492 #region Referers Walkers
493 private delegate void Visitor(LinkIndex link);
494 private delegate Int StopableVisitor(LinkIndex link);
495
496     /// <summary>
497     /// <para>
498     /// Walks the through all referers by source using the specified root.
499     /// </para>
500     /// <para></para>
501     /// </summary>
502     /// <param name="root">
503     /// <para>The root.</para>
504     /// <para></para>
505     /// </param>
506     /// <param name="action">
507     /// <para>The action.</para>
508     /// <para></para>
509     /// </param>
510     [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
511     private static extern void WalkThroughAllReferersBySource(LinkIndex root, Visitor
    ↪ action);
512
513     /// <summary>
514     /// <para>
515     /// Walks the through referers by source using the specified root.
516     /// </para>
517     /// <para></para>
518     /// </summary>
519     /// <param name="root">
520     /// <para>The root.</para>
521     /// <para></para>
522     /// </param>
523     /// <param name="func">
524     /// <para>The func.</para>
525     /// <para></para>
526     /// </param>
527     /// <returns>
528     /// <para>The int</para>
529     /// <para></para>
530     /// </returns>
531     [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
532     private static extern int WalkThroughReferersBySource(LinkIndex root, StopableVisitor
    ↪ func);
533
534     /// <summary>
535     /// <para>
536     /// Walks the through all referers by linker using the specified root.
537     /// </para>
538     /// <para></para>
539     /// </summary>
540     /// <param name="root">
541     /// <para>The root.</para>
542     /// <para></para>
543     /// </param>
544     /// <param name="action">
545     /// <para>The action.</para>
546     /// <para></para>
547     /// </param>
548     [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
549     private static extern void WalkThroughAllReferersByLinker(LinkIndex root, Visitor
    ↪ action);
550
551     /// <summary>
552     /// <para>
553     /// Walks the through referers by linker using the specified root.
554     /// </para>
555     /// <para></para>
556     /// </summary>

```

```

557     /// <param name="root">
558     /// <para>The root.</para>
559     /// <para></para>
560     /// </param>
561     /// <param name="func">
562     /// <para>The func.</para>
563     /// <para></para>
564     /// </param>
565     /// <returns>
566     /// <para>The int</para>
567     /// <para></para>
568     /// </returns>
569     [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
570     private static extern int WalkThroughReferersByLinker(LinkIndex root, StopableVisitor
        ↪ func);

571
572     /// <summary>
573     /// <para>
574     /// Walks the through all referers by target using the specified root.
575     /// </para>
576     /// <para></para>
577     /// </summary>
578     /// <param name="root">
579     /// <para>The root.</para>
580     /// <para></para>
581     /// </param>
582     /// <param name="action">
583     /// <para>The action.</para>
584     /// <para></para>
585     /// </param>
586     [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
587     private static extern void WalkThroughAllReferersByTarget(LinkIndex root, Visitor
        ↪ action);

588
589     /// <summary>
590     /// <para>
591     /// Walks the through referers by target using the specified root.
592     /// </para>
593     /// <para></para>
594     /// </summary>
595     /// <param name="root">
596     /// <para>The root.</para>
597     /// <para></para>
598     /// </param>
599     /// <param name="func">
600     /// <para>The func.</para>
601     /// <para></para>
602     /// </param>
603     /// <returns>
604     /// <para>The int</para>
605     /// <para></para>
606     /// </returns>
607     [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
608     private static extern int WalkThroughReferersByTarget(LinkIndex root, StopableVisitor
        ↪ func);

609
610     /// <summary>
611     /// <para>
612     /// Walks the through all links using the specified action.
613     /// </para>
614     /// <para></para>
615     /// </summary>
616     /// <param name="action">
617     /// <para>The action.</para>
618     /// <para></para>
619     /// </param>
620     [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
621     private static extern void WalkThroughAllLinks(Visitor action);
622
623     /// <summary>
624     /// <para>
625     /// Walks the through links using the specified func.
626     /// </para>
627     /// <para></para>
628     /// </summary>
629     /// <param name="func">
630     /// <para>The func.</para>
631     /// <para></para>

```

```

632    /// </param>
633    /// <returns>
634    /// <para>The int</para>
635    /// <para></para>
636    /// </returns>
637    [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
638    private static extern int WalkThroughLinks(StopableVisitor func);
639
640    #endregion
641
642    #endregion
643
644    #region Constains
645
646    /// <summary>
647    /// <para>
648    /// The itself.
649    /// </para>
650    /// <para></para>
651    /// </summary>
652    public static readonly Link Itself = null;
653    /// <summary>
654    /// <para>
655    /// The continue.
656    /// </para>
657    /// <para></para>
658    /// </summary>
659    public static readonly bool Continue = true;
660    /// <summary>
661    /// <para>
662    /// The stop.
663    /// </para>
664    /// <para></para>
665    /// </summary>
666    public static readonly bool Stop;
667
668    #endregion
669
670    #region Static Fields
671    private static readonly object _lockObject = new object();
672    private static volatile int _memoryManagerIsReady;
673    private static readonly Dictionary<ulong, long> _linkToMappingIndex = new
        ↪ Dictionary<ulong, long>();
674
675    #endregion
676
677    #region Fields
678
679    /// <summary>
680    /// <para>
681    /// The link.
682    /// </para>
683    /// <para></para>
684    /// </summary>
685    [DebuggerBrowsable(DebuggerBrowsableState.Never)]
686    private readonly LinkIndex _link;
687
688    #endregion
689
690    #region Properties
691
692    /// <summary>
693    /// <para>
694    /// Gets the source value.
695    /// </para>
696    /// <para></para>
697    /// </summary>
698    [DebuggerBrowsable(DebuggerBrowsableState.Never)]
699    public Link Source => GetSourceIndex(_link);
700
701    /// <summary>
702    /// <para>
703    /// Gets the linker value.
704    /// </para>
705    /// <para></para>
706    /// </summary>
707    [DebuggerBrowsable(DebuggerBrowsableState.Never)]
708    public Link Linker => GetLinkerIndex(_link);
709
710    /// <summary>

```

```

711     /// <para>
712     /// Gets the target value.
713     /// </para>
714     /// <para></para>
715     /// </summary>
716     [DebuggerBrowsable(DebuggerBrowsableState.Never)]
717     public Link Target => GetTargetIndex(_link);
718
719     /// <summary>
720     /// <para>
721     /// Gets the first referer by source value.
722     /// </para>
723     /// <para></para>
724     /// </summary>
725     [DebuggerBrowsable(DebuggerBrowsableState.Never)]
726     public Link FirstRefererBySource => GetFirstRefererBySourceIndex(_link);
727
728     /// <summary>
729     /// <para>
730     /// Gets the first referer by linker value.
731     /// </para>
732     /// <para></para>
733     /// </summary>
734     [DebuggerBrowsable(DebuggerBrowsableState.Never)]
735     public Link FirstRefererByLinker => GetFirstRefererByLinkerIndex(_link);
736
737     /// <summary>
738     /// <para>
739     /// Gets the first referer by target value.
740     /// </para>
741     /// <para></para>
742     /// </summary>
743     [DebuggerBrowsable(DebuggerBrowsableState.Never)]
744     public Link FirstRefererByTarget => GetFirstRefererByTargetIndex(_link);
745
746     /// <summary>
747     /// <para>
748     /// Gets the referers by source count value.
749     /// </para>
750     /// <para></para>
751     /// </summary>
752     [DebuggerBrowsable(DebuggerBrowsableState.Never)]
753     public Int ReferersBySourceCount => (Int)GetLinkNumberOfReferersBySource(_link);
754
755     /// <summary>
756     /// <para>
757     /// Gets the referers by linker count value.
758     /// </para>
759     /// <para></para>
760     /// </summary>
761     [DebuggerBrowsable(DebuggerBrowsableState.Never)]
762     public Int ReferersByLinkerCount => (Int)GetLinkNumberOfReferersByLinker(_link);
763
764     /// <summary>
765     /// <para>
766     /// Gets the referers by target count value.
767     /// </para>
768     /// <para></para>
769     /// </summary>
770     [DebuggerBrowsable(DebuggerBrowsableState.Never)]
771     public Int ReferersByTargetCount => (Int)GetLinkNumberOfReferersByTarget(_link);
772
773     /// <summary>
774     /// <para>
775     /// Gets the total referers value.
776     /// </para>
777     /// <para></para>
778     /// </summary>
779     [DebuggerBrowsable(DebuggerBrowsableState.Never)]
780     public Int TotalReferers => (Int)GetLinkNumberOfReferersBySource(_link) +
781         ↳ (Int)GetLinkNumberOfReferersByLinker(_link) +
782         ↳ (Int)GetLinkNumberOfReferersByTarget(_link);
783
784     /// <summary>
785     /// <para>
786     /// Gets the timestamp value.
787     /// </para>
788     /// <para></para>

```

```

787     /// </summary>
788     [DebuggerBrowsable(DebuggerBrowsableState.Never)]
789     public DateTime Timestamp => DateTime.FromFileTimeUtc(GetTime(_link));
790
791     #endregion
792
793     #region Infrastructure
794
795     /// <summary>
796     /// <para>
797     /// Initializes a new <see cref="Link"/> instance.
798     /// </para>
799     /// <para></para>
800     /// </summary>
801     /// <param name="link">
802     /// <para>A link.</para>
803     /// <para></para>
804     /// </param>
805     public Link(LinkIndex link) => _link = link;
806
807     /// <summary>
808     /// <para>
809     /// Starts the memory manager using the specified storage filename.
810     /// </para>
811     /// <para></para>
812     /// </summary>
813     /// <param name="storageFilename">
814     /// <para>The storage filename.</para>
815     /// <para></para>
816     /// </param>
817     /// <exception cref="InvalidOperationException">
818     /// <para>Файл ({storageFilename}) хранилища не удалось открыть.</para>
819     /// <para></para>
820     /// </exception>
821     public static void StartMemoryManager(string storageFilename)
822     {
823         lock (_lockObject)
824         {
825             if (_memoryManagerIsReady == default)
826             {
827                 if (OpenLinks(storageFilename) == 0)
828                 {
829                     throw new InvalidOperationException($"Файл ({storageFilename})
830                         ↪ хранилища не удалось открыть.");
831                 }
832                 Interlocked.Exchange(ref _memoryManagerIsReady, 1);
833             }
834         }
835
836     /// <summary>
837     /// <para>
838     /// Stops the memory manager.
839     /// </para>
840     /// <para></para>
841     /// </summary>
842     /// <exception cref="InvalidOperationException">
843     /// <para>Файл хранилища не удалось закрыть. Возможно он был уже закрыт, или не
844     /// ↪ открывался вовсе.</para>
845     /// <para></para>
846     /// </exception>
847     public static void StopMemoryManager()
848     {
849         lock (_lockObject)
850         {
851             if (_memoryManagerIsReady != default)
852             {
853                 if (CloseLinks() == 0)
854                 {
855                     throw new InvalidOperationException("Файл хранилища не удалось закрыть.
856                         ↪ Возможно он был уже закрыт, или не открывался вовсе.");
857                 }
858                 Interlocked.Exchange(ref _memoryManagerIsReady, 0);
859             }
860         }
861     }

```

```

861 public static implicit operator LinkIndex?(Link link) => link._link == 0 ?
    ↳ (LinkIndex?)null : link._link;
862
863 public static implicit operator Link(LinkIndex? link) => new Link(link ?? 0);
864
865 public static implicit operator Int(Link link) => (Int)link._link;
866
867 public static implicit operator Link(Int link) => new Link((LinkIndex)link);
868
869 public static implicit operator LinkIndex(Link link) => link._link;
870
871 public static implicit operator Link(LinkIndex link) => new Link(link);
872
873 public static explicit operator Link(List<Link> links) => LinkConverter.FromList(links);
874
875 public static explicit operator Link(Link[] links) => LinkConverter.FromList(links);
876
877 public static explicit operator Link(string @string) =>
    ↳ LinkConverter.FromString(@string);
878
879 public static bool operator ==(Link first, Link second) => first.Equals(second);
880
881 public static bool operator !=(Link first, Link second) => !first.Equals(second);
882
883 public static Link operator &(Link first, Link second) => Create(first, Net.And, second);
884
885 /// <summary>
886 /// <para>
887 /// Determines whether this instance equals.
888 /// </para>
889 /// <para></para>
890 /// </summary>
891 /// <param name="obj">
892 /// <para>The obj.</para>
893 /// <para></para>
894 /// </param>
895 /// <returns>
896 /// <para>The bool</para>
897 /// <para></para>
898 /// </returns>
899 public override bool Equals(object obj) => obj is Link link ? Equals(link) : false;
900
901 /// <summary>
902 /// <para>
903 /// Determines whether this instance equals.
904 /// </para>
905 /// <para></para>
906 /// </summary>
907 /// <param name="other">
908 /// <para>The other.</para>
909 /// <para></para>
910 /// </param>
911 /// <returns>
912 /// <para>The bool</para>
913 /// <para></para>
914 /// </returns>
915 public bool Equals(Link other) => _link == other._link || (LinkDoesNotExist(_link) &&
    ↳ LinkDoesNotExist(other._link));
916
917 /// <summary>
918 /// <para>
919 /// Gets the hash code.
920 /// </para>
921 /// <para></para>
922 /// </summary>
923 /// <returns>
924 /// <para>The int</para>
925 /// <para></para>
926 /// </returns>
927 public override int GetHashCode() => base.GetHashCode();
928 private static bool LinkDoesNotExist(LinkIndex link) => link == 0 ||
    ↳ GetLinkerIndex(link) == 0;
929 private static bool LinkWasDeleted(LinkIndex link) => link != 0 && GetLinkerIndex(link)
    ↳ == 0;
930 private bool IsMatchingTo(Link source, Link linker, Link target)
931     => ((Source == this && source == null) || (Source == source))
932     && ((Linker == this && linker == null) || (Linker == linker))
933     && ((Target == this && target == null) || (Target == target));

```

```

934
935     /// <summary>
936     /// <para>
937     /// Returns the index.
938     /// </para>
939     /// <para></para>
940     /// </summary>
941     /// <returns>
942     /// <para>The link index</para>
943     /// <para></para>
944     /// </returns>
945     public LinkIndex ToIndex() => _link;
946
947     /// <summary>
948     /// <para>
949     /// Returns the int.
950     /// </para>
951     /// <para></para>
952     /// </summary>
953     /// <returns>
954     /// <para>The int</para>
955     /// <para></para>
956     /// </returns>
957     public Int ToInt() => (Int)_link;
958
959     #endregion
960
961     #region Basic Operations
962
963     /// <summary>
964     /// <para>
965     /// Creates the source.
966     /// </para>
967     /// <para></para>
968     /// </summary>
969     /// <param name="source">
970     /// <para>The source.</para>
971     /// <para></para>
972     /// </param>
973     /// <param name="linker">
974     /// <para>The linker.</para>
975     /// <para></para>
976     /// </param>
977     /// <param name="target">
978     /// <para>The target.</para>
979     /// <para></para>
980     /// </param>
981     /// <exception cref="InvalidOperationException">
982     /// <para>Менеджер памяти ещё не готов.</para>
983     /// <para></para>
984     /// </exception>
985     /// <exception cref="InvalidOperationException">
986     /// <para>Невозможно создать связь.</para>
987     /// <para></para>
988     /// </exception>
989     /// <exception cref="ArgumentException">
990     /// <para>Удалённая связь не может использоваться в качестве значения. </para>
991     /// <para></para>
992     /// </exception>
993     /// <exception cref="ArgumentException">
994     /// <para>Удалённая связь не может использоваться в качестве значения. </para>
995     /// <para></para>
996     /// </exception>
997     /// <exception cref="ArgumentException">
998     /// <para>Удалённая связь не может использоваться в качестве значения. </para>
999     /// <para></para>
1000    /// </exception>
1001    /// <returns>
1002    /// <para>The link.</para>
1003    /// <para></para>
1004    /// </returns>
1005    public static Link Create(Link source, Link linker, Link target)
1006    {
1007        if (_memoryManagerIsReady == default)
1008        {
1009            throw new InvalidOperationException("Менеджер памяти ещё не готов.");
1010        }
1011        if (LinkWasDeleted(source))

```



```

1012     {
1013         throw new ArgumentException("Удалённая связь не может использоваться в качестве
↪ значения.", nameof(source));
1014     }
1015     if (LinkWasDeleted(linker))
1016     {
1017         throw new ArgumentException("Удалённая связь не может использоваться в качестве
↪ значения.", nameof(linker));
1018     }
1019     if (LinkWasDeleted(target))
1020     {
1021         throw new ArgumentException("Удалённая связь не может использоваться в качестве
↪ значения.", nameof(target));
1022     }
1023     Link link = CreateLink(source, linker, target);
1024     if (link == null)
1025     {
1026         throw new InvalidOperationException("Невозможно создать связь.");
1027     }
1028     CreatedEvent.Invoke(new LinkDefinition(link));
1029     return link;
1030 }
1031
1032 /// <summary>
1033 /// <para>
1034 /// Restores the index.
1035 /// </para>
1036 /// <para></para>
1037 /// </summary>
1038 /// <param name="index">
1039 /// <para>The index.</para>
1040 /// <para></para>
1041 /// </param>
1042 /// <returns>
1043 /// <para>The link</para>
1044 /// <para></para>
1045 /// </returns>
1046 public static Link Restore(Int index) => Restore((LinkIndex)index);
1047
1048 /// <summary>
1049 /// <para>
1050 /// Restores the index.
1051 /// </para>
1052 /// <para></para>
1053 /// </summary>
1054 /// <param name="index">
1055 /// <para>The index.</para>
1056 /// <para></para>
1057 /// </param>
1058 /// <exception cref="InvalidOperationException">
1059 /// <para>Менеджер памяти ещё не готов.</para>
1060 /// <para></para>
1061 /// </exception>
1062 /// <exception cref="InvalidOperationException">
1063 /// <para>Связь с указанным адресом удалена, либо не существовала.</para>
1064 /// <para></para>
1065 /// </exception>
1066 /// <exception cref="ArgumentException">
1067 /// <para>У связи не может быть нулевого адреса.</para>
1068 /// <para></para>
1069 /// </exception>
1070 /// <exception cref="InvalidOperationException">
1071 /// <para>Указатель не является корректным. </para>
1072 /// <para></para>
1073 /// </exception>
1074 /// <returns>
1075 /// <para>The link</para>
1076 /// <para></para>
1077 /// </returns>
1078 public static Link Restore(LinkIndex index)
1079 {
1080     if (_memoryManagerIsReady == default)
1081     {
1082         throw new InvalidOperationException("Менеджер памяти ещё не готов.");
1083     }
1084     if (index == 0)
1085     {
1086         throw new ArgumentException("У связи не может быть нулевого адреса.");

```

```

1087     }
1088     try
1089     {
1090         Link link = index;
1091         if (LinkDoesNotExist(link))
1092         {
1093             throw new InvalidOperationException("Связь с указанным адресом удалена, либо
↪             не существовала.");
1094         }
1095         return link;
1096     }
1097     catch (Exception ex)
1098     {
1099         throw new InvalidOperationException("Указатель не является корректным.", ex);
1100     }
1101 }
1102
1103 /// <summary>
1104 /// <para>
1105 /// Creates the mapped using the specified source.
1106 /// </para>
1107 /// <para></para>
1108 /// </summary>
1109 /// <param name="source">
1110 /// <para>The source.</para>
1111 /// <para></para>
1112 /// </param>
1113 /// <param name="linker">
1114 /// <para>The linker.</para>
1115 /// <para></para>
1116 /// </param>
1117 /// <param name="target">
1118 /// <para>The target.</para>
1119 /// <para></para>
1120 /// </param>
1121 /// <param name="mappingIndex">
1122 /// <para>The mapping index.</para>
1123 /// <para></para>
1124 /// </param>
1125 /// <returns>
1126 /// <para>The link</para>
1127 /// <para></para>
1128 /// </returns>
1129 public static Link CreateMapped(Link source, Link linker, Link target, object
↪ mappingIndex) => CreateMapped(source, linker, target, Convert.ToInt64(mappingIndex));
1130
1131 /// <summary>
1132 /// <para>
1133 /// Creates the mapped using the specified source.
1134 /// </para>
1135 /// <para></para>
1136 /// </summary>
1137 /// <param name="source">
1138 /// <para>The source.</para>
1139 /// <para></para>
1140 /// </param>
1141 /// <param name="linker">
1142 /// <para>The linker.</para>
1143 /// <para></para>
1144 /// </param>
1145 /// <param name="target">
1146 /// <para>The target.</para>
1147 /// <para></para>
1148 /// </param>
1149 /// <param name="mappingIndex">
1150 /// <para>The mapping index.</para>
1151 /// <para></para>
1152 /// </param>
1153 /// <exception cref="InvalidOperationException">
1154 /// <para>Менеджер памяти ещё не готов.</para>
1155 /// <para></para>
1156 /// </exception>
1157 /// <exception cref="InvalidOperationException">
1158 /// <para>Существующая привязанная связь не соответствует указанным Source, Linker и
↪ Target.</para>
1159 /// <para></para>
1160 /// </exception>
1161 /// <exception cref="InvalidOperationException">

```

```

1162     /// <para>Установить привязанную связь не удалось.</para>
1163     /// <para></para>
1164     /// </exception>
1165     /// <returns>
1166     /// <para>The mapped link.</para>
1167     /// <para></para>
1168     /// </returns>
1169     public static Link CreateMapped(Link source, Link linker, Link target, Int mappingIndex)
1170     {
1171         if (_memoryManagerIsReady == default)
1172         {
1173             throw new InvalidOperationException("Менеджер памяти ещё не готов.");
1174         }
1175         Link mappedLink = GetMappedLink(mappingIndex);
1176         if (mappedLink == null)
1177         {
1178             mappedLink = Create(source, linker, target);
1179             SetMappedLink(mappingIndex, mappedLink);
1180             if (GetMappedLink(mappingIndex) != mappedLink)
1181             {
1182                 throw new InvalidOperationException("Установить привязанную связь не
1183                     ↪ удалось.");
1184             }
1185         }
1186         else if (!mappedLink.IsMatchingTo(source, linker, target))
1187         {
1188             throw new InvalidOperationException("Существующая привязанная связь не
1189                 ↪ соответствует указанным Source, Linker и Target.");
1190         }
1191         _linkToMappingIndex[mappedLink] = mappingIndex;
1192         return mappedLink;
1193     }
1194     /// <summary>
1195     /// <para>
1196     /// Determines whether try set mapped.
1197     /// </para>
1198     /// </summary>
1199     /// <param name="link">
1200     /// <para>The link.</para>
1201     /// <para></para>
1202     /// </param>
1203     /// <param name="mappingIndex">
1204     /// <para>The mapping index.</para>
1205     /// <para></para>
1206     /// </param>
1207     /// <param name="rewrite">
1208     /// <para>The rewrite.</para>
1209     /// <para></para>
1210     /// </param>
1211     /// <returns>
1212     /// <para>The bool</para>
1213     /// <para></para>
1214     /// </returns>
1215     public static bool TrySetMapped(Link link, Int mappingIndex, bool rewrite = false)
1216     {
1217         Link mappedLink = GetMappedLink(mappingIndex);
1218
1219         if (mappedLink == null || rewrite)
1220         {
1221             mappedLink = link;
1222             SetMappedLink(mappingIndex, mappedLink);
1223             if (GetMappedLink(mappingIndex) != mappedLink)
1224             {
1225                 return false;
1226             }
1227         }
1228         else if (!mappedLink.IsMatchingTo(link.Source, link.Linker, link.Target))
1229         {
1230             return false;
1231         }
1232         _linkToMappingIndex[mappedLink] = mappingIndex;
1233         return true;
1234     }
1235     /// <summary>
1236     /// <para>
1237

```

```

1238     /// Gets the mapped using the specified mapping index.
1239     /// </para>
1240     /// <para></para>
1241     /// </summary>
1242     /// <param name="mappingIndex">
1243     /// <para>The mapping index.</para>
1244     /// <para></para>
1245     /// </param>
1246     /// <returns>
1247     /// <para>The link</para>
1248     /// <para></para>
1249     /// </returns>
1250     public static Link GetMapped(object mappingIndex) =>
1251         ↪ GetMapped(Convert.ToInt64(mappingIndex));
1252
1253     /// <summary>
1254     /// <para>
1255     /// Gets the mapped using the specified mapping index.
1256     /// </para>
1257     /// <para></para>
1258     /// </summary>
1259     /// <param name="mappingIndex">
1260     /// <para>The mapping index.</para>
1261     /// <para></para>
1262     /// </param>
1263     /// <exception cref="InvalidOperationException">
1264     /// <para>Mapped link with index {mappingIndex} is not set.</para>
1265     /// <para></para>
1266     /// </exception>
1267     /// <returns>
1268     /// <para>The mapped link.</para>
1269     /// <para></para>
1270     /// </returns>
1271     public static Link GetMapped(Int mappingIndex)
1272     {
1273         if (!TryGetMapped(mappingIndex, out Link mappedLink))
1274         {
1275             throw new InvalidOperationException($"Mapped link with index {mappingIndex} is
1276                 ↪ not set.");
1277         }
1278         return mappedLink;
1279     }
1280
1281     /// <summary>
1282     /// <para>
1283     /// Gets the mapped or default using the specified mapping index.
1284     /// </para>
1285     /// <para></para>
1286     /// </summary>
1287     /// <param name="mappingIndex">
1288     /// <para>The mapping index.</para>
1289     /// <para></para>
1290     /// </param>
1291     /// <returns>
1292     /// <para>The mapped link.</para>
1293     /// <para></para>
1294     /// </returns>
1295     public static Link GetMappedOrDefault(object mappingIndex)
1296     {
1297         TryGetMapped(mappingIndex, out Link mappedLink);
1298         return mappedLink;
1299     }
1300
1301     /// <summary>
1302     /// <para>
1303     /// Gets the mapped or default using the specified mapping index.
1304     /// </para>
1305     /// <para></para>
1306     /// </summary>
1307     /// <param name="mappingIndex">
1308     /// <para>The mapping index.</para>
1309     /// <para></para>
1310     /// </param>
1311     /// <returns>
1312     /// <para>The mapped link.</para>
1313     /// <para></para>
1314     /// </returns>
1315     public static Link GetMappedOrDefault(Int mappingIndex)

```

```

1314 {
1315     TryGetMapped(mappingIndex, out Link mappedLink);
1316     return mappedLink;
1317 }
1318
1319 /// <summary>
1320 /// <para>
1321 /// Determines whether try get mapped.
1322 /// </para>
1323 /// <para></para>
1324 /// </summary>
1325 /// <param name="mappingIndex">
1326 /// <para>The mapping index.</para>
1327 /// <para></para>
1328 /// </param>
1329 /// <param name="mappedLink">
1330 /// <para>The mapped link.</para>
1331 /// <para></para>
1332 /// </param>
1333 /// <returns>
1334 /// <para>The bool</para>
1335 /// <para></para>
1336 /// </returns>
1337 public static bool TryGetMapped(object mappingIndex, out Link mappedLink) =>
1338     ↪ TryGetMapped(Convert.ToInt64(mappingIndex), out mappedLink);
1339
1340 /// <summary>
1341 /// <para>
1342 /// Determines whether try get mapped.
1343 /// </para>
1344 /// <para></para>
1345 /// </summary>
1346 /// <param name="mappingIndex">
1347 /// <para>The mapping index.</para>
1348 /// <para></para>
1349 /// </param>
1350 /// <param name="mappedLink">
1351 /// <para>The mapped link.</para>
1352 /// <para></para>
1353 /// </param>
1354 /// <exception cref="InvalidOperationException">
1355 /// <para>Менеджер памяти ещё не готов.</para>
1356 /// <para></para>
1357 /// </exception>
1358 /// <returns>
1359 /// <para>The bool</para>
1360 /// <para></para>
1361 /// </returns>
1362 public static bool TryGetMapped(Int mappingIndex, out Link mappedLink)
1363 {
1364     if (_memoryManagerIsReady == default)
1365     {
1366         throw new InvalidOperationException("Менеджер памяти ещё не готов.");
1367     }
1368     mappedLink = GetMappedLink(mappingIndex);
1369     if (mappedLink != null)
1370     {
1371         _linkToMappingIndex[mappedLink] = mappingIndex;
1372     }
1373     return mappedLink != null;
1374 }
1375
1376 /// <summary>
1377 /// <para>
1378 /// Updates the link.
1379 /// </para>
1380 /// <para></para>
1381 /// </summary>
1382 /// <param name="link">
1383 /// <para>The link.</para>
1384 /// <para></para>
1385 /// </param>
1386 /// <param name="newSource">
1387 /// <para>The new source.</para>
1388 /// <para></para>
1389 /// </param>
1390 /// <param name="newLinker">
1391 /// <para>The new linker.</para>

```

```

1391    /// <para></para>
1392    /// </param>
1393    /// <param name="newTarget">
1394    /// <para>The new target.</para>
1395    /// <para></para>
1396    /// </param>
1397    /// <returns>
1398    /// <para>The link.</para>
1399    /// <para></para>
1400    /// </returns>
1401    public static Link Update(Link link, Link newSource, Link newLinker, Link newTarget)
1402    {
1403        Update(ref link, newSource, newLinker, newTarget);
1404        return link;
1405    }
1406
1407    /// <summary>
1408    /// <para>
1409    /// Updates the link.
1410    /// </para>
1411    /// <para></para>
1412    /// </summary>
1413    /// <param name="link">
1414    /// <para>The link.</para>
1415    /// <para></para>
1416    /// </param>
1417    /// <param name="newSource">
1418    /// <para>The new source.</para>
1419    /// <para></para>
1420    /// </param>
1421    /// <param name="newLinker">
1422    /// <para>The new linker.</para>
1423    /// <para></para>
1424    /// </param>
1425    /// <param name="newTarget">
1426    /// <para>The new target.</para>
1427    /// <para></para>
1428    /// </param>
1429    /// <exception cref="InvalidOperationException">
1430    /// <para>Менеджер памяти ещё не готов.</para>
1431    /// <para></para>
1432    /// </exception>
1433    /// <exception cref="ArgumentException">
1434    /// <para>Нельзя обновить несуществующую связь. </para>
1435    /// <para></para>
1436    /// </exception>
1437    /// <exception cref="ArgumentException">
1438    /// <para>Удалённая связь не может использоваться в качестве нового значения. </para>
1439    /// <para></para>
1440    /// </exception>
1441    /// <exception cref="ArgumentException">
1442    /// <para>Удалённая связь не может использоваться в качестве нового значения. </para>
1443    /// <para></para>
1444    /// </exception>
1445    /// <exception cref="ArgumentException">
1446    /// <para>Удалённая связь не может использоваться в качестве нового значения. </para>
1447    /// <para></para>
1448    /// </exception>
1449    public static void Update(ref Link link, Link newSource, Link newLinker, Link newTarget)
1450    {
1451        if (_memoryManagerIsReady == default)
1452        {
1453            throw new InvalidOperationException("Менеджер памяти ещё не готов.");
1454        }
1455        if (LinkDoesNotExist(link))
1456        {
1457            throw new ArgumentException("Нельзя обновить несуществующую связь.",
1458                ↳ nameof(link));
1459        }
1460        if (LinkWasDeleted(newSource))
1461        {
1462            throw new ArgumentException("Удалённая связь не может использоваться в качестве
1463                ↳ нового значения.", nameof(newSource));
1464        }
1465        if (LinkWasDeleted(newLinker))
1466        {
1467            throw new ArgumentException("Удалённая связь не может использоваться в качестве
1468                ↳ нового значения.", nameof(newLinker));
1469        }
1470    }

```

```

1466     }
1467     if (LinkWasDeleted(newTarget))
1468     {
1469         throw new ArgumentException("Удалённая связь не может использоваться в качестве
        ↳ нового значения.", nameof(newTarget));
1470     }
1471     LinkIndex previousLinkIndex = link;
1472     _linkToMappingIndex.TryGetValue(link, out long mappingIndex);
1473     var previousDefinition = new LinkDefinition(link);
1474     link = UpdateLink(link, newSource, newLinker, newTarget);
1475     if (mappingIndex >= 0 && previousLinkIndex != link)
1476     {
1477         _linkToMappingIndex.Remove(previousLinkIndex);
1478         SetMappedLink(mappingIndex, link);
1479         _linkToMappingIndex.Add(link, mappingIndex);
1480     }
1481     UpdatedEvent(previousDefinition, new LinkDefinition(link));
1482 }
1483
1484 /// <summary>
1485 /// <para>
1486 /// Deletes the link.
1487 /// </para>
1488 /// <para></para>
1489 /// </summary>
1490 /// <param name="link">
1491 /// <para>The link.</para>
1492 /// <para></para>
1493 /// </param>
1494 public static void Delete(Link link) => Delete(ref link);
1495
1496 /// <summary>
1497 /// <para>
1498 /// Deletes the link.
1499 /// </para>
1500 /// <para></para>
1501 /// </summary>
1502 /// <param name="link">
1503 /// <para>The link.</para>
1504 /// <para></para>
1505 /// </param>
1506 public static void Delete(ref Link link)
1507 {
1508     if (LinkDoesNotExist(link))
1509     {
1510         return;
1511     }
1512     LinkIndex previousLinkIndex = link;
1513     _linkToMappingIndex.TryGetValue(link, out long mappingIndex);
1514     var previousDefinition = new LinkDefinition(link);
1515     DeleteLink(link);
1516     link = null;
1517     if (mappingIndex >= 0)
1518     {
1519         _linkToMappingIndex.Remove(previousLinkIndex);
1520         SetMappedLink(mappingIndex, 0);
1521     }
1522     DeletedEvent(previousDefinition);
1523 }
1524
1525 //public static void Replace(ref Link link, Link replacement)
1526 //{
1527 //    if (!MemoryManagerIsReady)
1528 //        throw new InvalidOperationException("Менеджер памяти ещё не готов.");
1529 //    if (LinkDoesNotExist(link))
1530 //        throw new InvalidOperationException("Если связь не существует, её нельзя
        ↳ заменить.");
1531 //    if (LinkDoesNotExist(replacement))
1532 //        throw new ArgumentException("Пустая или удалённая связь не может быть
        ↳ замещаемым значением.", "replacement");
1533 //    link = ReplaceLink(link, replacement);
1534 //}
1535
1536 /// <summary>
1537 /// <para>
1538 /// Searches the source.
1539 /// </para>
1540 /// <para></para>

```

```

1541     /// </summary>
1542     /// <param name="source">
1543     /// <para>The source.</para>
1544     /// <para></para>
1545     /// </param>
1546     /// <param name="linker">
1547     /// <para>The linker.</para>
1548     /// <para></para>
1549     /// </param>
1550     /// <param name="target">
1551     /// <para>The target.</para>
1552     /// <para></para>
1553     /// </param>
1554     /// <exception cref="InvalidOperationException">
1555     /// <para>Выполнить поиск связи можно только по существующим связям.</para>
1556     /// <para></para>
1557     /// </exception>
1558     /// <exception cref="InvalidOperationException">
1559     /// <para>Менеджер памяти ещё не готов.</para>
1560     /// <para></para>
1561     /// </exception>
1562     /// <returns>
1563     /// <para>The link</para>
1564     /// <para></para>
1565     /// </returns>
1566 public static Link Search(Link source, Link linker, Link target)
1567 {
1568     if (_memoryManagerIsReady == default)
1569     {
1570         throw new InvalidOperationException("Менеджер памяти ещё не готов.");
1571     }
1572     if (LinkDoesNotExist(source) || LinkDoesNotExist(linker) || LinkDoesNotExist(target))
1573     {
1574         throw new InvalidOperationException("Выполнить поиск связи можно только по
1575             ↳ существующим связям.");
1576     }
1577     return SearchLink(source, linker, target);
1578 }
1579
1580     /// <summary>
1581     /// <para>
1582     /// Determines whether exists.
1583     /// </para>
1584     /// <para></para>
1585     /// </summary>
1586     /// <param name="source">
1587     /// <para>The source.</para>
1588     /// <para></para>
1589     /// </param>
1590     /// <param name="linker">
1591     /// <para>The linker.</para>
1592     /// <para></para>
1593     /// </param>
1594     /// <param name="target">
1595     /// <para>The target.</para>
1596     /// <para></para>
1597     /// </param>
1598     /// <returns>
1599     /// <para>The bool</para>
1600     /// <para></para>
1601     /// </returns>
1602 public static bool Exists(Link source, Link linker, Link target) => SearchLink(source,
1603     ↳ linker, target) != 0;
1604
1605 #endregion
1606
1607 #region Referers Walkers
1608
1609     /// <summary>
1610     /// <para>
1611     /// Determines whether this instance walk through referers as source.
1612     /// </para>
1613     /// <para></para>
1614     /// </summary>
1615     /// <param name="walker">
1616     /// <para>The walker.</para>
1617     /// <para></para>
1618     /// </param>

```



```

1617 /// <exception cref="InvalidOperationException">
1618 /// <para>С несуществующей связью нельзя производить операции.</para>
1619 /// <para></para>
1620 /// </exception>
1621 /// <returns>
1622 /// <para>The bool</para>
1623 /// <para></para>
1624 /// </returns>
1625 public bool WalkThroughReferersAsSource(Func<Link, bool> walker)
1626 {
1627     if (LinkDoesNotExist(this))
1628     {
1629         throw new InvalidOperationException("С несуществующей связью нельзя
1630         ↪ производить операции.");
1631     }
1632     var referers = ReferersBySourceCount;
1633     if (referers == 1)
1634     {
1635         return walker(FirstRefererBySource);
1636     }
1637     else if (referers > 1)
1638     {
1639         return WalkThroughReferersBySource(this, x => walker(x) ? 1 : 0) != 0;
1640     }
1641     else
1642     {
1643         return true;
1644     }
1645 }
1646 /// <summary>
1647 /// <para>
1648 /// Walks the through referers as source using the specified walker.
1649 /// </para>
1650 /// <para></para>
1651 /// </summary>
1652 /// <param name="walker">
1653 /// <para>The walker.</para>
1654 /// <para></para>
1655 /// </param>
1656 /// <exception cref="InvalidOperationException">
1657 /// <para>С несуществующей связью нельзя производить операции.</para>
1658 /// <para></para>
1659 /// </exception>
1660 public void WalkThroughReferersAsSource(Action<Link> walker)
1661 {
1662     if (LinkDoesNotExist(this))
1663     {
1664         throw new InvalidOperationException("С несуществующей связью нельзя
1665         ↪ производить операции.");
1666     }
1667     var referers = ReferersBySourceCount;
1668     if (referers == 1)
1669     {
1670         walker(FirstRefererBySource);
1671     }
1672     else if (referers > 1)
1673     {
1674         WalkThroughAllReferersBySource(this, x => walker(x));
1675     }
1676 }
1677 /// <summary>
1678 /// <para>
1679 /// Determines whether this instance walk through referers as linker.
1680 /// </para>
1681 /// <para></para>
1682 /// </summary>
1683 /// <param name="walker">
1684 /// <para>The walker.</para>
1685 /// <para></para>
1686 /// </param>
1687 /// <exception cref="InvalidOperationException">
1688 /// <para>С несуществующей связью нельзя производить операции.</para>
1689 /// <para></para>
1690 /// </exception>
1691 /// <returns>
1692 /// <para>The bool</para>

```

```

1693 /// <para></para>
1694 /// </returns>
1695 public bool WalkThroughReferersAsLinker(Func<Link, bool> walker)
1696 {
1697     if (LinkDoesNotExist(this))
1698     {
1699         throw new InvalidOperationException("С несуществующей связью нельзя
1700             ↳ производить операции.");
1701     }
1702     var referers = ReferersByLinkerCount;
1703     if (referers == 1)
1704     {
1705         return walker(FirstRefererByLinker);
1706     }
1707     else if (referers > 1)
1708     {
1709         return WalkThroughReferersByLinker(this, x => walker(x) ? 1 : 0) != 0;
1710     }
1711     else
1712     {
1713         return true;
1714     }
1715 }
1716 /// <summary>
1717 /// <para>
1718 /// Walks the through referers as linker using the specified walker.
1719 /// </para>
1720 /// <para></para>
1721 /// </summary>
1722 /// <param name="walker">
1723 /// <para>The walker.</para>
1724 /// <para></para>
1725 /// </param>
1726 /// <exception cref="InvalidOperationException">
1727 /// <para>С несуществующей связью нельзя производить операции.</para>
1728 /// <para></para>
1729 /// </exception>
1730 public void WalkThroughReferersAsLinker(Action<Link> walker)
1731 {
1732     if (LinkDoesNotExist(this))
1733     {
1734         throw new InvalidOperationException("С несуществующей связью нельзя
1735             ↳ производить операции.");
1736     }
1737     var referers = ReferersByLinkerCount;
1738     if (referers == 1)
1739     {
1740         walker(FirstRefererByLinker);
1741     }
1742     else if (referers > 1)
1743     {
1744         WalkThroughAllReferersByLinker(this, x => walker(x));
1745     }
1746 }
1747 /// <summary>
1748 /// <para>
1749 /// Determines whether this instance walk through referers as target.
1750 /// </para>
1751 /// <para></para>
1752 /// </summary>
1753 /// <param name="walker">
1754 /// <para>The walker.</para>
1755 /// <para></para>
1756 /// </param>
1757 /// <exception cref="InvalidOperationException">
1758 /// <para>С несуществующей связью нельзя производить операции.</para>
1759 /// <para></para>
1760 /// </exception>
1761 /// <returns>
1762 /// <para>The bool</para>
1763 /// <para></para>
1764 /// </returns>
1765 public bool WalkThroughReferersAsTarget(Func<Link, bool> walker)
1766 {
1767     if (LinkDoesNotExist(this))
1768     {

```

```

1769         throw new InvalidOperationException("С несуществующей связью нельзя
1770             ↪ производить операции.");
1771     }
1772     var referers = ReferersByTargetCount;
1773     if (referers == 1)
1774     {
1775         return walker(FirstRefererByTarget);
1776     }
1777     else if (referers > 1)
1778     {
1779         return WalkThroughReferersByTarget(this, x => walker(x) ? 1 : 0) != 0;
1780     }
1781     else
1782     {
1783         return true;
1784     }
1785 }
1786
1787 /// <summary>
1788 /// <para>
1789 /// Walks the through referers as target using the specified walker.
1790 /// </para>
1791 /// <para></para>
1792 /// </summary>
1793 /// <param name="walker">
1794 /// <para>The walker.</para>
1795 /// <para></para>
1796 /// </param>
1797 /// <exception cref="InvalidOperationException">
1798 /// <para>С несуществующей связью нельзя производить операции.</para>
1799 /// <para></para>
1800 /// </exception>
1801 public void WalkThroughReferersAsTarget(Action<Link> walker)
1802 {
1803     if (LinkDoesNotExist(this))
1804     {
1805         throw new InvalidOperationException("С несуществующей связью нельзя
1806             ↪ производить операции.");
1807     }
1808     var referers = ReferersByTargetCount;
1809     if (referers == 1)
1810     {
1811         walker(FirstRefererByTarget);
1812     }
1813     else if (referers > 1)
1814     {
1815         WalkThroughAllReferersByTarget(this, x => walker(x));
1816     }
1817 }
1818
1819 /// <summary>
1820 /// <para>
1821 /// Walks the through referers using the specified walker.
1822 /// </para>
1823 /// <para></para>
1824 /// </summary>
1825 /// <param name="walker">
1826 /// <para>The walker.</para>
1827 /// <para></para>
1828 /// </param>
1829 /// <exception cref="InvalidOperationException">
1830 /// <para>С несуществующей связью нельзя производить операции.</para>
1831 /// <para></para>
1832 /// </exception>
1833 public void WalkThroughReferers(Action<Link> walker)
1834 {
1835     if (LinkDoesNotExist(this))
1836     {
1837         throw new InvalidOperationException("С несуществующей связью нельзя
1838             ↪ производить операции.");
1839     }
1840     void wrapper(ulong x) => walker(x);
1841     WalkThroughAllReferersBySource(this, wrapper);
1842     WalkThroughAllReferersByLinker(this, wrapper);
1843     WalkThroughAllReferersByTarget(this, wrapper);
1844 }
1845
1846 /// <summary>

```

```

1844     /// <para>
1845     /// Walks the through referers using the specified walker.
1846     /// </para>
1847     /// <para></para>
1848     /// </summary>
1849     /// <param name="walker">
1850     /// <para>The walker.</para>
1851     /// <para></para>
1852     /// </param>
1853     /// <exception cref="InvalidOperationException">
1854     /// <para>C несуществующей связью нельзя производить операции.</para>
1855     /// <para></para>
1856     /// </exception>
1857     public void WalkThroughReferers(Func<Link, bool> walker)
1858     {
1859         if (LinkDoesNotExist(this))
1860         {
1861             throw new InvalidOperationException("C несуществующей связью нельзя
1862                 ↳ производить операции.");
1863         }
1864         long wrapper(ulong x) => walker(x) ? 1 : 0;
1865         WalkThroughReferersBySource(this, wrapper);
1866         WalkThroughReferersByLinker(this, wrapper);
1867         WalkThroughReferersByTarget(this, wrapper);
1868     }
1869     /// <summary>
1870     /// <para>
1871     /// Determines whether walk through all links.
1872     /// </para>
1873     /// <para></para>
1874     /// </summary>
1875     /// <param name="walker">
1876     /// <para>The walker.</para>
1877     /// <para></para>
1878     /// </param>
1879     /// <returns>
1880     /// <para>The bool</para>
1881     /// <para></para>
1882     /// </returns>
1883     public static bool WalkThroughAllLinks(Func<Link, bool> walker) => WalkThroughLinks(x =>
1884         ↳ walker(x) ? 1 : 0) != 0;
1885     /// <summary>
1886     /// <para>
1887     /// Walks the through all links using the specified walker.
1888     /// </para>
1889     /// <para></para>
1890     /// </summary>
1891     /// <param name="walker">
1892     /// <para>The walker.</para>
1893     /// <para></para>
1894     /// </param>
1895     public static void WalkThroughAllLinks(Action<Link> walker) => WalkThroughAllLinks(new
1896         ↳ Visitor(x => walker(x)));
1897     #endregion
1898 }
1899 }

```

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">

```

```

95     /// <para>The number.</para>
96     /// <para></para>
97     /// </param>
98     /// <returns>
99     /// <para>The link</para>
100    /// <para></para>
101    /// </returns>
102    public static Link FromNumber(long number) => NumberHelpers.FromNumber(number);
103
104    /// <summary>
105    /// <para>
106    /// 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     /// <para>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     /// <para>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     /// <para>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));

```

```

247
248    /// <summary>
249    /// <para>
250    /// Creates the numbers using the specified numbers.
251    /// </para>
252    /// <para></para>
253    /// </summary>
254    /// <param name="numbers">
255    /// <para>The numbers.</para>
256    /// <para></para>
257    /// </param>
258    /// <param name="takeFrom">
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) =>
271        ↪ FromObjectsToSequence(numbers, takeFrom, takeUntil, x => FromNumber(x));
272
273    /// <summary>
274    /// <para>
275    /// Creates the numbers using the specified numbers.
276    /// </para>
277    /// <para></para>
278    /// </summary>
279    /// <param name="numbers">
280    /// <para>The numbers.</para>
281    /// <para></para>
282    /// </param>
283    /// <returns>
284    /// <para>The link</para>
285    /// <para></para>
286    /// </returns>
287    public static Link FromNumbers(uint[] numbers) => FromObjectsToSequence(numbers, x =>
288        ↪ FromNumber(x));
289
290    /// <summary>
291    /// <para>
292    /// Creates the numbers using the specified numbers.
293    /// </para>
294    /// <para></para>
295    /// </summary>
296    /// <param name="numbers">
297    /// <para>The numbers.</para>
298    /// <para></para>
299    /// </param>
300    /// <param name="takeFrom">
301    /// <para>The take from.</para>
302    /// <para></para>
303    /// </param>
304    /// <param name="takeUntil">
305    /// <para>The take until.</para>
306    /// <para></para>
307    /// </param>
308    /// <returns>
309    /// <para>The link</para>
310    /// <para></para>
311    /// </returns>
312    public static Link FromNumbers(uint[] numbers, int takeFrom, int takeUntil) =>
313        ↪ FromObjectsToSequence(numbers, takeFrom, takeUntil, x => FromNumber(x));
314
315    /// <summary>
316    /// <para>
317    /// Creates the numbers using the specified numbers.
318    /// </para>
319    /// <para></para>
320    /// </summary>
321    /// <param name="numbers">
322    /// <para>The numbers.</para>
323    /// <para></para>
324    /// </param>

```



```

322    /// <returns>
323    /// <para>The link</para>
324    /// <para></para>
325    /// </returns>
326    public static Link FromNumbers(byte[] numbers) => FromObjectsToSequence(numbers, x =>
    ↪ FromNumber(x));

327
328    /// <summary>
329    /// <para>
330    /// Creates the numbers using the specified numbers.
331    /// </para>
332    /// <para></para>
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     <summary>
417     <para>
418     <para>Creates the objects to sequence using the specified objects.
419     </para>
420     <para></para>
421     </summary>
422     <typeparam name="T">
423     <para>The .</para>
424     <para></para>
425     </typeparam>
426     <param name="objects">
427     <para>The objects.</para>
428     <para></para>
429     </param>
430     <param name="takeFrom">
431     <para>The take from.</para>
432     <para></para>
433     </param>
434     <param name="takeUntil">
435     <para>The take until.</para>
436     <para></para>
437     </param>
438     <param name="converter">
439     <para>The converter.</para>
440     <para></para>
441     </param>
442     <exception cref="ArgumentOutOfRangeException">
443     <para>Нельзя преобразовать пустой список к связям.</para>
444     <para></para>
445     </exception>
446     <returns>
447     <para>The link</para>
448     <para></para>
449     </returns>
450     public static Link FromObjectsToSequence<T>(T[] objects, int takeFrom, int takeUntil,
451     ↪ Func<T, Link> converter)
452     {
453         var length = takeUntil - takeFrom;
454         if (length <= 0)
455         {
456             throw new ArgumentOutOfRangeException(nameof(takeUntil), "Нельзя преобразовать
457             ↪ пустой список к связям.");
458         }
459         var copy = new Link[length];
460         for (int i = takeFrom, j = 0; i < takeUntil; i++, j++)
461         {
462             copy[j] = converter(objects[i]);
463         }
464         return FromList(copy);
465     }
466     <summary>
467     <para>
468     <para>Creates the chars using the specified str.
469     </para>
470     <para></para>
471     </summary>

```

```

471     /// <param name="str">
472     /// <para>The str.</para>
473     /// <para></para>
474     /// </param>
475     /// <returns>
476     /// <para>The link</para>
477     /// <para></para>
478     /// </returns>
479     public static Link FromChars(string str)
480     {
481         var copy = new Link[str.Length];
482         for (var i = 0; i < copy.Length; i++)
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">

```

```

548     /// <para>The char links.</para>
549     /// <para></para>
550     /// </param>
551     /// <returns>
552     /// <para>The string</para>
553     /// <para></para>
554     /// </returns>
555     public static string ToString(List<Link> charLinks)
556     {
557         var chars = new char[charLinks.Count];
558         for (var i = 0; i < charLinks.Count; i++)
559         {
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         private static readonly HashSet<Link> _linksWithNamesGatheringProcess = new
44             ↪ HashSet<Link>();
45
46         /// <summary>
47         /// <para>
48         /// Determines whether try get name.
49         /// </para>
50         /// <para></para>
51         /// </summary>
52         /// <param name="link">
53         /// <para>The link.</para>
54         /// <para></para>
55         /// </param>
56         /// <param name="str">
57         /// <para>The str.</para>
58         /// <para></para>
59         /// </param>
60         /// <returns>
61         /// <para>The bool.</para>
62         /// <para></para>
63         /// </returns>
64     }
65 }

```

```

57     /// </param>
58     /// <returns>
59     /// <para>The bool</para>
60     /// <para></para>
61     /// </returns>
62     public static bool TryGetName(this Link link, out string str)
63     {
64         // Защита от зацикливания
65         if (!_linksWithNamesGatheringProcess.Add(link))
66         {
67             str = "...";
68             return true;
69         }
70         try
71         {
72             if (link != null)
73             {
74                 if (link.Linker == Net.And)
75                 {
76                     str = SequenceHelpers.FormatSequence(link);
77                     return true;
78                 }
79                 else if (link.IsGroup())
80                 {
81                     str = LinkConverter.ToString(LinkConverter.ToList(link.Target));
82                     return true;
83                 }
84                 else if (link.IsChar())
85                 {
86                     str = LinkConverter.ToChar(link).ToString();
87                     return true;
88                 }
89                 else if (link.TryGetSpecificName(out str))
90                 {
91                     return true;
92                 }
93             }
94             if (link.Source == link || link.Linker == link || link.Target == link)
95             {
96                 return false;
97             }
98             if (link.Source.TryGetName(out string sourceName) &&
99                 ↪ link.Linker.TryGetName(out string linkerName) &&
100                 ↪ link.Target.TryGetName(out string targetName))
101             {
102                 var sb = new StringBuilder();
103                 sb.Append(sourceName).Append(' ').Append(linkerName).Append(' ')
104                 ↪ .Append(targetName);
105                 str = sb.ToString();
106                 return true;
107             }
108             str = null;
109             return false;
110         }
111         finally
112         {
113             _linksWithNamesGatheringProcess.Remove(link);
114         }
115     }
116     /// <summary>
117     /// <para>
118     /// Determines whether try get specific name.
119     /// </para>
120     /// <para></para>
121     /// </summary>
122     /// <param name="link">
123     /// <para>The link.</para>
124     /// <para></para>
125     /// </param>
126     /// <param name="name">
127     /// <para>The name.</para>
128     /// <para></para>
129     /// </param>
130     /// <returns>
131     /// <para>The bool</para>
132     /// <para></para>

```

```

133 /// </returns>
134 public static bool TryGetSpecificName(this Link link, out string name)
135 {
136     string nameLocal = null;
137     if (Net.Name.ReferersBySourceCount < link.ReferersBySourceCount)
138     {
139         Net.Name.WalkThroughReferersAsSource(referer =>
140         {
141             if (referer.Linker == Net.ThatIsRepresentedBy)
142             {
143                 if (Link.Exists(link, Net.Has, referer))
144                 {
145                     nameLocal = LinkConverter.ToString(referer.Target);
146                     return false; // Останавливаем проход
147                 }
148             }
149             return true;
150         });
151     }
152     else
153     {
154         link.WalkThroughReferersAsSource(referer =>
155         {
156             if (referer.Linker == Net.Has)
157             {
158                 var nameLink = referer.Target;
159                 if (nameLink.Source == Net.Name && nameLink.Linker ==
160                     ↪ Net.ThatIsRepresentedBy)
161                 {
162                     nameLocal = LinkConverter.ToString(nameLink.Target);
163                     return false; // Останавливаем проход
164                 }
165             }
166             return true;
167         });
168     }
169     name = nameLocal;
170     return nameLocal != null;
171 }

```

```

172
173 // Проверка на принадлежность классу
174 /// <summary>
175 /// <para>
176 /// Determines whether is.
177 /// </para>
178 /// <para></para>
179 /// </summary>
180 /// <param name="link">
181 /// <para>The link.</para>
182 /// <para></para>
183 /// </param>
184 /// <param name="@class">
185 /// <para>The class.</para>
186 /// <para></para>
187 /// </param>
188 /// <returns>
189 /// <para>The bool</para>
190 /// <para></para>
191 /// </returns>

```

```

192 public static bool Is(this Link link, Link @class)
193 {
194     if (link.Linker == Net.IsA)
195     {
196         if (link.Target == @class)
197         {
198             return true;
199         }
200         else
201         {
202             return link.Target.Is(@class);
203         }
204     }
205     return false;
206 }

```

```

207
208 // Несколько не правильное определение, так выйдет, что любая сумма входящая в диапазон
209 ↪ значений char будет символом.
210 // Нужно изменить определение чара, идеально: char consists of sum of [8, 64].

```

```

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

```

```

285 /// <para>The bool</para>
286 /// <para></para>
287 /// </returns>
288 public static bool IsName(this Link link) => link != null && link.Source == Net.Name &&
    ↳ link.Linker == Net.Of;
289
290 /// <summary>
291 /// <para>
292 /// Gets the array of rererers by source using the specified link.
293 /// </para>
294 /// <para></para>
295 /// </summary>
296 /// <param name="link">
297 /// <para>The link.</para>
298 /// <para></para>
299 /// </param>
300 /// <returns>
301 /// <para>The link array</para>
302 /// <para></para>
303 /// </returns>
304 public static Link[] GetArrayOfRererersBySource(this Link link)
305 {
306     if (link == null)
307     {
308         return new Link[0];
309     }
310     else
311     {
312         var array = new Link[link.ReferersBySourceCount];
313         var index = 0;
314         link.WalkThroughReferersAsSource(referer => array[index++] = referer);
315         return array;
316     }
317 }
318
319 /// <summary>
320 /// <para>
321 /// Gets the array of rererers by linker using the specified link.
322 /// </para>
323 /// <para></para>
324 /// </summary>
325 /// <param name="link">
326 /// <para>The link.</para>
327 /// <para></para>
328 /// </param>
329 /// <returns>
330 /// <para>The link array</para>
331 /// <para></para>
332 /// </returns>
333 public static Link[] GetArrayOfRererersByLinker(this Link link)
334 {
335     if (link == null)
336     {
337         return new Link[0];
338     }
339     else
340     {
341         var array = new Link[link.ReferersByLinkerCount];
342         var index = 0;
343         link.WalkThroughReferersAsLinker(referer => array[index++] = referer);
344         return array;
345     }
346 }
347
348 /// <summary>
349 /// <para>
350 /// Gets the array of rererers by target using the specified link.
351 /// </para>
352 /// <para></para>
353 /// </summary>
354 /// <param name="link">
355 /// <para>The link.</para>
356 /// <para></para>
357 /// </param>
358 /// <returns>
359 /// <para>The link array</para>
360 /// <para></para>
361 /// </returns>

```



```

362 public static Link[] GetArrayOfRerersByTarget(this Link link)
363 {
364     if (link == null)
365     {
366         return new Link[0];
367     }
368     else
369     {
370         var array = new Link[link.ReferersByTargetCount];
371         var index = 0;
372         link.WalkThroughReferersAsTarget(referer => array[index++] = referer);
373         return array;
374     }
375 }
376
377 /// <summary>
378 /// <para>
379 /// Walks the through sequence using the specified link.
380 /// </para>
381 /// <para></para>
382 /// </summary>
383 /// <param name="link">
384 /// <para>The link.</para>
385 /// <para></para>
386 /// </param>
387 /// <param name="action">
388 /// <para>The action.</para>
389 /// <para></para>
390 /// </param>
391 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);
392 }
393 }

```

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>

```

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

```

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     private static void Create()
646     {
647         #region Core
648
649         IsA = Link.GetMappedOrDefault(NetMapping.IsA);
650         IsNotA = Link.GetMappedOrDefault(NetMapping.IsNotA);
651         Link = Link.GetMappedOrDefault(NetMapping.Link);
652         Thing = Link.GetMappedOrDefault(NetMapping.Thing);
653
654         if (IsA == null || IsNotA == null || Link == null || Thing == null)
655         {
656             // Наивная инициализация (Не является корректным объяснением).
657             IsA = Link.CreateMapped(Link.Itself, Link.Itself, Link.Itself, NetMapping.IsA);
658             ↪ // Строит переделат в "[x] is a member|instance|element of the class [y]"
659             IsNotA = Link.CreateMapped(Link.Itself, Link.Itself, IsA, NetMapping.IsNotA);
660             Link = Link.CreateMapped(Link.Itself, IsA, Link.Itself, NetMapping.Link);
661             Thing = Link.CreateMapped(Link.Itself, IsNotA, Link, NetMapping.Thing);
662
663             IsA = Link.Update(IsA, IsA, IsA, Link); // Исключение, позволяющие завершить
664             ↪ систему
665         }
666     }
667     #endregion

```

```

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

```

```

745         Group.SetName("group");
746
747         ParsedFrom.SetName("parsed from");
748         ThatIs.SetName("that is");
749         ThatIsBefore.SetName("that is before");
750         ThatIsAfter.SetName("that is after");
751         ThatIsBetween.SetName("that is between");
752         ThatIsRepresentedBy.SetName("that is represented by");
753         ThatHas.SetName("that has");
754
755         Text.SetName("text");
756         Path.SetName("path");
757         Content.SetName("content");
758         Empty.SetName("empty");
759         EmptyContent.SetName("empty content");
760         Alphabet.SetName("alphabet");
761         Letter.SetName("letter");
762         Case.SetName("case");
763         Upper.SetName("upper");
764         Lower.SetName("lower");
765         Code.SetName("code");
766     }
767 }
768 }

```

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         private static void Create()
41         {
42             NumbersToLinks = new Link[64];
43             LinksToNumbers = new Dictionary<Link, long>();
44             NumbersToLinks[0] = Net.One;
45             LinksToNumbers[Net.One] = 1;
46         }
47
48         /// <summary>
49         /// <para>
50         /// Recreates.
51         /// </para>
52         /// <para></para>
53         /// </summary>

```

```

54 public static void Recreate() => Create();
55 private static Link FromPowerOf2(long powerOf2)
56 {
57     var result = NumbersToLinks[powerOf2];
58     if (result == null)
59     {
60         var previousPowerOf2Link = NumbersToLinks[powerOf2 - 1];
61         if (previousPowerOf2Link == null)
62         {
63             previousPowerOf2Link = NumbersToLinks[0];
64             for (var i = 1; i < powerOf2; i++)
65             {
66                 if (NumbersToLinks[i] == null)
67                 {
68                     var numberLink = Link.Create(Net.Sum, Net.Of, previousPowerOf2Link &
69                     ↪ previousPowerOf2Link);
70                     var num = (long)System.Math.Pow(2, i);
71                     NumbersToLinks[i] = numberLink;
72                     LinksToNumbers[numberLink] = num;
73                     numberLink.SetName(num.ToString(CultureInfo.InvariantCulture));
74                 }
75                 previousPowerOf2Link = NumbersToLinks[i];
76             }
77             result = Link.Create(Net.Sum, Net.Of, previousPowerOf2Link &
78             ↪ previousPowerOf2Link);
79             var number = (long)System.Math.Pow(2, powerOf2);
80             NumbersToLinks[powerOf2] = result;
81             LinksToNumbers[result] = number;
82             result.SetName(number.ToString(CultureInfo.InvariantCulture));
83         }
84         return result;
85     }
86     /// <summary>
87     /// <para>
88     /// Creates the number using the specified number.
89     /// </para>
90     /// <para></para>
91     /// </summary>
92     /// <param name="number">
93     /// <para>The number.</para>
94     /// <para></para>
95     /// </param>
96     /// <exception cref="NotSupportedException">
97     /// <para>Negative numbers are not supported yet.</para>
98     /// <para></para>
99     /// </exception>
100    /// <returns>
101    /// <para>The sum.</para>
102    /// <para></para>
103    /// </returns>
104    public static Link FromNumber(long number)
105    {
106        if (number == 0)
107        {
108            return Net.Zero;
109        }
110        if (number == 1)
111        {
112            return Net.One;
113        }
114        var links = new Link[Bit.Count(number)];
115        if (number >= 0)
116        {
117            for (long key = 1, powerOf2 = 0, i = 0; key <= number; key *= 2, powerOf2++)
118            {
119                if ((number & key) == key)
120                {
121                    links[i] = FromPowerOf2(powerOf2);
122                    i++;
123                }
124            }
125        }
126        else
127        {
128            throw new NotSupportedException("Negative numbers are not supported yet.");
129        }

```

```

130         var sum = Link.Create(Net.Sum, Net.Of, LinkConverter.FromList(links));
131         return sum;
132     }
133
134     /// <summary>
135     /// <para>
136     /// Returns the number using the specified link.
137     /// </para>
138     /// <para></para>
139     /// </summary>
140     /// <param name="link">
141     /// <para>The link.</para>
142     /// <para></para>
143     /// </param>
144     /// <exception cref="ArgumentOutOfRangeException">
145     /// <para>Specified link is not a number.</para>
146     /// <para></para>
147     /// </exception>
148     /// <returns>
149     /// <para>The long</para>
150     /// <para></para>
151     /// </returns>
152     public static long ToNumber(Link link)
153     {
154         if (link == Net.Zero)
155         {
156             return 0;
157         }
158         if (link == Net.One)
159         {
160             return 1;
161         }
162         if (link.IsSum())
163         {
164             var numberParts = LinkConverter.ToList(link.Target);
165             long number = 0;
166             for (var i = 0; i < numberParts.Count; i++)
167             {
168                 GoDownAndTakeIt(numberParts[i], out long numberPart);
169                 number += numberPart;
170             }
171             return number;
172         }
173         throw new ArgumentOutOfRangeException(nameof(link), "Specified link is not a
174             ↪ number.");
175     }
176     private static void GoDownAndTakeIt(Link link, out long number)
177     {
178         if (!LinksToNumbers.TryGetValue(link, out number))
179         {
180             var previousNumberLink = link.Target.Source;
181             GoDownAndTakeIt(previousNumberLink, out number);
182             var previousNumberIndex = (int)System.Math.Log(number, 2);
183             var newNumberIndex = previousNumberIndex + 1;
184             var newNumberLink = Link.Create(Net.Sum, Net.Of, previousNumberLink &
185                 ↪ previousNumberLink);
186             number += number;
187             NumbersToLinks[newNumberIndex] = newNumberLink;
188             LinksToNumbers[newNumberLink] = number;
189         }
190     }
191 }

```

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      }
40      subject = Link.Create(subject, Net.And, @object);
41  }
42  //public static Link RightJoinUnsafe(Link subject, Link @object)
43  //{
44  //    if (subject.Linker == Net.And && subject.ReferersBySourceCount == 0 &&
45  //        ↪ subject.ReferersByTargetCount == 0)
46  //    {
47  //        Link subJoint = Link.Search(subject.Target, Net.And, @object);
48  //        if (subJoint != null && subJoint != subject)
49  //        {
50  //            Link.Update(ref subject, subject.Source, Net.And, subJoint);
51  //            return subject;
52  //        }
53  //    }
54  //    return Link.Create(subject, Net.And, @object);
55  //}
56  ///public static void LeftJoin(ref Link subject, Link @object)
57  {
58  ///    if (subject.Linker == Net.And && subject.ReferersBySourceCount == 0 &&
59  ///        ↪ subject.ReferersByTargetCount == 0)
60  ///    {
61  ///        Link subJoint = Link.Search(@object, Net.And, subject.Source);
62  ///        if (subJoint != null && subJoint != subject)
63  ///        {
64  ///            Link.Update(ref subject, subJoint, Net.And, subject.Target);
65  ///            return;
66  ///        }
67  ///    }
68  ///    subject = Link.Create(@object, Net.And, subject);
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 &&
87          ↪ subject.ReferersByTargetCount == 0)
88      {
89          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
105 // алгоритма.
106 //public static Link CombinedJoin(ref Link first, ref Link second)
107 //{
108     Link atomicConnection = Link.Search(first, Net.And, second);
109     if (atomicConnection != null)
110     {
111         return atomicConnection;
112     }
113     else
114     {
115         if (second.Linker == Net.And)
116         {
117             Link subJoint = Link.Search(first, Net.And, second.Source);
118             if (subJoint != null && subJoint != second) // && subJoint.TotalReferers >
119                 second.TotalReferers)
120             {
121                 //if (first.Linker == Net.And)
122                 //{
123                     // TODO: ...
124                 //}
125                 if (second.TotalReferers > 0)
126                 {
127                     // В данный момент это никак не влияет, из-за того что добавлено
128                     // условие по требованию
129                     // использования атомарного соединения если оно есть
130
131                     // В целом же приоритет между обходным соединением и атомарным
132                     // нужно определять по весу.
133                     // И если в сети обнаружено сразу два варианта прохода - простой и
134                     // обходной - нужно перебрасывать
135                     // пути с меньшим весом на использование путей с большим весом.
136                     // (Это и технически эффективнее и более оправдано
137                     // с точки зрения смысла).
138
139                     // Положительный эффект текущей реализации, что она быстро
140                     // "успокаивается" набирает критическую массу
141                     // и перестаёт вести себя непредсказуемо
142
143                     // Неприятность учёта веса в том, что нужно обрабатывать большое
144                     // количество комбинаций.
145                     // Но вероятно это оправдано.
146
147                     //var prev = Link.Search(first, Net.And, second);
148                     //if (prev != null && subJoint != prev) // && prev.TotalReferers <
149                     //    subJoint.TotalReferers)
150                     //{
151                         Link.Update(ref prev, subJoint, Net.And, second.Target);
152                         if (second.TotalReferers == 0)
153                         {
154                             Link.Delete(ref second);
155                         }
156                         return prev;
157                     //}
158                     //return Link.Create(subJoint, Net.And, second.Target);
159                 }
160             }
161             else
162             {
163                 Link.Update(ref second, subJoint, Net.And, second.Target);
164                 return second;
165             }
166         }
167     }
168 }

```

```

158 //         if (first.Linker == Net.And)
159 //         {
160 //             Link subJoint = Link.Search(first.Target, Net.And, second);
161 //             if (subJoint != null && subJoint != first)// && subJoint.TotalReferers >
→ first.TotalReferers)
162 //             {
163 //                 if (first.TotalReferers > 0)
164 //                 {
165 //                     //var prev = Link.Search(first, Net.And, second);
166 //                     //if (prev != null && subJoint != prev) // && prev.TotalReferers <
→ subJoint.TotalReferers)
167 //                     //{
168 //                         // Link.Update(ref prev, first.Source, Net.And, subJoint);
169 //                         // if (first.TotalReferers == 0)
170 //                         // {
171 //                             // Link.Delete(ref first);
172 //                             // }
173 //                             // return prev;
174 //                         //}
175 //                         //return Link.Create(first.Source, Net.And, subJoint);
176 //                     }
177 //                 else
178 //                 {
179 //                     Link.Update(ref first, first.Source, Net.And, subJoint);
180 //                     return first;
181 //                 }
182 //             }
183 //         }
184 //         return Link.Create(first, Net.And, second);
185 //     }
186 // }
187
188 /// <summary>
189 /// <para>
190 /// The compressions count.
191 /// </para>
192 /// <para></para>
193 /// </summary>
194 public static int CompressionsCount;
195
196 /// <summary>
197 /// <para>
198 /// Combineds the join using the specified first.
199 /// </para>
200 /// <para></para>
201 /// </summary>
202 /// <param name="first">
203 /// <para>The first.</para>
204 /// <para></para>
205 /// </param>
206 /// <param name="second">
207 /// <para>The second.</para>
208 /// <para></para>
209 /// </param>
210 /// <returns>
211 /// <para>The direct connection.</para>
212 /// <para></para>
213 /// </returns>
214 public static Link CombinedJoin(ref Link first, ref Link second)
215 {
216 // Перестроение работает хорошо только когда одна из связей является парой и
→ аккумулятор одновременно
217 // Когда обе связи - пары - нужно использовать другой алгоритм, иначе сжатие будет
→ отсутствовать.
218 //if ((first.Linker == Net.And && second.Linker != Net.And)
219 // || (second.Linker == Net.And && first.Linker != Net.And))
220 //{
221 //     Link connection = TryReconstructConnection(first, second);
222 //     if (connection != null)
223 //     {
224 //         CompressionsCount++;
225 //         return connection;
226 //     }
227 //}
228 //return first & second;
229 //long totalDoublets = Net.And.ReferersByLinkerCount;
230 if (first == null || second == null)
231 {

```



```

232 }
233 var directConnection = Link.Search(first, Net.And, second);
234 if (directConnection == null)
235 {
236     directConnection = TryReconstructConnection(first, second);
237 }
238 Link rightCrossConnection = null;
239 if (second.Linker == Net.And)
240 {
241     var assumedRightCrossConnection = Link.Search(first, Net.And, second.Source);
242     if (assumedRightCrossConnection != null && second != assumedRightCrossConnection)
243     {
244         rightCrossConnection = assumedRightCrossConnection;
245     }
246     else
247     {
248         rightCrossConnection = TryReconstructConnection(first, second.Source);
249     }
250 }
251 Link leftCrossConnection = null;
252 if (first.Linker == Net.And)
253 {
254     var assumedLeftCrossConnection = Link.Search(first.Target, Net.And, second);
255     if (assumedLeftCrossConnection != null && first != assumedLeftCrossConnection)
256     {
257         leftCrossConnection = assumedLeftCrossConnection;
258     }
259     else
260     {
261         leftCrossConnection = TryReconstructConnection(first.Target, second);
262     }
263 }
264 // Наверное имеет смысл только в "безвыходной" ситуации
265 //if (directConnection == null && rightCrossConnection == null &&
    ↳ leftCrossConnection == null)
266 //{{
267 //    directConnection = TryReconstructConnection(first, second);
268 //    // Может давать более агрессивное сжатие, но теряется стабильность
269 //    //if (directConnection == null)
270 //    //{
271 //        //if (second.Linker == Net.And)
272 //        //{
273 //            // Link assumedRightCrossConnection = TryReconstructConnection(first,
    ↳ second.Source);
274 //            // if (assumedRightCrossConnection != null && second !=
    ↳ assumedRightCrossConnection)
275 //            {
276 //                rightCrossConnection = assumedRightCrossConnection;
277 //            }
278 //        //}
279 //        //if (rightCrossConnection == null)
280 //        //{
281 //            //if (first.Linker == Net.And)
282 //            //{
283 //                // Link assumedLeftCrossConnection =
    ↳ TryReconstructConnection(first.Target, second);
284 //                // if (assumedLeftCrossConnection != null && first !=
    ↳ assumedLeftCrossConnection)
285 //                {
286 //                    leftCrossConnection = assumedLeftCrossConnection;
287 //                }
288 //            //}
289 //        //}
290 //    //}
291 //}
292 //Link middleCrossConnection = null;
293 //if (second.Linker == Net.And && first.Linker == Net.And)
294 //{{
295 //    Link assumedMiddleCrossConnection = Link.Search(first.Target, Net.And,
    ↳ second.Source);
296 //    if (assumedMiddleCrossConnection != null && first !=
    ↳ assumedMiddleCrossConnection && second != assumedMiddleCrossConnection)
297 //    {
298 //        middleCrossConnection = assumedMiddleCrossConnection;
299 //    }
300 //}
301 //Link rightMiddleCrossConnectinon = null;

```

```

302 //if (middleCrossConnection != null)
303 //{
304 //}
305 if (directConnection != null
306 && (rightCrossConnection == null || directConnection.TotalReferers >=
    ↳ rightCrossConnection.TotalReferers)
307 && (leftCrossConnection == null || directConnection.TotalReferers >=
    ↳ leftCrossConnection.TotalReferers))
308 {
309     if (rightCrossConnection != null)
310     {
311         var prev = Link.Search(rightCrossConnection, Net.And, second.Target);
312         if (prev != null && directConnection != prev)
313         {
314             Link.Update(ref prev, first, Net.And, second);
315         }
316         if (rightCrossConnection.TotalReferers == 0)
317         {
318             Link.Delete(ref rightCrossConnection);
319         }
320     }
321     if (leftCrossConnection != null)
322     {
323         var prev = Link.Search(first.Source, Net.And, leftCrossConnection);
324         if (prev != null && directConnection != prev)
325         {
326             Link.Update(ref prev, first, Net.And, second);
327         }
328         if (leftCrossConnection.TotalReferers == 0)
329         {
330             Link.Delete(ref leftCrossConnection);
331         }
332     }
333     TryReconstructConnection(first, second);
334     return directConnection;
335 }
336 else if (rightCrossConnection != null
337 && (directConnection == null || rightCrossConnection.TotalReferers >=
    ↳ directConnection.TotalReferers)
338 && (leftCrossConnection == null || rightCrossConnection.TotalReferers >=
    ↳ leftCrossConnection.TotalReferers))
339 {
340     if (directConnection != null)
341     {
342         var prev = Link.Search(first, Net.And, second);
343         if (prev != null && rightCrossConnection != prev)
344         {
345             Link.Update(ref prev, rightCrossConnection, Net.And, second.Target);
346         }
347     }
348     if (leftCrossConnection != null)
349     {
350         var prev = Link.Search(first.Source, Net.And, leftCrossConnection);
351         if (prev != null && rightCrossConnection != prev)
352         {
353             Link.Update(ref prev, rightCrossConnection, Net.And, second.Target);
354         }
355     }
356     //TryReconstructConnection(first, second.Source);
357     //TryReconstructConnection(rightCrossConnection, second.Target); // ухуждает
    ↳ стабильность
358     var resultConnection = rightCrossConnection & second.Target;
359     //if (second.TotalReferers == 0)
360     //    Link.Delete(ref second);
361     return resultConnection;
362 }
363 else if (leftCrossConnection != null
364 && (directConnection == null || leftCrossConnection.TotalReferers >=
    ↳ directConnection.TotalReferers)
365 && (rightCrossConnection == null || leftCrossConnection.TotalReferers >=
    ↳ rightCrossConnection.TotalReferers))
366 {
367     if (directConnection != null)
368     {
369         var prev = Link.Search(first, Net.And, second);
370         if (prev != null && leftCrossConnection != prev)
371         {

```

```

372         Link.Update(ref prev, first.Source, Net.And, leftCrossConnection);
373     }
374 }
375 if (rightCrossConnection != null)
376 {
377     var prev = Link.Search(rightCrossConnection, Net.And, second.Target);
378     if (prev != null && rightCrossConnection != prev)
379     {
380         Link.Update(ref prev, first.Source, Net.And, leftCrossConnection);
381     }
382 }
383 //TryReconstructConnection(first.Target, second);
384 //TryReconstructConnection(first.Source, leftCrossConnection); // ухудшает
    ↪ стабильность
385 var resultConnection = first.Source & leftCrossConnection;
386 //if (first.TotalReferers == 0)
387 //    Link.Delete(ref first);
388 return resultConnection;
389 }
390 else
391 {
392     if (directConnection != null)
393     {
394         return directConnection;
395     }
396     if (rightCrossConnection != null)
397     {
398         return rightCrossConnection & second.Target;
399     }
400     if (leftCrossConnection != null)
401     {
402         return first.Source & leftCrossConnection;
403     }
404 }
405 // Можно фиксировать по окончании каждой из веток, какой эффект от неё происходит
    ↪ (на сколько уменьшается/увеличивается количество связей)
406 directConnection = first & second;
407 //long difference = Net.And.ReferersByLinkerCount - totalDoublets;
408 //if (difference != 1)
409 //{
410 //    Console.WriteLine(Net.And.ReferersByLinkerCount - totalDoublets);
411 //}
412 return directConnection;
413 }
414 private static Link TryReconstructConnection(Link first, Link second)
415 {
416     Link directConnection = null;
417     if (second.ReferersBySourceCount < first.ReferersBySourceCount)
418     {
419         // o_|      x_o ...
420         // x_|      |___|
421         //
422         // <-
423         second.WalkThroughReferersAsSource(couple =>
424         {
425             if (couple.Linker == Net.And && couple.ReferersByTargetCount == 1 &&
    ↪ couple.ReferersBySourceCount == 0)
426             {
427                 var neighbour = couple.FirstRefererByTarget;
428                 if (neighbour.Linker == Net.And && neighbour.Source == first)
429                 {
430                     if (directConnection == null)
431                     {
432                         directConnection = first & second;
433                     }
434                     Link.Update(ref neighbour, directConnection, Net.And, couple.Target);
435                     //Link.Delete(ref couple); // Можно заменить удалением couple
436                 }
437             }
438             if (couple.Linker == Net.And)
439             {
440                 var neighbour = couple.FirstRefererByTarget;
441                 if (neighbour.Linker == Net.And && neighbour.Source == first)
442                 {
443                     throw new NotImplementedException();
444                 }
445             }
446         });

```

```

447 }
448 else
449 {
450     // o_|      x_o ...
451     // x_|      |___|
452     //
453     // ->
454     first.WalkThroughReferersAsSource(couple =>
455     {
456         if (couple.Linker == Net.And)
457         {
458             var neighbour = couple.Target;
459             if (neighbour.Linker == Net.And && neighbour.Source == second)
460             {
461                 if (neighbour.ReferersByTargetCount == 1 &&
462                     ↪ neighbour.ReferersBySourceCount == 0)
463                 {
464                     if (directConnection == null)
465                     {
466                         directConnection = first & second;
467                     }
468                     Link.Update(ref couple, directConnection, Net.And,
469                         ↪ neighbour.Target);
470                     //Link.Delete(ref neighbour);
471                 }
472             }
473         }
474     });
475 }
476 if (second.ReferersByTargetCount < first.ReferersByTargetCount)
477 {
478     // |_x      ... x_o
479     // |_o      |___|
480     //
481     // <-
482     second.WalkThroughReferersAsTarget(couple =>
483     {
484         if (couple.Linker == Net.And)
485         {
486             var neighbour = couple.Source;
487             if (neighbour.Linker == Net.And && neighbour.Target == first)
488             {
489                 if (neighbour.ReferersByTargetCount == 0 &&
490                     ↪ neighbour.ReferersBySourceCount == 1)
491                 {
492                     if (directConnection == null)
493                     {
494                         directConnection = first & second;
495                     }
496                     Link.Update(ref couple, neighbour.Source, Net.And,
497                         ↪ directConnection);
498                     //Link.Delete(ref neighbour);
499                 }
500             }
501         }
502     });
503 }
504 else
505 {
506     // |_x      ... x_o
507     // |_o      |___|
508     //
509     // ->
510     first.WalkThroughReferersAsTarget((couple) =>
511     {
512         if (couple.Linker == Net.And && couple.ReferersByTargetCount == 0 &&
513             ↪ couple.ReferersBySourceCount == 1)
514         {
515             var neighbour = couple.FirstRefererBySource;
516             if (neighbour.Linker == Net.And && neighbour.Target == second)
517             {
518                 if (directConnection == null)
519                 {
520                     directConnection = first & second;
521                 }
522                 Link.Update(ref neighbour, couple.Source, Net.And, directConnection);
523                 Link.Delete(ref couple);

```

```

520     }
521 }
522 });
523 }
524 if (directConnection != null)
525 {
526     CompressionsCount++;
527 }
528 return directConnection;
529 }
530
531 ///public static Link CombinedJoin(Link left, Link right)
532 ///{
533     Link rightSubJoint = Link.Search(left, Net.And, right.Source);
534     if (rightSubJoint != null && rightSubJoint != right)
535     {
536         long rightSubJointReferers = rightSubJoint.TotalReferers;
537         Link leftSubJoint = Link.Search(left.Target, Net.And, right);
538         if (leftSubJoint != null && leftSubJoint != left)
539         {
540             long leftSubJointReferers = leftSubJoint.TotalReferers;
541             if (leftSubJointReferers > rightSubJointReferers)
542             {
543                 long leftReferers = left.TotalReferers;
544                 if (leftReferers > 0)
545                 {
546                     return Link.Create(left.Source, Net.And, leftSubJoint);
547                 }
548                 else
549                 {
550                     Link.Update(ref left, left.Source, Net.And, leftSubJoint);
551                     return left;
552                 }
553             }
554         }
555         long rightReferers = right.TotalReferers;
556         if (rightReferers > 0)
557         {
558             return Link.Create(rightSubJoint, Net.And, right.Target);
559         }
560         else
561         {
562             Link.Update(ref right, rightSubJoint, Net.And, right.Target);
563             return right;
564         }
565     }
566     return Link.Create(left, Net.And, right);
567 }
568 //public static Link CombinedJoin(Link left, Link right)
569 //{
570 //    long leftReferers = left.TotalReferers;
571 //    Link leftSubJoint = Link.Search(left.Target, Net.And, right);
572 //    if (leftSubJoint != null && leftSubJoint != left)
573 //    {
574 //        long leftSubJointReferers = leftSubJoint.TotalReferers;
575 //    }
576 //    long rightReferers = left.TotalReferers;
577 //    Link rightSubJoint = Link.Search(left, Net.And, right.Source);
578 //    long rightSubJointReferers = rightSubJoint != null ? rightSubJoint.TotalReferers :
579 //    → long.MinValue;
580 //}
581 //public static Link LeftJoinUnsafe(Link subject, Link @object)
582 //{
583 //    if (subject.Linker == Net.And && subject.ReferersBySourceCount == 0 &&
584 //    → subject.ReferersByTargetCount == 0)
585 //    {
586 //        Link subJoint = Link.Search(@object, Net.And, subject.Source);
587 //        if (subJoint != null && subJoint != subject)
588 //        {
589 //            Link.Update(ref subject, subJoint, Net.And, subject.Target);
590 //            return subject;
591 //        }
592 //    }
593 //    return Link.Create(@object, Net.And, subject);
594 //}
595
596 /// <summary>
597 /// <para>

```

```

596     /// The chunk size.
597     /// </para>
598     /// <para></para>
599     /// </summary>
600     public static int ChunkSize = 2;
601
602     ///public static Link FromList(List<Link> links)
603     //{
604     //    Link element = links[0];
605     //    for (int i = 1; i < links.Count; i += ChunkSize)
606     //    {
607     //        int j = (i + ChunkSize - 1);
608     //        j = j < links.Count ? j : (links.Count - 1);
609     //        Link subElement = links[j];
610     //        while (--j >= i) LeftJoin(ref subElement, links[j]);
611     //        RightJoin(ref element, subElement);
612     //    }
613     //    return element;
614     //}
615
616     ///public static Link FromList(Link[] links)
617     //{
618     //    Link element = links[0];
619     //    for (int i = 1; i < links.Length; i += ChunkSize)
620     //    {
621     //        int j = (i + ChunkSize - 1);
622     //        j = j < links.Length ? j : (links.Length - 1);
623     //        Link subElement = links[j];
624     //        while (--j >= i) LeftJoin(ref subElement, links[j]);
625     //        RightJoin(ref element, subElement);
626     //    }
627     //    return element;
628     //}
629
630     ///public static Link FromList(ICollection<Link> links)
631     //{
632     //    Link element = links[0];
633     //    for (int i = 1; i < links.Count; i += ChunkSize)
634     //    {
635     //        int j = (i + ChunkSize - 1);
636     //        j = j < links.Count ? j : (links.Count - 1);
637     //        Link subElement = links[j];
638     //        while (--j >= i)
639     //        {
640     //            Link x = links[j];
641     //            subElement = CombinedJoin(ref x, ref subElement);
642     //        }
643     //        element = CombinedJoin(ref element, ref subElement);
644     //    }
645     //    return element;
646     //}
647     ///public static Link FromList(ICollection<Link> links)
648     //{
649     //    int i = 0;
650     //    Link element = links[i++];
651     //    if (links.Count % 2 == 0)
652     //    {
653     //        element = CombinedJoin(element, links[i++]);
654     //    }
655     //    for (; i < links.Count; i += 2)
656     //    {
657     //        Link doublet = CombinedJoin(links[i], links[i + 1]);
658     //        element = CombinedJoin(ref element, ref doublet);
659     //    }
660     //    return element;
661     //}
662
663     /// Заглушка, возможно опасная
664     private static Link CombinedJoin(Link element, Link link)
665     {
666         return CombinedJoin(ref element, ref link);
667     }
668
669     ///public static Link FromList(List<Link> links)
670     //{
671     //    int i = links.Count - 1;
672     //    Link element = links[i];

```

```

673 // while (--i >= 0) element = LinkConverterOld.ConnectLinks2(links[i], element,
674 // links, ref i);
675 // return element;
676 //}
677 //public static Link FromList(Link[] links)
678 //{
679 //    int i = links.Length - 1;
680 //    Link element = links[i];
681 //    while (--i >= 0) element = LinkConverterOld.ConnectLinks2(links[i], element,
682 // links, ref i);
683 //    return element;
684 //}
685 //public static Link FromList(List<Link> links)
686 //{
687 //    Link element = links[0];
688 //    for (int i = 1; i < links.Count; i += ChunkSize)
689 //    {
690 //        int j = (i + ChunkSize - 1);
691 //        j = j < links.Count ? j : (links.Count - 1);
692 //        Link subElement = links[j];
693 //        while (--j >= i) subElement = CombinedJoin(links[j], subElement);
694 //        element = CombinedJoin(element, subElement);
695 //    }
696 //    return element;
697 //}
698 //public static Link FromList(Link[] links)
699 //{
700 //    Link element = links[0];
701 //    for (int i = 1; i < links.Length; i += ChunkSize)
702 //    {
703 //        int j = (i + ChunkSize - 1);
704 //        j = j < links.Length ? j : (links.Length - 1);
705 //        Link subElement = links[j];
706 //        while (--j >= i) subElement = CombinedJoin(links[j], subElement);
707 //        element = CombinedJoin(element, subElement);
708 //    }
709 //    return element;
710 //}
711 //public static Link FromList(ICollection<Link> links)
712 //{
713 //    int leftBound = 0;
714 //    int rightBound = links.Count - 1;
715 //    if (leftBound == rightBound)
716 //    {
717 //        return links[0];
718 //    }
719 //    Link left = links[leftBound];
720 //    Link right = links[rightBound];
721 //    long leftReferers = left.ReferersBySourceCount + left.ReferersByTargetCount;
722 //    long rightReferers = right.ReferersBySourceCount + right.ReferersByTargetCount;
723 //    while (true)
724 //    {
725 //        //if (rightBound % 2 != leftBound % 2)
726 //        if (rightReferers >= leftReferers)
727 //        {
728 //            int nextRightBound = --rightBound;
729 //            if (nextRightBound == leftBound)
730 //            {
731 //                var x = CombinedJoin(ref left, ref right);
732 //                return x;
733 //            }
734 //            else
735 //            {
736 //                Link nextRight = links[nextRightBound];
737 //                right = CombinedJoin(ref nextRight, ref right);
738 //                rightReferers = right.ReferersBySourceCount +
739 // right.ReferersByTargetCount;
740 //            }
741 //        }
742 //        else
743 //        {
744 //            int nextLeftBound = ++leftBound;
745 //            if (nextLeftBound == rightBound)
746 //            {
747 //                return CombinedJoin(ref left, ref right);
748 //            }
749 //            else

```

```

747         //         {
748         //             Link nextLeft = links[nextLeftBound];
749         //             left = CombinedJoin(ref left, ref nextLeft);
750         //             leftReferers = left.ReferersBySourceCount + left.ReferersByTargetCount;
751         //         }
752         //     }
753     }
754 }
755 //public static Link FromList(IList<Link> links)
756 //{
757 //    int i = links.Count - 1;
758 //    Link element = links[i];
759 //    while (--i >= 0)
760 //    {
761 //        LeftJoin(ref element, links[i]); // LeftJoin(ref element, links[i]);
762 //    }
763 //    return element;
764 //}
765
766 /// <summary>
767 /// <para>
768 /// Creates the list using the specified links.
769 /// </para>
770 /// <para></para>
771 /// </summary>
772 /// <param name="links">
773 /// <para>The links.</para>
774 /// <para></para>
775 /// </param>
776 /// <returns>
777 /// <para>The element.</para>
778 /// <para></para>
779 /// </returns>
780 public static Link FromList(List<Link> links)
781 {
782     var i = links.Count - 1;
783     var element = links[i];
784     while (--i >= 0)
785     {
786         var x = links[i];
787         element = CombinedJoin(ref x, ref element); // LeftJoin(ref element, links[i]);
788     }
789     return element;
790 }
791
792 /// <summary>
793 /// <para>
794 /// Creates the list using the specified links.
795 /// </para>
796 /// <para></para>
797 /// </summary>
798 /// <param name="links">
799 /// <para>The links.</para>
800 /// <para></para>
801 /// </param>
802 /// <returns>
803 /// <para>The element.</para>
804 /// <para></para>
805 /// </returns>
806 public static Link FromList(Link[] links)
807 {
808     var i = links.Length - 1;
809     var element = links[i];
810     while (--i >= 0)
811     {
812         element = CombinedJoin(ref links[i], ref element); // LeftJoin(ref element,
813             ↪ links[i]);
814     }
815     return element;
816 }
817 }

```

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             sb.Append('}');
68             return sb.ToString();
69         }
70
71         /// <summary>
72         /// <para>
73         /// Collects the matching sequences using the specified links.
74         /// </para>
75         /// <para></para>
76         /// </summary>
77         /// <param name="links">
78         /// <para>The links.</para>
79         /// <para></para>
80         /// </param>
81         /// <exception cref="InvalidOperationException">
82         /// <para>Подпоследовательности с одним элементом не поддерживаются.</para>
83         /// <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     private static void CollectMatchingSequences(Link leftLink, int leftBound, Link[]
103         ↳ middleLinks, Link rightLink, int rightBound, ref List<Link> results)
104     {
105         var leftLinkTotalReferers = leftLink.ReferersBySourceCount +
106             ↳ leftLink.ReferersByTargetCount;
107         var rightLinkTotalReferers = rightLink.ReferersBySourceCount +
108             ↳ rightLink.ReferersByTargetCount;
109         if (leftLinkTotalReferers <= rightLinkTotalReferers)
110         {
111             var nextLeftLink = middleLinks[leftBound];
112             var elements = GetRightElements(leftLink, nextLeftLink);
113             if (leftBound <= rightBound)
114             {
115                 for (var i = elements.Length - 1; i >= 0; i--)
116                 {
117                     var element = elements[i];
118                     if (element != null)
119                     {
120                         CollectMatchingSequences(element, leftBound + 1, middleLinks,
121                             ↳ rightLink, rightBound, ref results);
122                     }
123                 }
124             }
125             else
126             {
127                 for (var i = elements.Length - 1; i >= 0; i--)
128                 {
129                     var element = elements[i];
130                     if (element != null)
131                     {
132                         results.Add(element);
133                     }
134                 }
135             }
136         }
137         else
138         {
139             var nextRightLink = middleLinks[rightBound];
140             var elements = GetLeftElements(rightLink, nextRightLink);
141             if (leftBound <= rightBound)
142             {
143                 for (var i = elements.Length - 1; i >= 0; i--)
144                 {
145                     var element = elements[i];
146                     if (element != null)
147                     {
148                         CollectMatchingSequences(leftLink, leftBound, middleLinks,
149                             ↳ elements[i], rightBound - 1, ref results);
150                     }
151                 }
152             }
153             else
154             {
155                 for (var i = elements.Length - 1; i >= 0; i--)
156                 {
157                     var element = elements[i];
158                     if (element != null)
159                     {
160

```

```

154         results.Add(element);
155     }
156 }
157 }
158 }
159 }
160
161 /// <summary>
162 /// <para>
163 /// Gets the right elements using the specified start link.
164 /// </para>
165 /// <para></para>
166 /// </summary>
167 /// <param name="startLink">
168 /// <para>The start link.</para>
169 /// <para></para>
170 /// </param>
171 /// <param name="rightLink">
172 /// <para>The right link.</para>
173 /// <para></para>
174 /// </param>
175 /// <returns>
176 /// <para>The result.</para>
177 /// <para></para>
178 /// </returns>
179 public static Link[] GetRightElements(Link startLink, Link rightLink)
180 {
181     var result = new Link[4];
182     TryStepRight(startLink, rightLink, result, 0);
183     startLink.WalkThroughReferersAsTarget(couple =>
184     {
185         if (couple.Linker == Net.And)
186         {
187             if (TryStepRight(couple, rightLink, result, 2))
188             {
189                 return Link.Stop;
190             }
191         }
192         return Link.Continue;
193     });
194     return result;
195 }
196
197 /// <summary>
198 /// <para>
199 /// Determines whether try step right.
200 /// </para>
201 /// <para></para>
202 /// </summary>
203 /// <param name="startLink">
204 /// <para>The start link.</para>
205 /// <para></para>
206 /// </param>
207 /// <param name="rightLink">
208 /// <para>The right link.</para>
209 /// <para></para>
210 /// </param>
211 /// <param name="result">
212 /// <para>The result.</para>
213 /// <para></para>
214 /// </param>
215 /// <param name="offset">
216 /// <para>The offset.</para>
217 /// <para></para>
218 /// </param>
219 /// <returns>
220 /// <para>The bool</para>
221 /// <para></para>
222 /// </returns>
223 public static bool TryStepRight(Link startLink, Link rightLink, Link[] result, int
    ↳ offset)
224 {
225     var added = 0;
226     startLink.WalkThroughReferersAsSource(couple =>
227     {
228         if (couple.Linker == Net.And)
229         {
230             var coupleTarget = couple.Target;

```

```

231         if (coupleTarget == rightLink)
232         {
233             result[offset] = couple;
234             if (++added == 2)
235             {
236                 return Link.Stop;
237             }
238         }
239         else if (coupleTarget.Linker == Net.And && coupleTarget.Source ==
↵ rightLink)
240         {
241             result[offset + 1] = couple;
242             if (++added == 2)
243             {
244                 return Link.Stop;
245             }
246         }
247     }
248     return Link.Continue;
249 });
250 return added > 0;
251 }
252
253 /// <summary>
254 /// <para>
255 /// Gets the left elements using the specified start link.
256 /// </para>
257 /// <para></para>
258 /// </summary>
259 /// <param name="startLink">
260 /// <para>The start link.</para>
261 /// <para></para>
262 /// </param>
263 /// <param name="leftLink">
264 /// <para>The left link.</para>
265 /// <para></para>
266 /// </param>
267 /// <returns>
268 /// <para>The result.</para>
269 /// <para></para>
270 /// </returns>
271 public static Link[] GetLeftElements(Link startLink, Link leftLink)
272 {
273     var result = new Link[4];
274     TryStepLeft(startLink, leftLink, result, 0);
275     startLink.WalkThroughReferersAsSource(couple =>
276     {
277         if (couple.Linker == Net.And)
278         {
279             if (TryStepLeft(couple, leftLink, result, 2))
280             {
281                 return Link.Stop;
282             }
283         }
284         return Link.Continue;
285     });
286     return result;
287 }
288
289 /// <summary>
290 /// <para>
291 /// Determines whether try step left.
292 /// </para>
293 /// <para></para>
294 /// </summary>
295 /// <param name="startLink">
296 /// <para>The start link.</para>
297 /// <para></para>
298 /// </param>
299 /// <param name="leftLink">
300 /// <para>The left link.</para>
301 /// <para></para>
302 /// </param>
303 /// <param name="result">
304 /// <para>The result.</para>
305 /// <para></para>
306 /// </param>
307 /// <param name="offset">

```

```

308     /// <para>The offset.</para>
309     /// <para></para>
310     /// </param>
311     /// <returns>
312     /// <para>The bool</para>
313     /// <para></para>
314     /// </returns>
315     public static bool TryStepLeft(Link startLink, Link leftLink, Link[] result, int offset)
316     {
317         var added = 0;
318         startLink.WalkThroughReferersAsTarget(couple =>
319         {
320             if (couple.Linker == Net.And)
321             {
322                 var coupleSource = couple.Source;
323                 if (coupleSource == leftLink)
324                 {
325                     result[offset] = couple;
326                     if (++added == 2)
327                     {
328                         return Link.Stop;
329                     }
330                 }
331                 else if (coupleSource.Linker == Net.And && coupleSource.Target ==
332                     leftLink)
333                 {
334                     result[offset + 1] = couple;
335                     if (++added == 2)
336                     {
337                         return Link.Stop;
338                     }
339                 }
340                 return Link.Continue;
341             });
342         return added > 0;
343     }
344 }
345 }

```

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      public static class LinkTests
9      {
10         public static object Lock = new object(); //-V3090
11         private static ulong _thingVisitorCounter;
12         private static ulong _isAVisitorCounter;
13         private static ulong _linkVisitorCounter;
14
15         static void ThingVisitor(Link linkIndex)
16         {
17             _thingVisitorCounter += linkIndex;
18         }
19
20         static void IsAVisitor(Link linkIndex)
21         {
22             _isAVisitorCounter += linkIndex;
23         }
24
25         static void LinkVisitor(Link linkIndex)
26         {
27             _linkVisitorCounter += linkIndex;
28         }
29
30         [Fact]
31         public static void CreateDeleteLinkTest()
32         {
33             lock (Lock)
34             {
35                 string filename = "db.links";
36
37                 File.Delete(filename);
38
39                 Link.StartMemoryManager(filename);

```

```

40
41     Link link1 = Link.Create(Link.Itself, Link.Itself, Link.Itself);
42
43     Link.Delete(link1);
44
45     Link.StopMemoryManager();
46
47     File.Delete(filename);
48 }
49 }
50
51 [Fact]
52 public static void DeepCreateUpdateDeleteLinkTest()
53 {
54     lock (Lock)
55     {
56         string filename = "db.links";
57
58         File.Delete(filename);
59
60         Link.StartMemoryManager(filename);
61
62         Link isA = Link.Create(Link.Itself, Link.Itself, Link.Itself);
63         Link isNotA = Link.Create(Link.Itself, Link.Itself, isA);
64         Link link = Link.Create(Link.Itself, isA, Link.Itself);
65         Link thing = Link.Create(Link.Itself, isNotA, link);
66
67         //Assert::IsTrue(GetLinksCount() == 4);
68
69         Assert.Equal(isA, isA.Target);
70
71         isA = Link.Update(isA, isA, isA, link); // Произведено замыкание
72
73         Assert.Equal(link, isA.Target);
74
75         Link.Delete(isA); // Одна эта операция удалит все 4 связи
76
77         //Assert::IsTrue(GetLinksCount() == 0);
78
79         Link.StopMemoryManager();
80
81         File.Delete(filename);
82     }
83 }
84
85 [Fact]
86 public static void LinkReferersWalkTest()
87 {
88     lock (Lock)
89     {
90         string filename = "db.links";
91
92         File.Delete(filename);
93
94         Link.StartMemoryManager(filename);
95
96         Link isA = Link.Create(Link.Itself, Link.Itself, Link.Itself);
97         Link isNotA = Link.Create(Link.Itself, Link.Itself, isA);
98         Link link = Link.Create(Link.Itself, isA, Link.Itself);
99         Link thing = Link.Create(Link.Itself, isNotA, link);
100         isA = Link.Update(isA, isA, isA, link);
101
102         Assert.Equal(1, thing.ReferersBySourceCount);
103         Assert.Equal(2, isA.ReferersByLinkerCount);
104         Assert.Equal(3, link.ReferersByTargetCount);
105
106         _thingVisitorCounter = 0;
107         _isAVisitorCounter = 0;
108         _linkVisitorCounter = 0;
109
110         thing.WalkThroughReferersAsSource(ThingVisitor);
111         isA.WalkThroughReferersAsLinker(IsAVisitor);
112         link.WalkThroughReferersAsTarget(LinkVisitor);
113
114         Assert.Equal(4UL, _thingVisitorCounter);
115         Assert.Equal(1UL + 3UL, _isAVisitorCounter);
116         Assert.Equal(1UL + 3UL + 4UL, _linkVisitorCounter);
117
118         Link.StopMemoryManager();
119

```

```

120         File.Delete(filename);
121     }
122 }
123
124 [Fact]
125 public static void MultipleRandomCreationsAndDeletionsTest()
126 {
127     lock (Lock)
128     {
129         string filename = "db.links";
130
131         File.Delete(filename);
132
133         Link.StartMemoryManager(filename);
134
135         TestMultipleRandomCreationsAndDeletions(2000);
136
137         Link.StopMemoryManager();
138
139         File.Delete(filename);
140     }
141 }
142 private static void TestMultipleRandomCreationsAndDeletions(int
↪ maximumOperationsPerCycle)
143 {
144     var and = Link.Create(Link.Itself, Link.Itself, Link.Itself);
145     //var comparer = Comparer<TLinkAddress>.Default;
146     for (var N = 1; N < maximumOperationsPerCycle; N++)
147     {
148         var random = new System.Random(N);
149         var linksCount = 1;
150         for (var i = 0; i < N; i++)
151         {
152             var createPoint = random.NextBoolean();
153             if (linksCount > 2 && createPoint)
154             {
155                 var linksAddressRange = new Range<ulong>(1, (ulong)linksCount);
156                 Link source = random.NextUInt64(linksAddressRange);
157                 Link target = random.NextUInt64(linksAddressRange); //-V3086
158                 var resultLink = Link.Create(source, and, target);
159                 if (resultLink > linksCount)
160                 {
161                     linksCount++;
162                 }
163             }
164             else
165             {
166                 Link.Create(Link.Itself, Link.Itself, Link.Itself);
167                 linksCount++;
168             }
169         }
170         for (var i = 0; i < N; i++)
171         {
172             Link link = i + 2;
173             if (link.Linker != null)
174             {
175                 Link.Delete(link);
176                 linksCount--;
177             }
178         }
179     }
180 }
181 }
182 }

```

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     public static class PersistentMemoryManagerTests
7     {
8         [Fact]
9         public static void FileMappingTest()
10         {
11             lock (LinkTests.Lock)
12             {
13                 string filename = "db.links";

```

```

14         File.Delete(filename);
15
16         Link.StartMemoryManager(filename);
17
18         Link.StopMemoryManager();
19
20         File.Delete(filename);
21     }
22 }
23
24 [Fact]
25 public static void AllocateAndFreeLinkTest()
26 {
27     lock (LinkTests.Lock)
28     {
29         string filename = "db.links";
30
31         File.Delete(filename);
32
33         Link.StartMemoryManager(filename);
34
35         Link link = Link.Create(Link.Itself, Link.Itself, Link.Itself);
36
37         Link.Delete(link);
38
39         Link.StopMemoryManager();
40
41         File.Delete(filename);
42     }
43 }
44
45 [Fact]
46 public static void AttachToUnusedLinkTest()
47 {
48     lock (LinkTests.Lock)
49     {
50         string filename = "db.links";
51
52         File.Delete(filename);
53
54         Link.StartMemoryManager(filename);
55
56         Link link1 = Link.Create(Link.Itself, Link.Itself, Link.Itself);
57         Link link2 = Link.Create(Link.Itself, Link.Itself, Link.Itself);
58
59         Link.Delete(link1); // Creates "hole" and forces "Attach" to be executed
60
61         Link.StopMemoryManager();
62
63         File.Delete(filename);
64     }
65 }
66
67 [Fact]
68 public static void DetachToUnusedLinkTest()
69 {
70     lock (LinkTests.Lock)
71     {
72         string filename = "db.links";
73
74         File.Delete(filename);
75
76         Link.StartMemoryManager(filename);
77
78         Link link1 = Link.Create(Link.Itself, Link.Itself, Link.Itself);
79         Link link2 = Link.Create(Link.Itself, Link.Itself, Link.Itself);
80
81         Link.Delete(link1); // Creates "hole" and forces "Attach" to be executed
82         Link.Delete(link2); // Removes both links, all "Attached" links forced to be
83             ↪ "Detached" here
84
85         Link.StopMemoryManager();
86
87         File.Delete(filename);
88     }
89 }
90
91 [Fact]
92 public static void GetSetMappedLinkTest()

```



```
93 {
94     lock (LinkTests.Lock)
95     {
96         string filename = "db.links";
97         File.Delete(filename);
98         Link.StartMemoryManager(filename);
99         Link.StartMemoryManager(filename);
100         Link.StartMemoryManager(filename);
101         var mapped = Link.GetMappedOrDefault(0);
102         var mappingSet = Link.TrySetMapped(mapped, 0);
103         Assert.True(mappingSet);
104         Link.StopMemoryManager();
105         File.Delete(filename);
106     }
107 }
108 }
109 }
110 }
```

Index

./csharp/Platform.Data.Triplets.Tests/LinkTests.cs, 77
./csharp/Platform.Data.Triplets.Tests/PersistentMemoryManagerTests.cs, 79
./csharp/Platform.Data.Triplets/CharacterHelpers.cs, 1
./csharp/Platform.Data.Triplets/GexfExporter.cs, 5
./csharp/Platform.Data.Triplets/ILink.cs, 7
./csharp/Platform.Data.Triplets/Link.Debug.cs, 9
./csharp/Platform.Data.Triplets/Link.cs, 11
./csharp/Platform.Data.Triplets/LinkConverter.cs, 36
./csharp/Platform.Data.Triplets/LinkExtensions.cs, 44
./csharp/Platform.Data.Triplets/Net.cs, 49
./csharp/Platform.Data.Triplets/NumberHelpers.cs, 59
./csharp/Platform.Data.Triplets/Sequences/CompressionExperiments.cs, 61
./csharp/Platform.Data.Triplets/Sequences/SequenceHelpers.cs, 72