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

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

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

namespace Platform.Data.Triplets
{
    // TODO: Split logic of Latin and Cyrillic alphabets into different files if possible
    /// <summary>
    /// <para>
    /// Represents the character helpers.
    /// </para>
    /// <para></para>
    /// </summary>
    public static class CharacterHelpers
    {
        /// <summary>
        /// <para>
        /// The character mapping enum.
        /// </para>
        /// <para></para>
        /// </summary>
        public enum CharacterMapping : long
        {
            /// <summary>
            /// <para>
            /// The latin alphabet character mapping.
            /// </para>
            /// <para></para>
            /// </summary>
            LatinAlphabet = 100,
            /// <summary>
            /// <para>
            /// The cyrillic alphabet character mapping.
            /// </para>
            /// <para></para>
            /// </summary>
            CyrillicAlphabet
        }
        private const char FirstLowerCaseLatinLetter = 'a';
        private const char LastLowerCaseLatinLetter = 'z';
        private const char FirstUpperCaseLatinLetter = 'A';
        private const char LastUpperCaseLatinLetter = 'Z';
        private const char FirstLowerCaseCyrillicLetter = 'а';
        private const char LastLowerCaseCyrillicLetter = 'я';
        private const char FirstUpperCaseCyrillicLetter = 'А';
        private const char LastUpperCaseCyrillicLetter = 'Я';
        private const char YoLowerCaseCyrillicLetter = 'ё';
        private const char YoUpperCaseCyrillicLetter = 'Ё';
        private static Link[] _charactersToLinks;
        private static Dictionary<Link, char> _linksToCharacters;

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="CharacterHelpers"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        static CharacterHelpers() => Create();
        private static void Create()
        {
            _charactersToLinks = new Link[char.MaxValue];
            _linksToCharacters = new Dictionary<Link, char>();
            // Create or restore characters
            CreateLatinAlphabet();
            CreateCyrillicAlphabet();
            RegisterExistingCharacters();
        }
        private static void RegisterExistingCharacters() =>
        ↪ Net.Character.WalkThroughReferersAsSource(referer =>
        ↪ RegisterExistingCharacter(referer));
        private static void RegisterExistingCharacter(Link character)
        {
            if (character.Source == Net.Character && character.Linker == Net.ThatHas)
            {
                var code = character.Target;
                if (code.Source == Net.Code && code.Linker == Net.ThatIsRepresentedBy)
```

```
            {
                var charCode = (char)LinkConverter.ToNumber(code.Target);
                _charactersToLinks[charCode] = character;
                _linksToCharacters[character] = charCode;
            }
        }
    }

    /// <summary>
    /// <para>
    /// Recreates.
    /// </para>
    /// <para></para>
    /// </summary>
    public static void Recreate() => Create();
    private static void CreateLatinAlphabet()
    {
        var lettersCharacters = new[]
        {
            'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j',
            'k', 'l', 'm', 'n', 'o', 'p', 'q', 'r', 's', 't',
            'u', 'v', 'w', 'x', 'y', 'z'
        };
        CreateAlphabet(lettersCharacters, "latin alphabet", CharacterMapping.LatinAlphabet);
    }
    private static void CreateCyrillicAlphabet()
    {
        var lettersCharacters = new[]
        {
            'а', 'б', 'в', 'г', 'д', 'е', 'ё', 'ж', 'з', 'и',
            'й', 'к', 'л', 'м', 'н', 'о', 'п', 'р', 'с', 'т',
            'у', 'ф', 'х', 'ц', 'ч', 'ш', 'щ', 'ъ', 'ы', 'ь',
            'э', 'ю', 'я'
        };
        CreateAlphabet(lettersCharacters, "cyrillic alphabet",
        ↪  CharacterMapping.CyrillicAlphabet);
    }
    private static void CreateAlphabet(char[] lettersCharacters, string alphabetName,
    ↪  CharacterMapping mapping)
    {
        if (Link.TryGetMapped(mapping, out Link alphabet))
        {
            var letters = alphabet.Target;
            letters.WalkThroughSequence(letter =>
            {
                var lowerCaseLetter = Link.Search(Net.LowerCase, Net.Of, letter);
                var upperCaseLetter = Link.Search(Net.UpperCase, Net.Of, letter);
                if (lowerCaseLetter != null && upperCaseLetter != null)
                {
                    RegisterExistingLetter(lowerCaseLetter);
                    RegisterExistingLetter(upperCaseLetter);
                }
                else
                {
                    RegisterExistingLetter(letter);
                }
            });
        }
        else
        {
            alphabet = Net.CreateMappedThing(mapping);
            var letterOfAlphabet = Link.Create(Net.Letter, Net.Of, alphabet);
            var lettersLinks = new Link[lettersCharacters.Length];
            GenerateAlphabetBasis(ref alphabet, ref letterOfAlphabet, lettersLinks);
            for (var i = 0; i < lettersCharacters.Length; i++)
            {
                var lowerCaseCharacter = lettersCharacters[i];
                SetLetterCodes(lettersLinks[i], lowerCaseCharacter, out Link lowerCaseLink,
                ↪  out Link upperCaseLink);
                _charactersToLinks[lowerCaseCharacter] = lowerCaseLink;
                _linksToCharacters[lowerCaseLink] = lowerCaseCharacter;
                if (upperCaseLink != null)
                {
                    var upperCaseCharacter = char.ToUpper(lowerCaseCharacter);
                    _charactersToLinks[upperCaseCharacter] = upperCaseLink;
                    _linksToCharacters[upperCaseLink] = upperCaseCharacter;
                }
            }
```

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

```
                Link.Create(upperCase, Net.Has, upperCaseCharacterCode);
            }
            else
            {
                lowerCase = letter;
                upperCase = null;
                Link.Create(letter, Net.Has, Link.Create(Net.Code, Net.ThatIsRepresentedBy,
                ↪  LinkConverter.FromNumber(lowerCaseCharacter)));
            }
        }
        private static Link CreateSimpleCharacterLink(char character) =>
        ↪  Link.Create(Net.Character, Net.ThatHas, Link.Create(Net.Code,
        ↪  Net.ThatIsRepresentedBy, LinkConverter.FromNumber(character)));
        private static bool IsLetterOfLatinAlphabet(char character)
            => (character >= FirstLowerCaseLatinLetter && character <= LastLowerCaseLatinLetter)
            || (character >= FirstUpperCaseLatinLetter && character <= LastUpperCaseLatinLetter);
        private static bool IsLetterOfCyrillicAlphabet(char character)
            => (character >= FirstLowerCaseCyrillicLetter && character <=
            ↪  LastLowerCaseCyrillicLetter)
            || (character >= FirstUpperCaseCyrillicLetter && character <=
            ↪  LastUpperCaseCyrillicLetter)
            || character == YoLowerCaseCyrillicLetter || character == YoUpperCaseCyrillicLetter;

        /// <summary>
        /// <para>
        /// Creates the char using the specified character.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="character">
        /// <para>The character.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        public static Link FromChar(char character)
        {
            if (_charactersToLinks[character] == null)
            {
                if (IsLetterOfLatinAlphabet(character))
                {
                    CreateLatinAlphabet();
                    return _charactersToLinks[character];
                }
                else if (IsLetterOfCyrillicAlphabet(character))
                {
                    CreateCyrillicAlphabet();
                    return _charactersToLinks[character];
                }
                else
                {
                    var simpleCharacter = CreateSimpleCharacterLink(character);
                    _charactersToLinks[character] = simpleCharacter;
                    _linksToCharacters[simpleCharacter] = character;
                    return simpleCharacter;
                }
            }
            else
            {
                return _charactersToLinks[character];
            }
        }

        /// <summary>
        /// <para>
        /// Returns the char using the specified link.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="link">
        /// <para>The link.</para>
        /// <para></para>
        /// </param>
        /// <exception cref="ArgumentOutOfRangeException">
        /// <para>Указанная связь не являяется символом.</para>
        /// <para></para>
```

```
        /// </exception>
        /// <returns>
        /// <para>The char.</para>
        /// <para></para>
        /// </returns>
        public static char ToChar(Link link)
        {
            if (!_linksToCharacters.TryGetValue(link, out char @char))
            {
                throw new ArgumentOutOfRangeException(nameof(link), "Указанная связь не
                ↪  является символом.");
            }
            return @char;
        }

        /// <summary>
        /// <para>
        /// Determines whether is char.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="link">
        /// <para>The link.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        public static bool IsChar(Link link) => link != null &&
        ↪  _linksToCharacters.ContainsKey(link);
    }
}
```

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

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Text;
using System.Xml;
using Platform.Collections.Sets;
using Platform.Communication.Protocol.Gexf;
using GexfNode = Platform.Communication.Protocol.Gexf.Node;

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

namespace Platform.Data.Triplets
{
    /// <summary>
    /// <para>
    /// Represents the gexf exporter.
    /// </para>
    /// <para></para>
    /// </summary>
    public static class GexfExporter
    {
        private const string SourceLabel = "Source";
        private const string LinkerLabel = "Linker";
        private const string TargetLabel = "Target";

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

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

        /// <summary>
        /// <para>
        /// Returns the file using the specified path.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="path">
        /// <para>The path.</para>
        /// <para></para>
        /// </param>
        /// <param name="filter">
        /// <para>The filter.</para>
        /// <para></para>
        /// </param>
        public static void ToFile(string path, Func<Link, bool> filter)
        {
            using (var file = File.OpenWrite(path))
            using (var writer = XmlWriter.Create(file))
            {
                WriteXml(writer, CollectLinks(filter));
            }
        }
        private static HashSet<Link> CollectLinks(Func<Link, bool> linkMatch)
        {
            var matchingLinks = new HashSet<Link>();
            Link.WalkThroughAllLinks(link =>
            {
                if (linkMatch(link))
                {
                    matchingLinks.Add(link);
                }
            });
            return matchingLinks;
        }
        private static HashSet<Link> CollectLinks()
        {
            var matchingLinks = new HashSet<Link>();
            Link.WalkThroughAllLinks(matchingLinks.AddAndReturnVoid);
            return matchingLinks;
        }
        private static void WriteXml(XmlWriter writer, HashSet<Link> matchingLinks)
        {
            var edgesCounter = 0;
            Gexf.WriteXml(writer,
            () => // nodes
            {
                foreach (var matchingLink in matchingLinks)
                {
                    GexfNode.WriteXml(writer, matchingLink.ToInt(), matchingLink.ToString());
                }
            },
            () => // edges
            {
                foreach (var matchingLink in matchingLinks)
                {
                    if (matchingLinks.Contains(matchingLink.Source))
                    {
```

```
                        Edge.WriteXml(writer, edgesCounter++, matchingLink.ToInt(),
                        ↪  matchingLink.Source.ToInt(), SourceLabel);
                    }
                    if (matchingLinks.Contains(matchingLink.Linker))
                    {
                        Edge.WriteXml(writer, edgesCounter++, matchingLink.ToInt(),
                        ↪  matchingLink.Linker.ToInt(), LinkerLabel);
                    }
                    if (matchingLinks.Contains(matchingLink.Target))
                    {
                        Edge.WriteXml(writer, edgesCounter++, matchingLink.ToInt(),
                        ↪  matchingLink.Target.ToInt(), TargetLabel);
                    }
                }
            });
        }
    }
}
```

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

```
using System;

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

namespace Platform.Data.Triplets
{
    /// <summary>
    /// <para>
    /// Defines the link.
    /// </para>
    /// <para></para>
    /// </summary>
    internal partial interface ILink<TLink>
        where TLink : ILink<TLink>
    {
        /// <summary>
        /// <para>
        /// Gets the source value.
        /// </para>
        /// <para></para>
        /// </summary>
        TLink Source { get; }
        /// <summary>
        /// <para>
        /// Gets the linker value.
        /// </para>
        /// <para></para>
        /// </summary>
        TLink Linker { get; }
        /// <summary>
        /// <para>
        /// Gets the target value.
        /// </para>
        /// <para></para>
        /// </summary>
        TLink Target { get; }
    }

    /// <summary>
    /// <para>
    /// Defines the link.
    /// </para>
    /// <para></para>
    /// </summary>
    internal partial interface ILink<TLink>
        where TLink : ILink<TLink>
    {
        /// <summary>
        /// <para>
        /// Determines whether this instance walk through referers as linker.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="walker">
        /// <para>The walker.</para>
        /// <para></para>
        /// </param>
        /// <returns>
```

```
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        bool WalkThroughReferersAsLinker(Func<TLink, bool> walker);
        /// <summary>
        /// <para>
        /// Determines whether this instance walk through referers as source.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="walker">
        /// <para>The walker.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        bool WalkThroughReferersAsSource(Func<TLink, bool> walker);
        /// <summary>
        /// <para>
        /// Determines whether this instance walk through referers as target.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="walker">
        /// <para>The walker.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        bool WalkThroughReferersAsTarget(Func<TLink, bool> walker);
        /// <summary>
        /// <para>
        /// Walks the through referers using the specified walker.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="walker">
        /// <para>The walker.</para>
        /// <para></para>
        /// </param>
        void WalkThroughReferers(Func<TLink, bool> walker);
    }

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

```
        /// <summary>
        /// <para>
        /// Walks the through referers as target using the specified walker.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="walker">
        /// <para>The walker.</para>
        /// <para></para>
        /// </param>
        void WalkThroughReferersAsTarget(Action<TLink> walker);
        /// <summary>
        /// <para>
        /// Walks the through referers using the specified walker.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="walker">
        /// <para>The walker.</para>
        /// <para></para>
        /// </param>
        void WalkThroughReferers(Action<TLink> walker);
    }
}
/*
using System;
namespace NetLibrary
{
    interface ILink
    {
        // Статические методы (общие для всех связей)
        public static ILink Create(ILink source, ILink linker, ILink target);
        public static void Update(ref ILink link, ILink newSource, ILink newLinker, ILink
↪  newTarget);
        public static void Delete(ref ILink link);
        public static ILink Search(ILink source, ILink linker, ILink target);
    }
}
*/
/*
Набор функций, который необходим для работы с сущностью Link:

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

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

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

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

```
using System;
using System.Diagnostics;

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

namespace Platform.Data.Triplets
{
    /// <summary>
    /// <para>
    /// The link.
```

```
    /// </para>
    /// <para></para>
    /// </summary>
    public partial struct Link
    {
        #region Properties

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

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

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

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

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

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

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

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

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

```
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>
```

```
    {
        /// <summary>
        /// <para>
        /// The source.
        /// </para>
        /// <para></para>
        /// </summary>
        public readonly Link Source;
        /// <summary>
        /// <para>
        /// The linker.
        /// </para>
        /// <para></para>
        /// </summary>
        public readonly Link Linker;
        /// <summary>
        /// <para>
        /// The target.
        /// </para>
        /// <para></para>
        /// </summary>
        public readonly Link Target;

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="LinkDefinition"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="source">
        /// <para>A source.</para>
        /// <para></para>
        /// </param>
        /// <param name="linker">
        /// <para>A linker.</para>
        /// <para></para>
        /// </param>
        /// <param name="target">
        /// <para>A target.</para>
        /// <para></para>
        /// </param>
        public LinkDefinition(Link source, Link linker, Link target)
        {
            Source = source;
            Linker = linker;
            Target = target;
        }

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="LinkDefinition"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="link">
        /// <para>A link.</para>
        /// <para></para>
        /// </param>
        public LinkDefinition(Link link) : this(link.Source, link.Linker, link.Target) { }

        /// <summary>
        /// <para>
        /// Determines whether this instance equals.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="other">
        /// <para>The other.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        public bool Equals(LinkDefinition other) => Source == other.Source && Linker ==
        ↪  other.Linker && Target == other.Target;
    }

```

```
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
            ↪  linkAfterUpdate);
124         public static event UpdatedDelegate UpdatedEvent = (linkBeforeUpdate, linkAfterUpdate)
            ↪  => { };
125
126         /// <summary>
127         /// <para>
128         /// The deleted delegate.
129         /// </para>
130         /// <para></para>
131         /// </summary>
132         public delegate void DeletedDelegate(LinkDefinition deletedLink);
133         public static event DeletedDelegate DeletedEvent = deletedLink => { };
134
135         #region Low Level
136
137         #region Basic Operations
138
139         /// <summary>
140         /// <para>
141         /// Gets the source index using the specified link.
142         /// </para>
143         /// <para></para>
144         /// </summary>
145         /// <param name="link">
146         /// <para>The link.</para>
147         /// <para></para>
148         /// </param>
149         /// <returns>
150         /// <para>The link index</para>
151         /// <para></para>
152         /// </returns>
153         [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
154         private static extern LinkIndex GetSourceIndex(LinkIndex link);
155
156         /// <summary>
157         /// <para>
158         /// Gets the linker index using the specified link.
159         /// </para>
160         /// <para></para>
161         /// </summary>
162         /// <param name="link">
163         /// <para>The link.</para>
164         /// <para></para>
165         /// </param>
166         /// <returns>
167         /// <para>The link index</para>
168         /// <para></para>
169         /// </returns>
170         [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
171         private static extern LinkIndex GetLinkerIndex(LinkIndex link);
172
```

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

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

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

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

        /// <summary>
        /// <para>
        /// Gets the time using the specified link.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="link">
        /// <para>The link.</para>
        /// <para></para>
        /// </param>
```

```
        /// <returns>
        /// <para>The int</para>
        /// <para></para>
        /// </returns>
        [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
        private static extern Int GetTime(LinkIndex link);

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

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

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

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

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

```
        /// </param>
        /// <returns>
        /// <para>The link index</para>
        /// <para></para>
        /// </returns>
        [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
        private static extern LinkIndex GetLinkNumberOfReferersByTarget(LinkIndex link);

        #endregion

        #region Referers Walkers
        private delegate void Visitor(LinkIndex link);
        private delegate Int StopableVisitor(LinkIndex link);

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

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

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

        /// <summary>
        /// <para>
        /// Walks the through referers by linker using the specified root.
        /// </para>
        /// <para></para>
        /// </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>
```

```
        /// <para>
        /// Gets the target value.
        /// </para>
        /// <para></para>
        /// </summary>
        [DebuggerBrowsable(DebuggerBrowsableState.Never)]
        public Link Target => GetTargetIndex(_link);

        /// <summary>
        /// <para>
        /// Gets the first referer by source value.
        /// </para>
        /// <para></para>
        /// </summary>
        [DebuggerBrowsable(DebuggerBrowsableState.Never)]
        public Link FirstRefererBySource => GetFirstRefererBySourceIndex(_link);

        /// <summary>
        /// <para>
        /// Gets the first referer by linker value.
        /// </para>
        /// <para></para>
        /// </summary>
        [DebuggerBrowsable(DebuggerBrowsableState.Never)]
        public Link FirstRefererByLinker => GetFirstRefererByLinkerIndex(_link);

        /// <summary>
        /// <para>
        /// Gets the first referer by target value.
        /// </para>
        /// <para></para>
        /// </summary>
        [DebuggerBrowsable(DebuggerBrowsableState.Never)]
        public Link FirstRefererByTarget => GetFirstRefererByTargetIndex(_link);

        /// <summary>
        /// <para>
        /// Gets the referers by source count value.
        /// </para>
        /// <para></para>
        /// </summary>
        [DebuggerBrowsable(DebuggerBrowsableState.Never)]
        public Int ReferersBySourceCount => (Int)GetLinkNumberOfReferersBySource(_link);

        /// <summary>
        /// <para>
        /// Gets the referers by linker count value.
        /// </para>
        /// <para></para>
        /// </summary>
        [DebuggerBrowsable(DebuggerBrowsableState.Never)]
        public Int ReferersByLinkerCount => (Int)GetLinkNumberOfReferersByLinker(_link);

        /// <summary>
        /// <para>
        /// Gets the referers by target count value.
        /// </para>
        /// <para></para>
        /// </summary>
        [DebuggerBrowsable(DebuggerBrowsableState.Never)]
        public Int ReferersByTargetCount => (Int)GetLinkNumberOfReferersByTarget(_link);

        /// <summary>
        /// <para>
        /// Gets the total referers value.
        /// </para>
        /// <para></para>
        /// </summary>
        [DebuggerBrowsable(DebuggerBrowsableState.Never)]
        public Int TotalReferers => (Int)GetLinkNumberOfRefererBySource(_link) +
        ↪   (Int)GetLinkNumberOfRefererByLinker(_link) +
        ↪   (Int)GetLinkNumberOfRefererByTarget(_link);

        /// <summary>
        /// <para>
        /// Gets the timestamp value.
        /// </para>
        /// <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})
                            ↪  хранилища не удалось открыть.");
830                     }
831                     Interlocked.Exchange(ref _memoryManagerIsReady, 1);
832                 }
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
            ↪  открывался вовсе.</para>
844         /// <para></para>
845         /// </exception>
846         public static void StopMemoryManager()
847         {
848             lock (_lockObject)
849             {
850                 if (_memoryManagerIsReady != default)
851                 {
852                     if (CloseLinks() == 0)
853                     {
854                         throw new InvalidOperationException("Файл хранилища не удалось закрыть.
                            ↪  Возможно он был уже закрыт, или не открывался вовсе.");
855                     }
856                     Interlocked.Exchange(ref _memoryManagerIsReady, 0);
857                 }
858             }
859         }
860
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

        /// <summary>
        /// <para>
        /// Gets the hash code.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <returns>
        /// <para>The int</para>
        /// <para></para>
        /// </returns>
        public override int GetHashCode() => base.GetHashCode();
        private static bool LinkDoesNotExist(LinkIndex link) => link == 0 ||
        ↪  GetLinkerIndex(link) == 0;
        private static bool LinkWasDeleted(LinkIndex link) => link != 0 && GetLinkerIndex(link)
        ↪  == 0;
        private bool IsMatchingTo(Link source, Link linker, Link target)
            => ((Source == this && source == null) || (Source == source))
            && ((Linker == this && linker == null) || (Linker == linker))
            && ((Target == this && target == null) || (Target == target));
```

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

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

        #endregion

        #region Basic Operations

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

```
            {
                throw new ArgumentException("Удалённая связь не может использоваться в качестве
                ↪  значения.", nameof(source));
            }
            if (LinkWasDeleted(linker))
            {
                throw new ArgumentException("Удалённая связь не может использоваться в качестве
                ↪  значения.", nameof(linker));
            }
            if (LinkWasDeleted(target))
            {
                throw new ArgumentException("Удалённая связь не может использоваться в качестве
                ↪  значения.", nameof(target));
            }
            Link link = CreateLink(source, linker, target);
            if (link == null)
            {
                throw new InvalidOperationException("Невозможно создать связь.");
            }
            CreatedEvent.Invoke(new LinkDefinition(link));
            return link;
        }

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

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

```
            }
            try
            {
                Link link = index;
                if (LinkDoesNotExist(link))
                {
                    throw new InvalidOperationException("Связь с указанным адресом удалена, либо не существовала.");
                }
                return link;
            }
            catch (Exception ex)
            {
                throw new InvalidOperationException("Указатель не является корректным.", ex);
            }
        }

        /// <summary>
        /// <para>
        /// Creates the mapped using the specified source.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="source">
        /// <para>The source.</para>
        /// <para></para>
        /// </param>
        /// <param name="linker">
        /// <para>The linker.</para>
        /// <para></para>
        /// </param>
        /// <param name="target">
        /// <para>The target.</para>
        /// <para></para>
        /// </param>
        /// <param name="mappingIndex">
        /// <para>The mapping index.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        public static Link CreateMapped(Link source, Link linker, Link target, object mappingIndex) => CreateMapped(source, linker, target, Convert.ToInt64(mappingIndex));

        /// <summary>
        /// <para>
        /// Creates the mapped using the specified source.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="source">
        /// <para>The source.</para>
        /// <para></para>
        /// </param>
        /// <param name="linker">
        /// <para>The linker.</para>
        /// <para></para>
        /// </param>
        /// <param name="target">
        /// <para>The target.</para>
        /// <para></para>
        /// </param>
        /// <param name="mappingIndex">
        /// <para>The mapping index.</para>
        /// <para></para>
        /// </param>
        /// <exception cref="InvalidOperationException">
        /// <para>Менеджер памяти ещё не готов.</para>
        /// <para></para>
        /// </exception>
        /// <exception cref="InvalidOperationException">
        /// <para>Существующая привязанная связь не соответствует указанным Source, Linker и Target.</para>
        /// <para></para>
        /// </exception>
        /// <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
                            ↪ удалось.");
1183                    }
1184                }
1185                else if (!mappedLink.IsMatchingTo(source, linker, target))
1186                {
1187                    throw new InvalidOperationException("Существующая привязанная связь не
                        ↪ соответствует указанным Source, Linker и Target.");
1188                }
1189                _linkToMappingIndex[mappedLink] = mappingIndex;
1190                return mappedLink;
1191            }
1192
1193            /// <summary>
1194            /// <para>
1195            /// Determines whether try set mapped.
1196            /// </para>
1197            /// <para></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
1236            /// <summary>
1237            /// <para>
```

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

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

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

        /// <summary>
        /// <para>
        /// Gets the mapped or default using the specified mapping index.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="mappingIndex">
        /// <para>The mapping index.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The mapped link.</para>
        /// <para></para>
        /// </returns>
        public static Link GetMappedOrDefault(Int mappingIndex)
```

```
        {
            TryGetMapped(mappingIndex, out Link mappedLink);
            return mappedLink;
        }

        /// <summary>
        /// <para>
        /// Determines whether try get mapped.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="mappingIndex">
        /// <para>The mapping index.</para>
        /// <para></para>
        /// </param>
        /// <param name="mappedLink">
        /// <para>The mapped link.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        public static bool TryGetMapped(object mappingIndex, out Link mappedLink) =>
        ↪  TryGetMapped(Convert.ToInt64(mappingIndex), out mappedLink);

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

        /// <summary>
        /// <para>
        /// Updates the link.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="link">
        /// <para>The link.</para>
        /// <para></para>
        /// </param>
        /// <param name="newSource">
        /// <para>The new source.</para>
        /// <para></para>
        /// </param>
        /// <param name="newLinker">
        /// <para>The new linker.</para>
```

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

        /// <summary>
        /// <para>
        /// Updates the link.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="link">
        /// <para>The link.</para>
        /// <para></para>
        /// </param>
        /// <param name="newSource">
        /// <para>The new source.</para>
        /// <para></para>
        /// </param>
        /// <param name="newLinker">
        /// <para>The new linker.</para>
        /// <para></para>
        /// </param>
        /// <param name="newTarget">
        /// <para>The new target.</para>
        /// <para></para>
        /// </param>
        /// <exception cref="InvalidOperationException">
        /// <para>Менеджер памяти ещё не готов.</para>
        /// <para></para>
        /// </exception>
        /// <exception cref="ArgumentException">
        /// <para>Нельзя обновить несуществующую связь. </para>
        /// <para></para>
        /// </exception>
        /// <exception cref="ArgumentException">
        /// <para>Удалённая связь не может использоваться в качестве нового значения. </para>
        /// <para></para>
        /// </exception>
        /// <exception cref="ArgumentException">
        /// <para>Удалённая связь не может использоваться в качестве нового значения. </para>
        /// <para></para>
        /// </exception>
        /// <exception cref="ArgumentException">
        /// <para>Удалённая связь не может использоваться в качестве нового значения. </para>
        /// <para></para>
        /// </exception>
        public static void Update(ref Link link, Link newSource, Link newLinker, Link newTarget)
        {
            if (_memoryManagerIsReady == default)
            {
                throw new InvalidOperationException("Менеджер памяти ещё не готов.");
            }
            if (LinkDoesNotExist(link))
            {
                throw new ArgumentException("Нельзя обновить несуществующую связь.",
                ↪  nameof(link));
            }
            if (LinkWasDeleted(newSource))
            {
                throw new ArgumentException("Удалённая связь не может использоваться в качестве
                ↪  нового значения.", nameof(newSource));
            }
            if (LinkWasDeleted(newLinker))
            {
                throw new ArgumentException("Удалённая связь не может использоваться в качестве
                ↪  нового значения.", nameof(newLinker));
```

```
            }
            if (LinkWasDeleted(newTarget))
            {
                throw new ArgumentException("Удалённая связь не может использоваться в качестве
                ↪  нового значения.", nameof(newTarget));
            }
            LinkIndex previousLinkIndex = link;
            _linkToMappingIndex.TryGetValue(link, out long mappingIndex);
            var previousDefinition = new LinkDefinition(link);
            link = UpdateLink(link, newSource, newLinker, newTarget);
            if (mappingIndex >= 0 && previousLinkIndex != link)
            {
                _linkToMappingIndex.Remove(previousLinkIndex);
                SetMappedLink(mappingIndex, link);
                _linkToMappingIndex.Add(link, mappingIndex);
            }
            UpdatedEvent(previousDefinition, new LinkDefinition(link));
        }

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

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

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

        /// <summary>
        /// <para>
        /// Searches the source.
        /// </para>
        /// <para></para>
```

```
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
                        ↪  существующим связям.");
1575                }
1576                return SearchLink(source, linker, target);
1577            }
1578
1579            /// <summary>
1580            /// <para>
1581            /// Determines whether exists.
1582            /// </para>
1583            /// <para></para>
1584            /// </summary>
1585            /// <param name="source">
1586            /// <para>The source.</para>
1587            /// <para></para>
1588            /// </param>
1589            /// <param name="linker">
1590            /// <para>The linker.</para>
1591            /// <para></para>
1592            /// </param>
1593            /// <param name="target">
1594            /// <para>The target.</para>
1595            /// <para></para>
1596            /// </param>
1597            /// <returns>
1598            /// <para>The bool</para>
1599            /// <para></para>
1600            /// </returns>
1601            public static bool Exists(Link source, Link linker, Link target) => SearchLink(source,
                ↪  linker, target) != 0;
1602
1603            #endregion
1604
1605            #region Referers Walkers
1606
1607            /// <summary>
1608            /// <para>
1609            /// Determines whether this instance walk through referers as source.
1610            /// </para>
1611            /// <para></para>
1612            /// </summary>
1613            /// <param name="walker">
1614            /// <para>The walker.</para>
1615            /// <para></para>
1616            /// </param>
```

```
        /// <exception cref="InvalidOperationException">
        /// <para>С несуществующей связью нельзя производитить операции.</para>
        /// <para></para>
        /// </exception>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        public bool WalkThroughReferersAsSource(Func<Link, bool> walker)
        {
            if (LinkDoesNotExist(this))
            {
                throw new InvalidOperationException("С несуществующей связью нельзя
                ↪  производитить операции.");
            }
            var referers = ReferersBySourceCount;
            if (referers == 1)
            {
                return walker(FirstRefererBySource);
            }
            else if (referers > 1)
            {
                return WalkThroughReferersBySource(this, x => walker(x) ? 1 : 0) != 0;
            }
            else
            {
                return true;
            }
        }

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

        /// <summary>
        /// <para>
        /// Determines whether this instance walk through referers as linker.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="walker">
        /// <para>The walker.</para>
        /// <para></para>
        /// </param>
        /// <exception cref="InvalidOperationException">
        /// <para>С несуществующей связью нельзя производитить операции.</para>
        /// <para></para>
        /// </exception>
        /// <returns>
        /// <para>The bool</para>
```

```
        /// <para></para>
        /// </returns>
        public bool WalkThroughReferersAsLinker(Func<Link, bool> walker)
        {
            if (LinkDoesNotExist(this))
            {
                throw new InvalidOperationException("С несуществующей связью нельзя
                ↪  производитить операции.");
            }
            var referers = ReferersByLinkerCount;
            if (referers == 1)
            {
                return walker(FirstRefererByLinker);
            }
            else if (referers > 1)
            {
                return WalkThroughReferersByLinker(this, x => walker(x) ? 1 : 0) != 0;
            }
            else
            {
                return true;
            }
        }

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

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

```
1769                throw new InvalidOperationException("С несуществующей связью нельзя
                    ↪  производитить операции.");
1770            }
1771            var referers = ReferersByTargetCount;
1772            if (referers == 1)
1773            {
1774                return walker(FirstRefererByTarget);
1775            }
1776            else if (referers > 1)
1777            {
1778                return WalkThroughReferersByTarget(this, x => walker(x) ? 1 : 0) != 0;
1779            }
1780            else
1781            {
1782                return true;
1783            }
1784        }
1785
1786        /// <summary>
1787        /// <para>
1788        /// Walks the through referers as target using the specified walker.
1789        /// </para>
1790        /// <para></para>
1791        /// </summary>
1792        /// <param name="walker">
1793        /// <para>The walker.</para>
1794        /// <para></para>
1795        /// </param>
1796        /// <exception cref="InvalidOperationException">
1797        /// <para>С несуществующей связью нельзя производитить операции.</para>
1798        /// <para></para>
1799        /// </exception>
1800        public void WalkThroughReferersAsTarget(Action<Link> walker)
1801        {
1802            if (LinkDoesNotExist(this))
1803            {
1804                throw new InvalidOperationException("С несуществующей связью нельзя
                    ↪  производитить операции.");
1805            }
1806            var referers = ReferersByTargetCount;
1807            if (referers == 1)
1808            {
1809                walker(FirstRefererByTarget);
1810            }
1811            else if (referers > 1)
1812            {
1813                WalkThroughAllReferersByTarget(this, x => walker(x));
1814            }
1815        }
1816
1817        /// <summary>
1818        /// <para>
1819        /// Walks the through referers using the specified walker.
1820        /// </para>
1821        /// <para></para>
1822        /// </summary>
1823        /// <param name="walker">
1824        /// <para>The walker.</para>
1825        /// <para></para>
1826        /// </param>
1827        /// <exception cref="InvalidOperationException">
1828        /// <para>С несуществующей связью нельзя производитить операции.</para>
1829        /// <para></para>
1830        /// </exception>
1831        public void WalkThroughReferers(Action<Link> walker)
1832        {
1833            if (LinkDoesNotExist(this))
1834            {
1835                throw new InvalidOperationException("С несуществующей связью нельзя
                    ↪  производитить операции.");
1836            }
1837            void wrapper(ulong x) => walker(x);
1838            WalkThroughAllReferersBySource(this, wrapper);
1839            WalkThroughAllReferersByLinker(this, wrapper);
1840            WalkThroughAllReferersByTarget(this, wrapper);
1841        }
1842
1843        /// <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>С несуществующей связью нельзя производитить 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("С несуществующей связью нельзя
                      ↪  производитить операции.");
1862              }
1863              long wrapper(ulong x) => walker(x) ? 1 : 0;
1864              WalkThroughReferersBySource(this, wrapper);
1865              WalkThroughReferersByLinker(this, wrapper);
1866              WalkThroughReferersByTarget(this, wrapper);
1867          }
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 =>
              ↪  walker(x) ? 1 : 0) != 0;
1884
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
              ↪  Visitor(x => walker(x)));
1896
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>
```

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

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

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

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

```
        /// <para>The number.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        public static Link FromNumber(long number) => NumberHelpers.FromNumber(number);

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

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

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

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

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

```
        /// </summary>
        /// <param name="chars">
        /// <para>The chars.</para>
        /// <para></para>
        /// </param>
        /// <param name="takeFrom">
        /// <para>The take from.</para>
        /// <para></para>
        /// </param>
        /// <param name="takeUntil">
        /// <para>The take until.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        public static Link FromChars(char[] chars, int takeFrom, int takeUntil) =>
        ↪  FromObjectsToSequence(chars, takeFrom, takeUntil, FromChar);

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

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

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

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

```
        /// <para></para>
        /// </summary>
        /// <typeparam name="T">
        /// <para>The .</para>
        /// <para></para>
        /// </typeparam>
        /// <param name="objects">
        /// <para>The objects.</para>
        /// <para></para>
        /// </param>
        /// <param name="converter">
        /// <para>The converter.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        public static Link FromObjectsToSequence<T>(T[] objects, Func<T, Link> converter) =>
        ↪ FromObjectsToSequence(objects, 0, objects.Length, converter);

        /// <summary>
        /// <para>
        /// Creates the objects to sequence using the specified objects.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <typeparam name="T">
        /// <para>The .</para>
        /// <para></para>
        /// </typeparam>
        /// <param name="objects">
        /// <para>The objects.</para>
        /// <para></para>
        /// </param>
        /// <param name="takeFrom">
        /// <para>The take from.</para>
        /// <para></para>
        /// </param>
        /// <param name="takeUntil">
        /// <para>The take until.</para>
        /// <para></para>
        /// </param>
        /// <param name="converter">
        /// <para>The converter.</para>
        /// <para></para>
        /// </param>
        /// <exception cref="ArgumentOutOfRangeException">
        /// <para>Нельзя преобразовать пустой список к связям.</para>
        /// <para></para>
        /// </exception>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        public static Link FromObjectsToSequence<T>(T[] objects, int takeFrom, int takeUntil,
        ↪ Func<T, Link> converter)
        {
            var length = takeUntil - takeFrom;
            if (length <= 0)
            {
                throw new ArgumentOutOfRangeException(nameof(takeUntil), "Нельзя преобразовать
                ↪ пустой список к связям.");
            }
            var copy = new Link[length];
            for (int i = takeFrom, j = 0; i < takeUntil; i++, j++)
            {
                copy[j] = converter(objects[i]);
            }
            return FromList(copy);
        }

        /// <summary>
        /// <para>
        /// Creates the chars using the specified str.
        /// </para>
        /// <para></para>
        /// </summary>
```

```
        /// <param name="str">
        /// <para>The str.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        public static Link FromChars(string str)
        {
            var copy = new Link[str.Length];
            for (var i = 0; i < copy.Length; i++)
            {
                copy[i] = FromChar(str[i]);
            }
            return FromList(copy);
        }

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

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

        /// <summary>
        /// <para>
        /// Returns the string using the specified char links.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="charLinks">
```

```
        /// <para>The char links.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The string</para>
        /// <para></para>
        /// </returns>
        public static string ToString(List<Link> charLinks)
        {
            var chars = new char[charLinks.Count];
            for (var i = 0; i < charLinks.Count; i++)
            {
                chars[i] = ToChar(charLinks[i]);
            }
            return new string(chars);
        }
    }
}
```

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

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

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

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

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

```
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
99                      if (link.Source.TryGetName(out string sourceName) &&
                        ↪  link.Linker.TryGetName(out string linkerName) &&
                        ↪  link.Target.TryGetName(out string targetName))
100                     {
101                         var sb = new StringBuilder();
102                         sb.Append(sourceName).Append(' ').Append(linkerName).Append('
                            ↪  ').Append(targetName);
103                         str = sb.ToString();
104                         return true;
105                     }
106                 }
107                 str = null;
108                 return false;
109             }
110             finally
111             {
112                 _linksWithNamesGatheringProcess.Remove(link);
113             }
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>
```

```
        /// </returns>
        public static bool TryGetSpecificName(this Link link, out string name)
        {
            string nameLocal = null;
            if (Net.Name.ReferersBySourceCount < link.ReferersBySourceCount)
            {
                Net.Name.WalkThroughReferersAsSource(referer =>
                {
                    if (referer.Linker == Net.ThatIsRepresentedBy)
                    {
                        if (Link.Exists(link, Net.Has, referer))
                        {
                            nameLocal = LinkConverter.ToString(referer.Target);
                            return false; // Останавливаем проход
                        }
                    }
                    return true;
                });
            }
            else
            {
                link.WalkThroughReferersAsSource(referer =>
                {
                    if (referer.Linker == Net.Has)
                    {
                        var nameLink = referer.Target;
                        if (nameLink.Source == Net.Name && nameLink.Linker ==
                        ↪  Net.ThatIsRepresentedBy)
                        {
                            nameLocal = LinkConverter.ToString(nameLink.Target);
                            return false; // Останавливаем проход
                        }
                    }
                    return true;
                });
            }

            name = nameLocal;
            return nameLocal != null;
        }

        // Проверка на пренадлежность классу
        /// <summary>
        /// <para>
        /// Determines whether is.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="link">
        /// <para>The link.</para>
        /// <para></para>
        /// </param>
        /// <param name="@class">
        /// <para>The class.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        public static bool Is(this Link link, Link @class)
        {
            if (link.Linker == Net.IsA)
            {
                if (link.Target == @class)
                {
                    return true;
                }
                else
                {
                    return link.Target.Is(@class);
                }
            }
            return false;
        }

        // Несколько не правильное определение, так выйдет, что любая сумма входящая в диапазон
        ↪  значений char будет символом.
        // 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[] GetArrayOfRererersByTarget(this Link link)
363         {
364             if (link == null)
365             {
366                 return new Link[0];
367             }
368             else
369             {
370                 var array = new Link[link.RefererersByTargetCount];
371                 var index = 0;
372                 link.WalkThroughRefererersAsTarget(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 
```

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

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

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

```
        /// <summary>
        /// <para>
        /// The name net mapping.
        /// </para>
        /// <para></para>
        /// </summary>
        Name,

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

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

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

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

```
        /// <summary>
        /// <para>
        /// The code net mapping.
        /// </para>
        /// <para></para>
        /// </summary>
        Code
    }

    /// <summary>
    /// <para>
    /// Represents the net.
    /// </para>
    /// <para></para>
    /// </summary>
    public static class Net
    {
        /// <summary>
        /// <para>
        /// Gets or sets the link value.
        /// </para>
        /// <para></para>
        /// </summary>
        public static Link Link { get; private set; }
        /// <summary>
        /// <para>
        /// Gets or sets the thing value.
        /// </para>
        /// <para></para>
        /// </summary>
        public static Link Thing { get; private set; }
        /// <summary>
        /// <para>
        /// Gets or sets the is a value.
        /// </para>
        /// <para></para>
        /// </summary>
        public static Link IsA { get; private set; }
        /// <summary>
        /// <para>
        /// Gets or sets the is not a value.
        /// </para>
        /// <para></para>
        /// </summary>
        public static Link IsNotA { get; private set; }

        /// <summary>
        /// <para>
        /// Gets or sets the of value.
        /// </para>
        /// <para></para>
        /// </summary>
        public static Link Of { get; private set; }
        /// <summary>
        /// <para>
        /// Gets or sets the and value.
        /// </para>
        /// <para></para>
        /// </summary>
        public static Link And { get; private set; }
        /// <summary>
        /// <para>
        /// Gets or sets the that consists of value.
        /// </para>
        /// <para></para>
        /// </summary>
        public static Link ThatConsistsOf { get; private set; }
        /// <summary>
        /// <para>
        /// Gets or sets the has value.
        /// </para>
        /// <para></para>
        /// </summary>
        public static Link Has { get; private set; }
        /// <summary>
        /// <para>
        /// Gets or sets the contains value.
        /// </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>
```

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

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

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

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

```
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);
                    ↪  // Стоит переделать в "[x] is a member|instance|element of the class [y]"
658                 IsNotA = Link.CreateMapped(Link.Itself, Link.Itself, IsA, NetMapping.IsNotA);
659                 Link = Link.CreateMapped(Link.Itself, IsA, Link.Itself, NetMapping.Link);
660                 Thing = Link.CreateMapped(Link.Itself, IsNotA, Link, NetMapping.Thing);
661
662                 IsA = Link.Update(IsA, IsA, IsA, Link); // Исключение, позволяющие завершить
                    ↪  систему
663             }
664
665             #endregion
```

```
            Of = CreateMappedLink(NetMapping.Of);
            And = CreateMappedLink(NetMapping.And);
            ThatConsistsOf = CreateMappedLink(NetMapping.ThatConsistsOf);
            Has = CreateMappedLink(NetMapping.Has);
            Contains = CreateMappedLink(NetMapping.Contains);
            ContainedBy = CreateMappedLink(NetMapping.ContainedBy);

            One = CreateMappedThing(NetMapping.One);
            Zero = CreateMappedThing(NetMapping.Zero);

            Sum = CreateMappedThing(NetMapping.Sum);
            Character = CreateMappedThing(NetMapping.Character);
            String = CreateMappedThing(NetMapping.String);
            Name = Link.CreateMapped(Link.Itself, IsA, String, NetMapping.Name);

            Set = CreateMappedThing(NetMapping.Set);
            Group = CreateMappedThing(NetMapping.Group);

            ParsedFrom = CreateMappedLink(NetMapping.ParsedFrom);
            ThatIs = CreateMappedLink(NetMapping.ThatIs);
            ThatIsBefore = CreateMappedLink(NetMapping.ThatIsBefore);
            ThatIsAfter = CreateMappedLink(NetMapping.ThatIsAfter);
            ThatIsBetween = CreateMappedLink(NetMapping.ThatIsBetween);
            ThatIsRepresentedBy = CreateMappedLink(NetMapping.ThatIsRepresentedBy);
            ThatHas = CreateMappedLink(NetMapping.ThatHas);

            Text = CreateMappedThing(NetMapping.Text);
            Path = CreateMappedThing(NetMapping.Path);
            Content = CreateMappedThing(NetMapping.Content);
            Empty = CreateMappedThing(NetMapping.Empty);
            EmptyContent = Link.CreateMapped(Content, ThatIs, Empty, NetMapping.EmptyContent);
            Alphabet = CreateMappedThing(NetMapping.Alphabet);
            Letter = Link.CreateMapped(Link.Itself, IsA, Character, NetMapping.Letter);
            Case = CreateMappedThing(NetMapping.Case);
            Upper = CreateMappedThing(NetMapping.Upper);
            UpperCase = Link.CreateMapped(Case, ThatIs, Upper, NetMapping.UpperCase);
            Lower = CreateMappedThing(NetMapping.Lower);
            LowerCase = Link.CreateMapped(Case, ThatIs, Lower, NetMapping.LowerCase);
            Code = CreateMappedThing(NetMapping.Code);

            SetNames();
        }

        /// <summary>
        /// <para>
        /// Recreates.
        /// </para>
        /// <para></para>
        /// </summary>
        public static void Recreate()
        {
            ThreadHelpers.InvokeWithExtendedMaxStackSize(() => Link.Delete(IsA));
            CharacterHelpers.Recreate();
            Create();
        }
        private static void SetNames()
        {
            Thing.SetName("thing");
            Link.SetName("link");
            IsA.SetName("is a");
            IsNotA.SetName("is not a");

            Of.SetName("of");
            And.SetName("and");
            ThatConsistsOf.SetName("that consists of");
            Has.SetName("has");
            Contains.SetName("contains");
            ContainedBy.SetName("contained by");

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

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

            Set.SetName("set");
```

```
            Group.SetName("group");

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

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

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

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

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

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

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="NumberHelpers"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        static NumberHelpers() => Create();
        private static void Create()
        {
            NumbersToLinks = new Link[64];
            LinksToNumbers = new Dictionary<Link, long>();
            NumbersToLinks[0] = Net.One;
            LinksToNumbers[Net.One] = 1;
        }

        /// <summary>
        /// <para>
        /// Recreates.
        /// </para>
        /// <para></para>
        /// </summary>
```

```
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 &
                                ↪  previousPowerOf2Link);
69                              var num = (long)System.Math.Pow(2, i);
70                              NumbersToLinks[i] = numberLink;
71                              LinksToNumbers[numberLink] = num;
72                              numberLink.SetName(num.ToString(CultureInfo.InvariantCulture));
73                          }
74                          previousPowerOf2Link = NumbersToLinks[i];
75                      }
76                  }
77                  result = Link.Create(Net.Sum, Net.Of, previousPowerOf2Link &
                    ↪  previousPowerOf2Link);
78                  var number = (long)System.Math.Pow(2, powerOf2);
79                  NumbersToLinks[powerOf2] = result;
80                  LinksToNumbers[result] = number;
81                  result.SetName(number.ToString(CultureInfo.InvariantCulture));
82              }
83              return result;
84          }
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             }
```

```
            var sum = Link.Create(Net.Sum, Net.Of, LinkConverter.FromList(links));
            return sum;
        }

        /// <summary>
        /// <para>
        /// Returns the number using the specified link.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="link">
        /// <para>The link.</para>
        /// <para></para>
        /// </param>
        /// <exception cref="ArgumentOutOfRangeException">
        /// <para>Specified link is not a number.</para>
        /// <para></para>
        /// </exception>
        /// <returns>
        /// <para>The long</para>
        /// <para></para>
        /// </returns>
        public static long ToNumber(Link link)
        {
            if (link == Net.Zero)
            {
                return 0;
            }
            if (link == Net.One)
            {
                return 1;
            }
            if (link.IsSum())
            {
                var numberParts = LinkConverter.ToList(link.Target);
                long number = 0;
                for (var i = 0; i < numberParts.Count; i++)
                {
                    GoDownAndTakeIt(numberParts[i], out long numberPart);
                    number += numberPart;
                }
                return number;
            }
            throw new ArgumentOutOfRangeException(nameof(link), "Specified link is not a
            ↪  number.");
        }
        private static void GoDownAndTakeIt(Link link, out long number)
        {
            if (!LinksToNumbers.TryGetValue(link, out number))
            {
                var previousNumberLink = link.Target.Source;
                GoDownAndTakeIt(previousNumberLink, out number);
                var previousNumberIndex = (int)System.Math.Log(number, 2);
                var newNumberIndex = previousNumberIndex + 1;
                var newNumberLink = Link.Create(Net.Sum, Net.Of, previousNumberLink &
                ↪  previousNumberLink);
                number += number;
                NumbersToLinks[newNumberIndex] = newNumberLink;
                LinksToNumbers[newNumberLink] = number;
            }
        }
    }
}
```

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

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

namespace Platform.Data.Triplets.Sequences
{
    /// <summary>
    /// <para>
    /// Represents the compression experiments.
    /// </para>
    /// <para></para>
    /// </summary>
    internal static class CompressionExperiments
    {
        /// <summary>
```

```
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 &&
                ↪  subject.ReferersByTargetCount == 0)
31              {
32                  var subJoint = Link.Search(subject.Target, Net.And, @object);
33                  if (subJoint != null && subJoint != subject)
34                  {
35                      Link.Update(ref subject, subject.Source, Net.And, subJoint);
36                      return;
37                  }
38              }
39              subject = Link.Create(subject, Net.And, @object);
40          }
41
42          //public static Link RightJoinUnsafe(Link subject, Link @object)
43          //{
44          //    if (subject.Linker == Net.And && subject.ReferersBySourceCount == 0 &&
                ↪  subject.ReferersByTargetCount == 0)
45          //    {
46          //        Link subJoint = Link.Search(subject.Target, Net.And, @object);
47          //        if (subJoint != null && subJoint != subject)
48          //        {
49          //            Link.Update(ref subject, subject.Source, Net.And, subJoint);
50          //            return subject;
51          //        }
52          //    }
53          //    return Link.Create(subject, Net.And, @object);
54          //}
55
56          ////public static void LeftJoin(ref Link subject, Link @object)
57          ////{
58          ////    if (subject.Linker == Net.And && subject.ReferersBySourceCount == 0 &&
                ↪  subject.ReferersByTargetCount == 0)
59          ////    {
60          ////        Link subJoint = Link.Search(@object, Net.And, subject.Source);
61          ////        if (subJoint != null && subJoint != subject)
62          ////        {
63          ////            Link.Update(ref subject, subJoint, Net.And, subject.Target);
64          ////            return;
65          ////        }
66          ////    }
67          ////    subject = Link.Create(@object, Net.And, subject);
68          ////}
69
70          /// <summary>
71          /// <para>
72          /// Lefts the join using the specified subject.
73          /// </para>
74          /// <para></para>
75          /// </summary>
76          /// <param name="subject">
77          /// <para>The subject.</para>
78          /// <para></para>
79          /// </param>
80          /// <param name="@object">
81          /// <para>The object.</para>
82          /// <para></para>
83          /// </param>
84          public static void LeftJoin(ref Link subject, Link @object)
85          {
86              if (subject.Linker == Net.And && subject.ReferersBySourceCount == 0 &&
                ↪  subject.ReferersByTargetCount == 0)
87              {
88                  var subJoint = Link.Search(@object, Net.And, subject.Source);
```

```
89                  if (subJoint != null && subJoint != subject)
90                  {
91                      Link.Update(ref subject, subJoint, Net.And, subject.Target);
92                      //var prev = Link.Search(@object, Net.And, subject);
93                      //if (prev != null)
94                      //{
95                      //    Link.Update(ref prev, subJoint, Net.And, subject.Target);
96                      //}
97                      return;
98                  }
99              }
100             subject = Link.Create(@object, Net.And, subject);
101         }
102
103         // Сначала сжатие налево, а затем направо (так эффективнее)
104         // Не приятный момент, что обе связи, и первая и вторая могут быть изменены в результате
            ↪  алгоритма.
105         //public static Link CombinedJoin(ref Link first, ref Link second)
106         //{
107         //    Link atomicConnection = Link.Search(first, Net.And, second);
108         //    if (atomicConnection != null)
109         //    {
110         //        return atomicConnection;
111         //    }
112         //    else
113         //    {
114         //        if (second.Linker == Net.And)
115         //        {
116         //            Link subJoint = Link.Search(first, Net.And, second.Source);
117         //            if (subJoint != null && subJoint != second)// && subJoint.TotalReferers >
            ↪  second.TotalReferers)
118         //            {
119         //                //if (first.Linker == Net.And)
120         //                //{
121         //                //    // TODO: ...
122         //                //}
123         //                if (second.TotalReferers > 0)
124         //                {
125         //                    // В данный момент это никак не влияет, из-за того что добавлено
            ↪  условие по требованию
126         //                    // использования атомарного соедининеия если оно есть
127
128         //                    // В целом же приоритет между обходным соединением и атомарным
            ↪  нужно определять по весу.
129         //                    // И если в сети обнаружено сразу два варианта прохода - простой и
            ↪  обходной - нужно перебрасывать
130         //                    // пути с меньшим весом на использование путей с большим весом.
            ↪  (Это и технически эффективнее и более оправдано
131         //                    // с точки зрения смысла).
132
133         //                    // Положительный эффект текущей реализации, что она быстро
            ↪  "успокаивается" набирает критическую массу
134         //                    // и перестаёт вести себя не предсказуемо
135
136         //                    // Неприятность учёта веса в том, что нужно обрабатывать большое
            ↪  количество комбинаций.
137         //                    // Но вероятно это оправдано.
138
139         //                    //var prev = Link.Search(first, Net.And, second);
140         //                    //if (prev != null && subJoint != prev) // && prev.TotalReferers <
            ↪  subJoint.TotalReferers)
141         //                    //{
142         //                    //    Link.Update(ref prev, subJoint, Net.And, second.Target);
143         //                    //    if (second.TotalReferers == 0)
144         //                    //    {
145         //                    //        Link.Delete(ref second);
146         //                    //    }
147         //                    //    return prev;
148         //                    //}
149         //                    //return Link.Create(subJoint, Net.And, second.Target);
150         //                }
151         //                else
152         //                {
153         //                    Link.Update(ref second, subJoint, Net.And, second.Target);
154         //                    return second;
155         //                }
156         //            }
157         //        }
```

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

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

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

```
372                         Link.Update(ref prev, first.Source, Net.And, leftCrossConnection);
373                     }
374                 }
375                 if (rightCrossConnection != null)
376                 {
377                     var prev = Link.Search(rightCrossConnection, Net.And, second.Target);
378                     if (prev != null && rightCrossConnection != prev)
379                     {
380                         Link.Update(ref prev, first.Source, Net.And, leftCrossConnection);
381                     }
382                 }
383                 //TryReconstructConnection(first.Target, second);
384                 //TryReconstructConnection(first.Source, leftCrossConnection); // ухудшает
                    ↪ стабильность
385                 var resultConnection = first.Source & leftCrossConnection;
386                 //if (first.TotalReferers == 0)
387                 //    Link.Delete(ref first);
388                 return resultConnection;
389             }
390             else
391             {
392                 if (directConnection != null)
393                 {
394                     return directConnection;
395                 }
396                 if (rightCrossConnection != null)
397                 {
398                     return rightCrossConnection & second.Target;
399                 }
400                 if (leftCrossConnection != null)
401                 {
402                     return first.Source & leftCrossConnection;
403                 }
404             }
405             // Можно фиксировать по окончанию каждой из веток, какой эффект от неё происходит
                ↪ (на сколько уменьшается/увеличивается количество связей)
406             directConnection = first & second;
407             //long difference = Net.And.ReferersByLinkerCount - totalDoublets;
408             //if (difference != 1)
409             //{
410             //    Console.WriteLine(Net.And.ReferersByLinkerCount - totalDoublets);
411             //}
412             return directConnection;
413         }
414         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 &&
                                    ↪  neighbour.ReferersBySourceCount == 0)
462                                 {
463                                     if (directConnection == null)
464                                     {
465                                         directConnection = first & second;
466                                     }
467                                     Link.Update(ref couple, directConnection, Net.And,
                                        ↪  neighbour.Target);
468                                     //Link.Delete(ref neighbour);
469                                 }
470                             }
471                         }
472                     });
473                 }
474
475                 if (second.ReferersByTargetCount < first.ReferersByTargetCount)
476                 {
477                     // |_x      ... x_o
478                     //  |_o      |___|
479                     //
480                     //   <-
481                     second.WalkThroughReferersAsTarget(couple =>
482                     {
483                         if (couple.Linker == Net.And)
484                         {
485                             var neighbour = couple.Source;
486                             if (neighbour.Linker == Net.And && neighbour.Target == first)
487                             {
488                                 if (neighbour.ReferersByTargetCount == 0 &&
                                    ↪  neighbour.ReferersBySourceCount == 1)
489                                 {
490                                     if (directConnection == null)
491                                     {
492                                         directConnection = first & second;
493                                     }
494                                     Link.Update(ref couple, neighbour.Source, Net.And,
                                        ↪  directConnection);
495                                     //Link.Delete(ref neighbour);
496                                 }
497                             }
498                         }
499                     });
500                 }
501                 else
502                 {
503                     // |_x      ... x_o
504                     //  |_o      |___|
505                     //
506                     //   ->
507                     first.WalkThroughReferersAsTarget((couple) =>
508                     {
509                         if (couple.Linker == Net.And && couple.ReferersByTargetCount == 0 &&
                            ↪  couple.ReferersBySourceCount == 1)
510                         {
511                             var neighbour = couple.FirstRefererBySource;
512                             if (neighbour.Linker == Net.And && neighbour.Target == second)
513                             {
514                                 if (directConnection == null)
515                                 {
516                                     directConnection = first & second;
517                                 }
518                                 Link.Update(ref neighbour, couple.Source, Net.And, directConnection);
519                                 Link.Delete(ref couple);
```

```
                    }
                }
            });
        }
        if (directConnection != null)
        {
            CompressionsCount++;
        }
        return directConnection;
    }

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

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

```
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(IList<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(IList<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,
            ↪  links, ref i);
674         //    return element;
675         //}
676         //public static Link FromList(Link[] links)
677         //{
678         //    int i = links.Length - 1;
679         //    Link element = links[i];
680         //    while (--i >= 0) element = LinkConverterOld.ConnectLinks2(links[i], element,
            ↪  links, ref i);
681         //    return element;
682         //}
683         //public static Link FromList(List<Link> links)
684         //{
685         //    Link element = links[0];
686         //    for (int i = 1; i < links.Count; i += ChunkSize)
687         //    {
688         //        int j = (i + ChunkSize - 1);
689         //        j = j < links.Count ? j : (links.Count - 1);
690         //        Link subElement = links[j];
691         //        while (--j >= i) subElement = CombinedJoin(links[j], subElement);
692         //        element = CombinedJoin(element, subElement);
693         //    }
694         //    return element;
695         //}
696         //public static Link FromList(Link[] links)
697         //{
698         //    Link element = links[0];
699         //    for (int i = 1; i < links.Length; i += ChunkSize)
700         //    {
701         //        int j = (i + ChunkSize - 1);
702         //        j = j < links.Length ? j : (links.Length - 1);
703         //        Link subElement = links[j];
704         //        while (--j >= i) subElement = CombinedJoin(links[j], subElement);
705         //        element = CombinedJoin(element, subElement);
706         //    }
707         //    return element;
708         //}
709         //public static Link FromList(IList<Link> links)
710         //{
711         //    int leftBound = 0;
712         //    int rightBound = links.Count - 1;
713         //    if (leftBound == rightBound)
714         //    {
715         //        return links[0];
716         //    }
717         //    Link left = links[leftBound];
718         //    Link right = links[rightBound];
719         //    long leftReferers = left.ReferersBySourceCount + left.ReferersByTargetCount;
720         //    long rightReferers = right.ReferersBySourceCount + right.ReferersByTargetCount;
721         //    while (true)
722         //    {
723         //        //if (rightBound % 2 != leftBound % 2)
724         //        if (rightReferers >= leftReferers)
725         //        {
726         //            int nextRightBound = --rightBound;
727         //            if (nextRightBound == leftBound)
728         //            {
729         //                var x = CombinedJoin(ref left, ref right);
730         //                return x;
731         //            }
732         //            else
733         //            {
734         //                Link nextRight = links[nextRightBound];
735         //                right = CombinedJoin(ref nextRight, ref right);
736         //                rightReferers = right.ReferersBySourceCount +
            ↪  right.ReferersByTargetCount;
737         //            }
738         //        }
739         //        else
740         //        {
741         //            int nextLeftBound = ++leftBound;
742         //            if (nextLeftBound == rightBound)
743         //            {
744         //                return CombinedJoin(ref left, ref right);
745         //            }
746         //            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,
                        ↪  links[i]);
813                 }
814                 return element;
815             }
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
        ↪  links, doublet links and so on.
13      /// </remarks>
14      public static class SequenceHelpers
15      {
16          /// <summary>
17          /// <para>
18          /// The max sequence format size.
19          /// </para>
20          /// <para></para>
21          /// </summary>
22          public static readonly int MaxSequenceFormatSize = 20;
23
24          //public static void DeleteSequence(Link sequence)
25          //{
26          //}
27
28          /// <summary>
29          /// <para>
30          /// Formats the sequence using the specified sequence.
31          /// </para>
32          /// <para></para>
33          /// </summary>
34          /// <param name="sequence">
35          /// <para>The sequence.</para>
36          /// <para></para>
37          /// </param>
38          /// <returns>
39          /// <para>The string</para>
40          /// <para></para>
41          /// </returns>
42          public static string FormatSequence(Link sequence)
43          {
44              var visitedElements = 0;
45              var sb = new StringBuilder();
46              sb.Append('{');
47              StopableSequenceWalker.WalkRight(sequence, x => x.Source, x => x.Target, x =>
                ↪  x.Linker != Net.And, element =>
48              {
49                  if (visitedElements > 0)
50                  {
51                      sb.Append(',');
52                  }
53                  sb.Append(element.ToString());
54                  visitedElements++;
55                  if (visitedElements < MaxSequenceFormatSize)
56                  {
57                      return true;
58                  }
59                  else
60                  {
61                      sb.Append(", ...");
62                      return false;
63                  }
64              });
65              sb.Append('}');
66              return sb.ToString();
67          }
68
69          /// <summary>
70          /// <para>
71          /// Collects the matching sequences using the specified links.
72          /// </para>
73          /// <para></para>
74          /// </summary>
75          /// <param name="links">
76          /// <para>The links.</para>
77          /// <para></para>
78          /// </param>
79          /// <exception cref="InvalidOperationException">
80          /// <para>Подпоследовательности с одним элементом не поддерживаются.</para>
81          /// <para></para>
```

```
        /// </exception>
        /// <returns>
        /// <para>The results.</para>
        /// <para></para>
        /// </returns>
        public static List<Link> CollectMatchingSequences(Link[] links)
        {
            if (links.Length == 1)
            {
                throw new InvalidOperationException("Подпоследовательности с одним элементом не
                ↪  поддерживаются.");
            }
            var leftBound = 0;
            var rightBound = links.Length - 1;
            var left = links[leftBound++];
            var right = links[rightBound--];
            var results = new List<Link>();
            CollectMatchingSequences(left, leftBound, links, right, rightBound, ref results);
            return results;
        }
        private static void CollectMatchingSequences(Link leftLink, int leftBound, Link[]
        ↪  middleLinks, Link rightLink, int rightBound, ref List<Link> results)
        {
            var leftLinkTotalReferers = leftLink.ReferersBySourceCount +
            ↪  leftLink.ReferersByTargetCount;
            var rightLinkTotalReferers = rightLink.ReferersBySourceCount +
            ↪  rightLink.ReferersByTargetCount;
            if (leftLinkTotalReferers <= rightLinkTotalReferers)
            {
                var nextLeftLink = middleLinks[leftBound];
                var elements = GetRightElements(leftLink, nextLeftLink);
                if (leftBound <= rightBound)
                {
                    for (var i = elements.Length - 1; i >= 0; i--)
                    {
                        var element = elements[i];
                        if (element != null)
                        {
                            CollectMatchingSequences(element, leftBound + 1, middleLinks,
                            ↪  rightLink, rightBound, ref results);
                        }
                    }
                }
                else
                {
                    for (var i = elements.Length - 1; i >= 0; i--)
                    {
                        var element = elements[i];
                        if (element != null)
                        {
                            results.Add(element);
                        }
                    }
                }
            }
            else
            {
                var nextRightLink = middleLinks[rightBound];
                var elements = GetLeftElements(rightLink, nextRightLink);
                if (leftBound <= rightBound)
                {
                    for (var i = elements.Length - 1; i >= 0; i--)
                    {
                        var element = elements[i];
                        if (element != null)
                        {
                            CollectMatchingSequences(leftLink, leftBound, middleLinks,
                            ↪  elements[i], rightBound - 1, ref results);
                        }
                    }
                }
                else
                {
                    for (var i = elements.Length - 1; i >= 0; i--)
                    {
                        var element = elements[i];
                        if (element != null)
                        {
```

```
                        results.Add(element);
                    }
                }
            }
        }

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

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

```
                        if (coupleTarget == rightLink)
                        {
                            result[offset] = couple;
                            if (++added == 2)
                            {
                                return Link.Stop;
                            }
                        }
                        else if (coupleTarget.Linker == Net.And && coupleTarget.Source ==
                        ↪  rightLink)
                        {
                            result[offset + 1] = couple;
                            if (++added == 2)
                            {
                                return Link.Stop;
                            }
                        }
                    }
                    return Link.Continue;
                });
            return added > 0;
        }

        /// <summary>
        /// <para>
        /// Gets the left elements using the specified start link.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="startLink">
        /// <para>The start link.</para>
        /// <para></para>
        /// </param>
        /// <param name="leftLink">
        /// <para>The left link.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The result.</para>
        /// <para></para>
        /// </returns>
        public static Link[] GetLeftElements(Link startLink, Link leftLink)
        {
            var result = new Link[4];
            TryStepLeft(startLink, leftLink, result, 0);
            startLink.WalkThroughReferersAsSource(couple =>
                {
                    if (couple.Linker == Net.And)
                    {
                        if (TryStepLeft(couple, leftLink, result, 2))
                        {
                            return Link.Stop;
                        }
                    }
                    return Link.Continue;
                });
            return result;
        }

        /// <summary>
        /// <para>
        /// Determines whether try step left.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="startLink">
        /// <para>The start link.</para>
        /// <para></para>
        /// </param>
        /// <param name="leftLink">
        /// <para>The left link.</para>
        /// <para></para>
        /// </param>
        /// <param name="result">
        /// <para>The result.</para>
        /// <para></para>
        /// </param>
        /// <param name="offset">
```

```
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 ==
                                 ↪ leftLink)
332                             {
333                                 result[offset + 1] = couple;
334                                 if (++added == 2)
335                                 {
336                                     return Link.Stop;
337                                 }
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);
```

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

            Link.Delete(link1);

            Link.StopMemoryManager();

            File.Delete(filename);
        }
    }

    [Fact]
    public static void DeepCreateUpdateDeleteLinkTest()
    {
        lock (Lock)
        {
            string filename = "db.links";

            File.Delete(filename);

            Link.StartMemoryManager(filename);

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

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

            Assert.Equal(isA, isA.Target);

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

            Assert.Equal(link, isA.Target);

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

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

            Link.StopMemoryManager();

            File.Delete(filename);
        }
    }

    [Fact]
    public static void LinkReferersWalkTest()
    {
        lock (Lock)
        {
            string filename = "db.links";

            File.Delete(filename);

            Link.StartMemoryManager(filename);

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

            Assert.Equal(1, thing.ReferersBySourceCount);
            Assert.Equal(2, isA.ReferersByLinkerCount);
            Assert.Equal(3, link.ReferersByTargetCount);

            _thingVisitorCounter = 0;
            _isAVisitorCounter = 0;
            _linkVisitorCounter = 0;

            thing.WalkThroughReferersAsSource(ThingVisitor);
            isA.WalkThroughReferersAsLinker(IsAVisitor);
            link.WalkThroughReferersAsTarget(LinkVisitor);

            Assert.Equal(4UL, _thingVisitorCounter);
            Assert.Equal(1UL + 3UL, _isAVisitorCounter);
            Assert.Equal(1UL + 3UL + 4UL, _linkVisitorCounter);

            Link.StopMemoryManager();

```

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

        [Fact]
        public static void MultipleRandomCreationsAndDeletionsTest()
        {
            lock (Lock)
            {
                string filename = "db.links";

                File.Delete(filename);

                Link.StartMemoryManager(filename);

                TestMultipleRandomCreationsAndDeletions(2000);

                Link.StopMemoryManager();

                File.Delete(filename);
            }
        }
        private static void TestMultipleRandomCreationsAndDeletions(int
        ↪ maximumOperationsPerCycle)
        {
            var and = Link.Create(Link.Itself, Link.Itself, Link.Itself);
            //var comparer = Comparer<TLink>.Default;
            for (var N = 1; N < maximumOperationsPerCycle; N++)
            {
                var random = new System.Random(N);
                var linksCount = 1;
                for (var i = 0; i < N; i++)
                {
                    var createPoint = random.NextBoolean();
                    if (linksCount > 2 && createPoint)
                    {
                        var linksAddressRange = new Range<ulong>(1, (ulong)linksCount);
                        Link source = random.NextUInt64(linksAddressRange);
                        Link target = random.NextUInt64(linksAddressRange); //-V3086
                        var resultLink = Link.Create(source, and, target);
                        if (resultLink > linksCount)
                        {
                            linksCount++;
                        }
                    }
                    else
                    {
                        Link.Create(Link.Itself, Link.Itself, Link.Itself);
                        linksCount++;
                    }
                }
                for (var i = 0; i < N; i++)
                {
                    Link link = i + 2;
                    if (link.Linker != null)
                    {
                        Link.Delete(link);
                        linksCount--;
                    }
                }
            }
        }
    }
}
```

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

```
using System.IO;
using Xunit;

namespace Platform.Data.Triplets.Tests
{
    public static class PersistentMemoryManagerTests
    {
        [Fact]
        public static void FileMappingTest()
        {
            lock (LinkTests.Lock)
            {
                string filename = "db.links";
```

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

```
        {
            lock (LinkTests.Lock)
            {
                string filename = "db.links";

                File.Delete(filename);

                Link.StartMemoryManager(filename);

                var mapped = Link.GetMappedOrDefault(0);

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

                Assert.True(mappingSet);

                Link.StopMemoryManager();

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

Index