./Exceptions/ArgumentLinkDoesNotExistsException.cs

```
using System;

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

namespace Platform.Data.Exceptions
{
    public class ArgumentLinkDoesNotExistsException<TLink> : ArgumentException
    {
        public ArgumentLinkDoesNotExistsException(TLink link, string paramName) : base(FormatMessage(link, paramName), paramName) { }
        public ArgumentLinkDoesNotExistsException(TLink link) : base(FormatMessage(link)) { }
        private static string FormatMessage(TLink link, string paramName) => $"Связь [{link}] переданная в аргумент [{paramName}] не существует.";
        private static string FormatMessage(TLink link) => $"Связь [{link}] переданная в качестве аргумента не существует.";
    }
}
```

./Exceptions/ArgumentLinkHasDependenciesException.cs

```
using System;

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

namespace Platform.Data.Exceptions
{
    public class ArgumentLinkHasDependenciesException<TLink> : ArgumentException
    {
        public ArgumentLinkHasDependenciesException(TLink link, string paramName) : base(FormatMessage(link, paramName), paramName) { }
        public ArgumentLinkHasDependenciesException(TLink link) : base(FormatMessage(link)) { }
        private static string FormatMessage(TLink link, string paramName) => $"У связи [{link}] переданной в аргумент [{paramName}] присутствуют зависимости, которые препятствуют изменению её внутренней структуры.";
        private static string FormatMessage(TLink link) => $"У связи [{link}] переданной в качестве аргумента присутствуют зависимости, которые препятствуют изменению её внутренней структуры.";
    }
}
```

./Exceptions/LinksLimitReachedException.cs

```
using System;

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

namespace Platform.Data.Exceptions
{
    public class LinksLimitReachedException : Exception
    {
        public static readonly string DefaultMessage = "Достигнут лимит количества связей в хранилище.";
        public LinksLimitReachedException(string message) : base(message) { }
        public LinksLimitReachedException(ulong limit) : this(FormatMessage(limit)) { }
        public LinksLimitReachedException() : base(DefaultMessage) { }
        private static string FormatMessage(ulong limit) => $"Достигнут лимит количества связей в хранилище ({limit}).";
    }
}
```

./Exceptions/LinkWithSameValueAlreadyExistsException.cs

```
using System;

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

namespace Platform.Data.Exceptions
{
    public class LinkWithSameValueAlreadyExistsException : Exception
    {
        public static readonly string DefaultMessage = "Связь с таким же значением уже существует.";
        public LinkWithSameValueAlreadyExistsException(string message) : base(message) { }
        public LinkWithSameValueAlreadyExistsException() : base(DefaultMessage) { }
    }
}
```

./ILinks.cs

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

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

namespace Platform.Data
{
    /// <summary>
    /// Представляет интерфейс для работы с данными в формате Links (хранилища взаимосвязей).
    /// </summary>
    /// <remarks>
    /// Этот интерфейс в данный момент не зависит от размера содержимого связи, а значит подходит как для дуплетов, так и для триплетов и т.п.
    /// Возможно этот интерфейс подходит даже для Sequences.
    /// </remarks>
    public interface ILinks<TLink, TConstants>
        where TConstants : LinksConstants<TLink>
    {
        #region Constants

        /// <summary>
        /// Возвращает набор констант, который необходим для эффективной коммуникации с методами этого интерфейса.
        /// Эти константы не меняются с момента создания точки доступа к хранилищу.
        /// </summary>
        TConstants Constants { get; }

        #endregion

        #region Read

        /// <summary>
        /// Подсчитывает и возвращает общее число связей находящихся в хранилище, соответствующих указанным ограничениям.
        /// </summary>
        /// <param name="restriction">Ограничения на содержимое связей.</param>
        /// <returns>Общее число связей находящихся в хранилище, соответствующих указанным ограничениям.</returns>
        TLink Count(IList<TLink> restriction);

        /// <summary>
        /// Выполняет проход по всем связям, соответствующим шаблону, вызывая обработчик (handler) для каждой подходящей связи.
        /// </summary>
        /// <param name="handler">Обработчик каждой подходящей связи.</param>
        /// <param name="restrictions">Ограничения на содержимое связей. Каждое ограничение может иметь значения: Constants.Null - 0-я связь, обозначающая ссылку на пустоту, Any - отсутствие ограничения, 1..∞ конкретный индекс связи.</param>
        /// <returns>True, в случае если проход по связям не был прерван и False в обратном случае.</returns>
        TLink Each(Func<IList<TLink>, TLink> handler, IList<TLink> restrictions);

        // TODO: Move to UniLinksExtensions
        //return Trigger(restrictions, (before, after) => handler(before), null, null);
        //// Trigger(restrictions, null, restrictions, null); - Должно быть синонимом

        #endregion

        #region Write

        /// <summary>
        /// Создаёт связь.
        /// </summary>
        /// <returns>Индекс созданной связи.</returns>
        TLink Create(); // TODO: Возможно всегда нужно принимать restrictions, возможно и возвращать связь нужно целиком.

        // TODO: Move to UniLinksExtensions
        //// { 0, 0, 0 } => { ifself, 0, 0 }
        //// { 0 } => { ifself, 0, 0 } *

        //T result = default(T);

        //Func<IList<T>, IList<T>, T> substitutedHandler = (before, after) =>
        //{
        //    result = after[Constants.IndexPart];
        //    return Constants.Continue;
        //};

```

```
        //// Сейчас будет полагать что соответствие шаблону (ограничению) произойдёт только один раз
        //Trigger(new[] { Constants.Null }, null,
        //        new[] { Constants.Itself, Constants.Null, Constants.Null },
        substitutedHandler);

        //// TODO: Возможна реализация опционального поведения (один ноль-пустота, бесконечность нолей-пустот)
        //// 0 => 1

        //// 0 => 1
        //// 0 => 2
        //// 0 => 3
        //// ...

        /// <summary>
        /// Обновляет связь с указанными restrictions[Constants.IndexPart] в адресом связи
        /// на связь с указанным новым содержимым.
        /// </summary>
        /// <param name="restrictions">
        /// Ограничения на содержимое связей.
        /// Предполагается, что будет указан индекс связи (в restrictions[Constants.IndexPart]) и далее за ним будет следовать содержимое связи.
        /// Каждое ограничение может иметь значения: Constants.Null - 0-я связь, обозначающая ссылку на пустоту,
        /// Constants.Itself - требование установить ссылку на себя, 1..∞ конкретный индекс другой связи.
        /// </param>
        /// <returns>Индекс обновлённой связи.</returns>
        TLink Update(IList<TLink> restrictions); // TODO: Возможно и возвращать связь нужно целиком.

        // TODO: Move to UniLinksExtensions
        //// { 1, any, any } => { 1, x, y }
        //// { 1 } => { 1, x, y } *
        //// { 1, 3, 4 }

        //Trigger(new[] { restrictions[Constants.IndexPart] }, null,
        //        new[] { restrictions[Constants.IndexPart], restrictions[Constants.SourcePart], restrictions[Constants.TargetPart] }, null);

        //return restrictions[Constants.IndexPart];

        /// <summary>Удаляет связь с указанным индексом.</summary>
        /// <param name="link">Индекс удаляемой связи.</param>
        void Delete(TLink link); // TODO: Возможно всегда нужно принимать restrictions, а так же возвращать удалённую связь, если удаление было реально выполнено, и Null, если нет.

        // TODO: Move to UniLinksExtensions
        //// { 1 } => { 0, 0, 0 }
        //// { 1 } => { 0 } *
        //Trigger(new[] { link }, null,
        //        new[] { Constants.Null }, null);

        // TODO: Если учесть последние TODO, тогда все функции Create, Update, Delete будут иметь один и тот же интерфейс - IList<TLink> Method(IList<TLink> restrictions);, что может быть удобно для "Create|Update|Delete" транзакционности, !! но нужна ли такая транзакционность? Ведь всё что нужно записывать в транзакцию это изменение с чего в во что. Создание это index, 0, 0 -> index, X, Y (и начало отслеживания связи).
        // Удаление это всегда index, X, Y -> index, 0, 0 (и прекращение отслеживания связи).
        // Обновление - аналогично, но состояние отслеживания не меняется.
        // TODO: Хотя пожалуй, выдавать дополнительное значение в виде True/False вряд ли допустимо для Delete. Тогда создание это 0,0,0 -> I,S,T и т.п.
        // TODO: Если все методы, Create, Update, Delete будут и принимать и возвращать IList<TLink>, то можно всё заменить одним единым Update, у которого для удаления нужно указать исходный индекс связи и Constans.Null в качестве его нового значения (возможно будет указано 2 ограничения из 3-х)

        #endregion
    }
}
```

./ILinksExtensions.cs

```
using System;
using System.Collections.Generic;
using System.Runtime.CompilerServices;
using Platform.Setters;
using Platform.Data.Exceptions;

```

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

namespace Platform.Data
{
    public static class ILinksExtensions
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static TLink Count<TLink, TConstants>(this ILinks<TLink, TConstants> links,
        ↪  params TLink[] restrictions)
            where TConstants : LinksConstants<TLink>
            => links.Count(restrictions);

        /// <summary>
        /// Возвращает значение, определяющее существует ли связь с указанным индексом в
        ↪  хранилище связей.
        /// </summary>
        /// <param name="links">Хранилище связей.</param>
        /// <param name="link">Индекс проверяемой на существование связи.</param>
        /// <returns>Значение, определяющее существует ли связь.</returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool Exists<TLink, TConstants>(this ILinks<TLink, TConstants> links, TLink
        ↪  link)
            where TConstants : LinksConstants<TLink>
            => Comparer<TLink>.Default.Compare(links.Count(link), default) > 0;

        /// <param name="links">Хранилище связей.</param>
        /// <param name="link">Индекс проверяемой на существование связи.</param>
        /// <remarks>
        /// TODO: May be move to EnsureExtensions or make it both there and here
        /// </remarks>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void EnsureLinkExists<TLink, TConstants>(this ILinks<TLink, TConstants>
        ↪  links, TLink link)
            where TConstants : LinksConstants<TLink>
        {
            if (!links.Exists(link))
            {
                throw new ArgumentLinkDoesNotExistsException<TLink>(link);
            }
        }

        /// <param name="links">Хранилище связей.</param>
        /// <param name="link">Индекс проверяемой на существование связи.</param>
        /// <param name="argumentName">Имя аргумента, в который передаётся индекс связи.</param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void EnsureLinkExists<TLink, TConstants>(this ILinks<TLink, TConstants>
        ↪  links, TLink link, string argumentName)
            where TConstants : LinksConstants<TLink>
        {
            if (!links.Exists(link))
            {
                throw new ArgumentLinkDoesNotExistsException<TLink>(link, argumentName);
            }
        }

        /// <summary>
        /// Выполняет проход по всем связям, соответствующим шаблону, вызывая обработчик
        ↪  (handler) для каждой подходящей связи.
        /// </summary>
        /// <param name="links">Хранилище связей.</param>
        /// <param name="handler">Обработчик каждой подходящей связи.</param>
        /// <param name="restrictions">Ограничения на содержимое связей. Каждое ограничение
        ↪  может иметь значения: Constants.Null - 0-я связь, обозначающая ссылку на пустоту,
        ↪  Any - отсутствие ограничения, 1..∞ конкретный индекс связи.</param>
        /// <returns>True, в случае если проход по связям не был прерван и False в обратном
        ↪  случае.</returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static TLink Each<TLink, TConstants>(this ILinks<TLink, TConstants> links,
        ↪  Func<IList<TLink>, TLink> handler, params TLink[] restrictions)
            where TConstants : LinksConstants<TLink>
            => links.Each(handler, restrictions);

        /// <summary>
        /// Возвращает части-значения для связи с указанным индексом.
        /// </summary>
        /// <param name="links">Хранилище связей.</param>
        /// <param name="link">Индекс связи.</param>
        /// <returns>Уникальную связь.</returns>
```

```
75          [MethodImpl(MethodImplOptions.AggressiveInlining)]
76          public static IList<TLink> GetLink<TLink, TConstants>(this ILinks<TLink, TConstants>
            ↪  links, TLink link)
77              where TConstants : LinksConstants<TLink>
78          {
79              var constants = links.Constants;
80              var linkPartsSetter = new Setter<IList<TLink>, TLink>(constants.Continue,
                ↪  constants.Break);
81              links.Each(linkPartsSetter.SetAndReturnTrue, link);
82              return linkPartsSetter.Result;
83          }
84
85          #region Points
86
87          /// <summary>Возвращает значение, определяющее является ли связь с указанным индексом
            ↪  точкой полностью (связью замкнутой на себе дважды).</summary>
88          /// <param name="links">Хранилище связей.</param>
89          /// <param name="link">Индекс проверяемой связи.</param>
90          /// <returns>Значение, определяющее является ли связь точкой полностью.</returns>
91          /// <remarks>
92          /// Связь точка - это связь, у которой начало (Source) и конец (Target) есть сама эта
            ↪  связь.
93          /// Но что, если точка уже есть, а нужно создать пару с таким же значением? Должны ли
            ↪  точка и пара существовать одновременно?
94          /// Или в качестве решения для точек нужно использовать 0 в качестве начала и конца, а
            ↪  сортировать по индексу в массиве связей?
95          /// Какое тогда будет значение Source и Target у точки? 0 или её индекс?
96          /// Или точка должна быть одновременно точкой и парой, а также последовательностями из
            ↪  самой себя любого размера?
97          /// Как только есть ссылка на себя, появляется этот парадокс, причём достаточно даже
            ↪  одной ссылки на себя (частичной точки).
98          /// А что если не выбирать что является точкой, пара нулей (цикл через пустоту) или
99          /// самостоятельный цикл через себя? Что если предоставить все варианты использования
            ↪  связей?
100         /// Что если разрешить и нули, а так же частичные варианты?
101         ///
102         /// Что если точка, это только в том случае когда link.Source == link &amp;amp;&amp;
            ↪  link.Target == link , т.е. дважды ссылка на себя.
103         /// А пара это тогда, когда link.Source == link.Target &amp;amp;&amp; link.Source != link ,
            ↪  т.е. ссылка не на себя а во вне.
104         ///
105         /// Тогда если у нас уже создана пара, но нам нужна точка, мы можем используя
            ↪  промежуточную связь,
106         /// например "DoubletOf" обозначить что является точно парой, а что точно точкой.
107         /// И наоборот этот же метод поможет, если уже существует точка, но нам нужна пара.
108         /// </remarks>
109         public static bool IsFullPoint<TLink, TConstants>(this ILinks<TLink, TConstants> links,
            ↪  TLink link)
110             where TConstants : LinksConstants<TLink>
111         {
112             links.EnsureLinkExists(link);
113             return Point<TLink>.IsFullPoint(links.GetLink(link));
114         }
115
116         /// <summary>Возвращает значение, определяющее является ли связь с указанным индексом
            ↪  точкой частично (связью замкнутой на себе как минимум один раз).</summary>
117         /// <param name="links">Хранилище связей.</param>
118         /// <param name="link">Индекс проверяемой связи.</param>
119         /// <returns>Значение, определяющее является ли связь точкой частично.</returns>
120         /// <remarks>
121         /// Достаточно любой одной ссылки на себя.
122         /// Также в будущем можно будет проверять и всех родителей, чтобы проверить есть ли
            ↪  ссылки на себя (на эту связь).
123         /// </remarks>
124         public static bool IsPartialPoint<TLink, TConstants>(this ILinks<TLink, TConstants>
            ↪  links, TLink link)
125             where TConstants : LinksConstants<TLink>
126         {
127             links.EnsureLinkExists(link);
128             return Point<TLink>.IsPartialPoint(links.GetLink(link));
129         }
130
131         #endregion
132     }
133 }
```

./ISynchronizedLinks.cs

```
using Platform.Threading.Synchronization;

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

namespace Platform.Data
{
    public interface ISynchronizedLinks<TLink, TLinks, TConstants> : ISynchronized<TLinks>,
    ↪  ILinks<TLink, TConstants>
        where TConstants : LinksConstants<TLink>
        where TLinks : ILinks<TLink, TConstants>
    {
    }
}
```

./LinksConstants.cs

```
using Platform.Numbers;
using Platform.Ranges;
using Platform.Reflection;

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

namespace Platform.Data
{
    public class LinksConstants<TAddress>
    {
        public static readonly int DefaultTargetPart = 2;

        #region Link parts

        /// <summary>Возвращает индекс части, которая отвечает за индекс (адрес, идентификатор)
        ↪  самой связи.</summary>
        public int IndexPart { get; }

        /// <summary>Возвращает индекс части, которая отвечает за ссылку на связь-начало (первая
        ↪  часть-значение).</summary>
        public int SourcePart { get; }

        /// <summary>Возвращает индекс части, которая отвечает за ссылку на связь-конец
        ↪  (последняя часть-значение).</summary>
        public int TargetPart { get; }

        #endregion

        #region Flow control

        /// <summary>Возвращает значение, обозначающее продолжение прохода по связям.</summary>
        /// <remarks>Используется в функции обработчике, который передаётся в функцию
        ↪  Each.</remarks>
        public TAddress Continue { get; }

        /// <summary>Возвращает значение, обозначающее пропуск в проходе по связям.</summary>
        public TAddress Skip { get; }

        /// <summary>Возвращает значение, обозначающее остановку прохода по связям.</summary>
        /// <remarks>Используется в функции обработчике, который передаётся в функцию
        ↪  Each.</remarks>
        public TAddress Break { get; }

        #endregion

        #region Special symbols

        /// <summary>Возвращает значение, обозначающее отсутствие связи.</summary>
        public TAddress Null { get; }

        /// <summary>Возвращает значение, обозначающее любую связь.</summary>
        /// <remarks>Возможно нужно зарезервировать отдельное значение, тогда можно будет
        ↪  создавать все варианты последовательностей в функции Create.</remarks>
        public TAddress Any { get; }

        /// <summary>Возвращает значение, обозначающее связь-ссылку на саму связь.</summary>
        public TAddress Itself { get; }

        #endregion

        #region References

        /// <summary>Возвращает диапазон возможных индексов для внутренних связей (внутренних
        ↪  ссылок).</summary>
```

```
58          public Range<TAddress> PossibleInnerReferencesRange { get; }
59
60          /// <summary>Возвращает диапазон возможных индексов для внешних связей (внешних
            ↪  ссылок).</summary>
61          public Range<TAddress>? PossibleExternalReferencesRange { get; }
62
63          #endregion
64
65          public LinksConstants(int targetPart, Range<TAddress> possibleInnerReferencesRange,
            ↪  Range<TAddress>? possibleExternalReferencesRange)
66          {
67              IndexPart = 0;
68              SourcePart = 1;
69              TargetPart = targetPart;
70              Null = Integer<TAddress>.Zero;
71              Break = Integer<TAddress>.Zero;
72              var currentInnerReferenceIndex = possibleInnerReferencesRange.Maximum;
73              Continue = currentInnerReferenceIndex;
74              Decrement(ref currentInnerReferenceIndex);
75              Skip = currentInnerReferenceIndex;
76              Decrement(ref currentInnerReferenceIndex);
77              Any = currentInnerReferenceIndex;
78              Decrement(ref currentInnerReferenceIndex);
79              Itself = currentInnerReferenceIndex;
80              Decrement(ref currentInnerReferenceIndex);
81              PossibleInnerReferencesRange = (possibleInnerReferencesRange.Minimum,
                ↪  currentInnerReferenceIndex);
82              PossibleExternalReferencesRange = possibleExternalReferencesRange;
83          }
84
85          private static void Decrement(ref TAddress currentInnerReferenceIndex) =>
            ↪  currentInnerReferenceIndex = Arithmetic.Decrement(currentInnerReferenceIndex);
86
87          public LinksConstants(Range<TAddress> possibleInnerReferencesRange, Range<TAddress>?
            ↪  possibleExternalReferencesRange) : this(DefaultTargetPart,
            ↪  possibleInnerReferencesRange, possibleExternalReferencesRange) { }
88
89          public LinksConstants(int targetPart, Range<TAddress> possibleInnerReferencesRange) :
            ↪  this(targetPart, possibleInnerReferencesRange, null) { }
90
91          public LinksConstants(Range<TAddress> possibleInnerReferencesRange) :
            ↪  this(DefaultTargetPart, possibleInnerReferencesRange, null) { }
92
93          public LinksConstants() : this(DefaultTargetPart, (Integer<TAddress>.One,
            ↪  NumericType<TAddress>.MaxValue), null) { }
94      }
95  }
```

./LinksConstantsExtensions.cs

```
1   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
2
3   using System.Runtime.CompilerServices;
4
5   namespace Platform.Data
6   {
7       public static class LinksConstantsExtensions
8       {
9           [MethodImpl(MethodImplOptions.AggressiveInlining)]
10          public static bool IsReference<TAddress>(this LinksConstants<TAddress> linksConstants,
            ↪  TAddress address) => linksConstants.IsInnerReference(address) ||
            ↪  linksConstants.IsExternalReference(address);
11
12          [MethodImpl(MethodImplOptions.AggressiveInlining)]
13          public static bool IsInnerReference<TAddress>(this LinksConstants<TAddress>
            ↪  linksConstants, TAddress address) =>
            ↪  linksConstants.PossibleInnerReferencesRange.ContainsValue(address);
14
15          [MethodImpl(MethodImplOptions.AggressiveInlining)]
16          public static bool IsExternalReference<TAddress>(this LinksConstants<TAddress>
            ↪  linksConstants, TAddress address) =>
            ↪  linksConstants.PossibleExternalReferencesRange?.ContainsValue(address) ?? false;
17      }
18  }
```

./Point.cs

```
1   using System.Collections.Generic;
2   using System.Runtime.CompilerServices;
3   using Platform.Exceptions;
4   using Platform.Ranges;
```

```
using Platform.Collections;

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

namespace Platform.Data
{
    public static class Point<TLink>
    {
        private static readonly EqualityComparer<TLink> _equalityComparer =
        ↪  EqualityComparer<TLink>.Default;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool IsFullPoint(params TLink[] link) => IsFullPoint((IList<TLink>)link);

        public static bool IsFullPoint(IList<TLink> link)
        {
            Ensure.Always.ArgumentNotEmpty(link, nameof(link));
            Ensure.Always.ArgumentInRange(link.Count, new Range<int>(2, int.MaxValue),
            ↪  nameof(link), "Cannot determine link's pointness using only its identifier.");
            return IsFullPointUnchecked(link);
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool IsFullPointUnchecked(IList<TLink> link)
        {
            var result = true;
            for (var i = 1; result && i < link.Count; i++)
            {
                result = _equalityComparer.Equals(link[0], link[i]);
            }
            return result;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool IsPartialPoint(params TLink[] link) =>
        ↪  IsPartialPoint((IList<TLink>)link);

        public static bool IsPartialPoint(IList<TLink> link)
        {
            Ensure.Always.ArgumentNotEmpty(link, nameof(link));
            Ensure.Always.ArgumentInRange(link.Count, new Range<int>(2, int.MaxValue),
            ↪  nameof(link), "Cannot determine link's pointness using only its identifier.");
            return IsPartialPointUnchecked(link);
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool IsPartialPointUnchecked(IList<TLink> link)
        {
            var result = false;
            for (var i = 1; !result && i < link.Count; i++)
            {
                result = _equalityComparer.Equals(link[0], link[i]);
            }
            return result;
        }
    }
}
```

./Sequences/ISequenceAppender.cs

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

namespace Platform.Data.Sequences
{
    public interface ISequenceAppender<TLink>
    {
        TLink Append(TLink sequence, TLink appendant);
    }
}
```

./Sequences/ISequences.cs

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

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

namespace Platform.Data.Sequences
{
    public interface ISequences<TLink>
    {
```

```
        ulong Count(params TLink[] sequence);
        bool Each(Func<TLink, bool> handler, IList<TLink> sequence);
        bool EachPart(Func<TLink, bool> handler, TLink sequence);
        TLink Create(params TLink[] sequence);
        TLink Update(TLink[] sequence, TLink[] newSequence);
        void Delete(params TLink[] sequence);
    }
}
```

./Sequences/ISequenceWalker.cs

```
using System.Collections.Generic;

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

namespace Platform.Data.Sequences
{
    public interface ISequenceWalker<TLink>
    {
        IEnumerable<IList<TLink>> Walk(TLink sequence);
    }
}
```

./Sequences/SequenceWalker.cs

```
using System;
using System.Collections.Generic;
using System.Runtime.CompilerServices;

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

namespace Platform.Data.Sequences
{
    /// <remarks>
    /// Реализованный внутри алгоритм наглядно показывает,
    /// что совершенно не обязательна рекурсивная реализация (с вложенным вызовом функцией самой себя),
    /// так как стэк можно использовать намного эффективнее при ручном управлении.
    ///
    /// При оптимизации можно использовать встроенную поддержку стеков в процессор.
    ///
    /// Решить объединять ли логику в одну функцию, или оставить 4 отдельных реализации?
    /// Решить встраивать ли защиту от зацикливания.
    /// Альтернативой защиты от закливания может быть заранее известное ограничение на погружение вглубь.
    /// А так же качественное распознавание прохода по циклическому графу.
    /// Ограничение на уровень глубины рекурсии может позволить использовать уменьшенный размер стека.
    /// Можно использовать глобальный стек (или несколько глобальных стеков на каждый поток).
    ///
    /// TODO: Попробовать реализовать алгоритм используя Sigil (MSIL) и низкоуровневый стек и сравнить производительность.
    /// </remarks>
    public static class SequenceWalker
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void WalkRight<TLink>(TLink sequence, Func<TLink, TLink> getSource, Func<TLink, TLink> getTarget, Func<TLink, bool> isElement, Action<TLink> visit)
        {
            var stack = new Stack<TLink>();
            var element = sequence;
            if (isElement(element))
            {
                visit(element);
            }
            else
            {
                while (true)
                {
                    if (isElement(element))
                    {
                        if (stack.Count == 0)
                        {
                            break;
                        }
                        element = stack.Pop();
                        var source = getSource(element);
                        var target = getTarget(element);
                        if (isElement(source))
                        {
                            visit(source);
```

```
                        }
                        if (isElement(target))
                        {
                            visit(target);
                        }
                        element = target;
                    }
                    else
                    {
                        stack.Push(element);
                        element = getSource(element);
                    }
                }
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void WalkLeft<TLink>(TLink sequence, Func<TLink, TLink> getSource,
        ↪  Func<TLink, TLink> getTarget, Func<TLink, bool> isElement, Action<TLink> visit)
        {
            var stack = new Stack<TLink>();
            var element = sequence;
            if (isElement(element))
            {
                visit(element);
            }
            else
            {
                while (true)
                {
                    if (isElement(element))
                    {
                        if (stack.Count == 0)
                        {
                            break;
                        }
                        element = stack.Pop();
                        var source = getSource(element);
                        var target = getTarget(element);
                        if (isElement(target))
                        {
                            visit(target);
                        }
                        if (isElement(source))
                        {
                            visit(source);
                        }
                        element = source;
                    }
                    else
                    {
                        stack.Push(element);
                        element = getTarget(element);
                    }
                }
            }
        }
    }
}
```

./Sequences/StopableSequenceWalker.cs

```
using System;
using System.Collections.Generic;
using System.Runtime.CompilerServices;

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

namespace Platform.Data.Sequences
{
    /// <remarks>
    /// Реализованный внутри алгоритм наглядно показывает,
    /// что совершенно не обязательна рекурсивная реализация (с вложенным вызовом функцией самой
    ↪  себя),
    /// так как стэк можно использовать намного эффективнее при ручном управлении.
    ///
    /// При оптимизации можно использовать встроенную поддержку стеков в процессор.
    ///
    /// Решить объединять ли логику в одну функцию, или оставить 4 отдельных реализации?
    /// Решить встраивать ли защиту от зацикливания.
```

```
18      /// Альтернативой защиты от закливания может быть заранее известное ограничение на
        ↪  погружение вглубь.
19      /// А так же качественное распознавание прохода по циклическому графу.
20      /// Ограничение на уровень глубины рекурсии может позволить использовать уменьшенный размер
        ↪  стека.
21      /// Можно использовать глобальный стек (или несколько глобальных стеков на каждый поток).
22      /// </remarks>
23      public static class StopableSequenceWalker
24      {
25          [MethodImpl(MethodImplOptions.AggressiveInlining)]
26          public static bool WalkRight<TLink>(TLink sequence, Func<TLink, TLink> getSource,
            ↪  Func<TLink, TLink> getTarget, Func<TLink, bool> isElement, Action<TLink> enter,
            ↪  Action<TLink> exit, Func<TLink, bool> canEnter, Func<TLink, bool> visit)
27          {
28              var exited = 0;
29              var stack = new Stack<TLink>();
30              var element = sequence;
31              if (isElement(element))
32              {
33                  return visit(element);
34              }
35              while (true)
36              {
37                  if (isElement(element))
38                  {
39                      if (stack.Count == 0)
40                      {
41                          return true;
42                      }
43                      element = stack.Pop();
44                      exit(element);
45                      exited++;
46                      var source = getSource(element);
47                      var target = getTarget(element);
48                      if ((isElement(source) || (exited == 1 && !canEnter(source))) &&
                        ↪  !visit(source))
49                      {
50                          return false;
51                      }
52                      if ((isElement(target) || !canEnter(target)) && !visit(target))
53                      {
54                          return false;
55                      }
56                      element = target;
57                  }
58                  else
59                  {
60                      if (canEnter(element))
61                      {
62                          enter(element);
63                          exited = 0;
64                          stack.Push(element);
65                          element = getSource(element);
66                      }
67                      else
68                      {
69                          if (stack.Count == 0)
70                          {
71                              return true;
72                          }
73                          element = stack.Pop();
74                          exit(element);
75                          exited++;
76                          var source = getSource(element);
77                          var target = getTarget(element);
78                          if ((isElement(source) || (exited == 1 && !canEnter(source))) &&
                            ↪  !visit(source))
79                          {
80                              return false;
81                          }
82                          if ((isElement(target) || !canEnter(target)) && !visit(target))
83                          {
84                              return false;
85                          }
86                          element = target;
87                      }
88                  }
89              }
90          }
```

```
91
92          [MethodImpl(MethodImplOptions.AggressiveInlining)]
93          public static bool WalkRight<TLink>(TLink sequence, Func<TLink, TLink> getSource,
            ↪  Func<TLink, TLink> getTarget, Func<TLink, bool> isElement, Func<TLink, bool> visit)
94          {
95              var stack = new Stack<TLink>();
96              var element = sequence;
97              if (isElement(element))
98              {
99                  return visit(element);
100             }
101             while (true)
102             {
103                 if (isElement(element))
104                 {
105                     if (stack.Count == 0)
106                     {
107                         return true;
108                     }
109                     element = stack.Pop();
110                     var source = getSource(element);
111                     var target = getTarget(element);
112                     if (isElement(source) && !visit(source))
113                     {
114                         return false;
115                     }
116                     if (isElement(target) && !visit(target))
117                     {
118                         return false;
119                     }
120                     element = target;
121                 }
122                 else
123                 {
124                     stack.Push(element);
125                     element = getSource(element);
126                 }
127             }
128         }
129
130         [MethodImpl(MethodImplOptions.AggressiveInlining)]
131         public static bool WalkLeft<TLink>(TLink sequence, Func<TLink, TLink> getSource,
            ↪  Func<TLink, TLink> getTarget, Func<TLink, bool> isElement, Func<TLink, bool> visit)
132         {
133             var stack = new Stack<TLink>();
134             var element = sequence;
135             if (isElement(element))
136             {
137                 return visit(element);
138             }
139             while (true)
140             {
141                 if (isElement(element))
142                 {
143                     if (stack.Count == 0)
144                     {
145                         return true;
146                     }
147                     element = stack.Pop();
148                     var source = getSource(element);
149                     var target = getTarget(element);
150                     if (isElement(target) && !visit(target))
151                     {
152                         return false;
153                     }
154                     if (isElement(source) && !visit(source))
155                     {
156                         return false;
157                     }
158                     element = source;
159                 }
160                 else
161                 {
162                     stack.Push(element);
163                     element = getTarget(element);
164                 }
165             }
166         }
```

```
    }
}
```

./Universal/IUniLinksCRUD.cs

```
using System;

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

namespace Platform.Data.Universal
{
    /// <remarks>
    /// CRUD aliases for IUniLinks.
    /// </remarks>
    public interface IUniLinksCRUD<TLink>
    {
        TLink Read(ulong partType, TLink link);
        bool Read(Func<TLink, bool> handler, params TLink[] pattern);
        TLink Create(TLink[] parts);
        TLink Update(TLink[] before, TLink[] after);
        void Delete(TLink[] parts);
    }
}
```

./Universal/IUniLinks.cs

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

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

namespace Platform.Data.Universal
{
    /// <remarks>Minimal sufficient universal Links API (for bulk operations).</remarks>
    public partial interface IUniLinks<TLink>
    {
        IList<IList<IList<TLink>>> Trigger(IList<TLink> condition, IList<TLink> substitution);
    }

    /// <remarks>Minimal sufficient universal Links API (for step by step operations).</remarks>
    public partial interface IUniLinks<TLink>
    {
        /// <returns>
        /// TLink that represents True (was finished fully) or TLink that represents False (was
        ↪  stopped).
        /// This is done to assure ability to push up stop signal through recursion stack.
        /// </returns>
        /// <remarks>
        /// { 0, 0, 0 } => { itself, itself, itself } // create
        /// { 1, any, any } => { itself, any, 3 } // update
        /// { 3, any, any } => { 0, 0, 0 } // delete
        /// </remarks>
        TLink Trigger(IList<TLink> patternOrCondition, Func<IList<TLink>, TLink> matchHandler,
                      IList<TLink> substitution, Func<IList<TLink>, IList<TLink>, TLink>
                      ↪  substitutionHandler);

        TLink Trigger(IList<TLink> restriction, Func<IList<TLink>, IList<TLink>, TLink>
        ↪  matchedHandler,
                      IList<TLink> substitution, Func<IList<TLink>, IList<TLink>, TLink>
                      ↪  substitutedHandler);
    }

    /// <remarks>Extended with small optimization.</remarks>
    public partial interface IUniLinks<TLink>
    {
        /// <remarks>
        /// Something simple should be simple and optimized.
        /// </remarks>
        TLink Count(IList<TLink> restrictions);
    }
}
```

./Universal/IUniLinksGS.cs

```
using System;

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

```

```
namespace Platform.Data.Universal
{
    /// <remarks>
    /// Get/Set aliases for IUniLinks.
    /// </remarks>
    public interface IUniLinksGS<TLink>
    {
        TLink Get(ulong partType, TLink link);
        bool Get(Func<TLink, bool> handler, params TLink[] pattern);
        TLink Set(TLink[] before, TLink[] after);
    }
}
```

./Universal/IUniLinksIO.cs

```
using System;

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

namespace Platform.Data.Universal
{
    /// <remarks>
    /// In/Out aliases for IUniLinks.
    /// TLink can be any number type of any size.
    /// </remarks>
    public interface IUniLinksIO<TLink>
    {
        /// <remarks>
        /// default(TLink) means any link.
        /// Single element pattern means just element (link).
        /// Handler gets array of link contents.
        /// * link[0] is index or identifier.
        /// * link[1] is source or first.
        /// * link[2] is target or second.
        /// * link[3] is linker or third.
        /// * link[n] is nth part/parent/element/value
        /// of link (if variable length links used).
        ///
        /// Stops and returns false if handler return false.
        ///
        /// Acts as Each, Foreach, Select, Search, Match &amp;amp; ...
        ///
        /// Handles all links in store if pattern/restrictions is not defined.
        /// </remarks>
        bool Out(Func<TLink[], bool> handler, params TLink[] pattern);

        /// <remarks>
        /// default(TLink) means itself.
        /// Equivalent to:
        /// * creation if before == null
        /// * deletion if after == null
        /// * update if before != null &amp;amp;&amp;amp; after != null
        /// * default(TLink) if before == null &amp;amp;&amp;amp; after == null
        ///
        /// Possible interpretation
        /// * In(null, new[] { }) creates point (link that points to itself using minimum number
        ↪  of parts).
        /// * In(new[] { 4 }, null) deletes 4th link.
        /// * In(new[] { 4 }, new [] { 5 }) delete 5th link if it exists and moves 4th link to
        ↪  5th index.
        /// * In(new[] { 4 }, new [] { 0, 2, 3 }) replaces 4th link with new doublet link (with
        ↪  2 as source and 3 as target), 0 means it can be placed in any address.
        /// ...
        /// </remarks>
        TLink In(TLink[] before, TLink[] after);
    }
}
```

./Universal/IUniLinksIOWithExtensions.cs

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

namespace Platform.Data.Universal
{
    /// <remarks>Contains some optimizations of Out.</remarks>
    public interface IUniLinksIOWithExtensions<TLink> : IUniLinksIO<TLink>
    {
        /// <remarks>
```

```
        /// default(TLink) means nothing or null.
        /// Single element pattern means just element (link).
        /// OutPart(n, null) returns default(TLink).
        /// OutPart(0, pattern) ~ Exists(link) or Search(pattern)
        /// OutPart(1, pattern) ~ GetSource(link) or GetSource(Search(pattern))
        /// OutPart(2, pattern) ~ GetTarget(link) or GetTarget(Search(pattern))
        /// OutPart(3, pattern) ~ GetLinker(link) or GetLinker(Search(pattern))
        /// OutPart(n, pattern) => For any variable length links, returns link or default(TLink).
        ///
        /// Outs(returns) inner contents of link, its part/parent/element/value.
        /// </remarks>
        TLink OutOne(ulong partType, params TLink[] pattern);

        /// <remarks>OutCount() returns total links in store as array.</remarks>
        TLink[][] OutAll(params TLink[] pattern);

        /// <remarks>OutCount() returns total amount of links in store.</remarks>
        ulong OutCount(params TLink[] pattern);
    }
}
```

./Universal/IUniLinksRW.cs

```
using System;

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

namespace Platform.Data.Universal
{
    /// <remarks>
    /// Read/Write aliases for IUniLinks.
    /// </remarks>
    public interface IUniLinksRW<TLink>
    {
        TLink Read(ulong partType, TLink link);
        bool Read(Func<TLink, bool> handler, params TLink[] pattern);
        TLink Write(TLink[] before, TLink[] after);
    }
}
```

Index