./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;
using Platform.Data.Constants;

#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;
        //};
```

```
71
72          //// Сейчас будет полагать что соответствие шаблону (ограничению) произойдёт только один
            ↪  раз
73          //Trigger(new[] { Constants.Null }, null,
74          //        new[] { Constants.Itself, Constants.Null, Constants.Null },
            ↪  substitutedHandler);
75
76          //// TODO: Возможна реализация опционального поведения (один ноль-пустота, бесконечность
            ↪  нолей-пустот)
77          //// 0 => 1
78
79          //// 0 => 1
80          //// 0 => 2
81          //// 0 => 3
82          //// ...
83
84          /// <summary>
85          /// Обновляет связь с указанными restrictions[Constants.IndexPart] в адресом связи
86          /// на связь с указанным новым содержимым.
87          /// </summary>
88          /// <param name="restrictions">
89          /// Ограничения на содержимое связей.
90          /// Предполагается, что будет указан индекс связи (в restrictions[Constants.IndexPart])
            ↪  и далее за ним будет следовать содержимое связи.
91          /// Каждое ограничение может иметь значения: Constants.Null - 0-я связь, обозначающая
            ↪  ссылку на пустоту,
92          /// Constants.Itself - требование установить ссылку на себя, 1..∞ конкретный индекс
            ↪  другой связи.
93          /// </param>
94          /// <returns>Индекс обновлённой связи.</returns>
95          TLink Update(IList<TLink> restrictions); // TODO: Возможно и возвращать связь нужно
            ↪  целиком.
96
97          // TODO: Move to UniLinksExtensions
98          //// { 1, any, any } => { 1, x, y }
99          //// { 1 } => { 1, x, y } *
100         //// { 1, 3, 4 }
101
102         //Trigger(new[] { restrictions[Constants.IndexPart] }, null,
103         //        new[] { restrictions[Constants.IndexPart], restrictions[Constants.SourcePart],
            ↪  restrictions[Constants.TargetPart] }, null);
104
105         //return restrictions[Constants.IndexPart];
106
107         /// <summary>Удаляет связь с указанным индексом.</summary>
108         /// <param name="link">Индекс удаляемой связи.</param>
109         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.
110
111         // TODO: Move to UniLinksExtensions
112         //// { 1 } => { 0, 0, 0 }
113         //// { 1 } => { 0 } *
114         //Trigger(new[] { link }, null,
115         //        new[] { Constants.Null }, null);
116
117         // 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.
118         // 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.
119         // 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)
120
121         #endregion
122     }
123 }
```

./ILinksExtensions.cs

```
1  using System;
2  using System.Collections.Generic;
3  using System.Runtime.CompilerServices;
4  using Platform.Setters;
5  using Platform.Data.Constants;
```

```
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>
```

```
74          /// <param name="link">Индекс связи.</param>
75          /// <returns>Уникальную связь.</returns>
76          [MethodImpl(MethodImplOptions.AggressiveInlining)]
77          public static ILink<TLink> GetLink<TLink, TConstants>(this ILinks<TLink, TConstants>
            ↪  links, TLink link)
78              where TConstants : LinksConstants<TLink>
79          {
80              var constants = links.Constants;
81              var linkPartsSetter = new Setter<ILink<TLink>, TLink>(constants.Continue,
                ↪  constants.Break);
82              links.Each(linkPartsSetter.SetAndReturnTrue, link);
83              return linkPartsSetter.Result;
84          }
85
86          #region Points
87
88          /// <summary>Возвращает значение, определяющее является ли связь с указанным индексом
            ↪  точкой полностью (связью замкнутой на себе дважды).</summary>
89          /// <param name="links">Хранилище связей.</param>
90          /// <param name="link">Индекс проверяемой связи.</param>
91          /// <returns>Значение, определяющее является ли связь точкой полностью.</returns>
92          /// <remarks>
93          /// Связь точка - это связь, у которой начало (Source) и конец (Target) есть сама эта
            ↪  связь.
94          /// Но что, если точка уже есть, а нужно создать пару с таким же значением? Должны ли
            ↪  точка и пара существовать одновременно?
95          /// Или в качестве решения для точек нужно использовать 0 в качестве начала и конца, а
            ↪  сортировать по индексу в массиве связей?
96          /// Какое тогда будет значение Source и Target у точки? 0 или её индекс?
97          /// Или точка должна быть одновременно точкой и парой, а также последовательностями из
            ↪  самой себя любого размера?
98          /// Как только есть ссылка на себя, появляется этот парадокс, причём достаточно даже
            ↪  одной ссылки на себя (частичной точки).
99          /// А что если не выбирать что является точкой, пара нулей (цикл через пустоту) или
100         /// самостоятельный цикл через себя? Что если предоставить все варианты использования
            ↪  связей?
101         /// Что если разрешить и нули, а так же частичные варианты?
102         ///
103         /// Что если точка, это только в том случае когда link.Source == link &amp;amp;&amp;amp;
            ↪  link.Target == link , т.е. дважды ссылка на себя.
104         /// А пара это тогда, когда link.Source == link.Target &amp;amp;&amp;amp; link.Source != link ,
            ↪  т.е. ссылка не на себя а во вне.
105         ///
106         /// Тогда если у нас уже создана пара, но нам нужна точка, мы можем используя
            ↪  промежуточную связь,
107         /// например "DoubletOf" обозначить что является точно парой, а что точно точкой.
108         /// И наоборот этот же метод поможет, если уже существует точка, но нам нужна пара.
109         /// </remarks>
110         public static bool IsFullPoint<TLink, TConstants>(this ILinks<TLink, TConstants> links,
            ↪  TLink link)
111             where TConstants : LinksConstants<TLink>
112         {
113             links.EnsureLinkExists(link);
114             return Point<TLink>.IsFullPoint(links.GetLink(link));
115         }
116
117         /// <summary>Возвращает значение, определяющее является ли связь с указанным индексом
            ↪  точкой частично (связью замкнутой на себе как минимум один раз).</summary>
118         /// <param name="links">Хранилище связей.</param>
119         /// <param name="link">Индекс проверяемой связи.</param>
120         /// <returns>Значение, определяющее является ли связь точкой частично.</returns>
121         /// <remarks>
122         /// Достаточно любой одной ссылки на себя.
123         /// Также в будущем можно будет проверять и всех родителей, чтобы проверить есть ли
            ↪  ссылки на себя (на эту связь).
124         /// </remarks>
125         public static bool IsPartialPoint<TLink, TConstants>(this ILinks<TLink, TConstants>
            ↪  links, TLink link)
126             where TConstants : LinksConstants<TLink>
127         {
128             links.EnsureLinkExists(link);
129             return Point<TLink>.IsPartialPoint(links.GetLink(link));
130         }
131
132         #endregion
133     }
134 }
```

./ISynchronizedLinks.cs

```
using Platform.Threading.Synchronization;
using Platform.Data.Constants;

#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.Constants
{
    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  }
```

./Point.cs

```
1   using System.Collections.Generic;
2   using System.Runtime.CompilerServices;
3   using Platform.Exceptions;
4   using Platform.Ranges;
5   using Platform.Collections;
6
7   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
8
9   namespace Platform.Data
10  {
11      public static class Point<TLink>
12      {
13          private static readonly EqualityComparer<TLink> _equalityComparer =
            ↪  EqualityComparer<TLink>.Default;
14
15          [MethodImpl(MethodImplOptions.AggressiveInlining)]
16          public static bool IsFullPoint(params TLink[] link) => IsFullPoint((IList<TLink>)link);
17
18          public static bool IsFullPoint(IList<TLink> link)
19          {
20              Ensure.Always.ArgumentNotEmpty(link, nameof(link));
21              Ensure.Always.ArgumentInRange(link.Count, new Range<int>(2, int.MaxValue),
                ↪  nameof(link), "Cannot determine link's pointness using only its identifier.");
22              return IsFullPointUnchecked(link);
23          }
24
25          [MethodImpl(MethodImplOptions.AggressiveInlining)]
26          public static bool IsFullPointUnchecked(IList<TLink> link)
27          {
28              var result = true;
29              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;
```

```
4
5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7  namespace Platform.Data.Sequences
8  {
9      /// <remarks>
10     /// Реализованный внутри алгоритм наглядно показывает,
11     /// что совершенно не обязательна рекурсивная реализация (с вложенным вызовом функцией самой
       ↪  себя),
12     /// так как стэк можно использовать намного эффективнее при ручном управлении.
13     ///
14     /// При оптимизации можно использовать встроенную поддержку стеков в процессор.
15     ///
16     /// Решить объединять ли логику в одну функцию, или оставить 4 отдельных реализации?
17     /// 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     ///
23     /// TODO: Попробовать реализовать алгоритм используя Sigil (MSIL) и низкоуровневый стек и
       ↪  сравнить производительность.
24     /// </remarks>
25     public static class SequenceWalker
26     {
27         [MethodImpl(MethodImplOptions.AggressiveInlining)]
28         public static void WalkRight<TLink>(TLink sequence, Func<TLink, TLink> getSource,
           ↪  Func<TLink, TLink> getTarget, Func<TLink, bool> isElement, Action<TLink> visit)
29         {
30             var stack = new Stack<TLink>();
31             var element = sequence;
32             if (isElement(element))
33             {
34                 visit(element);
35             }
36             else
37             {
38                 while (true)
39                 {
40                     if (isElement(element))
41                     {
42                         if (stack.Count == 0)
43                         {
44                             break;
45                         }
46                         element = stack.Pop();
47                         var source = getSource(element);
48                         var target = getTarget(element);
49                         if (isElement(source))
50                         {
51                             visit(source);
52                         }
53                         if (isElement(target))
54                         {
55                             visit(target);
56                         }
57                         element = target;
58                     }
59                     else
60                     {
61                         stack.Push(element);
62                         element = getSource(element);
63                     }
64                 }
65             }
66         }
67
68         [MethodImpl(MethodImplOptions.AggressiveInlining)]
69         public static void WalkLeft<TLink>(TLink sequence, Func<TLink, TLink> getSource,
           ↪  Func<TLink, TLink> getTarget, Func<TLink, bool> isElement, Action<TLink> visit)
70         {
71             var stack = new Stack<TLink>();
72             var element = sequence;
73             if (isElement(element))
74             {
75                 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.
    /// 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).
    /// </remarks>
    public static class StopableSequenceWalker
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        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)
        {
            var exited = 0;
            var stack = new Stack<TLink>();
            var element = sequence;
            if (isElement(element))
            {
                return visit(element);
            }
            while (true)
            {
                if (isElement(element))
                {
```

```
                    if (stack.Count == 0)
                    {
                        return true;
                    }
                    element = stack.Pop();
                    exit(element);
                    exited++;
                    var source = getSource(element);
                    var target = getTarget(element);
                    if ((isElement(source) || (exited == 1 && !canEnter(source))) &&
                    ↪  !visit(source))
                    {
                        return false;
                    }
                    if ((isElement(target) || !canEnter(target)) && !visit(target))
                    {
                        return false;
                    }
                    element = target;
                }
                else
                {
                    if (canEnter(element))
                    {
                        enter(element);
                        exited = 0;
                        stack.Push(element);
                        element = getSource(element);
                    }
                    else
                    {
                        if (stack.Count == 0)
                        {
                            return true;
                        }
                        element = stack.Pop();
                        exit(element);
                        exited++;
                        var source = getSource(element);
                        var target = getTarget(element);
                        if ((isElement(source) || (exited == 1 && !canEnter(source))) &&
                        ↪  !visit(source))
                        {
                            return false;
                        }
                        if ((isElement(target) || !canEnter(target)) && !visit(target))
                        {
                            return false;
                        }
                        element = target;
                    }
                }
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool WalkRight<TLink>(TLink sequence, Func<TLink, TLink> getSource,
        ↪  Func<TLink, TLink> getTarget, Func<TLink, bool> isElement, Func<TLink, bool> visit)
        {
            var stack = new Stack<TLink>();
            var element = sequence;
            if (isElement(element))
            {
                return visit(element);
            }
            while (true)
            {
                if (isElement(element))
                {
                    if (stack.Count == 0)
                    {
                        return true;
                    }
                    element = stack.Pop();
                    var source = getSource(element);
                    var target = getTarget(element);
                    if (isElement(source) && !visit(source))
                    {
                        return false;
```

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

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

./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>
```

```
12      public interface IUniLinksIO<TLink>
13      {
14          /// <remarks>
15          /// default(TLink) means any link.
16          /// Single element pattern means just element (link).
17          /// Handler gets array of link contents.
18          /// * link[0] is index or identifier.
19          /// * link[1] is source or first.
20          /// * link[2] is target or second.
21          /// * link[3] is linker or third.
22          /// * link[n] is nth part/parent/element/value
23          /// of link (if variable length links used).
24          ///
25          /// Stops and returns false if handler return false.
26          ///
27          /// Acts as Each, Foreach, Select, Search, Match &amp;amp; ...
28          ///
29          /// Handles all links in store if pattern/restrictions is not defined.
30          /// </remarks>
31          bool Out(Func<TLink[], bool> handler, params TLink[] pattern);
32
33          /// <remarks>
34          /// default(TLink) means itself.
35          /// Equivalent to:
36          /// * creation if before == null
37          /// * deletion if after == null
38          /// * update if before != null &amp;amp;&amp;amp; after != null
39          /// * default(TLink) if before == null &amp;amp;&amp;amp; after == null
40          ///
41          /// Possible interpretation
42          /// * In(null, new[] { }) creates point (link that points to itself using minimum number
            ↪  of parts).
43          /// * In(new[] { 4 }, null) deletes 4th link.
44          /// * In(new[] { 4 }, new [] { 5 }) delete 5th link if it exists and moves 4th link to
            ↪  5th index.
45          /// * 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.
46          /// ...
47          /// </remarks>
48          TLink In(TLink[] before, TLink[] after);
49      }
50  }
```

./Universal/IUniLinksIOWithExtensions.cs

```
1   // ReSharper disable TypeParameterCanBeVariant
2   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
3
4   namespace Platform.Data.Universal
5   {
6       /// <remarks>Contains some optimizations of Out.</remarks>
7       public interface IUniLinksIOWithExtensions<TLink> : IUniLinksIO<TLink>
8       {
9           /// <remarks>
10          /// default(TLink) means nothing or null.
11          /// Single element pattern means just element (link).
12          /// OutPart(n, null) returns default(TLink).
13          /// OutPart(0, pattern) ~ Exists(link) or Search(pattern)
14          /// OutPart(1, pattern) ~ GetSource(link) or GetSource(Search(pattern))
15          /// OutPart(2, pattern) ~ GetTarget(link) or GetTarget(Search(pattern))
16          /// OutPart(3, pattern) ~ GetLinker(link) or GetLinker(Search(pattern))
17          /// OutPart(n, pattern) => For any variable length links, returns link or default(TLink).
18          ///
19          /// Outs(returns) inner contents of link, its part/parent/element/value.
20          /// </remarks>
21          TLink OutOne(ulong partType, params TLink[] pattern);
22
23          /// <remarks>OutCount() returns total links in store as array.</remarks>
24          TLink[][] OutAll(params TLink[] pattern);
25
26          /// <remarks>OutCount() returns total amount of links in store.</remarks>
27          ulong OutCount(params TLink[] pattern);
28      }
29  }
```

./Universal/IUniLinksRW.cs

```
1   using System;
2
3   // 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