LinksPlatform's Platform.Data Class Library

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

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

1.3 ./Platform.Data/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 const string DefaultMessage = "Связь с таким же значением уже существует.";
        public LinkWithSameValueAlreadyExistsException(string message) : base(message) { }
        public LinkWithSameValueAlreadyExistsException() : base(DefaultMessage) { }
    }
}
```

1.4 ./Platform.Data/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<TLinkAddress> : Exception
    {
        public const string DefaultMessage = "Достигнут лимит количества связей в хранилище.";
        public LinksLimitReachedException(string message) : base(message) { }
        public LinksLimitReachedException(TLinkAddress limit) : this(FormatMessage(limit)) { }
        public LinksLimitReachedException() : base(DefaultMessage) { }
        private static string FormatMessage(TLinkAddress limit) => $"Достигнут лимит количества связей в хранилище ({limit}).";
    }
}
```

1.5 ./Platform.Data/Hybrid.cs

```
using System;
using System.Collections.Generic;
using System.Reflection;
using System.Reflection.Emit;
using System.Runtime.CompilerServices;
using Platform.Exceptions;
using Platform.Reflection;
using Platform.Converters;
using Platform.Numbers;

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

namespace Platform.Data
{
    public struct Hybrid<TLinkAddress> : IEquatable<Hybrid<TLinkAddress>>
    {
        private static readonly EqualityComparer<TLinkAddress> _equalityComparer = EqualityComparer<TLinkAddress>.Default;
        private static readonly UncheckedSignExtendingConverter<TLinkAddress, long> _addressToInt64Converter = UncheckedSignExtendingConverter<TLinkAddress, long>.Default;
        private static readonly UncheckedConverter<TLinkAddress, ulong> _addressToUInt64Converter = UncheckedConverter<TLinkAddress, ulong>.Default;
        private static readonly UncheckedConverter<ulong, TLinkAddress> _uInt64ToAddressConverter = UncheckedConverter<ulong, TLinkAddress>.Default;
        private static readonly Func<object, TLinkAddress> _unboxAbsAndConvert = CompileUnboxAbsAndConvertDelegate();
        private static readonly Func<object, TLinkAddress> _unboxAbsNegateAndConvert = CompileUnboxAbsNegateAndConvertDelegate();

        public static readonly ulong HalfOfNumberValuesRange = _addressToUInt64Converter.Convert(NumericType<TLinkAddress>.MaxValue) / 2;
        public static readonly TLinkAddress ExternalZero = _uInt64ToAddressConverter.Convert(HalfOfNumberValuesRange + 1UL);

        public readonly TLinkAddress Value;

        public bool IsNothing
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get => _equalityComparer.Equals(Value, ExternalZero) || SignedValue == 0;
        }

        public bool IsInternal
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get => SignedValue > 0;
        }

        public bool IsExternal
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get => _equalityComparer.Equals(Value, ExternalZero) || SignedValue < 0;
        }

        public long SignedValue
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get => _addressToInt64Converter.Convert(Value);
        }

        public long AbsoluteValue
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get => _equalityComparer.Equals(Value, ExternalZero) ? 0 : Platform.Numbers.Math.Abs(SignedValue);
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public Hybrid(TLinkAddress value)
        {
            Ensure.OnDebug.IsUnsignedInteger<TLinkAddress>();
            Value = value;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public Hybrid(TLinkAddress value, bool isExternal)
        {
            if (_equalityComparer.Equals(value, default) && isExternal)
```

```
            {
                Value = ExternalZero;
            }
            else
            {
                if (isExternal)
                {
                    Value = Math<TLinkAddress>.Negate(value);
                }
                else
                {
                    Value = value;
                }
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public Hybrid(object value) => Value =
        ↪  To.UnsignedAs<TLinkAddress>(Convert.ChangeType(value,
        ↪  NumericType<TLinkAddress>.SignedVersion));

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public Hybrid(object value, bool isExternal)
        {
            if (IsDefault(value) && isExternal)
            {
                Value = ExternalZero;
            }
            else
            {
                if (isExternal)
                {
                    Value = _unboxAbsNegateAndConvert(value);
                }
                else
                {
                    Value = _unboxAbsAndConvert(value);
                }
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static implicit operator Hybrid<TLinkAddress>(TLinkAddress integer) => new
        ↪  Hybrid<TLinkAddress>(integer);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static explicit operator Hybrid<TLinkAddress>(ulong integer) => new
        ↪  Hybrid<TLinkAddress>(integer);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static explicit operator Hybrid<TLinkAddress>(long integer) => new
        ↪  Hybrid<TLinkAddress>(integer);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static explicit operator Hybrid<TLinkAddress>(uint integer) => new
        ↪  Hybrid<TLinkAddress>(integer);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static explicit operator Hybrid<TLinkAddress>(int integer) => new
        ↪  Hybrid<TLinkAddress>(integer);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static explicit operator Hybrid<TLinkAddress>(ushort integer) => new
        ↪  Hybrid<TLinkAddress>(integer);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static explicit operator Hybrid<TLinkAddress>(short integer) => new
        ↪  Hybrid<TLinkAddress>(integer);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static explicit operator Hybrid<TLinkAddress>(byte integer) => new
        ↪  Hybrid<TLinkAddress>(integer);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static explicit operator Hybrid<TLinkAddress>(sbyte integer) => new
        ↪  Hybrid<TLinkAddress>(integer);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
137         public static implicit operator TLinkAddress(Hybrid<TLinkAddress> hybrid) =>
            ↪ hybrid.Value;
138
139         [MethodImpl(MethodImplOptions.AggressiveInlining)]
140         public static explicit operator ulong(Hybrid<TLinkAddress> hybrid) =>
            ↪ CheckedConverter<TLinkAddress, ulong>.Default.Convert(hybrid.Value);
141
142         [MethodImpl(MethodImplOptions.AggressiveInlining)]
143         public static explicit operator long(Hybrid<TLinkAddress> hybrid) =>
            ↪ hybrid.AbsoluteValue;
144
145         [MethodImpl(MethodImplOptions.AggressiveInlining)]
146         public static explicit operator uint(Hybrid<TLinkAddress> hybrid) =>
            ↪ CheckedConverter<TLinkAddress, uint>.Default.Convert(hybrid.Value);
147
148         [MethodImpl(MethodImplOptions.AggressiveInlining)]
149         public static explicit operator int(Hybrid<TLinkAddress> hybrid) =>
            ↪ (int)hybrid.AbsoluteValue;
150
151         [MethodImpl(MethodImplOptions.AggressiveInlining)]
152         public static explicit operator ushort(Hybrid<TLinkAddress> hybrid) =>
            ↪ CheckedConverter<TLinkAddress, ushort>.Default.Convert(hybrid.Value);
153
154         [MethodImpl(MethodImplOptions.AggressiveInlining)]
155         public static explicit operator short(Hybrid<TLinkAddress> hybrid) =>
            ↪ (short)hybrid.AbsoluteValue;
156
157         [MethodImpl(MethodImplOptions.AggressiveInlining)]
158         public static explicit operator byte(Hybrid<TLinkAddress> hybrid) =>
            ↪ CheckedConverter<TLinkAddress, byte>.Default.Convert(hybrid.Value);
159
160         [MethodImpl(MethodImplOptions.AggressiveInlining)]
161         public static explicit operator sbyte(Hybrid<TLinkAddress> hybrid) =>
            ↪ (sbyte)hybrid.AbsoluteValue;
162
163         [MethodImpl(MethodImplOptions.AggressiveInlining)]
164         public override string ToString() => IsExternal ? $"<{AbsoluteValue}>" :
            ↪ Value.ToString();
165
166         [MethodImpl(MethodImplOptions.AggressiveInlining)]
167         public bool Equals(Hybrid<TLinkAddress> other) => _equalityComparer.Equals(Value,
            ↪ other.Value);
168
169         [MethodImpl(MethodImplOptions.AggressiveInlining)]
170         public override bool Equals(object obj) => obj is Hybrid<TLinkAddress> hybrid ?
            ↪ Equals(hybrid) : false;
171
172         [MethodImpl(MethodImplOptions.AggressiveInlining)]
173         public override int GetHashCode() => Value.GetHashCode();
174
175         [MethodImpl(MethodImplOptions.AggressiveInlining)]
176         public static bool operator ==(Hybrid<TLinkAddress> left, Hybrid<TLinkAddress> right) =>
            ↪ left.Equals(right);
177
178         [MethodImpl(MethodImplOptions.AggressiveInlining)]
179         public static bool operator !=(Hybrid<TLinkAddress> left, Hybrid<TLinkAddress> right) =>
            ↪ !(left == right);
180
181         private static bool IsDefault(object value)
182         {
183             if (value == null)
184             {
185                 return true;
186             }
187             var type = value.GetType();
188             return type.IsValueType ? value.Equals(Activator.CreateInstance(type)) : false;
189         }
190
191         private static Func<object, TLinkAddress> CompileUnboxAbsNegateAndConvertDelegate()
192         {
193             return DelegateHelpers.Compile<Func<object, TLinkAddress>>(emiter =>
194             {
195                 Ensure.Always.IsUnsignedInteger<TLinkAddress>();
196                 emiter.LoadArgument(0);
197                 var signedVersion = NumericType<TLinkAddress>.SignedVersion;
198                 var signedVersionField =
                    ↪ typeof(NumericType<TLinkAddress>).GetTypeInfo().GetField("SignedVersion",
                    ↪ BindingFlags.Static | BindingFlags.Public);
```

```
199                     emiter.Emit(OpCodes.Ldsfld, signedVersionField);
200                     var changeTypeMethod = typeof(Convert).GetTypeInfo().GetMethod("ChangeType",
                        ↪  Types<object, Type>.Array);
201                     emiter.Call(changeTypeMethod);
202                     emiter.UnboxValue(signedVersion);
203                     var absMethod = typeof(System.Math).GetTypeInfo().GetMethod("Abs", new[] {
                        ↪  signedVersion });
204                     emiter.Call(absMethod);
205                     var negateMethod = typeof(Platform.Numbers.Math).GetTypeInfo().GetMethod("Negate
                        ↪  ").MakeGenericMethod(signedVersion);
206                     emiter.Call(negateMethod);
207                     var unsignedMethod = typeof(To).GetTypeInfo().GetMethod("Unsigned", new[] {
                        ↪  signedVersion });
208                     emiter.Call(unsignedMethod);
209                     emiter.Return();
210                 });
211             }
212
213             private static Func<object, TLinkAddress> CompileUnboxAbsAndConvertDelegate()
214             {
215                 return DelegateHelpers.Compile<Func<object, TLinkAddress>>(emiter =>
216                 {
217                     Ensure.Always.IsUnsignedInteger<TLinkAddress>();
218                     emiter.LoadArgument(0);
219                     var signedVersion = NumericType<TLinkAddress>.SignedVersion;
220                     var signedVersionField =
                        ↪  typeof(NumericType<TLinkAddress>).GetTypeInfo().GetField("SignedVersion",
                        ↪  BindingFlags.Static | BindingFlags.Public);
221                     emiter.Emit(OpCodes.Ldsfld, signedVersionField);
222                     var changeTypeMethod = typeof(Convert).GetTypeInfo().GetMethod("ChangeType",
                        ↪  Types<object, Type>.Array);
223                     emiter.Call(changeTypeMethod);
224                     emiter.UnboxValue(signedVersion);
225                     var absMethod = typeof(System.Math).GetTypeInfo().GetMethod("Abs", new[] {
                        ↪  signedVersion });
226                     emiter.Call(absMethod);
227                     var unsignedMethod = typeof(To).GetTypeInfo().GetMethod("Unsigned", new[] {
                        ↪  signedVersion });
228                     emiter.Call(unsignedMethod);
229                     emiter.Return();
230                 });
231             }
232         }
233     }
```

1.6 ./Platform.Data/ILinks.cs

```
1   using System;
2   using System.Collections.Generic;
3
4   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6   namespace Platform.Data
7   {
8       /// <summary>
9       /// Представляет интерфейс для работы с данными в формате Links (хранилища взаимосвязей).
10      /// </summary>
11      /// <remarks>
12      /// Этот интерфейс в данный момент не зависит от размера содержимого связи, а значит
            ↪  подходит как для дуплетов, так и для триплетов и т.п.
13      /// Возможно этот интерфейс подходит даже для Sequences.
14      /// </remarks>
15      public interface ILinks<TLinkAddress, TConstants>
16          where TConstants : LinksConstants<TLinkAddress>
17      {
18          #region Constants
19
20          /// <summary>
21          /// Возвращает набор констант, который необходим для эффективной коммуникации с методами
                ↪  этого интерфейса.
22          /// Эти константы не меняются с момента создания точки доступа к хранилищу.
23          /// </summary>
24          TConstants Constants { get; }
25
26          #endregion
27
28          #region Read
29
30          /// <summary>
```

```
31          /// Подсчитывает и возвращает общее число связей находящихся в хранилище,
            ↪ соответствующих указанным ограничениям.
32          /// </summary>
33          /// <param name="restriction">Ограничения на содержимое связей.</param>
34          /// <returns>Общее число связей находящихся в хранилище, соответствующих указанным
            ↪ ограничениям.</returns>
35          TLinkAddress Count(IList<TLinkAddress> restriction);
36
37          /// <summary>
38          /// Выполняет проход по всем связям, соответствующим шаблону, вызывая обработчик
            ↪ (handler) для каждой подходящей связи.
39          /// </summary>
40          /// <param name="handler">Обработчик каждой подходящей связи.</param>
41          /// <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>
42          /// <returns>True, в случае если проход по связям не был прерван и False в обратном
            ↪ случае.</returns>
43          TLinkAddress Each(Func<IList<TLinkAddress>, TLinkAddress> handler, IList<TLinkAddress>
            ↪ restrictions);
44
45          #endregion
46
47          #region Write
48
49          /// <summary>
50          /// Создаёт связь.
51          /// </summary>
52          /// <returns>Индекс созданной связи.</returns>
53          TLinkAddress Create(IList<TLinkAddress> restrictions); // TODO: Возможно всегда нужно
            ↪ принимать restrictions, возможно и возвращать связь нужно целиком.
54
55          /// <summary>
56          /// Обновляет связь с указанными restrictions[Constants.IndexPart] в адресом связи
57          /// на связь с указанным новым содержимым.
58          /// </summary>
59          /// <param name="restrictions">
60          /// Ограничения на содержимое связей.
61          /// Предполагается, что будет указан индекс связи (в restrictions[Constants.IndexPart])
            ↪ и далее за ним будет следовать содержимое связи.
62          /// Каждое ограничение может иметь значения: Constants.Null - 0-я связь, обозначающая
            ↪ ссылку на пустоту,
63          /// Constants.Itself - требование установить ссылку на себя, 1..∞ конкретный индекс
            ↪ другой связи.
64          /// </param>
65          /// <param name="substitution"></param>
66          /// <returns>Индекс обновлённой связи.</returns>
67          TLinkAddress Update(IList<TLinkAddress> restrictions, IList<TLinkAddress> substitution);
            ↪ // TODO: Возможно и возвращать связь нужно целиком.
68
69          /// <summary>Удаляет связь с указанным индексом.</summary>
70          void Delete(IList<TLinkAddress> restrictions); // TODO: Возможно всегда нужно принимать
            ↪ restrictions, а так же возвращать удалённую связь, если удаление было реально
            ↪ выполнено, и Null, если нет.
71
72          #endregion
73      }
74  }
```

1.7 ./Platform.Data/ILinksExtensions.cs

```
1   using System;
2   using System.Collections.Generic;
3   using System.Runtime.CompilerServices;
4   using Platform.Setters;
5   using Platform.Data.Exceptions;
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 ILinksExtensions
12      {
13          [MethodImpl(MethodImplOptions.AggressiveInlining)]
14          public static TLinkAddress Count<TLinkAddress, TConstants>(this ILinks<TLinkAddress,
            ↪ TConstants> links, params TLinkAddress[] restrictions)
15              where TConstants : LinksConstants<TLinkAddress>
16              => links.Count(restrictions);
17
18          /// <summary>
```

```
19          /// Возвращает значение, определяющее существует ли связь с указанным индексом в
            ↪  хранилище связей.
20          /// </summary>
21          /// <param name="links">Хранилище связей.</param>
22          /// <param name="link">Индекс проверяемой на существование связи.</param>
23          /// <returns>Значение, определяющее существует ли связь.</returns>
24          [MethodImpl(MethodImplOptions.AggressiveInlining)]
25          public static bool Exists<TLinkAddress, TConstants>(this ILinks<TLinkAddress,
            ↪  TConstants> links, TLinkAddress link)
26              where TConstants : LinksConstants<TLinkAddress>
27          {
28              var constants = links.Constants;
29              return constants.IsExternalReference(link) || (constants.IsInternalReference(link)
                ↪  && Comparer<TLinkAddress>.Default.Compare(links.Count(new
                ↪  LinkAddress<TLinkAddress>(link)), default) > 0);
30          }
31
32          /// <param name="links">Хранилище связей.</param>
33          /// <param name="link">Индекс проверяемой на существование связи.</param>
34          /// <remarks>
35          /// TODO: May be move to EnsureExtensions or make it both there and here
36          /// </remarks>
37          [MethodImpl(MethodImplOptions.AggressiveInlining)]
38          public static void EnsureLinkExists<TLinkAddress, TConstants>(this ILinks<TLinkAddress,
            ↪  TConstants> links, TLinkAddress link)
39              where TConstants : LinksConstants<TLinkAddress>
40          {
41              if (!links.Exists(link))
42              {
43                  throw new ArgumentLinkDoesNotExistsException<TLinkAddress>(link);
44              }
45          }
46
47          /// <param name="links">Хранилище связей.</param>
48          /// <param name="link">Индекс проверяемой на существование связи.</param>
49          /// <param name="argumentName">Имя аргумента, в который передаётся индекс связи.</param>
50          [MethodImpl(MethodImplOptions.AggressiveInlining)]
51          public static void EnsureLinkExists<TLinkAddress, TConstants>(this ILinks<TLinkAddress,
            ↪  TConstants> links, TLinkAddress link, string argumentName)
52              where TConstants : LinksConstants<TLinkAddress>
53          {
54              if (!links.Exists(link))
55              {
56                  throw new ArgumentLinkDoesNotExistsException<TLinkAddress>(link, argumentName);
57              }
58          }
59
60          /// <summary>
61          /// Выполняет проход по всем связям, соответствующим шаблону, вызывая обработчик
            ↪  (handler) для каждой подходящей связи.
62          /// </summary>
63          /// <param name="links">Хранилище связей.</param>
64          /// <param name="handler">Обработчик каждой подходящей связи.</param>
65          /// <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>
66          /// <returns>True, в случае если проход по связям не был прерван и False в обратном
            ↪  случае.</returns>
67          [MethodImpl(MethodImplOptions.AggressiveInlining)]
68          public static TLinkAddress Each<TLinkAddress, TConstants>(this ILinks<TLinkAddress,
            ↪  TConstants> links, Func<IList<TLinkAddress>, TLinkAddress> handler, params
            ↪  TLinkAddress[] restrictions)
69              where TConstants : LinksConstants<TLinkAddress>
70              => links.Each(handler, restrictions);
71
72          /// <summary>
73          /// Возвращает части-значения для связи с указанным индексом.
74          /// </summary>
75          /// <param name="links">Хранилище связей.</param>
76          /// <param name="link">Индекс связи.</param>
77          /// <returns>Уникальную связь.</returns>
78          [MethodImpl(MethodImplOptions.AggressiveInlining)]
79          public static IList<TLinkAddress> GetLink<TLinkAddress, TConstants>(this
            ↪  ILinks<TLinkAddress, TConstants> links, TLinkAddress link)
80              where TConstants : LinksConstants<TLinkAddress>
81          {
82              var constants = links.Constants;
83              if (constants.IsExternalReference(link))
```

```
84              {
85                  return new Point<TLinkAddress>(link, constants.TargetPart + 1);
86              }
87              var linkPartsSetter = new Setter<IList<TLinkAddress>,
                ↪  TLinkAddress>(constants.Continue, constants.Break);
88              links.Each(linkPartsSetter.SetAndReturnTrue, link);
89              return linkPartsSetter.Result;
90          }
91
92          #region Points
93
94          /// <summary>Возвращает значение, определяющее является ли связь с указанным индексом
                ↪  точкой полностью (связью замкнутой на себе дважды).</summary>
95          /// <param name="links">Хранилище связей.</param>
96          /// <param name="link">Индекс проверяемой связи.</param>
97          /// <returns>Значение, определяющее является ли связь точкой полностью.</returns>
98          /// <remarks>
99          /// Связь точка - это связь, у которой начало (Source) и конец (Target) есть сама эта
                ↪  связь.
100         /// Но что, если точка уже есть, а нужно создать пару с таким же значением? Должны ли
                ↪  точка и пара существовать одновременно?
101         /// Или в качестве решения для точек нужно использовать 0 в качестве начала и конца, а
                ↪  сортировать по индексу в массиве связей?
102         /// Какое тогда будет значение Source и Target у точки? 0 или её индекс?
103         /// Или точка должна быть одновременно точкой и парой, а также последовательностями из
                ↪  самой себя любого размера?
104         /// Как только есть ссылка на себя, появляется этот парадокс, причём достаточно даже
                ↪  одной ссылки на себя (частичной точки).
105         /// А что если не выбирать что является точкой, пара нулей (цикл через пустоту) или
106         /// самостоятельный цикл через себя? Что если предоставить все варианты использования
                ↪  связей?
107         /// Что если разрешить и нули, а так же частичные варианты?
108         ///
109         /// Что если точка, это только в том случае когда link.Source == link &amp;amp;&amp;
                ↪  link.Target == link , т.е. дважды ссылка на себя.
110         /// А пара это тогда, когда link.Source == link.Target &amp;amp;&amp; link.Source != link ,
                ↪  т.е. ссылка не на себя а во вне.
111         ///
112         /// Тогда если у нас уже создана пара, но нам нужна точка, мы можем используя
                ↪  промежуточную связь,
113         /// например "DoubletOf" обозначить что является точно парой, а что точно точкой.
114         /// И наоборот этот же метод поможет, если уже существует точка, но нам нужна пара.
115         /// </remarks>
116         public static bool IsFullPoint<TLinkAddress, TConstants>(this ILinks<TLinkAddress,
                ↪  TConstants> links, TLinkAddress link)
117             where TConstants : LinksConstants<TLinkAddress>
118         {
119             if (links.Constants.IsExternalReference(link))
120             {
121                 return true;
122             }
123             links.EnsureLinkExists(link);
124             return Point<TLinkAddress>.IsFullPoint(links.GetLink(link));
125         }
126
127         /// <summary>Возвращает значение, определяющее является ли связь с указанным индексом
                ↪  точкой частично (связью замкнутой на себе как минимум один раз).</summary>
128         /// <param name="links">Хранилище связей.</param>
129         /// <param name="link">Индекс проверяемой связи.</param>
130         /// <returns>Значение, определяющее является ли связь точкой частично.</returns>
131         /// <remarks>
132         /// Достаточно любой одной ссылки на себя.
133         /// Также в будущем можно будет проверять и всех родителей, чтобы проверить есть ли
                ↪  ссылки на себя (на эту связь).
134         /// </remarks>
135         public static bool IsPartialPoint<TLinkAddress, TConstants>(this ILinks<TLinkAddress,
                ↪  TConstants> links, TLinkAddress link)
136             where TConstants : LinksConstants<TLinkAddress>
137         {
138             if (links.Constants.IsExternalReference(link))
139             {
140                 return true;
141             }
142             links.EnsureLinkExists(link);
143             return Point<TLinkAddress>.IsPartialPoint(links.GetLink(link));
144         }
145
146         #endregion
```

```
    }
}
```

1.8 ./Platform.Data/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<TLinkAddress, TLinks, TConstants> :
    ↪  ISynchronized<TLinks>, ILinks<TLinkAddress, TConstants>
        where TLinks : ILinks<TLinkAddress, TConstants>
        where TConstants : LinksConstants<TLinkAddress>
    {
    }
}
```

1.9 ./Platform.Data/LinkAddress.cs

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

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

namespace Platform.Data
{
    public class LinkAddress<TLinkAddress> : IEquatable<LinkAddress<TLinkAddress>>,
    ↪  IList<TLinkAddress>
    {
        private static readonly EqualityComparer<TLinkAddress> _equalityComparer =
        ↪  EqualityComparer<TLinkAddress>.Default;

        public TLinkAddress Index
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
        }

        public TLinkAddress this[int index]
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get
            {
                if (index == 0)
                {
                    return Index;
                }
                else
                {
                    throw new IndexOutOfRangeException();
                }
            }
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            set => throw new NotSupportedException();
        }

        public int Count => 1;

        public bool IsReadOnly => true;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LinkAddress(TLinkAddress index) => Index = index;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void Add(TLinkAddress item) => throw new NotSupportedException();

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void Clear() => throw new NotSupportedException();

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public virtual bool Contains(TLinkAddress item) => _equalityComparer.Equals(item, Index)
        ↪  ? true : false;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void CopyTo(TLinkAddress[] array, int arrayIndex) => array[arrayIndex] = Index;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public IEnumerator<TLinkAddress> GetEnumerator()
```

```
59          {
60              yield return Index;
61          }
62
63          [MethodImpl(MethodImplOptions.AggressiveInlining)]
64          public virtual int IndexOf(TLinkAddress item) => _equalityComparer.Equals(item, Index) ?
            ↪  0 : -1;
65
66          [MethodImpl(MethodImplOptions.AggressiveInlining)]
67          public void Insert(int index, TLinkAddress item) => throw new NotSupportedException();
68
69          [MethodImpl(MethodImplOptions.AggressiveInlining)]
70          public bool Remove(TLinkAddress item) => throw new NotSupportedException();
71
72          [MethodImpl(MethodImplOptions.AggressiveInlining)]
73          public void RemoveAt(int index) => throw new NotSupportedException();
74
75          [MethodImpl(MethodImplOptions.AggressiveInlining)]
76          IEnumerator IEnumerable.GetEnumerator()
77          {
78              yield return Index;
79          }
80
81          [MethodImpl(MethodImplOptions.AggressiveInlining)]
82          public virtual bool Equals(LinkAddress<TLinkAddress> other) => other == null ? false :
            ↪  _equalityComparer.Equals(Index, other.Index);
83
84          [MethodImpl(MethodImplOptions.AggressiveInlining)]
85          public static implicit operator TLinkAddress(LinkAddress<TLinkAddress> linkAddress) =>
            ↪  linkAddress.Index;
86
87          [MethodImpl(MethodImplOptions.AggressiveInlining)]
88          public static implicit operator LinkAddress<TLinkAddress>(TLinkAddress linkAddress) =>
            ↪  new LinkAddress<TLinkAddress>(linkAddress);
89
90          [MethodImpl(MethodImplOptions.AggressiveInlining)]
91          public override bool Equals(object obj) => obj is LinkAddress<TLinkAddress> linkAddress
            ↪  ? Equals(linkAddress) : false;
92
93          [MethodImpl(MethodImplOptions.AggressiveInlining)]
94          public override int GetHashCode() => Index.GetHashCode();
95
96          [MethodImpl(MethodImplOptions.AggressiveInlining)]
97          public override string ToString() => Index.ToString();
98
99          [MethodImpl(MethodImplOptions.AggressiveInlining)]
100         public static bool operator ==(LinkAddress<TLinkAddress> left, LinkAddress<TLinkAddress>
            ↪  right)
101         {
102             if (left == null && right == null)
103             {
104                 return true;
105             }
106             if (left == null)
107             {
108                 return false;
109             }
110             return left.Equals(right);
111         }
112
113         [MethodImpl(MethodImplOptions.AggressiveInlining)]
114         public static bool operator !=(LinkAddress<TLinkAddress> left, LinkAddress<TLinkAddress>
            ↪  right) => !(left == right);
115     }
116 }
```

1.10 ./Platform.Data/LinksConstants.cs

```
1  using Platform.Ranges;
2  using Platform.Reflection;
3  using Platform.Converters;
4  using Platform.Numbers;
5
6  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8  namespace Platform.Data
9  {
10     public class LinksConstants<TLinkAddress>
11     {
12         public const int DefaultTargetPart = 2;
```

```
13
14          private static readonly TLinkAddress _one = Arithmetic<TLinkAddress>.Increment(default);
15          private static readonly UncheckedConverter<ulong, TLinkAddress>
            ↪  _uInt64ToAddressConverter = UncheckedConverter<ulong, TLinkAddress>.Default;
16
17          #region Link parts
18
19          /// <summary>Возвращает индекс части, которая отвечает за индекс (адрес, идентификатор)
            ↪  самой связи.</summary>
20          public int IndexPart { get; }
21
22          /// <summary>Возвращает индекс части, которая отвечает за ссылку на связь-начало (первая
            ↪  часть-значение).</summary>
23          public int SourcePart { get; }
24
25          /// <summary>Возвращает индекс части, которая отвечает за ссылку на связь-конец
            ↪  (последняя часть-значение).</summary>
26          public int TargetPart { get; }
27
28          #endregion
29
30          #region Flow control
31
32          /// <summary>Возвращает значение, обозначающее продолжение прохода по связям.</summary>
33          /// <remarks>Используется в функции обработчике, который передаётся в функцию
            ↪  Each.</remarks>
34          public TLinkAddress Continue { get; }
35
36          /// <summary>Возвращает значение, обозначающее пропуск в проходе по связям.</summary>
37          public TLinkAddress Skip { get; }
38
39          /// <summary>Возвращает значение, обозначающее остановку прохода по связям.</summary>
40          /// <remarks>Используется в функции обработчике, который передаётся в функцию
            ↪  Each.</remarks>
41          public TLinkAddress Break { get; }
42
43          #endregion
44
45          #region Special symbols
46
47          /// <summary>Возвращает значение, обозначающее отсутствие связи.</summary>
48          public TLinkAddress Null { get; }
49
50          /// <summary>Возвращает значение, обозначающее любую связь.</summary>
51          /// <remarks>Возможно нужно зарезервировать отдельное значение, тогда можно будет
            ↪  создавать все варианты последовательностей в функции Create.</remarks>
52          public TLinkAddress Any { get; }
53
54          /// <summary>Возвращает значение, обозначающее связь-ссылку на саму связь.</summary>
55          public TLinkAddress Itself { get; }
56
57          #endregion
58
59          #region References
60
61          /// <summary>Возвращает диапазон возможных индексов для внутренних связей (внутренних
            ↪  ссылок).</summary>
62          public Range<TLinkAddress> InternalReferencesRange { get; }
63
64          /// <summary>Возвращает диапазон возможных индексов для внешних связей (внешних
            ↪  ссылок).</summary>
65          public Range<TLinkAddress>? ExternalReferencesRange { get; }
66
67          #endregion
68
69          public LinksConstants(int targetPart, Range<TLinkAddress>
            ↪  possibleInternalReferencesRange, Range<TLinkAddress>?
            ↪  possibleExternalReferencesRange)
70          {
71              IndexPart = 0;
72              SourcePart = 1;
73              TargetPart = targetPart;
74              Null = default;
75              Break = default;
76              var currentInternalReferenceIndex = possibleInternalReferencesRange.Maximum;
77              Continue = currentInternalReferenceIndex;
78              Decrement(ref currentInternalReferenceIndex);
79              Skip = currentInternalReferenceIndex;
80              Decrement(ref currentInternalReferenceIndex);
81              Any = currentInternalReferenceIndex;
```

```
82              Decrement(ref currentInternalReferenceIndex);
83              Itself = currentInternalReferenceIndex;
84              Decrement(ref currentInternalReferenceIndex);
85              InternalReferencesRange = (possibleInternalReferencesRange.Minimum,
                ↪  currentInternalReferenceIndex);
86              ExternalReferencesRange = possibleExternalReferencesRange;
87          }
88
89          public LinksConstants(int targetPart, bool enableExternalReferencesSupport) :
            ↪  this(targetPart, GetDefaultInternalReferencesRange(enableExternalReferencesSupport),
            ↪  GetDefaultExternalReferencesRange(enableExternalReferencesSupport)) {}
90
91          public LinksConstants(Range<TLinkAddress> possibleInternalReferencesRange,
            ↪  Range<TLinkAddress>? possibleExternalReferencesRange) : this(DefaultTargetPart,
            ↪  possibleInternalReferencesRange, possibleExternalReferencesRange) { }
92
93          public LinksConstants(bool enableExternalReferencesSupport) :
            ↪  this(GetDefaultInternalReferencesRange(enableExternalReferencesSupport),
            ↪  GetDefaultExternalReferencesRange(enableExternalReferencesSupport)) { }
94
95          public LinksConstants(int targetPart, Range<TLinkAddress>
            ↪  possibleInternalReferencesRange) : this(targetPart, possibleInternalReferencesRange,
            ↪  null) { }
96
97          public LinksConstants(Range<TLinkAddress> possibleInternalReferencesRange) :
            ↪  this(DefaultTargetPart, possibleInternalReferencesRange, null) { }
98
99          public LinksConstants() : this(DefaultTargetPart, enableExternalReferencesSupport:
            ↪  false) { }
100
101         public static Range<TLinkAddress> GetDefaultInternalReferencesRange(bool
            ↪  enableExternalReferencesSupport)
102         {
103             if (enableExternalReferencesSupport)
104             {
105                 return (_one, _uInt64ToAddressConverter.Convert(Hybrid<TLinkAddress>.HalfOfNumbe
                    ↪  rValuesRange));
106             }
107             else
108             {
109                 return (_one, NumericType<TLinkAddress>.MaxValue);
110             }
111         }
112
113         public static Range<TLinkAddress>? GetDefaultExternalReferencesRange(bool
            ↪  enableExternalReferencesSupport)
114         {
115             if (enableExternalReferencesSupport)
116             {
117                 return (_uInt64ToAddressConverter.Convert(Hybrid<TLinkAddress>.HalfOfNumberValue
                    ↪  sRange + 1UL),
                    ↪  NumericType<TLinkAddress>.MaxValue);
118             }
119             else
120             {
121                 return null;
122             }
123         }
124
125         private static void Decrement(ref TLinkAddress currentInternalReferenceIndex) =>
            ↪  currentInternalReferenceIndex = Arithmetic.Decrement(currentInternalReferenceIndex);
126     }
127 }
```

1.11 ./Platform.Data/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<TLinkAddress>(this LinksConstants<TLinkAddress>
           ↪  linksConstants, TLinkAddress address) => linksConstants.IsInternalReference(address)
           ↪  || linksConstants.IsExternalReference(address);
11
```

```
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool IsInternalReference<TLinkAddress>(this LinksConstants<TLinkAddress>
        ↪  linksConstants, TLinkAddress address) =>
        ↪  linksConstants.InternalReferencesRange.Contains(address);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool IsExternalReference<TLinkAddress>(this LinksConstants<TLinkAddress>
        ↪  linksConstants, TLinkAddress address) =>
        ↪  linksConstants.ExternalReferencesRange?.Contains(address) ?? false;
    }
}
```

1.12 ./Platform.Data/Numbers/Raw/AddressToRawNumberConverter.cs

```
using Platform.Converters;

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

namespace Platform.Data.Numbers.Raw
{
    public class AddressToRawNumberConverter<TLink> : IConverter<TLink>
    {
        public TLink Convert(TLink source) => new Hybrid<TLink>(source, isExternal: true);
    }
}
```

1.13 ./Platform.Data/Numbers/Raw/RawNumberToAddressConverter.cs

```
using Platform.Converters;

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

namespace Platform.Data.Numbers.Raw
{
    public class RawNumberToAddressConverter<TLink> : IConverter<TLink>
    {
        static private readonly UncheckedConverter<long, TLink> _converter =
        ↪  UncheckedConverter<long, TLink>.Default;

        public TLink Convert(TLink source) => _converter.Convert(new
        ↪  Hybrid<TLink>(source).AbsoluteValue);
    }
}
```

1.14 ./Platform.Data/Point.cs

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Runtime.CompilerServices;
using Platform.Exceptions;
using Platform.Ranges;
using Platform.Collections;

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

namespace Platform.Data
{
    public class Point<TLinkAddress> : IEquatable<LinkAddress<TLinkAddress>>, IList<TLinkAddress>
    {
        private static readonly EqualityComparer<TLinkAddress> _equalityComparer =
        ↪  EqualityComparer<TLinkAddress>.Default;

        public TLinkAddress Index
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
        }

        public int Size
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
        }

        public TLinkAddress this[int index]
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get
            {
                if (index < Size)
                {
                    return Index;
```

```
                }
                else
                {
                    throw new IndexOutOfRangeException();
                }
            }
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            set => throw new NotSupportedException();
        }

        public int Count => int.MaxValue;

        public bool IsReadOnly => true;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public Point(TLinkAddress index, int size)
        {
            Index = index;
            Size = size;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void Add(TLinkAddress item) => throw new NotSupportedException();

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void Clear() => throw new NotSupportedException();

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public virtual bool Contains(TLinkAddress item) => _equalityComparer.Equals(item, Index)
        ↪  ? true : false;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void CopyTo(TLinkAddress[] array, int arrayIndex) => array[arrayIndex] = Index;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public IEnumerator<TLinkAddress> GetEnumerator()
        {
            for (int i = 0; i < Size; i++)
            {
                yield return Index;
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public virtual int IndexOf(TLinkAddress item) => _equalityComparer.Equals(item, Index) ?
        ↪  0 : -1;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void Insert(int index, TLinkAddress item) => throw new NotSupportedException();

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public bool Remove(TLinkAddress item) => throw new NotSupportedException();

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void RemoveAt(int index) => throw new NotSupportedException();

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        IEnumerator IEnumerable.GetEnumerator()
        {
            for (int i = 0; i < Size; i++)
            {
                yield return Index;
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public virtual bool Equals(LinkAddress<TLinkAddress> other) => other == null ? false :
        ↪  _equalityComparer.Equals(Index, other.Index);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static implicit operator TLinkAddress(Point<TLinkAddress> linkAddress) =>
        ↪  linkAddress.Index;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public override bool Equals(object obj) => obj is Point<TLinkAddress> linkAddress ?
        ↪  Equals(linkAddress) : false;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public override int GetHashCode() => Index.GetHashCode();

```

```
112         [MethodImpl(MethodImplOptions.AggressiveInlining)]
113         public override string ToString() => Index.ToString();
114 
115         [MethodImpl(MethodImplOptions.AggressiveInlining)]
116         public static bool operator ==(Point<TLinkAddress> left, Point<TLinkAddress> right)
117         {
118             if (left == null && right == null)
119             {
120                 return true;
121             }
122             if (left == null)
123             {
124                 return false;
125             }
126             return left.Equals(right);
127         }
128 
129         [MethodImpl(MethodImplOptions.AggressiveInlining)]
130         public static bool operator !=(Point<TLinkAddress> left, Point<TLinkAddress> right) =>
            ↪ !(left == right);
131 
132         [MethodImpl(MethodImplOptions.AggressiveInlining)]
133         public static bool IsFullPoint(params TLinkAddress[] link) =>
            ↪ IsFullPoint((IList<TLinkAddress>)link);
134 
135         [MethodImpl(MethodImplOptions.AggressiveInlining)]
136         public static bool IsFullPoint(IList<TLinkAddress> link)
137         {
138             Ensure.Always.ArgumentNotEmpty(link, nameof(link));
139             Ensure.Always.ArgumentInRange(link.Count, new Range<int>(2, int.MaxValue),
                ↪ nameof(link), "Cannot determine link's pointness using only its identifier.");
140             return IsFullPointUnchecked(link);
141         }
142 
143         [MethodImpl(MethodImplOptions.AggressiveInlining)]
144         public static bool IsFullPointUnchecked(IList<TLinkAddress> link)
145         {
146             var result = true;
147             for (var i = 1; result && i < link.Count; i++)
148             {
149                 result = _equalityComparer.Equals(link[0], link[i]);
150             }
151             return result;
152         }
153 
154         [MethodImpl(MethodImplOptions.AggressiveInlining)]
155         public static bool IsPartialPoint(params TLinkAddress[] link) =>
            ↪ IsPartialPoint((IList<TLinkAddress>)link);
156 
157         [MethodImpl(MethodImplOptions.AggressiveInlining)]
158         public static bool IsPartialPoint(IList<TLinkAddress> link)
159         {
160             Ensure.Always.ArgumentNotEmpty(link, nameof(link));
161             Ensure.Always.ArgumentInRange(link.Count, new Range<int>(2, int.MaxValue),
                ↪ nameof(link), "Cannot determine link's pointness using only its identifier.");
162             return IsPartialPointUnchecked(link);
163         }
164 
165         [MethodImpl(MethodImplOptions.AggressiveInlining)]
166         public static bool IsPartialPointUnchecked(IList<TLinkAddress> link)
167         {
168             var result = false;
169             for (var i = 1; !result && i < link.Count; i++)
170             {
171                 result = _equalityComparer.Equals(link[0], link[i]);
172             }
173             return result;
174         }
175     }
176 }
```

1.15 ./Platform.Data/Sequences/ISequenceAppender.cs

```
1  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
2 
3  namespace Platform.Data.Sequences
4  {
5      public interface ISequenceAppender<TLinkAddress>
6      {
7          TLinkAddress Append(TLinkAddress sequence, TLinkAddress appendant);
```

```
    }
}
```

1.16 ./Platform.Data/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<TLinkAddress>
    {
        IEnumerable<IList<TLinkAddress>> Walk(TLinkAddress sequence);
    }
}
```

1.17 ./Platform.Data/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.
    ///
    /// 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 SequenceWalker
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void WalkRight<TLinkAddress>(TLinkAddress sequence, Func<TLinkAddress, TLinkAddress> getSource, Func<TLinkAddress, TLinkAddress> getTarget, Func<TLinkAddress, bool> isElement, Action<TLinkAddress> visit)
        {
            var stack = new Stack<TLinkAddress>();
            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<TLinkAddress>(TLinkAddress sequence, Func<TLinkAddress,
        ↪  TLinkAddress> getSource, Func<TLinkAddress, TLinkAddress> getTarget,
        ↪  Func<TLinkAddress, bool> isElement, Action<TLinkAddress> visit)
        {
            var stack = new Stack<TLinkAddress>();
            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);
                    }
                }
            }
        }
    }
}
```

1.18 ./Platform.Data/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.
    ///
    /// 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)]
```

```
24          public static bool WalkRight<TLinkAddress>(TLinkAddress sequence, Func<TLinkAddress,
            ↪ TLinkAddress> getSource, Func<TLinkAddress, TLinkAddress> getTarget,
            ↪ Func<TLinkAddress, bool> isElement, Action<TLinkAddress> enter, Action<TLinkAddress>
            ↪ exit, Func<TLinkAddress, bool> canEnter, Func<TLinkAddress, bool> visit)
25          {
26              var exited = 0;
27              var stack = new Stack<TLinkAddress>();
28              var element = sequence;
29              if (isElement(element))
30              {
31                  return visit(element);
32              }
33              while (true)
34              {
35                  if (isElement(element))
36                  {
37                      if (stack.Count == 0)
38                      {
39                          return true;
40                      }
41                      element = stack.Pop();
42                      exit(element);
43                      exited++;
44                      var source = getSource(element);
45                      var target = getTarget(element);
46                      if ((isElement(source) || (exited == 1 && !canEnter(source))) &&
                        ↪ !visit(source))
47                      {
48                          return false;
49                      }
50                      if ((isElement(target) || !canEnter(target)) && !visit(target))
51                      {
52                          return false;
53                      }
54                      element = target;
55                  }
56                  else
57                  {
58                      if (canEnter(element))
59                      {
60                          enter(element);
61                          exited = 0;
62                          stack.Push(element);
63                          element = getSource(element);
64                      }
65                      else
66                      {
67                          if (stack.Count == 0)
68                          {
69                              return true;
70                          }
71                          element = stack.Pop();
72                          exit(element);
73                          exited++;
74                          var source = getSource(element);
75                          var target = getTarget(element);
76                          if ((isElement(source) || (exited == 1 && !canEnter(source))) &&
                            ↪ !visit(source))
77                          {
78                              return false;
79                          }
80                          if ((isElement(target) || !canEnter(target)) && !visit(target))
81                          {
82                              return false;
83                          }
84                          element = target;
85                      }
86                  }
87              }
88          }
89
90          [MethodImpl(MethodImplOptions.AggressiveInlining)]
91          public static bool WalkRight<TLinkAddress>(TLinkAddress sequence, Func<TLinkAddress,
            ↪ TLinkAddress> getSource, Func<TLinkAddress, TLinkAddress> getTarget,
            ↪ Func<TLinkAddress, bool> isElement, Func<TLinkAddress, bool> visit)
92          {
93              var stack = new Stack<TLinkAddress>();
94              var element = sequence;
95              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<TLinkAddress>(TLinkAddress sequence, Func<TLinkAddress,
        ↪  TLinkAddress> getSource, Func<TLinkAddress, TLinkAddress> getTarget,
        ↪  Func<TLinkAddress, bool> isElement, Func<TLinkAddress, bool> visit)
        {
            var stack = new Stack<TLinkAddress>();
            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);
                }
            }
        }
    }
}
```

1.19 ./Platform.Data/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<TLinkAddress>
    {
        IList<IList<IList<TLinkAddress>>> Trigger(IList<TLinkAddress> condition,
        ↪  IList<TLinkAddress> substitution);
    }

    /// <remarks>Minimal sufficient universal Links API (for step by step operations).</remarks>
    public partial interface IUniLinks<TLinkAddress>
    {
        /// <returns>
        /// TLinkAddress that represents True (was finished fully) or TLinkAddress 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>
        TLinkAddress Trigger(IList<TLinkAddress> patternOrCondition, Func<IList<TLinkAddress>,
        ↪  TLinkAddress> matchHandler,
                             IList<TLinkAddress> substitution, Func<IList<TLinkAddress>,
                             ↪  IList<TLinkAddress>, TLinkAddress> substitutionHandler);

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

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

1.20 ./Platform.Data/Universal/IUniLinksCRUD.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>
    /// CRUD aliases for IUniLinks.
    /// </remarks>
    public interface IUniLinksCRUD<TLinkAddress>
    {
        TLinkAddress Read(int partType, TLinkAddress link);
        TLinkAddress Read(Func<TLinkAddress, bool> handler, IList<TLinkAddress> pattern);
        TLinkAddress Create(IList<TLinkAddress> parts);
        TLinkAddress Update(IList<TLinkAddress> before, IList<TLinkAddress> after);
        void Delete(IList<TLinkAddress> parts);
    }
}
```

1.21 ./Platform.Data/Universal/IUniLinksGS.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>
    /// Get/Set aliases for IUniLinks.
    /// </remarks>
```

```
12      public interface IUniLinksGS<TLinkAddress>
13      {
14          TLinkAddress Get(int partType, TLinkAddress link);
15          TLinkAddress Get(Func<TLinkAddress, bool> handler, IList<TLinkAddress> pattern);
16          TLinkAddress Set(IList<TLinkAddress> before, IList<TLinkAddress> after);
17      }
18  }
```

1.22 ./Platform.Data/Universal/IUniLinksIO.cs

```
1   using System;
2   using System.Collections.Generic;
3
4   // ReSharper disable TypeParameterCanBeVariant
5   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7   namespace Platform.Data.Universal
8   {
9       /// <remarks>
10      /// In/Out aliases for IUniLinks.
11      /// TLinkAddress can be any number type of any size.
12      /// </remarks>
13      public interface IUniLinksIO<TLinkAddress>
14      {
15          /// <remarks>
16          /// default(TLinkAddress) means any link.
17          /// Single element pattern means just element (link).
18          /// Handler gets array of link contents.
19          /// * link[0] is index or identifier.
20          /// * link[1] is source or first.
21          /// * link[2] is target or second.
22          /// * link[3] is linker or third.
23          /// * link[n] is nth part/parent/element/value
24          /// of link (if variable length links used).
25          ///
26          /// Stops and returns false if handler return false.
27          ///
28          /// Acts as Each, Foreach, Select, Search, Match &amp;amp; ...
29          ///
30          /// Handles all links in store if pattern/restrictions is not defined.
31          /// </remarks>
32          bool Out(Func<IList<TLinkAddress>, bool> handler, IList<TLinkAddress> pattern);
33
34          /// <remarks>
35          /// default(TLinkAddress) means itself.
36          /// Equivalent to:
37          /// * creation if before == null
38          /// * deletion if after == null
39          /// * update if before != null &amp;amp;&amp;amp; after != null
40          /// * default(TLinkAddress) if before == null &amp;amp;&amp;amp; after == null
41          ///
42          /// Possible interpretation
43          /// * In(null, new[] { }) creates point (link that points to itself using minimum number
            ↪  of parts).
44          /// * In(new[] { 4 }, null) deletes 4th link.
45          /// * In(new[] { 4 }, new [] { 5 }) delete 5th link if it exists and moves 4th link to
            ↪  5th index.
46          /// * 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.
47          /// ...
48          /// </remarks>
49          TLinkAddress In(IList<TLinkAddress> before, IList<TLinkAddress> after);
50      }
51  }
```

1.23 ./Platform.Data/Universal/IUniLinksIOWithExtensions.cs

```
1   // ReSharper disable TypeParameterCanBeVariant
2   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
3
4   using System.Collections.Generic;
5
6   namespace Platform.Data.Universal
7   {
8       /// <remarks>Contains some optimizations of Out.</remarks>
9       public interface IUniLinksIOWithExtensions<TLinkAddress> : IUniLinksIO<TLinkAddress>
10      {
11          /// <remarks>
12          /// default(TLinkAddress) means nothing or null.
13          /// Single element pattern means just element (link).
```

```
        /// OutPart(n, null) returns default(TLinkAddress).
        /// 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) ~ GeTLinkAddresser(link) or GeTLinkAddresser(Search(pattern))
        /// OutPart(n, pattern) => For any variable length links, returns link or default(TLinkAddress).
        ///
        /// Outs(returns) inner contents of link, its part/parent/element/value.
        /// </remarks>
        TLinkAddress OutOne(int partType, IList<TLinkAddress> pattern);

        /// <remarks>OutCount() returns total links in store as array.</remarks>
        IList<IList<TLinkAddress>> OutAll(IList<TLinkAddress> pattern);

        /// <remarks>OutCount() returns total amount of links in store.</remarks>
        ulong OutCount(IList<TLinkAddress> pattern);
    }
}
```

1.24 ./Platform.Data/Universal/IUniLinksRW.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>
    /// Read/Write aliases for IUniLinks.
    /// </remarks>
    public interface IUniLinksRW<TLinkAddress>
    {
        TLinkAddress Read(int partType, TLinkAddress link);
        bool Read(Func<TLinkAddress, bool> handler, IList<TLinkAddress> pattern);
        TLinkAddress Write(IList<TLinkAddress> before, IList<TLinkAddress> after);
    }
}
```

1.25 ./Platform.Data.Tests/LinksConstantsTests.cs

```
using Xunit;
using Platform.Reflection;
using Platform.Converters;
using Platform.Numbers;

namespace Platform.Data.Tests
{
    public static class LinksConstantsTests
    {
        [Fact]
        public static void ExternalReferencesTest()
        {
            TestExternalReferences<ulong, long>();
            TestExternalReferences<uint, int>();
            TestExternalReferences<ushort, short>();
            TestExternalReferences<byte, sbyte>();
        }

        private static void TestExternalReferences<TUnsigned, TSigned>()
        {
            var unsingedOne = Arithmetic.Increment(default(TUnsigned));
            var converter = UncheckedConverter<TSigned, TUnsigned>.Default;
            var half = converter.Convert(NumericType<TSigned>.MaxValue);
            LinksConstants<TUnsigned> constants = new LinksConstants<TUnsigned>((unsingedOne, half), (Arithmetic.Add(half, unsingedOne), NumericType<TUnsigned>.MaxValue));

            var minimum = new Hybrid<TUnsigned>(default, isExternal: true);
            var maximum = new Hybrid<TUnsigned>(half, isExternal: true);

            Assert.True(constants.IsExternalReference(minimum));
            Assert.True(minimum.IsExternal);
            Assert.False(minimum.IsInternal);
            Assert.True(constants.IsExternalReference(maximum));
            Assert.True(maximum.IsExternal);
            Assert.False(maximum.IsInternal);
        }
    }
}
```

Index