1.1 ./Platform.Converters/CachingConverterDecorator.cs

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

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

namespace Platform.Converters
{
    public class CachingConverterDecorator<TSource, TTarget> : IConverter<TSource, TTarget>
    {
        private readonly IConverter<TSource, TTarget> _baseConverter;
        private readonly IDictionary<TSource, TTarget> _cache;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public CachingConverterDecorator(IConverter<TSource, TTarget> baseConverter,
        ↪  IDictionary<TSource, TTarget> cache) => (_baseConverter, _cache) = (baseConverter,
        ↪  cache);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public CachingConverterDecorator(IConverter<TSource, TTarget> baseConverter) :
        ↪  this(baseConverter, new Dictionary<TSource, TTarget>()) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TTarget Convert(TSource source)
        {
            if (!_cache.TryGetValue(source, out TTarget value))
            {
                value = _baseConverter.Convert(source);
                _cache.Add(source, value);
            }
            return value;
        }
    }
}
```

1.2 ./Platform.Converters/CheckedConverter.cs

```
using System;
using System.Reflection;
using System.Reflection.Emit;
using System.Runtime.CompilerServices;
using Platform.Reflection;

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

namespace Platform.Converters
{
    public abstract class CheckedConverter<TSource, TTarget> : IConverter<TSource, TTarget>
    {
        public static CheckedConverter<TSource, TTarget> Default
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        static CheckedConverter()
        {
            AssemblyName assemblyName = new AssemblyName(GetNewName());
            var assembly = AssemblyBuilder.DefineDynamicAssembly(assemblyName,
            ↪  AssemblyBuilderAccess.Run);
            var module = assembly.DefineDynamicModule(GetNewName());
            var type = module.DefineType(GetNewName(), TypeAttributes.Public |
            ↪  TypeAttributes.Class | TypeAttributes.Sealed, typeof(CheckedConverter<TSource,
            ↪  TTarget>));
            type.EmitVirtualMethod<Converter<TSource, TTarget>>("Convert", il =>
            {
                il.LoadArgument(1);
                if (typeof(TSource) != typeof(TTarget))
                {
                    il.CheckedConvert<TSource, TTarget>();
                }
                il.Return();
            });
            var typeInfo = type.CreateTypeInfo();
            Default = (CheckedConverter<TSource, TTarget>)Activator.CreateInstance(typeInfo);
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
        private static string GetNewName() => Guid.NewGuid().ToString("N");

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public abstract TTarget Convert(TSource source);
    }
}
```

1.3 ./Platform.Converters/IConverter[TSource, TTarget].cs

```
namespace Platform.Converters
{
    /// <summary>
    /// <para>Defines a converter between two types (TSource and TTarget).</para>
    /// <para>Определяет конвертер между двумя типами (исходным TSource и целевым TTarget).</para>
    /// </summary>
    /// <typeparam name="TSource"><para>Source type of conversion.</para><para>Исходный тип конверсии.</para></typeparam>
    /// <typeparam name="TTarget"><para>Target type of conversion.</para><para>Целевой тип конверсии.</para></typeparam>
    public interface IConverter<in TSource, out TTarget>
    {
        /// <summary>
        /// <para>Converts the value of the source type (TSource) to the value of the target type.</para>
        /// <para>Конвертирует значение исходного типа (TSource) в значение целевого типа.</para>
        /// </summary>
        /// <param name="source"><para>The source type value (TSource).</para><para>Значение исходного типа (TSource).</para></param>
        /// <returns><para>The value is converted to the target type (TTarget).</para><para>Значение ковертированное в целевой тип (TTarget).</para></returns>
        TTarget Convert(TSource source);
    }
}
```

1.4 ./Platform.Converters/IConverter[T].cs

```
namespace Platform.Converters
{
    /// <summary>
    /// <para>Defines a converter between two values of the same type.</para>
    /// <para>Определяет конвертер между двумя значениями одного типа.</para>
    /// </summary>
    /// <typeparam name="T"><para>Type of value to convert.</para><para>Тип преобразуемого значения.</para></typeparam>
    public interface IConverter<T> : IConverter<T, T>
    {
    }
}
```

1.5 ./Platform.Converters/To.cs

```
using System;
using System.Runtime.CompilerServices;

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

namespace Platform.Converters
{
    [Obsolete]
    public static class To
    {
        public static readonly char UnknownCharacter = '';

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static ulong UInt64(ulong value) => value;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static long Int64(ulong value) => unchecked(value > long.MaxValue ? long.MaxValue : (long)value);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static uint UInt32(ulong value) => unchecked(value > uint.MaxValue ? uint.MaxValue : (uint)value);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static int Int32(ulong value) => unchecked(value > int.MaxValue ? int.MaxValue : (int)value);

```

```
25          [MethodImpl(MethodImplOptions.AggressiveInlining)]
26          public static ushort UInt16(ulong value) => unchecked(value > ushort.MaxValue ?
            ↪  ushort.MaxValue : (ushort)value);
27
28          [MethodImpl(MethodImplOptions.AggressiveInlining)]
29          public static short Int16(ulong value) => unchecked(value > (ulong)short.MaxValue ?
            ↪  short.MaxValue : (short)value);
30
31          [MethodImpl(MethodImplOptions.AggressiveInlining)]
32          public static byte Byte(ulong value) => unchecked(value > byte.MaxValue ? byte.MaxValue
            ↪  : (byte)value);
33
34          [MethodImpl(MethodImplOptions.AggressiveInlining)]
35          public static sbyte SByte(ulong value) => unchecked(value > (ulong)sbyte.MaxValue ?
            ↪  sbyte.MaxValue : (sbyte)value);
36
37          [MethodImpl(MethodImplOptions.AggressiveInlining)]
38          public static bool Boolean(ulong value) => value > 0UL;
39
40          [MethodImpl(MethodImplOptions.AggressiveInlining)]
41          public static char Char(ulong value) => unchecked(value > char.MaxValue ?
            ↪  UnknownCharacter : (char)value);
42
43          [MethodImpl(MethodImplOptions.AggressiveInlining)]
44          public static DateTime DateTime(ulong value) => unchecked(value > long.MaxValue ?
            ↪  System.DateTime.MaxValue : new DateTime((long)value));
45
46          [MethodImpl(MethodImplOptions.AggressiveInlining)]
47          public static TimeSpan TimeSpan(ulong value) => unchecked(value > long.MaxValue ?
            ↪  System.TimeSpan.MaxValue : new TimeSpan((long)value));
48
49          [MethodImpl(MethodImplOptions.AggressiveInlining)]
50          public static ulong UInt64(long value) => unchecked(value < (long)ulong.MinValue ?
            ↪  ulong.MinValue : (ulong)value);
51
52          [MethodImpl(MethodImplOptions.AggressiveInlining)]
53          public static ulong UInt64(int value) => unchecked(value < (int)ulong.MinValue ?
            ↪  ulong.MinValue : (ulong)value);
54
55          [MethodImpl(MethodImplOptions.AggressiveInlining)]
56          public static ulong UInt64(short value) => unchecked(value < (short)ulong.MinValue ?
            ↪  ulong.MinValue : (ulong)value);
57
58          [MethodImpl(MethodImplOptions.AggressiveInlining)]
59          public static ulong UInt64(sbyte value) => unchecked(value < (sbyte)ulong.MinValue ?
            ↪  ulong.MinValue : (ulong)value);
60
61          [MethodImpl(MethodImplOptions.AggressiveInlining)]
62          public static ulong UInt64(bool value) => value ? 1UL : 0UL;
63
64          [MethodImpl(MethodImplOptions.AggressiveInlining)]
65          public static ulong UInt64(char value) => value;
66
67          [MethodImpl(MethodImplOptions.AggressiveInlining)]
68          public static long Signed(ulong value) => unchecked((long)value);
69
70          [MethodImpl(MethodImplOptions.AggressiveInlining)]
71          public static int Signed(uint value) => unchecked((int)value);
72
73          [MethodImpl(MethodImplOptions.AggressiveInlining)]
74          public static short Signed(ushort value) => unchecked((short)value);
75
76          [MethodImpl(MethodImplOptions.AggressiveInlining)]
77          public static sbyte Signed(byte value) => unchecked((sbyte)value);
78
79          [MethodImpl(MethodImplOptions.AggressiveInlining)]
80          public static object Signed<T>(T value) => To<T>.Signed(value);
81
82          [MethodImpl(MethodImplOptions.AggressiveInlining)]
83          public static ulong Unsigned(long value) => unchecked((ulong)value);
84
85          [MethodImpl(MethodImplOptions.AggressiveInlining)]
86          public static uint Unsigned(int value) => unchecked((uint)value);
87
88          [MethodImpl(MethodImplOptions.AggressiveInlining)]
89          public static ushort Unsigned(short value) => unchecked((ushort)value);
90
91          [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
        public static byte Unsigned(sbyte value) => unchecked((byte)value);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static object Unsigned<T>(T value) => To<T>.Unsigned(value);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static T UnsignedAs<T>(object value) => To<T>.UnsignedAs(value);
    }
}
```

1.6 ./Platform.Converters/To[T].cs

```
using System;
using System.Runtime.CompilerServices;
using Platform.Exceptions;
using Platform.Reflection;

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

namespace Platform.Converters
{
    [Obsolete]
    public static class To<T>
    {
        public static readonly Func<T, object> Signed = CompileSignedDelegate();
        public static readonly Func<T, object> Unsigned = CompileUnsignedDelegate();
        public static readonly Func<object, T> UnsignedAs = CompileUnsignedAsDelegate();

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        static private Func<T, object> CompileSignedDelegate()
        {
            return DelegateHelpers.Compile<Func<T, object>>(emiter =>
            {
                Ensure.Always.IsUnsignedInteger<T>();
                emiter.LoadArgument(0);
                var method = typeof(To).GetMethod("Signed", Types<T>.Array);
                emiter.Call(method);
                emiter.Box(method.ReturnType);
                emiter.Return();
            });
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        static private Func<T, object> CompileUnsignedDelegate()
        {
            return DelegateHelpers.Compile<Func<T, object>>(emiter =>
            {
                Ensure.Always.IsSignedInteger<T>();
                emiter.LoadArgument(0);
                var method = typeof(To).GetMethod("Unsigned", Types<T>.Array);
                emiter.Call(method);
                emiter.Box(method.ReturnType);
                emiter.Return();
            });
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        static private Func<object, T> CompileUnsignedAsDelegate()
        {
            return DelegateHelpers.Compile<Func<object, T>>(emiter =>
            {
                Ensure.Always.IsUnsignedInteger<T>();
                emiter.LoadArgument(0);
                var signedVersion = NumericType<T>.SignedVersion;
                emiter.UnboxValue(signedVersion);
                var method = typeof(To).GetMethod("Unsigned", new[] { signedVersion });
                emiter.Call(method);
                emiter.Return();
            });
        }
    }
}
```

1.7 ./Platform.Converters/UncheckedConverter.cs

```
using System;
using System.Reflection;
using System.Reflection.Emit;
using System.Runtime.CompilerServices;
using Platform.Reflection;

```

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

namespace Platform.Converters
{
    public abstract class UncheckedConverter<TSource, TTarget> : IConverter<TSource, TTarget>
    {
        public static UncheckedConverter<TSource, TTarget> Default
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        static UncheckedConverter()
        {
            AssemblyName assemblyName = new AssemblyName(GetNewName());
            var assembly = AssemblyBuilder.DefineDynamicAssembly(assemblyName,
            ↪  AssemblyBuilderAccess.Run);
            var module = assembly.DefineDynamicModule(GetNewName());
            var type = module.DefineType(GetNewName(), TypeAttributes.Public |
            ↪  TypeAttributes.Class | TypeAttributes.Sealed, typeof(UncheckedConverter<TSource,
            ↪  TTarget>));
            type.EmitVirtualMethod<Converter<TSource, TTarget>>("Convert", il =>
            {
                il.LoadArgument(1);
                if (typeof(TSource) != typeof(TTarget))
                {
                    il.UncheckedConvert<TSource, TTarget>();
                }
                il.Return();
            });
            var typeInfo = type.CreateTypeInfo();
            Default = (UncheckedConverter<TSource, TTarget>)Activator.CreateInstance(typeInfo);
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static string GetNewName() => Guid.NewGuid().ToString("N");

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public abstract TTarget Convert(TSource source);
    }
}
```

1.8 ./Platform.Converters.Tests/ConverterTests.cs

```
using Platform.Diagnostics;
using System;
using System.Globalization;
using System.Runtime.CompilerServices;
using Xunit;
using Xunit.Abstractions;

namespace Platform.Converters.Tests
{
    public class ConverterTests
    {
        private readonly ITestOutputHelper _output;
        private static readonly UncheckedConverter<ulong, ulong> _uInt64ToUInt64Converter =
        ↪  UncheckedConverter<ulong, ulong>.Default;
        private static readonly UncheckedConverter<int, ulong> _int32ToUInt64converter =
        ↪  UncheckedConverter<int, ulong>.Default;

        public ConverterTests(ITestOutputHelper output) => _output = output;

        [Fact]
        public void SameTypeTest()
        {
            var result = UncheckedConverter<ulong, ulong>.Default.Convert(2UL);
            Assert.Equal(2UL, result);
            result = CheckedConverter<ulong, ulong>.Default.Convert(2UL);
            Assert.Equal(2UL, result);
        }

        [Fact]
        public void SameTypePerformanceComparisonTest()
        {
            var N = 10000000;
            var result = 0UL;

            // Warmup
            for (int i = 0; i < N; i++)
```

```
            {
                result = _uInt64ToUInt64Converter.Convert(2UL);
            }
            for (int i = 0; i < N; i++)
            {
                result = UncheckedConverter<ulong, ulong>.Default.Convert(2UL);
            }
            for (int i = 0; i < N; i++)
            {
                result = Convert(2UL);
            }
            for (int i = 0; i < N; i++)
            {
                result = To.UInt64(2UL);
            }
            for (int i = 0; i < N; i++)
            {
                result = System.Convert.ToUInt64(2UL);
            }
            for (int i = 0; i < N; i++)
            {
                result = (ulong)System.Convert.ChangeType(2UL, typeof(ulong));
            }

            var ts1 = Performance.Measure(() =>
            {
                for (int i = 0; i < N; i++)
                {
                    result = _uInt64ToUInt64Converter.Convert(2UL);
                }
            });
            var ts2 = Performance.Measure(() =>
            {
                for (int i = 0; i < N; i++)
                {
                    result = UncheckedConverter<ulong, ulong>.Default.Convert(2UL);
                }
            });
            var ts3 = Performance.Measure(() =>
            {
                for (int i = 0; i < N; i++)
                {
                    result = Convert(2UL);
                }
            });
            var ts4 = Performance.Measure(() =>
            {
                for (int i = 0; i < N; i++)
                {
                    result = To.UInt64(2UL);
                }
            });
            var ts5 = Performance.Measure(() =>
            {
                for (int i = 0; i < N; i++)
                {
                    result = System.Convert.ToUInt64(2UL);
                }
            });
            var ts6 = Performance.Measure(() =>
            {
                for (int i = 0; i < N; i++)
                {
                    result = (ulong)System.Convert.ChangeType(2UL, typeof(ulong));
                }
            });
            IFormatProvider formatProvider = CultureInfo.InvariantCulture;
            var ts7 = Performance.Measure(() =>
            {
                for (int i = 0; i < N; i++)
                {
                    result = ((IConvertible)2UL).ToUInt64(formatProvider);
                }
            });
            var ts8 = Performance.Measure(() =>
            {
                for (int i = 0; i < N; i++)
                {
```

```
                result = (ulong)((IConvertible)2UL).ToType(typeof(ulong), formatProvider);
            }
        });

        _output.WriteLine($"{ts1} {ts2} {ts3} {ts4} {ts5} {ts6} {ts7} {ts8} {result}");
    }

    [Fact]
    public void Int32ToUInt64Test()
    {
        var result = UncheckedConverter<int, ulong>.Default.Convert(2);
        Assert.Equal(2UL, result);
        result = CheckedConverter<int, ulong>.Default.Convert(2);
        Assert.Equal(2UL, result);
    }

    [Fact]
    public void Int32ToUInt64PerformanceComparisonTest()
    {
        var N = 10000000;
        var result = 0UL;

        // Warmup
        for (int i = 0; i < N; i++)
        {
            result = _int32ToUInt64converter.Convert(2);
        }
        for (int i = 0; i < N; i++)
        {
            result = UncheckedConverter<ulong, ulong>.Default.Convert(2);
        }
        for (int i = 0; i < N; i++)
        {
            result = Convert(2);
        }
        for (int i = 0; i < N; i++)
        {
            result = To.UInt64(2);
        }
        for (int i = 0; i < N; i++)
        {
            result = System.Convert.ToUInt64(2);
        }
        for (int i = 0; i < N; i++)
        {
            result = (ulong)System.Convert.ChangeType(2, typeof(ulong));
        }

        var ts1 = Performance.Measure(() =>
        {
            for (int i = 0; i < N; i++)
            {
                result = _int32ToUInt64converter.Convert(2);
            }
        });
        var ts2 = Performance.Measure(() =>
        {
            for (int i = 0; i < N; i++)
            {
                result = UncheckedConverter<ulong, ulong>.Default.Convert(2);
            }
        });
        var ts3 = Performance.Measure(() =>
        {
            for (int i = 0; i < N; i++)
            {
                result = Convert(2);
            }
        });
        var ts4 = Performance.Measure(() =>
        {
            for (int i = 0; i < N; i++)
            {
                result = To.UInt64(2);
            }
        });
        var ts5 = Performance.Measure(() =>
        {
```

```
                for (int i = 0; i < N; i++)
                {
                    result = System.Convert.ToUInt64(2);
                }
            });
            var ts6 = Performance.Measure(() =>
            {
                for (int i = 0; i < N; i++)
                {
                    result = (ulong)System.Convert.ChangeType(2, typeof(ulong));
                }
            });
            IFormatProvider formatProvider = CultureInfo.InvariantCulture;
            var ts7 = Performance.Measure(() =>
            {
                for (int i = 0; i < N; i++)
                {
                    result = ((IConvertible)2).ToUInt64(formatProvider);
                }
            });
            var ts8 = Performance.Measure(() =>
            {
                for (int i = 0; i < N; i++)
                {
                    result = (ulong)((IConvertible)2).ToType(typeof(ulong), formatProvider);
                }
            });

            _output.WriteLine($"{ts1} {ts2} {ts3} {ts4} {ts5} {ts6} {ts7} {ts8} {result}");
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static ulong Convert(ulong value) => _uInt64ToUInt64Converter.Convert(value);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static ulong Convert(int value) => _int32ToUInt64converter.Convert(value);
    }
}
```

Index