LinksPlatform's Platform.Disposables Class Library

./DisposableBase.cs

```
using System;
using System.Diagnostics;
using System.Threading;
using Platform.Exceptions;

namespace Platform.Disposables
{
    /// <summary>
    /// Provides a base implementation for IDisposable interface with the basic logic necessary to increase the likelihood of correct unmanaged resources release.
    /// Предоставляет базовую реализацию для интерфейса IDisposable с основной логикой необходимой для повышения вероятности корректного высвобождения неуправляемых ресурсов.
    /// </summary>
    public abstract class DisposableBase : IDisposable
    {
        private static readonly Process _currentProcess = Process.GetCurrentProcess();

        private volatile int _disposed;

        public bool IsDisposed => _disposed > 0;

        protected virtual string ObjectName => GetType().Name;

        protected virtual bool AllowMultipleDisposeAttempts => false;

        protected virtual bool AllowMultipleDisposeCalls => false;

        protected DisposableBase()
        {
            _disposed = 0;
            _currentProcess.Exited += OnProcessExit;
        }

        ~DisposableBase() => Destruct();

        protected abstract void Dispose(bool manual, bool wasDisposed);

        public void Dispose()
        {
            GC.SuppressFinalize(this);
            Dispose(true);
        }

        public void Destruct()
        {
            try
            {
                if (!IsDisposed)
                {
                    Dispose(false);
                }
            }
            catch (Exception exception)
            {
                exception.Ignore();
            }
        }

        private void OnProcessExit(object sender, EventArgs e)
        {
            GC.SuppressFinalize(this);
            Destruct();
        }

        private void Dispose(bool manual)
        {
            var originalDisposedValue = Interlocked.CompareExchange(ref _disposed, 1, 0);
            var wasDisposed = originalDisposedValue > 0;
            if (!wasDisposed)
            {
                UnsubscribeFromProcessExitedEventIfPossible();
            }
            if (wasDisposed && !AllowMultipleDisposeCalls && manual)
            {
                Ensure.Always.NotDisposed(this, ObjectName, "Multiple dispose calls are now allowed. Override AllowMultipleDisposeCalls property to modify behaviour.");
            }
            if (AllowMultipleDisposeAttempts || !wasDisposed)
```

```
            {
                Dispose(manual, wasDisposed);
            }
        }

        private void UnsubscribeFromProcessExitedEventIfPossible()
        {
            try
            {
                if (_currentProcess != null)
                {
                    _currentProcess.Exited -= OnProcessExit;
                }
                else
                {
                    Process.GetCurrentProcess().Exited -= OnProcessExit;
                }
            }
            catch (Exception exception)
            {
                exception.Ignore();
            }
        }
    }
}
```

./DisposableBaseExtensions.cs

```
namespace Platform.Disposables
{
    public static class DisposableBaseExtensions
    {
        public static void DisposeIfNotDisposed(this DisposableBase disposable)
        {
            if (!disposable.IsDisposed)
            {
                disposable.Dispose();
            }
        }
    }
}
```

./Disposable.cs

```
using System;

namespace Platform.Disposables
{
    /// <summary>
    /// Represents disposable object that contains OnDispose event which is raised when the
    ↪  object itself is disposed.
    /// Представляет высвобождаемый объект, который содержит событие OnDispose, которое
    ↪  возникает при высвобождении самого объекта.
    /// </summary>
    public class Disposable : DisposableBase
    {
        private static readonly Disposal _emptyDelegate = (manual, wasDisposed) => { };

        public event Disposal OnDispose;

        public Disposable(Action disposed)
        {
            OnDispose = (manual, wasDisposed) =>
            {
                if (!wasDisposed)
                {
                    disposed();
                }
            };
        }

        public Disposable(Disposal disposed) => OnDispose = disposed;

        public Disposable() => OnDispose = _emptyDelegate;

        public static implicit operator Disposable(Action action) => new Disposable(action);

        public static implicit operator Disposable(Disposal disposal) => new
        ↪  Disposable(disposal);

```

```
        protected override void Dispose(bool manual, bool wasDisposed) => OnDispose(manual,
        ↪ wasDisposed);

        protected void RaiseOnDisposeEvent(bool manual, bool wasDisposed) => OnDispose(manual,
        ↪ wasDisposed);

        public static bool TryDisposeAndResetToDefault<T>(ref T @object)
        {
            var result = @object.TryDispose();
            if (result)
            {
                @object = default;
            }
            return result;
        }
    }
}
```

./Disposable[T].cs

```
using System;

namespace Platform.Disposables
{
    /// <summary>
    /// Represents disposable container that disposes contained object when the container itself
    ↪ is disposed.
    /// Представляет высвобождаемый контейнер, который высвобождает содержащийся в нём объект
    ↪ при высвобождении самого контейнера.
    /// </summary>
    public class Disposable<T> : Disposable
    {
        public T Object { get; }

        public Disposable(T @object, Action<T> disposed)
        {
            Object = @object;
            OnDispose += (manual, wasDisposed) =>
            {
                if (!wasDisposed)
                {
                    disposed(Object);
                }
            };
        }

        public Disposable(T @object, Action disposed) : base(disposed) => Object = @object;

        public Disposable(T @object, Disposal disposed) : base(disposed) => Object = @object;

        public Disposable(T @object) => Object = @object;

        public static implicit operator Disposable<T>(ValueTuple<T, Action<T>> tuple) => new
        ↪ Disposable<T>(tuple.Item1, tuple.Item2);

        public static implicit operator Disposable<T>(ValueTuple<T, Action> tuple) => new
        ↪ Disposable<T>(tuple.Item1, tuple.Item2);

        public static implicit operator Disposable<T>(ValueTuple<T, Disposal> tuple) => new
        ↪ Disposable<T>(tuple.Item1, tuple.Item2);

        public static implicit operator Disposable<T>(T @object) => new Disposable<T>(@object);

        public static implicit operator T(Disposable<T> disposable) => disposable.Object;

        protected override void Dispose(bool manual, bool wasDisposed)
        {
            base.Dispose(manual, wasDisposed);
            Object.TryDispose();
        }
    }
}
```

./Disposable[TPrimary, TAuxiliary].cs

```
using System;

namespace Platform.Disposables
{
    /// <summary>
```

```
    /// Represents disposable container that disposes two contained objects when the container itself is disposed.
    /// Представляет высвобождаемый контейнер, который высвобождает два содержащийся в нём объектов при высвобождении самого контейнера.
    /// </summary>
    public class Disposable<TPrimary, TAuxiliary> : Disposable<TPrimary>
    {
        public TAuxiliary AuxiliaryObject { get; }

        public Disposable(TPrimary @object, TAuxiliary auxiliaryObject, Action<TPrimary, TAuxiliary> action)
            : base(@object)
        {
            AuxiliaryObject = auxiliaryObject;
            OnDispose += (manual, wasDisposed) =>
            {
                if (!wasDisposed)
                {
                    action(Object, AuxiliaryObject);
                }
            };
        }

        public Disposable(TPrimary @object, TAuxiliary auxiliaryObject, Action action) : base(@object, action) => AuxiliaryObject = auxiliaryObject;

        public Disposable(TPrimary @object, TAuxiliary auxiliaryObject, Disposal disposal) : base(@object, disposal) => AuxiliaryObject = auxiliaryObject;

        public Disposable(TPrimary @object, TAuxiliary auxiliaryObject) : base(@object) => AuxiliaryObject = auxiliaryObject;

        public Disposable(TPrimary @object) : base(@object) { }

        public static implicit operator Disposable<TPrimary, TAuxiliary>(ValueTuple<TPrimary, TAuxiliary, Action<TPrimary, TAuxiliary>> tuple) => new Disposable<TPrimary, TAuxiliary>(tuple.Item1, tuple.Item2, tuple.Item3);

        public static implicit operator Disposable<TPrimary, TAuxiliary>(ValueTuple<TPrimary, TAuxiliary, Action> tuple) => new Disposable<TPrimary, TAuxiliary>(tuple.Item1, tuple.Item2, tuple.Item3);

        public static implicit operator Disposable<TPrimary, TAuxiliary>(ValueTuple<TPrimary, TAuxiliary, Disposal> tuple) => new Disposable<TPrimary, TAuxiliary>(tuple.Item1, tuple.Item2, tuple.Item3);

        public static implicit operator Disposable<TPrimary, TAuxiliary>(ValueTuple<TPrimary, TAuxiliary> tuple) => new Disposable<TPrimary, TAuxiliary>(tuple.Item1, tuple.Item2);

        public static implicit operator TPrimary(Disposable<TPrimary, TAuxiliary> disposableContainer) => disposableContainer.Object;

        public static implicit operator TAuxiliary(Disposable<TPrimary, TAuxiliary> disposableContainer) => disposableContainer.AuxiliaryObject;

        protected override void Dispose(bool manual, bool wasDisposed)
        {
            RaiseOnDisposeEvent(manual, wasDisposed);
            AuxiliaryObject.TryDispose();
            Object.TryDispose();
        }
    }
}
```

./Disposal.cs

```
namespace Platform.Disposables
{
    public delegate void Disposal(bool manual, bool wasDisposed);
}
```

./EnsureExtensions.cs

```
using System;
using System.Diagnostics;
using System.Runtime.CompilerServices;
using Platform.Exceptions;
using Platform.Exceptions.ExtensionRoots;

#pragma warning disable IDE0060 // Remove unused parameter
```

```

namespace Platform.Disposables
{
    public static class EnsureExtensions
    {
        #region Always

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void NotDisposed(this EnsureAlwaysExtensionRoot root, IDisposable
        ↪  disposable) => NotDisposed(root, disposable, null, null);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void NotDisposed(this EnsureAlwaysExtensionRoot root, IDisposable
        ↪  disposable, string objectName) => NotDisposed(root, disposable, objectName, null);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void NotDisposed(this EnsureAlwaysExtensionRoot root, IDisposable
        ↪  disposable, string objectName, string message)
        {
            if (disposable.IsDisposed)
            {
                throw new ObjectDisposedException(objectName, message);
            }
        }

        #endregion

        #region OnDebug

        [Conditional("DEBUG")]
        public static void NotDisposed(this EnsureOnDebugExtensionRoot root, IDisposable
        ↪  disposable) => Ensure.Always.NotDisposed(disposable, null, null);

        [Conditional("DEBUG")]
        public static void NotDisposed(this EnsureOnDebugExtensionRoot root, IDisposable
        ↪  disposable, string objectName) => Ensure.Always.NotDisposed(disposable, objectName,
        ↪  null);

        [Conditional("DEBUG")]
        public static void NotDisposed(this EnsureOnDebugExtensionRoot root, IDisposable
        ↪  disposable, string objectName, string message) =>
        ↪  Ensure.Always.NotDisposed(disposable, objectName, message);

        #endregion
    }
}
```

./GenericObjectExtensions.cs

```
using System;
using Platform.Exceptions;

namespace Platform.Disposables
{
    static public class GenericObjectExtensions
    {
        public static bool TryDispose<T>(this T @object)
        {
            try
            {
                if (@object is DisposableBase disposableBase)
                {
                    disposableBase.DisposeIfNotDisposed();
                }
                else if (@object is System.IDisposable disposable)
                {
                    disposable.Dispose();
                }
                return true;
            }
            catch (Exception exception)
            {
                exception.Ignore();
            }
            return false;
        }

        public static void DisposeIfPossible<T>(this T @object) => TryDispose(@object);
    }
}
```

./IDisposable.cs

```
namespace Platform.Disposables
{
    /// <summary>
    /// Представляет расширенный интерфейс IDisposable.
    /// Represents an extended IDisposable interface.
    /// </summary>
    public interface IDisposable : System.IDisposable
    {
        /// <summary>
        /// Gets a value indicating whether the object was disposed.
        /// Возвращает значение определяющее был ли высвобожден объект.
        /// </summary>
        bool IsDisposed { get; }

        /// <summary>
        /// Performs application-defined tasks associated with freeing, releasing, or resetting
        ↪  unmanaged resources without throwing any exceptions.
        /// Выполняет определенные пользователем задачи, связанные с освобождением,
        ↪  высвобождением или сбросом неуправляемых ресурсов без выбрасывания исключений.
        /// </summary>
        /// <remarks>
        /// Should be called only from classes destructors, or in case exceptions should be not
        ↪  thrown.
        /// Должен вызываться только из деструкторов классов, или в случае, если исключения
        ↪  выбрасывать нельзя.
        /// </remarks>
        void Destruct();
    }
}
```

Index