LinksPlatform's Platform.Scopes Class Library

```csharp
using System;
using System.Collections.Generic;
using System.Reflection;
using System.Linq;
using Platform.Interfaces;
using Platform.Exceptions;
using Platform.Disposables;
using Platform.Collections.Lists;
using Platform.Reflection;
using Platform.Singletons;
using System.Runtime.CompilerServices;

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

namespace Platform.Scopes
{
    public class Scope : DisposableBase
    {
        public static readonly Scope Global = new Scope(autoInclude: true, autoExplore: true);

        private readonly bool _autoInclude;
        private readonly bool _autoExplore;
        private readonly Stack<object> _dependencies = new Stack<object>();
        private readonly HashSet<object> _excludes = new HashSet<object>();
        private readonly HashSet<object> _includes = new HashSet<object>();
        private readonly HashSet<object> _blocked = new HashSet<object>();
        private readonly Dictionary<Type, object> _resolutions = new Dictionary<Type, object>();

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public Scope(bool autoInclude, bool autoExplore)
        {
            _autoInclude = autoInclude;
            _autoExplore = autoExplore;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public Scope(bool autoInclude) : this(autoInclude, false) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public Scope() { }

        #region Exclude

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void ExcludeAssemblyOf<T>() => ExcludeAssemblyOfType(typeof(T));

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void ExcludeAssemblyOfType(Type type) => ExcludeAssembly(type.GetAssembly());

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void ExcludeAssembly(Assembly assembly) =>
            assembly.GetCachedLoadableTypes().ForEach(Exclude);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void Exclude<T>() => Exclude(typeof(T));

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void Exclude(object @object) => _excludes.Add(@object);

        #endregion

        #region Include

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void IncludeAssemblyOf<T>() => IncludeAssemblyOfType(typeof(T));

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void IncludeAssemblyOfType(Type type) => IncludeAssembly(type.GetAssembly());

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void IncludeAssembly(Assembly assembly) =>
            assembly.GetExportedTypes().ForEach(Include);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void Include<T>()
        {
            var types = Types<T>.Array;
            if (types.Length > 0)
            {
```

```csharp
                    types.ForEach(Include);
                }
                else
                {
                    Include(typeof(T));
                }
            }

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public void Include(object @object)
            {
                if (@object == null)
                {
                    return;
                }
                if (_includes.Add(@object))
                {
                    var type = @object as Type;
                    if (type != null)
                    {
                        type.GetInterfaces().ForEach(Include);
                        Include(type.GetBaseType());
                    }
                }
            }

            #endregion

            #region Use

            /// <remarks>
            /// TODO: Use Default[T].Instance if the only constructor object has is parameterless.
            /// TODO: Think of interface chaining IDoubletLinks[T] (default) -> IDoubletLinks[T]
            ↪   (checker) -> IDoubletLinks[T] (synchronizer) (may be UseChain[IDoubletLinks[T],
            ↪   Types[DefaultLinks, DefaultLinksDependencyChecker, DefaultSynchronizedLinks]]
            /// TODO: Add support for factories
            /// </remarks>
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public T Use<T>()
            {
                if (_excludes.Contains(typeof(T)))
                {
                    throw new InvalidOperationException($"Type {typeof(T).Name} is excluded and
                    ↪   cannot be used.");
                }
                if (_autoInclude)
                {
                    Include<T>();
                }
                if (!TryResolve(out T resolved))
                {
                    throw new InvalidOperationException($"Dependency of type {typeof(T).Name}
                    ↪   cannot be resolved.");
                }
                if (!_autoInclude)
                {
                    Include<T>();
                }
                Use(resolved);
                return resolved;
            }

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public T UseSingleton<T>(IFactory<T> factory) => UseAndReturn(Singleton.Get(factory));

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public T UseSingleton<T>(Func<T> creator) => UseAndReturn(Singleton.Get(creator));

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public T UseAndReturn<T>(T @object)
            {
                Use(@object);
                return @object;
            }

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public void Use(object @object)
            {
```

```csharp
152            Include(@object);
153            _dependencies.Push(@object);
154        }

156        #endregion

158        #region Resolve

160        [MethodImpl(MethodImplOptions.AggressiveInlining)]
161        public bool TryResolve<T>(out T resolved)
162        {
163            resolved = default;
164            var result = TryResolve(typeof(T), out object resolvedObject);
165            if (result)
166            {
167                resolved = (T)resolvedObject;
168            }
169            return result;
170        }

172        [MethodImpl(MethodImplOptions.AggressiveInlining)]
173        public bool TryResolve(Type requiredType, out object resolved)
174        {
175            resolved = null;
176            if (!_blocked.Add(requiredType))
177            {
178                return false;
179            }
180            try
181            {
182                if (_excludes.Contains(requiredType))
183                {
184                    return false;
185                }
186                if (_resolutions.TryGetValue(requiredType, out resolved))
187                {
188                    return true;
189                }
190                if (_autoExplore)
191                {
192                    IncludeAssemblyOfType(requiredType);
193                }
194                var resultInstances = new List<object>();
195                var resultConstructors = new List<ConstructorInfo>();
196                foreach (var include in _includes)
197                {
198                    if (!_excludes.Contains(include))
199                    {
200                        var type = include as Type;
201                        if (type != null)
202                        {
203                            if (requiredType.IsAssignableFrom(type))
204                            {
205                                resultConstructors.AddRange(GetValidConstructors(type));
206                            }
207                            else if (type.GetTypeInfo().IsGenericTypeDefinition &&
                             ↪    requiredType.GetTypeInfo().IsGenericType &&
                             ↪    type.GetInterfaces().Any(x => x.Name == requiredType.Name))
208                            {
209                                var genericType =
                                 ↪    type.MakeGenericType(requiredType.GenericTypeArguments);
210                                if (requiredType.IsAssignableFrom(genericType))
211                                {
212                                    resultConstructors.AddRange(GetValidConstructors(genericType
                                     ↪    ));
213                                }
214                            }
215                        }
216                        else if (requiredType.IsInstanceOfType(include) ||
                             ↪    requiredType.IsAssignableFrom(include.GetType()))
217                        {
218                            resultInstances.Add(include);
219                        }
220                    }
221                }
222                if (resultInstances.Count == 0 && resultConstructors.Count == 0)
223                {
224                    return false;
```

```csharp
                }
                else if (resultInstances.Count > 0)
                {
                    resolved = resultInstances[0];
                }
                else
                {
                    SortConstructors(resultConstructors);
                    if (!TryResolveInstance(resultConstructors, out resolved))
                    {
                        return false;
                    }
                }
                _resolutions.Add(requiredType, resolved);
                return true;
            }
            finally
            {
                _blocked.Remove(requiredType);
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual void SortConstructors(List<ConstructorInfo> resultConstructors) =>
            resultConstructors.Sort((x, y) =>
            -x.GetParameters().Length.CompareTo(y.GetParameters().Length));

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual bool TryResolveInstance(List<ConstructorInfo> constructors, out object
            resolved)
        {
            for (var i = 0; i < constructors.Count; i++)
            {
                try
                {
                    var resultConstructor = constructors[i];
                    if (TryResolveConstructorArguments(resultConstructor, out object[]
                        arguments))
                    {
                        resolved = resultConstructor.Invoke(arguments);
                        return true;
                    }
                }
                catch (Exception exception)
                {
                    exception.Ignore();
                }
            }
            resolved = null;
            return false;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private ConstructorInfo[] GetValidConstructors(Type type)
        {
            var constructors = type.GetConstructors();
            if (!_autoExplore)
            {
                constructors = constructors.ToArray(x =>
                {
                    var parameters = x.GetParameters();
                    for (var i = 0; i < parameters.Length; i++)
                    {
                        if (!_includes.Contains(parameters[i].ParameterType))
                        {
                            return false;
                        }
                    }
                    return true;
                });
            }
            return constructors;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private bool TryResolveConstructorArguments(ConstructorInfo constructor, out object[]
            arguments)
        {
```

```csharp
298            var parameters = constructor.GetParameters();
299            arguments = new object[parameters.Length];
300            for (var i = 0; i < parameters.Length; i++)
301            {
302                if (!TryResolve(parameters[i].ParameterType, out object argument))
303                {
304                    return false;
305                }
306                Use(argument);
307                arguments[i] = argument;
308            }
309            return true;
310        }

312        #endregion

314        [MethodImpl(MethodImplOptions.AggressiveInlining)]
315        protected override void Dispose(bool manual, bool wasDisposed)
316        {
317            if (!wasDisposed)
318            {
319                while (_dependencies.Count > 0)
320                {
321                    _dependencies.Pop().DisposeIfPossible();
322                }
323            }
324        }
325    }
326 }
```

1.2 ./csharp/Platform.Scopes/Scope[TInclude].cs

```csharp
1  using System.Runtime.CompilerServices;
2
3  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5  namespace Platform.Scopes
6  {
7      public class Scope<TInclude> : Scope
8      {
9          [MethodImpl(MethodImplOptions.AggressiveInlining)]
10         public Scope() : this(false, false) { }
11
12         [MethodImpl(MethodImplOptions.AggressiveInlining)]
13         public Scope(bool autoInclude) : this(autoInclude, false) { }
14
15         [MethodImpl(MethodImplOptions.AggressiveInlining)]
16         public Scope(bool autoInclude, bool autoExplore) : base(autoInclude, autoExplore) =>
17             ↪  Include<TInclude>();
18      }
19  }
```

1.3 ./csharp/Platform.Scopes/Use.cs

```csharp
1  using System.Runtime.CompilerServices;
2  using Platform.Disposables;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Scopes
7  {
8      public static class Use<T>
9      {
10         public static T Single
11         {
12             [MethodImpl(MethodImplOptions.AggressiveInlining)]
13             get => Scope.Global.Use<T>();
14         }
15
16         public static Disposable<T> New
17         {
18             [MethodImpl(MethodImplOptions.AggressiveInlining)]
19             get
20             {
21                 var scope = new Scope(autoInclude: true, autoExplore: true);
22                 return new Disposable<T, Scope>(scope.Use<T>(), scope);
23             }
24         }
25      }
26  }
```

```csharp
using Xunit;
using Platform.Reflection;

namespace Platform.Scopes.Tests
{
    public class ScopeTests
    {
        public interface IInterface
        {
        }

        public class Class : IInterface
        {
        }

        [Fact]
        public static void SingleDependencyTest()
        {
            using var scope = new Scope();
            scope.IncludeAssemblyOf<IInterface>();
            var instance = scope.Use<IInterface>();
            Assert.IsType<Class>(instance);
        }

        [Fact]
        public static void TypeParametersTest()
        {
            using var scope = new Scope<Types<Class>>();
            var instance = scope.Use<IInterface>();
            Assert.IsType<Class>(instance);
        }
    }
}
```

Index