1.1 ./csharp/Platform.Scopes/Scope.cs

```
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
{
    /// <summary>
    /// <para>
    /// Represents the scope.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="DisposableBase"/>
    public class Scope : DisposableBase
    {
        /// <summary>
        /// <para>
        /// The auto explore.
        /// </para>
        /// <para></para>
        /// </summary>
        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>();

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="Scope"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="autoInclude">
        /// <para>A auto include.</para>
        /// <para></para>
        /// </param>
        /// <param name="autoExplore">
        /// <para>A auto explore.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public Scope(bool autoInclude, bool autoExplore)
        {
            _autoInclude = autoInclude;
            _autoExplore = autoExplore;
        }

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="Scope"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="autoInclude">
        /// <para>A auto include.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public Scope(bool autoInclude) : this(autoInclude, false) { }

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="Scope"/> instance.
```

```
        /// </para>
        /// <para></para>
        /// </summary>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public Scope() { }

        #region Exclude

        /// <summary>
        /// <para>
        /// Excludes the assembly of.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <typeparam name="T">
        /// <para>The .</para>
        /// <para></para>
        /// </typeparam>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void ExcludeAssemblyOf<T>() => ExcludeAssemblyOfType(typeof(T));

        /// <summary>
        /// <para>
        /// Excludes the assembly of type using the specified type.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="type">
        /// <para>The type.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void ExcludeAssemblyOfType(Type type) => ExcludeAssembly(type.GetAssembly());

        /// <summary>
        /// <para>
        /// Excludes the assembly using the specified assembly.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="assembly">
        /// <para>The assembly.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void ExcludeAssembly(Assembly assembly) =>
        ↪  assembly.GetCachedLoadableTypes().ForEach(Exclude);

        /// <summary>
        /// <para>
        /// Excludes this instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <typeparam name="T">
        /// <para>The .</para>
        /// <para></para>
        /// </typeparam>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void Exclude<T>() => Exclude(typeof(T));

        /// <summary>
        /// <para>
        /// Excludes the object.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="@object">
        /// <para>The object.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void Exclude(object @object) => _excludes.Add(@object);

        #endregion

        #region Include

        /// <summary>
```

```
        /// <para>
        /// Includes the assembly of.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <typeparam name="T">
        /// <para>The .</para>
        /// <para></para>
        /// </typeparam>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void IncludeAssemblyOf<T>() => IncludeAssemblyOfType(typeof(T));

        /// <summary>
        /// <para>
        /// Includes the assembly of type using the specified type.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="type">
        /// <para>The type.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void IncludeAssemblyOfType(Type type) => IncludeAssembly(type.GetAssembly());

        /// <summary>
        /// <para>
        /// Includes the assembly using the specified assembly.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="assembly">
        /// <para>The assembly.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void IncludeAssembly(Assembly assembly) =>
        ↪  assembly.GetExportedTypes().ForEach(Include);

        /// <summary>
        /// <para>
        /// Includes this instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <typeparam name="T">
        /// <para>The .</para>
        /// <para></para>
        /// </typeparam>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void Include<T>()
        {
            var types = Types<T>.Array;
            if (types.Length > 0)
            {
                types.ForEach(Include);
            }
            else
            {
                Include(typeof(T));
            }
        }

        /// <summary>
        /// <para>
        /// Includes the object.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="@object">
        /// <para>The object.</para>
        /// <para></para>
        /// </param>
        [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;
        }

        /// <summary>
        /// <para>
        /// Uses the singleton using the specified factory.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <typeparam name="T">
        /// <para>The .</para>
        /// <para></para>
        /// </typeparam>
        /// <param name="factory">
        /// <para>The factory.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public T UseSingleton<T>(IFactory<T> factory) => UseAndReturn(Singleton.Get(factory));

        /// <summary>
        /// <para>
        /// Uses the singleton using the specified creator.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <typeparam name="T">
        /// <para>The .</para>
```

```
        /// <para></para>
        /// </typeparam>
        /// <param name="creator">
        /// <para>The creator.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public T UseSingleton<T>(Func<T> creator) => UseAndReturn(Singleton.Get(creator));

        /// <summary>
        /// <para>
        /// Uses the and return using the specified object.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <typeparam name="T">
        /// <para>The .</para>
        /// <para></para>
        /// </typeparam>
        /// <param name="@object">
        /// <para>The object.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The object.</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public T UseAndReturn<T>(T @object)
        {
            Use(@object);
            return @object;
        }

        /// <summary>
        /// <para>
        /// Uses the object.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="@object">
        /// <para>The object.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void Use(object @object)
        {
            Include(@object);
            _dependencies.Push(@object);
        }

        #endregion

        #region Resolve

        /// <summary>
        /// <para>
        /// Determines whether this instance try resolve.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <typeparam name="T">
        /// <para>The .</para>
        /// <para></para>
        /// </typeparam>
        /// <param name="resolved">
        /// <para>The resolved.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The result.</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
385         public bool TryResolve<T>(out T resolved)
386         {
387             resolved = default;
388             var result = TryResolve(typeof(T), out object resolvedObject);
389             if (result)
390             {
391                 resolved = (T)resolvedObject;
392             }
393             return result;
394         }
395 
396         /// <summary>
397         /// <para>
398         /// Determines whether this instance try resolve.
399         /// </para>
400         /// <para></para>
401         /// </summary>
402         /// <param name="requiredType">
403         /// <para>The required type.</para>
404         /// <para></para>
405         /// </param>
406         /// <param name="resolved">
407         /// <para>The resolved.</para>
408         /// <para></para>
409         /// </param>
410         /// <returns>
411         /// <para>The bool</para>
412         /// <para></para>
413         /// </returns>
414         [MethodImpl(MethodImplOptions.AggressiveInlining)]
415         public bool TryResolve(Type requiredType, out object resolved)
416         {
417             resolved = null;
418             if (!_blocked.Add(requiredType))
419             {
420                 return false;
421             }
422             try
423             {
424                 if (_excludes.Contains(requiredType))
425                 {
426                     return false;
427                 }
428                 if (_resolutions.TryGetValue(requiredType, out resolved))
429                 {
430                     return true;
431                 }
432                 if (_autoExplore)
433                 {
434                     IncludeAssemblyOfType(requiredType);
435                 }
436                 var resultInstances = new List<object>();
437                 var resultConstructors = new List<ConstructorInfo>();
438                 foreach (var include in _includes)
439                 {
440                     if (!_excludes.Contains(include))
441                     {
442                         var type = include as Type;
443                         if (type != null)
444                         {
445                             if (requiredType.IsAssignableFrom(type))
446                             {
447                                 resultConstructors.AddRange(GetValidConstructors(type));
448                             }
449                             else if (type.GetTypeInfo().IsGenericTypeDefinition &&
                                 ↪  requiredType.GetTypeInfo().IsGenericType &&
                                 ↪  type.GetInterfaces().Any(x => x.Name == requiredType.Name))
450                             {
451                                 var genericType =
                                     ↪  type.MakeGenericType(requiredType.GenericTypeArguments);
452                                 if (requiredType.IsAssignableFrom(genericType))
453                                 {
454                                     resultConstructors.AddRange(GetValidConstructors(genericType
                                         ↪  ));
455                                 }
456                             }
457                         }
```

```
                        else if (requiredType.IsInstanceOfType(include) ||
                        ↪  requiredType.IsAssignableFrom(include.GetType()))
                        {
                            resultInstances.Add(include);
                        }
                    }
                }
                if (resultInstances.Count == 0 && resultConstructors.Count == 0)
                {
                    return false;
                }
                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);
            }
        }

        /// <summary>
        /// <para>
        /// Sorts the constructors using the specified result constructors.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="resultConstructors">
        /// <para>The result constructors.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual void SortConstructors(List<ConstructorInfo> resultConstructors) =>
        ↪  resultConstructors.Sort((x, y) =>
        ↪  -x.GetParameters().Length.CompareTo(y.GetParameters().Length));

        /// <summary>
        /// <para>
        /// Determines whether this instance try resolve instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="constructors">
        /// <para>The constructors.</para>
        /// <para></para>
        /// </param>
        /// <param name="resolved">
        /// <para>The resolved.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [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)
        {
            var parameters = constructor.GetParameters();
            arguments = new object[parameters.Length];
            for (var i = 0; i < parameters.Length; i++)
            {
                if (!TryResolve(parameters[i].ParameterType, out object argument))
                {
                    return false;
                }
                Use(argument);
                arguments[i] = argument;
            }
            return true;
        }

        #endregion

        /// <summary>
        /// <para>
        /// Disposes the manual.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="manual">
        /// <para>The manual.</para>
        /// <para></para>
        /// </param>
        /// <param name="wasDisposed">
        /// <para>The was disposed.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void Dispose(bool manual, bool wasDisposed)
        {
            if (!wasDisposed)
            {
                while (_dependencies.Count > 0)
                {
                    _dependencies.Pop().DisposeIfPossible();
                }
            }
        }
    }
```

```
}
```

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

```
using System.Runtime.CompilerServices;

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

namespace Platform.Scopes
{
    /// <summary>
    /// <para>
    /// Represents the scope.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="Scope"/>
    public class Scope<TInclude> : Scope
    {
        /// <summary>
        /// <para>
        /// Initializes a new <see cref="Scope"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public Scope() : this(false, false) { }

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="Scope"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="autoInclude">
        /// <para>A auto include.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public Scope(bool autoInclude) : this(autoInclude, false) { }

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="Scope"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="autoInclude">
        /// <para>A auto include.</para>
        /// <para></para>
        /// </param>
        /// <param name="autoExplore">
        /// <para>A auto explore.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public Scope(bool autoInclude, bool autoExplore) : base(autoInclude, autoExplore) =>
        ↪  Include<TInclude>();
    }
}
```

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

```
using System.Runtime.CompilerServices;
using Platform.Disposables;

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

namespace Platform.Scopes
{
    /// <summary>
    /// <para>
    /// Represents the use.
    /// </para>
    /// <para></para>
    /// </summary>
    public static class Use<T>
    {
        /// <summary>
        /// <para>
```

```
        /// Gets the single value.
        /// </para>
        /// <para></para>
        /// </summary>
        public static T Single
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get => Scope.Global.Use<T>();
        }

        /// <summary>
        /// <para>
        /// Gets the new value.
        /// </para>
        /// <para></para>
        /// </summary>
        public static Disposable<T> New
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get
            {
                var scope = new Scope(autoInclude: true, autoExplore: true);
                return new Disposable<T, Scope>(scope.Use<T>(), scope);
            }
        }
    }
}
```

1.4 ./csharp/Platform.Scopes.Tests/ScopeTests.cs

```
using Xunit;
using Platform.Reflection;

namespace Platform.Scopes.Tests
{
    /// <summary>
    /// <para>
    /// Represents the scope tests.
    /// </para>
    /// <para></para>
    /// </summary>
    public class ScopeTests
    {
        /// <summary>
        /// <para>
        /// Defines the interface.
        /// </para>
        /// <para></para>
        /// </summary>
        public interface IInterface
        {
        }

        /// <summary>
        /// <para>
        /// Represents the .
        /// </para>
        /// <para></para>
        /// </summary>
        /// <seealso cref="IInterface"/>
        public class Class : IInterface
        {
        }

        /// <summary>
        /// <para>
        /// Tests that single dependency test.
        /// </para>
        /// <para></para>
        /// </summary>
        [Fact]
        public static void SingleDependencyTest()
        {
            using var scope = new Scope();
            scope.IncludeAssemblyOf<IInterface>();
            var instance = scope.Use<IInterface>();
            Assert.IsType<Class>(instance);
        }

        /// <summary>
```

```
51          /// <para>
52          /// Tests that type parameters test.
53          /// </para>
54          /// <para></para>
55          /// </summary>
56          [Fact]
57          public static void TypeParametersTest()
58          {
59              using var scope = new Scope<Types<Class>>();
60              var instance = scope.Use<IInterface>();
61              Assert.IsType<Class>(instance);
62          }
63      }
64  }
```

Index