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">
```

```
307         /// <para>The .</para>
308         /// <para></para>
309         /// </typeparam>
310         /// <param name="creator">
311         /// <para>The creator.</para>
312         /// <para></para>
313         /// </param>
314         /// <returns>
315         /// <para>The</para>
316         /// <para></para>
317         /// </returns>
318         [MethodImpl(MethodImplOptions.AggressiveInlining)]
319         public T UseSingleton<T>(Func<T> creator) => UseAndReturn(Singleton.Get(creator));
320 
321         /// <summary>
322         /// <para>
323         /// Uses the and return using the specified object.
324         /// </para>
325         /// <para></para>
326         /// </summary>
327         /// <typeparam name="T">
328         /// <para>The .</para>
329         /// <para></para>
330         /// </typeparam>
331         /// <param name="@object">
332         /// <para>The object.</para>
333         /// <para></para>
334         /// </param>
335         /// <returns>
336         /// <para>The object.</para>
337         /// <para></para>
338         /// </returns>
339         [MethodImpl(MethodImplOptions.AggressiveInlining)]
340         public T UseAndReturn<T>(T @object)
341         {
342             Use(@object);
343             return @object;
344         }
345 
346         /// <summary>
347         /// <para>
348         /// Uses the object.
349         /// </para>
350         /// <para></para>
351         /// </summary>
352         /// <param name="@object">
353         /// <para>The object.</para>
354         /// <para></para>
355         /// </param>
356         [MethodImpl(MethodImplOptions.AggressiveInlining)]
357         public void Use(object @object)
358         {
359             Include(@object);
360             _dependencies.Push(@object);
361         }
362 
363         #endregion
364 
365         #region Resolve
366 
367         /// <summary>
368         /// <para>
369         /// Determines whether this instance try resolve.
370         /// </para>
371         /// <para></para>
372         /// </summary>
373         /// <typeparam name="T">
374         /// <para>The .</para>
375         /// <para></para>
376         /// </typeparam>
377         /// <param name="resolved">
378         /// <para>The resolved.</para>
379         /// <para></para>
380         /// </param>
381         /// <returns>
382         /// <para>The result.</para>
383         /// <para></para>
384         /// </returns>
```

```
385         [MethodImpl(MethodImplOptions.AggressiveInlining)]
386         public bool TryResolve<T>(out T resolved)
387         {
388             resolved = default;
389             var result = TryResolve(typeof(T), out object resolvedObject);
390             if (result)
391             {
392                 resolved = (T)resolvedObject;
393             }
394             return result;
395         }
396 
397         /// <summary>
398         /// <para>
399         /// Determines whether this instance try resolve.
400         /// </para>
401         /// <para></para>
402         /// </summary>
403         /// <param name="requiredType">
404         /// <para>The required type.</para>
405         /// <para></para>
406         /// </param>
407         /// <param name="resolved">
408         /// <para>The resolved.</para>
409         /// <para></para>
410         /// </param>
411         /// <returns>
412         /// <para>The bool</para>
413         /// <para></para>
414         /// </returns>
415         [MethodImpl(MethodImplOptions.AggressiveInlining)]
416         public bool TryResolve(Type requiredType, out object resolved)
417         {
418             resolved = null;
419             if (!_blocked.Add(requiredType))
420             {
421                 return false;
422             }
423             try
424             {
425                 if (_excludes.Contains(requiredType))
426                 {
427                     return false;
428                 }
429                 if (_resolutions.TryGetValue(requiredType, out resolved))
430                 {
431                     return true;
432                 }
433                 if (_autoExplore)
434                 {
435                     IncludeAssemblyOfType(requiredType);
436                 }
437                 var resultInstances = new List<object>();
438                 var resultConstructors = new List<ConstructorInfo>();
439                 foreach (var include in _includes)
440                 {
441                     if (!_excludes.Contains(include))
442                     {
443                         var type = include as Type;
444                         if (type != null)
445                         {
446                             if (requiredType.IsAssignableFrom(type))
447                             {
448                                 resultConstructors.AddRange(GetValidConstructors(type));
449                             }
450                             else if (type.GetTypeInfo().IsGenericTypeDefinition &&
                                 ↪  requiredType.GetTypeInfo().IsGenericType &&
                                 ↪  type.GetInterfaces().Any(x => x.Name == requiredType.Name))
451                             {
452                                 var genericType =
                                     ↪  type.MakeGenericType(requiredType.GenericTypeArguments);
453                                 if (requiredType.IsAssignableFrom(genericType))
454                                 {
455                                     resultConstructors.AddRange(GetValidConstructors(genericType
                                         ↪  ));
456                                 }
457                             }
458                         }
```

```
                        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;
    }

    /// <summary>
    /// <para>
    /// Gets the valid constructors using the specified type.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <param name="type">
    /// <para>The type.</para>
    /// <para></para>
    /// </param>
    /// <returns>
    /// <para>The constructors.</para>
    /// <para></para>
    /// </returns>
    [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;
    }

    /// <summary>
    /// <para>
    /// Determines whether this instance try resolve constructor arguments.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <param name="constructor">
    /// <para>The constructor.</para>
    /// <para></para>
    /// </param>
    /// <param name="arguments">
    /// <para>The arguments.</para>
    /// <para></para>
    /// </param>
    /// <returns>
    /// <para>The bool</para>
    /// <para></para>
    /// </returns>
    [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>
        /// <para>
        /// Tests that type parameters test.
        /// </para>
        /// <para></para>
        /// </summary>
        [Fact]
        public static void TypeParametersTest()
        {
            using var scope = new Scope<Types<Class>>();
            var instance = scope.Use<IInterface>();
            Assert.IsType<Class>(instance);
        }
    }
}
```

Index