## 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);

        /// <summary>
        /// <para>
        /// The auto include.
        /// </para>
        /// <para></para>
        /// </summary>
        private readonly bool _autoInclude;
        /// <summary>
        /// <para>
        /// The auto explore.
        /// </para>
        /// <para></para>
        /// </summary>
        private readonly bool _autoExplore;
        /// <summary>
        /// <para>
        /// The stack.
        /// </para>
        /// <para></para>
        /// </summary>
        private readonly Stack<object> _dependencies = new Stack<object>();
        /// <summary>
        /// <para>
        /// The hash set.
        /// </para>
        /// <para></para>
        /// </summary>
        private readonly HashSet<object> _excludes = new HashSet<object>();
        /// <summary>
        /// <para>
        /// The hash set.
        /// </para>
        /// <para></para>
        /// </summary>
        private readonly HashSet<object> _includes = new HashSet<object>();
        /// <summary>
        /// <para>
        /// The hash set.
        /// </para>
        /// <para></para>
        /// </summary>
        private readonly HashSet<object> _blocked = new HashSet<object>();
        /// <summary>
        /// <para>
```

```
        /// The type.
        /// </para>
        /// <para></para>
        /// </summary>
        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)
            {
```

```
307                 Include<T>();
308             }
309             if (!TryResolve(out T resolved))
310             {
311                 throw new InvalidOperationException($"Dependency of type {typeof(T).Name}
                    ↪  cannot be resolved.");
312             }
313             if (!_autoInclude)
314             {
315                 Include<T>();
316             }
317             Use(resolved);
318             return resolved;
319         }
320 
321         /// <summary>
322         /// <para>
323         /// Uses the singleton using the specified factory.
324         /// </para>
325         /// <para></para>
326         /// </summary>
327         /// <typeparam name="T">
328         /// <para>The .</para>
329         /// <para></para>
330         /// </typeparam>
331         /// <param name="factory">
332         /// <para>The factory.</para>
333         /// <para></para>
334         /// </param>
335         /// <returns>
336         /// <para>The</para>
337         /// <para></para>
338         /// </returns>
339         [MethodImpl(MethodImplOptions.AggressiveInlining)]
340         public T UseSingleton<T>(IFactory<T> factory) => UseAndReturn(Singleton.Get(factory));
341 
342         /// <summary>
343         /// <para>
344         /// Uses the singleton using the specified creator.
345         /// </para>
346         /// <para></para>
347         /// </summary>
348         /// <typeparam name="T">
349         /// <para>The .</para>
350         /// <para></para>
351         /// </typeparam>
352         /// <param name="creator">
353         /// <para>The creator.</para>
354         /// <para></para>
355         /// </param>
356         /// <returns>
357         /// <para>The</para>
358         /// <para></para>
359         /// </returns>
360         [MethodImpl(MethodImplOptions.AggressiveInlining)]
361         public T UseSingleton<T>(Func<T> creator) => UseAndReturn(Singleton.Get(creator));
362 
363         /// <summary>
364         /// <para>
365         /// Uses the and return using the specified object.
366         /// </para>
367         /// <para></para>
368         /// </summary>
369         /// <typeparam name="T">
370         /// <para>The .</para>
371         /// <para></para>
372         /// </typeparam>
373         /// <param name="@object">
374         /// <para>The object.</para>
375         /// <para></para>
376         /// </param>
377         /// <returns>
378         /// <para>The object.</para>
379         /// <para></para>
380         /// </returns>
381         [MethodImpl(MethodImplOptions.AggressiveInlining)]
382         public T UseAndReturn<T>(T @object)
383         {
```

```
            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)]
        public bool TryResolve<T>(out T resolved)
        {
            resolved = default;
            var result = TryResolve(typeof(T), out object resolvedObject);
            if (result)
            {
                resolved = (T)resolvedObject;
            }
            return result;
        }

        /// <summary>
        /// <para>
        /// Determines whether this instance try resolve.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="requiredType">
        /// <para>The required type.</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)]
        public bool TryResolve(Type requiredType, out object resolved)
        {
            resolved = null;
            if (!_blocked.Add(requiredType))
            {
```

```
463                 return false;
464             }
465             try
466             {
467                 if (_excludes.Contains(requiredType))
468                 {
469                     return false;
470                 }
471                 if (_resolutions.TryGetValue(requiredType, out resolved))
472                 {
473                     return true;
474                 }
475                 if (_autoExplore)
476                 {
477                     IncludeAssemblyOfType(requiredType);
478                 }
479                 var resultInstances = new List<object>();
480                 var resultConstructors = new List<ConstructorInfo>();
481                 foreach (var include in _includes)
482                 {
483                     if (!_excludes.Contains(include))
484                     {
485                         var type = include as Type;
486                         if (type != null)
487                         {
488                             if (requiredType.IsAssignableFrom(type))
489                             {
490                                 resultConstructors.AddRange(GetValidConstructors(type));
491                             }
492                             else if (type.GetTypeInfo().IsGenericTypeDefinition &&
                                 ↪  requiredType.GetTypeInfo().IsGenericType &&
                                 ↪  type.GetInterfaces().Any(x => x.Name == requiredType.Name))
493                             {
494                                 var genericType =
                                     ↪  type.MakeGenericType(requiredType.GenericTypeArguments);
495                                 if (requiredType.IsAssignableFrom(genericType))
496                                 {
497                                     resultConstructors.AddRange(GetValidConstructors(genericType
                                         ↪  ));
498                                 }
499                             }
500                         }
501                         else if (requiredType.IsInstanceOfType(include) ||
                             ↪  requiredType.IsAssignableFrom(include.GetType()))
502                         {
503                             resultInstances.Add(include);
504                         }
505                     }
506                 }
507                 if (resultInstances.Count == 0 && resultConstructors.Count == 0)
508                 {
509                     return false;
510                 }
511                 else if (resultInstances.Count > 0)
512                 {
513                     resolved = resultInstances[0];
514                 }
515                 else
516                 {
517                     SortConstructors(resultConstructors);
518                     if (!TryResolveInstance(resultConstructors, out resolved))
519                     {
520                         return false;
521                     }
522                 }
523                 _resolutions.Add(requiredType, resolved);
524                 return true;
525             }
526             finally
527             {
528                 _blocked.Remove(requiredType);
529             }
530         }
531 
532         /// <summary>
533         /// <para>
534         /// Sorts the constructors using the specified result constructors.
535         /// </para>
```

```
536         /// <para></para>
537         /// </summary>
538         /// <param name="resultConstructors">
539         /// <para>The result constructors.</para>
540         /// <para></para>
541         /// </param>
542         [MethodImpl(MethodImplOptions.AggressiveInlining)]
543         protected virtual void SortConstructors(List<ConstructorInfo> resultConstructors) =>
            ↪ resultConstructors.Sort((x, y) =>
            ↪ -x.GetParameters().Length.CompareTo(y.GetParameters().Length));
544
545         /// <summary>
546         /// <para>
547         /// Determines whether this instance try resolve instance.
548         /// </para>
549         /// <para></para>
550         /// </summary>
551         /// <param name="constructors">
552         /// <para>The constructors.</para>
553         /// <para></para>
554         /// </param>
555         /// <param name="resolved">
556         /// <para>The resolved.</para>
557         /// <para></para>
558         /// </param>
559         /// <returns>
560         /// <para>The bool</para>
561         /// <para></para>
562         /// </returns>
563         [MethodImpl(MethodImplOptions.AggressiveInlining)]
564         protected virtual bool TryResolveInstance(List<ConstructorInfo> constructors, out object
            ↪ resolved)
565         {
566             for (var i = 0; i < constructors.Count; i++)
567             {
568                 try
569                 {
570                     var resultConstructor = constructors[i];
571                     if (TryResolveConstructorArguments(resultConstructor, out object[]
                        ↪ arguments))
572                     {
573                         resolved = resultConstructor.Invoke(arguments);
574                         return true;
575                     }
576                 }
577                 catch (Exception exception)
578                 {
579                     exception.Ignore();
580                 }
581             }
582             resolved = null;
583             return false;
584         }
585
586         /// <summary>
587         /// <para>
588         /// Gets the valid constructors using the specified type.
589         /// </para>
590         /// <para></para>
591         /// </summary>
592         /// <param name="type">
593         /// <para>The type.</para>
594         /// <para></para>
595         /// </param>
596         /// <returns>
597         /// <para>The constructors.</para>
598         /// <para></para>
599         /// </returns>
600         [MethodImpl(MethodImplOptions.AggressiveInlining)]
601         private ConstructorInfo[] GetValidConstructors(Type type)
602         {
603             var constructors = type.GetConstructors();
604             if (!_autoExplore)
605             {
606                 constructors = constructors.ToArray(x =>
607                 {
608                     var parameters = x.GetParameters();
609                     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>
```

```
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