1.1 ./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
{
    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)
            {
```

```
78                  types.ForEach(Include);
79              }
80              else
81              {
82                  Include(typeof(T));
83              }
84          }
85
86          [MethodImpl(MethodImplOptions.AggressiveInlining)]
87          public void Include(object @object)
88          {
89              if (@object == null)
90              {
91                  return;
92              }
93              if (_includes.Add(@object))
94              {
95                  var type = @object as Type;
96                  if (type != null)
97                  {
98                      type.GetInterfaces().ForEach(Include);
99                      Include(type.GetBaseType());
100                 }
101             }
102         }
103
104         #endregion
105
106         #region Use
107
108         /// <remarks>
109         /// TODO: Use Default[T].Instance if the only constructor object has is parameterless.
110         /// TODO: Think of interface chaining IDoubletLinks[T] (default) -> IDoubletLinks[T]
            ↪  (checker) -> IDoubletLinks[T] (synchronizer) (may be UseChain[IDoubletLinks[T],
            ↪  Types[DefaultLinks, DefaultLinksDependencyChecker, DefaultSynchronizedLinks]]
111         /// TODO: Add support for factories
112         /// </remarks>
113         [MethodImpl(MethodImplOptions.AggressiveInlining)]
114         public T Use<T>()
115         {
116             if (_excludes.Contains(typeof(T)))
117             {
118                 throw new InvalidOperationException($"Type {typeof(T).Name} is excluded and
                    ↪  cannot be used.");
119             }
120             if (_autoInclude)
121             {
122                 Include<T>();
123             }
124             if (!TryResolve(out T resolved))
125             {
126                 throw new InvalidOperationException($"Dependency of type {typeof(T).Name}
                    ↪  cannot be resolved.");
127             }
128             if (!_autoInclude)
129             {
130                 Include<T>();
131             }
132             Use(resolved);
133             return resolved;
134         }
135
136         [MethodImpl(MethodImplOptions.AggressiveInlining)]
137         public T UseSingleton<T>(IFactory<T> factory) => UseAndReturn(Singleton.Get(factory));
138
139         [MethodImpl(MethodImplOptions.AggressiveInlining)]
140         public T UseSingleton<T>(Func<T> creator) => UseAndReturn(Singleton.Get(creator));
141
142         [MethodImpl(MethodImplOptions.AggressiveInlining)]
143         public T UseAndReturn<T>(T @object)
144         {
145             Use(@object);
146             return @object;
147         }
148
149         [MethodImpl(MethodImplOptions.AggressiveInlining)]
150         public void Use(object @object)
151         {
```

```
            Include(@object);
            _dependencies.Push(@object);
        }

        #endregion

        #region Resolve

        [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;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public bool TryResolve(Type requiredType, out object resolved)
        {
            resolved = null;
            if (!_blocked.Add(requiredType))
            {
                return false;
            }
            try
            {
                if (_excludes.Contains(requiredType))
                {
                    return false;
                }
                if (_resolutions.TryGetValue(requiredType, out resolved))
                {
                    return true;
                }
                if (_autoExplore)
                {
                    IncludeAssemblyOfType(requiredType);
                }
                var resultInstances = new List<object>();
                var resultConstructors = new List<ConstructorInfo>();
                foreach (var include in _includes)
                {
                    if (!_excludes.Contains(include))
                    {
                        var type = include as Type;
                        if (type != null)
                        {
                            if (requiredType.IsAssignableFrom(type))
                            {
                                resultConstructors.AddRange(GetValidConstructors(type));
                            }
                            else if (type.GetTypeInfo().IsGenericTypeDefinition &&
                            ↪  requiredType.GetTypeInfo().IsGenericType &&
                            ↪  type.GetInterfaces().Any(x => x.Name == requiredType.Name))
                            {
                                var genericType =
                                ↪  type.MakeGenericType(requiredType.GenericTypeArguments);
                                if (requiredType.IsAssignableFrom(genericType))
                                {
                                    resultConstructors.AddRange(GetValidConstructors(genericType
                                    ↪  ));
                                }
                            }
                        }
                        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);
            }
        }

        [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)
        {
```

```
            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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void Dispose(bool manual, bool wasDisposed)
        {
            if (!wasDisposed)
            {
                while (_dependencies.Count > 0)
                {
                    _dependencies.Pop().DisposeIfPossible();
                }
            }
        }
    }
}
```

1.2 ./Scope[TInclude].cs

```
using System.Runtime.CompilerServices;

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

namespace Platform.Scopes
{
    public class Scope<TInclude> : Scope
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public Scope() : this(false, false) { }

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public Scope(bool autoInclude, bool autoExplore) : base(autoInclude, autoExplore) =>
        ↪  Include<TInclude>();
    }
}
```

1.3 ./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
{
    public static class Use<T>
    {
        public static T Single
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get => Scope.Global.Use<T>();
        }

        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);
            }
        }
    }
}
```

Index