./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;

#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>();

        public Scope(bool autoInclude, bool autoExplore)
        {
            _autoInclude = autoInclude;
            _autoExplore = autoExplore;
        }

        public Scope(bool autoInclude) : this(autoInclude, false) { }

        public Scope() { }

        #region Exclude

        public void ExcludeAssemblyOf<T>() => ExcludeAssemblyOfType(typeof(T));

        public void ExcludeAssemblyOfType(Type type) => ExcludeAssembly(type.GetAssembly());

        public void ExcludeAssembly(Assembly assembly) =>
        ↪  assembly.GetCachedLoadableTypes().ForEach(Exclude);

        public void Exclude<T>() => Exclude(typeof(T));

        public void Exclude(object @object) => _excludes.Add(@object);

        #endregion

        #region Include

        public void IncludeAssemblyOf<T>() => IncludeAssemblyOfType(typeof(T));

        public void IncludeAssemblyOfType(Type type) => IncludeAssembly(type.GetAssembly());

        public void IncludeAssembly(Assembly assembly) =>
        ↪  assembly.GetExportedTypes().ForEach(Include);

        public void Include<T>()
        {
            var types = Types<T>.Array;
            if (types.Length > 0)
            {
                types.ForEach(Include);
            }
            else
            {
                Include(typeof(T));
            }
        }

        public void Include(object @object)
        {
            if (@object == null)
            {
                return;
```

```
78              }
79              if (_includes.Add(@object))
80              {
81                  var type = @object as Type;
82                  if (type != null)
83                  {
84                      type.GetInterfaces().ForEach(Include);
85                      Include(type.GetBaseType());
86                  }
87              }
88          }
89
90          #endregion
91
92          #region Use
93
94          /// <remarks>
95          /// TODO: Use Default[T].Instance if the only constructor object has is parameterless.
96          /// TODO: Think of interface chaining IDoubletLinks[T] (default) -> IDoubletLinks[T]
            ↪  (checker) -> IDoubletLinks[T] (synchronizer) (may be UseChain[IDoubletLinks[T],
            ↪  Types[DefaultLinks, DefaultLinksDependencyChecker, DefaultSynchronizedLinks]]
97          /// TODO: Add support for factories
98          /// </remarks>
99          public T Use<T>()
100         {
101             if (_excludes.Contains(typeof(T)))
102             {
103                 throw new InvalidOperationException($"Type {typeof(T).Name} is excluded and
                    ↪  cannot be used.");
104             }
105             if (_autoInclude)
106             {
107                 Include<T>();
108             }
109             if (!TryResolve(out T resolved))
110             {
111                 throw new InvalidOperationException($"Dependency of type {typeof(T).Name}
                    ↪  cannot be resolved.");
112             }
113             if (!_autoInclude)
114             {
115                 Include<T>();
116             }
117             Use(resolved);
118             return resolved;
119         }
120
121         public T UseSingleton<T>(IFactory<T> factory) => UseAndReturn(Singleton.Get(factory));
122
123         public T UseSingleton<T>(Func<T> creator) => UseAndReturn(Singleton.Get(creator));
124
125         public T UseAndReturn<T>(T @object)
126         {
127             Use(@object);
128             return @object;
129         }
130
131         public void Use(object @object)
132         {
133             Include(@object);
134             _dependencies.Push(@object);
135         }
136
137         #endregion
138
139         #region Resolve
140
141         public bool TryResolve<T>(out T resolved)
142         {
143             resolved = default;
144             var result = TryResolve(typeof(T), out object resolvedObject);
145             if (result)
146             {
147                 resolved = (T)resolvedObject;
148             }
149             return result;
150         }
151
152         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);
            }
        }

```

```
226         protected virtual void SortConstructors(List<ConstructorInfo> resultConstructors) =>
            ↪ resultConstructors.Sort((x, y) =>
            ↪ -x.GetParameters().Length.CompareTo(y.GetParameters().Length));
227
228         protected virtual bool TryResolveInstance(List<ConstructorInfo> constructors, out object
            ↪ resolved)
229         {
230             for (var i = 0; i < constructors.Count; i++)
231             {
232                 try
233                 {
234                     var resultConstructor = constructors[i];
235                     if (TryResolveConstructorArguments(resultConstructor, out object[]
                        ↪ arguments))
236                     {
237                         resolved = resultConstructor.Invoke(arguments);
238                         return true;
239                     }
240                 }
241                 catch (Exception exception)
242                 {
243                     exception.Ignore();
244                 }
245             }
246             resolved = null;
247             return false;
248         }
249
250         private ConstructorInfo[] GetValidConstructors(Type type)
251         {
252             var constructors = type.GetConstructors();
253             if (!_autoExplore)
254             {
255                 constructors = constructors.ToArray(x =>
256                 {
257                     var parameters = x.GetParameters();
258                     for (var i = 0; i < parameters.Length; i++)
259                     {
260                         if (!_includes.Contains(parameters[i].ParameterType))
261                         {
262                             return false;
263                         }
264                     }
265                     return true;
266                 });
267             }
268             return constructors;
269         }
270
271         private bool TryResolveConstructorArguments(ConstructorInfo constructor, out object[]
            ↪ arguments)
272         {
273             var parameters = constructor.GetParameters();
274             arguments = new object[parameters.Length];
275             for (var i = 0; i < parameters.Length; i++)
276             {
277                 if (!TryResolve(parameters[i].ParameterType, out object argument))
278                 {
279                     return false;
280                 }
281                 Use(argument);
282                 arguments[i] = argument;
283             }
284             return true;
285         }
286
287         #endregion
288
289         protected override void Dispose(bool manual, bool wasDisposed)
290         {
291             if (!wasDisposed)
292             {
293                 while (_dependencies.Count > 0)
294                 {
295                     _dependencies.Pop().DisposeIfPossible();
296                 }
297             }
298         }
```

```
    }
}
```

./Scope[TInclude].cs

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

namespace Platform.Scopes
{
    public class Scope<TInclude> : Scope
    {
        public Scope() : this(false, false) { }
        public Scope(bool autoInclude) : this(autoInclude, false) { }
        public Scope(bool autoInclude, bool autoExplore) : base(autoInclude, autoExplore) =>
        ↪  Include<TInclude>();
    }
}
```

./Use.cs

```
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 => Scope.Global.Use<T>();

        public static Disposable<T> New
        {
            get
            {
                var scope = new Scope(autoInclude: true, autoExplore: true);
                return new Disposable<T, Scope>(scope.Use<T>(), scope);
            }
        }
    }
}
```

Index