

LinksPlatform's Platform.Scopes Class Library

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

```
1  using System;
2  using System.Collections.Generic;
3  using System.Reflection;
4  using System.Linq;
5  using Platform.Interfaces;
6  using Platform.Exceptions;
7  using Platform.Disposables;
8  using Platform.Collections.Lists;
9  using Platform.Reflection;
10 using Platform.Singletons;
11 using System.Runtime.CompilerServices;
12
13 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
14
15 namespace Platform.Scopes
16 {
17     /// <summary>
18     /// <para>
19     /// Represents the scope.
20     /// </para>
21     /// <para></para>
22     /// </summary>
23     /// <seealso cref="DisposableBase"/>
24     public class Scope : DisposableBase
25     {
26         /// <summary>
27         /// <para>
28         /// The auto explore.
29         /// </para>
30         /// <para></para>
31         /// </summary>
32         public static readonly Scope Global = new Scope(autoInclude: true, autoExplore: true);
33
34         /// <summary>
35         /// <para>
36         /// The auto include.
37         /// </para>
38         /// <para></para>
39         /// </summary>
40         private readonly bool _autoInclude;
41         /// <summary>
42         /// <para>
43         /// The auto explore.
44         /// </para>
45         /// <para></para>
46         /// </summary>
47         private readonly bool _autoExplore;
48         /// <summary>
49         /// <para>
50         /// The stack.
51         /// </para>
52         /// <para></para>
53         /// </summary>
54         private readonly Stack<object> _dependencies = new Stack<object>();
55         /// <summary>
56         /// <para>
57         /// The hash set.
58         /// </para>
59         /// <para></para>
60         /// </summary>
61         private readonly HashSet<object> _excludes = new HashSet<object>();
62         /// <summary>
63         /// <para>
64         /// The hash set.
65         /// </para>
66         /// <para></para>
67         /// </summary>
68         private readonly HashSet<object> _includes = new HashSet<object>();
69         /// <summary>
70         /// <para>
71         /// The hash set.
72         /// </para>
73         /// <para></para>
74         /// </summary>
75         private readonly HashSet<object> _blocked = new HashSet<object>();
76         /// <summary>
77         /// <para>
```

```

78     /// The type.
79     /// </para>
80     /// <para></para>
81     /// </summary>
82     private readonly Dictionary<Type, object> _resolutions = new Dictionary<Type, object>();
83
84     /// <summary>
85     /// <para>
86     /// Initializes a new <see cref="Scope"/> instance.
87     /// </para>
88     /// <para></para>
89     /// </summary>
90     /// <param name="autoInclude">
91     /// <para>A auto include.</para>
92     /// <para></para>
93     /// </param>
94     /// <param name="autoExplore">
95     /// <para>A auto explore.</para>
96     /// <para></para>
97     /// </param>
98     [MethodImpl(MethodImplOptions.AggressiveInlining)]
99     public Scope(bool autoInclude, bool autoExplore)
100     {
101         _autoInclude = autoInclude;
102         _autoExplore = autoExplore;
103     }
104
105     /// <summary>
106     /// <para>
107     /// Initializes a new <see cref="Scope"/> instance.
108     /// </para>
109     /// <para></para>
110     /// </summary>
111     /// <param name="autoInclude">
112     /// <para>A auto include.</para>
113     /// <para></para>
114     /// </param>
115     [MethodImpl(MethodImplOptions.AggressiveInlining)]
116     public Scope(bool autoInclude) : this(autoInclude, false) { }
117
118     /// <summary>
119     /// <para>
120     /// Initializes a new <see cref="Scope"/> instance.
121     /// </para>
122     /// <para></para>
123     /// </summary>
124     [MethodImpl(MethodImplOptions.AggressiveInlining)]
125     public Scope() { }
126
127     #region Exclude
128
129     /// <summary>
130     /// <para>
131     /// Excludes the assembly of.
132     /// </para>
133     /// <para></para>
134     /// </summary>
135     /// <typeparam name="T">
136     /// <para>The .</para>
137     /// <para></para>
138     /// </typeparam>
139     [MethodImpl(MethodImplOptions.AggressiveInlining)]
140     public void ExcludeAssemblyOf<T>() => ExcludeAssemblyOfType(typeof(T));
141
142     /// <summary>
143     /// <para>
144     /// Excludes the assembly of type using the specified type.
145     /// </para>
146     /// <para></para>
147     /// </summary>
148     /// <param name="type">
149     /// <para>The type.</para>
150     /// <para></para>
151     /// </param>
152     [MethodImpl(MethodImplOptions.AggressiveInlining)]
153     public void ExcludeAssemblyOfType(Type type) => ExcludeAssembly(type.GetAssembly());
154
155     /// <summary>

```

```

156     /// <para>
157     /// Excludes the assembly using the specified assembly.
158     /// </para>
159     /// <para></para>
160     /// </summary>
161     /// <param name="assembly">
162     /// <para>The assembly.</para>
163     /// <para></para>
164     /// </param>
165     [MethodImpl(MethodImplOptions.AggressiveInlining)]
166     public void ExcludeAssembly(Assembly assembly) =>
167         ↪ assembly.GetCachedLoadableTypes().ForEach(Exclude);
168
169     /// <summary>
170     /// <para>
171     /// Excludes this instance.
172     /// </para>
173     /// <para></para>
174     /// </summary>
175     /// <typeparam name="T">
176     /// <para>The .</para>
177     /// <para></para>
178     /// </typeparam>
179     [MethodImpl(MethodImplOptions.AggressiveInlining)]
180     public void Exclude<T>() => Exclude(typeof(T));
181
182     /// <summary>
183     /// <para>
184     /// Excludes the object.
185     /// </para>
186     /// <para></para>
187     /// </summary>
188     /// <param name="@object">
189     /// <para>The object.</para>
190     /// <para></para>
191     /// </param>
192     [MethodImpl(MethodImplOptions.AggressiveInlining)]
193     public void Exclude(object @object) => _excludes.Add(@object);
194
195     #endregion
196
197     #region Include
198
199     /// <summary>
200     /// <para>
201     /// Includes the assembly of.
202     /// </para>
203     /// <para></para>
204     /// </summary>
205     /// <typeparam name="T">
206     /// <para>The .</para>
207     /// <para></para>
208     /// </typeparam>
209     [MethodImpl(MethodImplOptions.AggressiveInlining)]
210     public void IncludeAssemblyOf<T>() => IncludeAssemblyOfType(typeof(T));
211
212     /// <summary>
213     /// <para>
214     /// Includes the assembly of type using the specified type.
215     /// </para>
216     /// <para></para>
217     /// </summary>
218     /// <param name="type">
219     /// <para>The type.</para>
220     /// <para></para>
221     /// </param>
222     [MethodImpl(MethodImplOptions.AggressiveInlining)]
223     public void IncludeAssemblyOfType(Type type) => IncludeAssembly(type.GetAssembly());
224
225     /// <summary>
226     /// <para>
227     /// Includes the assembly using the specified assembly.
228     /// </para>
229     /// <para></para>
230     /// </summary>
231     /// <param name="assembly">
232     /// <para>The assembly.</para>
233     /// <para></para>

```

```

233     /// </param>
234     [MethodImpl(MethodImplOptions.AggressiveInlining)]
235     public void IncludeAssembly(Assembly assembly) =>
236         ↪ assembly.GetExportedTypes().ForEach(Include);
237
238     /// <summary>
239     /// <para>
240     /// Includes this instance.
241     /// </para>
242     /// <para></para>
243     /// </summary>
244     /// <typeparam name="T">
245     /// <para>The .</para>
246     /// <para></para>
247     /// </typeparam>
248     [MethodImpl(MethodImplOptions.AggressiveInlining)]
249     public void Include<T>()
250     {
251         var types = Types<T>.Array;
252         if (types.Length > 0)
253         {
254             types.ForEach(Include);
255         }
256         else
257         {
258             Include(typeof(T));
259         }
260     }
261
262     /// <summary>
263     /// <para>
264     /// Includes the object.
265     /// </para>
266     /// <para></para>
267     /// </summary>
268     /// <param name="@object">
269     /// <para>The object.</para>
270     /// <para></para>
271     /// </param>
272     [MethodImpl(MethodImplOptions.AggressiveInlining)]
273     public void Include(object @object)
274     {
275         if (@object == null)
276         {
277             return;
278         }
279         if (_includes.Add(@object))
280         {
281             var type = @object as Type;
282             if (type != null)
283             {
284                 type.GetInterfaces().ForEach(Include);
285                 Include(type.GetBaseType());
286             }
287         }
288     }
289
290 #endregion
291
292 #region Use
293
294     /// <remarks>
295     /// TODO: Use Default[T].Instance if the only constructor object has is parameterless.
296     /// TODO: Think of interface chaining IDoubletLinks[T] (default) -> IDoubletLinks[T]
297     ↪ (checker) -> IDoubletLinks[T] (synchronizer) (may be UseChain[IDoubletLinks[T],
298     ↪ Types[DefaultLinks, DefaultLinksDependencyChecker, DefaultSynchronizedLinks]])
299     /// TODO: Add support for factories
300     /// </remarks>
301     [MethodImpl(MethodImplOptions.AggressiveInlining)]
302     public T Use<T>()
303     {
304         if (_excludes.Contains(typeof(T)))
305         {
306             throw new InvalidOperationException($"Type {typeof(T).Name} is excluded and
307             ↪ cannot be used.");
308         }
309         if (_autoInclude)
310         {

```

```

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 /// </summary>
326 /// <typeparam name="T">
327 /// <para>The .</para>
328 /// <para></para>
329 /// </typeparam>
330 /// <param name="factory">
331 /// <para>The factory.</para>
332 /// <para></para>
333 /// </param>
334 /// <returns>
335 /// <para>The</para>
336 /// <para></para>
337 /// </returns>
338 [MethodImpl(MethodImplOptions.AggressiveInlining)]
339 public T UseSingleton<T>(IFactory<T> factory) => UseAndReturn(Singleton.Get(factory));
340
341 /// <summary>
342 /// <para>
343 /// Uses the singleton using the specified creator.
344 /// </para>
345 /// </summary>
346 /// <typeparam name="T">
347 /// <para>The .</para>
348 /// <para></para>
349 /// </typeparam>
350 /// <param name="creator">
351 /// <para>The creator.</para>
352 /// <para></para>
353 /// </param>
354 /// <returns>
355 /// <para>The</para>
356 /// <para></para>
357 /// </returns>
358 [MethodImpl(MethodImplOptions.AggressiveInlining)]
359 public T UseSingleton<T>(Func<T> creator) => UseAndReturn(Singleton.Get(creator));
360
361 /// <summary>
362 /// <para>
363 /// Uses the and return using the specified object.
364 /// </para>
365 /// </summary>
366 /// <typeparam name="T">
367 /// <para>The .</para>
368 /// <para></para>
369 /// </typeparam>
370 /// <param name="@object">
371 /// <para>The object.</para>
372 /// <para></para>
373 /// </param>
374 /// <returns>
375 /// <para>The object.</para>
376 /// <para></para>
377 /// </returns>
378 [MethodImpl(MethodImplOptions.AggressiveInlining)]
379 public T UseAndReturn<T>(T @object)
380 {
381
382
383

```

```

384     Use(@object);
385     return @object;
386 }
387
388 /// <summary>
389 /// <para>
390 /// Uses the object.
391 /// </para>
392 /// <para></para>
393 /// </summary>
394 /// <param name="@object">
395 /// <para>The object.</para>
396 /// <para></para>
397 /// </param>
398 [MethodImpl(MethodImplOptions.AggressiveInlining)]
399 public void Use(object @object)
400 {
401     Include(@object);
402     _dependencies.Push(@object);
403 }
404
405 #endregion
406
407 #region Resolve
408
409 /// <summary>
410 /// <para>
411 /// Determines whether this instance try resolve.
412 /// </para>
413 /// <para></para>
414 /// </summary>
415 /// <typeparam name="T">
416 /// <para>The .</para>
417 /// <para></para>
418 /// </typeparam>
419 /// <param name="resolved">
420 /// <para>The resolved.</para>
421 /// <para></para>
422 /// </param>
423 /// <returns>
424 /// <para>The result.</para>
425 /// <para></para>
426 /// </returns>
427 [MethodImpl(MethodImplOptions.AggressiveInlining)]
428 public bool TryResolve<T>(out T resolved)
429 {
430     resolved = default;
431     var result = TryResolve(typeof(T), out object resolvedObject);
432     if (result)
433     {
434         resolved = (T)resolvedObject;
435     }
436     return result;
437 }
438
439 /// <summary>
440 /// <para>
441 /// Determines whether this instance try resolve.
442 /// </para>
443 /// <para></para>
444 /// </summary>
445 /// <param name="requiredType">
446 /// <para>The required type.</para>
447 /// <para></para>
448 /// </param>
449 /// <param name="resolved">
450 /// <para>The resolved.</para>
451 /// <para></para>
452 /// </param>
453 /// <returns>
454 /// <para>The bool</para>
455 /// <para></para>
456 /// </returns>
457 [MethodImpl(MethodImplOptions.AggressiveInlining)]
458 public bool TryResolve(Type requiredType, out object resolved)
459 {
460     resolved = null;
461     if (!_blocked.Add(requiredType))
462     {

```

```

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 &&
493                             ↪ requiredType.GetTypeInfo().IsGenericType &&
494                             ↪ type.GetInterfaces().Any(x => x.Name == requiredType.Name))
495                     {
496                         var genericType =
497                             ↪ type.MakeGenericType(requiredType.GenericTypeArguments);
498                         if (requiredType.IsAssignableFrom(genericType))
499                         {
500                             resultConstructors.AddRange(GetValidConstructors(genericType)
501                             ↪ );
502                         }
503                     }
504                 }
505             }
506             else if (requiredType.IsInstanceOfType(include) ||
507                     ↪ requiredType.IsAssignableFrom(include.GetType()))
508             {
509                 resultInstances.Add(include);
510             }
511         }
512         if (resultInstances.Count == 0 && resultConstructors.Count == 0)
513         {
514             return false;
515         }
516         else if (resultInstances.Count > 0)
517         {
518             resolved = resultInstances[0];
519         }
520         else
521         {
522             SortConstructors(resultConstructors);
523             if (!TryResolveInstance(resultConstructors, out resolved))
524             {
525                 return false;
526             }
527         }
528         _resolutions.Add(requiredType, resolved);
529         return true;
530     }
531     finally
532     {
533         _blocked.Remove(requiredType);
534     }
535 }

```

/// <summary>
 /// <para>
 /// Sorts the constructors using the specified result constructors.
 /// </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             catch (Exception exception)
577             {
578                 exception.Ignore();
579             }
580         }
581         resolved = null;
582         return false;
583     }
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++)

```



```

610         {
611             if (!_includes.Contains(parameters[i].ParameterType))
612             {
613                 return false;
614             }
615         }
616         return true;
617     });
618 }
619 return constructors;
620 }
621
622 /// <summary>
623 /// <para>
624 /// Determines whether this instance try resolve constructor arguments.
625 /// </para>
626 /// <para></para>
627 /// </summary>
628 /// <param name="constructor">
629 /// <para>The constructor.</para>
630 /// <para></para>
631 /// </param>
632 /// <param name="arguments">
633 /// <para>The arguments.</para>
634 /// <para></para>
635 /// </param>
636 /// <returns>
637 /// <para>The bool</para>
638 /// <para></para>
639 /// </returns>
640 [MethodImpl(MethodImplOptions.AggressiveInlining)]
641 private bool TryResolveConstructorArguments(ConstructorInfo constructor, out object[]
    ↪ arguments)
642 {
643     var parameters = constructor.GetParameters();
644     arguments = new object[parameters.Length];
645     for (var i = 0; i < parameters.Length; i++)
646     {
647         if (!TryResolve(parameters[i].ParameterType, out object argument))
648         {
649             return false;
650         }
651         Use(argument);
652         arguments[i] = argument;
653     }
654     return true;
655 }
656
657 #endregion
658
659 /// <summary>
660 /// <para>
661 /// Disposes the manual.
662 /// </para>
663 /// <para></para>
664 /// </summary>
665 /// <param name="manual">
666 /// <para>The manual.</para>
667 /// <para></para>
668 /// </param>
669 /// <param name="wasDisposed">
670 /// <para>The was disposed.</para>
671 /// <para></para>
672 /// </param>
673 [MethodImpl(MethodImplOptions.AggressiveInlining)]
674 protected override void Dispose(bool manual, bool wasDisposed)
675 {
676     if (!wasDisposed)
677     {
678         while (_dependencies.Count > 0)
679         {
680             _dependencies.Pop().DisposeIfPossible();
681         }
682     }
683 }
684 }
685 }

```

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

```
1 using System.Runtime.CompilerServices;
2
3 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5 namespace Platform.Scopes
6 {
7     /// <summary>
8     /// <para>
9     /// Represents the scope.
10    /// </para>
11    /// <para></para>
12    /// </summary>
13    /// <seealso cref="Scope"/>
14    public class Scope<TInclude> : Scope
15    {
16        /// <summary>
17        /// <para>
18        /// Initializes a new <see cref="Scope"/> instance.
19        /// </para>
20        /// <para></para>
21        /// </summary>
22        [MethodImpl(MethodImplOptions.AggressiveInlining)]
23        public Scope() : this(false, false) { }
24
25        /// <summary>
26        /// <para>
27        /// Initializes a new <see cref="Scope"/> instance.
28        /// </para>
29        /// <para></para>
30        /// </summary>
31        /// <param name="autoInclude">
32        /// <para>A auto include.</para>
33        /// <para></para>
34        /// </param>
35        [MethodImpl(MethodImplOptions.AggressiveInlining)]
36        public Scope(bool autoInclude) : this(autoInclude, false) { }
37
38        /// <summary>
39        /// <para>
40        /// Initializes a new <see cref="Scope"/> instance.
41        /// </para>
42        /// <para></para>
43        /// </summary>
44        /// <param name="autoInclude">
45        /// <para>A auto include.</para>
46        /// <para></para>
47        /// </param>
48        /// <param name="autoExplore">
49        /// <para>A auto explore.</para>
50        /// <para></para>
51        /// </param>
52        [MethodImpl(MethodImplOptions.AggressiveInlining)]
53        public Scope(bool autoInclude, bool autoExplore) : base(autoInclude, autoExplore) =>
54            ↪ Include<TInclude>();
55    }
56 }
```

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

```
1 using System.Runtime.CompilerServices;
2 using Platform.Disposables;
3
4 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6 namespace Platform.Scopes
7 {
8     /// <summary>
9     /// <para>
10    /// Represents the use.
11    /// </para>
12    /// <para></para>
13    /// </summary>
14    public static class Use<T>
15    {
16        /// <summary>
17        /// <para>
18        /// Gets the single value.
19        /// </para>
20        /// <para></para>
21    }
```

```

21     /// </summary>
22     public static T Single
23     {
24         [MethodImpl(MethodImplOptions.AggressiveInlining)]
25         get => Scope.Global.Use<T>();
26     }
27
28     /// <summary>
29     /// <para>
30     /// Gets the new value.
31     /// </para>
32     /// <para></para>
33     /// </summary>
34     public static Disposable<T> New
35     {
36         [MethodImpl(MethodImplOptions.AggressiveInlining)]
37         get
38         {
39             var scope = new Scope(autoInclude: true, autoExplore: true);
40             return new Disposable<T, Scope>(scope.Use<T>(), scope);
41         }
42     }
43 }
44 }

```

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

```

1  using Xunit;
2  using Platform.Reflection;
3
4  namespace Platform.Scopes.Tests
5  {
6      /// <summary>
7      /// <para>
8      /// Represents the scope tests.
9      /// </para>
10     /// <para></para>
11     /// </summary>
12     public class ScopeTests
13     {
14         /// <summary>
15         /// <para>
16         /// Defines the interface.
17         /// </para>
18         /// <para></para>
19         /// </summary>
20         public interface IInterface
21         {
22         }
23
24         /// <summary>
25         /// <para>
26         /// Represents the .
27         /// </para>
28         /// <para></para>
29         /// </summary>
30         /// <seealso cref="IInterface"/>
31         public class Class : IInterface
32         {
33         }
34
35         /// <summary>
36         /// <para>
37         /// Tests that single dependency test.
38         /// </para>
39         /// <para></para>
40         /// </summary>
41         [Fact]
42         public static void SingleDependencyTest()
43         {
44             using var scope = new Scope();
45             scope.IncludeAssemblyOf<IInterface>();
46             var instance = scope.Use<IInterface>();
47             Assert.IsType<Class>(instance);
48         }
49
50         /// <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

- ./csharp/Platform.Scopes.Tests/ScopeTests.cs, 11
- ./csharp/Platform.Scopes/Scope.cs, 1
- ./csharp/Platform.Scopes/Scope[TInclude].cs, 9
- ./csharp/Platform.Scopes/Use.cs, 10