

PostSharp 6.4

User Manual

Table of Contents

Introduction	7
Quick Examples	9
Why Use PostSharp	13
Which Problems Does PostSharp Solve	13
Benefits of Pattern-Aware Compiler Extensions	14
Benefits of PostSharp vs Alternatives	15
How Does PostSharp Work	19
Key Technologies	19
How to Learn PostSharp	23
Architecture Role: Selecting and Creating Aspects	23
Deployment Role: Installing and Deploying PostSharp	25
Developer Role: Using Aspects	25
What's New	27
What's New in PostSharp 6.4	27
What's New in PostSharp 6.3	29
What's New in PostSharp 6.2	30
What's New in PostSharp 6.1	31
What's New in PostSharp 6.0	33
What's New in PostSharp 5.0	35
What's New in PostSharp 4.3	38
What's New in PostSharp 4.2	39
What's New in PostSharp 4.1	42
What's New in PostSharp 4.0	42
What's New in PostSharp 3.1	44
What's New in PostSharp 3.0	45
What's New in PostSharp 2.1	46
What's New in PostSharp 2.0	47
What's New in PostSharp 1.5	49
Deployment and Configuration	51
Deployment	53
Requirements and Compatibility	53
PostSharp Components	57
Installing PostSharp Tools for Visual Studio	59
Installing PostSharp Tools for Visual Studio Silently	59
Installing PostSharp Into a Project	60
Installing PostSharp without NuGet	61
Using PostSharp on a Build Server	62
Upgrading from a Previous Version of PostSharp	65
Uninstalling PostSharp	67
Deploying PostSharp to End-User Devices	73
Licensing	75
Deploying License Keys	75
License Audit	79
Limitations of PostSharp Essentials	79
Sharing Source Code With Unlicensed Teams	80
Installing and Servicing PostSharp License Server	81
Using PostSharp License Server	85
Configuration	91
Configuring Projects in Visual Studio	91
Configuring Projects Using MSBuild	93
Working with PostSharp Configuration Files	97
Accessing Configuration from Source Code	103
Working with Errors, Warnings, and Messages	104

Resolution of assembly binding redirections	106
Adding Aspects to Code	109
Adding Aspects Declaratively Using Attributes	111
Adding Aspects to a Single Declaration	112
Adding Aspects to Multiple Declarations	112
Adding Aspects to Derived Classes and Methods	117
Overriding and Removing Aspect Instances	123
Reflecting Aspect Instances at Runtime	126
Adding Aspects Using XML	129
Adding Aspects Programmatically using IAspectProvider	131
Logging	133
Adding Detailed Logging to your Solution	135
Configuring Specific Logging Frameworks	139
Logging to the System Console	140
Logging to log4net	140
Logging to NLog	141
Logging to Serilog	142
Logging to ETW	144
Logging to Common.Logging	144
Logging to Loupe	145
Customizing the Appearance of Log Records	147
Adding Custom Log Records Manually	151
Writing Custom Messages	151
Working with Custom Activities	155
Adding Properties to Messages and Activities	158
Enabling and Disabling Logging At Run Time	161
Implementing Logging for a Distributed System	163
Choosing Which Requests to Log	169
Implementing a Custom Formatter	171
Adding Audit to your Solution	173
Implementing an Adapter to a Custom Logging Framework	177
Handling Faults in the Logging Component	183
Licensing of PostSharp Logging	185
Upgrading Logging from an Earlier Version	187
Contracts	189
Adding Contracts to Code	191
Creating Custom Contracts	195
Localizing Contract Error Messages	197
Customizing Contract Exceptions	201
INotifyPropertyChanged	203
Implementing INotifyPropertyChanged	205
Working with Properties that Depend on Other Objects	207
Implementing INotifyPropertyChanging	209
Handling Corner Cases	211
Integrating with UI Frameworks	215
Caliburn.Micro	215
MVVM Light	217
Understanding the NotifyPropertyChanged Aspect	219
Suppressing False Positives	223
Weak Event	225
XAML	227
Command	229
Dependency Property	233

Attached Property	237
Parent/Child, Visitor and Disposable	239
Annotating an Object Model for Parent/Child Relationships (Aggregatable)	241
Rule-Based Annotation	243
Enumerating Child Objects (Visitor)	245
Automatically Disposing Children Objects (Disposable)	247
Working With Child Collections	249
Undo/Redo	255
Making Your Model Recordable	257
Adding Undo/Redo to the User Interface	259
Customizing Undo/Redo Operation Names	261
Assigning Recorders Manually	265
Adding Callbacks on Undo and Redo	267
Understanding the Recordable Aspect	269
Caching	273
Caching Method Return Values	275
Removing Items From the Cache	279
Working with Cache Dependencies	283
Customizing Cache Keys	289
Caching Back-Ends	293
Using In-Memory Cache	293
Using Redis Cache	293
Synchronizing Local In-Memory Caches for Multiple Servers	297
Caching Special Types with Value Adapters	299
Preventing Concurrent Execution of Cached Methods	303
Troubleshooting Caching	305
Multithreading	307
Writing Thread-Safe Code with Threading Models	309
Immutable Threading Model	310
Freezable Threading Model	314
Thread Affine Threading Model	316
Synchronized Threading Model	317
Reader/Writer Synchronized Threading Model	319
Actor Threading Model	325
Thread-Unsafe Threading Model	328
Making a Whole Project or Solution Thread Safe	329
Opting In and Out From Thread Safety	331
Compatibility of Threading Models	332
Enabling and Disabling Runtime Verification	333
Run-Time Performance of Threading Model	336
Dispatching a Method to Background	339
Dispatching a Method to the UI Thread	341
Detecting Deadlocks at Runtime	343
Developing Custom Aspects	349
Developing Simple Aspects	351
Injecting Behaviors Before and After Method Execution	351
Handling Exceptions	359
Intercepting Methods	364
Intercepting Properties and Fields	368
Intercepting Events	374
Introducing Custom Attributes	377
Introducing Managed Resources	381
Semantic Advising of Iterator and Async Methods	382
Understanding Aspect Lifetime and Scope	389

Table of Contents

Initializing Aspects	391
Coping with Custom Object Serializers	391
Validating Aspect Usage	393
Developing Composite Aspects	397
Adding Behaviors to Existing Members	398
Introducing Interfaces, Methods, Properties and Events	402
Accessing Members of the Target Class	406
Adding Aspects Dynamically	408
Coping with Several Aspects on the Same Target	409
Ordering Advices	412
Understanding Aspect Serialization	413
Aspect Configuration	415
Customizing Aspect Appearance in Visual Studio	417
Customizing Aspect Description in Tooltips	417
Estimating Code Savings	418
Pushing Information to PostSharp Tools Programmatically	419
Consuming Dependencies from an Aspect	421
Using a Global Composition Container	421
Using a Global Service Locator	424
Using Dynamic Dependency Resolution	426
Using Contextual Dependency Resolution	429
Importing Dependencies from the Target Object	432
Validating Architecture	435
Restricting Interface Implementation	437
Controlling Component Visibility Beyond Private and Internal	441
Developing Custom Architectural Constraints	451
Testing and Debugging	459
Debugging Run-Time Logic	461
Debugging Build-Time Logic	465
Testing that an Aspect has been Applied	467
Testing Run-Time Logic	469
Testing Build-Time Logic	471
Hacking	473
Executing Code Just After the Assembly is Loaded	475

PART 1

Introduction

CHAPTER 1

Quick Examples

This section shows a few examples to demonstrate what PostSharp is about:

- [Standard patterns on page 9](#)
- [Thread safety patterns on page 10](#)
- [Implementation of custom patterns on page 11](#)
- [Validation of custom patterns on page 11](#)

Standard patterns

PostSharp provides implementations of some of the patterns that are the most commonly found in .NET code bases:

- [INotifyPropertyChanged: see INotifyPropertyChanged on page 203.](#)
- [Parent/child relationships: see Parent/Child, Visitor and Disposable on page 239.](#)
- [Undo/redo: see Undo/Redo on page 255.](#)
- [Code contracts: see Contracts on page 189.](#)
- [Logging: see Logging on page 133.](#)

Example

The following code snippet illustrates an object model where `INotifyPropertyChanged`, undo/redo, code contracts, aggregation and code contracts are all implemented using PostSharp ready-made attributes.

```
[NotifyPropertyChanged]
publicclass CustomerViewModel
{
    [Required]
    public Customer Customer { get; set; }

    publicstring FullName { get { returnthis.Customer.FirstName + " " + this.Customer.LastName; } }
}

[NotifyPropertyChanged]
[Recordable]
publicclass Customer
{
    publicstring FirstName { get; set; }
    publicstring LastName { get; set; }

    [Child]
    public AdvisableCollection<Address> Addresses { get; set; }

    [Url]
    publicstring HomePage { get; set; }

    [Log]
    publicvoid Save(DbConnection connection)
    {
        // ...
    }
}

[NotifyPropertyChanged]
```

```
[Recordable]
publicclass Address
{
    [Parent]
    public Customer Parent { get; private set; }

    publicstring Line1 { get; set; }
}
```

Thread safety patterns

Multithreading is a great demonstration of the limitations of conventional object-oriented programming. Thread synchronization is traditionally addressed at an absurdly low level of abstraction, resulting in excessive complexity and defects.

Yet, several design patterns exist to bring down the complexity of multithreading. New programming languages have been designed around these patterns: for instance Erlang over the Actor pattern and functional programming over the Immutable pattern.

PostSharp gives you the benefits of threading design patterns without leaving C# or VB.

PostSharp supports the following threading models and features:

- [Immutable](#): see [Immutable Threading Model on page 310](#).
- [Freezable](#): see [Freezable Threading Model on page 314](#).
- [Actor](#): see [Actor Threading Model on page 325](#).
- [Reader/Writer Synchronized](#): see [Reader/Writer Synchronized Threading Model on page 319](#).
- [Synchronized](#): see [Synchronized Threading Model on page 317](#).
- [Thread Unsafe](#): see [Thread-Unsafe Threading Model on page 328](#).
- [Thread Affine](#): see [Thread Affine Threading Model on page 316](#).
- [Thread Dispatching](#): see [Dispatching a Method to Background on page 339](#) and [Dispatching a Method to the UI Thread on page 341](#).
- [Deadlock Detection](#): see [Detecting Deadlocks at Runtime on page 343](#).

Example

The following code snippet shows how a data transfer object can be made freezable, recursively but easily:

```
[Freezable]
publicclass Customer
{
    publicstring Name { get; set; }

    [Child]
    public AdvisableCollection<Address> Addresses { get; set; }
}

[Freezable]
publicclass Address
{
    [Parent]
    public Customer Parent { get; private set; }

    publicstring Line1 { get; set; }
}

publicclass Program
{
    publicstaticvoid Main()
    {
        Customer customer = ReadCustomer( "http://customers.org/11234" );
    }
}
```

```

    // Prevent changes.
    ((IFreezable)customer).Freeze();

    // The following line will cause an ObjectReadOnlyException.
    customer.Addresses[0].Line1 = "Here";
}
}

```

Implementation of custom patterns

The attributes that implement the standard and thread safety patterns are called *aspects*. This term comes from the paradigm of *aspect-oriented programming* (AOP). An *aspect* is a class that encapsulates behaviors that are injected into another class, method, field, property or event. The process of injecting an aspect into another piece of code is called *weaving*. PostSharp weaves aspects at build time; it is also named a *build-time aspect weaver*.

PostSharp Aspect Framework is a pragmatic implementation of AOP concepts. All ready-made implementations of patterns are built using PostSharp Aspect Framework. You can use the same technology to automate the implementation of your own patterns.

To learn more about developing your own aspects, see [Developing Custom Aspects on page 349](#).

Example

The following code snippet shows a simple [PrintException] aspect that writes an exception message to the console before rethrowing it:

```

[Serializable]
class PrintExceptionAttribute : OnExceptionAspect
{
    public override void OnException(MethodExecutionArgs args)
    {
        Console.WriteLine( args.Exception.Message );
    }
}

```

In the next snippet, the [PrintException] aspect is applied to a method:

```

class Customer
{
    public string FirstName { get; set; }
    public string LastName { get; set; }

    [PrintException]
    public void Store(string path)
    {
        File.WriteAllText( path, string.Format( "{0} {1}", this.FirstName, this.LastName ) );
    }
}

```

Validation of custom patterns

Not all patterns can be fully implemented by the compiler. Many patterns involve a lot of handwritten code. However, they are still patterns because we want to follow the same conventions and approach when solving the same problem. In this case, we have to validate the code against implementation guidelines of the pattern. This is typically achieved during code reviews, but as any algorithmic work, it can be partially automated using the right tool. This is the job of the *PostSharp Architecture Framework*.

PostSharp Architecture Framework also contains pre-built architectural constraints that help to solve common design problems. For instance, the `InternalImplementAttribute` constraint prevents an interface to be implemented in an external assembly.

See [Validating Architecture on page 435](#) for more details about architecture validation.

Example

Consider a form-processing application. There may be hundreds of forms, and each form can have dozens of business rules. In order to reduce complexity, the team decides that all business rules will respect the same pattern. The team decides that each class representing a business rule must contain a public nested class named `Factory`, and that this class must have an `[Export(IBusinessRuleFactory)]` custom attribute and a default public constructor. The team wants all developers to follow the convention. Therefore, the team decides to create an architectural constraint that will validate the code against the project-specific *Business Rule Factory* pattern.

```
[MulticastAttributeUsage(MulticastTargets.Class, Inheritance = MulticastInheritance.Strict)]
publicclass BusinessRulePatternValidation : ScalarConstraint
{
    publicoverridevoid ValidateCode(object target)
    {
        var targetType = (Type)target;

        if (targetType.GetNestedType( "Factory" ) == null)
        {
            Message.Write( targetType, SeverityType.Error, "2001",
                "The {0} type does not have a nested type named 'Factory'.",
                targetType.DeclaringType, targetType.Name );
        }

        // ...
    }
}

[BusinessRulePatternValidation]
publicabstract BusinessRule
{
    // ...
}
```

CHAPTER 2

Why Use PostSharp

How many times did you find yourself:

- copying and pasting blocks of code to implement a functionality (for instance logging)?
- implementing `INotifyPropertyChanged` manually – and forget a notification?
- trying to understand the business logic behind a cluttered codebase?
- struggling to add or modify functionality in an existing software?
- debugging data races in multithreaded applications?
- wondering why it's so hard to build enterprise-grade software?

PostSharp started as an open-source project in 2004 and due to its popularity, it soon became a commercial product trusted by over 50,000 developers worldwide and over 1,000 leading corporations. More than 10% of all Fortune 500 companies including Microsoft, Intel, Bank of America, Phillips, NetApp, BP, Comcast, Volkswagen, Hitachi, Deutsche Bank, Bosch, Siemens, and Oracle rely on PostSharp to reduce their development and maintenance costs.

With over a decade experience in boilerplate reduction, PostSharp is now the #1 best-selling pattern-aware extension to C# and VB and the only commercially-supported development tool for .NET.

In this chapter

Topic	Description
Which Problems Does PostSharp Solve on page 13	This topic describes which problems PostSharp attends to address.
Benefits of Pattern-Aware Compiler Extensions on page 14	This topic defines the concept of Patter-Aware Compiler and explains its benefits.
Benefits of PostSharp vs Alternatives on page 15	This topic lists the competitive benefits of using PostSharp over alternatives.

2.1. Which Problems Does PostSharp Solve

Conventional programming languages miss a concept of pattern, therefore patterns are hand-coded and result in boilerplate code.

Boilerplate code has the following impact:

- [High development effort on page 14](#)
- [Poor quality software on page 14](#)
- [Difficulty to add/modify functionality after release 1.0 on page 14](#)
- [Slow ramp-up of new team members on page 14](#)

High development effort

- **Large codebases.** Some application features require a large amount of repetitive code (boilerplate) when implemented with existing mainstream compiler technologies.
- **Reinventing the wheel.** Solutions to problems like `INotifyPropertyChanged` are always being reinvented because there are no reusable options within conventional programming languages.

Poor quality software

- **High number of defects.** Every line of code has a possibility of defect, but code that stems from copy-paste programming is more likely than other to be buggy because subtle differences are often overlooked.
- **Multithreading issues.** Object-oriented programming does not deliver much value when it comes to developing multithreaded applications since it addresses issues at a low level of abstraction with locks, events or interlocked accesses that can easily result in deadlocks or random data races.
- **Lack of robustness.** Enterprise-grade features such as exception handling or caching are often deliberately omitted because of the high amount of source code they imply, unintentionally forbidden in some parts of the applications, simply left untested and unreliable.

Difficulty to add/modify functionality after release 1.0

- **Unreadable code that's difficult to maintain.** Business code is often littered with low-level non-functional requirements and is more difficult to understand and maintain, especially when the initial developer left.
- **Strong coupling.** Poor problem decomposition results in duplicate code and strong coupling making it very difficult to change the implementation of features like logging, exception handling or `INotifyPropertyChanged` because it is often scattered among thousands of files.

Slow ramp-up of new team members

- **Too much knowledge required.** When new team members come to work on a specific feature, they often must first learn about caching, threading and other highly technical issues before being able to contribute to the business value: an example of bad division of labor.
- **Long feedback loops.** Even with small development teams, common patterns like diagnostics, logging, threading, `INotifyPropertyChanged` and undo/redo can be handled differently by each developer. Architects need to make sure new team members understand and follow the internal design standards and have to spend more time on manual code reviews--delaying progress while new team members wait to get feedback from code review.

2.2. Benefits of Pattern-Aware Compiler Extensions

Pattern-aware programming extends conventional object-oriented programming with a concept of pattern, which becomes a first-class element of the programming language.

Most mainstream programming languages can be extended with a concept of pattern, avoiding the cost of rewriting applications in a new language.

Because patterns are supported by the compiler extension (100% compatible with your existing compiler), they do not need to be manually implemented as boilerplate code. Features such as `INotifyPropertyChanged`, logging, and transactions are implemented in a cleaner, more concise way, making development and maintenance much easier.

There are 4 reasons to consider using a pattern-aware compiler extension:

- [Stop writing boilerplate code and deliver faster on page 15](#)

- [Build more reliable software on page 15](#)
- [Add/modify functionality more easily after first release on page 15](#)
- [Help new members contribute quicker on page 15](#)

Stop writing boilerplate code and deliver faster

- **Fewer lines of code means fewer hours of work.** Patterns are repetitive, with little or no decision left to the developer. However, repetition is exactly what computers are good at. Let the compiler do the repetitive work and save development time and costs immediately.

Build more reliable software

- **Cleaner code means fewer defects.** With a pattern-aware compiler eliminating the boilerplate, your code becomes easier to read, understand and modify, and contains fewer defects.
- **Reliability becomes much more affordable.** Because they no longer require so much manual coding, reliability features such as caching or exception handling are much easier and cheaper to implement, so you can spend your extra time building a more robust app.

Add/modify functionality more easily after first release

- **Cleaner and shorter code is easier to understand.** After the initial release, too much development time is spent reading and analyzing source code, especially if the initial developer left. With minimized boilerplate code, developers can easily focus on business logic and spend much less time trying to understanding the code.
- **Better architecture is future-proof.** Using a pattern-aware compiler, features like logging, exception handling or transactions are no longer scattered among thousands of files but they are defined in one place, making it much easier and fast to modify when necessary.

Help new members contribute quicker

- **Achieve a better division of labor.** Using a pattern-aware compiler makes the introduction of new or junior team members less onerous since they can focus on simpler, more business logic-oriented tasks rather than having to waste so much time learning complex architectural structures.
- **Implement a tighter feedback loop.** A pattern-aware compiler can validate that handwritten code respects a pattern or a model, and it can detect bugs at build time instead of during code reviews, testing, or in production.

2.3. Benefits of PostSharp vs Alternatives

PostSharp is the #1 pattern-aware extension to C#/VB. It adds a concept of pattern to the languages, resulting in a dramatic reduction of boilerplate code, lower development and maintenance costs and fewer errors.

With PostSharp you can:

- [Get more productive in minutes with ready-made pattern implementations on page 16](#)
- [Automate more complex patterns and remove more boilerplate on page 16](#)
- [Build thread-safe apps--without a PhD on page 17](#)
- [Maintain your existing codebase in C# or Visual Basic on page 17](#)
- [Benefit from much better run-time performance on page 18](#)

Get more productive in minutes with ready-made pattern implementations

- **INotifyPropertyChanged Pattern.** Automates the implementation of INotifyPropertyChanged and automatically raises notifications for you. It also analyzes chains of dependencies between properties, methods and fields in your source code, and understands that property getters can access several fields and call different methods, or even depend on properties of other objects. PostSharp eliminates all the repetition and lets you go from three lines of code per property to one attribute per base class... so you will never forget to raise a property change notification again.
- **Undo/Redo Pattern.** Makes the implementation of the end-users most-wanted features easy and affordable by recording changes at model level. Provides built-in user controls or allows you to create your own. You can deliver the familiar Undo/Redo experience to your users without getting stuck writing large amounts of code.
- **Code Contracts.** Provide validation for valid URLs, email addresses, positive numbers or not-null values and many more, right out of the box. Allows you to use contract attributes without limitations at any location in your codebase and validate methods, fields, properties and parameters. This enables you to protect your code from invalid inputs with custom attributes.
- **Logging Pattern.** Adds comprehensive logging in a few clicks – without impact on your source code – and lets you remove it just as quickly. Provides parameter and return values providing added information for maintenance and support work. Supports most popular backends, including log4net, NLog, Enterprise Library, System Console, System Diagnostics. You can trace everything you need in minutes without cluttering your code.

Automate more complex patterns and remove more boilerplate

- **PostSharp Aspect Framework.** PostSharp is hands down the most robust and exhaustive implementation of aspect-oriented programming for .NET and was evolved into the world's best pattern compiler. It is the most powerful toolset available to implement automation for your own patterns.
- **Largest choice of possible transformations.** Includes decoration of methods, iterators and async state machines, interception of methods, events or properties, introduction of interfaces, methods, events, properties, custom attributes or resources, and more.
- **Composition of several transformations** to easily automate complex patterns.
- **Dynamic aspect/advice providers.** Addresses situations where it is not possible to add aspects declaratively (using custom attributes) to the source code with dynamic aspect/advice providers.
- **Aspect inheritance.** Apply an aspect to a base class, specify that you want it to be inherited and all derived classes will automatically have the aspect applied to them. Relieves you from implementing the aspects manually and ensures that all derived classes using this aspect's logic is correct.
- **Architecture framework.** Validates handwritten source code against your own custom pattern guidelines. It then express the rules in C# using the familiar System.Reflection API, extended with features commonly found in decompilers, such as "find usage", and more.

Build thread-safe apps--without a PhD

Starting new threads and tasks in .NET languages is simple, but ensuring that objects are thread-safe is not with mainstream programming languages. That's why PostSharp extends C# and VB with thread-safety features.

- **7 different threading models.** Threading models are design patterns that guarantee your code executes safely even when used from multiple threads. Threading models raise the level of abstraction at which multithreading is addressed. Unlike working directly with locks and other low-level threading primitives, threading models decrease the number of lines of code, the number of defects and reduce development and maintenance costs – without having to have expertise in multithreading. Includes:
 1. **Immutable Threading Model.** Allows you to make select objects in your codebase immutable so that they can be safely accessed by several threads concurrently, without the need for locking or other synchronization.
 2. **Freezable Threading Model.** This is the milder brother of the Immutable pattern. It is suitable when you need to prevent changes to an instance of an object most of, but not all of the time. Lets you define the point in time where immutability begins.
 3. **Synchronized Threading Model.** Makes sure the objects are accessed by a single thread at a time. Other threads will wait until the object is available so you'll avoid data races.
 4. **Reader-Writer Synchronized Threading Model.** This pattern relies on the fact that most objects are much more often read than modified. Compared to traditional locking, it maximizes read throughput and minimizes the odds of deadlocks.
 5. **Actor Threading Model.** Actors are classes that essentially run within a single thread. Other code communicates with actors using asynchronous calls. Celebrated by Erlang, Scala and F# developers, this pattern is now available to .NET thanks to PostSharp and C# 5.0.
 6. **Thread Affine Threading Model.** Limits object instance access to the thread that created the instance.
 7. **Thread Unsafe Threading Model.** Perfect pattern to make sure that objects will never be accessed concurrently by several threads. Get an exception instead of a random data corruption.
- **Model validation.** Catches most defects during the build or during single-threaded test coverage.
- **Thread dispatching patterns.** Causes the execution of a method to be dispatched to the UI thread or to a background thread. Much easier than using nested anonymous methods.
- **Deadlock detection.** Causes an easy-to-diagnose exception in case of deadlock instead of allowing the application to freeze and create user's frustration.

Maintain your existing codebase in C# or Visual Basic

Despite the hype around functional programming languages, C#/VB and .NET remain an excellent platform for enterprise development. PostSharp respects your technology assets and will work incrementally with your existing code base – there is NO need for a full rewrite or redesign.

- **Design neutrality.** Unlike alternatives, PostSharp takes minimal assumptions on your code. It does not force you to adopt any specific architecture or threading model. You can add aspects to anything, not just interface/virtual methods. Plus, it is fully orthogonal from dependency injection. You don't have to dissect your application into components and interfaces in order to use PostSharp.
- **Plain C# and VB.** PostSharp provides advanced features present in F#, Scala, Nemerle, Python, Ruby or JavaScript, but your code is still 100% C# and VB, and it is still compiled by the proved Microsoft compilers.
- **Cross-platform.** PostSharp supports the .NET Framework, Windows Phone, WinRT, Xamarin and Portable Class Libraries.
- **Standard skill set.** No complex API. Reuse what you already know from C# and System.Reflection.

Benefit from much better run-time performance

Start-up latency, execution speed and memory consumption matter. Whether you're building a mobile app or a backend server, PostSharp delivers exceptional run-time performance.

- **Build-time code generation.** Unlike proxy-based solutions, PostSharp modifies your code at build time. It also allows for much more powerful enhancements that produces dramatically faster applications.
- **No reflection.** PostSharp does not rely on reflection at run-time. The only code that is executed is what you can see with a decompiler.
- **Build-time initialization.** Many patterns make decisions based on the shape of the code which they are applied. With PostSharp, you can analyze the target code at build-time and store the decisions into serializable fields. At runtime, the aspects will be deserialized and you won't need to analyze the code at run-time using reflection.

CHAPTER 3

How Does PostSharp Work

On a conceptual level, you can think of PostSharp as an extension to the C# or VB compiler. Practically, Microsoft's compilers themselves are not extensible, but the build process can be easily extended. That's exactly what PostSharp is doing: it inserts itself into the build process and post-processes the output of the compiler.

This topic contains the following sections:

- [MSBuild Integration on page 19](#)
- [MSIL Rewriting on page 19](#)

MSBuild Integration

PostSharp integrates itself in the build process thanks to *PostSharp.targets*, which is imported into each project using PostSharp by the NuGet installation script *install.ps1*. *PostSharp.targets* adds a few steps to the build process. The principal step is the post-processing of the compiler's output by PostSharp itself.

See [Configuring Projects Using MSBuild on page 93](#) for details.

MSIL Rewriting

PostSharp post-processes the compiler output by reading and disassembling the intermediate assembly, executing the required transformations and validations, and writing the final assembly back to disk.

Although this might sound magic or dangerous, PostSharp's MSIL technology is stable and mature, and has been used by tens of thousands of projects since 2004. Other .NET products relying on MSIL transformation or analysis include Microsoft Code Contracts, Microsoft Code Analysis, and Microsoft Code Coverage.

3.1. Key Technologies

PostSharp combines several technologies to leverage design pattern automation:

- [Metaprogramming on page 19](#)
- [Aspect-Oriented Programming on page 20](#)
- [Static Program Analysis on page 20](#)
- [Dynamic Program Analysis on page 20](#)

Metaprogramming

Metaprogramming is the writing of a program that analyzes and transforms itself or other programs. PostSharp internally represents a .NET program as a mutable .NET object model, so PostSharp can be considered a metaprogramming tool for .NET.

However, general metaprogramming (the ability to perform arbitrary modifications on a program) is a highly complex discipline. Although it may seem easy to perform simple modifications on simple programs, it is actually much more difficult to implement non-trivial transformations that work in all cases. Metaprogramming can result in a decrease in

productivity when used improperly and it is very difficult, for application developers who lack specific training in compilers and metaprogramming, to use it properly.

Since general metaprogramming is too complex and too low level, we need a higher layer of abstraction that makes it easier and safer to express program transformations. Essential qualities of this abstraction layer would include safe composition of several transformations on the same declaration and restrictions on changing the program semantics.

Aspect-oriented programming fulfills these qualities as a disciplined approach to metaprogramming.

Aspect-Oriented Programming

PostSharp Aspect Framework is built on the principle of *Aspect-Oriented Programming* (AOP), a well-established programming paradigm, orthogonal to (and non-competing with) object-oriented programming or functional programming, that allows to modularize the implementation of some features that would otherwise crosscut a large number of classes and methods.

We can confidently say that PostSharp is the most advanced AOP framework for Microsoft .NET.

For details on PostSharp's implementation of AOP, see [Developing Custom Aspects on page 349](#).

Static Program Analysis

Static program analysis is the analysis of a program without executing it.

There are two families of static analysis tools:

- *Structural static analysis* tools analyze the program's declarations and instructions, but do not attempt to understand the run-time behavior of the program. Microsoft Code Analysis belongs to this category.
- *Behavioral static analysis* tools are based on iterative techniques like abstract interpretation, model checking or data-flow analysis. Behavioral static analysis is much more complex and time consuming. Microsoft Code Contracts belong to this category.

PostSharp contains tools for *structural* static analysis only. These tools consist in complete access to the `System.Reflection` model of the assembly being built, navigating through code relationships using the `ReflectionSearch` facility, and an expression tree decompiler.

The most important use case for static analysis in PostSharp is when writing aspects, in order to determine how the target program should be transformed. Any of the static analysis tools can be used to build aspects. This contrasts with other AOP implementation like AspectJ, which defines its own specific language (*pointcut* language) to select target declarations.

Aspects like `NotifyPropertyChangedAttribute` or threading models make advanced use of static analysis.

A secondary role for static analysis is architecture validation. This role is marketed as the *PostSharp Architecture Framework*, which defines a notion of architectural constraint. see [Validating Architecture on page 435](#) for more information.

Dynamic Program Analysis

Dynamic program analysis is the analysis of a program during its execution. Dynamic analysis is often used to detect issues early, before they cause bigger damage. A typical example of dynamic analysis is the one that occurs when an object is cast to a type: when the safety of the type conversion cannot be proved using static analysis, the type conversion must be verified at run time, and an `InvalidCastException` is thrown when an invalid cast is detected.

PostSharp uses dynamic analysis to check the program against threading models. Since many model properties cannot be reliably verified at build time, they must be enforced at run time. For instance, with the Synchronized threading model, accessing a field without owning access to the object would result in a `ThreadAccessException`. For details, see [Writing Thread-Safe Code with Threading Models on page 309](#).

Another example of use of dynamic program analysis in PostSharp is deadlock detection. For details, see [Detecting Deadlocks at Runtime on page 343](#).

In PostSharp, dynamic analysis is achieved by adding instrumentation aspects to the program.

CHAPTER 4

How to Learn PostSharp

A PostSharp implementation project is typically composed of three phases, in which three roles typically interact differently with the product. Each role requires different skills and knowledge. We have created a learning path for each of these roles. Depending on your team's organization, you may be involved in one or more roles.

Topic	Description
Architecture Role: Selecting and Creating Aspects on page 23	<p>In the first phase, the team learns about the concepts and abilities of PostSharp, and identifies how it could fit into the application's design and architecture. The team selects the ready-made aspects that will be used in the application and, when no ready-made aspects can be used, it creates custom aspects or architecture rules.</p> <p>People in this role are typically called architects, technical leads or senior developers. They must acquire an extensive knowledge of PostSharp and their decisions affect the whole team's productivity.</p>
Deployment Role: Installing and Deploying PostSharp on page 25	<p>The next typical step is to deploy PostSharp into your development infrastructure and configure licensing. PostSharp works mostly out of the box for individual developers and small teams, but you may want some planning if you are responsible for a complex solution.</p>
Developer Role: Using Aspects on page 25	<p>Your team is finally ready to use PostSharp on a daily basis. At this point, typically, the team already knows which aspects will be used and when and how they will be used. As a developer who will <i>use</i> existing aspects, you don't need to invest a large amount of effort to learn PostSharp upfront. However, you will need to understand the aspects that have been selected by the team, and how to apply them to a code base.</p>

4.1. Architecture Role: Selecting and Creating Aspects

The first step in the process of adopting PostSharp is typically to understand *what* the product can do for you and *why* you should use (or not use) its features. This activity is typically part of the architecture role.

As you will see, PostSharp offers a set of pre-built aspects implementing some of the most common patterns. As an architect, you will need to understand what these aspects can do for you and how they could fit and simplify your architecture.

However, standard patterns are only the top of the iceberg. To cover your specific needs, PostSharp includes construction kits that allow you to build your own pattern automation, namely the PostSharp Aspect Framework and the PostSharp Architecture Framework. Determining the need for custom aspects or architecture validation rules is typically also a part of the architecture role.

In a typical team, only a few people must be able to create custom aspects or architecture rules. These people must have a deeper understanding of PostSharp than the developers who will only use existing aspects and rules. This is why this skill set is included in the current section.

NOTE

When writing this section, we realized that the current documentation has some serious weaknesses regarding conceptual and architectural materials. This is why we are also referring to other resources hosted on our web site.

Introduction

Understanding the principles behind PostSharp will give you a foundation to build on. All patterns and techniques used by PostSharp relate back to this foundation.

Topic	Articles
About PostSharp	Why Use PostSharp on page 13 How Does PostSharp Work on page 19 Requirements and Compatibility on page 53
More About Design Pattern Automation	Article: Design Pattern Automation¹
More About Aspect-Oriented Programming	Aspect-Oriented Programming in Microsoft .NET² White Paper: Producing High-Quality Software with Aspect-Oriented Programming³

Selecting pre-built pattern implementations

PostSharp offers a number of different pre-built patterns. The following documentation will outline how to use each of the available patterns.

Topic	Articles
General patterns	Contracts on page 189 Parent/Child, Visitor and Disposable on page 239
User interface patterns	Understanding the NotifyPropertyChanged Aspect on page 219 Understanding the Recordable Aspect on page 269
Multithreading	White Paper: Threading Models for Object-Oriented Programming⁴ Detecting Deadlocks at Runtime on page 343 Dispatching a Method to the UI Thread on page 341 Dispatching a Method to Background on page 339
Diagnostics	Logging on page 133

Creating automation for custom patterns

PostSharp's built-in patterns won't cover all scenarios in your codebase that can benefit from AOP. Learn how to build custom patterns using the same foundational components as are used for the built-in patterns.

1. <https://www.postsharp.net/downloads/documentation/Design%20Pattern%20Automation.pdf>
2. <https://www.postsharp.net/aop.net>
3. <https://www.postsharp.net/downloads/documentation/Producing%20High-Quality%20Software%20with%20Aspect-Oriented%20Programming.pdf>
4. <https://www.postsharp.net/downloads/documentation/Threading%20Models%20for%20OOP.pdf>

Topic	Articles
Aspects	Developing Custom Aspects on page 349
Architecture Validation	Validating Architecture on page 435

4.2. Deployment Role: Installing and Deploying PostSharp

This section describes how to deploy PostSharp in different situations. Read it if you are responsible to integrate PostSharp into your build process.

Topic	Articles
Deploying to the Development Environment	Installing PostSharp Tools for Visual Studio on page 59
	Installing PostSharp Into a Project on page 60
	Deploying License Keys on page 75
	Uninstalling PostSharp on page 67
Deploying to the Build Infrastructure	Using PostSharp on a Build Server on page 62
	Restoring Packages at Build Time on page 62
	Using PostSharp with Visual Studio Online on page 64
Deploying to Production or End-User Devices	Deploying PostSharp to End-User Devices on page 73
Deploying to Large or Regulated Development Environments	License Audit on page 79
	Using PostSharp License Server on page 85
	Upgrading Large Repositories from a Previous Version of PostSharp on page 65

4.3. Developer Role: Using Aspects

Using aspects requires much less training than creating new ones. In typical large teams, only a few developers or architects develop new aspects, while the rest of the team uses existing aspects. This section focuses on the skill set that you need to acquire if you have to be able to use PostSharp aspects but don't need to create your own.

In this session, we also assume that PostSharp has been properly deployed into your development and build environments.

NOTE

Of course, you can learn PostSharp as much as you want. The role of this section is to provide a short list of articles to minimize your learning curve and get you productive as quickly as possible, but this should not stop you from learning and experimenting more.

This topic contains the following sections:

- Installing and upgrading PostSharp
- Working with pre-built patterns

- Working with Patterns

Installing and upgrading PostSharp

Every process has a starting point. Learn how to add PostSharp to your project so that you can get started with improving your codebase.

Topic	Articles
Install PostSharp to your machine	Requirements and Compatibility on page 53 Installing PostSharp Tools for Visual Studio on page 59
Add PostSharp to a project and keep it up-to-date	Installing PostSharp Into a Project on page 60 Upgrading from a Previous Version of PostSharp on page 65

Working with pre-built patterns

PostSharp offers a number of different pre-built patterns. You will need to learn those that will be used in your application.

Topic	Articles
Diagnostics	Logging on page 133
Code Contracts	Contracts on page 189
INotifyPropertyChanged	INotifyPropertyChanged on page 203
Aggregatable	Parent/Child, Visitor and Disposable on page 239
Disposable	Automatically Disposing Children Objects (Disposable) on page 247
Undo and Redo	Undo/Redo on page 255
Threading Models	Writing Thread-Safe Code with Threading Models on page 309 Freezable on page 314 , Immutable on page 310 , Actor on page 325 , Reader/Writer Synchronized on page 319 , Synchronized on page 317 , Thread-Unsafe on page 328 , Thread Affine on page 316 Compatibility of Threading Models on page 332 Opting In and Out From Thread Safety on page 331
Deadlock Detection	Detecting Deadlocks at Runtime on page 343
Dispatching Threads	Dispatching a Method to the UI Thread on page 341 Dispatching a Method to Background on page 339
Architecture Validation	Restricting Interface Implementation on page 437 Controlling Component Visibility Beyond Private and Internal on page 441

Working with Patterns

The following resources are for all aspects. You can save a great amount of time in learning to master them.

Topic	Articles
Adding aspects to several declarations	Adding Aspects Declaratively Using Attributes on page 111
Resolving Errors	Working with Errors, Warnings, and Messages on page 104

CHAPTER 5

What's New

PostSharp has been around since the early days of .NET 2.0 in 2004. Since the first version, many features have been added to make PostSharp the most popular and by far the most powerful tool for aspect-oriented programming and design pattern automation in .NET.

This chapter contains the following sections:

- [What's New in PostSharp 6.4 on page 27](#)
- [What's New in PostSharp 6.3 on page 29](#)
- [What's New in PostSharp 6.2 on page 30](#)
- [What's New in PostSharp 6.1 on page 31](#)
- [What's New in PostSharp 6.0 on page 33](#)
- [What's New in PostSharp 5.0 on page 35](#)
- [What's New in PostSharp 4.3 on page 38](#)
- [What's New in PostSharp 4.2 on page 39](#)
- [What's New in PostSharp 4.1 on page 42](#)
- [What's New in PostSharp 4.0 on page 42](#)
- [What's New in PostSharp 3.1 on page 44](#)
- [What's New in PostSharp 3.0 on page 45](#)
- [What's New in PostSharp 2.1 on page 46](#)
- [What's New in PostSharp 2.0 on page 47](#)
- [What's New in PostSharp 1.5 on page 49](#)

5.1. What's New in PostSharp 6.4

This release contains the following improvements:

- [Support for .NET Core 3.0 and .NET Standard 2.1 on page 28](#)
- [Support for C# 8.0 on page 28](#)
- [Support for field/property initializers on page 28](#)
- [Free ordering of OnMethodBoundary aspects on iterators with semantic advising on page 28](#)
- [Export of build-time profiling information to a CSV file on page 28](#)

Note there is a breaking change in `LocationInterceptionAspect`. See [Breaking Changes in PostSharp 6.4 on page 28](#) for details.

Support for .NET Core 3.0 and .NET Standard 2.1

PostSharp now fully supports .NET Core 3.0 and .NET Standard 2.1. We've also updated our package *PostSharp.Patterns.Xaml* to make sure it works with WPF on .NET Core 3.0.

Support for C# 8.0

Several new features of C# 8.0 affected PostSharp: default interface methods, nullable reference types, async streams, and read-only struct members.

We have tested and fixed PostSharp for all these features.

Note that async streams are not yet idiomatically supported in PostSharp Aspect Framework, i.e. PostSharp will not be able to apply semantic advising to methods returning an async stream (see [Semantic Advising of Iterator and Async Methods on page 382](#) for details). PostSharp will treat them as plain methods returning an object. The caching aspect also does not support methods returning an async stream.

Support for field/property initializers

In previous versions of PostSharp, a `LocationInterceptionAspect` could not properly intercept field and property initializers. That is, you could not react to the situation where the field or property was assigned on the same line as the declaration. Initializers were simply ignored.

This has been fixed in PostSharp 6.4, and this is a breaking change.

Initialization of *static* fields and properties is now intercepted by the `LocationInterceptionAspectOnSetValue(LocationInterceptionArgs)` advice just as any other assignment.

However, initialization of *instance* fields cannot be intercepted because `LocationInterceptionAspect` expects the current object to be already initialized (and the `this` reference to be usable), which is not the case until the base constructor has been called. Therefore, we defined a new advice method `OnInstanceLocationInitialized(LocationInitializationArgs)` that is being called as soon as the base constructor has completed.

Free ordering of OnMethodBoundary aspects on iterators with semantic advising

It is now possible to freely order an `OnMethodBoundaryAspect` before or after a `MethodInterceptionAspect` on iterator methods, even with semantic advising enabled.

Previously, on iterator methods, aspects of type `OnMethodBoundaryAspect` had to be ordered after any `MethodInterceptionAspect`.

For details, see [Semantic Advising of Iterator and Async Methods on page 382](#)

Export of build-time profiling information to a CSV file

This option is meant to help us diagnose performance issues with PostSharp when there are several projects in the solution. It is now possible to export build-time profiling information to a CSV file. This performance file can be shared by several projects, and then analyzed by a tool like Pivot Table in Excel. For details, see the [BenchmarkOutputFile property in Well-Known PostSharp Properties on page 101](#).

Note that this feature is meant to be used by our support team only. It currently does not include any information that could be useful to users.

5.1.1. Breaking Changes in PostSharp 6.4

This release contains the following breaking changes:

- [LocationInterceptionAspect and initializers of static fields and properties on page 29](#)

LocationInterceptionAspect and initializers of static fields and properties

The `LocationInterceptionAspect.OnSetValue(LocationInterceptionArgs)` advice will now be invoked for initializers of static fields and properties. These assignments were previously ignored. The breaking change does not affect binaries that have already been compiled. It only affects new compilations.

5.2. What's New in PostSharp 6.3

This release contains the following improvements:

- [Support for building on Linux and MacOS on page 29](#)
- [PostSharp Tools for Visual Studio improvements on page 29](#)
- [Support for Deterministic Build on page 29](#)
- [Free ordering of OnMethodBoundary aspects without OnYield/OnResume advices on async methods on page 29](#)
- [Contracts: ability to customize the type of thrown exceptions on page 30](#)

Note that we've updated our platform requirements. See [Breaking Changes in PostSharp 6.3 on page 30](#) for details.

Support for building on Linux and MacOS

It is now possible to build .NET Core projects on Linux and MacOS. All distributions officially supported by .NET Core have been tested.

PostSharp Tools for Visual Studio improvements

- **Performance improvements.** We now use all asynchronous APIs of Visual Studio SDK, and we've rewritten some old WinForms controls into WPF.
- **Support for shared and multi-target projects.** When editing a source file shared by several projects or targets, and if different aspects were used (e.g. using conditional compilation), the code adornments and tooltips now accurately reflect the aspects applied for the current project and/or target. The Aspect Explorer has been updated as well. Previously, PostSharp Tools for Visual Studio did not differentiate assemblies of the same name.
- **Support for per-monitor awareness of DPI.**
- **Redesign of the Code Action manager** dialog, which is now available as a page under the Options dialog box.

Support for Deterministic Build

PostSharp should now respect the `-deterministic`⁵ compiler option. When this option is enabled, subsequent builds with the exact same input will result in the exact same output.

Free ordering of OnMethodBoundary aspects without OnYield/OnResume advices on async methods

It is now possible to freely order an `OnMethodBoundaryAspect` before or after a `MethodInterceptionAspect`, but only if the `OnMethodBoundaryAspect` does not implement the `OnYield(MethodExecutionArgs)` or `OnResume(MethodExecutionArgs)` advices.

Previously, on async or iterator methods, aspects of type `OnMethodBoundaryAspect` had to be ordered after any `MethodInterceptionAspect`.

Note that ordering is still limited for iterator methods.

5. <https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/compiler-options/deterministic-compiler-option>

Contracts: ability to customize the type of thrown exceptions

In previous versions, it was possible to customize the exception messages, but not the exception types itself. We now use a factory pattern to instantiate exceptions, so you can now completely customize the exceptions thrown by standard contracts. See [Customizing Contract Exceptions on page 201](#) for details.

5.2.1. Breaking Changes in PostSharp 6.3

This release contains the following breaking changes:

- [Deprecated target platforms on page 30](#)
- [Deprecated development platforms on page 30](#)

Deprecated target platforms

- .NET Core 1.1 is no longer supported as a target platform.

Deprecated development platforms

- .NET Framework 4.7.2 or later is now required on developer machines and build servers.

5.3. What's New in PostSharp 6.2

Starting from PostSharp 6.2, we are trying to deliver smaller releases more often.

This release contains the following improvements:

- [Support for Visual Studio 2019 on page 30](#)

IMPORTANT NOTE

Despite being a minor version, PostSharp 6.2 contains some low-impact breaking changes. See [Breaking Changes in PostSharp 6.2 on page 30](#) for details.

Support for Visual Studio 2019

PostSharp 6.2 now supports Visual Studio 2019. We also improved the startup performance of our Visual Studio extension.

5.3.1. Breaking Changes in PostSharp 6.2

PostSharp 6.2 contains the following breaking changes:

Changes in supported platforms and packages

- The *PostSharp.Patterns.Caching.Redis* package now depends on *StackExchange.Redis* v2.0.495 instead of the deprecated *StackExchange.Redis.StrongName* v1.2.3.
- The *PostSharp.Patterns.Caching.Redis* package now requires .NET Framework 4.6.1 instead of 4.5.

5.4. What's New in PostSharp 6.1

In PostSharp 6.1, we primarily focused on two directions: the logging API so that we can cope with distributed systems, and the debugging experience. Additionally, we fixed gapped in .NET Standard and C# 7.3 support.

This topic contains the following sections:

- [Logging: major improvements in the front-end API on page 31](#)
- [Visual Studio debugger: better support for async methods on page 32](#)
- [Support for C# 7.3 on page 32](#)
- [Fixing gaps in platform support on page 32](#)
- [Miscellaneous on page 32](#)

IMPORTANT NOTE

Despite being a minor version, PostSharp 6.1 contains some low-impact breaking changes in the logging API. The chances that you would be negatively affected are minor. See [Breaking Changes in PostSharp 6.1 on page 32](#) for details.

Logging: major improvements in the front-end API

We've worked hard to improve the scenario of logging high-load distributed systems (such as microservices) into a structured database (such as Elastic Search), and enable statistical processing of the log records.

To achieve this goal, we had to bring several improvements to PostSharp Logging:

- **Semantic-First Logging**, suitable for statistical processing of messages and machine learning. See [Writing Custom Messages on page 151](#) for details.
- **Custom Properties** on custom messages and activities, including cross-process properties (aka baggage). See [Adding Properties to Messages and Activities on page 158](#) for details.
- **Hierarchical and Cross-Process Activity ID** that can be easily filtered and sorted on to get a logical view of a request in a distributed system. See [Implementing Logging for a Distributed System on page 163](#) for details.
- **Sampled Logging**: Instead of hoarding gigabytes of useless logs, you can now enable logging for selected requests only. See [Choosing Which Requests to Log on page 169](#) for details.
- **Execution time measurement for custom activities**. See [Working with Custom Activities on page 155](#) for details.
- **Serilog backend improvements**: You now have more control over generated Serilog properties. See [SerilogLoggingBackendOptionsIncludedSpecialProperties](#) and [SerilogLoggingBackendOptionsLog-EventEnricher](#) properties for details.
- **Application Insights backend improvements**: the backend now supports custom properties. See [ApplicationInsightsLoggingBackendOptions](#) class for details.

Additionally, PostSharp 6.1 brings the following improvements to logging:

- We are now using the type name instead of the role name by default to read the logging level from the backend.
- Simplified API to set up verbosity. See [LoggingBackendDefaultVerbosity](#) for details.
- Exception logging is now again implemented in a catch block instead of an exception filter block.
- Exception formatting can now be customized by setting the [TextLoggingBackendOptionsException-Formatter](#) property.

Visual Studio debugger: better support for async methods

We are releasing a complete refactoring of our add-in to the Visual Studio debugger. It solves a dozen of issues and finally fixes the debugging behavior of intercepted async methods.

Support for C# 7.3

PostSharp was affected by the following features of C# 7.3:

- `in` parameters,
- `unmanaged`, `Enum` and `Delegate` constraints.

Fixing gaps in platform support

- `PostSharp.Patterns.Caching` now supports .NET Standard 2.0 and .NET Framework 4.5.
- `PostSharp.Patterns.Diagnostics.Backends.CommonLogging` now supports .NET Standard 2.0.

Miscellaneous

- Resolved ambiguous method matching in `LocationValidationAdvice` by adding the `Priority` property.

5.4.1. Breaking Changes in PostSharp 6.1

PostSharp 6.1 contains the following breaking changes:

Changes causing silent changes of behavior

- Earlier versions of PostSharp Logging used the `role` as the name of the logger (in most frameworks) or context (in Serilog). PostSharp Logging 6.1 uses the `source` type instead, which makes more sense, but is a breaking change.

Changes causing build errors

- We changed some method signatures in the API that allows you to build customized logging back-ends. The chances that you may be affected are minor.

Changes causing build warnings

- We replaced the `Logger` class by `LogSource`. The new API requires C# 7.3. The old `Logger` still works. You will get a warning only when you call the `GetLogger(String)` method. For existing developments, we recommend to disable the warning using a `#pragma` directive. See [Upgrading Logging from an Earlier Version on page 187](#) for details.

Deprecated platforms and features

- Visual Studio 2013 is no longer supported.
- .NET Core SDK 2.0 is no longer supported as a build platform. See [Requirements and Compatibility on page 53](#) for details.

5.5. What's New in PostSharp 6.0

The primary focus of PostSharp 6.0 was to execute the PostSharp compiler itself on .NET Core when a .NET Core project is being built. This required the biggest refactoring of our compiler's internals since PostSharp 2.0, which broke backward-compatibility of build-time components, and warranted a major version change.

Additionally, PostSharp 6.0 brings support for C# 7.2 and improves the robustness of the logging component.

IMPORTANT NOTE

As a major version, PostSharp 6.0 contains some breaking changes. See [Breaking Changes in PostSharp 6.0 on page 34](#) for details.

This topic contains the following sections:

- [Support for .NET Core 2.0-2.1 and .NET Standard 2.0 on page 33](#)
- [PostSharp compiler runs natively in .NET Core on page 33](#)
- [Support for Portable PDB on page 33](#)
- [Support for C# 7.2 on page 33](#)
- [Logging: robustness to logging faults on page 34](#)
- [Logging: adapters for Log4Net and NLog now support .NET Standard on page 34](#)
- [Logging: no need to initialize before the first logged method is hit on page 34](#)
- [Caching: preventing concurrent execution \(locking\) on page 34](#)
- [Tools for Visual Studio: support for the new CPS-based project systems. on page 34](#)
- [GDPR Compliance on page 34](#)
- [End of support for Visual Studio 2012 on page 34](#)
- [End of support for .NET Core 1.1 as a build platform on page 34](#)
- [Removal of the UI to use PostSharp without NuGet on page 34](#)

Support for .NET Core 2.0-2.1 and .NET Standard 2.0

.NET Core 2.0-2.1 and .NET Standard 2.0 are now completely supported. PostSharp 5.0 used to have partial and buggy support for these frameworks, and significant work was required to get it right (see below).

PostSharp compiler runs natively in .NET Core

Up to version 5.0, the PostSharp compiler would always run the .NET Framework, even if it was building a project targeting .NET Core. That caused some serious challenges and problems because PostSharp is executing user code at build time, therefore it executed in .NET Framework code written for .NET Core.

Starting from PostSharp 6.0, PostSharp will run under .NET Core if you're building a project for .NET Core. Most of the PostSharp build-time code has been ported to .NET Standard 2.0.

This was a major refactoring and we've repaid a great deal of our technical debt. Support for new releases of .NET Core should become much quicker. This evolution also opens the path to supporting PostSharp on Linux, but there is still work to reach this objective.

Support for Portable PDB

Portable PDB (both standalone and embedded) are now fully supported.

Support for C# 7.2

PostSharp 6.0 properly handles and understands `in` parameters. It also understands `ref struct`, but prevents them from being used as parameters in methods that are enhanced by an aspect (except logging).

Logging: robustness to logging faults

Errors in the logging component or in your logging code will no longer cause your application to crash. For details, see [Handling Faults in the Logging Component on page 183](#).

Logging: adapters for Log4Net and NLog now support .NET Standard

PostSharp.Patterns.Diagnostics.Log4Net and *PostSharp.Patterns.Diagnostics.NLog* now support .NET Standard.

Logging: no need to initialize before the first logged method is hit

It is now possible to change the logging back-ends dynamically, even after a log record has been emitted. As a result, it is no longer necessary to avoid any logging until the logging service was initialized.

Caching: preventing concurrent execution (locking)

You can now configure a lock manager to prevent concurrent execution of the same method with the same arguments on different threads, processes, or machines. See [Preventing Concurrent Execution of Cached Methods on page 303](#) for details.

Tools for Visual Studio: support for the new CPS-based project systems.

The new project systems, which work with simpler project files and include built-in support for NuGet package references, are now properly supported.

GDPR Compliance

PostSharp is now compliant with GDPR. A few changes were needed, including requiring explicit consent before performing license audit (instead of just informing) or sending the newsletter (instead of soft opt in), and making all data transfers secure.

End of support for Visual Studio 2012

PostSharp 6.0 no longer includes support for Visual Studio 2012, as it is no longer in Microsoft mainstream support.

End of support for .NET Core 1.1 as a build platform

You can still build .NET Core 1.1 and .NET Standard 1.* projects, but using the .NET Core 2.0 SDK.

Removal of the UI to use PostSharp without NuGet

It is still possible to use PostSharp without NuGet, but we removed the UI from PostSharp Tools for Visual Studio.

5.5.1. Breaking Changes in PostSharp 6.0

PostSharp 6.0 contains the following breaking changes:

Changes causing silent changes of behavior

- The internal details of the assembly loading algorithms have been changed; therefore, in some rare situations, PostSharp may fail to find or load assemblies in cases that used to be successful with PostSharp 5.0. Please report these cases to the PostSharp support team for individual resolution.
- All MSBuild properties that used to be prefixed PostSharp30 are now prefixed just PostSharp. Check your projects for occurrences of the PostSharp30 substring.
- The `RequiredAttribute` code contract now throws `ArgumentOutOfRangeException` instead of `ArgumentNullException` when a target element is assigned an empty or white-space string.

Changes causing build errors

- Parts of the API that were marked `[Obsolete]` in PostSharp 5.0 have been removed.
- The way exception handlers and filters are represented in the `PostSharp.Reflection.MethodBody` has changed. They used to be tree nodes, but are now just properties of `IBlockExpression`. The `MethodBodyVisitor` class has been modified accordingly.
- The `PostSharp.Patterns.Diagnostics.RecordBuilders.ParameterDirection` has been renamed `ParameterKind` and moved to `PostSharp.dll`.
- The notorious `QueryInterface(object)` extension method has been replaced by the non-extension `QueryInterfaceT(Object, Boolean)`, in an effort to limit the pollution of Intellisense suggestions and documentation.

Deprecated platforms and features

- Visual Studio 2012 is no longer supported.
- .NET Core SDK 1.1 is no longer supported as a build platform. You can still build .NET Core 1.1 libraries under the .NET Core SDK 2.0 or later. See [Requirements and Compatibility on page 53](#) for details.
- .NET Framework 4.7.1 or later is now required on build platforms. Older versions are still supported as runtime platforms. See [Requirements and Compatibility on page 53](#) for details.
- Windows 7 SP1, Windows 8.1, Windows Server 2008 R2 and early versions of Windows 10 are no longer supported as build platforms. See [Requirements and Compatibility on page 53](#) for details.
- Invoking PostSharp from the command line is no longer supported.
- IncrediBuild is no longer supported.

Licensing changes

- It is now mandatory to add your license key to the build server or the source repository if you are using the `PostSharp.Patterns.Diagnostics` namespace. See [Licensing of PostSharp Logging on page 185](#) for details.

5.6. What's New in PostSharp 5.0

As a new major version, PostSharp 5.0 was the opportunity to introduce a few breaking changes. First, we dropped support for Microsoft's failed platforms Windows Phone and WinRT in favor of .NET Core, .NET Standard and Visual Studio 2017. Then, we completely revisited PostSharp Logging. Additionally, we added a caching aspect and three MVVM aspects.

IMPORTANT NOTE

As a major version, PostSharp 5.0 contains some breaking changes. See [Breaking Changes in PostSharp 5.0 on page 37](#) for details.

PostSharp 5.0 includes the following improvements:

- [Logging: complete revamping on page 36](#)
- [Caching: a brand new feature on page 36](#)
- [Filled gaps in support for async methods on page 36](#)
- [XAML: Command, Dependency Property and Attached Property on page 36](#)

- [Code Contracts: support for out parameters and return values on page 36](#)
- [Architecture Framework: new constraints on page 36](#)
- [Support for Visual Studio 2017 on page 37](#)
- [Support for .NET Core and .NET Standard on page 37](#)
- [Support for NuGet 3 on page 37](#)
- [End of support for Windows Phone, WinRT and Silverlight on page 37](#)
- [Suspended support for Xamarin on page 37](#)

Logging: complete revamping

That's a complete rewrite! The new PostSharp Logging is fully customizable and faster than ever. See [Logging on page 133](#) for details.

Caching: a brand new feature

We've added a brand new ready-made caching framework, which includes not only a caching aspect but also a cache invalidation aspect. PostSharp Caching 5.0 comes with support for MemoryCache and Redis. See [Caching on page 273](#) for details.

Filled gaps in support for async methods

We've put a lot of efforts to put async methods on par with normal methods in PostSharp:

- The `MethodInterceptionAspect` now has an `OnInvokeAsync(MethodInterceptionArgs)` advice, which you can implement to intercept async methods.
- You can now set the return value and change the `ReturnValue` and the `FlowBehavior` of an async method in an `OnMethodBoundaryAspect` aspect.
- The `MethodInterceptionAspect` and `OnMethodBoundaryAspect` aspects now advise async methods and iterators semantically by default (i.e. the state machine itself is advised), while previous versions advised the kick-off methods by default. The new property `SemanticallyAdvisedMethodKinds` controls whether the advice is applied semantically or not. The `UnsupportedTargetAction` property determines what should be done when semantic advising of the current method is not supported.

XAML: Command, Dependency Property and Attached Property

If you're writing XAML applications, you probably wrote a lot of boilerplate code for commands and dependency properties. We've created new aspects to automate that. See [XAML on page 227](#) for details.

Code Contracts: support for out parameters and return values

It is now possible to add code contracts to return values and `out` or `ref` parameters. The values are validated when the method succeeds.

Architecture Framework: new constraints

PostSharp 5.0 adds three constraints:

- The `NamingConventionAttribute` constraint is a simple way to force derived class to respect some naming convention.
- The `ParameterValueConstraint` abstract constraint lets you validate, at build-time, the value passed to method parameters.
- The `ReferenceConstraint` abstract constraint allows you to validate which code uses (i.e. references) your type or method.

Support for Visual Studio 2017

PostSharp 5.0 supports Visual Studio 2017 including the new project format (package references) and the new features of C# 7.0.

Support for .NET Core and .NET Standard

PostSharp 5.0 supports .NET Core 1.0 and .NET Standard 1.3 in Visual Studio 2017 and .NET Core CLI. See [Requirements and Compatibility on page 53](#) for details.

Support for NuGet 3

PostSharp 5.0 supports NuGet 3 and it no longer needs *install.ps1* to introduce itself into the build chain.

End of support for Windows Phone, WinRT and Silverlight

Let's face it, these platforms were a failure. We no longer want to pay a price for that and we're dropping support in PostSharp 5.0.

Suspended support for Xamarin

Usage data show that only a few customers are using PostSharp with Xamarin, so we demoted the priority of this platform. It is not supported in PostSharp 5.0. Affected customers should contact our support team. We're considering to support Xamarin through .NET Standard if there is significant demand for it.

5.6.1. Breaking Changes in PostSharp 5.0

PostSharp 5.0 contains the following breaking changes:

Changes causing silent changes of behavior

- The `OnMethodBoundaryAspect` aspect will now advise the state machine of async methods and iterators by default instead of the kick-off method. The `ApplyToStateMachine` property is deprecated in favor of `SemanticallyAdvisedMethodKinds` and its default value is now true. The behavior of existing code that ignored the PS0215 warning will silently change.

Changes causing build errors

- The `PostSharp.Reflection.Syntax` namespace has been renamed `PostSharp.Reflection.MethodBody` and classes inside this namespace have been renamed to reflect the fact this namespace is definitively not a syntax tree. (We were tired to apologize for this misnaming.) You must modify your code (hopefully just a few files) otherwise it will no longer build.
- The `PostSharp.MessageLocation` class has been moved to the `PostSharp.Extensibility` namespace. You must modify your code (hopefully just a few files) otherwise it will no longer build.
- The `PostSharp.Patterns.Diagnostics` has been completely revamped. See [Upgrading Logging from an Earlier Version on page 187](#) for details. You must modify your code (hopefully just a few files) otherwise it will no longer build.
- The `GenericArgs` family of classes, a legacy from PostSharp 1.5 that no longer worked, was deleted.
- It is no longer allowed to add an aspect or an advice to an anonymous method or to a method of a compiler-generated type, including the `MoveNext` method of a state machine type. To avoid the error, implementations of `IAAspectProvider` and pointcuts should use `method.GetSemanticInfo().IsSelectable` to determine whether a method is a valid aspect or advice target.

Changes causing build warnings

- The `PostSharp.IgnoreWarningAttribute` class has been renamed `SuppressWarningAttribute` and moved to the `PostSharp.Extensibility` namespace. You will get an obsolescence warning until you modify your code.

Deprecated platforms

- Silverlight, Windows Phone and WinRT are no longer supported. These projects will no longer build.
- Xamarin is temporarily unsupported. These projects will no longer build. Please contact our support team if you are affected by this change.

Licensing changes

- PostSharp Professional has been renamed PostSharp Framework and no longer includes PostSharp Logging. Please contact our sales team if you have a PostSharp Professional license and rely on PostSharp Logging.
- PostSharp Express has been renamed PostSharp Essentials is no longer compatible with PostSharp Express 4.2 and earlier. The licensing of PostSharp Express was modified in PostSharp 4.3, but PostSharp 4.3 included a backward-compatibility licensing mode. It has now been disabled.
- Use of the license server is now restricted to the new PostSharp Enterprise license. Please contact our sales team if you are affected by this change.

TIP

Please contact our sales team if you are a commercial user and are affected by the licensing changes. We will find a solution that is acceptable for both.

5.7. What's New in PostSharp 4.3

The objective of PostSharp 4.3 is to address the most important concerns of current PostSharp customers. We focused on improving the existing features without adding brand new ones.

PostSharp 4.3 includes the following improvements:

- [Improved build-time performance on page 38](#)
- [Improved debugging experience on page 39](#)
- [More flexible deployment on page 39](#)
- [Improvements in the NotifyPropertyChanged aspect on page 39](#)
- [Simplified licensing of PostSharp Express on page 39](#)
- [Source code sharing with non-licensed teams on page 39](#)
- [Automatic computing of build-time assembly binding redirections on page 39](#)

Improved build-time performance

We understand that nobody likes to wait for the build to complete, so we've been working hard to optimize PostSharp's build performance. PostSharp 4.3 is up to 1.5 times faster than PostSharp 4.2 (the improvement is especially visible in large projects) and you don't need to change anything in your code.

But there's more. PostSharp 4.3 introduces a new feature called *solution-wide build optimizations*, which can double the build speed in large solutions. Since this feature can break custom build-time logic, it is disabled by default. For details, see [Reducing Build Time on page 0](#).

Improved debugging experience

Debugging an application enhanced with aspects is now even easier thanks to the following improvements:

- Full support for Just My Code.
- During Step Into, aspect code is now stepped over by default.
- The call stack no longer contains PostSharp implementation details by default.

To learn more about the new debugging behaviors and how to disable them, see [Debugging Run-Time Logic on page 461](#).

More flexible deployment

PostSharp 4.3 brings more freedom when it comes to deployment and installation:

- **Alternative to NuGet.** Between versions 3.0 and 4.2, the PostSharp compiler and libraries were only distributed as NuGet packages. Starting from version 4.3, we are re-introducing the old good zip file, and integrate it better with PostSharp Tools for Visual Studio. See [Installing PostSharp without NuGet on page 61](#) for details.
- **Command Line Tool.** Using PostSharp as a command-line tool is now a supported and documented scenario. For details, see [\[command-line\]](#)
- **PostSharp Tools for Visual Studio no longer required.** You will now be able to build a project that uses PostSharp without having PostSharp Tools installed in Visual Studio. The tooling is still highly recommended but no longer strictly required.

Improvements in the NotifyPropertyChanged aspect

PostSharp 4.3 brings two improvements to the `NotifyPropertyChangedAttribute` aspect:

- Support for Caliburn.Micro and MVVM Light. See [Integrating with UI Frameworks on page 215](#).
- Option to avoid false positives. See [Suppressing False Positives on page 223](#) for details.

Simplified licensing of PostSharp Express

The limitations of PostSharp Express, the free edition of PostSharp, are now clearer and easier to understand and remember. For details, see [Limitations of PostSharp Essentials on page 79](#).

Source code sharing with non-licensed teams

You no longer need a PostSharp license to build code that someone else wrote and uses PostSharp. A license is only required for code that you build or edited yourself. See [Sharing Source Code With Unlicensed Teams on page 80](#) for details.

Automatic computing of build-time assembly binding redirections

It is no longer necessary to manually create an assembly binding redirection file for PostSharp. For details, see [Resolution of assembly binding redirections on page 106](#).

5.8. What's New in PostSharp 4.2

With PostSharp 4.2, we had two major objectives. The first was to dogfood our Threading Models into PostSharp Tools for Visual Studio, which pushed us to add many improvements both to the threading aspects and to the underlying aspect framework. The second focus was to expose code saving metrics, so you know how many lines of code you likely saved

thanks to PostSharp. Additionally, we kept up with Microsoft and implemented support for the Elvis operator in `NotifyPropertyChanged`, and added experimental support for ASP.NET v5.

PostSharp 4.2 includes the following improvements:

- [Improvements to Aggregatable Pattern on page 40](#)
- [Improvements to NotifyPropertyChanged on page 40](#)
- [Improvements to Threading Models on page 40](#)
- [Improvements to the Aspect Framework on page 41](#)
- [Improved Support for Visual Basic on page 41](#)
- [Code Saving Metrics on page 41](#)
- [Module Initializers on page 41](#)
- [Support for IncrediBuild \(Experimental\) on page 41](#)
- [Support for ASP.NET v5 \(Experimental\) on page 42](#)

Improvements to Aggregatable Pattern

We added the following improvements to the Aggregatable pattern and to other patterns that depend on it:

- New advisable class `AdvisableHashSetT` in replacement of `HashSetT`.
- New methods to the `AdvisableCollectionT` class: `AddRange(IEnumerableT)`, `InsertRange(Int32, IEnumerableT)`, `RemoveRange(Int32, Int32)`.
- Support for immutable collections like `ImmutableArray` or `ImmutableDictionary`.
- Support for type adapters to allow third-party classes (at least read-only ones) to work with the Aggregatable pattern. See `TypeAdapter` for details.
- Ability to programmatically and automatically mark a field as child or reference without having to use a custom attribute in source code. See `FieldRule` for details.
- Performance improvement: memory no longer needs to be allocated at run time after objects are constructed, resulting in lower load on garbage collection.
- New extensions methods for advisable collections. See `Extensions`.

Improvements to NotifyPropertyChanged

Improvements include to `NotifyPropertyChangedAttribute` include:

- Support for the Elvis operator (`?.`) and properties of local variables.
- Notification of the `PropertyChanging` event. See [Implementing INotifyPropertyChanging on page 209](#) for details.
- Better error messages.
- Inclusion of property dependencies in Visual Studio tooltips.
- Performance improvement: memory no longer needs to be allocated at run time after objects are constructed, resulting in lower load on garbage collection.

Improvements to Threading Models

Improvements to threading models include:

- Performance improvement: memory no longer needs to be allocated at run time after objects are constructed, resulting in lower load on garbage collection.
- Several performance improvements regarding the instantiation of large object graphs. See [Run-Time Performance of Threading Model on page 336](#) for details.

- New `ThreadSafetyPolicy` attribute to emit warning when a class is not assigned to any threading model or when a static field is not of a thread-safe type. See [Making a Whole Project or Solution Thread Safe on page 329](#) for details.
- Better support for async methods: for lock-based models, locks are being awaited asynchronously instead of synchronously. These new high-performance advices are not yet fully implemented and tested for general use, therefore they are unsupported (except when they are used with our ready-made patterns) and undocumented.
- Complete dogfooding in our PostSharp Tools for Visual Studio, resulting in dozen of bug fixes and usability improvements.
- In the actor model, methods with non-void return type are now allowed and their execution will be done in the actor context, but the calling thread will wait synchronously for the execution to complete. Void but non-async methods must now be annotated with `[Dispatched]` (and until the next major version) to specify if execution must be synchronous or asynchronous.

Improvements to the Aspect Framework

Improvements to the aspect framework include:

- New advice `OnAspectsInitializedAdvice` invoked after all aspects on the current objects have been initialized.
- Ability to customize the description of aspects and advices in Visual Studio tooltips. See [Customizing Aspect Description in Tooltips on page 417](#) for details.
- In the `OnMethodBoundaryAspect` and related aspects, ability to yield (await) a state machine upon on entry and resume.

CAUTION NOTE

To support performance improvements in ready-made patterns, we included a new family of advices that accept context on the stack instead of the heap.

Improved Support for Visual Basic

Visual Basic is now supported and tested at the same level as C# both in PostSharp Compiler and PostSharp Tools for Visual Studio.

Code Saving Metrics

PostSharp will now estimate the number of handwritten lines of code and the number of lines of code that you likely saved using PostSharp. For details, see [Estimating Code Savings on page 418](#).

Module Initializers

You can now define methods that get executed immediately after the assembly is loaded, before any other code is executed. See [Executing Code Just After the Assembly is Loaded on page 475](#) and `ModuleInitializerAttribute` for details.

Support for IncrediBuild (Experimental)

If you use IncrediBuild, PostSharp can now be executed on a remote computer. Please contact PostSharp support for details.

Support for ASP.NET v5 (Experimental)

You can now use PostSharp in ASP.NET v5 code. The support is currently limited to the .NET Framework (CoreCLR is not supported). Support is not built-in in the normal PostSharp distribution. You need to download the *PostSharp.Dnx* project from GitHub. Please see <https://github.com/postsharp/PostSharp.Dnx> for details.

5.9. What's New in PostSharp 4.1

The focus of PostSharp 4.1 was to broaden the set of supported platforms for both PostSharp, with the addition of Xamarin and Visual Studio 2015, and improvements in the support of Windows Phone and Windows Store.

PostSharp 4.1 includes the following improvements:

- [Support for Xamarin on page 42](#)
- [Threading Pattern Library: support for Windows Phone and Windows Store on page 42](#)
- [Support for Visual Studio 2015 on page 42](#)
- [PostSharp Assistant on page 42](#)

Support for Xamarin

Xamarin has become an inseparable part of the .NET ecosystem and was the number-one feature request of the PostSharp community. PostSharp 4.1 makes it possible to build applications for iOS and Android using Xamarin.

Note that Xamarin applications must be built using Visual Studio. Xamarin Studio is not supported.

Threading Pattern Library: support for Windows Phone and Windows Store

Threading Pattern Library newly supports Windows Phone, Windows Store and Xamarin. This allows you to create thread-safe applications for both Windows and Windows Phone (both Silverlight and WinRT) in the same way as for desktop applications.

Support for Visual Studio 2015

PostSharp Tools for Visual Studio have been almost completely rewritten to take advantage of the new compiler family "Roslyn" at the heart of Visual Studio 2015. New features include integration with the light bulb (instead of the smart tag), live code diagnostics and a few refactorings.

PostSharp Assistant

PostSharp Assistant guides you when you are implementing various patterns from Pattern Libraries so that you don't miss any detail. For instance, it would point at relevant documentation articles or at pieces of code that need to be fixed.

PostSharp Assistant is supported in Visual Studio 2015.

5.10. What's New in PostSharp 4.0

The principal focus of PostSharp 4 was to redesign the Threading Pattern Library from the ground up and make it a real solution to write thread-safe code with C# and VB. Additionally, we've introduced the undo/redo feature into the Model Pattern Library. To achieve these objectives properly, we had to implement a good old concept from UML and object-oriented modeling: aggregation and composition. We introduced significant improvements in the PostSharp Aspect Framework to support these new features.

PostSharp 4.0 includes the following improvements:

- [Aggregatable pattern on page 43](#)
- [Disposable pattern on page 43](#)
- [Immutable threading model on page 43](#)
- [Freezable threading model on page 43](#)
- [Synchronized threading model on page 43](#)
- [Redesign of reader-writer-synchronized, actor, and thread-unsafe threading models on page 43](#)
- [Recordable pattern \(undo/redo\) on page 43](#)
- [Dynamic location imports on page 44](#)
- [Aspect repository on page 44](#)
- [OnInstanceConstructed advice on page 44](#)
- [InitializeAspectInstance advice on page 44](#)
- [NotifyPropertyChanged optimization on page 44](#)

Aggregatable pattern

As it turns out, multiple patterns rely on the notion of parent-child relationships. These concepts are a part of the UML specification, where it is known as aggregation, but even modern programming languages don't implement the notions. We fixed that in PostSharp 4.0 with our `AggregatableAttribute` aspect. For details, see [Parent/Child, Visitor and Disposable on page 239](#).

Disposable pattern

Once we have a notion of parent-child relationship, it is easy to build an aspect that recursively disposes a whole object tree. This is our `DisposableAttribute` aspect. For details, see [Automatically Disposing Children Objects \(Disposable\) on page 247](#).

Immutable threading model

The Immutable patterns made functional languages popular for its great usefulness in multithreaded programs. Unfortunately, the concept has traditionally been difficult to object-oriented programming. PostSharp 4.0 provides a pragmatic implementation with the `ImmutableAttribute` aspect. For details, see [Immutable Threading Model on page 310](#).

Freezable threading model

Even a well-implemented Immutable pattern can be too strict for some object-oriented scenarios. In this case, the Freezable patterns may be more suitable. Based on the Aggregatable pattern, the `FreezableAttribute` aspect makes it possible to build freezable object trees. For details, see [Freezable Threading Model on page 314](#).

Synchronized threading model

A threading model library could not be complete without it, so we added the `SynchronizedAttribute` aspect. For details, see [Synchronized Threading Model on page 317](#).

Redesign of reader-writer-synchronized, actor, and thread-unsafe threading models

We took the right way in PostSharp 3.0 with threading models, but the vision was not yet fully consistent and the implementation was only partial. With PostSharp 3.2, we felt we had a better understanding of what we wanted to achieve, and completely revisited our threading models. Based on the Aggregatable pattern, and based on a consistent object model, the Threading Pattern Library is now much more powerful and consistent.

Recordable pattern (undo/redo)

The `RecordableAttribute` aspect, together with the `Recorder` class, make it possible to implement an undo/redo feature at the domain level.

Dynamic location imports

To allow to import several fields and properties into a single aspect field (which was not possible using `ImportMemberAttribute`, we added the `IAdviceProvider` interface and the `ImportLocationAdviceInstance` class.

Aspect repository

The new `IAAspectRepositoryService` service exposes the list of all aspects added to the code model, both using custom attributes or `IAAspectProvider`, and offer a way to execute validation logic after all aspects have been discovered.

OnInstanceConstructed advice

The `OnInstanceConstructedAdvice` custom attribute allows you to define an advice that is executed after the instance constructor exits.

InitializeAspectInstance advice

The `InitializeAspectInstanceAdvice` custom attribute allows you to define an advice that is similar to `RuntimeInitializeInstance` but passes information about the reason why the aspect is initialized (constructor, clone, deserialization).

NotifyPropertyChanged optimization

Our `NotifyPropertyChangedAttribute` is four times faster at run time on average.

5.11. What's New in PostSharp 3.1

PostSharp 3.1 builds on the vision of PostSharp 3.0, but makes it more convenient to use. It also catches up with the C# compiler features, and add more flexible licensing options.

PostSharp 3.1 includes the following improvements:

- [Better support for iterator and async methods on page 44](#)
- [Improved configuration system on page 44](#)
- [Build-time performance improvement on page 45](#)
- [Resolution of file and line of error messages on page 45](#)
- [Indentation in logging on page 45](#)
- [Separate licensing of Pattern Libraries on page 45](#)

Better support for iterator and async methods

When you applied an `OnMethodBoundaryAspect` to a method that was compiled into a state machine, whether an iterator or an async method, the code generated by PostSharp would not be very useful: the aspect would just be applied to the method that implements the state machine. An `OnException` advice had no chance to get ever fired.

Starting from PostSharp 3.1, `OnMethodBoundaryAspect` understands that is being applied to a state machine, and works as you would expect.

Improved configuration system

PostSharp 3.1 makes it easier to share configuration across several projects. For instance, you can now add aspects to all projects of a solution in just a few clicks. This is not just a UI tweak. This scenario has been made possible by significant improvements in the PostSharp configuration system:

- Support for solution-level configuration files (*SolutionName.pssln*), and well-known configuration files (*postsharp.config*) additionally to project-level files (*ProjectName.psproj*). See [Working with PostSharp Configuration Files on page 97](#) for details.

- Support for conditional configuration elements
- Support for XPath in expressions (instead of only property references as previously). See [Using Expressions in Configuration Files on page 103](#) for details.

Build-time performance improvement

PostSharp can now optionally install itself in GAC and generate native images. This decreases build time of a fraction of a second for each project: a substantial gain if you have a lot of projects.

Resolution of file and line of error messages

When previous versions of PostSharp had to report an error or a warning, it would include the name of the type and/or method causing the message, but was unable to determine the file and line number. You can now double-click on an error message in Visual Studio and you'll get to the relevant location for the error message.

Indentation in logging

For better log readability, PostSharp Logging Pattern Library now automatically indents log entries when entering and exiting methods.

Separate licensing of Pattern Libraries

PostSharp Pattern Libraries can now be purchased separately, so you don't have to buy the full PostSharp Ultimate if you just want to use `INotifyPropertyChanged`. The licensing system has been modified to support this scenario.

5.12. What's New in PostSharp 3.0

The focus in PostSharp 3.0 was to deliver more value to customers with less initial learning. Instead of having to learn the product before being able to build aspects, customers can now choose from a set of ready-made implementations of some of the most popular design pattern, and apply them to their application from the Visual Studio code editor, using smart tags and wizards. We also improved support for Windows Phone, Silverlight, Windows Store and Portable Class Library.

PostSharp 3.0 includes the following improvements:

- [Model Pattern Library on page 45](#)
- [Diagnostics Pattern Library on page 45](#)
- [Threading Pattern Library on page 46](#)
- [Smart tags and wizards in Visual Studio on page 46](#)
- [Better platform support through Portable Class Libraries on page 46](#)
- [Unified deployment through NuGet and Visual Studio Gallery on page 46](#)
- [Transparency to obfuscators on page 46](#)
- [Deprecation of old platforms on page 46](#)

Model Pattern Library

The `NotifyPropertyChangedAttribute` aspect is a ready-made implementation of the `NotifyPropertyChanged` design pattern. The `PostSharp.Patterns.Contracts` namespace provides code contracts that can validate, at run time, the value of a parameter, a property, or a field.

Diagnostics Pattern Library

The `LogAttribute` and `LogExceptionAttribute` aspects provide a ready-made and high-performance implementation of a tracing aspect. They are compatible with the most popular logging framework, including log4net, nlog, and Enterprise Library.

Threading Pattern Library

PostSharp Threading Pattern Library invites you to raise the level of abstraction in which multithreading is being addressed. It provides three threading models: actors (**Actor**), reader-writer synchronized (`ReaderWriterSynchronizedAttribute`) and thread unsafe (`ThreadUnsafeAttribute`). Additionally, `BackgroundAttribute` and `DispatchedAttribute` allow you to easily dispatch a thread back and forth between a background and the UI thread.

Smart tags and wizards in Visual Studio

Smart tags allow for better discoverability of ready-made aspects and pattern implementations. When the aspect requires configuration, a wizard user interface collects the parameters and then generates the proper code.

Better platform support through Portable Class Libraries

Windows Phone, Windows Store and Silverlight are now first-class citizens. All features that are available for the .NET Framework now also work with these platforms. All platforms are supported transparently through the portable class library. To provide this feature, we had to develop the `PortableFormatter`, a portable serializer similar in function to the `BinaryFormatter`. All you have to do is to replace `[Serializable]` with `[PSerializable]`.

Unified deployment through NuGet and Visual Studio Gallery

Installation of PostSharp is now unified and built on top of Visual Studio Gallery and NuGet Package Manager.

Transparency to obfuscators

PostSharp no longer requires specific support from obfuscators, as it no longer uses strings to refer to metadata declarations.

Deprecation of old platforms

Support for Silverlight 3, .NET Compact Framework, and Mono has been deprecated.

5.13. What's New in PostSharp 2.1

The objective of release 2.1 was to fix a number of 'gray points' of the version 2.0, which added friction to the adoption path of PostSharp, or even prevented people from using the product.

PostSharp 2.1 includes the following improvements:

- [Build-time performance improvement on page 46](#)
- [Support for NuGet and improved no-setup experience on page 47](#)
- [Compatibility with obfuscators on page 47](#)
- [Extended reflection API on page 47](#)
- [Architectural validation on page 47](#)
- [Compatibility with Code Contracts on page 47](#)
- [Support for Silverlight 5.0 on page 47](#)
- [License server on page 47](#)

Build-time performance improvement

We traded our old text-based compilation engine to a brand new binary writer.

Support for NuGet and improved no-setup experience

PostSharp 2.1 can be installed directly from [NuGet](#)⁶. Local installation is no longer a requirement to use the Visual Studio Extension. However, because the setup program creates ngenned images, it still provides the faster experience.

Compatibility with obfuscators

PostSharp can now be used jointly, and without limitation of features, with some obfuscators.

Extended reflection API

The class `ReflectionSearch` allows you to programmatically navigate the structure of an assembly: find custom attributes of a given type, find children of a given type, find members of a given type, find methods referring a given type or members, or find members accessed from a given method.

Architectural validation

Architecture Validation allows you annotate your code with constraints, which define the conditions in which your API is allowed to be used. Constraints are verified at build time and their violation generates a build warning and an error. See [Validating Architecture on page 435](#) for details.

Compatibility with Code Contracts

PostSharp 2.1 can be used jointly with Microsoft Code Contracts. Aspects and contracts can be applied to the same method.

Support for Silverlight 5.0

Silverlight 5.0 is added to the list of supported platforms.

License server

The license server helps customer manage and deploy license keys. The license server is a simple ASP.NET application that can be deployed easily on any Windows machine. Its use is optional.

5.14. What's New in PostSharp 2.0

PostSharp 1.0 and 1.5 made aspect-oriented programming (AOP) popular in the .NET community. PostSharp 2.0 makes it mainstream by enhancing convenience (Visual Studio Extension), reliability (dependency enforcement), run-time performance (optimizer), and features (composite aspects, property- and event-level aspects).

PostSharp 2.0 includes the following improvements:

- [Visual Studio Extension on page 48](#)
- [Composite aspects \(advices and pointcuts\) on page 48](#)
- [Adaptive code generation on page 48](#)
- [Interception aspect for fields and properties on page 48](#)
- [Interception aspect for events on page 48](#)
- [Aspect dependencies on page 48](#)
- [Instance-scoped aspects on page 48](#)
- [Support for new platforms on page 48](#)
- [Build performance improvements on page 48](#)

6. <http://www.nuget.org/List/Packages/PostSharp>

Visual Studio Extension

As developers start being comfortable with PostSharp and add more and more aspects to their code, two questions become manifest: How can I know to which elements of code my aspect has been applied? How can I know which aspects have been applied to the element of code I am looking at? Answering these two questions is precisely what the PostSharp Extension for Visual Studio 2008 and 2010 has been designed for. It provides two new features to the IDE: an Aspect Browser tool window and new adornments of enhanced elements of code with clickable tooltip.

Composite aspects (advices and pointcuts)

Part of the success of PostSharp 1.5 was due to its ability to introduce aspects without appealing to barbaric terms such as advices and pointcuts. So why introduce them now? Because they make it easier to develop complex aspects. Thanks to advices and pointcuts, you can implement complex patterns such as observability awareness (`INotifyPropertyChanged`) with just a few lines of code. And just with PostSharp 1.5, you can still write your own aspects without knowing about advices and pointcuts.

Adaptive code generation

PostSharp 2.0 generates much smarter, faster, and smaller code than before. Let's face it: PostSharp 1.5 was quite dumb. It generated a lot of instructions that your aspects did not even need. PostSharp 2.0 analyzes your aspect to see which features are actually being used at run time, and generates only instructions that support these features. Result: you could probably not write much faster code by hand.

Interception aspect for fields and properties

PostSharp 2.0 comes with a new kind of aspect that handles fields and properties: `LocationInterceptionAspect` (in replacement of `OnFieldAccessAspect`). The aspect is much more usable than its predecessor; for instance, it is possible to call the field or property getter from the setter.

Interception aspect for events

The new aspect kind `EventInterceptionAspect` allows an aspect to intercept all event semantics: add, remove, and fire.

Aspect dependencies

By enforcing aspect dependency rules, PostSharp ensures that aspects behave in a predictable and robust way, even when multiple aspects are applied to the same element of code. This feature is important for large and complex projects, where aspects may be written by different teams, or provided by numerous third-party vendors who don't know about each other.

Instance-scoped aspects

In PostSharp 1.5, all aspects had static scope, i.e. there was a single instance of the aspect for every element of code to which they applied. It is now possible to define aspects that have instance lifetime. For instance, if the aspect is applied to an instance field, a new instance of the aspect will be created for every instance of the type declaring the field. This is named an instance-scoped aspect.

Support for new platforms

- Microsoft .NET Framework 4.0
- Microsoft Silverlight 3.0
- Microsoft Silverlight 4.0
- Microsoft Windows Phone 7 (Applications and Games)
- Microsoft .NET Compact Framework 3.5
- Novell Mono 2.6

Build performance improvements

Just starting the CLR and loading system assemblies takes considerable time, too much for an application (such as PostSharp) that is typically started very frequently and whose running time is just a couple of seconds. To cope with this issue, PostSharp now preferably runs as a background application

5.15. What's New in PostSharp 1.5

PostSharp 1.5 was published 3 years after the start of the project, and was the first release to be really production-ready.

PostSharp 1.5 includes the following improvements:

- [Aspect inheritance on page 49](#)
- [Reading assemblies without loading them in the CLR on page 49](#)
- [Lazy loading of assemblies on page 49](#)
- [Build-time performance improvement on page 49](#)
- [Support for Mono on page 49](#)
- [Support for Silverlight 2.0 and the Compact Framework 2.0 on page 49](#)
- [Pluggable aspect serializer & partial trust on page 49](#)

Aspect inheritance

It is now possible to put an aspect on an interface and have it implicitly applied to all classes implementing that interface. The same works with classes, virtual or interface methods, and parameters of virtual or interface methods. Read more...

Reading assemblies without loading them in the CLR

In version 1.0, PostSharp required assemblies to be loaded in the CLR (i.e. in the application domain) to be able to read them. This limitation belongs to the past. When PostSharp processes a Silverlight or a Compact Framework assembly, it is never loaded by the CLR.

Lazy loading of assemblies

When PostSharp has to load a dependency assembly, it now reads only the metadata objects it really needs, resulting in a huge performance improvement and much lower memory consumption.

Build-time performance improvement

The code has been carefully profiled and optimized for maximal performance.

Support for Mono

PostSharp is now truly cross-platform. Binaries compiled on the Microsoft platform can be executed under Novell Mono. Both Windows and Linux are tested and supported. A NAnt task makes it easier to use PostSharp in these environments.

Support for Silverlight 2.0 and the Compact Framework 2.0

You can add aspects to your projects targeting Silverlight 2.0 or the Compact Framework 2.0.

Pluggable aspect serializer & partial trust

Previously, all aspects were serializers using the standard .NET binary formatter. It is now possible to choose another serializer or implement your own, and enhance assemblies that be executed with partial trust.

PART 2

Deployment and Configuration

CHAPTER 6

Deployment

PostSharp has been designed for easy deployment in typical development environments. Over the years, source control and build servers have become the norm, so we optimized PostSharp for this deployment scenario.

In most situations, PostSharp should work just fine without any advanced configuration. This chapter includes a detailed description of all deployment and configuration scenarios.

It contains the following topics:

Chapter	Description
Requirements and Compatibility on page 53	This topic lists the requirements for development, build and end-user devices.
PostSharp Components on page 57	This topic contains a summary of PostSharp components.
Installing PostSharp Tools for Visual Studio on page 59	This topic shows how to install PostSharp Tools for Visual Studio using an interactive installer.
Installing PostSharp Tools for Visual Studio Silently on page 59	This topic shows how to install PostSharp Tools for Visual Studio silently using a command line interface.
Installing PostSharp Into a Project on page 60	This topic shows how to install PostSharp Compiler into your project using NuGet Package Manager.
Installing PostSharp without NuGet on page 61	This topic shows how to install PostSharp Compiler into your project without NuGet Package Manager.
Using PostSharp on a Build Server on page 62	This topic shows how to use PostSharp on a build server and describes licensing details for this scenario.
Upgrading from a Previous Version of PostSharp on page 65	This topic explains how to upgrade from a previous version of PostSharp.
Uninstalling PostSharp on page 67	This topic shows how to uninstall PostSharp from your projects and from Visual Studio.
Deploying PostSharp to End-User Devices on page 73	This topic describes redistribution of PostSharp run-time libraries to end-user devices.

6.1. Requirements and Compatibility

You can use PostSharp to build applications that target a wide range of target devices. This article lists the requirements for development, build and end-user devices.

This topic contains the following sections:

- [Supported programming languages on page 54](#)
- [Requirements on development workstations and build servers on page 54](#)

- [Requirements on end-user devices on page 55](#)
- [Compatibility with ASP.NET on page 55](#)
- [Compatibility with Microsoft Code Analysis on page 55](#)
- [Compatibility with Microsoft Code Contracts on page 55](#)
- [Compatibility with Obfuscators on page 56](#)
- [Known Incompatibilities on page 56](#)

IMPORTANT NOTE

Please read our [Supported Platforms Policies](#)⁷ on our web site as it contains important explanations, restrictions and disclaimers regarding this article.

Supported programming languages

This version of PostSharp supports the following languages:

- C# 8.0,
- VB 15.5.

You may use PostSharp with an unsupported language version at your own risks by setting the `PostSharpSkipLanguageVersionValidation` MSBuild property to `True`. There are two risks in doing that: inconsistent or erroneous behavior of the current version of PostSharp, and breaking changes in the future version of PostSharp that will support this language version.

Requirements on development workstations and build servers

This section lists the supported platforms, and most importantly platform versions, on which PostSharp is intended to run.

The following software components need to be installed before PostSharp can be used:

- Any of the following versions of Microsoft Visual Studio:
 - Visual Studio 2015 Update 3.
 - Visual Studio 2017 RTW (15.0) or Update 1 (15.9).
 - Visual Studio 2019.

The debugging experience may be inconsistent with other IDEs than Visual Studio or when PostSharp Tools for Visual Studio are not installed.

- .NET Framework 4.7.2 or later.
- Any of the following operating systems:
 - Windows 10: any version in mainstream Microsoft support, except LTSB and S editions.
 - On build agents only: Windows Server 2012, Windows Server 2012 R2, Windows Server 2016, Ubuntu 16.04, Ubuntu 18.04, Alpine 3.10, macOS 10.14.
- Optionally, one of the following versions of .NET Core:
 - .NET Core SDK 2.2.
 - .NET Core SDK 3.0.

7. <https://www.postsharp.net/support/policies#platforms>

Requirements on end-user devices

The following table displays the versions of the target frameworks that are supported by the current release of PostSharp and its components.

Package	.NET Framework	.NET Core	.NET Standard*
<i>PostSharp</i>	3.5 SP1, 4.0*, 4.5*, 4.6, 4.7, 4.8	2.1, 2.2, 3.0	1.3, 1.4, 1.5, 1.6, 2.0, 2.1
<i>PostSharp.Patterns.Common</i>	4.0*, 4.5*, 4.6, 4.7, 4.8	2.1, 2.2, 3.0	1.3, 1.4, 1.5, 1.6, 2.0, 2.1
<i>PostSharp.Patterns.Aggregation</i>			
<i>PostSharp.Patterns.Threading</i>			
<i>PostSharp.Patterns.Model</i>			
<i>PostSharp.Patterns.Diagnostics</i>			
<i>PostSharp.Patterns.Xaml</i>	4.0*, 4.5*, 4.6, 4.7, 4.8	-	-
<i>PostSharp.Patterns.Caching</i>	4.6, 4.7, 4.8	2.1, 2.2, 3.0	2.0, 2.1

NOTE

.NET Framework 4.0 and 4.5 are no longer supported by Microsoft. Although we still provide libraries targeting them, we no longer run our tests on these specific versions of the .NET Framework.

NOTE

PostSharp does not implicitly support all platforms that support .NET Standard. Only platforms mentioned in this table are supported.

Compatibility with ASP.NET

There are two ways to develop web applications using Microsoft .NET:

- **ASP.NET Application projects** are very similar to other projects; they need to be built before they can be executed. Since they are built using MSBuild, you can use PostSharp as with any other kind of project.
- **ASP.NET Site projects** are very specific: there is no MSBuild project file (a site is actually a directory), and these projects must not be built. ASP.NET Site projects are not supported.

Compatibility with Microsoft Code Analysis

By default, PostSharp reconfigures the build process so that Code Analysis is executed on the assemblies as they were *before* being enhanced by PostSharp. If you are using Code Analysis as an integrated part of Visual, no change of configuration is required.

You can request the Code Analysis to execute on the output of PostSharp by setting the `ExecuteCodeAnalysisOnPostSharpOutput` MSBuild property to `True`. For more information, see [Configuring Projects Using MSBuild on page 93](#).

Compatibility with Microsoft Code Contracts

PostSharp configures the build process so that Microsoft Code Contracts is executed before PostSharp. Additionally, Microsoft Code Contracts' static analyzer will be executed synchronously (instead of asynchronously without PostSharp), which will significantly impact the build performance.

Compatibility with Obfuscators

PostSharp generates assemblies that are theoretically compatible with all obfuscators.

NOTE

PostSharp Logging is not designed to work with obfuscated assemblies.

CAUTION NOTE

PostSharp emits constructs that are not emitted by Microsoft compilers (for instance `methodof`). These unusual constructs may reveal bugs in third-party tools, because they are generally tested against the output of Microsoft compilers.

Known Incompatibilities

PostSharp is not compatible with the following products or features:

Product or Feature	Reason	Workaround
Visual Studio 2013	No longer under Microsoft mainstream support	Use PostSharp 6.0.
Visual Studio 2010	No longer under Microsoft mainstream support	Use PostSharp 3.1.
Visual Studio 2012	No longer under Microsoft mainstream support	Use PostSharp 5.0.
ILMerge	Bug in ILMerge	Use another merging product (such as ILPack, SmartAssembly).
Edit-and-Continue	Not Supported	Rebuild the project after edits
Silverlight 3 or earlier	No longer under Microsoft mainstream support	Use PostSharp 2.1.
Silverlight 4	No longer under Microsoft mainstream support	Use PostSharp 3.1.
Silverlight 5	Low customer demand.	Use PostSharp 4.3.
.NET Compact Framework	No support for PCL	Use PostSharp 2.1.
.NET Framework 2.0	No longer under Microsoft mainstream support	Target .NET Framework 3.5 or use PostSharp 3.1.
Windows Phone 7	No longer under Microsoft mainstream support	Use PostSharp 3.1
Windows Phone 8, WinRT	Low customer demand.	Use PostSharp 4.3
Visual Studio Express	Microsoft's licensing policy	Use Visual Studio Community Edition
ASP.NET Web Sites	Not built using MSBuild	Convert the ASP.NET Web Site to an ASP.NET Web Application.
Universal Windows Platform (UWP)	Not supported (low customer demand)	Contact PostSharp support team.

Product or Feature	Reason	Workaround
Xamarin	Support suspended (deprioritized because of low customer demand)	Use PostSharp 4.3. Contact PostSharp support team to discuss prioritization.
Mono, Unity3D	Unsupported	None.

6.2. PostSharp Components

PostSharp is composed of the following artifacts:

- [PostSharp Tools for Visual Studio on page 57](#)
- [NuGet packages on page 57](#)
- [Zip distribution on page 58](#)

PostSharp Tools for Visual Studio

This is the user interface of PostSharp. It extends the Visual Studio editor and provides a new menu, option pages, toolbox windows, diagnostics, code actions, and debugging enhancements.

For details regarding the installation of this component, see [Installing PostSharp Tools for Visual Studio on page 59](#) and [Installing PostSharp Tools for Visual Studio Silently on page 59](#).

NuGet packages

All build-time and run-time artifacts are released as NuGet packages. Build-time packages are required to build your projects, but only the content of run-time packages is required to execute your applications.

If you build NuGet packages that use PostSharp but does not define custom aspects, your package should only reference the relevant PostSharp run-time packages, not the build-time ones.

NOTE

The PostSharp License Agreement refers to run-time packages as *redistributables*. The license agreement allows for royalty-free redistribution of run-time packages, but stricter conditions apply to the redistribution of build-time packages.

The following table lists all PostSharp packages:

Run-time package	Build-time package	Description
<i>PostSharp.Redist</i>	<i>PostSharp</i>	PostSharp Framework. The build-time package includes the PostSharp compiler.
<i>PostSharp.Patterns.Common.Redist</i>	<i>PostSharp.Patterns.Common</i>	Common logic shared between pattern libraries. Code contracts.
<i>PostSharp.Patterns.Aggregation.Redist</i>	<i>PostSharp.Patterns.Aggregation</i>	Aggregatable and Disposable aspects.
<i>PostSharp.Patterns.Model.Redist</i>	<i>PostSharp.Patterns.Model</i>	NotifyPropertyChanged aspect and Undo/Redo.
<i>PostSharp.Patterns.XAML.Redist</i>	<i>PostSharp.Patterns.XAML</i>	Command, Dependency Property and Attached Property aspects. WPF controls for undo/redo.

Run-time package	Build-time package	Description
<i>PostSharp.Patterns.Threading.Redist</i>	<i>PostSharp.Patterns.Threading</i>	Threading models, thread dispatching aspects, deadlock detection.
<i>PostSharp.Patterns.Caching.Redist</i>	<i>PostSharp.Patterns.Caching</i>	Caching aspect.
<i>PostSharp.Patterns.Caching.Redis</i>	N/A	Redis connector for <i>PostSharp.Patterns.Caching</i> .
<i>PostSharp.Patterns.Caching.Azure</i>	N/A	Azure connector for <i>PostSharp.Patterns.Caching</i> .
<i>PostSharp.Patterns.Diagnostics.Redist</i>	<i>PostSharp.Patterns.Diagnostics</i>	Logging aspect.
<i>PostSharp.Patterns.Diagnostics.ApplicationInsights</i>	N/A	Application Insights connector for <i>PostSharp.Patterns.Diagnostics</i> .
<i>PostSharp.Patterns.Diagnostics.CommonLogging</i>	N/A	Common.Logging connector for <i>PostSharp.Patterns.Diagnostics</i> .
<i>PostSharp.Patterns.Diagnostics.EnterpriseLibrary</i>	N/A	Enterprise Library connector for <i>PostSharp.Patterns.Diagnostics</i> .
<i>PostSharp.Patterns.Diagnostics.Log4Net</i>	N/A	Log4Net connector for <i>PostSharp.Patterns.Diagnostics</i> .
<i>PostSharp.Patterns.Diagnostics.Microsoft</i>	N/A	Microsoft.Extensions.Logging connector for <i>PostSharp.Patterns.Diagnostics</i> .
<i>PostSharp.Patterns.Diagnostics.NLog</i>	N/A	NLog connector for <i>PostSharp.Patterns.Diagnostics</i> .
<i>PostSharp.Patterns.Diagnostics.Serilog</i>	N/A	Serilog connector for <i>PostSharp.Patterns.Diagnostics</i> .
<i>PostSharp.Patterns.Diagnostics.Tracing</i>	N/A	System.Diagnostics connector for <i>PostSharp.Patterns.Diagnostics</i> .
<i>PostSharp.Patterns.Diagnostics.Loupe</i>	N/A	Loupe connector for <i>PostSharp.Patterns.Diagnostics</i> .

Zip distribution

For teams that cannot use NuGet, PostSharp also comes as one zip archive containing the files otherwise contained in all NuGet packages.

In this archive, the *lib* folder contains run-time libraries (*redistributables*), and the *tools* folder contains all build-time components.

See [Installing PostSharp without NuGet on page 61](#) for details.

6.3. Installing PostSharp Tools for Visual Studio

PostSharp Tools for Visual Studio are PostSharp's user interface. Install them on a developer's computer, does not affect the projects until the PostSharp NuGet package has been added to this project. See [Installing PostSharp Into a Project on page 60](#) for details.

To install PostSharp Tools for Visual Studio:

1. Download the file *PostSharp-full-X.X.X.exe* from <https://www.postsharp.net/download>.
2. Run the file *PostSharp-full-X.X.X.exe*.
3. Complete the installation configuration wizard. You will be asked to enter a license key or to start the trial period. The wizard may ask the permission to install NuGet Package Manager or to uninstall the user interface of PostSharp.

NOTE

The installer mentioned above contains Visual Studio extension packages for each version of Visual Studio supported by PostSharp. To save some space, use the file *PostSharp-web-X.X.X.exe* from <https://www.postsharp.net/downloads>. This file contains the installer only and each Visual Studio extension package is downloaded on the fly during the installation process. The packages for Visual Studio versions not installed on your machine are not downloaded.

Other Resources

[Installing PostSharp Into a Project on page 60](#)

[Uninstalling PostSharp on page 67](#)

[PostSharp Technologies](#)

6.4. Installing PostSharp Tools for Visual Studio Silently

PostSharp is composed of a user interface (PostSharp Tools for Visual Studio) and build components (NuGet packages). NuGet packages are usually checked into source control or retrieved from a package repository at build time (see [Restoring Packages at Build Time on page 62](#)), so its deployment does not require additional automation. The user interface is typically installed by each user. It does not require administrative privileges.

In large teams, it might be inconvenient to install PostSharp Tools for Visual Studio on each machine manually. For this purpose, PostSharp installer enables silent installation using a command line interface. You can install PostSharp automatically for a large number of users using the silent installer.

To install PostSharp unattended:

1. Download the installer from <https://www.postsharp.net/download>. The installer is a file named *PostSharp-full-X.X.X.exe*.
2. Extract the installer files using the following command line:

```
PostSharp-full-X.X.X.exe /extract %TEMP%\PostSharp-full.X.X.X
```

- Execute the following command line:

```
%TEMP%\PostSharp-full.X.X.X\PostSharp.Settings.exe /setup
```

In the command line above, the following arguments are optional but recommended:

Argument	Description
/ceip	Allows PostSharp to collect anonymous information about the way you are using the software. We never collect your source code. See our privacy policy for details. If you don't specify this flag, CEIP will be disabled, and the user will not be asked to enable it.
/license 000-AAAAAAAAAAAAAAAA	Installs the license key for the current user in registry. In this argument, 000-AAAAAAAAAAAAAAAA must be replaced by the license key or the URL to the license server.
/license-all 000-AAAAAAAAAAAAAAAA	Installs the license key for all users in registry. In this argument, 000-AAAAAAAAAAAAAAAA must be replaced by the license key or the URL to the license server.

CAUTION NOTE

If the /license and /license-all arguments are omitted, the license keys on the machine will remain unchanged or uninstalled.

NOTE

If any Visual Studio instance or other processes affected by the installer is running during the installation process, the installer lists all the blocking processes, and fails. You need to run silent installation when none of these processes is running.

6.5. Installing PostSharp Into a Project

The compiler components of PostSharp are distributed as a NuGet package named simply *PostSharp*. If you want to use PostSharp in a project, you simply have to add this NuGet package to the project.

This topic contains the following sections:

- [Adding PostSharp to a project on page 60](#)
- [Including files in your source control system on page 61](#)

Adding PostSharp to a project

To add PostSharp to a project:

- Open the **Solution Explorer** in Visual Studio.
- Right-click on the project.
- Click on **Add PostSharp**.

- <https://www.postsharp.net/company/legal/privacy-policy#ceip>

TIP

Remember that adding PostSharp to a project just means adding the *PostSharp* NuGet package. If you want to add PostSharp to several projects in a solution, it may be easier to use NuGet to manage packages at the solution level. You may need to select the **Include Prerelease** option to install a prerelease version of PostSharp.

TIP

NuGet Package Manager can be configured using a file named *nuget.config*, which can be checked into source control and can specify, among other settings, the location of the package repository (if it must be shared among several solutions, for instance) or package sources (if packages must be pre-approved). See [NuGet Configuration File](#)⁹ and [NuGet Configuration Settings](#)¹⁰ for more information.

Including files in your source control system

After you add PostSharp to a project, you need to add the following files to source control:

- *packages.config*
- *postsharp.config*, if any
- **.psproj*, if any
- **.pssln*, if any

Some of the files above might not be present depending on the package management format you use (*packages.config*/*PackageReference*) and on which PostSharp packages you have installed in your project.

Optionally, if you use the *packages.config* package management format, you can also include the *packages* folder in your source control. Note that there are negative consequences on this practice. See [Omitting NuGet packages in source control systems](#)¹¹ for more information. If you choose not to include the *packages* folder in your source control system, read [Restoring Packages at Build Time](#) on page 62.

Once you have all of these files included in your source code repository, any other developer getting that source code from the repository will have the required information to be able to build the application.

6.6. Installing PostSharp without NuGet

The most common way to add PostSharp to your project is by installing PostSharp NuGet packages. The main benefit of using NuGet Package Manager is that it provides a standard way to install and manage all dependencies for your .NET projects.

Previous versions of NuGet had several issues that made it an impractical solution for some teams. For this reason, we allow to use PostSharp without NuGet, by downloading and extracting a standard zip file. However, the installation procedure is significantly more cumbersome without NuGet than with NuGet.

To install PostSharp into a project without NuGet:

1. Download the zip distribution from <https://www.postsharp.net/downloads> (a file named *PostSharp-x.x.x.zip*).
2. Extract the zip file into some local directory.

9. <http://docs.nuget.org/docs/reference/nuget-config-file>

10. <http://docs.nuget.org/docs/reference/nuget-config-settings>

11. <https://docs.microsoft.com/en-us/nuget/consume-packages/packages-and-source-control>

3. Using Visual Studio, add references to the relevant PostSharp assemblies. They are located under the *lib* directory.
4. Open the project file with a text editor and add the following line just after the last `Import` element.

```
<ImportProject="..\..\..\postsharp\Tools\PostSharp.targets"/>
```

6.7. Using PostSharp on a Build Server

PostSharp has been designed for frictionless use on build servers. PostSharp build-time components are deployed as NuGet packages, and are integrated with MSBuild. No component needs to be installed or configured on the build server, and no extra build step is necessary. If you choose not to check in NuGet packages in your source control, read [Restoring Packages at Build Time on page 62](#).

Installing a License on the Build Server

There are several ways to install a license on the build server:

- Don't install it. It is not necessary to install the license key on the build server unless you are using the features of PostSharp Logging.
- Add your license key to the *postsharp.config* file and add this file to the source repository as described in [Deploying License Keys on page 75](#).
- Set an environment variable named `PostSharpLicense` to a semicolon-separated list of your license keys.

We do not recommend to install the license key on a build server using the user interface.

6.7.1. Restoring Packages at Build Time

NuGet Package Manager has the ability to restore packages from their repository during the build. This allows teams to avoid storing NuGet packages in their source repository.

You can restore the PostSharp package at build time as long as the package is restored before MSBuild is invoked to build the project.

The reason is that the *PostSharp.targets* file is required during the build, otherwise PostSharp is not inserted in the build process, and simply does not work. Because of the design of MSBuild, *PostSharp.targets* must be present when the build starts, so it cannot be restored from the package repository during the same build. The build that triggers the package restore will either fail or run without PostSharp, and subsequent builds will succeed. A rebuild is then required.

This behavior is acceptable on developer workstations. However, on build servers, you must ensure that the packages are restored *before* the project is built.

The way you restore packages differs with the version of NuGet:

- [NuGet 2.7 and Later on page 62](#)
- [Visual Studio 2017 and Later on page 63](#)
- [.NET Core Command Line Interface on page 63](#)
- [NuGet 2.0 to 2.6 on page 63](#)

NuGet 2.7 and Later

To restore the PostSharp package at build time, add a preliminary step before building the Visual Studio solutions or projects. This step should execute the following command:

```
NuGet.exe restore MySolution.sln
```

In this command, the *MySolution.sln* is the solution for which packages have to be restored.

To restore packages for a solution where some projects use the packages.config package management format and others use the PackageReference package management format, use this way as well. It will restore packages in all the projects, regardless of the package management format used.

See [NuGet Command-Line Reference](#)¹² for details.

Visual Studio 2017 and Later

If all your projects use the PackageReference package management format, you can use the MSBuild Restore target to restore NuGet packages. To restore the PostSharp package at build time, add a preliminary step before building the Visual Studio solutions or projects. This step should execute the following command:

```
MSBuild /T:Restore MySolution.sln
```

In this command, the *MySolution.sln* is the solution for which packages have to be restored. Eventually, you can call the MSBuild target from your MSBuild script.

CAUTION NOTE

If some projects of your solution use the packages.config project management format, those will not get the NuGet packages restored this way. Use the NuGet.exe command described in the first section instead.

See [NuGet pack and restore as MSBuild targets](#)¹³ for details.

.NET Core Command Line Interface

If all your projects are .NET Core projects, you can use the .NET Core Command Line Interface to restore NuGet packages. To restore the PostSharp package at build time, add a preliminary step before building the Visual Studio solutions or projects. This step should execute the following command:

```
dotnet restore MySolution.sln
```

In this command, the *MySolution.sln* is the solution for which packages have to be restored.

CAUTION NOTE

If some projects of your solution are not .NET Core projects, those will not get the NuGet packages restored this way correctly. Use the NuGet.exe command described in the first section instead.

See [dotnet-restore](#)¹⁴ for details.

NuGet 2.0 to 2.6

To restore the PostSharp package at build time, add a preliminary step before building the Visual Studio solutions or projects. This step should execute the following command for every *packages.config* file in your solution (typically, for every project):

```
NuGet.exe install packages.config -OutputDirectory SolutionDirectory\packages
```

12. <http://docs.nuget.org/docs/reference/command-line-reference>

13. <https://docs.microsoft.com/en-us/nuget/schema/msbuild-targets#restore-target>

14. <https://docs.microsoft.com/en-us/dotnet/core/tools/dotnet-restore>

In this command, where *SolutionDirectory\packages* is the directory where the NuGet packages should be installed. Please look at the [NuGet Command-Line Reference](#)¹⁵ for details.

TIP

You can use PowerShell or MSBuild to execute the `nuget install` command to all *packages.config* files in your source repository.

6.7.2. Using PostSharp with Visual Studio Online

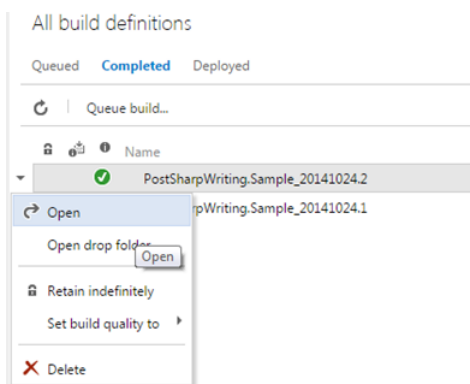
When hosting your source code on Visual Studio Online, adding PostSharp to the codebase is no different than for any other build server situation.

Visual Studio Online offers an online build server environment. Once configured the build server will retrieve your source code and compile the application for you. As part of this build process, you will want any PostSharp aspects to be added in the same way that it occurs on your local development machine. To do this you will have to ensure that your codebase includes PostSharp as outlined in the [Installing PostSharp Into a Project on page 60](#) section. You will also need to configure a build definition as outlined in the [Create or edit build definition](#)¹⁶ article on MSDN.

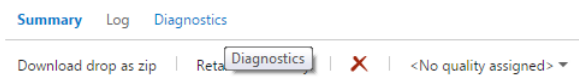
Once you are able to successfully run the build you will want to review the build logs and verify the artifacts that were created by that build. Here's how you can verify that your build included your PostSharp aspects.

Verifying Visual Studio Online Builds

1. To review the build logs, open the successful build.



2. Select the Diagnostics tab.



15. <http://docs.nuget.org/docs/reference/command-line-reference>

16. <http://msdn.microsoft.com/en-us/library/ms181716.aspx>

3. Ensure that the installation of PostSharp and any PostSharp patterns libraries that you used.

```

Pull sources from Git repo ▾
Cloning repository 'https://postsharpwriting.visualstudio.com/Def
20 object(s) were downloaded with a total size of 0.01 MB.
Checking out branch refs/heads/master.
Setting the source get version of the build to LG:refs/heads/maste
Associate the changesets that occurred since the last good build ▾
Try
Compile, Test and Publish
Run optional script before MSBuild ▾
Run MSBuild ▾
C:\Nightrail\Services\Mms\BuildProvisioner\Tools\nuget.exe re
Installing 'PostSharp 4.0.34'.
Installing 'PostSharp.Patterns.Common 4.0.34'.
Installing 'PostSharp.Patterns.Diagnostics 4.0.34'.
Installing 'PostSharp.Patterns.Threading 4.0.34'.
Successfully installed 'PostSharp.Patterns.Threading 4.0.34'.
Successfully installed 'PostSharp.Patterns.Diagnostics 4.0.34'.
Successfully installed 'PostSharp.Patterns.Common 4.0.34'.
Successfully installed 'PostSharp 4.0.34'.
C:\Program Files (x86)\MSBuild\12.0\bin\amd64\MSBuild.exe
/p:SkipInvalidConfigurations=true /m /p:OutDir="C:\a\bin\\" /
/d:WorkflowCentralLogger,"C:\Nightrail\Services\Mms\BuildPr
utDirectoryItems,GetTargetPath;TFSUrl=https://postsharpwriti
b832-cc4e35a6f773.vstfs:///Build/Build/4" /p:BuildLabel="Pos

```

If you see entries like these in your build log you know that the build process correctly downloaded the PostSharp components.

If you do not see any entries for the downloading of the PostSharp components you will want to ensure that the packages.config file is correctly included in your source code repository and that the PostSharp dependencies are referenced in the appropriate projects.

6.8. Upgrading from a Previous Version of PostSharp

This section explains how to upgrade from a previous version of PostSharp.

This topic contains the following sections:

- [Upgrading PostSharp Tools for Visual Studio on page 65](#)
- [Upgrading solutions from PostSharp 3 or later on page 66](#)
- [Upgrading large repositories on page 66](#)
- [Upgrading solutions from PostSharp 2 on page 66](#)

TIP

Other sections of this chapter, specifically [Installing PostSharp Tools for Visual Studio on page 59](#), [Deploying License Keys on page 75](#) and [Using PostSharp on a Build Server on page 62](#), are also useful if you need to upgrade from an earlier version of PostSharp.

Upgrading PostSharp Tools for Visual Studio

After you install PostSharp Tools for Visual Studio, you will still be able to open solutions that use older versions of PostSharp.

PostSharp Tools for Visual Studio are backward compatible with older versions of PostSharp. However, several versions of the extension cannot coexist. Therefore, installing a new version of PostSharp Tools will uninstall the previous version.

To upgrade PostSharp Tools for Visual Studio, simply download it from <https://www.postsharp.net/download> and execute the installation package.

CAUTION NOTE

Upgrading PostSharp Tools for Visual Studio does not implicitly upgrade your source code.

Upgrading solutions from PostSharp 3 or later

CAUTION NOTE

Before you upgrade your project to a different major release of PostSharp, check that the new version still supports your version Visual Studio and the target framework of your application. Check the release notes for an accurate compatibility list of the specific version you are installing.

You can use several versions of PostSharp side-by-side on the same machine. However, it is recommended that you use the same version in all projects of the same solution.

To upgrade a solution from PostSharp 3 or later:

1. Open the **Solution Explorer** in Visual Studio.
2. Right-click on the solution.
3. Click on **Manage NuGet Packages for Solution**.
4. Click on **Updates**.
5. Find the *PostSharp* package and click on **Update**.
6. Select all projects, click **OK**.
7. Repeat the operation for all *PostSharp.Patterns.** packages.

Upgrading large repositories

If your source contains a large number of solutions, upgrading manually using NuGet may be too labor intensive. In this situation, it is better to use our upgrade PowerShell script.

To upgrade a large number of solutions with the PowerShell script:

1. Download the following Git repository: <https://github.com/sharpcrafters/PostSharp.Utilities>. You can download it manually from the web page or execute the following command:

```
git clone https://github.com/sharpcrafters/PostSharp.Utilities.git
```

2. Follow instructions on in *README.md*.

CAUTION NOTE

This script does not support other platforms than the .NET Framework and does not support PostSharp Pattern Libraries.

Upgrading solutions from PostSharp 2

Every project can have only references to a single version of PostSharp. This applies both to direct and indirect references. The PostSharp 3 or later compiler is not backward compatible with PostSharp 2, and PostSharp 3 will refuse to compile projects that have a reference to PostSharp 2. Therefore, you will typically use a single version of PostSharp in every solution.

You can upgrade projects from PostSharp 2 to PostSharp 3 by adding the *PostSharp* NuGet package to these projects.

To upgrade a solution from PostSharp 2:

1. Open the **Solution Explorer** in Visual Studio.
2. Right-click on the solution.
3. Click on **Manage NuGet Packages for Solution**.
4. Click on **Online**.
5. In the search box, type PostSharp. You may want to select the **Select prereleases** option (instead of the default **Stable Only**) to install a pre-release version of PostSharp.
6. Find the *PostSharp* package and click on **Install**.
7. Select all projects, click **OK**.

Although PostSharp 3 or later is mostly backward compatible with PostSharp 2 at source-code level, you may need to perform small adjustments to your source code:

- Every occurrence of the `_Assembly` interface has been replaced by the `Assembly` classes. You may have to change the signatures of some methods derived from `AssemblyLevelAspect`.
- Aspects that target Silverlight, Windows Phone or Windows Store must be annotated with the `PSerializableAttribute` custom attribute.
- PostSharp Toolkits 2.1 need to be uninstalled using NuGet. Instead, you can install PostSharp Pattern Libraries 3 from NuGet. Namespaces and some type names have been changed.

6.9. Uninstalling PostSharp

If you make the decision to remove PostSharp from your project we are sorry to see you leave.

There are two scenarios you may want to consider: removing PostSharp from individual projects or solutions, and removing PostSharp from Visual Studio.

This topic contains the following sections:

- [Removing PostSharp from your projects and solutions on page 67](#)
- [Removing PostSharp from Visual Studio on page 70](#)

Removing PostSharp from your projects and solutions

Here are some steps to follow to remove PostSharp from your project.

CAUTION NOTE

As you'll see in these steps, removing the product from your project is not that difficult. However, replacing the aspects that you were using will be a much more arduous task that will require a great deal of planning.

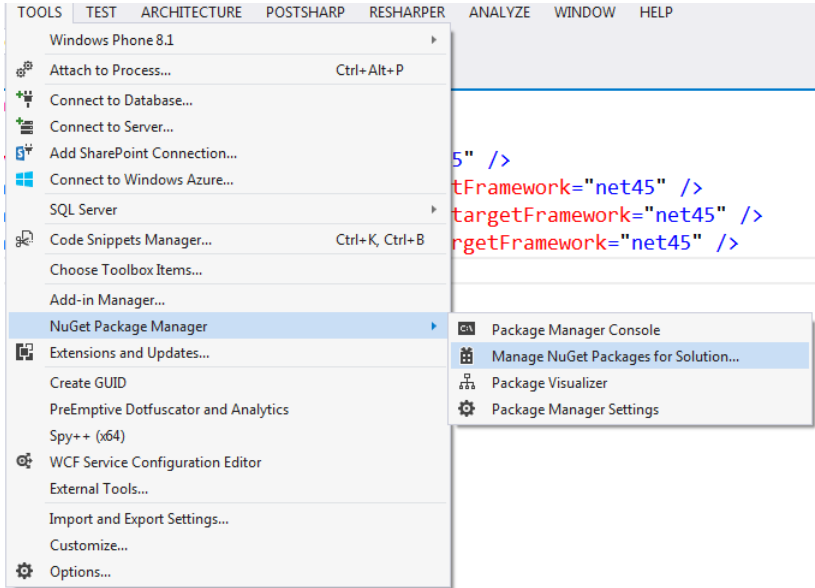
You will need to replace the aspects by handwritten source code that implement the same behaviors. Depending on how intensively you used PostSharp, your codebase could significantly increase as a result of stopping using PostSharp. Other products and frameworks that pretend to implement aspect-oriented programming actually only provide a small subset of the features you are got used to with PostSharp.

Because every project will use aspects differently, and some will have custom aspects, we are unable to provide you with any generic piece of advice about how to replace specific aspects.

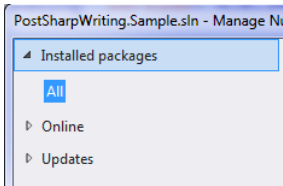
To remove PostSharp from a project, you simply have to remove all PostSharp packages from it. The following procedure demonstrates how to remove PostSharp for the whole solution.

Removing PostSharp with NuGet Packages Manager for Solution

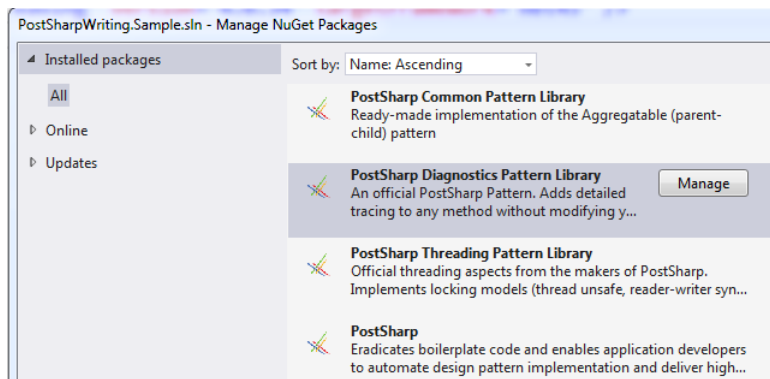
1. Open the Package Manager for Solution windows



2. Select the All tab from the left side of the window.



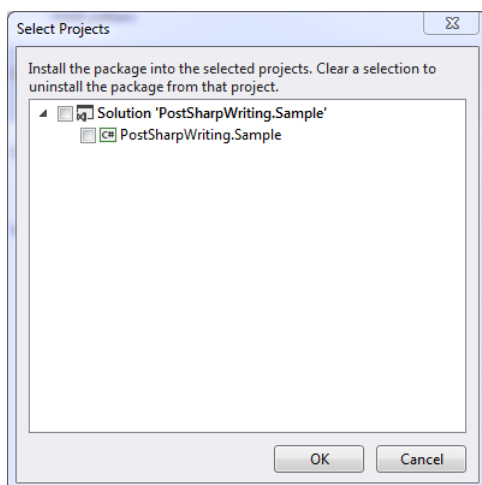
- Find the PostSharp packages in the list and select one of the PostSharp Library packages. Click the Manage button.



NOTE

Start by selecting the PostSharp Library Packages and working in reverse dependency order. This will result in the main PostSharp package being the last one that you select to remove.

- Ensure that you uncheck all of the projects listed in the window and click OK.



- Repeat steps 3 and 4 for each of the PostSharp packages that show in the Packages Manager for Solution window.
- To verify that all of the PostSharp packages have been removed from your codebase, ensure that there are no PostSharp packages listed in the Packages Manager for Solution window.

Once you have removed all of the PostSharp packages from your codebase it is most probable that your application will no longer compile. Compilation errors will be registered where PostSharp aspect attributes exist in the codebase as well as where custom aspects were written. You will need to remove these entries from your codebase to get it to compile again.

Simply deleting the offending code can accomplish this. You must remember that in the process of removing PostSharp from your codebase these errors indicate locations where you are removing functionality from the codebase as well. If the functionality that is being removed is required by the application you will need to determine how to provide that functionality in the codebase going forward. This is the most difficult part of removing PostSharp from your codebase.

Because aspects can be used in a multitude of different manners, and custom aspects can be created for any number of different uses, there is no practical way to tell you how to replace the functionality being lost.

NOTE

You now have removed PostSharp from your codebase. At this point, you are able to continue on your development effort without making use of PostSharp.

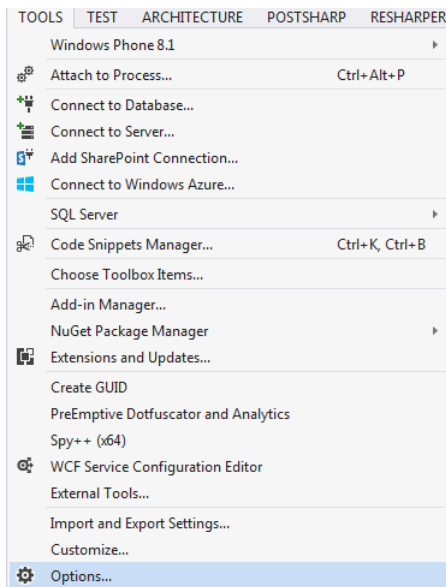
If you would like to remove PostSharp from Visual Studio, proceed with the following steps.

Removing PostSharp from Visual Studio

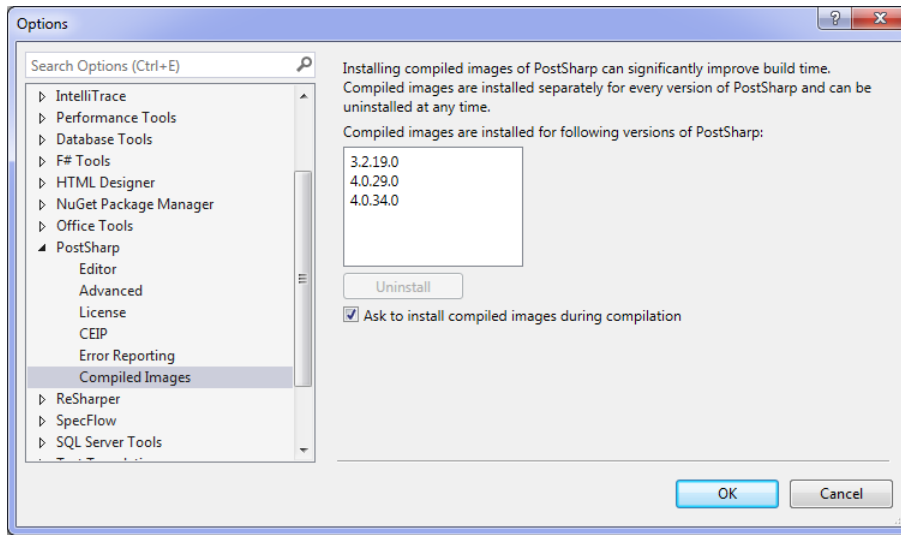
Before you uninstall PostSharp Tools for Visual Studio, we suggest you remove the compiled images.

Removing Native Compiled Images

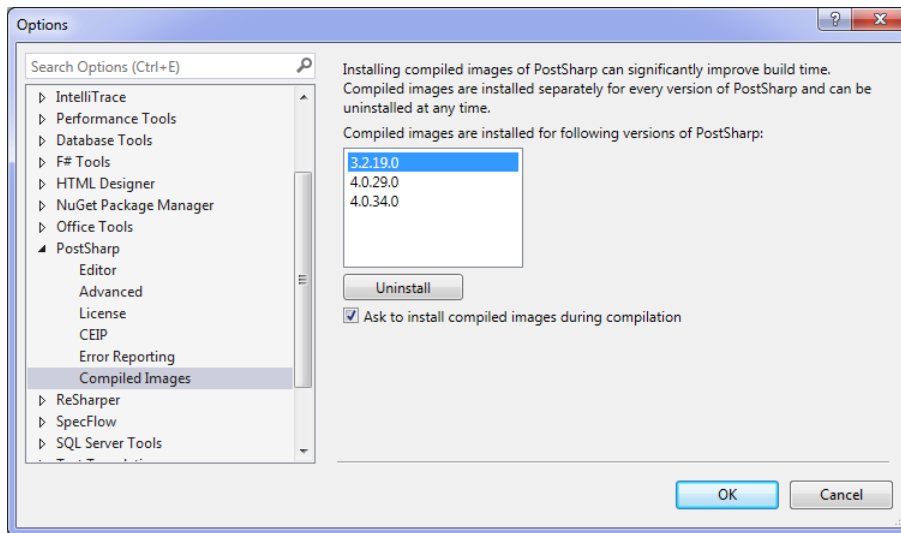
1. Open the Visual Studio Options dialog.



- Expand the PostSharp node in the tree and select the Compiled Images node.



- Select an entry in the list box and click **Uninstall**.



- Follow the wizard to uninstall the compiled images for the selection you made. There are no choices to be made, simply click Next and Finish until the wizard is completed.

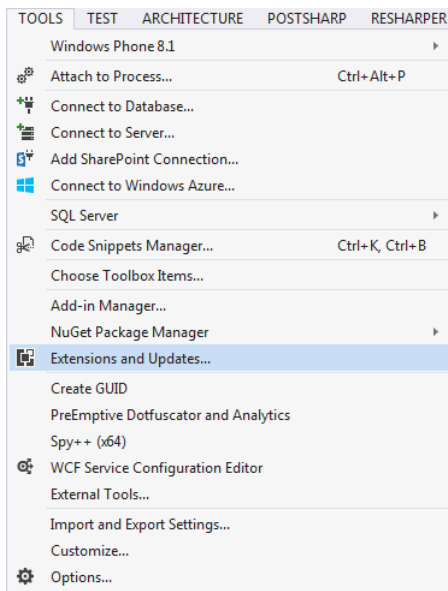
NOTE

You will need to perform the previous steps for each of the versions listed in the PostSharp Compiled Image page in the Options dialog.

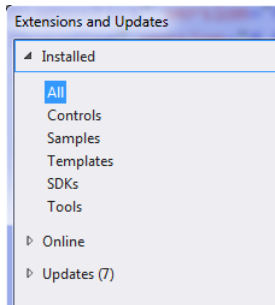
The next step is to remove PostSharp from Visual Studio.

Uninstalling the PostSharp Tools for Visual Studio

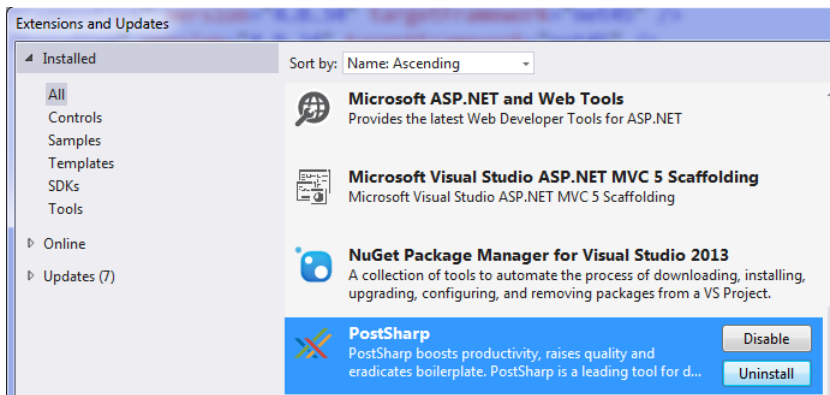
1. Open the Extensions and Updates window.



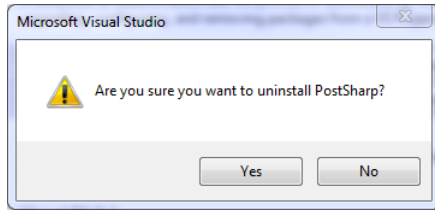
2. Select the All tab on the left of the window.



3. Find the **PostSharp** entry, select it and click the **Uninstall** button.



- Click **Yes** to confirm that you want to uninstall the PostSharp extension.



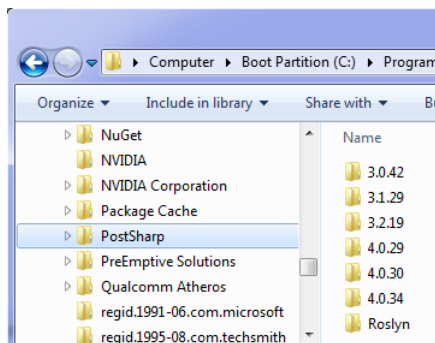
- Repeat the two previous steps and uninstall the extension named **PostSharp Backward Compatibility Tools**.
- Restart Visual Studio.

Finally, you can remove the temporary files created by PostSharp. These files would be recreated as necessary the next time you run PostSharp.

Other than occupying disk space, there is no impact of not removing these files.

Cleaning temporary files

- Open Windows Explorer and navigate to `C:\ProgramData\PostSharp`.



- Select the `C:\ProgramData\PostSharp` folder and delete it.

6.10. Deploying PostSharp to End-User Devices

Although PostSharp is principally a compiler technology, it contains run-time libraries that need to be deployed along with your application to end-user devices.

This topic contains the following sections:

- Redistribution of PostSharp run-time libraries without NuGet
- Redistribution of PostSharp run-time libraries using NuGet

Redistribution of PostSharp run-time libraries without NuGet

If you do not deliver your final product using NuGet, you can bundle all the referenced PostSharp run-time libraries in your product. These libraries are the ones included in the `lib` subdirectory of the NuGet packages and the ZIP distribution.

These run-time libraries can be distributed to end-users free of charge. However, the build-time parts of PostSharp cannot be redistributed under the terms of the standard license agreement.

Besides including these run-time libraries, no other action or configuration is required.

Redistribution of PostSharp run-time libraries using NuGet

If you deliver your final product using NuGet, you can add PostSharp redistributable NuGet packages as a dependency to your NuGet package instead of including all the run-time libraries inside your NuGet package.

PostSharp NuGet packages are either intended for build-time or for run-time. We do not mix both in any of our NuGet packages. See [PostSharp Components on page 57](#) for a description of how to distinguish them.

Thanks to this fact, you only need to set the run-time PostSharp NuGet packages as a dependency of your NuGet package.

Sample NuGet package

In the following example, there is a NuGet package specification of a package which has the PostSharp.Redist package as its dependency. This way, there's no need to include the PostSharp run-time libraries inside the package.

```
<?xmlversion="1.0"encoding="utf-8"?><packagexmlns="http://schemas.microsoft.com/packaging/2013/05/nuspec.xsd"><metadata><i
```

See [.nuspec reference](#)¹⁷ for details.

Using the Pack MSBuid target or .NET Core CLI

In the following example, you see a .NET Core project which uses the PackageReference package management format. Creating a NuGet package using the *Pack* MSBuid target or .NET Core CLI command `dotnet pack` from this project will create a NuGet package which will depend on the run-time PostSharp packages only.

```
<ProjectSdk="Microsoft.NET.Sdk"><PropertyGroup><OutputType>Exe</OutputType><TargetFramework>netcoreapp1.1</TargetFramework  
    will not be a dependency of the NuGet package created from this project. --><PrivateAssets>All</PrivateAs  
    will be a dependency of the NuGet package created from this project. --></ItemGroup></Project>
```

See [Package references \(PackageReference\) in project files](#)¹⁸ for details.

17. <https://docs.microsoft.com/en-us/nuget/schema/nuspec#dependencies>

18. <https://docs.microsoft.com/en-us/nuget/consume-packages/package-references-in-project-files>

CHAPTER 7

Licensing

This chapter addresses some technical questions related to the licensing of PostSharp. For business questions, please refer to our [web site](#)¹⁹.

It contains the following topics:

Chapter	Description
Deploying License Keys on page 75	This topic shows how each developer can install the license key using the UI and how this can be done centrally by adding the license key to source control.
License Audit on page 79	This topic explains that PostSharp does not validate the license key in a blocking way, but audits the use of license keys.
Limitations of PostSharp Essentials on page 79	This topic explains the limitations of PostSharp Essentials.
Sharing Source Code With Unlicensed Teams on page 80	This topic explains that you don't need to purchase a license if you need to build code that you didn't write yourself but uses PostSharp.
Using PostSharp License Server on page 85	This topic shows how users can obtain a license from a license server.
Installing and Servicing PostSharp License Server on page 81	This topic describes how system administrators can install and maintain the license server.

7.1. Deploying License Keys

This section explains how to install PostSharp license keys.

Whether you are using a free or commercial edition, PostSharp requires you to enter a license key before being able to build a project.

This topic contains the following sections:

- [Registering a license key using the user interface on page 75](#)
- [Installing the license key in your source control on page 78](#)

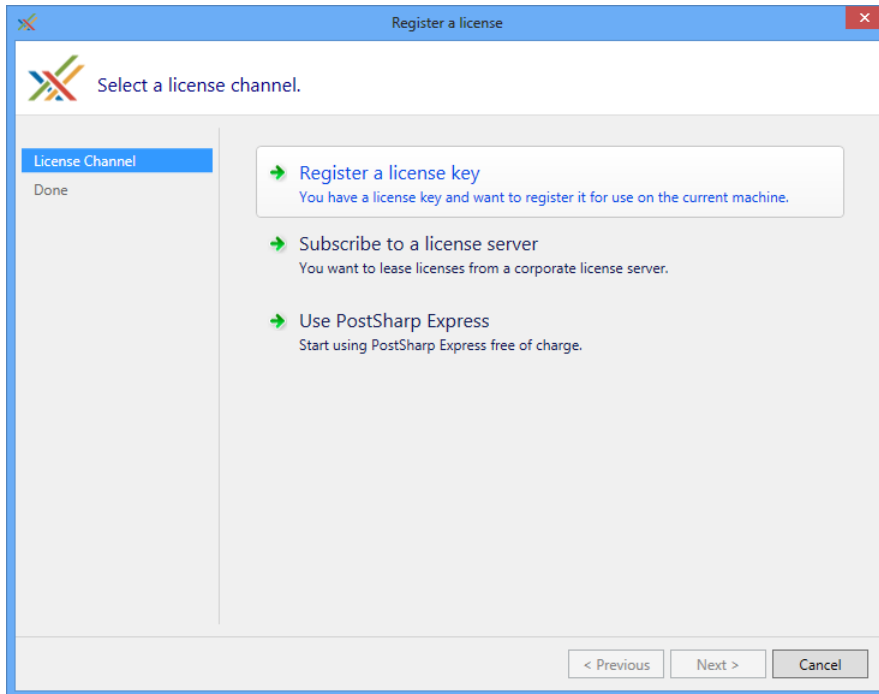
Registering a license key using the user interface

Registering a license key using the user interface is the preferred procedure for individual developers and small teams.

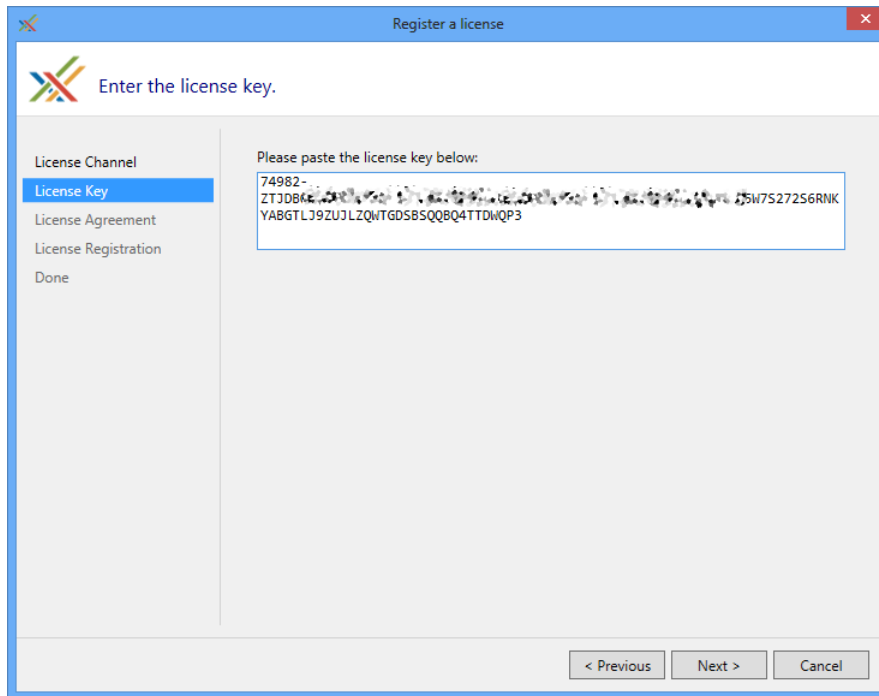
¹⁹ <https://www.postsharp.net/purchase/faq>

To register a license key using the user interface:

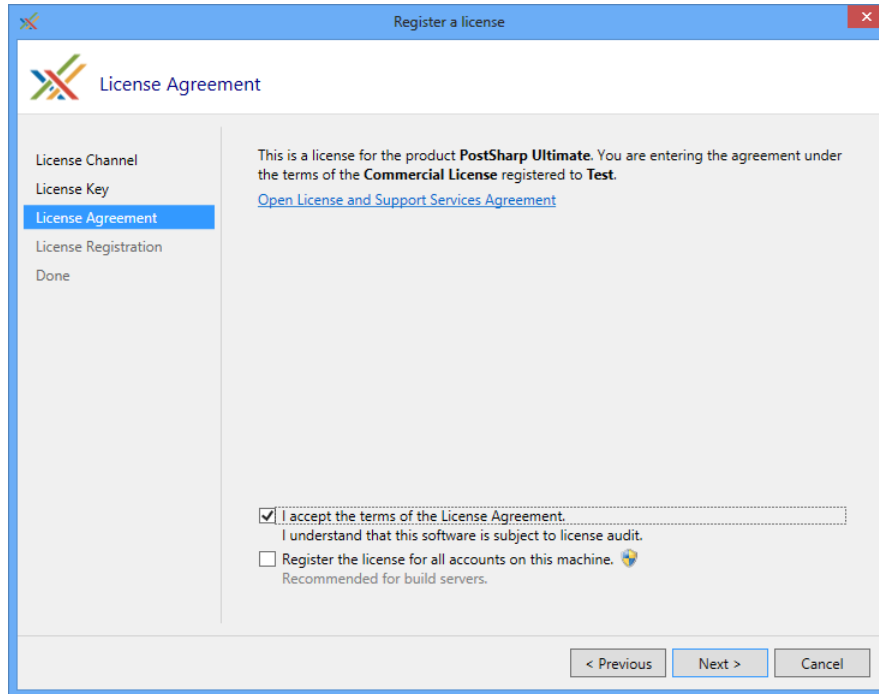
1. Open Visual Studio.
2. Click on menu **PostSharp**, then **Options**.
3. Open the **License** option page.
4. Click on the **Register a license** link.
5. Click on **Register a license**.



6. Paste the license key and click **Next**.



7. Read the license agreement and check the option **I agree**. Click on **Next**.



TIP

If you are registering the license key on a build server, also check the option **Register these settings for all accounts on this machine**.

8. Click **Next** on the notice regarding license metering.

Installing the license key in your source control

It is possible to install the license key in your source control, so that these settings are automatically applied during the build.

To install the license the in source control:

1. Create a file named *postsharp.config* in the root directory of your source repository, or in any parent directory of the Visual Studio project file (*.csproj or *.vbproj).
2. Add the following content to the *postsharp.config* file:

```
<?xmlversion="1.0"encoding="utf-8"?><Projectxmlns="http://schemas.postsharp.org/1.0/configuration"><LicenseValue-
```

In this code, *000-AAAAAAAAAAAAAAAA* must be replaced by the license key or the URL to the license server.

See [Working with PostSharp Configuration Files](#) on page 97 for details about this configuration file.

7.2. License Audit

Although most software packages are protected with a license activation mechanism, we think that the practice is not adequate for software development tools:

- The source code is sometimes compiled several years after it has been written, and there is no guarantee that the license activation server will still be functional.
- Development teams want their tools to be included in the source control repository together with the source code, and want the license key to be deployed the same way.

Instead of license activation, PostSharp relies on asynchronous, fail-safe license audit. PostSharp audits the use of license keys on each client machine and periodically reports it to our license servers. The mechanism does not require a permanent network connection, and PostSharp will not fail if the license server is not available.

The licensing client will contact our licensing servers in the following cases:

- When a license is registered on a computer with the user interface.
- Once per week, for every user and every device using PostSharp.

No personally identifiable information is transmitted during this process except the license key. In case we suspect a rough violation of the License Agreement, we reserve the right to contact the legitimate owner of this license.

TIP

If license audit is not acceptable in your company, please contact us with a request to disable license audit. Our sales teams will evaluate your request and answer with a license key containing an audit waiver. Global licenses and site licenses are not subject to license audit by default. The use of the license server does not implicitly disable license audit. For more information, see [Using PostSharp License Server on page 85](#).

7.3. Limitations of PostSharp Essentials

PostSharp Essentials contains all the features of PostSharp Ultimate, but the number of types to which you can apply aspects is limited to 10 per project or 50 per solution.

To know how many types are already using aspects, open the **PostSharp Metrics** tool window in Visual Studio.

This topic contains the following sections:

- [Limitations of PostSharp Architecture Framework on page 79](#)
- [Limitations of PostSharp Logging on page 80](#)
- [Enforcement of the solution-level limit on page 80](#)
- [Diagnosing licensing issues on page 80](#)

Limitations of PostSharp Architecture Framework

PostSharp Architecture Framework has no concept of aspect and no concept of aspect target, therefore the number of types is computed differently. Instead, what is limited is the number of types for which you can call APIs like `ReflectionSearch` or `ISyntaxReflectionService`.

Limitations of PostSharp Logging

There is no limit on the number of types to which you can apply aspects of the `PostSharp.Patterns.Diagnostics` namespace. However, when using PostSharp Essentials, your application will emit log records only one day after it has been built. After this period, your application will still work normally, but log records will no longer be emitted.

See [Licensing of PostSharp Logging on page 185](#) for details.

Enforcement of the solution-level limit

The limitation of 50 types per solution is implemented not by looking at the `sln` file, but by counting the number of classes in all assemblies that are referenced by the current assembly. That is, the limit is actually 50 types in the whole assembly closure.

Diagnosing licensing issues

If you don't understand why PostSharp is requiring a commercial license, you can generate a licensing diagnostic log by building your project with the following command line:

```
msbuild /t:Rebuild /v:detailed /p:PostSharpTrace=Licensing > msbuild.log
```

7.4. Sharing Source Code With Unlicensed Teams

You only need a license if you *create or modify* code using PostSharp. If you only *build* code that is using PostSharp, you don't need to purchase a license.

PostSharp can determine whether you modify the code or just build it by looking at your source control repository. If your working copy has modifications against your base commit in your source control repository, PostSharp will consider that you are creating or modifying the code yourself, and will require a valid license.

By default, checking the modifications in the source control is disabled for performance reasons. It means that by default PostSharp always requires a valid license during the build. You can enable source control checking by editing the *PostSharp Configuration File* for your project or solution (see [Working with PostSharp Configuration Files on page 97](#)).

To enable source code sharing with unlicensed teams:

1. Open the file `postsharp.config` that is located in the root directory of your solution or project. If the file doesn't exist then create a new `postsharp.config` file in that location with the following content:

```
<?xmlVersion="1.0"encoding="utf-8"?><Projectxmlns="http://schemas.postsharp.org/1.0/configuration"></Project>
```

2. Add a Property element under the Project element, set the Name attribute to `VcsCheckEnabled` and the Value attribute to `True`.

```
<?xmlVersion="1.0"encoding="utf-8"?><Projectxmlns="http://schemas.postsharp.org/1.0/configuration"><PropertyName="Vc
```

NOTE

The following source control systems are supported: **Git** and **TFS**. If you are not using any of these systems, PostSharp will always require a license at build time.

CAUTION NOTE

When building unmodified source code without a commercial license, PostSharp Logging Developer Edition and limitations to PostSharp Logging features will apply to your build. See [Licensing of PostSharp Logging on page 185](#) for details.

7.5. Installing and Servicing PostSharp License Server

This topic covers PostSharp License Server from the point of view of the system administrator and license administrator.

We designed our license server to help our customers, not to enforce our license agreements. The application is open-source, it uses a clear SQL database and provides the ability to workaround issues by canceling leases or purging tables.

This topic contains the following sections:

- [System Requirements on page 81](#)
- [Installing PostSharp License Server on page 81](#)
- [Installing a license key on page 82](#)
- [Testing the license server on page 83](#)
- [Displaying license usage on page 83](#)
- [Canceling leases on page 84](#)
- [Maintenance on page 84](#)

System Requirements

PostSharp License Server is an ASP.NET 4.5 application backed by a Microsoft SQL database.

It requires:

- Windows Server 2003 or later with Internet Information Services installed.
- Microsoft SQL Server 2005 or later (any edition, including the Express edition).

If the license server can be configured with a sufficiently long lease renewal period, there is no need to deploy the application and its database in high-availability conditions. You need to plan that the amount of time between the lease renewal and the lease end is larger than the longest expected outage. Frequent backups are not critical unless usage information is required for accounting purposes in case of pay-as-you-use licenses, where the **Export Logs** feature is required.

CAUTION NOTE

Deploying the ASP.NET application to several machines is not supported because the lease algorithm uses application locking instead of database locking.

Installing PostSharp License Server

The setup procedure is simple but must be performed manually.

To install PostSharp License Server, you will need administrative access on a Windows Server machine and the permission to create a new database.

To install PostSharp License Server:

1. Download the latest version of PostSharp License Server from [GitHub](#)²⁰.

NOTE

It is not necessary that the version of PostSharp License Server exactly matches the version of PostSharp. You do not have to upgrade PostSharp License Server as long as the license keys you want to use are compatible with the server version.

2. Using Internet Information Services (IIS) Manager, configure a new web application whose root is the folder containing the file *web.config*.
3. The application pool should be configured to use ASP.NET 4.5.
4. Configure the authentication mode of the web application: disable **Anonymous Authentication**, and enable **Windows Authentication**.
5. Create an MS SQL database (the free MS SQL Express server is supported).
6. Execute the script *CreateTables.sql* in the context of this database. The file is located in the zip archive of the license server.
7. Set up the security on this database so that the user account under which the ASP.NET application pool is running (typically NETWORK SERVICE) can access the database.
8. Edit the file *web.config*:
 - a. `configuration/connectionStrings`: Correct the connection string (server name and database name) to match your settings.
 - b. `configuration/system.net/mailSettings`: Set the name of the SMTP server used to send warning emails.
 - c. `configuration/applicationSettings`: this section contains several settings that are documented inside the *web.config* file. The most important settings are the email addresses for shortage notifications and the duration of leases duration and renewal delay.
 - d. `configuration/system.web/authorization`: Set up the security of the whole application to restrict the persons who are allowed to borrow a license. Optional.
 - e. `configuration/location[@path='Admin']/system.web/authorization`: Specify who has access to the administrative interface of the application. Optional.

Installing a license key

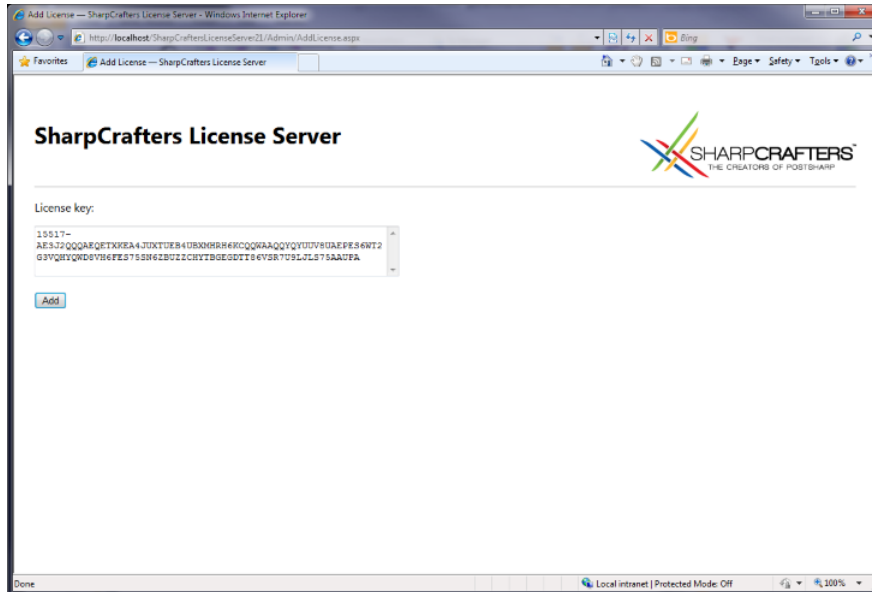
Before developers can start using the license server, you need to install a license key.

To install a license key into the license server:

1. Open the home page of the license server using a web browser.
2. Click on the link **Install a new license**.

20. <https://github.com/postsharp/PostSharp.LicenseServer/releases>

- Paste the license key and click on button **Add**.



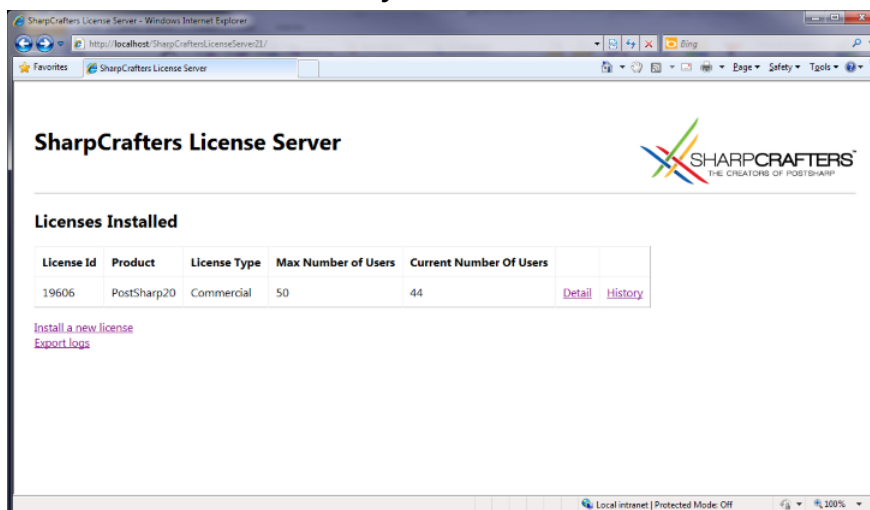
Testing the license server

If you have installed the license server at the address *http://localhost/PostSharpLicenseServer*, you can test it opening the URL *http://localhost/PostSharpLicenseServer/Lease.ashx?product=PostSharp30&user=me&machine=other* using Firefox or Chrome. If the license acquisition was successful, the browser will display the license key and the duration of the lease. You can also then see the resulting allocation on the home page *http://localhost/PostSharpLicenseServer*, and cancel the lease.

Displaying license usage

PostSharp License Server makes it easy to know how many people are currently using the product, and to display historical data.

Overview of installed license keys and current number of leases



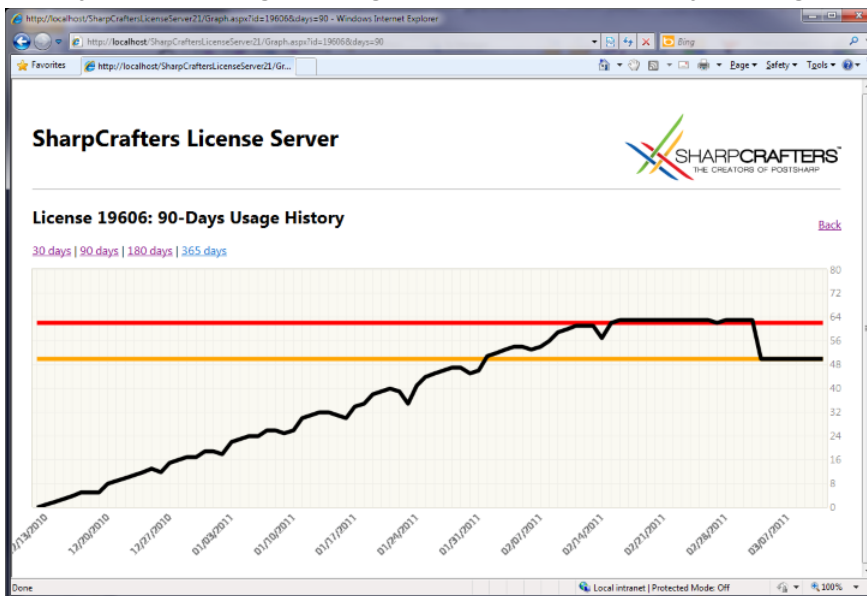
Detail of leases

SharpCrafters License Server

Current leases on license 19606

User	Authenticated User	Machine	Start Time (UTC)	End Time (UTC)	
GARY.TURNER	CONTOSO\GARY.TURNER	DESKTOP-f2a1	2011-01-27 07:00:00	2011-03-13 07:00:00	Cancel
WM.MARTINEZ	CONTOSO\WM.MARTINEZ	DESKTOP-5a2f	2011-02-21 07:00:00	2011-03-14 07:00:00	Cancel
WILLIAM.DAVIS	CONTOSO\WILLIAM.DAVIS	DESKTOP-1f18	2011-02-21 07:00:00	2011-03-14 07:00:00	Cancel
DAVID.RAHM	CONTOSO\DAVID.RAHM	DESKTOP-4b2	2011-03-03 07:00:00	2011-03-14 07:00:00	Cancel
BOBBIE.SHERMAN	CONTOSO\BOBBIE.SHERMAN	DESKTOP-73d3	2011-03-03 07:00:00	2011-03-14 07:00:00	Cancel
JEREMY.THOMPSON	CONTOSO\JEREMY.THOMPSON	DESKTOP-6465	2011-03-03 07:00:00	2011-03-14 07:00:00	Cancel
HUBERT.NUNNALLY	CONTOSO\HUBERT.NUNNALLY	DESKTOP-e995	2011-03-03 07:00:00	2011-03-14 07:00:00	Cancel

History of license usage. The figure demonstrates a 30-day, +30% grace period.



Canceling leases

You can cancel a lease from the lease list by clicking on the **Cancel** hyperlink. Note that canceling a lease on the server does not cancel the lease on the client. It is not possible to cancel leases on the client.

Maintenance

The license server is designed to keep a history of all leases. Therefore, it can grow indefinitely, depending on the number of users and lease duration. It is safe to delete all records from the Lease table at any time unless these records are necessary for accounting purposes.

Using SQL Agent, you can schedule a job that purges old records of the Lease table:

```
DELETEFROM [dbo].[Leases] WHERE EndTime < DATEADD( day, -90, GETDATE() );
```

7.6. Using PostSharp License Server

If the license audit is not acceptable in your company for regulatory or other reasons, you can consider using the PostSharp License Server.

PostSharp License Server is a server application that customers can install into on their own premises to measure the number of concurrent users of PostSharp. The application is based on ASP.NET and Microsoft SQL Server.

This topic contains the following sections:

- [Disclaimer on page 85](#)
- [Design Principles on page 85](#)
- [Subscribing to the license server on page 85](#)
- [Installing the license settings in your source control on page 89](#)

If you are a system administrator or license administrator, see also [Installing and Servicing PostSharp License Server on page 81](#).

Disclaimer

Using the license server is optional. Only a few enterprise customers chose to use it. Alternative approaches are:

- To rely on the default license audit mechanism (see [License Audit on page 79](#) for details).
- To acquire enough licenses for the whole team with some reserve margin.
- To use a spreadsheet to keep track of who is using the software.
- To use other software audit products, although this approach is imperfect because PostSharp is not installed on the developer's machine as a standalone and identifiable application.

Design Principles

The license server manages leases of a license to a given user on a given machine (the "client"). Once the lease is provided to the client, it is cached on the client. Upon client request, the server will return a license key and two dates: the lease end date and the lease renewal date. The lengths of the lease and of the renewal period are configurable. The client will not contact the license server before the renewal date, so the client can go offline during the duration of the lease. Then, a new lease will be reserved for the client. If the client is not able to renew its lease from the server after the lease renewal date, the client will still be able to use PostSharp until the end of the lease. Then, the use of PostSharp will be prevented. To avoid loss of productivity due to lack of network connection or server outages, we recommend setting a large delay between the lease renewal period and the lease period.

If the number of concurrent users exceeds the licensed number, the license administrator will receive an email, and additional users will be allowed during a grace period. At the end of the grace period, only the licensed number of concurrent users will be allowed. The duration of the grace period and the number of excess users depend on the kind of license. By default, it is set to 30% of users and 30 days.

IMPORTANT NOTE

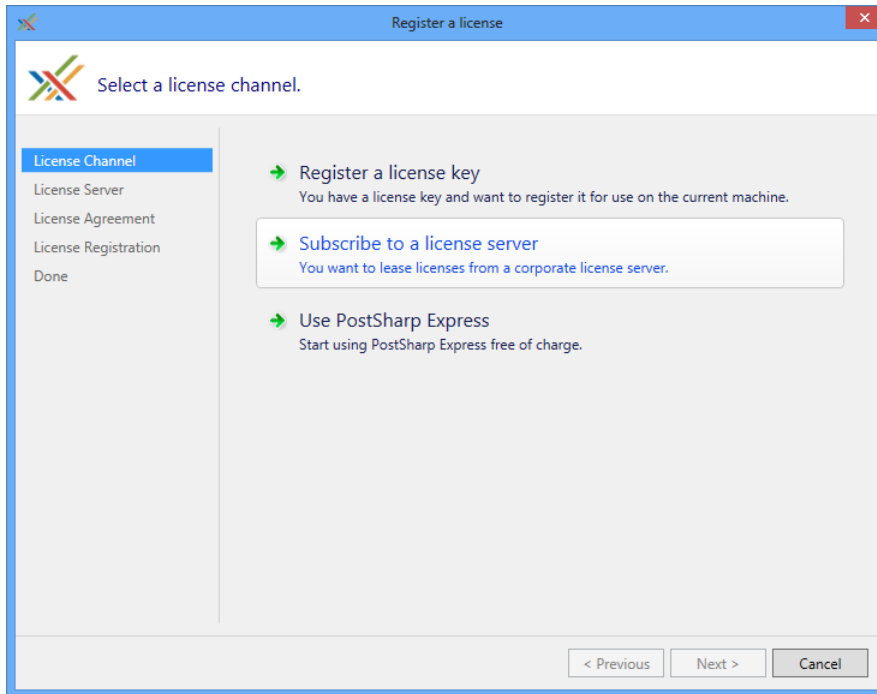
If you have subscribed to a license server, you will need periodic connections to the company network. The licensing client will automatically try to renew a lease when it comes close to expiration and if the license server is available. Lease duration and renewal settings can be configured by the administrator of the license server. A connection to the license server is not necessary while the lease is valid.

Subscribing to the license server

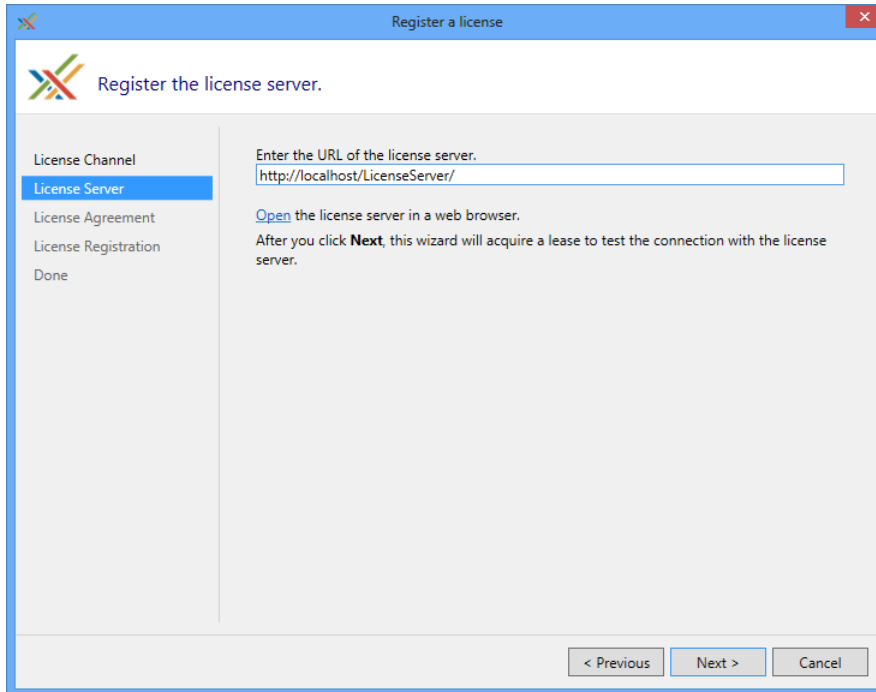
If your company uses PostSharp License Server, you can register using a similar procedure as for registering a license key:

To subscribe to a license server using the user interface:

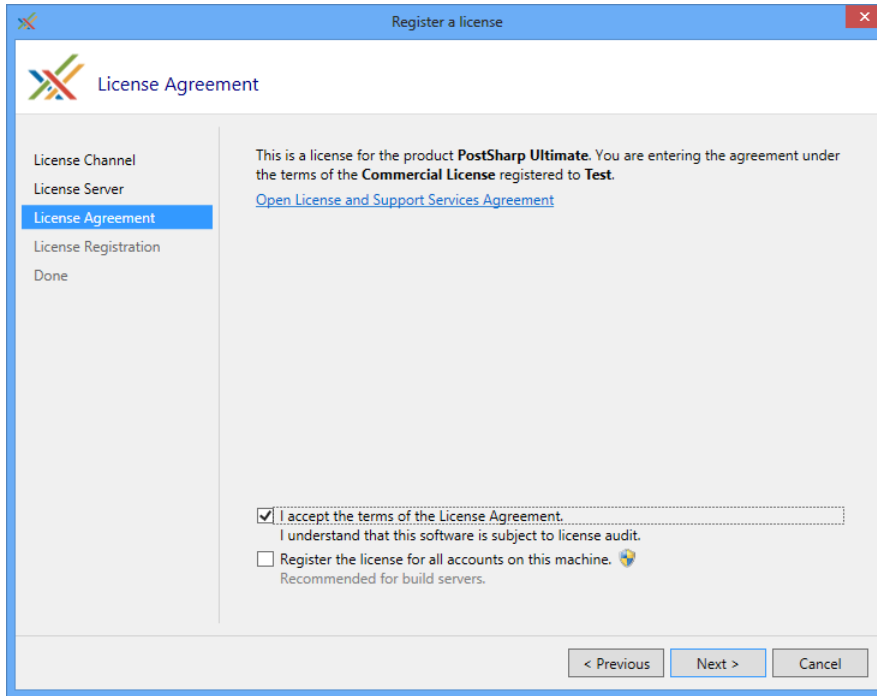
1. Open Visual Studio.
2. Click on menu **PostSharp**, then **Options**.
3. Open the **License** option page.
4. Click on the **Subscribe to a license server** link.



5. Paste the URL of the license server. You can click on the **Open** hyperlink to verify that the URL is correct and that you have access to it. Click **Next**.



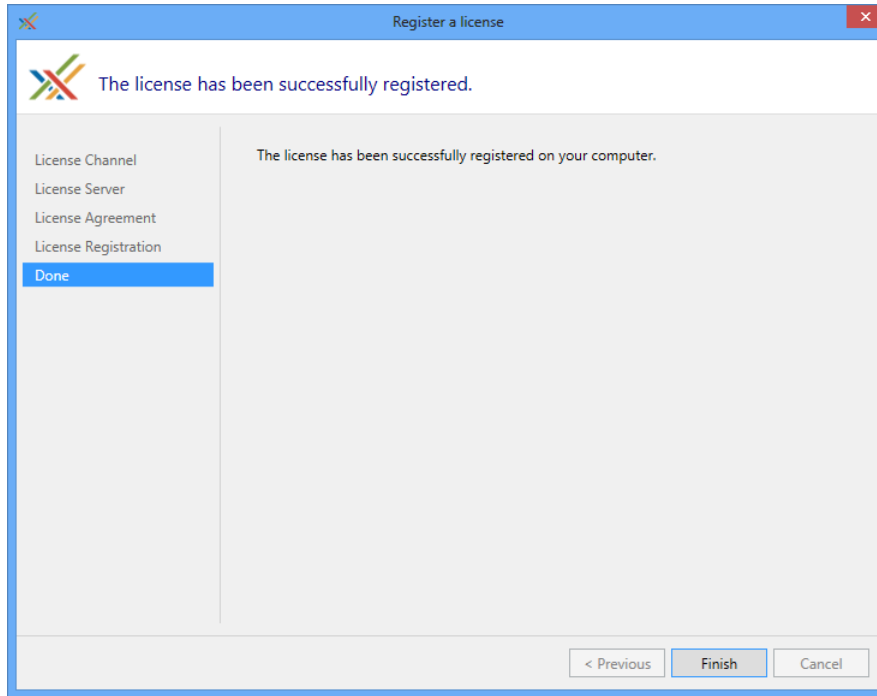
6. Read the license agreement and check the option **I accept**. Click on **Next**.



TIP

If you are registering the license server on a build server, also check the option **Register the license for all accounts on this machine**.

7. You are done.



Installing the license settings in your source control

It is possible to subscribe to the license server using a file stored in your source control system by using the exact same mechanism as to register a license key.

To install license settings in your source control system:

1. Create a file named *postsharp.config* in the root directory of your source repository, or in any parent directory of the Visual Studio project file (*.csproj or *.vbproj).
2. Add the following content to the *postsharp.config* file:

```
<?xmlVersion="1.0"encoding="utf-8"?><Projectxmlns="http://schemas.postsharp.org/1.0/configuration"x:xmlns="http:
```

In this code, *http://server/path* must be replaced by the URL to the license server.

See [Working with PostSharp Configuration Files](#) on page 97 for details about this configuration file.

CHAPTER 8

Configuration

For most use cases, PostSharp does not require any custom configuration. PostSharp gets its default configuration from three sources:

- **MSBuild integration.** PostSharp gets most of its configuration settings directly from the parent MSBuild project.
- **NuGet integration.** Some PostSharp plug-ins delivered as NuGet packages may modify PostSharp configuration files during installation.
- **PostSharp Tools.** When adding aspects and policies from Visual Studio, PostSharp may automatically modify some configuration files.

Even if most configuration settings are correct by default, you may want to understand the configuration system to troubleshoot configuration and installation issues, or simply to implement more advanced configuration scenarios.

PostSharp can be configured using the Visual Studio user interface, by editing MSBuild project files, or by editing PostSharp configuration files.

8.1. Configuring Projects in Visual Studio

PostSharp accepts several configuration settings such as the version and processor architecture of the CLR that is used at build time, the search path of dependencies, and whether some features are enabled.

Although the default configuration is appropriate for most situations, you may have to fine-tune some of them to cope with particular cases.

Most common properties can be edited directly from Visual Studio using the PostSharp project property page.

This topic contains the following sections:

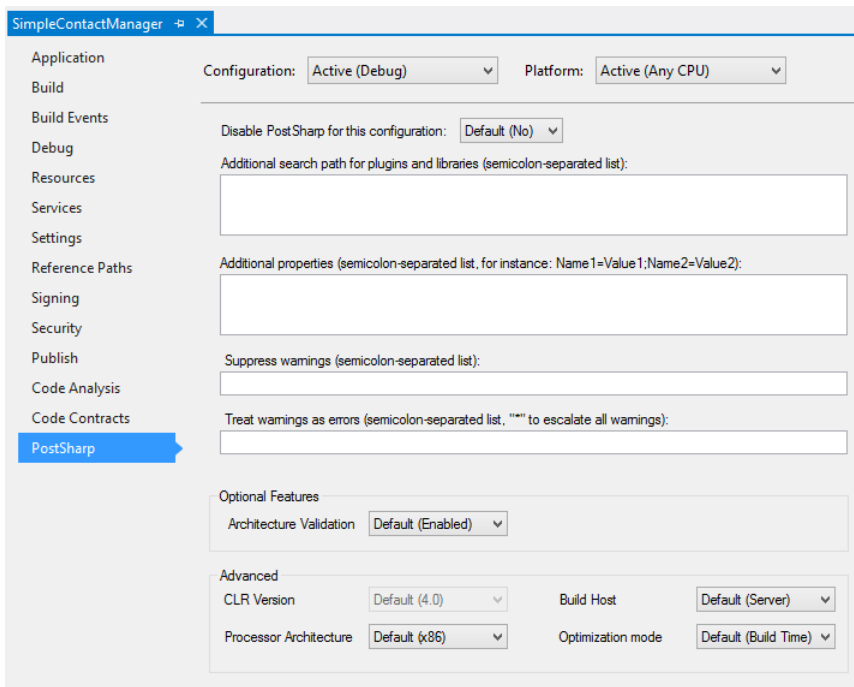
- [Opening PostSharp project properties tab on page 91](#)
- [Understanding configuration settings on page 92](#)

Opening PostSharp project properties tab

To open the PostSharp project property page in Visual Studio

1. Open the **Solution Explorer**.
2. Right-click on the project then select **Properties** at the bottom of the menu.
3. Select the **PostSharp** tab.

The PostSharp property page in the Visual Studio project properties dialog.



Understanding configuration settings

Note that all settings are dependent on the selected project configuration (for instance Debug) and platform (for instance Any CPU).

Setting	Description
Disable PostSharp	True if PostSharp should not execute in for the selected configuration and platform, otherwise False. This setting maps to the MSBuild property SkipPostSharp.
Additional search path	A semicolon-separated list of directories where plug-ins and libraries have to be searched for. This property can reference other MSBuild properties, for instance: <code>..\MyWeaver\bin\\$(Configuration)</code> . All project references are already added to the search path by default. This setting maps to the MSBuild property PostSharpSearchPath.
Additional properties	A semicolon-separated list of property definition, for instance: <code>Name1=Value1;Name2=Value2</code> . This property can reference other MSBuild properties, for instance: <code>RootNamespace=\$(RootNamespace)</code> . Several properties are defined by default; for details, see Well-Known PostSharp Properties on page 101 . This setting maps to the MSBuild property PostSharpProperties
Suppress warnings	A semicolon-separated list of warning identifiers that must be ignored, or <code>*</code> if all warnings have to be ignored. This setting maps to the MSBuild property PostSharpDisabledMessages.
Treat warnings as errors	A semicolon-separated list of warning identifiers that must be escalated into errors, or <code>*</code> if all warnings must be treated as errors. This setting maps to the MSBuild property PostSharpEscalatedMessages.
Architecture Validation	Enabled if constraints must be validated, otherwise Disabled. The default value is Enabled. For details regarding architecture validation, see Validating Architecture on page 435 .
CLR Version	The version of the CLR that hosts PostSharp. PostSharp currently only supports the CLR 4.0 so this setting is disabled. This setting maps to the MSBuild property PostSharpTargetFrameworkVersion.

Setting	Description
Processor Architecture	The processor architecture (x86 or x64) of the process hosting PostSharp. Since PostSharp needs to execute the current project during the build, the processor architecture of the PostSharp process must be compatible with the target platform of the current project. The default value is x86, or x64 if the target platform of the current project is x64. This setting maps to the MSBuild property <code>PostSharpTargetProcessor</code> .
Build Host	The kind of process hosting PostSharp, which influences the assembly loading mechanism, compatibility and performance features. This setting maps to the MSBuild property <code>PostSharpHost</code> . For details, see Configuring Projects Using MSBuild on page 93 .
Optimization Mode	When set to <code>Build Time</code> , PostSharp will use a faster algorithm to generate the final assembly. When set to <code>Size</code> , PostSharp will use a slower algorithm that generates smaller assemblies. The default value is <code>Build Time</code> by default, or <code>Size</code> when the C# or VB compiler is set to generate optimal code (typically, in release builds). This settings maps to the MSBuild property <code>PostSharpOptimizationMode</code> .

8.2. Configuring Projects Using MSBuild

Most configuration settings of PostSharp can be set as MSBuild properties.

NOTE

The integration of PostSharp with MSBuild is implemented in files *PostSharp.tasks* and *PostSharp.targets*. These files define some properties and items that are not documented here. They are considered implementation details and may change without notice.

This topic contains the following sections:

- [Setting MSBuild properties with a text editor on page 93](#)
- [Configuring several projects at a time on page 94](#)
- [Setting MSBuild properties from the command line on page 94](#)
- [List of properties on page 94](#)

Setting MSBuild properties with a text editor

To set a property that persistently applies to a specific project, but not to the whole solution, the best solution is to define it directly inside the C# or VB project file (*.csproj or *.vbproj, respectively) using a text editor.

Adding a project-level MSBuild property using Visual Studio

1. Open the **Solution Explorer**, right-click on the project name, click on **Unload project**, then right-click again on the same project and click on **Edit**.
2. Insert the following XML fragment just *before* the `<Import />` elements:

```
<PropertyGroup><PropertyName>PropertyValue</PropertyName></PropertyGroup>
```

See [Configuring Projects Using MSBuild on page 93](#) for the list of MSBuild properties used by PostSharp.

3. Save the file. If the project was open in Visual Studio, go to the Solution Explorer, right-click on the project name, then click on **Reload project**.

NOTE

For more information regarding MSBuild properties, see [MSDN Documentation](#)²¹.

Configuring several projects at a time

Instead of editing every project file, you can define shared settings in a file named *PostSharp.Custom.targets* and store in the same directory as the project file or in any parent directory of the parent file (up to 7 levels from the project directory).

Files *PostSharp.Custom.targets* are loaded from the root directory to the project directory, so that files that are closer to the project directory are loaded after and override files in parent directories.

Thanks to this mechanism, it is possible to define settings that apply to a large set of projects and control the grain of settings.

Files *PostSharp.Custom.targets* are normal MSBuild project or targets files; they should have the following content:

```
<?xmlVersion="1.0"encoding="utf-8"?><Projectxmlns="http://schemas.microsoft.com/developer/msbuild/2003"><PropertyGroup><Pr
```

See [Configuring Projects Using MSBuild on page 93](#) for the list of MSBuild properties used by PostSharp.

NOTE

For more information regarding MSBuild project files, see [MSDN Documentation](#)²².

Setting MSBuild properties from the command line

When an MSBuild property does not need to be set permanently, it is convenient to set it from the command prompt by appending the flag `/p:PropertyName=PropertyValue` to the command line of **msbuild.exe**, for instance:

```
msbuild.exe /p:PostSharpHost=Native
```

List of properties**General Properties**

The following properties are most commonly overwritten. They can also be edited in Visual Studio using the PostSharp project property page.

Property Name	Description
PostSharpSearchPath	A semicolon-separated list of directories added to the PostSharp search path. PostSharp will probe these directories when looking for an assembly or an add-in. Note that several directories are automatically added to the search path: the .NET Framework reference directory, the directories containing the dependencies of your project and the directories added to the reference path of your project (tab Reference Path in the Visual Studio project properties dialog box).
SkipPostSharp	True if PostSharp should not be executed.

21. <http://msdn.microsoft.com/en-us/library/vstudio/ms171458.aspx>

22. <http://msdn.microsoft.com/en-us/library/vstudio/dd637714.aspx>

Property Name	Description
<code>PostSharpOptimizationMode</code>	When set to <code>OptimizeForBuildTime</code> , PostSharp will use a faster algorithm to generate the final assembly. The other possible value is <code>OptimizeForSize</code> . The default value of the <code>PostSharpOptimizationMode</code> property is <code>OptimizeForBuildTime</code> by default and <code>OptimizeForSize</code> when the C# or VB compiler is set to generate optimal code (typically, in release builds).
<code>PostSharpDisabledMessages</code>	Comma-separated list of warnings and messages that should be ignored.
<code>PostSharpEscalatedMessages</code>	Comma-separated list of warnings that should be escalated to errors. Use * to escalate all warnings.
<code>PostSharpLicense</code>	License key or URL of the license server.
<code>PostSharpProperties</code>	Additional properties passed to the PostSharp project, in format <code>Name1=Value1;Name2=Value2</code> . See Well-Known PostSharp Properties on page 101 .
<code>PostSharpConstraintVerificationEnabled</code>	Determines whether verification of architecture constraints is enabled. The default value is <code>True</code> .

Hosting Properties

Because PostSharp not only reads but also executes the assemblies it transforms, it must run under the proper version and processor architecture of the CLR. Additionally, for each version and processor architecture. The following properties allow influencing the choice of the PostSharp host process.

Property Name	Description
<code>PostSharpTargetFrameworkVersion</code>	Version of the CLR that hosts the PostSharp process. The only valid value is 4.0.
<code>PostSharpTargetProcessor</code>	Processor architecture of the PostSharp hosting process. Valid values are <code>x86</code> and <code>x64</code> . Since PostSharp needs to execute the current project during the build, the processor architecture of the PostSharp process must be compatible with the target platform of the current project. The default value is <code>x86</code> , or <code>x64</code> if the target platform of the current project is <code>x64</code> .
<code>PostSharpHost</code>	Kind of process hosting PostSharp. This setting is usually changed for troubleshooting only. The following values are supported: <ul style="list-style-type: none"> <code>PipeServer</code> means that PostSharp will run as a background process invoked synchronously from MSBuild. Using the pipe server results in lower build time, since PostSharp would otherwise have to be started every time a project is built. The pipe server uses native code and the CLR Hosting API to control the way assemblies are loaded in application domains; the assembly loading algorithm is generally more accurate and predictable than with the managed host. <code>Native</code> uses the same native code as the pipe server, but the process runs synchronously and terminates immediately after an assembly has been processed. For this reason, it does not have the same build-time performance as the pipe server, but it has exactly the same assembly loading algorithm and is useful for diagnostics. <code>Managed</code> is a purely managed application. The assembly loading algorithm may be less reliable in some situations because PostSharp has less control over it. Note that this host is no longer being tested and officially supported.

Property Name	Description
PostSharpBuild	Build configuration of PostSharp. Valid values are ReLease, DiAg and Debug. Only the ReLease build is distributed in the normal PostSharp packages.
PostSharpHostConfigurationFile	A semicolon-separated list of configuration files containing assembly binding redirections that should be taken into account by the PostSharp hosting process, such as <code>app.config</code> or <code>web.config</code> .

Diagnostic Properties

Property Name	Description
PostSharpAttachDebugger	If this property is set to True, PostSharp will break before starting execution, allowing you to attach a debugger to the PostSharp process. The default value is False. For details, see Debugging Build-Time Logic on page 465 .
PostSharpTrace	A semicolon-separated list of trace categories to be enabled. The property is effective only when PostSharp runs in diagnostic mode (see property <code>PostSharpBuild</code> here above). Additionally, the MSBuild verbosity should be set to detailed at least. For details, see Debugging Build-Time Logic on page 465 .
PostSharpUpdateCheckDisabled	True if the periodic update check mechanism should be disabled, False otherwise.
PostSharpExpectedMessages	A semicolon-separated list of codes of expected messages. PostSharp will return a failure code if any expected message was not emitted. This property is used in unit tests of aspects, to ensure that the application of an aspect results in the expected error message.
PostSharpIgnoreError	If this property is set to True, the PostSharp MSBuild task will succeed even if PostSharp returns an error code, allowing the build process to continue. The project or targets file can check the value of the <code>ExitCode</code> output property of the PostSharp MSBuild task to take action.
PostSharpFailOnUnexpectedMessage	This property should be used jointly with <code>PostSharpExpectedMessages</code> . If it set to True, PostSharp will fail if any unexpected message was emitted, even if this message was not an error. This property is used in unit tests of aspects, to ensure that the application of an aspect did not result in other messages than expected.

Other Properties

Property Name	Description
PostSharpProject	Location of the PostSharp project (<code>*.psproj</code>) to be executed by PostSharp, or the string None to specify that PostSharp should not be invoked. If this property is defined, the standard detection mechanism based on references to the <code>PostSharp.dll</code> is disabled.
PostSharpUseHardLink	Use hard links instead of file copies when creating the snapshot for Visual Studio Code Analysis (FxCop). This property is True by default.
ExecuteCodeAnalysisOnPostSharpOutput	When set to True, executes Microsoft Code Analysis on the <i>output</i> of PostSharp. By default, the analysis is done on the <i>input</i> of PostSharp, i.e. on the output of the compiler. This property has no effect when Microsoft Code Analysis is disabled for the current build.

Property Name	Description
PostSharpCopyCodeAnalysisDependenciesDisabled	When set to True, PostSharp will not copy the all dependencies of the current project output into the <i>obj\Debug\Before-PostSharp</i> directory, which contains the copy of the assembly on which Microsoft Code Analysis is executed by default. This property has no effect when Microsoft Code Analysis is disabled for the current build or when the <code>ExecuteCodeAnalysisOnPostSharpOutput</code> property has been set to True.

8.3. Working with PostSharp Configuration Files

PostSharp is designed as a modular post-compilation platform, whose functionality can be extended using plug-ins. For instance, the Diagnostics Pattern Library is implemented as a plug-in. Although writing custom plug-ins is out of the scope of this documentation, you should be able, as a PostSharp user, to understand how plug-ins can be added to a project and how they can be configured.

This topic contains the following sections:

- [PostSharp configuration files on page 97](#)
- [Sharing configuration between projects on page 97](#)
- [Order of processing of configuration files on page 98](#)

PostSharp configuration files

The configuration system of PostSharp is based on configuration files.

By default, if you have a project named `MyProject.csproj`, PostSharp will try to load, from the same directory, a configuration file, named `MyProject.psproj`. Configuration file is optional. Most projects don't need it.

See [Configuration File Schema Reference on page 98](#) for details about the format of this file.

For instance, the following code is the configuration file of a project using two plug-ins:

```
<Project xmlns="http://schemas.postsharp.org/1.0/configuration"><PropertyName="LoggingBackend"Value="nlog"/><UsingFile="..\
```

The principal use cases for end-users are the following:

- Adding license keys. See the [License on page 98](#) configuration element for details.
- Configuring properties. See the [Property on page 98](#) configuration element for details.
- Including a plug-in. See the [Using on page 98](#) configuration element for details.
- Adding aspects or constraints without modifying your source code. See [Adding Aspects Using XML on page 129](#) for details.
- Editing logging profiles. See [Configuration File Schema Reference on page 98](#) and **[logging-profiles]** for details.

Sharing configuration between projects

You will often want to share some configuration settings between several projects. A typical example is to add the license key to all projects of your source code repository.

This can be achieved by adding a well-known configuration file to your source tree, or thanks to the [Using on page 98](#) configuration element.

Well-known configuration files

PostSharp will automatically load a few well-known configuration files if they are present on the file system, in the following order:

1. Any file named `postsharp.config` located in the directory containing the MSBuild project file (`csproj` or `vbproj`, typically), or in any parent directory, up to the root. These files are loaded in ascending order, i.e. up from the root directory to the project directory.
2. Any file named `MySolution.pssln` located in the same directory as the solution file `MySolution.sln`.
3. Any file named `MyProject.psproj` located in the same directory as the project file `MyProject.csproj` or `MyProject.vbproj`.

For instance, the files may be loaded in the following order:

1. `c:\src\BlueGray\postsharp.config`
2. `c:\src\BlueGray\FrontEnd\postsharp.config`
3. `c:\src\BlueGray\FrontEnd\BlueGray.FrontEnd.Web\postsharp.config`
4. `c:\src\BlueGray\Solutions\BlueGray.pssln` assuming that the current solution file is `c:\src\BlueGray\Solutions\BlueGray.sln`.
5. `c:\src\BlueGray\FrontEnd\BlueGray.FrontEnd.Web\BlueGray.FrontEnd.Web.psproj` assuming that the current project file is `c:\src\BlueGray\Solutions\BlueGray.sln`.

Explicit configuration sharing

The second technique to share a configuration file among several projects is to use the [Using on page 98](#) configuration element to import a configuration file into another configuration file.

Order of processing of configuration files

Elements of configuration files are processed in the following order:

1. [License on page 98](#) elements are loaded.
2. [Property on page 98](#) elements are loaded. Properties are evaluated at this moment unless they are marked for deferred evaluation.
3. [SearchPath on page 98](#) elements are loaded.
4. [Using on page 98](#) elements are loaded and referenced plug-ins and configuration files are immediately loaded.
5. [SectionType on page 98](#) elements are loaded.
6. [Service on page 98](#) elements are loaded, but they are not yet instantiated.
7. Extension elements are loaded, but they are not evaluated at this moment.
8. Finally, services and other tasks are instantiated and the project is executed.

8.3.1. Configuration File Schema Reference

The basic format of a PostSharp configuration file is as follows:

```
<Project xmlns="http://schemas.postsharp.org/1.0/configuration" xmlns:x="http://schemas.postsharp.org/1.0/configuration"><!-- Th
  PostSharp itself defines the following extension elements: --><Multicast><MyMulticastAspectMyAttributeName="<value>"xml
```

Schema elements

The configuration file includes these elements, described in detail in subsequent sections in this topic:

[Project on page 99](#)

[License on page 99](#)

[SearchPath on page 99](#)

[Using on page 99](#)

[SectionType on page 100](#)

[Property on page 100](#)

[Service on page 100](#)

[Multicast on page 0](#)

Project

This element is the root of the configuration file.

License

This element allows loading one or more license keys.

Attribute	Type	Description
Value	string	Required. A semicolon-separated list of license keys, or an URL to the license server.

SearchPath

This element adds a file or a directory to the list of paths in which PostSharp searches for assemblies and plug-ins.

Attribute	Type	Description
Path	string expression	Required. A semicolon-separated list of files or directories that must be added to the path.
PathKind	File or Directory	Optional. Specifies the kind of items contained in the Path property. If this property is not specified, the item kind is automatically determined for each individual item of the Path property.
ReferenceDirectory	string expression	Optional. The directory from which relative paths in the Path property.
Condition	boolean expression	Optional. true if the element is considered, false if it must be ignored.

Using

This element imports another configuration file into the current project.

Attribute	Type	Description
File	string expression	Required. Name of the file to be imported. Unless the name is qualified by a relative or absolute path, the file will be searched for using the search path. In this case, a file with extension <i>psplugin</i> or <i>dll</i> will be searched.
ProjectName	string	Optional. In case that a single <i>dll</i> includes several configurations, specifies which configuration should be loaded.

Attribute	Type	Description
Condition	boolean expression	Optional. true if the element is considered, false if it must be ignored.

SectionType

This element defines custom sections for the current project. Custom sections can appear under the Project element under all system-defined elements.

Attribute	Type	Description
LocalName	string	Required. The local name of the XML element representing the custom section.
Namespace	string	Required. The namespace of the XML element representing the custom section.

Property

This element defines a property for the current project.

Attribute	Type	Description
Name	string	Required. The property name.
Value	string expression	Required. The value that is assigned to the property.
Overwrite	boolean	Optional. true if the element will overwrite any previously-defined property of the same name, otherwise false. The default value is true.
Sealed	boolean	Optional. true if an attempt to overwrite this property should result in an error, otherwise false. The default value is false.
Deferred	boolean	Optional. true if the expression in the Value attribute should be dynamically evaluated every time the property value is requested, or false if the expression should be set at the time the property is defined. The default value is false.
Condition	boolean expression	Optional. true if the element is considered, false if it must be ignored.

Service

This element registers a service to the service locator for the current project.

Attribute	Type	Description
TypeName	string	Required. The full type name implementing the service. This class must have a public parameterless constructor and implement the <code>IService</code> interface.
AssemblyFile	string expression	Optional. The path of the assembly defining the service class. If the attribute is not provided, the type will be searched for in the assembly being currently processed by PostSharp.
Condition	boolean expression	Optional. true if the element is considered, false if it must be ignored.

Multicast

This element can be used to add aspects, policies or constraints to a project without adding the C# project as custom attributes. Adding elements to this section is equivalent to adding them to source code at assembly level.

The Multicast section is convenient to add aspect to several projects from a single file.

For details regarding this section, see [Including CLR Objects in Configuration on page 102](#).

8.3.2. Well-Known PostSharp Properties

The following table lists the PostSharp properties that may be set from the MSBuild project. The second column specifies the name of the MSBuild property that influences the value of the PostSharp property, if any.

Property Name	MSBuild Property Name	Description
Configuration	Configuration	Build configuration (typically Debug or Release).
Platform	Platform	Target processor architecture (typically AnyCPU, x86 or x64).
MSBuildProjectFullPath	MSBuildProjectFullPath	Full path of the C# or VB project being built.
IgnoredAssemblies		Comma-separated list of assembly short names (without extension) that should be ignored by the dependency scanning algorithm. Add an assembly to this list if it is obfuscated, or contains native code, and causes PostSharp to fail.
ReferenceDirectory	MSBuildProjectDirectory	Directory with respect to which relative paths are resolved.
SearchPath	PostSharpSearchPath	Comma-separated list of directories containing reference assemblies and plug-ins.
TargetFrameworkIdentifier	TargetFrameworkIdentifier	Identifier of the target framework of the current project (i.e. the framework on which the application will run). For instance .NETFramework or Silverlight.
TargetFrameworkVersion	TargetFrameworkVersion	Version of the target framework of the current project (i.e. the framework on which the application will run). For instance v4.0.
TargetFrameworkProfile	TargetFrameworkProfile	Profile of the target framework of the current project (i.e. the framework on which the application will run). For instance WindowsPhone.
BenchmarkOutputFile	PostSharpBenchmarkOutputFile	When this property is set, PostSharp will append build-time profiling information to a file whose path is set in this property. If the file already exists, PostSharp will append new data to the existing file. PostSharp will lock the file to make sure the option can be used in parallel builds. If the path is a relative path, it will be resolved relatively to the project directory.

Other properties are recognized but are of little interest for end-users. For a complete list of properties, see *PostSharp.targets*.

Using custom properties

By defining your own PostSharp properties, you can pass information from the build environment to aspects, or to any code running in PostSharp. Custom PostSharp properties behave exactly as other PostSharp properties, so they can be defined and read using the same procedures.

8.3.3. Including CLR Objects in Configuration

PostSharp includes a basic facility to describe CLR objects using XML. This facility is used to implement the [Multicast on page 98](#) and [LoggingProfiles on page 98](#) sections of the configuration file, and can be used to define custom sections.

The facility is consciously limited in features. It was only designed to provide the same features as custom attributes in programming languages.

This topic contains the following sections:

- [Basic rules on page 102](#)
- [Formatting of attributes on page 102](#)
- [Specific rules for the Multicast section on page 102](#)

Basic rules

The basic rules apply to XML serialized objects:

- The local name of the XML element must exactly match the type name of the CLR type. An exception to this rule is that the Attribute prefix can be omitted.
- The XML namespace of the element must be in the form `clr-namespace:namespace;assembly:assembly` where *namespace* is the namespace of the CLR type and *assembly* is the name of the assembly declaring the type.
- The type must have a public parameterless constructor.
- Names of XML attributes must exactly match the name of a public field or property of the CLR type.

Formatting of attributes

Values of XML attributes, mapping to CLR fields and properties, must be formatted according to the rule relevant for each type:

Type	Formatting
Intrinsics	An intrinsic is any of the following types: <code>bool</code> , <code>char</code> , <code>sbyte</code> , <code>byte</code> , <code>short</code> , <code>ushort</code> , <code>int</code> , <code>uint</code> , <code>long</code> , <code>ulong</code> , <code>float</code> , <code>double</code> or <code>string</code> . Conversion is done using the <code>Convert</code> class.
Arrays	A semicolon-separated list of elements.
Enumerations	A list of enumeration member names separated by characters <code> </code> or <code>+</code> . Names in the list are combined using the <code>+</code> operator.
Types	An assembly-qualified type name.
Object	Fields and properties of type <code>Object</code> are not supported.

Specific rules for the Multicast section

The following additional rules apply to the [Multicast on page 98](#) section of the configuration file:

- The class must derive from `MulticastAttribute`.
- The `AttributePriority` property may not be defined. This attribute is added automatically according to the order of the XML element in the section.

8.3.4. Using Expressions in Configuration Files

Many attributes of the configuration schema accept expressions, which are dynamically evaluated. Expressions in the PostSharp configuration system work similarly as in XSLT. Substrings enclosed by curled brackets, for instance `{$property}`, are interpreted as XPath expressions.

For instance, the following code contains two XPath expressions:

```
<Projectxmlns="http://schemas.postsharp.org/1.0/configuration"><PropertyName="LoggingEnabled"Value="{has-plugin('PostSharp.Pat
```

Please check the [MSDN documentation](#)²³ for general information about XPath.

NOTE

In the context of PostSharp configuration files, XPath expressions cannot refer to XML elements or attributes, but only to variables, functions, operators and constants.

Accessing properties

PostSharp properties are mapped to XPath variables.

For instance, the expression `{$LoggingEnabled}` evaluates to the value of the *LoggingEnabled* property.

Using operators and functions

You can use any XPath function and operators.

Additionally to [standard XPath 1.0 functions](#)²⁴, PostSharp defines the following functions:

Function	Description
<code>has-plugin(name)</code>	Evaluates to true if the given plug-in is loaded, otherwise false.
<code>environment(variable)</code>	Returns the value of an environment variable.

Mixing expressions and literal strings

An attribute value can contain both text and expressions. This is illustrated in the following example:

```
<Projectxmlns="http://schemas.postsharp.org/1.0/configuration"><PropertyName="A"Value="A"/><!-- Evaluates to A --><Propert
```

8.4. Accessing Configuration from Source Code

Even if most configuration settings are consumed by PostSharp or its add-in, it is sometimes useful to access configuration elements from user code. The *PostSharp.dll* library gives access to both configuration properties and extension configuration elements.

This topic contains the following sections:

- [Accessing properties on page 104](#)
- [Accessing custom sections on page 104](#)

23. [http://msdn.microsoft.com/en-us/library/ms256138\(v=vs.110\).aspx](http://msdn.microsoft.com/en-us/library/ms256138(v=vs.110).aspx)

24. [http://msdn.microsoft.com/en-us/library/ms256138\(v=vs.110\).aspx](http://msdn.microsoft.com/en-us/library/ms256138(v=vs.110).aspx)

Accessing properties

You can read the value of any PostSharp property by including it in an XPath expression and evaluating it using the `EvaluateExpression(String)` method of `PostSharpEnvironmentCurrentProject`:

```
stringValue = PostSharpEnvironment.CurrentProject.EvaluateExpression("${PropertyName}")
```

For details regarding expressions, see [Using Expressions in Configuration Files on page 103](#).

Accessing custom sections

You can get a list of custom sections of a given name and namespace by calling the `GetExtensionElements(String, String)` method of `PostSharpEnvironmentCurrentProject`:

```
IEnumerable<ProjectExtensionElement> elements =
    PostSharpEnvironment.CurrentProject.GetExtensionElements( "MyElement", "uri:MyNamespace" );
```

Extension elements must be declared using the [SectionType on page 97](#) element.

8.5. Working with Errors, Warnings, and Messages

As any compiler, PostSharp can emit messages, warnings, and errors, commonly referred to as *message*. Custom code running at build time (typically the implementation of `CompileTimeValidate` or of a custom constraint) can use PostSharp messaging facility to emit their own messages.

In this section:

- [Ignoring and Escalating Warnings on page 104](#)
- [Emitting Errors, Warnings, and Messages on page 105](#).

TIP

PostSharp 2.1 contains an experimental feature that adds file and line information to errors and warnings. The feature requires Visual Studio. It must be enabled manually in the **PostSharp** tab of Visual Studio options.

8.5.1. Ignoring and Escalating Warnings

As with conventional compilers, warnings emitted by PostSharp, as well as those emitted by custom code running at build time in PostSharp, can be ignored (in that case they will not be displayed) or escalated into errors.

Warnings can be ignored either globally, using a project-wide setting, or locally for a given element of code. Warnings can be escalated only globally.

Ignoring or escalating warnings globally

There are several ways to ignore or escalate a warning for a complete project:

- In Visual Studio, in the **PostSharp** tab of the project properties dialog. See [Configuration on page 91](#) for details.
- By defining the `PostSharpDisabledMessages` or `PostSharpEscalatedMessages` MSBuild properties. See [Configuration on page 91](#) and [Configuring Projects Using MSBuild on page 93](#) for details.

NOTE

The value `*` can be used to escalate all warnings into errors.

Ignoring warnings locally

Most warnings are related to a specific element of code. To disable a specific warning for a specific element of code, add the **IgnoreWarningAttribute** custom attribute to that element of code, or to any enclosing element of code (for instance, adding the attribute to a type will make it effective for all members of this type).

You can create your own custom attribute derived from **IgnoreWarningAttribute** and make it conditional to a compilation symbol by using the `ConditionalAttribute` custom attribute.

8.5.2. Emitting Errors, Warnings, and Messages

Custom code running in PostSharp at build time can use the messaging facility to emit its own messages, warnings, and errors. These messages will appear in the MSBuild output and/or in Visual Studio. User-emitted warnings can be ignored or escalated using the same mechanism as for system messages.

Emitting messages

If you just have a few messages to emit, you may simply use one of the overloads of the `Write` method of the `Message` class.

All messages must have a severity `SeverityType`, a message number (used as a reference when ignoring or escalating messages), and a message text. Additionally, and preferably, messages may have a location (**MessageLocation**).

NOTE

To benefit from the possibility to ignore messages locally, you should always use provide a relevant location with your messages. Previous API overloads, which did not require a message location, are considered obsolete.

TIP

Do not use `string.Format` to format your messages. Instead, pass message arguments to the messaging facility, which will format some argument types, for instance reflection objects, in a more readable way.

Emitting messages using a message source

If you want the text of all messages to be stored in a single location, you have to emit messages through a `MessageSource`. Typically, you would create a singleton instance of `MessageSource` for each component, and associate each instance with a message dispenser. A message dispenser is a custom-written class implementing the `IMessageDispenser` interface. The `MessageDispenser` provides a convenient abstract implementation.

NOTE

Although it is tempting to use a `ResourceManager` as the backend of a message dispenser, comes with a non-negligible performance penalty because of the cost of instantiating the `ResourceManager`.

8.6. Resolution of assembly binding redirections

PostSharp executes your project assemblies at compile time. This is why PostSharp must follow assembly binding redirections at compile time. Although the default assembly redirection mechanism works fine in most cases, there may be situations where you will need to override it.

NOTE

To learn why PostSharp executes your project assemblies at compile time, see [Understanding Aspect Lifetime and Scope on page 389](#).

Default assembly binding redirections

In order to follow the same assembly binding redirections as your application does at run time, PostSharp analyzes your projects and configuration files (typically *app.config* or *web.config*) and generates assembly binding redirection configuration. PostSharp stores the assembly binding redirection in a file named *PostSharpHost.config* and stored in the *obj* folder. You can review the *PostSharpHost.config* file to get an idea of what configuration PostSharp uses to resolve assemblies. For an empty ASP.NET MVC 5 application the *PostSharpHost.config* may look like this

```
<?xmlVersion="1.0"encoding="utf-8"?><configuration><runtime><assemblyBindingxmlns="urn:schemas-microsoft-com:asm.v1"><depe
```

NOTE

We didn't reinvent the wheel. Under the hood, PostSharp relies on the [GenerateBindingRedirects²⁵](#) MSBuild task.

Overriding default assembly redirections

In case the default mechanism does not work, you can disable it by setting the MSBuild property `PostSharpDisableDefaultBindingRedirects` to `True` in your project file:

```
<PropertyGroup><PostSharpDisableDefaultBindingRedirects>True</PostSharpDisableDefaultBindingRedirects></PropertyGroup>
```

With this configuration PostSharp doesn't analyze assembly binding redirections.

CAUTION NOTE

Do not set `PostSharpDisableDefaultBindingRedirects` to `True` unless you really have to. It may produce difficult to predict results.

NOTE

The default algorithm is always disabled for Windows Phone Silverlight projects because it does not work.

If you disable default binding redirections, you may want to specify a file with your own assembly binding redirection configuration

```
<PropertyGroup><PostSharpDisableDefaultBindingRedirects>True</PostSharpDisableDefaultBindingRedirects><PostSharpHostConfig
```

25. <https://msdn.microsoft.com/en-us/library/microsoft.build.tasks.generatebindingredirects.aspx>

With this configuration PostSharp uses explicit assembly binding redirection configuration from the `web.config` file.

PART 3

Adding Aspects to Code

CHAPTER 9

Adding Aspects Declaratively Using Attributes

In .NET, you normally need to write one line of code for any application of a target attribute. If a custom attribute applies to all types in a namespace, you have to manually add the custom attribute to every single type.

By contrast, multicast custom attributes allow you to apply a custom attribute on multiple declarations from a single line of code by using wildcards or regular expressions, or by filtering on some attributes. It makes it easy to apply an aspect to, say, all public static methods of a namespace, with a single line of code.

Multicast attributes can be inherited: you can put an attribute on an interface and ask it to apply to all classes implementing this interface. Attribute inheritance also works for classes, virtual or interface methods, and parameters of virtual or interface methods.

Custom attributes supporting multicasting need to be derived from `MulticastAttribute`. All PostSharp aspects and constraints are derived from this class.

NOTE

Multicasting of custom attribute is a feature of PostSharp. If you do not transform your assembly using PostSharp, multicast attributes will behave as plain old custom attributes.

NOTE

This documentation often refers to this as “*aspect*” multicasting and inheritance. This is not totally accurate. Although this feature has been developed to support aspects, you can use it for your own custom attributes, even if they are not aspects. To use multicasting and inheritance for custom attributes that are not aspects, simply derive the attribute class from `MulticastAttribute` instead of `Attribute`.

Attribute multicasting supports the following scenarios:

- [Adding Aspects to a Single Declaration on page 112](#)
- [Adding Aspects to Multiple Declarations on page 112](#)
- [Adding Aspects to Derived Classes and Methods on page 117](#)
- [Overriding and Removing Aspect Instances on page 123](#)
- [Reflecting Aspect Instances at Runtime on page 126](#)

For a conceptual overview of this feature, see:

- [Understanding Attribute Multicasting on page 114](#)
- [Understanding Aspect Inheritance on page 121](#)

9.1. Adding Aspects to a Single Declaration

Aspects in PostSharp are plain custom attributes. You can apply them to any element of code as usual.

In the following example, the `Trace` aspect is applied to two methods.

```
publicclass CustomerService
{
    [Trace]
    public Custom GetCustomer( int customerId )
    {
        // Details skipped.
    }

    [Trace]
    publicvoid MergeCustomers( Customer customer1, Customer customer2 );
    {
        // Details skipped.
    }
}
```

9.2. Adding Aspects to Multiple Declarations

Having written an aspect we have to apply it to the application code so that it will be used. There are a number of ways to do this so let's take a look at one of them: custom attribute multicasting. Other ways include XML Multicasting (see the section [Adding Aspects Using XML on page 129](#)) and dynamic aspect providers (see more in the section [Adding Aspects Programmatically using IAspectProvider on page 131](#)).

This topic contains the following sections:

- [Applying to all members of a class on page 112](#)
- [Applying an aspect to all types in a namespace on page 113](#)
- [Excluding an aspect from some members on page 113](#)
- [Filtering by class visibility on page 114](#)
- [Filtering by method modifiers on page 114](#)
- [Programmatic filtering on page 114](#)

Applying to all members of a class

When we are trying to apply a method-level aspect, we can place an attribute to each of the methods. As our codebase grows, this approach becomes tedious. We need to remember to add the attribute to all of the methods on the class. If you have hundreds of classes, you may have thousands of methods you need to manually add the aspect attribute to. It is an unsustainable proposition. Thankfully, there is a way to make this easier. Instead of applying your aspect to each method you can add that attribute to the class and PostSharp will ensure that the aspect is applied to all of the methods on that class.

```
[OurLoggingAspect]
publicclass CustomerServices
```

You can also add a location-level aspect to a class which applies it to all properties and all fields in that class. Note that this includes backing fields of auto-implemented properties.

Applying an aspect to all types in a namespace

Even though we don't have to apply an aspect to all methods in all classes in our application, adding the aspect attribute to every class could still be an overwhelming task. If we want to apply our aspect in a broad stroke we can make use of PostSharp's `MulticastAttribute`.

The `MulticastAttribute` is a special attribute that will apply other attributes throughout your codebase. Here's how we would use it.

1. Open the `AssemblyInfo.cs`, or create a new file `GlobalAspects.cs` if you prefer to keep things separate (the name of this file does not matter).
2. Add an `[assembly:]` attribute that references the aspect you want to apply.
3. Add the `AttributeTargetTypes` property to the aspect constructor and define the namespace that you would like the aspect applied to.

```
[assembly: OurLoggingAspect(AttributeTargetTypes="OurCompany.OurApplication.Controllers.*")]
```

This one line of code is the equivalent of adding the aspect attribute to every class in the desired namespace.

NOTE

When setting the `AttributeTargetTypes` you can use wildcards (*) to indicate that all sub-namespaces should have the aspect applied to them. It is also possible to indicate the targets of the aspect using regex. Add "regex:" as a prefix to the pattern you wish to use for matching.

Excluding an aspect from some members

Multicasting an attribute can apply the aspect with a very broad brush. It is possible to use `AttributeExclude` to restrict where the aspect is attached.

```
[assembly: OurLoggingAspect(AttributeTargetTypes="OurCompany.OurApplication.Controllers.*", AttributePriority = 1)]
[assembly: OurLoggingAspect(AttributeTargetMembers="Dispose", AttributeExclude = true, AttributePriority = 2)]
```

In the example above, the first multicast line indicates that the `OurLoggingAspect` should be attached to all methods in the `Controllers` namespace. The second multicast line indicates that the `OurLoggingAspect` should not be applied to any method named `Dispose`.

NOTE

Notice the `AttributePriority` property that is set in both of the multicast lines. Since there is no guarantee that the compiler will apply the attributes in the order you have specified in the code, it is necessary to declare an order to ensure processing is completed as desired.

In this case, the `OurLoggingAspect` will be applied to all methods in the `Controllers` namespace first. After that is completed, the second multicast of `OurLoggingAspect` is performed which then excludes the aspect from methods named `Dispose`.

See [Overriding and Removing Aspect Instances on page 123](#) for more details about excluding and overriding aspects.

Filtering by class visibility

Now that you've been able to apply our aspect to all classes in a namespace and its sub-namespaces, you may be faced with the need to restrict that broad stroke. For example, you may want to apply your aspect only to classes defined as being public.

1. Add the `AttributeTargetTypeAttributes` property to the `MulticastAttribute` constructor.
2. Set the `AttributeTargetTypeAttributes` value to `Public`.

```
[assembly: OurLoggingAspect(AttributeTargetTypes="OurCompany.OurApplication.Controllers.*",
    AttributeTargetTypeAttributes = MulticastAttributes.Public)]
```

By combining `AttributeTargetTypeAttributes` values you are able to create many combinations that are appropriate for your needs.

NOTE

When specifying attributes of target members or types, do not forget to provide all categories of flags, not only the category on which you want to put a restriction.

Filtering by method modifiers

Filtering at the class level may not be granular enough for your needs. Aspects can be attached at the method level and you will want to control filtering on these aspects as well. Let's look at an example of how to apply aspects only to methods marked as virtual.

1. Add the `AttributeTargetMemberAttributes` property to the `MulticastAttribute`'s constructor.
2. Set the `AttributeTargetMemberAttributes` value to `Virtual`.

```
[assembly: OurLoggingAspect(AttributeTargetTypes="OurCompany.OurApplication.Controllers.*", AttributeTargetMember
```

Using this technique you can apply a method-level aspect, or stop it from being applied, based on the existence or non-existence of things like the `static`, `abstract`, and `virtual` keywords.

Programmatic filtering

There are situations where you will want to filter in a way that isn't based on class or method declarations. You may want to apply an aspect only if a class inherits from a specific class or implements a certain interface. There needs to be a way for you to accomplish this.

The easiest way is to override the `CompileTimeValidate(Object)` method of your aspect class, where you can perform your custom filtering. This is the opt-out approach. Have the `CompileTimeValidate(Object)` method return `false` without throwing an exception, and the target candidate will be ignored. See the section [Validating Aspect Usage on page 393](#) for details.

The second approach is opt-in. See the section [Adding Aspects Programmatically using `IAспектProvider` on page 131](#) for details.

9.2.1. Understanding Attribute Multicasting

This topic contains the following sections:

- Overview of the Multicasting Algorithm
- Filtering Target Elements of Code
- Filtering Properties

- Overriding Filtering Attributes

Overview of the Multicasting Algorithm

Every multicast attribute class must be assigned a set of legitimate targets using the `MulticastAttributeUsageAttribute` custom attribute, which is the equivalent and complement of `AttributeUsageAttribute` for multicast attributes. Multicast attributes can be applied to types, methods, fields, properties, events, and parameters. For instance, a caching aspect targets methods. A field validation aspect targets fields.

When a field-level multicast attribute is applied to a type, the attribute is implicitly applied to all fields of that type. When it is applied to an assembly, it is implicitly applied to all fields of that assembly.

The general rule is: when a multicast attribute is applied to a container, it is implicitly (and recursively) applied to all elements of that container.

The next table illustrates how this rule translates for different kinds of targets.

Directly applied to	Implicitly applied to
Assembly or Module	Types, methods, fields, properties, parameters, and events contained in this assembly or module.
Type	Methods, fields, properties, parameters, and events contained in this type.
Property or Event	Accessors of this property or event.
Method	This method and the parameters of this method.
Field	This field.
Parameter	This parameter.

Filtering Target Elements of Code

Note that the default behavior is maximalist: we apply the attribute to *all* contained elements. However, PostSharp provides a way to restrict the set of elements to which the attribute is multicast: filtering.

Both the attribute developer and the user of the aspect can specify filters.

Developer-Specified Filtering

Just like normal custom attributes should be decorated with the `[AttributeUsage]` custom attribute, multicast custom attributes must be decorated by the `[MulticastAttributeUsage]` attribute (see `MulticastAttributeUsageAttribute`). It specifies which are the valid targets of the multicast attributes.

For instance, the following piece of code specifies that the attribute `GuiThreadAttribute` can be applied to instance methods. Aspect users experience a build-time error when trying to use this aspect on a constructor or static method.

```
[MulticastAttributeUsage(MulticastTargets.Method, TargetMemberAttributes = MulticastAttributes.Instance)]
[AttributeUsage(AttributeTargets.Assembly|AttributeTargets.Class|AttributeTargets.Method, AllowMultiple = true)]
[Serializable]
publicclass GuiThreadAttribute : MethodInterceptionAspect
{
    // Details skipped.
}
```

Note the presence of the `AttributeUsageAttribute` attribute in the sample above. It tells the C# or VB compiler that the attribute can be directly applied to assemblies, classes, constructors, or methods. But this aspect will never be eventually applied to an assembly or a class. Indeed, the `MulticastAttributeUsageAttribute` attribute specifies that the sole valid targets are methods. Furthermore, the `TargetMemberAttributes` property establishes a filter that includes only instance methods.

Therefore, if the aspect is applied to a type containing an abstract method, the aspect will not be multicast to this method, nor to its constructors.

TIP

In addition to multicast filtering, consider using programmatic validation of aspect usage. Any custom attribute can implement `IValidableAnnotation` to implement build-time validation of targets. Aspects that derive from `Aspect` already implement these interfaces: your aspect can override the method `CompileTimeValidate(Object)`.

TIP

As an aspect developer, you should enforce as many restrictions as necessary to ensure that your aspect is only used in the way you intended, and raise errors in other cases. Using an aspect in an unexpected way may result in runtime errors that are difficult to debug.

User-Specified Filtering

The attribute user can specify multicasting filters using specific properties of the `MulticastAttribute` class. To make it clear that these properties only impact the multicasting process, they have the prefix `Attribute`. Only the attribute user can set the values of these properties.

As an aspect developer, do not set the values of these properties in the aspect constructor. Instead, use multicast filtering or programmatic validation of aspect usage described in the section [Developer-Specified Filtering on page 115](#).

As an aspect user, it is important to understand that you can only apply aspects to elements of code that have been allowed by the developer of the aspect.

For instance, the following element of code adds a tracing aspect to all public methods of a namespace:

```
[assembly: Trace( AttributeTargetTypes="AdventureWorks.BusinessLayer.*", AttributeTargetMemberAttributes = MulticastAttribi
```

Filtering Properties

The following table lists the filters available to users and developers of aspects:

MulticastAttribute Property (for ;aspect users)	MulticastAttributeUsage- Attribute Property (for aspect developers)	Description
<code>AttributeTargetElements</code>	<code>ValidOn</code>	Restricts the kinds of targets (assemblies, classes, value types, delegates, interfaces, properties, events, properties, methods, constructors, parameters) to which the attribute can be indirectly applied.
<code>AttributeTargetAssemblies</code>		Wildcard expression or regular expression specifying to which assemblies the attribute is multicast.
	<code>AllowExternalAssemblies</code>	Determines whether the aspect can be applied to elements defined in a different assembly than the current one.
<code>AttributeTargetTypes</code>		Wildcard expression or regular expression filtering by name the type to which the attribute is applied, or the declaring type of the member to which the attribute is applied.
<code>AttributeTargetTypeAttributes</code>	<code>TargetTypeAttributes</code>	Restricts the visibility of the type to which the aspect is applied, or of the type declaring the member to which the aspect is applied.

MulticastAttribute Property (for ;aspect users)	MulticastAttributeUsage- Attribute Property (for aspect developers)	Description
<code>AttributeTargetMembers</code>		Wildcard expression or regular expression filtering by name the member to which the attribute is applied.
<code>AttributeTargetMemberAttributes</code>	<code>TargetMemberAttributes</code>	Restricts the attributes (visibility, virtuality, abstraction, literality, ...) of the member to which the aspect is applied.
<code>AttributeTargetParameters</code>		Wildcard expression or regular expression specifying to which parameter the attribute is multicast.
<code>AttributeTargetParameterAttributes</code>	<code>TargetParameterAttributes</code>	Restricts the attributes (in/out/ref) of the parameter to which the aspect is applied.
<code>AttributeInheritance</code>	<code>Inheritance</code>	Specifies whether the aspect is propagated along the lines of inheritance of the target interface, class, method, or parameter (see Understanding Aspect Inheritance on page 121).

CAUTION NOTE

Whenever possible, do not rely on naming conventions to apply aspects (properties `AttributeTargetTypes`, `AttributeTargetMembers` and `AttributeTargetParameters`). This may work perfectly today, and break tomorrow if someone renames an element of code without being aware of the aspect.

Overriding Filtering Attributes

Suppose we have two classes A and B, B being derived from A. Both A and B can be decorated with `MulticastAttributeUsageAttribute`. However, since B is derived from A, filters on B cannot be more permissive than filters on A.

In other words, the `MulticastAttributeUsageAttribute` custom attribute is inherited. It can be overwritten in derived classes, but derived class cannot *enlarge* the set of possible targets. They can only *restrict* it.

Similarly (and hopefully predictably), the aspect user is subject to the same rule: they can restrict the set of possible targets supported by the aspect, but cannot enlarge it.

9.3. Adding Aspects to Derived Classes and Methods

By default, aspects apply to the class or class member which your attribute has been applied to. However, PostSharp provides the ability to specify aspect inheritance which can allow your attributes to be inherited in derived classes. This feature, named *aspect inheritance* can be specified on types, methods, and parameters, but not on properties or events.

This topic contains the following sections:

- [Applying aspects to derived types on page 118](#)
- [Setting inheritance on a per-usage basis on page 118](#)
- [Applying aspects to overridden methods on page 119](#)
- [Applying aspects to new methods of derived types on page 121](#)

Applying aspects to derived types

One way to implement aspect inheritance is to add a `MulticastAttributeUsageAttribute` custom attribute to your aspect class. Aspects that apply to types are typically derived from `TypeLevelAspect` or `InstanceLevelAspect`.

The benefit of this approach is that the aspect will be automatically applied to all derived classes, eliminating the need to manually setup attributes in the derived classes. Moreover, this logic lives in one place.

The following steps describe how to enable aspect inheritance on existing aspect, derived from `TypeLevelAspect`, which applies a `DataContractAttribute` attribute to the base and all derived classes, and a `DataMemberAttribute` attribute to all properties of the base class and those of derived classes:

How to enable aspect inheritance on existing aspect:

1. Create a `TypeLevelAspect` which implements `IAAspectProvider`.
2. Decorate `AutoDataContractAttribute` with the `MulticastAttribute`, and set the `Inheritance` to `Strict`. Note that `MulticastInheritance.Strict` and `MulticastInheritance.Multicast` have the same effect when applied to type-level aspects.

```
[MulticastAttributeUsage( Inheritance = MulticastInheritance.Strict )]
[Serializable]
publicsealedclass AutoDataContractAttribute : TypeLevelAspect, IAAspectProvider
{
    // Details skipped - see the full sample.
}
```

NOTE

Aspect classes need to be serializable. For further details, see [Understanding Aspect Lifetime and Scope on page 389](#).

3. Decorate your base class with `AutoDataContractAttribute`. The following snippet shows a base customer class and a derived customer class:

```
[AutoDataContract]
class Document
{
    publicstring Title { get; set; }
    publicstring Author { get; set; }
    public DateTime PublishedOn { get; set; }
}

class MultiPageArticle : Document
{
    public List<ArticlePage> Pages { get; set; }
}
```

When the attribute is applied to the base class, the `DataContractAttribute` and `DataMemberAttribute` attributes will be applied at compile time to both classes. If other derived classes were added, then these would be decorated automatically as well.

Setting inheritance on a per-usage basis

Specifying targets and attribute inheritance can also be done on a per-usage basis rather than hard-coding it into the custom attribute. In the following snippet, we've removed the `MulticastAttributeUsageAttribute` attribute from `AutoDataContractAttribute`:

```
// [MulticastAttributeUsage( Inheritance = MulticastInheritance.Strict )]
[Serializable]
```

```
publicsealedclass AutoDataContractAttribute : TypeLevelAspect, IAspectProvider
{
    // Details skipped.
}
```

Now the inheritance mode can be specified directly on the `AutoDataContractAttribute` instance by setting the `AttributeInheritance` property as shown here:

```
[AutoDataContract( AttributeInheritance = MulticastInheritance.Strict )]
class Document
{
    // Details skipped.
}
```

Applying aspects to overridden methods

The following example shows a custom attribute which when applied to a class, writes a message to the console window whenever a method enters and exits:

```
[PSerializable]
publicsealedclass TraceMethodAttribute : OnMethodBoundaryAspect
{
    publicoverridevoid OnEntry(MethodExecutionArgs args)
    {
        Console.WriteLine( string.Format( "Entering {0}.{1}.", args.Method.DeclaringType.Name, args.Method.Name ) );
    }

    publicoverridevoid OnExit(MethodExecutionArgs args)
    {
        Console.WriteLine( string.Format( "Leaving {0}.{1}.", args.Method.DeclaringType.Name, args.Method.Name ) );
    }
}
```

Specifying inheritance is simply a matter of adding the `MulticastAttributeUsageAttribute` attribute and specifying the inheritance type, or to set the `AttributeInheritance` property on the custom attribute usage.

In the snippet below, we have added the `TraceMethod` aspect to a virtual method and used the `AttributeInheritance` property to require the aspect to be automatically applied to all overriding methods:

```
class Document
{
    // Details skipped.// This method will be traced.
    [TraceMethod( AttributeInheritance = MulticastInheritance.Strict )]
    publicvirtualvoid RenderHtml(StringBuilder html)
    {
        html.AppendLine( this.Title );
        html.AppendLine( this.Author );
    }
}

class MultiPageArticle: Document
{
    // This method will be traced.publicoverridevoid RenderHtml(StringBuilder html)
    {
        base.RenderHtml(html);
        foreach ( ArticlePage page inthis.Pages )
        {
            page.RenderHtml( html );
        }
    }

    // This method will NOT be traced.publicvoid RenderHtmlPage(StringBuilder html, int pageIndex )
    {
        html.AppendFormat ( "{0}, page {1}", this.Title, pageIndex+1 );
        html.AppendLine();
        html.AppendLine( this.Author );
    }
}
```

```
}
}
```

In this example, `TraceMethodAttribute` will output entry and exit messages for `Document.RenderHtml` method and `MultiPageArticle.RenderHtml` method as shown here:

```
Entering MultiPageArticle.RenderHtml
Entering Document.RenderHtml
Leaving Document.RenderHtml
Leaving MultiPageArticle.RenderHtml
```

NOTE

Aspect inheritance works with virtual, abstract and interface methods and their parameters.

We would get a similar result by adding the `TraceMethod` attribute to the `Document` class. Indeed, by virtue of attribute multicasting (see section [Adding Aspects to Multiple Declarations on page 112](#) for more details), adding a method-level attribute to a class implicitly adds it to all method of this class.

```
[TraceMethod(AttributeInheritance = MulticastInheritance.Strict)]
class Document
{
    // All property getters and setters will be traced.    publicstring Title { get; set; }
    publicstring Author { get; set; }
    public DateTime PublishedOn { get; set; }

    // This method will be traced.publicvirtualvoid RenderHtml(StringBuilder html)
    {
        html.AppendLine( this.Title );
        html.AppendLine( this.Author );
    }
}

class MultiPageArticle: Document
{
    // Property getters and setters will NOT be traced.public List<ArticlePage> Pages { get; set; }

    // This method will be traced.publicoverridevoid RenderHtml(StringBuilder html)
    {
        base.RenderHtml( html );
        foreach ( ArticlePage page inthis.Pages )
        {
            page.RenderHtml( html );
        }
    }

    // This method will NOT be traced.publicvoid RenderHtmlPage(StringBuilder html, int pageIndex )
    {
        html.AppendFormat ( "{0}, page {1}", this.Title, pageIndex+1 );
        html.AppendLine();
        html.AppendLine( this.Author );
    }
}
```

However, by adding the `TraceMethod` aspect to all methods of the `Document` type, we added it to property getters and setters, influencing the output:

```
Entering MultiPageArticle.RenderHtml
Entering Document.RenderHtml
Entering Document.get_Title
Leaving Document.get_Title
Entering Document.get_Author
Leaving Document.get_Author
```



```
Leaving Document.RenderHtml
Leaving MultiPageArticle.RenderHtml
```

Applying aspects to new methods of derived types

In the previous section the `TraceMethod` attribute used *Strict inheritance* which means that if the base class is decorated with the attribute, it will only be applied to methods which are declared in the base class and overridden in the derived class.

By changing the inheritance mode to `Multicast`, we specify that the aspect should be also be applied to new methods of the derived class, i.e. not only methods that are overridden from the base class.

In the following snippet we've changed inheritance from `Strict` to `Multicast`:

```
[TraceMethod(AttributeInheritance = MulticastInheritance.Multicast)]
class Document
{
    // All property getters and setters will be traced.    public string Title { get; set; }
    public string Author { get; set; }
    public DateTime PublishedOn { get; set; }

    // This method will be traced. public virtual void RenderHtml(StringBuilder html)
    {
        html.AppendLine( this.Title );
        html.AppendLine( this.Author );
    }
}

class MultiPageArticle: Document
{
    // Property getters and setters will ALSO be traced. public List<ArticlePage> Pages { get; set; }

    // This method will be traced. public override void RenderHtml(StringBuilder html)
    {
        base.RenderHtml( html );
        foreach ( ArticlePage page in this.Pages )
        {
            page.RenderHtml( html );
        }
    }

    // This method will ALSO be traced. public void RenderHtmlPage(StringBuilder html, int pageIndex )
    {
        html.AppendFormat ( "{0}, page {1}", this.Title, pageIndex+1 );
        html.AppendLine();
        html.AppendLine( this.Author );
    }
}
```

With *Strict inheritance* in use, `TraceMethodAttribute` applied to `Document` was not applied to the `RenderHtmlPage` method and the `Pages` property. In other words, as the name suggests, *Strict inheritance* is strictly applying the attribute on base members and any derived members which are inherited. However, with *Multicast inheritance*, the aspect is also applied to the `RenderHtmlPage` method and the `Pages` property.

Strict inheritance evaluates multicasting and then inheritance, but *Multicast inheritance* evaluates inheritance and then multicasting.

9.3.1. Understanding Aspect Inheritance

This topic contains the following sections:

- Lines of Inheritance
- Strict and Multicast Inheritance

Lines of Inheritance

Aspect inheritance is supported on the following elements.

Aspect Applied On	Aspect Propagated To
Interface	Any class implementing this interface or any other interface deriving this interface.
Class	Any class derived from this class.
Virtual or Abstract Methods	Any method implementing or overriding this method.
Interface Methods	Any method implementing that interface semantic.
Parameter or Return Value of an abstract, virtual or interface method	The corresponding parameter or to the return value of derived methods using the method-level rules described above.
Assembly	All assemblies referencing (directly or not) this assembly.

NOTE

Aspect inheritance is not supported on events and properties, but it is supported on event and property accessors. The reason for this limitation is that there is actually nothing like *event inheritance* or *property inheritance* in MSIL (events and properties have nearly no existence for the CLR: they are pure metadata intended for compilers). Obviously, aspect inheritance is not supported on fields.

Strict and Multicast Inheritance

To understand the difference between strict and multicast inheritance, remember the original role of `MulticastAttribute`: to propagate custom attributes along the lines of containment. So, if you apply a method-level attribute to a type, the attribute will be propagated to all the methods of this type (some methods can be filtered out using specific properties of `MulticastAttribute`, or `MulticastAttributeUsageAttribute`; see [Adding Aspects Declaratively Using Attributes on page 111](#) for details).

The difference between strict and multicast inheritance is that, with multicasting inheritance (but not with strict inheritance), even inherited attributes are propagated along the lines of containment.

Consider the following piece of code, where `A` and `B` are both method-level aspects.

```
[A(AttributeInheritance = MulticastInheritance.Strict)]
[B(AttributeInheritance = MulticastInheritance.Multicast)]
public class BaseClass
{
    // Aspect A, B. public virtual void Method1();
}

public class DerivedClass : BaseClass
{
    // Aspects A, B. public override void Method1() {}

    // Aspect B. public void Method2();
}
```

If you just look at `BaseClass`, there is no difference between strict and multicasting inheritance. However, if you look at `DerivedClass`, you see the difference: only aspect `B` is applied to `MethodB`.

The multicasting mechanism for aspect `A` is the following:

1. Propagation along the lines of containment from `BaseClass` to `BaseClass.Method1`.
2. Propagation along the lines of inheritance from `BaseClass.Method1` to `DerivedClass.Method1`.

For aspect B, the mechanism is the following:

1. Propagation along the lines of containment from `BaseClass` to `BaseClass.Method1`.
2. Propagation along the lines of inheritance from `BaseClass.Method1` to `DerivedClass.Method1`.
3. Propagation along the lines of inheritance from `BaseClass` to `DerivedClass`.
4. Propagation along the lines of containment from `DerivedClass` to `DerivedClass.Method1` and `DerivedClass.Method2`.

In other words, the difference between strict and multicasting inheritance is that multicasting inheritance applies containment propagation rules to inherited aspects; strict inheritance does not.

Avoiding Duplicate Aspects

If you read again the multicasting mechanism for aspect B, you will notice that the aspect B is actually applied twice to `DerivedClass.Method1`: one instance comes from the inheritance propagation from `BaseClass.Method1`, the other instance comes from containment propagation from `DerivedClass`.

To avoid surprises, PostSharp implements a mechanism to avoid duplicate aspect instances. The rule: if many paths lead from the same custom attribute usage to the same target element, only one instance of this custom attribute is applied to the target element.

CAUTION NOTE

Attention: you can still have many instances of the same custom attribute on the same target element if they have *different origins* (i.e. they originate from different lines of code, typically). You can enforce uniqueness of custom attribute instances by using `AllowMultiple`. See the section [Overriding and Removing Aspect Instances on page 123](#) for details.

9.4. Overriding and Removing Aspect Instances

Having multiple instances of the same aspect on the same element of code is sometimes the desired behavior. With multicasting custom attributes (`MulticastAttribute`), it is easy to end up with that situation. Indeed, many multicasting paths can lead to the same target.

However, most of the time, a different behavior is preferred. We could define a method-level aspect on the type (this aspect would apply to all methods) and override (or even exclude) the aspect on a specific method.

The multicasting engine has both the ability to apply multiple aspect instances on the same target, and the ability to replace or remove custom attributes.

Understanding the Multicasting Algorithm

Before going ahead, it is important to understand the multicasting algorithm. The algorithm relies on a notion of *order of processing* of aspect instances.

IMPORTANT NOTE

This section covers how PostSharp handles multiple instances of the **same aspect type** for the sole purpose of computing how aspect instances should be overridden or removed. See [Coping with Several Aspects on the Same Target on page 409](#) to understand how to cope with multiple instances of different aspects.

The following rules apply:

1. Aspect instances defined on a container (for instance a type) have always precedence over instances defined on an item of that container (for instance a method). Elements of code are processed in the following order: assembly, module, type, field, property, event, method, parameter.
2. When multiple aspect instances are defined on the same level, they are sorted by increasing value of the `AttributePriority`.

The algorithm builds a list of aspect instances applied (directly and indirectly) on an element of code, sorts these instances, and processes overrides or removals as described below.

Applying Multiple Instances of the Same Aspect

The property `MulticastAttributeUsageAttribute.AllowMultiple` determines whether multiple instances of the same aspect are allowed on an element of code. By default, this property is set to `true` for all aspects.

In the following example, the methods in type `MyClass` are enhanced by one, two and three instances of the `Trace` aspect (see code comments).

```
using System;
using System.Diagnostics;
using PostSharp.Aspects;
using PostSharp.Extensibility;
using PostSharp.Serialization;
using Samples3;

[assembly: Trace(AttributeTargetTypes = "Samples3.My*", Category = "A")]
[assembly: Trace(AttributeTargetTypes = "Samples3.My*",
    AttributeTargetMemberAttributes = MulticastAttributes.Public, Category = "B")]

namespace Samples3
{
    [PSerializable]
    publicsealedclass TraceAttribute : OnMethodBoundaryAspect
    {
        publicstring Category { get; set; }

        publicoverridevoid OnEntry(MethodExecutionArgs args)
        {
            Trace.WriteLine("Entering " +
                args.Method.DeclaringType.FullName + "." + args.Method.Name, this.Category);
        }
    }

    publicclass MyClass
    {
        // This method will have 1 Trace aspect with Category set to A.
        privatevoid Method1()
        {
        }

        // This method will have 2 Trace aspects with Category set to A, B.
        publicvoid Method2()
        {
        }

        // This method will have 3 Trace aspects with Category set to A, B, C.
        [Trace(Category = "C")]
        publicvoid Method3()
        {
        }
    }
}
```

Overriding an Aspect Instance Manually

You can require an aspect instance to override any previous one by setting the aspect property `AttributeReplace`. This is equivalent to a deletion followed by an insertion (see below).

In the following examples, the first two methods of type `MyClass` are enhanced by aspects applied at assembly level, but these aspects are replaced by a different one on `Method3`.

```
using System;
using System.Diagnostics;
using PostSharp.Aspects;
using PostSharp.Extensibility;
using PostSharp.Serialization;
using Samples5;

[assembly: Trace(AttributeTargetTypes = "Samples5.My*", Category = "A")]
[assembly: Trace(AttributeTargetTypes = "Samples5.My*",
    AttributeTargetMemberAttributes = MulticastAttributes.Public, Category = "B")]

namespace Samples5
{
    [Serializable]
    publicsealedclass TraceAttribute : OnMethodBoundaryAspect
    {
        publicstring Category { get; set; }

        publicoverridevoid OnEntry(MethodExecutionArgs args)
        {
            Trace.WriteLine("Entering " +
                args.Method.DeclaringType.FullName + "." + args.Method.Name, this.Category);
        }
    }

    publicclass MyClass
    {
        // This method will have 1 Trace aspect with Category set to A.
        privatevoid Method1()
        {
        }

        // This method will have 2 Trace aspect with Category set to A, B.
        publicvoid Method2()
        {
        }

        // This method will have 1 Trace aspects with Category set to C.
        [Trace(Category = "C", AttributeReplace = true)]
        publicvoid Method3()
        {
        }
    }
}
```

Overriding an Aspect Instance Automatically

To cause a new aspect instance to automatically override any previous one, the aspect developer must disallow multiple instances by annotating the aspect class with the custom attribute `MulticastAttributeUsageAttribute` and setting the property `AllowMultiple` to `false`.

In the following example, the methods in type `MyClass` are enhanced by a single `Trace` aspect:

```
using System;
using System.Diagnostics;
using PostSharp.Aspects;
using PostSharp.Extensibility;
using PostSharp.Serialization;
using Samples4;
```

Adding Aspects Declaratively Using Attributes

```
[assembly: Trace(AttributeTargetTypes = "Samples4.My*", AttributePriority = 1, Category = "A")]
[assembly: Trace(AttributeTargetTypes = "Samples4.My*",
    AttributeTargetMemberAttributes = MulticastAttributes.Public, AttributePriority = 2, Category = "B")]

namespace Samples4
{
    [MulticastAttributeUsage(MulticastTargets.Method, AllowMultiple = false)]
    [Serializable]
    publicsealedclass TraceAttribute : OnMethodBoundaryAspect
    {
        publicstring Category { get; set; }

        publicoverridevoid OnEntry(MethodExecutionArgs args)
        {
            Trace.WriteLine("Entering " +
                args.Method.DeclaringType.FullName + "." + args.Method.Name, this.Category);
        }
    }

    publicclass MyClass
    {
        // This method will have 1 Trace aspect with Category set to A.
        privatevoid Method1()
        {
        }

        // This method will have 1 Trace aspects with Category set to B.
        publicvoid Method2()
        {
        }

        // This method will have 1 Trace aspects with Category set to C.
        [Trace(Category = "C")]
        publicvoid Method3()
        {
        }
    }
}
```

Deleting an Aspect Instance

The `MulticastAttributeAttributeExclude` property removes any previous instance of the same aspect on a target.

This is useful, for instance, when you need to exclude a target from the matching set of a wildcard expression. For instance:

```
[assembly: Configurable( AttributeTypes = "BusinessLayer.*" )]

namespace BusinessLayer
{
    [Configurable( AttributeExclude = true )]
    publicstaticclass Helpers
    {
    }
}
```

9.5. Reflecting Aspect Instances at Runtime

Attribute multicasting has been primarily designed as a mechanism to add aspects to a program. Most of the time, the custom attribute representing an aspect can be removed after the aspect has been applied.

By default, if you add an aspect to a program and look at the resulting program using a disassembler or `System.Reflection`, you will not find these corresponding custom attributes.

If you need your aspect (or any other multicast attribute) to be reflected by `System.Reflection` or any other tool, you have to set the `MulticastAttributeUsageAttributePersistMetaData` property to `true`.

For instance:

```
[MulticastAttributeUsage( MulticastTargets.Class, PersistMetaData = true )]
publicclass TagAttribute : MulticastAttribute
{
    publicstring Tag;
}
```

NOTE

Multicasting of attributes is not limited only to PostSharp aspects. You can multicast any custom attribute in your codebase in the same way as shown here. If a custom attribute is multicast with the `PersistMetaData` property set to `true`, when reflected on the compiled code will look as if you had manually added the custom attribute in all of the locations.

CHAPTER 10

Adding Aspects Using XML

PostSharp not only allows aspects to be applied in code, but also through XML. This is accomplished by adding them to your project's .psproj file.

Adding aspects through XML gives the advantage of applying aspects without modifying the source code, which could be an advantage in some legacy projects.

Specifying an attribute in XML

```
namespace MyCustomAttributes
{
    // We set up multicast inheritance so the aspect is automatically added to children types.
    [MulticastAttributeUsage( MulticastTargets.Class, Inheritance = MulticastInheritance.Strict )]
    [Serializable]
    publicsealedclass AutoDataContractAttribute : TypeLevelAspect, IAspectProvider
    {
        // Details skipped.
    }
}
```

Normally `AutoDataContractAttribute` would be applied to `Customer` in code as follows:

```
namespace MyNamespace
{
    [AutoDataContractAttribute]
    class Customer
    {
        publicstring FirstName { get; set; }
        publicstring LastName { get; set; }
    }
}
```

Using XML instead, we can remove the custom attribute from source code and instead specify a `Multicast` element in the PostSharp project file, a file that has the same name as your project file (csproj or vbproj), but with the .psproj extension:

```
<?xmlversion="1.0"encoding="utf-8"?><Projectxmlns="http://schemas.postsharp.org/1.0/configuration"><Multicastxmlns:my="clr
```

In this snippet, the `xmlns:my` attribute associates a prefix to an XML namespace, which must be mapped to the .NET namespace and assembly where custom attributes classes are defined:

```
<Multicastxmlns:my="clr-namespace:MyCustomAttributes;assembly:MyAssembly">
```

The next line then specifies the custom attribute to apply and the target attributes to apply the custom attributes to:

```
<my:AutoDataContractAttributeAttributeTargetTypes="MyNamespace.Customer"/>
```

The XML element name must be the name of a class inside the .NET namespace and assembly as defined by the XML namespace. Attributes of this XML element map to public properties or fields of this class.

Note that any property inherited from `MulticastAttribute` can be used here in order to apply the aspect to several classes at a time. See the section [Adding Aspects to Multiple Declarations on page 112](#) for details about these properties.

CHAPTER 11

Adding Aspects Programmatically using IAspectProvider

You may have situations where you are looking to implement an aspect as part of a larger pattern. Perhaps you want to add an aspect, implement an interface and dynamically inject some logic into the target code. In those situations, you will want to apply an aspect to the target code and have that aspect then add other aspects to other elements of code.

The theoretical concept can cause some mental gymnastics, so let's take a look at the implementation.

1. Create an aspect that implements that `IAspectProvider` interface.

```
public class ProviderAspect : IAspectProvider
{
    public IEnumerable<AspectInstance> ProvideAspects(object targetElement)
    {
        throw new System.NotImplementedException();
    }
}
```

2. Cast the target object parameter to the type that will be targeted by this aspect: `Assembly`, `Type`, `MethodInfo`, `ConstructorInfo` or `LocationInfo`.

```
public IEnumerable<AspectInstance> ProvideAspects(object targetElement)
{
    Type type = (Type) targetElement;

    throw new NotImplementedException();
}
```

3. The `ProvideAspects(Object)` method returns an `AspectInstance` of the aspect type you want, for every target element of code.

```
public IEnumerable<AspectInstance> ProvideAspects(object targetElement)
{
    Type type = (Type) targetElement;

    return type.GetMethods().Select(
        m => new AspectInstance(m, new LoggingAspect() ) );
}
```

This aspect will now add aspects dynamically at compile time. Use of the `IAspectProvider` interface and technique is usually reserved for situations where you are trying to implement a larger design pattern. For example, it would be used when implementing an aspect that created the `NotifyPropertyChangedAttribute` pattern across a large number of locations in your codebase. It is overkill for many of the situations that you will encounter. Use it only for complicated pattern implementation aspects that you will create.

NOTE

To read more about `NotifyPropertyChangedAttribute`, see [Handling Corner Cases on page 211](#).

NOTE

PostSharp does not automatically initialize the aspects provided by `IAspectProvider`, even if the method `CompileTimeInitialize` is defined. Any initialization, if necessary, should be done in the `ProvideAspects` method or in the constructor of provided aspects.

However, these aspects are initialized at run time just like normal aspects using the `RunTimeInitialize` method.

Creating Graphs of Aspects

It is common that aspects provided by `IAspectProvider` (children aspects) form a complex object graph. For instance, children aspects may contain a reference to the parent aspect.

An interesting feature of PostSharp is that object graphs instantiated at compile-time are serialized, and can be used at run-time. In other words, if you store a reference to another aspect in a child aspect, you will be able to use this reference at run time.

PART 4

Logging

CHAPTER 12

Adding Detailed Logging to your Solution

When you're working with your codebase, it's common to need to add logging either as a non-functional requirement or simply to assist during the development process. In either situation, you will want to include information about the parameters passed to the method when it was called as well as the parameter values once the method call has completed. This can be a tedious and brittle process. As you work and refactor methods, the order and types of parameters may change, parameters may be added and some may be removed. Along with performing these refactorings, you have to remember to update the logging messages to keep them in sync. This is something that is easy to forget, and once forgotten, the output of the logging is much less useful.

Logging is one of the examples of a boilerplate code. Performing logging imperatively not only leads to the problems with refactoring mentioned above. It also makes your code harder to understand.

PostSharp offers a solution to all of these problems. PostSharp Logging allows you to configure where logging should be performed and takes over the task of keeping your log entries in sync as you add, remove and refactor your codebase. Using PostSharp Logging does not require changing your codebase allowing you to keep the production code clearly understandable with no boilerplate code.

Let's take a look at how you can add trace logging for the start and completion of method calls.

This topic contains the following sections:

- [Step 1. Adding logging to your projects on page 135](#)
- [Step 2. Choose your logging framework on page 136](#)
- [Step 3. Configure PostSharp logging at run-time on page 136](#)
- [Step 4. Configure your logging framework on page 137](#)
- [Result on page 137](#)

Step 1. Adding logging to your projects

To add logging to a specific project:

1. Add a reference to the *PostSharp.Patterns.Diagnostics* package to your project.
2. Create a source code file where you will add all project-wise aspects. We suggest naming this file *Global-Aspects.cs*.

Add the following content to this file:

```
using PostSharp.Patterns.Diagnostics;
using PostSharp.Extensibility;
```

```
[assembly: Log(AttributePriority = 1, AttributeTargetMemberAttributes = MulticastAttributes.Protected | Multicast
[assembly: Log(AttributePriority = 2, AttributeExclude = true, AttributeTargetMembers = "get_*" )]
```

This code adds logging to all methods except private methods and except property getters. You can edit this code to target methods relevant to your scenarios. See [Adding Aspects to Code on page 109](#) for details.

PostSharp will now add logging before and after the execution of all methods targeted by the logging aspect.

NOTE

If there are several projects in your solution, repeat this procedure for each project. Note that you can share the *GlobalAspects.cs* file among several projects.

The next step is to configure logging at run-time. You should at least determine using which logging framework should the records be written with. We call this concept the *logging backend*.

Step 2. Choose your logging framework

The role of PostSharp Logging is to generate logging records, but PostSharp itself does not intend to write these logs to files, databases, or network services. Several excellent open-source projects and commercial services already fulfill this role. In PostSharp terminology, the target logging framework is called the logging *back-end*. In order to see the logged records, you first need to choose and then configure a logging back-end.

PostSharp integrates with several logging frameworks right out of the box. You can choose from the following implementations:

Name	Class	Package
Microsoft Application Insights	ApplicationInsightsLoggingBackend	PostSharp.Patterns.Diagnostics.ApplicationInsights
Common.Logging	CommonLoggingLoggingBackend	PostSharp.Patterns.Diagnostics.CommonLogging
System.Console.WriteLine	ConsoleLoggingBackend	PostSharp.Patterns.Diagnostics
Enterprise Library	EnterpriseLibraryLoggingBackend	PostSharp.Patterns.Diagnostics.EnterpriseLibrary
ETW (System.Diagnostics.EventSource)	EventSourceLoggingBackend	PostSharp.Patterns.Diagnostics.Tracing
Log4Net	Log4NetLoggingBackend	PostSharp.Patterns.Diagnostics.Log4Net
Microsoft.Extensions.Logging	MicrosoftLoggingBackend	PostSharp.Patterns.Diagnostics.Microsoft
NLog	NLogLoggingBackend	PostSharp.Patterns.Diagnostics.NLog
Serilog	SerilogLoggingBackend	PostSharp.Patterns.Diagnostics.Serilog
System.Diagnostics.Trace	TraceLoggingBackend	PostSharp.Patterns.Diagnostics.Tracing
System.Diagnostics.TraceSource	TraceSourceLoggingBackend	PostSharp.Patterns.Diagnostics.Tracing
Loupe	LoupeLoggingBackend	PostSharp.Patterns.Diagnostics.Loupe

Alternatively, you can easily implement a custom backend. See [Implementing an Adapter to a Custom Logging Framework on page 177](#) for details.

Step 3. Configure PostSharp logging at run-time**To configure logging:**

1. Identify the startup method of your solution. In a console application and an ASP.NET Core application, the startup method is usually named `Program.Main`. In a XAML application, this is the `Startup` event handler. In an ASP.NET application, this is the `Application_Start` method in the *Global.asax* source file.
2. To the startup project, add a reference to the package containing the implementation of your logging backend, as listed in the table above.
3. Import the `PostSharp.Patterns.Diagnostics` namespace in the startup file.

```
using PostSharp.Patterns.Diagnostics;
```


4. Add the following code on the top of the startup method:

```
LoggingServices.DefaultBackend = new Patterns.Diagnostics.Backends.Console.ConsoleLoggingBackend();
```

In the code snippet above, you can replace **ConsoleLoggingBackend** by any of the logging backends.

5. It is essential that you configure logging before any logged type (i.e. any type containing a logged method or field) is initialized, otherwise `TypeInitializationException` will be thrown at build time. Therefore, you should add the following custom attribute on the top of the startup class:

```
[Log(AttributeExclude = true)]  
class Program  
{  
    // ...  
}
```

IMPORTANT NOTE

You should exclude the logging aspect from any class that is executed before you set up and configure the backend. If this is too cumbersome, you can create a module initializer for your startup project using the `ModuleInitializerAttribute` aspect.

Step 4. Configure your logging framework.

Remember that PostSharp emits records to the logging framework of your choice. Some of these frameworks may need additional configuration. Please refer to the documentation of the logging framework for details.

Result

Now that you have added logging to your project method, you will get a super-detailed log of your program execution, including parameter values and return values. You are able to add, remove, or rename a method or its parameters with the confidence that your log entries will be kept in sync with each of those changes. Adding logging to your codebase and maintaining it becomes a very easy task.

CHAPTER 13

Configuring Specific Logging Frameworks

This chapter explains how to configure PostSharp Logging to specifically work with different logging frameworks.

It contains the following articles:

[Logging to NLog on page 141](#)

[Logging to log4net on page 140](#)

[Logging to Serilog on page 142](#)

[Logging to the System Console on page 140](#)

[Logging to ETW on page 144](#)

[Logging to Common.Logging on page 144](#)

[Logging to Loupe on page 145](#)

13.1. Logging to the System Console

This article shows how to target the output of PostSharp Logging to the console and how to configure it.

Configuring PostSharp Logging to use the console

To use target PostSharp Logging to the console:

1. Add PostSharp logging to your codebase as described in [Adding Detailed Logging to your Solution on page 135](#).
2. In the application startup file, include the following namespace imports:

```
using PostSharp.Patterns.Diagnostics;  
using PostSharp.Patterns.Diagnostics.Backends.Console;
```

In the application startup method, include the following code:

```
LoggingServices.DefaultBackend = new ConsoleLoggingBackend();
```

Theming the console

13.2. Logging to log4net

This article shows how to use PostSharp Logging and log4net together.

To use PostSharp Logging with log4net:

1. Add PostSharp logging to your codebase as described in [Adding Detailed Logging to your Solution on page 135](#).
2. Add the *PostSharp.Patterns.Diagnostics.Log4Net* package to your startup project.
3. Create an XML file named *log4net.config*. In the file properties, set the **Copy to Output Directory** property to **Copy always**.

Add the following content to this file:

```
<log4net><appendername="file" type="log4net.Appender.FileAppender"><filevalue="log4net.log"/><layouttype="log4net.Lay
```

This example configuration file instructs log4net to write all log records to a file named *log4net.log* and to the console.

See the [log4net documentation](#)²⁶ for details about this configuration file.

26. <https://logging.apache.org/log4net/release/manual/configuration.html>

4. In the application startup file, include the following namespace imports:

```
using log4net.Config;
using PostSharp.Patterns.Diagnostics;
using PostSharp.Patterns.Diagnostics.Backends.Log4Net;
```

In the application startup method, include the following code:

```
// Configure Log4Net
XmlConfigurator.Configure(new FileInfo("log4net.config"));

// Configure PostSharp Logging to use Log4Net
LoggingServices.DefaultBackend = new Log4NetLoggingBackend();
```

If you run your application, you should now see a detailed log in a file named *log4net.log*. If your application is a console application, you should also see the log in the console.

13.3. Logging to NLog

This article shows how to use PostSharp Logging and nlog together.

To use PostSharp Logging with NLog:

1. Add PostSharp logging to your codebase as described in [Adding Detailed Logging to your Solution on page 135](#).
2. Add the *PostSharp.Patterns.Diagnostics.NLog* package to your startup project.

3. In the application startup file, include the following namespace imports:

```
using NLog;
using NLog.Config;
using NLog.Targets;
using PostSharp.Patterns.Diagnostics;
using PostSharp.Patterns.Diagnostics.Backends.NLog;
```

In the application startup method, include the following code:

```
// Configure NLog.
var nlogConfig = new LoggingConfiguration();

var fileTarget = new FileTarget("file")
{
    FileName = "nlog.log",
    KeepFileOpen = true,
    ConcurrentWrites = false,
};

nlogConfig.AddTarget(fileTarget);
nlogConfig.LoggingRules.Add(new LoggingRule("*", LogLevel.Debug, fileTarget));

var consoleTarget = new ConsoleTarget("console");
nlogConfig.AddTarget(consoleTarget);
nlogConfig.LoggingRules.Add(new LoggingRule("*", LogLevel.Debug, consoleTarget));

LogManager.EnableLogging();

// Configure PostSharp Logging to use NLog.
LoggingServices.DefaultBackend = new NLogLoggingBackend(new LogFactory(nlogConfig));
```

This example code instructs NLog to write all log records to a file named *nlog.log* and to the console. If you prefer, you can configure NLog with a configuration file. See the [NLog documentation](#)²⁷ for details.

If you run your application, you should now see a detailed log in a file named *nlog.log*. If your application is a console application, you should also see the log in the console.

13.4. Logging to Serilog

This article shows how to use PostSharp Logging and Serilog together.

This topic contains the following sections:

- [Getting started with PostSharp Logging and Serilog on page 142](#)
- [Using Serilog formatters instead of PostSharp ones on page 143](#)
- [Including more semantic parameters on page 143](#)

Getting started with PostSharp Logging and Serilog

To use PostSharp Logging with Serilog:

1. Add PostSharp logging to your codebase as described in [Adding Detailed Logging to your Solution on page 135](#).

27. <https://github.com/NLog/NLog/wiki/Tutorial#configuration>

2. Add the `PostSharp.Patterns.Diagnostics.Serilog` package to your startup project, as well as the packages for all Serilog sinks you will need. For the next code examples, install the packages `Serilog.Sinks.ColoredConsole` and `Serilog.Sinks.File`.
3. In the application startup file, include the following namespace imports:

```
using PostSharp.Patterns.Diagnostics;
using PostSharp.Patterns.Diagnostics.Backends.Serilog;
using Serilog;
```

In the application startup method, include the following code:

```
// The output template must include {Indent} for nice output.conststring template = "{Timestamp:yyyy-MM-dd HH:mm:ss} {Level:u} {Message}";

// Configure a Serilog logger.var logger = new LoggerConfiguration()
    .MinimumLevel.Debug()
    .WriteTo.File("serilog.log", outputTemplate: template)
    .WriteTo.ColoredConsole(outputTemplate: template)
    .CreateLogger();

// Configure PostSharp Logging to use Serilog
LoggingServices.DefaultBackend = new SerilogLoggingBackend(logger);
```

This example code instructs Serilog to write all log records to a file named `serilog.log` and to the console.

Note that this code snippet supplies a custom output template. By default, Serilog does not indent the logs generated by PostSharp. PostSharp passes the indentation string to Serilog as a parameter named `Indent`. In order to produce indented logs, you need to include the `{Indent:1}` parameter into your output template.

See the [Serilog documentation](#)²⁸ for more details.

If you run your application, you should now see a detailed log in a file named `serilog.log`. If your application is a console application, you should also see the log in the console.

Using Serilog formatters instead of PostSharp ones

By default, PostSharp Logging will use its own formatters to format parameter values into a string and will pass the already-formatted string to Serilog.

If you want the unformatted parameter value to be passed to Serilog instead of the string, set the `SerilogLoggingBackendOptionsUseSerilogFormatters` property to `true`.

Including more semantic parameters

By default, PostSharp Logging passes the following pieces of information to Serilog as semantic parameters: parameter values, return values, `this` value, and execution time. Other pieces of information such as for instance the type and method name are included in the message template.

If you want pass more pieces of information as semantic parameters, set the `SerilogLoggingBackendOptionsSemanticParametersTreatedSemantically` and `SerilogLoggingBackendOptionsIncludedSpecialProperties` properties.

NOTE

Semantic parameters have a relatively high performance cost in Serilog. We suggest you do not use more semantic parameters than necessary.

28. <https://github.com/serilog/serilog/wiki/Configuration-Basics>

13.5. Logging to ETW

This article shows how to use PostSharp Logging and Event Tracing for Windows (ETW) together.

If you run your application, you should now see a detailed log in a file named *nlog.log*. If your application is a console application, you should also see the log in the console.

This topic contains the following sections:

- [Targeting ETW on page 144](#)
- [Listening to ETW events on page 144](#)

Targeting ETW

To target ETW with PostSharp Logging:

1. Add PostSharp logging to your codebase as described in [Adding Detailed Logging to your Solution on page 135](#).
2. Add the *PostSharp.Patterns.Diagnostics.Tracing* package to your startup project.
3. In the application startup file, include the following namespace imports:

```
using PostSharp.Patterns.Diagnostics;  
using PostSharp.Patterns.Diagnostics.Backends.EventSource;
```

In the application startup method, include the following code:

```
var eventSourceBackend = new EventSourceLoggingBackend(new PostSharpEventSource());  
if (eventSourceBackend.EventSource.ConstructionException != null)  
    throw eventSourceBackend.EventSource.ConstructionException;
```

As a result of this procedure, PostSharp Logging will emit records to the ETW event source named `PostSharp-Patterns-Diagnostics`. You now need to attach a listener to this event source.

Listening to ETW events

ETW is a very complex system and explaining it is beyond the scope of this documentation. Let's just show how you can collect and view ETW events on your development machine.

We will use a tool named **PerfView** developed by Microsoft.

To collect and view PostSharp Logging logs using PerfView:

13.6. Logging to Common.Logging

This article shows how to use PostSharp Logging and Common.Logging together.

To use PostSharp Logging with Common.Logging:

1. Add PostSharp logging to your codebase as described in [Adding Detailed Logging to your Solution on page 135](#).
2. Add the *PostSharp.Patterns.Diagnostics.CommonLogging* package to your startup project.

3. In the application startup file, include the following namespace imports:

```
using Common.Logging;
using Common.Logging.Simple;
using PostSharp.Patterns.Diagnostics;
using PostSharp.Patterns.Diagnostics.Backends.CommonLogging;
```

In the application startup method, include the following code:

```
// Configure Common.Logging to direct outputs to the system console.
LogManager.Adapter = new ConsoleOutLoggerFactoryAdapter();

// Configure PostSharp Logging to direct outputs to Common.Logging.
LoggingServices.DefaultBackend = new CommonLoggingLoggingBackend();
```

This example code instructs Common.Logging to write all log records to the system console. See the [Common.Logging documentation](#)²⁹ for details.

If you run your application, you should also see the log in the console.

13.7. Logging to Loupe

This article shows how to use PostSharp Logging and Loupe together.

To use PostSharp Logging with Loupe:

1. Add PostSharp logging to your codebase as described in [Adding Detailed Logging to your Solution on page 135](#).
2. Add the *PostSharp.Patterns.Diagnostics.Loupe* package to your startup project.
3. Follow the instructions of the first step [Loupe configuration tutorial](#)³⁰ to properly initialize Loupe for your application kind.

In the following examples, we will show instructions for a console or WinForms application.

In the application startup file, include the following namespace imports:

```
using Gibraltar.Agent;
using PostSharp.Patterns.Diagnostics;
using PostSharp.Patterns.Diagnostics.Backends.Loupe;
```

In the application startup method, include the following code before any logged code:

```
// Initialize Loupe.
Log.StartSession();

// Configure PostSharp Logging to use Loupe.
LoggingServices.DefaultBackend = new LoupeLoggingBackend();
```

If you followed the [Loupe configuration tutorial](#)³¹ for a different type of application, do not forget to set the `DefaultBackend` just after the call to `Log.StartSession()`.

29. <http://netcommon.sourceforge.net/docs/2.1.0/reference/html/ch01.html#logging-config>

30. https://doc.onloupe.com/#GettingStarted_Introduction.html

31. https://doc.onloupe.com/#GettingStarted_Introduction.html

4. Before the application exits, include the following code:

```
// Initialize Loupe.  
Log.EndSession();
```

5. In order to enable source file information, create an XML file named *postsharp.config* in your project with the following content:

```
<?xmlversion="1.0"encoding="utf-8"?><Projectxmlns="http://schemas.postsharp.org/1.0/configuration"><Loggingxmlns="cl
```

6. At this point, you should be able to use Loupe Viewer to see the live log of your running application. You can [download Loupe Viewer](#)³² and open it in the background. When you start your application, you will see a desktop notification from Loupe. Click on this notification to open the live log. Note that the live log will disappear as soon as `Log.EndSession` has been called.
7. Configure your application to send data to Loupe server as described in the second step of the [Loupe configuration tutorial](#)³³.

You will need to add some settings in your *app.config* or *web.config*. Replace the `customerName` attribute with the name of your Loupe subscription. If you don't have a Loupe subscription, you can [sign up for a free trial](#)³⁴.

```
<configuration><configSections><sectionGroupName="gibraltar"><sectionname="publisher" type="Gibraltar.Agent.Publisher
```

32. <https://onloupe.com/local-logging/free-net-log-viewer/>

33. https://doc.onloupe.com/#GettingStarted_Introduction.html

34. <https://onloupe.com/>

CHAPTER 14

Customizing the Appearance of Log Records

PostSharp offers several ways to customize the appearance and the content of log records. This article maps different customization scenarios to a procedure, then explains each procedure in detail.

This topic contains the following sections:

- [Scenarios on page 147](#)
- [Editing a build-time configuration on page 148](#)
- [Editing run-time options on page 149](#)
- [Implementing a custom formatter on page 149](#)
- [Overriding a backend on page 149](#)

Scenarios

Scenario	Procedure
Include execution time. Emit a warning when execution time exceeds a given threshold. Include source file and line information. Include information about which task or method is being awaited by the <code>await</code> operator.	Edit the build-time configuration on page 148 (LoggingProfile) in <i>postsharp.config</i>.
Include parameter name, type, or value. Include return value. Include <code>this</code> parameter. Set the <code>LogLevel</code> .	Edit the build-time configuration on page 148 (LoggingOptions) in <i>postsharp.config</i>.
Set the maximum length of a record. Set formatting options such as the delimiter character or the number of indentation spaces. Include the type name, namespace, or name. Include exception details. Specify which exceptions should be logged.	Edit run-time options on page 149 (TextLoggingBackendOptions or a derived class).
Change how parameter values of a specific type are rendered.	Implement a custom formatter on page 149.
Change any other behavior.	Override the backend on page 149.

Editing a build-time configuration

Some configuration settings affect the way how PostSharp generates instructions. These settings can have an important effect on run-time execution speed or assembly size. These settings are defined in the *postsharp.config* file. This file can be private to a project or shared among several projects. See [Working with PostSharp Configuration Files on page 97](#) for details about the *postsharp.config* file.

Because settings in *postsharp.config* affect code generation, you will have to rebuild your project after you modify this file.

In large applications, you may need to configure logging differently for different areas or layers in your application. For example, exceptions in the service that cleans up old data in the database can be logged with a "Warning" level, while exceptions in the customer-facing web service must be logged with the "Error" level.

PostSharp enables you to organize your logging options using *Logging Profiles*. You apply a given logging profile by providing its name as an argument for the constructor of the `LogAttribute` constructor.

Logging aspects are assigned to a default profile, as shown in the following table:

Aspect	Default logging profile
<code>LogAttribute</code>	default
<code>LogExceptionAttribute</code>	exceptions
<code>AuditAttribute</code>	audit (see Adding Audit to your Solution on page 173).

To edit the default logging profile:

1. Open the file *postsharp.config* in your project. If it does not exist, create an XML file with the following content:

```
<?xmlversion="1.0"encoding="utf-8"?><Projectxmlns="http://schemas.postsharp.org/1.0/configuration"></Project>
```

NOTE

If you create *postsharp.config* in a directory, the file will be used for any project located under that directory. This allows you to share the logging profile among several projects. See [Working with PostSharp Configuration Files on page 97](#) for details.

2. Edit *postsharp.config* as in the following example:

```
<?xmlversion="1.0"encoding="utf-8"?><Projectxmlns="http://schemas.postsharp.org/1.0/configuration"><Loggingxmlns=
```

See the documentation of the `LoggingProfile` and `LoggingOptions` classes for details.

To create and use a new logging profile:

1. Configure a logging profile in *postsharp.config*, but choose a different name than default, exceptions or audit.

```
<?xmlversion="1.0"encoding="utf-8"?><Projectxmlns="http://schemas.postsharp.org/1.0/configuration"><Loggingxmlns=
```

- Specify the profile name when you add the `LogAttribute` aspect to a method, type, or project.

```
[assembly: Log("detailed", AttributeTargetTypes = "My.Namespace.*", AttributePriority = 1, AttributeTargetMember/
```

NOTE

Alternatively, you can create a new aspect class and derive it from `LogAttributeBase`.

Editing run-time options

The run-time options are available on the `Options` property of the backend class. Each backend can expose a specific set of options.

Contrarily to the build-time options, the run-time options are not organized into profiles. They affect all profiles that use the backend you configure.

To edit run-time options, go to the startup method where PostSharp Logging is initialized, and set the backend options:

```
var backend = new Patterns.Diagnostics.Backends.Console.ConsoleLoggingBackend();
backend.Options.Delimiter = " \u00A6 ";
backend.Options.UseColors = false;
LoggingServices.DefaultBackend = backend ;
```

Implementing a custom formatter

When you include parameter values in log records, PostSharp will invoke the `ToString` method of the object by default. To learn when and how to override the default formatter, see [Implementing a Custom Formatter on page 171](#).

Overriding a backend

If none of the previous options are sufficient, you can create your own backend. Instead of creating your own implementation from scratch, you can derive any existing implementation and override relevant methods.

See [Implementing an Adapter to a Custom Logging Framework on page 177](#) for details.

CHAPTER 15

Adding Custom Log Records Manually

When you're using the `LogAttribute` aspect, PostSharp automatically generates code that emits log records before and after the execution of a method. However, there are times when you would want to write your own records - for example, you may want to log a custom error or warning. Such messages are to be displayed even when trace-level logging is disabled, and when it is enabled, they are to appear in the right context and with the proper indentation.

For these scenarios, you can use the methods provided by the `LogSource` class.

In this chapter

Section	Description
Writing Custom Messages on page 151	This article shows how to write standalone messages to the log. It discusses the difference between formatted-text messages and semantic messages, the latter being more appropriate for statistical processing.
Working with Custom Activities on page 155	This article describes how to define custom activities, i.e. nested scopes of operations that have a description and can succeed or fail.
Adding Properties to Messages and Activities on page 158	This article shows how to add and configure properties to your custom messages and custom activities.

15.1. Writing Custom Messages

This section describes how to write a standalone message using PostSharp Logging. Instead of a standalone message, you will often want to write a pair of messages to mark the beginning and the end (successful or not) of an activity. For this scenario, see [Working with Custom Activities on page 155](#).

This topic contains the following sections:

- [Writing text messages on page 152](#)
- [Writing text messages with parameters on page 152](#)
- [Writing semantic messages for easy statistical processing on page 153](#)
- [Using default log levels on page 154](#)
- [Optimizing performance with the 'Elvis' operator on page 155](#)
- [Implementing your own type of messages on page 155](#)

IMPORTANT NOTE

All projects that write manual log records and activities should be processed by PostSharp, otherwise the execution of your application will be slower. Therefore your projects should have a reference to the `PostSharp.Patterns.Common` package (a reference to `PostSharp.Patterns.Common.Redist` is not sufficient), and PostSharp should not be disabled on this project.

Writing text messages

The simplest scenario is to write a constant string.

To write a custom text message to the log:

1. Add the following statements on the top of your C# file:

```
using PostSharp.Patterns.Diagnostics;
usingstatic PostSharp.Patterns.Diagnostics.FormattedMessageBuilder;
```

2. Add and initialize a static read-only field of type `LogSource` and initialize it with the `LogSource.Get` method:

```
privatestaticreadonly LogSource logSource = LogSource.Get();
```

3. Evaluate the property corresponding to the desired log level, e.g. `Debug` or `Error`.

4. Invoke the `Write(T, WriteMessageOptions)` method. For the first method, construct the message object by invoking the `Formatted` method.

```
logSource.Debug.Write( Formatted( "Hello, world." ) );
```

Example

```
using PostSharp.Patterns.Diagnostics;
usingstatic PostSharp.Patterns.Diagnostics.FormattedMessageBuilder;

staticclass Hasher
{
    privatestaticreadonly LogSource logSource = LogSource.Get();

    publicstaticasync Task ReadAndHashAsync(string url)
    {
        if ( string.IsNullOrEmpty( url ) )
        {
            logSource.Warning.Write( Formatted( "Empty URL passed. Skipping this method." ) );
            return;
        }

        // Details skipped.
    }
}
```

Writing text messages with parameters

Most of the time, you won't log constant strings, but you will want to include variable pieces of information. In this case, you can use one of the overloads of the `Formatted` method that accepts formatting parameters.

Note that the specification of the formatting string used by the `Formatted` method is **not** identical to the one used by `string.Format`. The formatting string used by the `Formatted` method is designed to support named parameters (for use with logging backends that support it, e.g. Serilog connected to Elastic Search) and for high-performance evaluation.

The formatting string has the following specifications:

- Named parameters must be surrounded by curly brackets, e.g. `{MyParameter}`.
- Values are matched to named parameters by position. This means that the order of named parameters in the formatting string must match the order of corresponding values passed to the `Formatted` method and that two named parameters with the same name are not matched to the same value.
- Anything inside a pair of curly brackets is considered as the parameter name and will be passed to the backend as it, without further parsing.

- Formatting specifiers are not supported but may be partially supported in the future. Do not use colons (:) in your parameter names, as they may be interpreted differently in future versions of PostSharp.
- Use the escaped form of curly brackets {{ and }} if you want to include curly brackets in the formatted string.

IMPORTANT NOTE

Even if you are not using a semantic backend, consider the performance impact of using `string.Format` or equivalent constructs such as interpolated strings. PostSharp Logging is highly optimized and is able to generate a logging record without allocating any memory on the heap. If you're using a high-performance backend, using `string.Format` can bring a significant performance overhead to your logging.

Example

```
using PostSharp.Patterns.Diagnostics;
using static PostSharp.Patterns.Diagnostics.FormattedMessageBuilder;

static class Hasher
{
    private static readonly LogSource logSource = LogSource.Get();

    public static byte[] ReadAndHash(string url)
    {
        var hashAlgorithm = HashAlgorithm.Create("SHA256");
        hashAlgorithm.Initialize();

        var webClient = new WebClient();
        var buffer = new byte[16 * 1024];

        logSource.Info.Write( Formatted( "Using a {BufferSize}-byte buffer.", buffer.Length ) );

        using (var stream = webClient.OpenRead(url))
        {
            int countRead;
            while ((countRead = stream.Read(buffer, 0, buffer.Length)) != 0)
            {
                logSource.Info.Write( Formatted( "Got {CountRead} bytes.", countRead ) );
                hashAlgorithm.ComputeHash(buffer, 0, countRead);
            }
        }

        return hashAlgorithm.Hash;
    }
}
```

Writing semantic messages for easy statistical processing

One common scenario in production is to prioritize the resolution of warnings based on the number of occurrences. It often makes more sense to start working on a warning happening 10 times per second than on one happening once per day.

Unfortunately, with text-based messages like those produced by `FormattedMessageBuilder`, this is cumbersome to achieve. The reason is that there is no message property to bucketize on.

When statistical processing of messages is important to you, you should use, instead of formatted messages, semantic messages produced by the `SemanticMessageBuilder` class. Semantic messages have a name and a list of properties, but no text.

To emit semantic messages, use the `Semantic` method instead of `Formatted`.

Example

```
using PostSharp.Patterns.Diagnostics;
using static PostSharp.Patterns.Diagnostics.SemanticMessageBuilder;
```

Adding Custom Log Records Manually

```
staticclass Hasher
{
    privatestaticreadonly LogSource logSource = LogSource.Get();

    publicstaticbyte[] ReadAndHash(string url)
    {
        var hashAlgorithm = HashAlgorithm.Create("SHA256");
        hashAlgorithm.Initialize();

        var webClient = new WebClient();
        var buffer = newbyte[16 * 1024];

        logSource.Info.Write( Semantic( "Initialize", ("BufferSize", buffer.Length ) ) );

        using (var stream = webClient.OpenRead(url))
        {
            int countRead;
            while ((countRead = stream.Read(buffer, 0, buffer.Length)) != 0)
            {
                logSource.Info.Write( Semantic( "ReadChunk", ("CountRead", countRead ) ) );
                hashAlgorithm.ComputeHash(buffer, 0, countRead);
            }
        }

        return hashAlgorithm.Hash;
    }
}
```

Using default log levels

In the previous sections, we have explicitly specified the message severity by using the `Debug`, `Trace`, `Info`, `Warning`, `Error` and `Critical` properties. Instead of using these properties, you can use the `WithLevel(LogLevel)` method, which takes a parameter of type `LogLevel`.

Alternatively, you can use the `Default` and `Failure` properties. They resolve to the default level for success or failure messages. The default levels are `Debug` for `Default` and `Error` for `Failure`. The default levels can be configured with the `WithLevels(LogLevel, LogLevel)` method. This method allows you to configure the default levels from a central place.

To configure the default logging levels centrally:

1. Define an internal static field of type `LogSource` for the prototype instance. Instantiate the prototype instance using `LogSourceGet` and configure it using `LogSourceWithLevels(LogLevel, LogLevel)`.
2. For each type, clone the prototype using the `LogSourceForType(Type)` or `LogSourceForCurrentType` method.

Example

The following example illustrates how to configure log sources centrally:

```
using PostSharp.Patterns.Diagnostics;
usingstatic PostSharp.Patterns.Diagnostics.FormattedMessageBuilder;

staticclass LogSources
{
    // Configure a prototype LogSource that you will reuse in several classes.publicstaticreadonly LogSource Default = LogS
}

class MyClass
{
    // Instantiates a LogSource from the prototype for the current type.staticreadonly LogSource logSource = LogSources.Def

    void MyMethod()
    {
        // Write a message with default verbosity.
        logSource.Default.Write( Formatted( "Hello, World." ) );
    }
}
```

Optimizing performance with the 'Elvis' operator

When you are writing a message, the first thing that the `LogLevelSourceWriteT(T, WriteMessageOptions)` method does is to check whether the requested logging level is enabled. If not, nothing else happens. If yes, the message is rendered and emitted. Although this may seem fast enough for many scenarios, sometimes the cost of evaluating the parameters passed to `FormattedMessageBuilderFormatted` and `SemanticMessageBuilderSemantic` is prohibitive in itself. For these situations, it is preferable to skip the invocation of `Formatted`, `Semantic` and `WriteT(T, WriteMessageOptions)` altogether if the current verbosity is insufficient.

One way to conditionally emit a message is to use an `if` and test for the `IsEnabled` property.

However, it is much more convenient to use the `EnabledOrNull` and the C# "Elvis" operator, e.g. as in the construct `logSource.Debug.EnabledOrNull?.Write`.

Example

The following example uses the `EnabledOrNull` and the Elvis operator to make sure the expensive `File.GetLastWriteTime` method is evaluated only when debug-level logging is enabled.

```
logSource.Debug.EnabledOrNull?.Write( Formatted( "The last change date of the file is {LastChangeDate}.", File.GetLastWriteTime)
```

Implementing your own type of messages

If neither `FormattedMessageBuilder` nor `SemanticMessageBuilder` fit your needs, you can create your own type of messages.

Messages are types (preferably value types) that implement the `IMessage` interface. It has a single method, `Write(ICustomLogRecordBuilder, CustomLogRecordItem)`, which should render the message into an `ICustomLogRecordBuilder`.

You can use the `FormattingStringParser` type if you want to reuse the same formatting string syntax as the one used by `FormattedMessageBuilder`.

15.2. Working with Custom Activities

More often than not, you will find yourself logging the beginning and the end of an activity, e.g. `Starting to read MyFile.xml` and `Succeeded to read MyFile.xml` or `Cannot read MyFile.xml: unexpected element at line 5`. An *activity* is anything that begins and eventually ends.

You can open custom activities using the `OpenActivityT(T, OpenActivityOptions)`, `LogActivity` or `LogActivityAsync` method.

Activities define a scope, so anything executing within that activity is properly indented. Therefore, activities are hierarchical: they have a notion of a child and of a parent and form a tree.

When you use the `LogAttribute` aspect to log a method, it translates into an activity behind the scenes, which belongs to the same tree as custom activities.

This topic contains the following sections:

- [Logging an activity, the explicit way, on page 156](#)
- [Logging an activity the compact way, with an anonymous method, on page 157](#)
- [Changing the default level for exceptions, on page 158](#)
- [Measuring the execution time of activities, on page 158](#)

Logging an activity, the explicit way

To log a custom activity:

1. Add a `LogSource` to your class as described in [Writing Custom Messages on page 151](#).
2. In a `using` statement, invoke the `OpenActivityT(T, OpenActivityOptions)` method. Specify the activity description with the `FormattedMessageBuilderFormatted` or `SemanticMessageBuilderSemantic` method.

TIP

We strongly advise to use the `var` keyword to receive the return value of `OpenActivityT(T, OpenActivityOptions)` method because the return type of the method depends on the number and types of arguments sent to `FormattedMessageBuilderFormatted` or `SemanticMessageBuilderSemantic`. Alternatively, you can use a variable of type **`ILogMessage`**, but this will cause boxing of the `LogActivityTActivityDescription` value type.

3. Wrap the code of the activity with a `try/catch` construct.
4. In the `catch` block, call `SetException(Exception, CloseActivityOptions)`.
5. In the `try`, call `SetSuccess(CloseActivityOptions)` or `SetResultTResult(TResult, CloseActivityOptions)` or, if you need more flexibility, `SetOutcomeTMessage(LogLevel, TMessage, Exception, CloseActivityOptions)`.

CAUTION NOTE

All projects that use the `OpenActivityT(T, OpenActivityOptions)` method in `async` methods MUST be processed by `PostSharp`, otherwise the execution of your application will fail. Therefore your projects must have a reference to the `PostSharp.Patterns.Common` package (a reference to `PostSharp.Patterns.Common.Redist` is not sufficient), and `PostSharp` must not be disabled on this project. Even if you don't have asynchronous code, it is recommended that you use the `PostSharp.Patterns.Common` package for performance reasons.

Example

The following code shows how to open and close an activity.

```
staticclass Hasher
{
    staticreadonly LogSource logSource = Logger.GetLogger();

    privatestaticbyte[] ReadAndHash(string file)
    {
        using ( var activity = logSource.Default.OpenActivity( Formatted( "Processing file {Url}", file) ) )
        {
            try
            {
                var totalSize = 0;

                var hashAlgorithm = HashAlgorithm.Create("SHA256");
                hashAlgorithm.Initialize();

                var buffer = newbyte[128 * 1024];

                logSource.Info.Write( Formatted( "Working with a {BufferSize}-byte buffer.", buffer.Length ) );

                using (var stream = File.OpenRead(file))
                {
                    int countRead;
```

```

        while ((countRead = stream.Read(buffer, 0, buffer.Length)) != 0)
        {
            logSource.Info.Write( Formatted( "Got {CountRead} bytes.", countRead) );

            hashAlgorithm.ComputeHash(buffer, 0, countRead);
            totalSize += countRead;
        }

        activity.SetOutcome( LogLevel.Info, Formatted( "Success. Read {TotalSize} bytes in total.", totalSize) );

        return hashAlgorithm.Hash;
    }
    catch ( Exception e )
    {
        activity.SetException(e);
        throw;
    }
}
}
}
}

```

Logging an activity the compact way, with an anonymous method

If you don't want to write the boilerplate code of the previous example over and over again, you can use the `Log-Activity` or `LogActivityAsync` method. These methods will execute a specified delegate within a new custom activity, and will automatically invoke the `SetSuccess(CloseActivityOptions)`, `SetResultTResult(TResult, CloseActivityOptions)` or `SetException(Exception, CloseActivityOptions)` method as needed.

The inconvenient consequence of using `LogActivity` is that it will likely result in the allocation of an instance of a closure class on the heap, and affect performance even when logging is disabled.

Example

The following code does almost the same as the previous example but with fewer lines of code. The main difference is that, in the current example, the `SetSuccess(CloseActivityOptions)` method is be invoked instead of `SetOutcomeTMessage(LogLevel, TMessage, Exception, CloseActivityOptions)`, and there is heap allocation even when logging is disabled.

```

staticclass Hasher
{
    staticreadonly LogSource logSource = Logger.GetLogger();

    privatestaticbyte[] ReadAndHash(string file)
    {
        var activity = logSource.Default.LogActivity( Formatted( "Processing file {Url}", file),
            () => {
                var totalSize = 0;

                var hashAlgorithm = HashAlgorithm.Create("SHA256");
                hashAlgorithm.Initialize();

                var buffer = newbyte[128 * 1024];

                logSource.Info.Write( Formatted( "Working with a {BufferSize}-byte buffer.", buffer.Length ) );

                using (var stream = File.OpenRead(file))
                {
                    int countRead;
                    while ((countRead = stream.Read(buffer, 0, buffer.Length)) != 0)
                    {
                        logSource.Info.Write( Formatted( "Got {CountRead} bytes.", countRead) );

                        hashAlgorithm.ComputeHash(buffer, 0, countRead);
                        totalSize += countRead;
                    }
                }
            }
        );
    }
}

```

```
        return hashAlgorithm.Hash;  
    } );  
}
```

Changing the default level for exceptions

When you open an activity with a specific level, say `Debug`, and then close the activity successfully, the close message will be emitted with the same level as the open message, i.e. `Debug`.

However, when the activity fails with an exception, the close message level will be `Failure`, whose value can be changed only by the `WithLevel(LogLevel)` method. See [Writing Custom Messages on page 151](#) for information about changing the default levels.

Measuring the execution time of activities

You can measure the execution time of all activities by enabling the `LoggingBackendOptionsIncludeActivityExecutionTime` property:

```
backend.Options.IncludeActivityExecutionTime = true;
```

This will cause the execution time to be appended to the close message.

15.3. Adding Properties to Messages and Activities

When you write a message, open an activity, or close an activity, you can specify additional properties. Properties are name-value pairs that are passed to the logging backend. Unlike message parameters, properties are not rendered into the message text. When defined on activities, properties are inherited by children contexts and activities.

This topic contains the following sections:

- [Defining a property on page 158](#)
- [Late evaluation of properties on page 158](#)
- [Inheritance of properties on page 159](#)
- [Other properties of `LoggingProperty` on page 159](#)

Defining a property

Every method that allows you to write a message, open an activity, or close an activity accepts an optional parameter named `options`. This object has a property named `Properties`. For instance, the `LogLevelSourceWriteT(T, WriteMessageOptions)` method accepts an `options` parameter of type `WriteMessageOptions`, and this type has a `Properties` property.

To add a property to a message or an activity, assign the `Properties` property to an array of `LoggingProperty` instances. A `LoggingProperty` is basically a name-value pair enhanced with a few options.

```
logSource.Default.Write( Formatted( "Hello, World." ),  
    new WriteMessageOptions { Properties = new [] { new LoggingProperty( "User", User.Identity.Name
```

Late evaluation of properties

Since properties added to activities apply to children activities and messages by default, there can be some time between the moment the user code instantiates a `LoggingProperty` and the moment the property is emitted to the back-end. Sometimes, it is desirable to evaluate a property exactly at the moment when the property is being written to the back-end.

Example

The following code opens an activity and defines a property named `Culture`, which should be evaluated to the current culture of the user. Since the user can change the culture during the activity, the property must be evaluated dynamically.

```
using ( var activity = logSource.Default.OpenActivity( Formatted( "Opening MainWindow" ),
                                                    new OpenActivityOptions { Properties = new [] { new LoggingProperty( "Culture", () => Cul
{
    // Long-running operation.
}
```

Inheritance of properties

Activity properties are inherited by default, which means that the property will be added to any child activity or any message within this activity. To disable inheritance, set the `IsInherited` property to `false`.

Property inheritance works even when the activity is hidden because of low verbosity.

NOTE

As a side effect of property inheritance, the `OpenActivityT(T, OpenActivityOptions)` method always opens an activity when it has at least one inherited property, even if the level of the open message is too low to be emitted.

Other properties of LoggingProperty

The `LoggingProperty` class has several properties useful when logging distributed systems. See [Implementing Logging for a Distributed System on page 163](#) for details.

CHAPTER 16

Enabling and Disabling Logging At Run Time

Sometimes when an issue happens in production, you will want to enable tracing for a specific type or namespace dynamically, without rebuilding the application. To prepare for this scenario, you need to add as much logging as is reasonable at build time, but disable it by default at run time, and then enable it selectively.

This topic contains the following sections:

- [Enabling/disabling with the PostSharp Logging API on page 161](#)
- [Optimizing the execution time when logging is disabled on page 161](#)
- [Enabling/disabling with the backend API on page 162](#)

Enabling/disabling with the PostSharp Logging API

To tune the logging verbosity for a specific type or namespace, first get the default `LoggingVerbosityConfiguration` by evaluating the `DefaultVerbosity` property, then use one of the `SetMinimalLevel`, `SetMinimalLevelForType` or `SetMinimalLevelForNamespace` method.

The default logging level is Debug.

NOTE

PostSharp Logging does not implement a mechanism to enable or disable tracing through a configuration file. If you need to be able to configure logging without editing source code, you will have to implement yourself the code reading the configuration and calling the `LoggingVerbosityConfiguration` methods with proper arguments. Alternatively, you can use backend configuration mechanism (see below).

Example

The following code will cause PostSharp to log only exceptions for the `MyCompany` namespace and at the same time all messages for the `MyCompany.BusinessLayer` namespace. The order of the two lines of code is important.

```
LoggingServices.DefaultBackend.DefaultVerbosity.SetMinimalLevelForNamespace(LogLevel.Error, "MyCompany");
LoggingServices.DefaultBackend.DefaultVerbosity.SetMinimalLevelForNamespace(LogLevel.Debug, "MyCompany.BusinessLayer");
```

Changing verbosity for the current execution context only

The `DefaultVerbosity` property has influence on the default execution context. To learn how to override the verbosity for a specific execution context, letting the default verbosity unchanged, see [Choosing Which Requests to Log on page 169](#).

Optimizing the execution time when logging is disabled

By default, PostSharp generates code that allows you to dynamically enable or disable logging for a specific type and severity. However, even if logging is disabled, your CPU still needs to execute the code that evaluates whether logging is enabled.

By setting the `AllowDynamicEnabling` property of the logging profile to `false`, you can ask PostSharp to generate instructions that can be fully eliminated by the JIT compiler when logging is disabled. Therefore, the cost of inactive logging will be close to zero. Note that our last tests show that the JIT compiler still emits instructions when logging is disabled, but it emits the equivalent of `if (false) { Log; }`, which has a very low performance overhead because of branch prediction.

When you set the `AllowDynamicEnabling` property to `false`, you need to configure the `LoggingVerbosity-Configuration` object when the application initializes. Any change done after a logged type is JIT-compiled will be ignored for this specific type.

Example

The following `postsharp.config` enables the JIT-compiler optimizations for the default profile.

```
<?xmlversion="1.0"encoding="utf-8"?><Projectxmlns="http://schemas.postsharp.org/1.0/configuration"><Loggingxmlns="clr-name
```

Enabling/disabling with the backend API

Several logging frameworks offer a configuration mechanism that allows you to enable or disable logging. For a PostSharp log record to be emitted, two conditions need to be met: logging must be enabled by PostSharp (see above) and by the backend logging framework.

Logging frameworks generally have a concept of a *category* or of a *source* (the terminology can vary), which typically is just determined by a string (typically a type or a namespace). The corresponding concept in PostSharp Logging is the `LoggingTypeSource` class, which is determined by two strings: a role (see `LoggingRoles`) and a type name or namespace.

By default, a `LoggingTypeSource` is mapped to a category according to the full type name. If you want to control logging using the backend framework, you will need to customize this mapping. This can be done by setting a property of the `LoggingBackendOptions` class. This property is different for each backend framework.

Therefore, you enable or disable logging using the facilities implemented by the backend logging framework, using the full type name as the source or category name.

CHAPTER 17

Implementing Logging for a Distributed System

A distributed system is composed of several applications calling each other to complete one operation. Each of these applications emits its own logs and often stores them in different locations. With distributed systems, logging is the easy part. What's much harder is to make sense of this ocean of logs from a logical point of view.

If you need to implement logging for a distributed system, the first step is to select a single location to store the logs. Dozens of solutions are available and PostSharp Diagnostics is agnostic about this choice.

The next challenge is to correlate the log records coming from different applications to get a logical view of all records relevant to the processing of a specific request. Typically, this means that you will need to set some HTTP header on client side and read it on server side. If you're using a different technology, such as WCF or MSMQ, you will need to take a different approach.

PostSharp Logging does not automatically add headers to cross-process messages, but it provides the necessary API to do so. It is structured on the [HTTP Correlation Protocol](#)³⁵ specification.

This topic contains the following sections:

- [Generating hierarchical context ids on page 163](#)
- [Marking a property as cross-process on page 164](#)
- [Example on page 164](#)

Generating hierarchical context ids

To correlate traces across processes, it is necessary to assign a proper identifier to each context and record. PostSharp Logging offers a suitable context identifier in the `SyntheticId` property. You can read this identifier from `logSource.CurrentContext.SyntheticId` or `activity.Context.SyntheticId`.

The default implementation of the `SyntheticId` property respects the [Hierarchical Request-Id](#)³⁶ specification.

The most important benefit of this specification is that it is possible to select all children records of a logical distributed operation only by filtering by the beginning of the `SyntheticId`, e.g. in pseudo-SQL:

```
SELECT * FROM Records WHERE SyntheticId LIKE '|45ed51e.1.da4e9679%';
```

Additionally, our implementation generates sortable identifiers, so that ordering records by `SyntheticId` makes sense:

```
SELECT * FROM Records WHERE SyntheticId LIKE '|45ed51e.1.da4e9679%' ORDERBY SyntheticId ASC;
```

The filtering and ordering characteristics of the identifiers depend on the generation strategy, as explained below.

35. <https://github.com/dotnet/corefx/blob/master/src/System.Diagnostics.DiagnosticSource/src/HttpCorrelationProtocol.md>

36. <https://github.com/dotnet/corefx/blob/master/src/System.Diagnostics.DiagnosticSource/src/HierarchicalRequestId.md>

Comparing generation strategies

The generation strategy of the `SyntheticId` property depends on several configuration properties: `ContextIdGenerationStrategy`, `RootSyntheticId`, and `SyntheticIdFormatter`.

`ContextIdGenerationStrategy` is the most important configuration option. It determines how the `LoggingContextId` property (a 64-bit integer) is generated. There are basically two strategies: global and hierarchical. They are compared in the next table.

	Global Strategy	Hierarchical Strategy
Description	This strategy uses a single AppDomain-wide static 64-bit counter to generate the identifier. The <code>SyntheticId</code> is therefore composed of the <code>RootSyntheticId</code> (by default, a random 64-bit integer) followed by the 64-bit <code>Id</code> itself.	The <code>Id</code> property is generated using a counter in the scope of the parent context. If there is no parent context, the global strategy is used. Therefore, the <code>SyntheticId</code> property is composed of the <code>SyntheticId</code> of the parent context, plus the <code>Id</code> of the current context.
Length of generated identifiers	Short.	Potentially very long.
Performance	May cause thread contention on highly loaded systems because several threads may be trying to get exclusive access to the global counter at the same time.	No thread contention issue, but more CPU time required to render the id.
Ordering	Preserving time, but not causality (parallel async calls are mixed).	Preserving causality, but not time (parallel async calls are separated from each other).
Filtering	Only based on cross-process boundaries (see discussion on <code>SyntheticParentId</code> below).	Based on any context (external activity, method, custom in-process activity).

Setting the ParentId of a request

When the logical parent of an activity stems from an external process, you need to read the parent id from the incoming headers, open a new activity (see [Working with Custom Activities on page 155](#)), and set the `OpenActivityOptionsSyntheticParentId` property.

When this property is set, the `SyntheticId` will start with the value of the `SyntheticParentId` property, and the in-process parent activity, if any, will be ignored.

Marking a property as cross-process

You can use the `LoggingPropertyIsBaggage` property to mean that a **LoggingProperty** should be carried across process boundaries. This flag is not used by PostSharp Logging itself. It exists to help you implement the process of writing the proper HTTP headers when emitting an HTTP request.

Example

The following code snippet shows a possible server-side implementation of the [HTTP Correlation Protocol](#)³⁷ and [Hierarchical Request-Id](#)³⁸ specifications. It is based on a `MethodInterceptionAspect` aspect applied to the `HttpClient` class.

```
using System;
using System.Collections.Generic;
using System.Net.Http;
using System.Runtime.Serialization;
using System.Text;
using System.Threading.Tasks;
```

37. <https://github.com/dotnet/corefx/blob/master/src/System.Diagnostics.DiagnosticSource/src/HttpCorrelationProtocol.md>

38. <https://github.com/dotnet/corefx/blob/master/src/System.Diagnostics.DiagnosticSource/src/HierarchicalRequestId.md>

```

using ClientExample;
using PostSharp.Aspects;
using PostSharp.Patterns.Diagnostics;
using PostSharp.Patterns.Formatter;
using PostSharp.Serialization;
using static PostSharp.Patterns.Diagnostics.SemanticMessageBuilder;

// The following attribute intercepts all calls to the specified methods of HttpClient.
[assembly: InstrumentOutgoingRequestsAspect(
    AttributeTargetAssemblies = "System.Net.Http",
    AttributeTargetTypes = "System.Net.Http.HttpClient",
    AttributeTargetMembers = "regex:(Get*|Delete|Post|Push|Patch)Async")]

namespace ClientExample
{
    [PSerializable]
    internalclass InstrumentOutgoingRequestsAspect : MethodInterceptionAspect
    {
        privatestaticreadonly LogSource logSource = LogSource.Get();

        publicoverrideasync Task OnInvokeAsync(MethodInterceptionArgs args)
        {
            var http = (HttpClient) args.Instance;
            var verb = Trim(args.Method.Name, "Async");

            using (var activity = logSource.Default.OpenActivity(Semantic(verb, ("Url", args.Arguments[0]))))
            {
                try
                {
                    // TODO: this implementation conflicts with System.Diagnostics.Activity and therefore Application Insi
                    http.DefaultRequestHeaders.Remove("Request-Id");
                    http.DefaultRequestHeaders.Remove("Correlation-Context");

                    // Set Request-Id header.
                    http.DefaultRequestHeaders.Add("Request-Id", activity.Context.SyntheticId);

                    // Generate the Correlation-Context header.
                    UnsafeStringBuilder correlationContextBuilder = null;
                    var propertyNames = new HashSet<string>();
                    try
                    {
                        activity.Context.ForEachProperty((LoggingProperty property, objectvalue, refobject _) =>
                        {
                            if (!property.IsBaggage || !propertyNames.Add(property.Name)) return;

                            if (correlationContextBuilder == null)
                            {
                                propertyNames = new HashSet<string>();
                                correlationContextBuilder = new UnsafeStringBuilder(1024);
                            }

                            if (correlationContextBuilder.Length > 0)
                            {
                                correlationContextBuilder.Append(", ");
                            }

                            correlationContextBuilder.Append(property.Name);
                            correlationContextBuilder.Append('=');

                            var formatter =
                                property.Formatter ?? LoggingServices.Formatter.Get(value.GetType());

                            formatter.Write(correlationContextBuilder, value);
                        });

                        if (correlationContextBuilder != null)
                        {
                            http.DefaultRequestHeaders.Add("Correlation-Context", correlationContextBuilder.ToString());
                        }
                    }
                }
            }
        }
    }
}

```

Implementing Logging for a Distributed System

```
        finally
        {
            correlationContextBuilder?.Dispose();
        }

        var t = base.OnInvokeAsync(args);

        // We need to call Suspend/Resume because we're calling LogActivity from an aspect and // aspects are |
        {
            activity.Suspend();
            try
            {
                await t;
            }
            finally
            {
                activity.Resume();
            }
        }

        var response = (HttpResponseMessage) args.ReturnValue;

        if (response.IsSuccessStatusCode)
        {
            activity.SetOutcome(LogLevel.Info, Semantic("Succeeded", ("StatusCode", response.StatusCode)));
        }
        else
        {
            activity.SetOutcome(LogLevel.Warning, Semantic("Failed", ("StatusCode", response.StatusCode)));
        }
    }
    catch (Exception e)
    {
        activity.SetException(e);
        throw;
    }
    finally
    {
        http.DefaultRequestHeaders.Remove("Request-Id");
    }
}

}

private static string Trim(string s, string suffix)
    => s.EndsWith(suffix) ? s.Substring(0, s.Length - suffix.Length) : s;
}
}
```

Here is a client-side implementation of the same protocol. It is based on an ASP.NET Action Filter. The filter needs to be added in the `Startup.ConfigureServices` method.

```
using System;
using System.Collections.Generic;
using System.Net;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc.Filters;
using PostSharp.Patterns.Diagnostics;
using static PostSharp.Patterns.Diagnostics.SemanticMessageBuilder;

namespace MicroserviceExample
{
    [Log(AttributeExclude = true)]
    public class LoggingActionFilter : IAsyncActionFilter
    {
        private static readonly LogSource logger = LogSource.Get();

        public async Task OnActionExecutionAsync(ActionExecutingContext context, ActionExecutionDelegate next)
        {
            // Implementation of the logging filter
        }
    }
}
```

```

{
// Read the Request-Id header so we can assign it to the activity.string parentOperationId = context.HttpConte

OpenActivityOptions options = default;

// Set the parent id.if (!string.IsNullOrEmpty(parentOperationId))
{
options.SyntheticParentId = parentOperationId;
}
else
{
options.IsSyntheticRootId = true;
}

// Process cross-context properties (aka baggage).string correlationContext = context.HttpContext.Request.Head
if (!string.IsNullOrEmpty(correlationContext))
{
var properties = new List<LoggingProperty>();
foreach (var pair in correlationContext.Split(',', StringSplitOptions.RemoveEmptyEntries))
{
var posOfEqual = pair.IndexOf('=');
if (posOfEqual <= 0) continue;
var propertyName = pair.Substring(0, posOfEqual);
var propertyValue = pair.Substring(posOfEqual + 1);
properties.Add(new LoggingProperty(propertyName, propertyValue) {IsBaggage = true});
}
options.Properties = properties.ToArray();
}

var request = context.HttpContext.Request;
using (var activity = logger.Default.OpenActivity(Semantic("Request", ("Path", request.Path),
("Query", request.QueryString), ("Method", request.Method)), options))
{
try
{
await next();

var response = context.HttpContext.Response;

if (response.StatusCode >= (int) HttpStatusCode.OK && response.StatusCode <= 299)
{
// Success.
activity.SetOutcome(LogLevel.Info, Semantic("Success", ("StatusCode", response.StatusCode)));
}
else
{
// Failure.
activity.SetOutcome(LogLevel.Warning, Semantic("Failure", ("StatusCode", response.StatusCode)));
}
}
catch (Exception e)
{
activity.SetException(e);
}
}
}
}
}
}
}
}
}
}
}
}
}
}

```


CHAPTER 18

Choosing Which Requests to Log

PostSharp Logging makes it so easy to add logging to your application that you can easily end up capturing gigabytes of data every minute. As it goes, most of this data won't ever be useful, but you still need to pay for storage and bandwidth. The ability to trace an application at a high level of detail is very useful, but it is important to be able to select *when* you want to log.

If you are running a web application, it is probably useless to log every single request with the highest level of detail, especially for request types that are served 100 times per second. Therefore, it is important to be able to decide, at run-time, which requests need to be logged. You may choose to disable logging by default and to enable logging only for select requests only.

The following table describes a few example strategies:

Strategy	Description
Random sampling	You decide, based on a randomly generated number, whether the current request should be logged. The problem of this approach is that frequently-served requests may be overrepresented, and rarely-served requests may be completely absent from the log.
Time window sampling	You log the given request type once per time period (e.g. per 200 ms resulting in maximum 5 requests per second) and ignore all other requests. This approach still results in semi-random sampling and addresses the downside of the previous strategy. However, the efficacy of this approach heavily depends on an algorithm that categorizes request types. If multiple requests of the same type are done in rapid succession, the last request has a lower probability of being logged.
Suspect requests	You log only requests that are likely to fail based on the path, the query string, the user name, the client IP, or any other piece of information available when the request is accepted. The "suspicion level" of a request can be coded manually or could come from statistical processing (AI-based or not) of data of your APM.

To selectively enable logging at request level

1. Disable detailed logging at global level.

```
LoggingServices.DefaultBackend.DefaultVerbosity.SetMinimalLevel( LogLevel.Warning );
```

2. Create a `LoggingVerbosityConfiguration` by calling the `CreateVerbosityConfiguration` method. This `LoggingVerbosityConfiguration` will be used by all requests where you want to enable logging. Store it in a field. Note that the `LoggingVerbosityConfiguration` is a memory-heavy type, so you should avoid creating one instance for every request.

By default, the verbosity is set to Debug. You can configure the `LoggingVerbosityConfiguration` as needed.

3. Intercept the request before it is passed to your application code (typically using an ASP.NET Action Filter or a WCF Behavior), decide whether the request should be logged, and call the `[M:PostSharp.Patterns.Diagnostics.LoggingVerbosityConfiguration.Use()]` method. Dispose the returned token once the request processing has completed.

Example

The following code snippet is an ASP.NET Action Filter that enables detailed logging for a random 10% sample of requests.

```
using System;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc.Filters;
using PostSharp.Patterns.Diagnostics;

namespace MicroserviceExample
{
    [Log(AttributeExclude = true)]
    public class SampledLoggingActionFilter : IAsyncActionFilter
    {
        private static readonly Random random = new Random();
        private static LoggingVerbosityConfiguration verbosityManager;

        public static void Initialize( LoggingBackend backend )
        {
            verbosityManager = backend.CreateVerbosityConfiguration();
        }

        public static bool IsInitialized => verbosityManager != null;

        private static bool IsLogged(ActionExecutingContext context)
        {
            lock (random)
            {
                // Log 10% of requests. return random.NextDouble() < 0.1;
            }
        }

        public async Task OnActionExecutionAsync(ActionExecutingContext context, ActionExecutionDelegate next)
        {
            if ( IsInitialized && IsLogged(context) )
            {
                using ( verbosityManager.Use() )
                {
                    await next();
                }
            }
            else
            {
                await next();
            }
        }
    }
}
```

CHAPTER 19

Implementing a Custom Formatter

Formatters are responsible for representing an object as a `string`. Formatters are used in two contexts: logging and caching. This article describes how to implement a custom formatter.

This topic contains the following sections:

- [When to implement a custom formatter on page 171](#)
- [Implementing the `IFormattable` interface on page 171](#)
- [Implementing the `Formatter` class on page 172](#)
- [Registering the custom formatter on page 172](#)

When to implement a custom formatter

You may consider implementing a custom formatter in two situations:

- You want the object to be formatted differently in different contexts, i.e. you want the logging representation to be different than the caching representation or than the `ToString` representation.
- The formatting is performance-critical. Since custom formatters are based on the `UnsafeStringBuilder` class, they are much faster than formatters based on `ToString` or `string.Format`.

Implementing the `IFormattable` interface

If you own the source code of a type, the easiest way to implement a custom formatter is to make the type implement the `IFormattable` interface, which has a single method named `Format(UnsafeStringBuilder, FormattingRole)`.

The following example shows how to implement the `IFormattable` interface:

```
using PostSharp.Patterns.Diagnostics;
using PostSharp.Patterns.Formatters;

namespace PostSharp.Samples.Logging
{
    class CustomerData : PostSharp.Patterns.Formatters.IFormattable
    {
        publicstring FirstName { get; set; }
        publicstring LastName { get; set; }

        [Log(AttributeExclude=true)]
        void PostSharp.Patterns.Formatters.IFormattable.Format(UnsafeStringBuilder stringBuilder, FormattingRole role)
        {
            stringBuilder.Append("{CustomerData FirstName=\"");
            stringBuilder.Append(this.FirstName);
            stringBuilder.Append("\", LastName=\"");
            stringBuilder.Append(this.LastName);
            stringBuilder.Append("\"");
        }
    }
}
```

TIP

To prevent the formatter from being logged, add `[Log(AttributeExclude=true)]` to the formatting method.

When you implement the `IFormattable` interface, you don't need to register the formatter because the formatter is the object itself.

Implementing the Formatter class

If you don't own the source code of a type, you cannot implement the `IFormattable`. Instead, you can create a new class derived from the `FormatterT` class and implement the `Write(UnsafeStringBuilder, T)` method.

The following example illustrates a formatter for the `Int32` class.

```
using PostSharp.Patterns.Diagnostics;
using PostSharp.Patterns.Formatters;

[Log(AttributeExclude=true)]
class FancyIntFormatter : Formatter<int>
{
    public override void Write(UnsafeStringBuilder stringBuilder, int value)
    {
        switch ( value )
        {
            case 0:
                stringBuilder.Append("zero");
                break;

            case 1:
                stringBuilder.Append("one");
                break;

            case 2:
                stringBuilder.Append("two");
                break;

            case 3:
                stringBuilder.Append("three");
                break;

            default:
                stringBuilder.Append(value);
                break;
        }
    }
}
```

TIP

To prevent the formatter from being logged, add `[Log(AttributeExclude=true)]` to the formatting method.

Registering the custom formatter

Creating a new formatter class does not cause PostSharp to use it. You still need to register it.

Use the following code to register your formatter with PostSharp Logging:

```
LoggingServices.Formatters.Register(new FancyIntFormatter());
```

To use the same formatter in PostSharp Caching, use:

```
CachingServices.Formatters.Register(new FancyIntFormatter());
```

CHAPTER 20

Adding Audit to your Solution

Auditing and logging are technically very similar but serve different purposes. Whereas application logs are typically used by server administrators or technical support engineers to diagnose technical issues, audit trails are primarily used by security analysts and business users.

A typical business application will need to implement both logging and audit.

This article explains how to add audit into your application and how to customize the default behaviors.

This topic contains the following sections:

- [Adding audit to your projects on page 173](#)
- [Adding more information to the audit record on page 174](#)

Adding audit to your projects

To audit your code, you have to add the `AuditAttribute` attribute to all methods to audit, and implement a handler for the `AuditServicesRecordPublished` event. Let's see that in details.

To add audit to your application:

1. Add a reference to the `PostSharp.Patterns.Diagnostics` package to your project.
2. Decide how you want to tag methods that need to be audited. If you want to select them individually, add the `AuditAttribute` attribute to each of them. For instance:

```
[Audit]
publicvoid AssignTo(Employee employee)
{
}
```

If you need to audit dozens or hundreds of methods, you can select them using project-wide matching rule. Create a source code file where you will add all project-wise aspects. We suggest naming this file `Global-Aspects.cs`. Then add the following content to this file:

```
using PostSharp.Patterns.Diagnostics;
using PostSharp.Extensibility;

[assembly: Audit(AttributePriority = 1, AttributeTargetTypes = "Contoso.BusinessObjects.*", AttributeTargetMember
[assembly: Audit(AttributePriority = 2, AttributeExclude = true, AttributeTargetMembers = "get_*" )]
```

This code adds audit to all public methods except property getters. You can edit this code to target methods relevant to your scenarios. See [Adding Aspects to Code on page 109](#) for details.

3. Implement the code that appends the audit record into your database table. This code needs to react to the `AuditServicesRecordPublished` event.

```
AuditServices.RecordPublished += delegate(object o, AuditRecordEventArgs e)
{
    var record = new DbAuditRecord(
        WindowsIdentity.GetCurrent().Name,
        ((BusinessObject)e.Record.Target).Id,
        e.Record.MemberName,
        e.Record.Text
    );
    record.AppendToDatabase();
};
```

Adding more information to the audit record

The `AuditRecord` class exposed by the `AuditServicesRecordPublished` event only contain the most common pieces of information regarding the audited operation. If you want to include more information in the record event, you need to override the audit back-end.

Since the audit back-end is simply another logging back-end, you can use the same approach as described in [Implementing an Adapter to a Custom Logging Framework on page 177](#). Here are the specific steps you need to follow to add more information to the `AuditRecord` class class.

To extend the audit record object:

1. Create a new class derived from `AuditRecord` add all necessary fields or properties.

```
using System;
using System.Collections.Generic;
using PostSharp.Patterns.Diagnostics;
using PostSharp.Patterns.Diagnostics.Audit;

namespace PostSharp.Samples.Audit.Extended
{
    publicclass ExtendedAuditRecord : AuditRecord
    {
        public ExtendedAuditRecord(Type declaringType, string memberName, LogRecordKind recordKind) : base(declaringType, memberName, recordKind)
        {
        }

        public List<BusinessObject> RelatedBusinessObjects { get; } = new List<BusinessObject>();
    }
}
```

2. Create a new class derived from `AuditRecordBuilder`.

Override the `CreateRecord(LoggingContext, LogRecordInfo, LogMemberInfo)` method and return an instance of your new custom `AuditRecord` class. The instance you return here will be stored in the `AuditRecordBuilderCurrentRecord` property.

Then override other methods such as **`SetParameter`1(Int32, String, ParameterDirection, String, UMP, IFormatterUMP)`** and assign the custom fields of properties of the `CurrentRecord` property.

```
using PostSharp.Patterns.Diagnostics.Audit;
using PostSharp.Patterns.Diagnostics.Backends.Audit;
using PostSharp.Patterns.Diagnostics.Contexts;
using PostSharp.Patterns.Diagnostics.RecordBuilders;
using PostSharp.Patterns.Formatters;
using PostSharp.Reflection;

namespace PostSharp.Samples.Audit.Extended
{
    publicclass ExtendedAuditRecordBuilder : AuditRecordBuilder
    {
        public ExtendedAuditRecordBuilder(AuditBackend backend) : base(backend)
        {
        }

        protectedoverride AuditRecord CreateRecord(LoggingContext context, ref LogRecordInfo recordInfo,
            ref LogMemberInfo memberInfo)
        {
            // Return an instance of our own extended class.
            returnnew ExtendedAuditRecord(context.Source.SourceType, me
        }

        publicoverridevoid SetParameter<T>(
            int index, string parameterName, ParameterDirection parameterKind, string typeName, T value, IFormatter<T> form
        {
            base.SetParameter(index, parameterName, parameterKind, typeName, value, formatter);

            // When the parameter is a business object, add it to the list of correlated business objects.
            var bus

            if (businessObject != null)
                ((ExtendedAuditRecord) CurrentRecord).RelatedBusinessObjects.Add(businessObject);
        }
    }
}
```

3. Create a new class derived from **`AuditBackend`**. Override the `CreateRecordBuilder` method and return a new instance of your custom `AuditRecordBuilder`.

```
using PostSharp.Patterns.Diagnostics.Backends.Audit;
using PostSharp.Patterns.Diagnostics.RecordBuilders;

namespace PostSharp.Samples.Audit.Extended
{
    publicclass ExtendedAuditBackend : AuditBackend
    {
        publicoverride LogRecordBuilder CreateRecordBuilder()
        {
            returnnew ExtendedAuditRecordBuilder(this);
        }
    }
}
```

4. Set an instance of new class as the logging back-end for the Audit role:

```
LoggingServices.Roles[LoggingRoles.Audit].Backend = new ExtendedAuditBackend();
```


CHAPTER 21

Implementing an Adapter to a Custom Logging Framework

The adapter between PostSharp Logging and the target logging framework is called a *back-end* in PostSharp terminology. PostSharp comes with ready-made support for the most popular logging frameworks in .NET. If you need to integrate with a different logging solution, you can implement a custom logging back-end.

You need to implement at least 4 classes to implement a custom logging back-end.

This topic contains the following sections:

- [Override the `TextLoggingBackendOptions` class on page 177](#)
- [Override the `LoggingTypeSource` class on page 177](#)
- [Override the `TextLogRecordBuilder` class on page 178](#)
- [Override the `TextLoggingBackend` class on page 179](#)
- [Overriding context classes on page 180](#)

Override the `TextLoggingBackendOptions` class

By design, every logging back-end must implement a class exposing all available run-time options. This is a design decision to expose options on a different class than the back-end class itself. Even if your custom back-end does not expose new options, we suggest you respect this convention and create a new empty class.

Example

```
using PostSharp.Patterns.Diagnostics.Backends;

namespace PostSharp.Samples.Logging.CustomBackend.ServiceStack
{
    public class ServiceStackLoggingBackendOptions : TextLoggingBackendOptions
    {
    }
}
```

Override the `LoggingTypeSource` class

The **LoggingTypeSource** class corresponds to the `ILog` or `ILogger` concept of several logging frameworks. Your **LoggingTypeSource** will typically contain one field of this type, which will be initialized in the constructor.

Additionally to exposing the back-end logger, the **LoggingTypeSource** class exposes the `IsBackendEnabled(LogLevel)` method, which determines whether logging is enabled for the specified level and the current type and role.

Your implementation of **LoggingTypeSource** must be immutable or thread-safe.

Example

```
using System;
using PostSharp.Patterns.Diagnostics;
using ServiceStack.Logging;

namespace PostSharp.Samples.Logging.CustomBackend.ServiceStack
{
    public class ServiceStackLoggingTypeSource : LoggingTypeSource
```

Implementing an Adapter to a Custom Logging Framework

```
{
    public ServiceStackLoggingTypeSource(LoggingNamespaceSource parent, Type sourceType) : base(parent, sourceType)
    {
        Log = LogManager.GetLogger(sourceType);
    }

    public ILog Log { get; }

    protected override bool IsBackendEnabled(LogLevel level)
    {
        switch (level)
        {
            case LogLevel.Trace:
            case LogLevel.Debug:
                return Log.IsDebugEnabled;

            default:
                return true;
        }
    }
}
```

Override the `TextLogRecordBuilder` class

The role of the **TextLogRecordBuilder** is to create a string representing the current log record and finally to emit this string to the target logging framework.

The **TextLogRecordBuilder** class already contains the logic that creates the string. If you don't need to alter the string formatting logic, all you have to do is to implement the `Write(UnsafeString)` method.

If you need to customize the formatting of the string, or if you need to implement semantic logging, you will need to override other virtual methods of the **TextLogRecordBuilder** class. Please refer to the API reference for details.

Your implementation of the **TextLogRecordBuilder** class does not need to be thread-safe. All threads involved in logging have their own instance of the **TextLogRecordBuilder** class.

Example

```
using PostSharp.Patterns.Diagnostics;
using PostSharp.Patterns.Diagnostics.Backends;
using PostSharp.Patterns.Diagnostics.RecordBuilders;
using PostSharp.Patterns.Formatters;

namespace PostSharp.Samples.Logging.CustomBackend.ServiceStack
{
    public class ServiceStackLogRecordBuilder : TextLogRecordBuilder
    {
        public ServiceStackLogRecordBuilder(TextLoggingBackend backend) : base(backend)
        {
        }

        protected override void Write(UnsafeString message)
        {
            var log = ((ServiceStackLoggingTypeSource) TypeSource).Log;
            var messageString = message.ToString();

            switch (Level)
            {
                case LogLevel.None:
                    break;

                case LogLevel.Trace:
                case LogLevel.Debug:
                    if (Exception == null)
                        log.Debug(messageString);
                    else
                        log.Debug(messageString, Exception);
                    break;
            }
        }
    }
}
```

```

    case LogLevel.Info:
        if (Exception == null)
            log.Info(messageString);
        else
            log.Info(messageString, Exception);
        break;

    case LogLevel.Warning:
        if (Exception == null)
            log.Warn(messageString);
        else
            log.Warn(messageString, Exception);
        break;

    case LogLevel.Error:
        if (Exception == null)
            log.Error(messageString);
        else
            log.Error(messageString, Exception);
        break;

    case LogLevel.Critical:
        if (Exception == null)
            log.Fatal(messageString);
        else
            log.Fatal(messageString, Exception);
        break;
    }
}
}
}

```

Override the TextLoggingBackend class

The root of a logging back-end is the `TextLoggingBackend` class. You can consider this class as a factory type that instantiates other facilities needed to log with your custom logging back-end.

Since the `TextLoggingBackend` is an abstract class, you will have to implement several factory methods

To implement the back-end class:

1. Create a new class and derive it from the `TextLoggingBackend` class.
2. Add a get-only `Options` property and initialize it to a new instance of your `LoggingBackendOptions` class.
3. Implement the `GetTextBackendOptions` abstract method so that it returns the value of your `Options` property.
4. Implement the `CreateTypeSource(LoggingNamespaceSource, Type)` abstract method so that it returns a new instance of your `LoggingTypeSource` class.
5. Implement the `CreateRecordBuilder` abstract method so that it returns a new instance of your `LogRecord-Builder` class.

Example

```

using System;
using PostSharp.Patterns.Diagnostics;
using PostSharp.Patterns.Diagnostics.Backends;
using PostSharp.Patterns.Diagnostics.RecordBuilders;

namespace PostSharp.Samples.Logging.CustomBackend.ServiceStack
{
    public class ServiceStackLoggingBackend : TextLoggingBackend
    {
        public new ServiceStackLoggingBackendOptions Options { get; } = new ServiceStackLoggingBackendOptions();

        protected override LoggingTypeSource CreateTypeSource(LoggingNamespaceSource parent, Type type)
        {

```

```

        return new ServiceStackLoggingTypeSource(parent, type);
    }

    public override LogRecordBuilder CreateRecordBuilder()
    {
        return new ServiceStackLogRecordBuilder(this);
    }

    protected override TextLoggingBackendOptions GetTextBackendOptions()
    {
        return Options;
    }
}
}
}

```

Overriding context classes

Contexts are typically used to represent the execution of a logged method and a custom activity. There are typically two log records per context: the entry record and the exit record. The role of context classes is to expose or store pieces of information that are shared by several records of the same context. Unless you need to store more pieces of information in the context, you do not need to extend the context classes. Most back-end implementations do not extend context classes.

Context classes all derive from the `LoggingContext` class. The following table lists all context classes. If you choose to derive a context class, you will also need to override the corresponding factory method in your **LoggingBackend** class.

Context class	Factory method	Description
<code>SyncMethodLoggingContext</code>	<code>CreateSyncMethodContext(ThreadLoggingContext)</code>	Normal method (neither async neither iterator).
<code>AsyncMethodLoggingContext</code>	<code>CreateAsyncMethodContext</code>	async method.
<code>IteratorLoggingContext</code>	<code>CreateIteratorContext</code>	Iterator method.
<code>SyncCustomActivityLoggingContext</code>	<code>CreateSyncCustomActivityContext(ThreadLoggingContext)</code>	Synchronous custom activity.
<code>AsyncCustomActivityLoggingContext</code>	<code>CreateAsyncCustomActivityContext</code>	Asynchronous custom activity.
<code>ThreadLoggingContext</code>	<code>CreateThreadContext</code>	A special kind of context that represents the roots of the context tree and contains all thread-static fields.

Context class	Factory method	Description
EphemeralLoggingContext	CreateEphemeralContext(ThreadLoggingContext)	A degenerated kind of context used when a log record is emitted out of a context, for instance when an exception record is logged without the corresponding entry record.

CHAPTER 22

Handling Faults in the Logging Component

The last thing you want is to have your application to fail in production because of a failure of the logging feature. In order to prevent this from happening, PostSharp catches all exceptions that are thrown by the logging code and allows you to override the behavior.

This topic contains the following sections:

- [Default exception handling policy on page 183](#)
- [Customizing the exception handling logic on page 183](#)

Default exception handling policy

By default, when an exception occurs in the logging feature, the following happens:

1. A message is logged to the Meta role. By default, the Meta role uses the default `LoggingBackend`, so it is likely that the logging of the logging failure will fail. It is recommended to configure the Meta logging role separately by using the following code snippet:

```
LoggingServices.Roles[LoggingRoles.Meta].Backend = new MyBackend();
```

2. The whole `LoggingBackend` responsible for the failure is disabled, unless the reason of the failure is `LoggingBackend` is a defect in user code (typically an invalid use of the `Logger` API) and not in the logging component itself. This pattern is called a *circuit breaker*. You can enable the `LoggingBackend` again by setting the `IsEnabled` property to true.

CAUTION NOTE

When the logging exception happens during the logging of a user-code exception, the details of the logging exception are not available. The reason for this behavior is that user-code exceptions are logged from exception filters in order to preserve the call stack, and exception thrown in exception filters cannot be caught.

Customizing the exception handling logic

To override the default exception handling behavior:

1. Create a class that implements the `ILoggingExceptionHandler` interface.
2. Assign the `ExceptionHandler` property to an instance of your class.

The following code snippet implements an exception handler that throws an exception upon any failure. The `Initialize` method registers the exception handler.

```
class ThrowLoggingExceptionHandler : ILoggingExceptionHandler
{
    public void OnInternalException( LoggingExceptionInfo exceptionInfo )
    {
        throw new Exception("Internal logging exception.", exceptionInfo.Exception);
    }
}
```

Handling Faults in the Logging Component

```
public void OnInvalidUserCode( ref CallerInfo callerInfo, LoggingTypeSource source, string message, params object[] args
{
    throw new InvalidOperationException( string.Format( message, args ) );
}

public static void Initialize()
{
    LoggingServices.ExceptionHandler = new ThrowLoggingExceptionHandler();
}
}
```


CHAPTER 23

Licensing of PostSharp Logging

The way PostSharp Logging is licensed differs from the other components of PostSharp. Depending on the license available during the build of your project, either PostSharp Logging *Developer Edition* or PostSharp Logging *Standard Edition* is selected.

This topic contains the following sections:

- [PostSharp Logging Developer Edition on page 185](#)
- [PostSharp Logging Standard Edition on page 185](#)
- [Using PostSharp Logging on a Build Server on page 185](#)

PostSharp Logging Developer Edition

During a software development process, you often need to trace the project behavior in run-time. This is where you choose the PostSharp Logging Developer Edition. It allows you to use the logging pattern anywhere in your project. PostSharp Logging Developer Edition is free of charge and included in all PostSharp products, even the free PostSharp Essentials.

The limitation of this edition is that the logging messages will be emitted up to 24 hours after the build of your assembly. After this period of time, your application will keep working, but the logging messages will no longer be emitted.

PostSharp Logging will switch itself to Developer Edition in the following situations:

- When using PostSharp Essentials or another PostSharp product that does not include PostSharp Logging.
- When building your code on a build server, unless the license key has been added to *postsharp.config*.
- When building unmodified code on the machine of an unlicensed developer, using the feature described in [Sharing Source Code With Unlicensed Teams on page 80](#).

PostSharp Logging Standard Edition

PostSharp Logging Standard Edition does not have the limitations of the Developer Edition.

PostSharp Logging Standard Edition is a commercial product. It can be purchased separately or as a part of PostSharp Ultimate or PostSharp Enterprise. See our [web site](#)³⁹ for details regarding the PostSharp commercial offering.

Using PostSharp Logging on a Build Server

If your project uses PostSharp Logging and you need to build your project on a build server, you **must** configure a license key on the build server (regardless of the kind of license you are using). We recommend you add your license key to *postsharp.config* as described in [Deploying License Keys on page 75](#).

39. <https://www.postsharp.net/purchase>

CHAPTER 24

Upgrading Logging from an Earlier Version

This topic contains the following sections:

- [Update from PostSharp 6.0 or earlier on page 187](#)
- [Update from PostSharp 4.3 or earlier on page 187](#)

Update from PostSharp 6.0 or earlier

The API to deal with manual messages and custom activities (see [Adding Custom Log Records Manually on page 151](#)) has been completely revamped. The old API, based on the `Logger`, still works but is no longer recommended for new developments. However, there is no compelling reasons to replace it with the new `LogSource`-based API in existing developments.

For existing code, we suggest you selectively disable the obsolescence warning using a `#pragma` directive:

```
#pragma warning disable 618
    Logger.GetLogger().Write( LogLevel.Error, "Text" );
#pragma warning restore 618
```

Update from PostSharp 4.3 or earlier.

PostSharp Logging has been completely rewritten between PostSharp 4.3 and PostSharp 5.0. There are some significant breaking changes that you will need to take into account.

To migrate your logging from PostSharp 4.3 or earlier:

1. Uninstall all *PostSharp.Patterns.Diagnostics.** packages except *PostSharp.Patterns.Diagnostics* itself
2. Upgrade all PostSharp packages.
3. Add the package for your backend logging framework to the start-up project (instead of adding it to all projects prior to PostSharp 5.0) and configure it at run-time as described in [Adding Detailed Logging to your Solution on page 135](#).
4. Migrate the logging profiles to the new format as described in [Adding Detailed Logging to your Solution on page 135](#). Some options that used to be accessible from logging profiles are now run-time options and are described in [Customizing the Appearance of Log Records on page 147](#).

PART 5

Contracts

CHAPTER 25

Adding Contracts to Code

This section describes how to add a contract to a field, property, or parameter.

This topic contains the following sections:

- [Introduction on page 191](#)
- [Adding contracts on page 192](#)
- [Contract inheritance on page 192](#)

Introduction

Consider the following method which checks if a valid string has been passed in:

```
publicclass CustomerModel
{
    publicvoid SetFullName(string firstName, string lastName)
    {
        if(firstName == null)
            throw NullReferenceException();

        if(lastName == null)
            throw NullReferenceException();

        this.FullName = firstName + " " + lastName;
    }
}
```

In this example, checks have been added to ensure that both parameters contain a valid string. A better solution is to place the logic which performs this check into its own reusable class, especially such boilerplate logic is involved, and then reuse/invoke this class whenever the check needs to be performed.

PostSharp's Contract attributes do just that by moving such checks out of code and into parameter attributes. For example, PostSharp's `RequiredAttribute` contract could be used to simplify the example as follows:

```
publicclass CustomerModel
{
    publicvoid SetFullName([Required] string firstName, [Required] string lastName)
    {
        this.FullName = firstName + " " + lastName;
    }
}
```

In this example, the `RequiredAttribute` attribute performs the check for null, thus eliminating the need to write the boiler plate code for the check in line with other code.

A contract can also be applied to a property or field as shown in the following example:

```
publicclass CustomerModel
{
    [Required]
    public FirstName { get; set; }
}
```

Using a contract in a property ensures that the value being passed into set is validated before the logic (if any) for set is executed.

Similarly, a contract can be used directly on a field which will validate the value being assigned to the field:

```
publicclass CustomerModel
{
    [Required]
    privatestring mFirstName = "Not filled in yet";

    publicvoid SetFirstName(string firstName)
    {
        mFirstName = firstName;
    }
}
```

In this example, `firstName` will be validated by the `Required` contract before being assigned to `mFirstName`. Placing a contract on a field provides the added benefit of validating the field regardless of where it's set from.

Note that `PostSharp` also includes a number of built-in contracts which range from checks for null values to testing for valid phone numbers. You can also develop your own contracts with custom logic for your own types as described below.

There are two ways to add contracts:

Adding contracts

To add a contract to an element of code:

1. Add the `PostSharp.Patterns.Common` package to your project.
2. Import the `PostSharp.Patterns.Contracts` namespace into your file:

```
using PostSharp.Patterns.Contracts;
```

3. Add the contract custom attribute to the parameter, field, property, or return value. See the `PostSharp.Patterns.Contracts` namespace for a list of available contract attributes. For example:

```
[return: Required]
publicvoid SetFullName([Required] string firstName, [Required] string lastName)
```

This example will make sure that both the parameters and the return value are not null.

Contract inheritance

`PostSharp` ensures that any contracts which have been applied to an abstract, virtual, or interface method are inherited along with that method in derived classes, all without the need to re-specify the contract in the derived methods. This is shown in the following example:

```
publicinterface ICustomerModel
{
    void SetFullName([Required] string firstName, [Required] string lastName);
}

publicclass CustomerModel : ICustomerModel
{
    publicvoid SetFullName(string firstName, string lastName)
    {
        this.FullName = firstName + " " + lastName;
    }
}
```


Here `ICustomerModel.SetFullName` method specifies that the `firstName` and `lastName` parameters are required using the `RequiredAttribute` attribute. Since the `CustomerModel.SetFullName` method implements this method, these attributes will also be applied to its parameters.

NOTE

If the derived class exists in a separate assembly, that assembly must be processed by PostSharp and must reference the `PostSharp` and `PostSharp.Patterns.Model` package

CHAPTER 26

Creating Custom Contracts

Given the benefits that contracts provide over manually checking values and throwing exceptions in code, you will likely want to implement your own contracts to perform your own custom checks and handle your own custom types.

The following steps show how to implement a contract which throws an exception if a numeric parameter is zero:

To implement a contract throwing an exception if a numeric parameter is zero:

1. Use the following namespaces: `PostSharp.Aspects` and `PostSharp.Reflection`.
2. Derive a class from `LocationContractAttribute`:

```
publicclass NonZeroAttribute : LocationContractAttribute
{
    public NonZeroAttribute()
        : base()
    {
    }
}
```

3. Implement the `ILocationValidationAspect` interface in the new contract class which exposes the `ValidateValue(T, String, LocationKind, LocationValidationContext)` method. Note that this interface must be implemented for each type that is to be handled by the contract. In this example, the contract will handle both `int` and `uint`, so the interface is implemented for both integer types. If additional integer types were to be handled by this class (e.g. `long`), then additional implementations of `ILocationValidationAspect` would have to be added:

```
publicclass NonZeroAttribute : LocationContractAttribute, ILocationValidationAspect<int>, ILocationValidationAspect<uint>
{
    public NonZeroAttribute()
        : base()
    {
    }

    public Exception ValidateValue(intvalue, string locationName, LocationKind locationKind, LocationValidationContext context)
    {
        if (value == 0)
            returnnew ArgumentOutOfRangeException($"The value of {locationName} cannot be 0.");
        elsereturnnull;
    }

    public Exception ValidateValue(uintvalue, string locationName, LocationKind locationKind, LocationValidationContext context)
    {
        if (value == 0)
            returnnew ArgumentOutOfRangeException($"The value of {locationName} cannot be 0.");
        elsereturnnull;
    }
}
```

The **`ValidateValue(UTP, String, LocationKind)`** method takes in the value to test, the name of the parameter, property or field, and the usage (i.e. whether it's a parameter, property, or field). The method must throw an exception if a check fails, or null or if no exception is to be raised.

Creating Custom Contracts

With the contract now created it can be used. For example, the following methods, which calculate the modulus between two numbers, can use the contract defined above to ensure that neither of their input parameters are zero:

```
bool Mod([NonZero] int number, [NonZero] int dividend)
{
    return ((number % dividend) == 0);
}

bool Mod([NonZero] uint number, [NonZero] uint dividend)
{
    return ((number % dividend) == 0);
}
```

CHAPTER 27

Localizing Contract Error Messages

You can customize all texts of exceptions raised by built-in contract. This allows you to localize error messages into different languages.

Contracts use the `ContractLocalizedTextProvider` class to obtain the text of an error message. This class follows a simple chain of responsibilities pattern where each provider has a reference to the next provider in the chain. When a message is queried, the provider either returns a message or passes control to the next provider in the chain.

Each message is identified by a string identifier and can refer to 4 basic arguments and additional arguments specific to a message type. For general information about message arguments please see remarks section of `LocationContractAttribute`. To find the identifier of a particular message and its additional arguments, please see remarks section of contract classes in `PostSharp.Patterns.Contracts`.

This topic contains the following sections:

- [Localizing a built-in error message on page 197](#)
- [Localizing custom contracts on page 198](#)

Localizing a built-in error message

Following steps illustrate how to override an error message of a given contract:

To override a contract error message:

1. Declare a class that derives from `ContractLocalizedTextProvider` and implement the chain constructor.

```
public class CzechContractLocalizedTextProvider : ContractLocalizedTextProvider
{
    public CzechContractLocalizedTextProvider(ContractLocalizedTextProvider next)
        : base(next)
    {
    }
}
```

2. Implement the `GetMessage(String)` method. In the next code snippet, we show how to build a simple and efficient dictionary-based implementation.

```
publicclass CzechContractLocalizedTextProvider : ContractLocalizedTextProvider
{
    readonly Dictionary<string, string> messages = new Dictionary<string, string>
    {
        {RegularExpressionErrorMessage, "Hodnota {2} neodpovídá regulárnímu výrazu"};
    };

    public CzechContractLocalizedTextProvider(ContractLocalizedTextProvider next)
    : base(next)
    {
    }

    publicoverridestring GetMessage( string messageId )
    {
        if ( string.IsNullOrEmpty( messageId ) )
            thrownew ArgumentNullException("messageId");

        string message;
        if ( this.messages.TryGetValue( messageId, out message ) )
        {
            return message;
        }
        else
        {
            // Fall back to the default provider.
            returnbase.GetMessage( messageId );
        }
    }
}
```

NOTE

If you need to support several languages, you can make your implementation of the `GetMessage(String)` method depend on the value of the `CultureInfo.CurrentCulture` property. You can optionally store your error messages in a managed resource and use the `ResourceManager` class to access it and manage localization issues. The design of PostSharp Code Contracts is agnostic to these decisions.

3. In the beginning of an application, create a new instance of the provider and set the current provider as its successor.

```
publicstaticvoid Main()
{
    ContractServices.LocalizedTextProvider = new CzechContractLocalizedTextProvider(ContractServices.LocalizedTextProvider.Current);
    // ...
}
```

Localizing custom contracts

Once you have configured a text provider, you can use it to localize error messages of custom contracts. In the following procedure, we will localize the error message of the example contract described in [Creating Custom Contracts on page 195](#).

To localize a custom contract:

1. Edit the code contract class (`NonZeroAttribute` in our case) to call `ContractServices.ExceptionFactory.CreateException()` in the validation method(s), as shown in [Customizing Contract Exceptions on page 201](#). Use a unique `messageId` (e.g. "NonZeroErrorMessage") to identify the message text template.

2. Edit your implementation of the `LocalizedTextProvider` class and include the message for your custom contract:

```
privatereadonly Dictionary<string, string> messages = new Dictionary<string, string>
{
    {RegularExpressionErrorMessage, "Hodnota {2} neodpovídá regulárnímu výrazu '{4}'."},
    {"NonZeroErrorMessage", "Hodnota {2} nesmí být 0."}
};
```


CHAPTER 28

Customizing Contract Exceptions

PART 6

INotifyPropertyChanged

CHAPTER 29

Implementing INotifyPropertyChanged

This section shows how to make your class automatically implements the `INotifyPropertyChanged` interface `NotifyPropertyChangedAttribute` aspect.

This topic contains the following sections:

- [Adding the `NotifyPropertyChanged` aspect on page 205](#)
- [Consuming the `INotifyPropertyChanged` interface on page 206](#)

Adding the `NotifyPropertyChanged` aspect

To add `INotifyPropertyChanged` aspect:

1. Use NuGet Package Manager to add the `PostSharp.Patterns.Model` package to your project.
2. Import the `PostSharp.Patterns.Model` namespace into your file.
3. Add the `[NotifyPropertyChanged]` custom attribute to the class.

All properties of the class now fire the `PropertyChanged` event whenever they are changed. You can prevent a property from automatically firing the `PropertyChanged` event with `IgnoreAutoChangeNotificationAttribute`.

By using the Model Pattern Library to add `NotifyPropertyChangedAttribute` to your model classes you are able to eliminate all of the repetitive boilerplate coding tasks and code from the codebase.

NOTE

This procedure has added `NotifyPropertyChangedAttribute` to one class. If you need to implement `NotifyPropertyChangedAttribute` to many different classes in your codebase you will want to read about using aspect multicasting. See the section [Adding Aspects to Multiple Declarations on page 112](#).

Example

```
[NotifyPropertyChanged]
publicclass CustomerForEditing
{
    publicstring FirstName { get; set; }
    publicstring LastName { get; set; }

    publicstring FullName
    {
        get { returnstring.Format("{0} {1}", this.FirstName, this.LastName); }
    }
}
```

Consuming the INotifyPropertyChanged interface

Since the `INotifyPropertyChanged` interface is implemented by PostSharp at build time after the compiler has completed, the interface will neither be visible to Intellisense or other tools like Resharper, neither to the compiler. The same is true for the `PropertyChanged` event.

In many cases, this limitation does not matter because the interface is consumed from a framework (like WPF) that is not coupled with your project. However, in some situations, you may need to access the `INotifyPropertyChanged` interface.

There are two ways to access the `INotifyPropertyChanged` interface from your code:

- You can cast your object to `INotifyPropertyChanged`, for instance:

```
((INotifyPropertyChanged) obj).PropertyChanged += obj_OnPropertyChanged;
```

If your tooling complains that the object does not implement the interface, you can first cast to `object`:

```
((INotifyPropertyChanged) (object) obj).PropertyChanged += obj_OnPropertyChanged;
```

- You can use the `Post.Cast(TSource, TTarget(TSource))` method. The benefit of using this method is that the cast operation is validated by PostSharp, so the build will fail if you try to cast an object that does not implement the `INotifyPropertyChanged` interface. For instance:

```
Post.Cast<Foo, INotifyPropertyChanged>(obj).PropertyChanged += obj_OnPropertyChanged;
```

CHAPTER 30

Working with Properties that Depend on Other Objects

It's very common for the properties of one class to be dependent on the properties of another class. For example, a view-model layer will often contain a reference to a model object, and public properties which are, in turn, forwarded to the underlying properties of this referenced object. In this scenario, the view-model component's properties have a dependency on the referenced model's properties. Subsequently, the referenced model may also have properties which depend on the properties of other objects.

PostSharp easily handles transitive dependencies that follow the `this.field.Property.Property` form. Simply add the `NotifyPropertyChangedAttribute` class attribute to each class in the dependency chain. This will ensure that property change notifications are propagated up and down the dependency chain. PostSharp takes care of the rest and will even handle circular dependencies.

NOTE

Read the article [Handling Corner Cases on page 211](#) to learn about referencing more complex properties that do not follow the `this.field.Property.Property` form.

Example

In the following example, the `CustomerModel` class is used as a dependency of a `CustomerViewModel` class containing `FirstName` and `LastName` properties, both of which directly map to properties of the `CustomerModel` class, and a public read only property called `FullName`, which is calculated based on the value of the underlying customer's `FirstName` and `LastName` properties.

```
[NotifyPropertyChanged]
publicclass CustomerModel
{
    publicstring FirstName { get; set; }
    publicstring LastName { get; set; }
    publicstring Phone { get; set; }
    publicstring Mobile { get; set; }
    publicstring Email { get; set; }
}

[NotifyPropertyChanged]
class CustomerViewModel
{
    CustomerModel model;

    public CustomerViewModel(CustomerModel m)
    {
        this.model = m;
    }

    publicstring FirstName { get { returnthis.model.FirstName; } set { this.model.FirstName = value;}}
    publicstring LastName { get { returnthis.model.LastName; } set { this.model.LastName = value; }}
    publicstring FullName { get { returnstring.Format("{0} {1}", this.model.FirstName, this.model.LastName); } }
```

```
}
```

You now have a view-model class which can be used to bridge a view (e.g. an application's user interface) with the underlying data, and calls to get/set will be propagated across the chain of dependencies.

CHAPTER 31

Implementing INotifyPropertyChanging

By convention, the `PropertyChanged` event must be raised *after* the property value has changed. However, some components need to be signaled *before* the property value will be changed. This is the role of the `INotifyPropertyChanging` interface.

Because the `INotifyPropertyChanging` interface is not portable (in Xamarin, it is even a part of a different namespace), PostSharp cannot introduce it. However, if you implement the `INotifyPropertyChanging` interface yourself in your code, PostSharp will signal the `PropertyChanging` event. To make that work, you need to create an `OnPropertyChanging` method with the right signature.

To add the INotifyPropertyChanging interface to a class:

1. Make your class implement `INotifyPropertyChanging` and add the `PropertyChanging` event.
2. Add the `OnPropertyChanging` method with exactly the following signature, and invoke the `PropertyChanging` event.

```
protectedvoid OnPropertyChanging( string propertyName )
{
    if ( this.PropertyChanging != null )
    {
        this.PropertyChanging( this, new PropertyChangingEventArgs ( propertyName ) );
    }
}
```

3. Add the `NotifyPropertyChangedAttribute` aspect to your class as described in `INotifyPropertyChanged` on page 203.

NOTE

The contract between your class and the `NotifyPropertyChangedAttribute` is only the `OnPropertyChanging` method. As long as this method exists in the class, it will be invoked by the aspect before the value of a property changes.

Example

The following example demonstrates how to implement `INotifyPropertyChanging`.

```
[NotifyPropertyChanged]
publicclass MyClass : INotifyPropertyChanging
{
    publicevent PropertyChangedEventHandler PropertyChanging;

    protectedvoid OnPropertyChanging( string propertyName )
    {
        if ( this.PropertyChanging != null )
        {
            this.PropertyChanging( this, new PropertyChangingEventArgs ( propertyName ) );
        }
    }
}
```

Implementing INotifyPropertyChanged

```
    public string MyProperty { get; set; }  
}
```

CHAPTER 32

Handling Corner Cases

PostSharp includes a number of attributes for customizing the default behavior and for handling special dependencies.

This topic contains the following sections:

- [Ignoring changes to properties on page 211](#)
- [Handling virtual calls, delegates, external methods, or complex data flows on page 211](#)
- [Handling dependencies on pure methods on page 212](#)

Ignoring changes to properties

Use the `IgnoreAutoChangeNotificationAttribute` class attribute to prevent an `OnPropertyChanged` event from being invoked when setting a property.

Example

In this example, the `CustomerModel` class contains a `Country` property amongst others. To prevent a property notification from being invoked when the value of this property is set, simply place the `IgnoreAutoChangeNotificationAttribute` attribute above the property.

```
[NotifyPropertyChanged]
publicclass CustomerModel
{
    publicstring FirstName { get; set; }
    publicstring LastName { get; set; }
    publicstring Phone { get; set; }
    publicstring Mobile { get; set; }
    publicstring Email { get; set; }

    [IgnoreAutoChangeNotification]
    publicstring Country { get; set;}
}
```

Handling virtual calls, delegates, external methods, or complex data flows

If a property getter calls a virtual method from its class or a delegate, or references a property of another object (without using canonical form `this.field.Property`), PostSharp will generate an error because it cannot resolve such a dependency at build time. The same limitations apply when your property getter contains complex data flows, such as loops, or calls to methods (except property getters) of other classes.

When this happens, you can either refactor your code so that it can be automatically analyzed by PostSharp, or you can take over the responsibility for analyzing the code.

Taking responsibility for dependency analysis

To suppress the error that PostSharp emits when it is unable to fully analyze your code, add the `SafeForDependencyAnalysisAttribute` custom attribute to the property accessor (or in any method used by the property accessor).

NOTE

By using `SafeForDependencyAnalysisAttribute`, you are taking the responsibility that your code only has dependencies that are given either in the canonical form of `this.field.Property` either explicitly using the `DependsOn` construct (see below). If you are using this custom attribute but have non-canonical dependencies, some property changes may not be detected in which case no notification will be generated.

Adding dependencies manually

Even when a method has the `SafeForDependencyAnalysisAttribute` attribute, PostSharp still discovers the dependencies that are in canonical form `this.field.Property`. However, PostSharp does not discover dependencies hidden under delegate or virtual method calls.

To explicitly add a dependency to a property, you can use the `DependsOn` method. The expression passed to the `DependsOn` must be in canonical form `this.field.Property`, i.e. `Depends.On(this.field.Property)`.

Example

In the following example, the `CustomerModel` class contains a virtual method called `ValidateCountry` which is used by the get accessor of its `Country` property. The presence of the call to the virtual method prevents PostSharp from fully understanding the dependencies of the `Country` property. PostSharp discovers the dependency to the `_country` field but cannot analyze the implementations of the `ValidateCountry` method, and therefore emits an error. By adding the `SafeForDependencyAnalysisAttribute` attribute, to the `Country` property, you remove the error.

Even if you remove the error, PostSharp still analyzes the `Country` property getter and finds the dependency on the `_country` field. However, it does not follow the call to the `ValidateCountry` method and does not find the dependency to the `Continent` property. That is why we have to add this dependency manually by calling the `Depends.On` method.

```
[NotifyPropertyChanged]
publicclass Address
{
    string _country;
    publicstring Continent { get; set; }

    publicvirtualbool ValidateCountry(string country)
    {
        return GeoService.ContinentContains( this.Continent, country );
    }

    [SafeForDependencyAnalysisAttribute]
    publicstring Country
    {
        get
        {
            Depends.On( this.Continent );

            if(this.ValidateCountry(_country))
                return _country;
            elsereturn"Lilliput";
        }
        set;
    }
}
```

Handling dependencies on pure methods

Often times a property will depend on a method which is solely dependent on its input parameters to produce a return value. These methods are called *pure* and do not need to be analyzed. To mark a method as pure, use the `PureAttribute` custom attribute.

Example

Consider the following variation to `CustomerModel` where the `ValidPhoneNumber` property logic has been moved into a static method called `GetValidPhoneNumber()` which exists in a separate helper class called `ContactHelper`:

Since `GetValidPhoneNumber()` is a standalone method of another class, it is not analyzed. Therefore the `PureAttribute` attribute needs to be applied to this method to acknowledge this dependency.

```
publicclass ContactHelper
{
    [Pure]
    publicstaticstring GetValidPhoneNumber(string firstPhoneNumber, string secondPhoneNumber)
    {
        return firstPhoneNumber ?? secondPhoneNumber;
    }
}

[NotifyPropertyChanged]
publicclass CustomerModel
{
    public Contact PrimaryContact {get; set;}
    public Contact SecondaryContact {get; set;}

    publicstring ValidPhoneNumber
    {
        get {
            return ContactHelper.GetValidPhoneNumber(this.PrimaryContact.Phone, this.SecondaryContact.Phone);
        }
    }
}
```


CHAPTER 33

Integrating with UI Frameworks

There are frameworks that help you to create XAML applications including binding of objects in a simplified way. But usually, you still need to include some repetitive code anyway.

PostSharp can eliminate most of that repetition for you. All you have to do is make use of the `NotifyPropertyChangedAttribute` aspect.

In this chapter, you can find examples of using the `NotifyPropertyChangedAttribute` aspect with some of the popular UI frameworks.

This list of supported frameworks is not exhaustive. If the framework you are using is not listed here, it does not necessarily mean that PostSharp will not work with the framework.

In this chapter

Topic	Description
Caliburn.Micro on page 215	This section shows how to use the <code>NotifyPropertyChangedAttribute</code> aspect with the Caliburn.Micro framework.
MVVM Light on page 217	This section shows how to use the <code>NotifyPropertyChangedAttribute</code> aspect with the MVVM Light Toolkit.

33.1. Caliburn.Micro

Caliburn.Micro is a popular framework designed for building applications across all XAML platforms. Caliburn.Micro includes several features, and one of those is to simplify the implementation of the `INotifyPropertyChanged` interface. However, Caliburn.Micro still requires you to write a lot of boilerplate code. This article shows how to use PostSharp together with Caliburn.Micro, whether because you are upgrading an existing project that already uses this framework or because you want to use the other features of Caliburn.Micro.

Without PostSharp

The example below shows a View-Model implemented according to the Caliburn.Micro specification. The class `ShellViewModel` inherits from the class `PropertyChangedBase`, which implements the `INotifyPropertyChanged` interface and thus is already helping you to use the Notify Property Changed pattern.

On the other hand, there is still part of the boilerplate usually appearing within the `NotifyPropertyChanged` pattern. For each property, you must have an explicit field, and you must manually notify all dependencies and the property itself. The result is not only more code written, but also a big space for bugs because you must discover and maintain the chain of dependent properties yourself.

```
using Caliburn.Micro;
using System.Windows;

namespace CaliburnMicroWithPostSharp
```

```

{
    publicclass ShellViewModel : PropertyChangedBase
    {
        string name;

        publicstring Name
        {
            get { returnthis.name; }
            set
            {
                this.name = value;
                this.NotifyOfPropertyChange( () => this.Name );
                this.NotifyOfPropertyChange( () => this.CanSayHello );
            }
        }

        publicbool CanSayHello => !string.IsNullOrEmpty( this.Name );

        publicvoid SayHello()
        {
            MessageBox.Show( $"Hello {this.Name}!" );
        }
    }
}

```

With PostSharp

With PostSharp, you don't need to do any of that. You just indicate which classes should have the NotifyProperty-Changed pattern implemented using the `NotifyPropertyChangedAttribute` attribute and PostSharp does the hard work for you. You can see the difference in the second example, where the previous code got refactored keeping the same functionality.

```

using System.Windows;
using PostSharp.Patterns.Model;

namespace CaliburnMicroWithPostSharp
{
    [NotifyPropertyChanged]
    publicclass ShellViewModel : PropertyChangedBase
    {
        publicstring Name { get; set; }

        publicbool CanSayHello => !string.IsNullOrEmpty( this.Name );

        publicvoid SayHello()
        {
            MessageBox.Show( $"Hello {this.Name}!" );
        }
    }
}

```

Note that it is no longer necessary to derive your class from the `PropertyChangedBase` class. Even if you suppress the inheritance from `PropertyChangedBase` class, you can still use other Caliburn.Micro features in your code such as commands. However, if you do keep the inheritance from `PropertyChangedBase`, the `NotifyPropertyChangedAttribute` aspect will invoke the `NotifyOfPropertyChange` method of the `PropertyChangedBase` class, consistently with the coding practices of Caliburn.Micro.

33.2. MVVM Light

MVVM Light is a framework that helps you build XAML applications according to the Model-View-View-Model architectural pattern. MVVM Light includes several features, and one of those is to simplify the implementation of the `INotifyPropertyChanged` interface. However, MVVM Light still requires you to write a lot of boilerplate code. This article shows how to use PostSharp together with MVVM Light, whether because you are upgrading an existing project that already uses this framework or because you want to use the other features of MVVM Light.

Without PostSharp

In the example below, there is a View-Model implemented according to the MVVM Light Toolkit specification. The class `MainViewModel` inherits from the class `ViewModelBase`, which implements the `INotifyPropertyChanged` interface and thus is already helping you to use the Notify Property Changed pattern.

On the other hand, there is still part of the boilerplate usually appearing within the `NotifyPropertyChanged` pattern. For each property, you must have an explicit field, and you must manually notify all dependencies and the property itself. The result is not only more code written, but also a big space for bugs because you must discover and maintain the chain of dependent properties yourself.

```
using GalaSoft.MvvmLight;
using GalaSoft.MvvmLight.Command;
using System.Windows.Input;
using System.Windows;

namespace MvvmLightTest.ViewModel
{
    publicclass MainViewModel : ViewModelBase
    {
        int _exampleValue;

        publicint ExampleValue
        {
            get
            {
                return _exampleValue;
            }
            set
            {
                if (_exampleValue == value)
                    return;
                _exampleValue = value;
                RaisePropertyChanged("ExampleValue");
            }
        }
    }
}
```

With PostSharp

With PostSharp, you don't need to do any of that. You just indicate which classes should have the `NotifyPropertyChanged` pattern implemented using the `NotifyPropertyChangedAttribute` attribute and PostSharp does the hard work for you. You can see the difference in the second example, where the previous code got refactored keeping the same functionality.

```
using GalaSoft.MvvmLight.Command;
using System.Windows.Input;
using System.Windows;
using PostSharp.Patterns.Model;

namespace MvvmLightTest.ViewModel
{
    [NotifyPropertyChanged]
```

```
publicclass MainViewModel : ViewModelBase
{
    publicint ExampleValue { get; set; }
}
```

Note that it is no longer necessary to derive your class from the `ViewModelBase` class. Even if you suppress the inheritance from `ViewModelBase` class, you can still use other MVVM Light features in your code such as commands. However, if you do keep the inheritance from `ViewModelBase`, the `NotifyPropertyChangedAttribute` aspect will invoke the `RaisePropertyChanged` method of the `ViewModelBase` class, consistently with the coding practices of MVVM Light.

CHAPTER 34

Understanding the NotifyPropertyChanged Aspect

This section describes the principles and algorithm on which the `NotifyPropertyChangedAttribute` aspect is based. It helps developers and architects to understand the behavior and limitation of the aspect.

This topic contains the following sections:

- [Implementation of the `INotifyPropertyChanged` interface on page 219](#)
- [Instrumentation of fields on page 219](#)
- [Analysis of field-property dependencies on page 219](#)
- [Limitations on page 221](#)
- [Raising notifications on page 221](#)
- [Remarks on page 222](#)

Implementation of the `INotifyPropertyChanged` interface

The `NotifyPropertyChangedAttribute` aspect introduces the `INotifyPropertyChanged` interface to the target class unless the target class already implements the interface. Additionally, the aspect also introduces the **`OnPropertyChanged(String)`** method. The aspect always introduces this method as protected and virtual, so it can be overridden in derived classes.

If the target class already implements the `INotifyPropertyChanged` interface, the aspect requires the class to expose the **`OnPropertyChanged(String)`** method.

The aspect uses the **`OnPropertyChanged(String)`** to raise the `PropertyChanged` event. Thanks to this method, the aspect is able to raise the event even when the `INotifyPropertyChanged` is not implemented by the aspect. This mechanism also allows user code to raise notifications that are not automatically handled by the `NotifyPropertyChangedAttribute` aspect.

Instrumentation of fields

Although most implementations of the `INotifyPropertyChanged` interface rely on instrumenting the property setter, this strategy has severe limitations: it is unable to handle *composite properties*, which return a value based on several other fields or properties. Composite properties have no setter, rendering this strategy unusable.

Instead, the `NotifyPropertyChangedAttribute` aspect instruments all write operations to fields (for instance a `FullName` property appending `FirstName` and `LastName`). It analyzes dependencies between fields and properties and raises a change notification for any property affected by a change in this specific field.

All methods, and not just property setters, can make a change to a field and therefore cause the `PropertyChanged` event to be raised. Property setters do not have any specific status in the `NotifyPropertyChangedAttribute` implementation.

Analysis of field-property dependencies

In order to adequately raise the `PropertyChanged` event, the `NotifyPropertyChangedAttribute` aspect needs to know which properties are affected by a change of a class field. The field-property dependency map is created at build time

by analyzing the source code: the analyzer reads the getter of all properties and check for field references. The map is then serialized inside the assembly and used at run time to raise relevant events when a field has changed.

Dependencies on fields of the current object

Consider the following code snippet:

```
[NotifyPropertyChanged]
class Invoice
{
    private decimal _amount;
    private decimal _tax;

    public decimal Amount { get { return this._amount; } set { this._amount = value; } }
    public decimal Tax { get { return this._tax; } set { this._tax = value; } }

    public void Set( decimal amount, decimal tax )
    {
        this._amount = amount;
        this._tax = tax;
    }

    public decimal Total { get { return this._amount + this._tax; } }
}
```

The result of the analysis for the code snippet above would be the map { `_amount => (Amount, Total)`, `_tax => (Tax, Total)` }. Whenever the `_amount` field is changed, the `PropertyChanged` event will be raised for properties `Amount` and `Total`.

Automatic properties are processed as handwritten properties; in this case, the implicit backing field is taken into account for the dependency analysis.

Recursive analysis of the call graph

Field references are not only looked for in the getter, but in any method invoked from the getter, and recursively.

Consider the following code snippet:

```
[NotifyPropertyChanged]
class Invoice
{
    private decimal _amount;
    private decimal _exchangeRate;

    public decimal Amount { get { return this._amount; } set { this._amount = value; } }
    public decimal ExchangeRate { get { return this._exchangeRate; } set { this._exchangeRate = value; } }

    private decimal Convert( decimal amount )
    {
        return amount * this.ExchangeRate;
    }

    public int AmountBase { get { return this.Convert( this.Amount ); } }
}
```

In the code snippet above, the analyzer starts from the getter of the `AmountBase` property, follows the call to the `Amount` property getter, then call to the `AmountBase` method and recursively follows the `ExchangeRate` property getter. Therefore, the resulting property map remains { `_amount => (Amount, AmountBase)`, `_exchangeRate => (ExchangeRate, AmountBase)` }.

Dependencies on properties of external objects

The `NotifyPropertyChangedAttribute` aspect does not just handle dependencies between a property and a field of the same class. It also handles dependencies on properties of properties or properties of fields, and recursively. That is, it supports expressions of the form `_f.P1.P2.P3` where `_f` is a field or property and `P1`, `P2` and `P3` are properties.

Consider the following code snippet:

```
[NotifyPropertyChanged]
class InvoiceModel
{
    private decimal _amount;
    private decimal _tax;

    public decimal Amount { get { return this._amount; } set { this._amount = value; } }
    public decimal Tax { get { return this._tax; } set { this._tax = value; } }
}

[NotifyPropertyChanged]
class InvoiceViewModel
{
    InvoiceModel _model;

    public InvoiceModel Model { get { return this._model; } }

    public decimal Total { get { return this._model.Amount + this.Model.Tax; } }
}
```

In the example above, the `InvoiceViewModel.Total` property is dependent on properties `Amount` and `Tax` of the `_model` field. Therefore, changes in the `InvoiceModel._amount` field will trigger a change notification for the `InvoiceModel.Amount` and `InvoiceViewModel.Total` properties.

The `NotifyPropertyChangedAttribute` aspect automatically subscribes to the `PropertyChanged` event of the child object, and unsubscribes whenever the value of the field in the parent object (`_model` in our example) is modified. However, the parent object does not unsubscribe upon disposal because the `NotifyPropertyChangedAttribute` makes no assumption that the `IDisposable` interface has been implemented. Therefore, the implementation of the `INotifyPropertyChanged` of the external object must hold weak references to clients of the `PropertyChanged` event.

Recursive dependencies to external objects are handled thanks to an auxiliary interface named `INotifyChildPropertyChanged`. This interface is implemented by the `NotifyPropertyChangedAttribute` aspect. It is considered an implementation detail and cannot be implemented manually. Classes that do not implement the `INotifyChildPropertyChanged` interface can only participate as terminal dependencies, i.e. they can be leaves but not intermediate nodes.

Limitations

The design goal of the `NotifyPropertyChangedAttribute` aspect is to be able to handle the majority of use cases in real-world source code while requiring only an acceptable amount of compilation time. The dependency analysis algorithm imposes several limitations:

- Calls to virtual methods (other than through the `base` keyword), abstract methods, interface methods or delegates are not supported.
- Calls to static methods or methods of external classes are not supported unless they are decorated with the `PureAttribute` custom attribute, or unless the method is a property getter in a supported dependency chain.
- Valuations of properties method return values are not supported. Only properties of fields or properties are supported.
- Dependencies on properties of variables are not supported if the variable is assigned in a loop (`while`, `for`, ...) or in an exception handling block.

See [Handling Corner Cases on page 211](#) to learn how to cope with these limitations.

Raising notifications

Simplistic implementations of the `INotifyPropertyChanged` interface signal a change notification immediately after a property has been changed. However, this strategy may cause subtle errors in client code.

Consider the following code:

```
[NotifyPropertyChanged]
class Invoice
{
    publicdecimal Amount { get; private set; }
    publicdecimal Tax { get; private set; }
    publicdecimal Total { get; private set; }

    publicvoid Set( decimal amount, decimal tax )
    {
        /* 1 */this.Amount = amount;
        /* 2 */this.Tax = tax;
        /* 3 */this.Total = amount + tax;
    }
}
```

As a class invariant, the assumption `Total == Amount + Tax` should always be true.

However, suppose that the `PropertyChanged` event is raised immediately after the `Amount` property is set at line 1 of the `Set` method. Clearly, for a client subscribing to this event, the class invariant would be broken at this specific moment.

Therefore, it is not safe to raise change notifications immediately after a change has been achieved. It is necessary to wait until the object can be safely observed by external code, when all class invariants are valid again (i.e. when the object state is consistent). A common best practice in object-oriented programming is to ensure that class invariants are valid before the control flow goes back from the current object to the caller. Typically, it means that a private or protected method can exit with an inconsistent object state, but public and internal methods must guarantee that the object state is consistent upon exit.

The `NotifyPropertyChangedAttribute` aspect relies on this best practice and raises the property change notifications just before the control flow exits the current object, that is, just before the last public or internal method in the call stack for the current object exits.

Besides avoiding to expose invalid object state, this strategy also avoids the same property to be notified for change several times during the execution of a single public method, which a potentially great positive performance impact.

To solve this problem, `NotifyPropertyChangedAttribute` aspect uses the following strategy:

1. Instead of causing immediate change notifications, field changes are buffered into a thread-local storage named the *accumulator*.
2. Calls to public and methods are instrumented so the aspect can detect when the control flow exits the object. At this moment, the accumulator is flushed and all change notifications are triggered.

It is possible to flush the accumulator at any time by invoking the `NotifyPropertyChangedServicesRaiseEvents-Immediate(Object)` method.

You can suspend and resume notifications using the `NotifyPropertyChangedServicesSuspendEvents` and `Notify-Property-Changed-Services-Resume-Events` methods.

Remarks

The `NotifyPropertyChangedAttribute` aspect never evaluates property getters at run time. This decision is deliberate and aims at avoiding possible side-effects (lazy-initialization, logging, etc.). Therefore, it is possible that the algorithms emit false positives, i.e. change notifications for properties whose values did not actually change.

The algorithm heuristically detects dependency cycles. If a cycle is detected, an exception is thrown instead of allowing for an infinite update cycle.

All notifications are invoked on the thread on which the change is being made. The accumulator that buffers the changes is a thread-local storage.

CHAPTER 35

Suppressing False Positives

The `NotifyPropertyChangedAttribute` aspect, when applied to a class, raises the `PropertyChanged` event every time it detects a possible change of a property, even when the actual value of the property doesn't change. By default, the aspect doesn't keep track of the property values because that would require the aspect to invoke property getters arbitrarily outside of the developer's control. And when property getters have any side effects (lazy-initialization, logging, etc.), invoking them randomly is not a safe behavior.

In certain scenarios, such as rich client applications with many UI controls, redundant event notifications are not desired because they cause excessive UI updates and can degrade the application responsiveness. You can avoid these redundant event notifications by suppressing false positives in the `NotifyPropertyChangedAttribute` aspect.

This topic contains the following sections:

- [Example of a false positive on page 223](#)
- [How to suppress false positives on page 224](#)

Example of a false positive

The following `Calc` class has two integer fields `a` and `b`, and a property `Sum` that returns the sum of these two numbers. The `Main` method creates an instance of the `Calc` class and changes the fields from (`a=1, b=2`) to (`a=2, b=1`). There are two `PropertyChanged` event notifications shown in the output, even though the actual `Sum` value doesn't change in the second case.

```
[NotifyPropertyChanged]
class Calc
{
    private int a, b;

    public int Sum
    {
        get { return this.a + this.b; }
    }

    public void Update(int a1, int b1)
    {
        this.a = a1;
        this.b = b1;
    }
}

static void Main()
{
    Calc calc = new Calc();
    ((INotifyChildPropertyChanged) calc).PropertyChanged +=
        (sender, eventArgs) =>
        {
            Console.WriteLine("Property {0} changed. New value = {1}.",
                eventArgs.PropertyName,
                sender.GetType().GetProperty(eventArgs.PropertyName).GetValue(sender));
        };

    calc.Update(1, 2);
    calc.Update(2, 1);
}
```

Output:

```
Property Sum changed. Value = 3.  
Property Sum changed. Value = 3.
```

How to suppress false positives

To suppress false positive event notifications, set the `PreventFalsePositives` property to `true` when you apply the aspect to the target element.

```
[NotifyPropertyChanged(PreventFalsePositives = true)]  
class Calc  
{  
    // ...  
}
```

If you run the test code snippet again after this change, you can see that there's only one change notification now.

```
Property Sum changed. Value = 3.
```

CAUTION NOTE

Do not suppress false positive notifications in your class when the property getters in the class have side effects. In this case, reset the `PreventFalsePositives` property to its default value of `false`.

By enabling the `PreventFalsePositives` option of the `NotifyPropertyChangedAttribute` aspect you can reduce the number of events raised in your application and improve your UI responsiveness.

PART 7

Weak Event

PART 8

XAML

CHAPTER 36

Command

In the GUI applications built using MVVM pattern, it's common for the view model class to define commands that can be performed by the user in the associated XAML view. These commands are implemented as nested classes of the view model classes, and require a lot of boilerplate code.

The `CommandAttribute` aspect allows you to implement commands in your view model classes without adding nested classes, with greatly reduces boilerplate code. The aspect also integrates with the `NotifyPropertyChangedAttribute` aspect, which makes it much easier to implement the `CanExecute` logic.

This topic contains the following sections:

- [Creating a simple command on page 229](#)
- [Determining whether your command can be executed on page 229](#)
- [Overriding naming conventions on page 230](#)

Creating a simple command

To add a command named `LoadContactsList` to your view model class

1. Implement the command logic in the `ExecuteLoadContactsList` method in your view model class. This method will be the equivalent to the `ICommandExecute(Object)` method.

Optionally, the `Execute` method can have one parameter of any type, which becomes the command parameter.

2. Add a `LoadContactsListCommand` property to your view model class and mark it with the `CommandAttribute` attribute. The property must be of type `ICommand` and have both a getter and a setter.

Example

The following code snippet defines a dependency property named `LoadContactsList`.

```
[Command]
public ICommand LoadContactsListCommand { get; private set; }

public void ExecuteLoadContactsList()
{
    // ...
}
```

After adding a new command to your view model class you can bind it to a UI control in your XAML code.

```
<Button x:Name="LoadButton" Content="Load contacts" Command="{Binding LoadContactsListCommand}" HorizontalAlignment="Left" Marg
```

Determining whether your command can be executed

The UI controls in the GUI application can be enabled or disabled based on what actions are available to the user in the current state of the application. This feature is usually implemented by the `ICommandCanExecute(Object)` method.

The `CommandAttribute` aspect provides a clean way to determine the availability of the actions. For each command in your view model class, you can implement a `CanExecute` property or method that validates whether this command can be currently executed.

Adding a `CanExecute` property

To allow the view to determine the availability of the `CreateContact` command, simply add a `public bool` property named `CanExecuteCreateContact` to the same class.

```
[Command]
public ICommand CreateContactCommand { get; private set; }

public bool CanExecuteCreateContact
{
    get
    {
        bool canCreateContact = true;
        // ...return canCreateContact;
    }
}

public void ExecuteCreateContact()
{
    // ...
}
```

The `CanExecute` properties associated with the commands support the `NotifyPropertyChangedAttribute` aspect. Therefore, if you add the `NotifyPropertyChangedAttribute` aspect to the view model class, the UI will be notified of changes in the `CanExecute` properties.

Adding a `CanExecute` method

A limitation of the `CanExecute` is that they cannot depend on the input argument of the command. For example, with the `DeleteContact` command, you may want to validate whether the current user has the permission to delete the currently selected contact. If your availability logic depends on the input argument, you must use a `CanExecute` method instead of a property.

To allow the UI to determine the availability of the `DeleteContact` command, add a `CanExecuteDeleteContact` method to the view model class. The method must have one parameter and return a `bool` value. The parameter type must be assignable from the type of the command's input argument.

```
[Command]
public ICommand DeleteContactCommand { get; private set; }

public bool CanExecuteDeleteContact(int contactId)
{
    bool canDeleteContact = true;
    // ...return canDeleteContact;
}

public void ExecuteDeleteContact(int contactId)
{
    // ...
}
```

Overriding naming conventions

The `CommandAttribute` aspect follows a predefined naming convention when looking for methods and properties associated with the command in your view model class. You can override the naming convention and choose your own member names by setting properties on the `CommandAttribute`. The following table shows the default naming convention and the properties used to override member names.

Member kind	Default name	Example	Override property
Command property	<i>CommandName</i> -or- <i>CommandNameCommand</i>	UpdateContact -or- UpdateContactCommand	N/A
Execute method	<i>ExecuteCommandName</i>	ExecuteUpdateContact	ExecuteMethod
CanExecute method	<i>CanExecuteCommandName</i>	CanExecuteUpdateContact	CanExecuteMethod
CanExecute property	<i>CanExecuteCommandName</i>	CanExecuteUpdateContact	CanExecuteProperty

The following examples shows a command that does not respect naming conventions.

```
[Command(ExecuteMethod = nameof(UpdateContact), CanExecuteMethod = nameof(CanUpdate))]
public ICommand UpdateContactCommand { get; private set; }

public bool CanUpdate(object newData)
{
    bool canUpdate = true;
    // ...return canUpdate;
}

public void UpdateContact(object newData)
{
    // ...
}
```


CHAPTER 37

Dependency Property

XAML dependency properties⁴⁰ extend the functionality of the CLR properties with features such as data binding, styling, animation, etc. Whenever you need to define a dependency property in your XAML application, you typically have to follow a strict implementation pattern and write a fair amount of boilerplate code. The `DependencyPropertyAttribute` aspect allows you to create custom dependency properties much faster, without writing repetitive code.

This topic contains the following sections:

- [Creating a simple dependency property on page 233](#)
- [Exposing the `DependencyProperty` object on page 233](#)
- [Validating the value of the dependency property with contracts on page 234](#)
- [Validating the value of the dependency property with a validation method on page 234](#)
- [Reacting to the changes of the dependency property value on page 235](#)
- [Implementing `INotifyPropertyChanged` on page 235](#)
- [Overriding the naming conventions on page 235](#)

Creating a simple dependency property

To add a new dependency property to your class

1. Add a new property to your class with a chosen name, type and a public getter and setter.
2. Mark your new property with the `DependencyPropertyAttribute` attribute.

After you have marked your property with the `DependencyPropertyAttribute` attribute, you can immediately start using advanced features in your XAML code.

Example

The following code snippet comes from a custom control named `ContactCard`. The snippet defines a dependency property named `Phone`. The custom control defines other dependency properties not listed here.

```
[DependencyProperty]
publicstring Phone { get; set; }
```

The dependency properties can be used in XAML for advanced features. In the following code snippet, we're using data binding with the dependency properties of the `ContactCard` custom control.

```
<local:ContactCardFullName="{Binding ContactName}"Phone="{Binding ContactPhone}"Email="{Binding ContactEmail}"Notes="{Bind
```

Exposing the `DependencyProperty` object

If you want to manipulate the dependency property using the XAML API in C# or VB, then you need to get the `DependencyProperty` for this specific property. One solution is to use the `GetDependencyProperty(Type, String)` method, but this is long and error-prone.

40. <https://docs.microsoft.com/en-us/dotnet/framework/wpf/advanced/dependency-properties-overview>

A better solution is to add a `public static` property of type `DependencyProperty` with the same name as your dependency property, but the `Property` suffix. This property will be picked by the `DependencyPropertyAttribute` aspect and set to the proper `DependencyProperty` value.

Example

The following example shows how to expose the `DependencyProperty` for the `Phone` property.

```
[DependencyProperty]
publicstring Phone { get; set; }

publicstatic DependencyProperty PhoneProperty { get; private set; }

this.CurrentContactCard.SetBinding(ContactCard.PhoneProperty, new Binding("ContactPhone"));
```

Validating the value of the dependency property with contracts

PostSharp Code Contracts (see [Contracts on page 189](#)) provide a convenient way to validate the values of the dependency properties. To add the validation to your dependency property, you just need to apply a contract attribute to that property.

Example

The following code snippet shows how to validate a dependency property using a code contract.

```
[DependencyProperty]
[NotEmpty]
publicstring FullName { get; set; }
```

Validating the value of the dependency property with a validation method

If you need more complex validation for your dependency property, you can implement it in a dedicated validation method. To define a validation method for the `Email` dependency property, add a new method named `ValidateEmail` to the same class where the property is declared. The method must accept one argument with the type assignable from the property type and return a `bool` value.

The following list shows the method signatures you can use when implementing the validation method where `TPropertyType` is the type of the dependency property and `TDeclaringType` is the class where your property is declared.

- `static bool ValidatePropertyName(TPropertyType value)`
- `static bool ValidatePropertyName(DependencyProperty property, TPropertyType value)`
- `static bool ValidatePropertyName(TDeclaringType instance, TPropertyType value)`
- `static bool ValidatePropertyName(DependencyProperty property, TDeclaringType instance, TPropertyType value)`
- `bool ValidatePropertyName(TPropertyType value)`
- `bool ValidatePropertyName(DependencyProperty property, TPropertyType value)`

Example

The following code snippet shows how to validate a dependency property using a validation method.

```
[DependencyProperty]
publicstring Email { get; set; }

privatebool ValidateEmail(stringvalue)
{
    return EmailRegex.IsMatch(value);
}
```

Reacting to the changes of the dependency property value

The WPF property system can automatically notify you about the dependency property value changes via callback methods. This can be useful when, for example, you need to update the visual presentation of your custom UI control in response to a change of its property. This section shows how you can define a property change callback method with the PostSharp's dependency property pattern.

To define a property change callback method for the `PictureUr1` dependency property, add a new method named `OnPictureUr1Changed` to the same class where the property is declared. The method doesn't have to accept any arguments and must have a `void` return type. Implement your property change handling logic inside this new method.

The following list shows the method signatures you can use when implementing the property change callback method. `TDeclaringType` is the class where your property is declared.

- `static void OnPropertyNameChanged()`
- `static void OnPropertyNameChanged(DependencyProperty property)`
- `static void OnPropertyNameChanged(TDeclaringType instance)`
- `static void OnPropertyNameChanged(DependencyProperty property, TDeclaringType instance)`
- `void OnPropertyNameChanged()`
- `void OnPropertyNameChanged(DependencyProperty property)`

Example

```
[DependencyProperty]
publicstring PictureUr1 { get; set; }

privatevoid OnPictureUr1Changed()
{
    this.ProfileImage.Source = this.LoadImageFromUr1(this.PictureUr1);
}
```

Implementing INotifyPropertyChanged

You may also want to notify the users of your class when a dependency property value changes. In this case, you would normally need to implement the `INotifyPropertyChanged` interface in your class and raise the `PropertyChanged` event. PostSharp helps you to automate this task using [INotifyPropertyChanged on page 203](#) pattern. To raise the `PropertyChanged` event every time any of the dependency properties in your class changes its value, mark your class with the `NotifyPropertyChangedAttribute` attribute.

```
[NotifyPropertyChanged]
publicpartialclass ContactCard : UserControl
{
    // ...
}
```

Overriding the naming conventions

The `DependencyPropertyAttribute` aspect follows a predefined naming convention when looking for methods and properties associated with the dependency property in your class. You can override the naming convention and choose your own member names by setting properties on the `DependencyPropertyAttribute`. The following table shows the default naming convention and the properties used to override member names.

Member kind	Default name	Example	Override property
Value validation method	<code>ValidatePropertyName</code>	<code>ValidatePhoneNumber</code>	ValidateValueMethod
Property changed callback method	<code>OnPropertyNameChanged</code>	<code>OnPhoneNumberChanged</code>	PropertyChangedMethod

Member kind	Default name	Example	Override property
Registration property	<i>PropertyNameProperty</i>	PhoneNumberProperty	RegistrationProperty

Example

```
[DependencyProperty(ValidateValueMethod = "ValidateStringMaxLength" )]
publicstring Notes { get; set; }

privatebool ValidateStringMaxLength(stringvalue)
{
    if (string.IsNullOrEmpty(value))
        returntrue;

    returnvalue.Length <= MAX_LENGTH;
}
```

CHAPTER 38

Attached Property

In XAML, [Attached properties](#)⁴¹ are a special kind of properties that are settable on a different object than the one that exposes it. An example of attached property is `DockPanel.Dock`. Attached properties are a special kind of dependency properties and thus require the same amount of boilerplate code to be written.

The `AttachedPropertyAttribute` aspect allows you to automate the implementation of attached properties in your classes and eliminates the boilerplate.

This topic contains the following sections:

- [Creating a simple attached property on page 237](#)
- [Using the dependency property on page 237](#)

The `AttachedPropertyAttribute` aspect is a special case of the `DependencyPropertyAttribute` aspect. Therefore you can refer to the [Dependency Property on page 233](#) article for additional documentation on topics such as validation, property change callbacks, and naming conventions.

Creating a simple attached property

To add a new attached property to your class

1. Add a new static property to your class with a chosen name, a public getter and private setter. Declare the property type as `Attached<T>` where T is the desired type of your attached property (see `AttachedT`).
2. Mark your new property with the `AttachedPropertyAttribute` attribute.

```
[AttachedProperty]
publicstatic Attached<Dock> Dock { get; set; }
```

Using the dependency property

After you have marked your property with the `AttachedPropertyAttribute` attribute, you can directly start to use it in XAML. For example, you can set the value of the property on any object using the attached property syntax in XAML.

```
<MyDockPanel><CheckBoxMyDockPanel.Dock="Top">Hello</CheckBox></MyDockPanel>
```

You can also read and write the property value on any object in C# by using the `Attached.GetValue(DependencyObject)` and `Attached.SetValue(DependencyObject, UTP)` methods.

```
MyDockPanel.Dock.SetValue(this.HelloCheckBox, Dock.Top);
```

To get a `DependencyProperty` instance that is backing your attached property, read the value of the `AttachedTRuntimeProperty` property.

```
MyDockPanel.Dock.RuntimeProperty.OverrideMetadata( typeof(MyPropertiesListView), new PropertyMetadata(Dock.Right) );
```

41. <https://docs.microsoft.com/en-us/dotnet/framework/wpf/advanced/attached-properties-overview>

PART 9

Parent/Child, Visitor and Disposable

CHAPTER 39

Annotating an Object Model for Parent/Child Relationships (Aggregatable)

PostSharp provides several custom attributes that you can apply to your object model to describe the parent-child relationships in a natural and concise way. The `AggregatableAttribute` aspect is applied to the object model classes, and the properties are marked with `ChildAttribute`, `ReferenceAttribute` and `ParentAttribute` custom attributes. You can also use `AdvisableCollectionT` and `AdvisableDictionaryTKey, TValue` classes to make your collection properties aware of the `Aggregatable` pattern.

Below you can find a detailed walkthrough on how to add parent-child relationships implementation into existing object models.

To apply the `Aggregatable` to an object model:

1. Add the `PostSharp.Patterns.Model` package to your project.
2. Add the `AggregatableAttribute` aspect to all base classes. Note that the aspect is inherited, so it is not necessary to explicitly add the aspect to classes that derive from a class that already has the aspect. Note also that `AggregatableAttribute` aspect is implicitly added by other aspects, including all threading models (`ThreadAwareAttribute`), `DisposableAttribute` and `RecordableAttribute`.

NOTE

It is not strictly necessary to add the `AggregatableAttribute` aspect to a class whose instances will be children but not parents unless you want to track the relationship to the parent using the `IAggregatableParent` property or the `ParentAttribute` custom attribute in this class (see below).

3. Annotate fields and automatic properties of all aggregatable classes with the `ChildAttribute` or `ReferenceAttribute` custom attribute. Fields or properties of a value type must not be annotated.
4. Collections require special attention:
 - Modify the code to use `AdvisableCollectionT`, `AdvisableDictionaryTKey, TValue` or `AdvisableKeyedCollectionTKey, TItem` instead of standard .NET collections for children fields. This change is necessary because all objects assigned to children fields/properties must be aware of the `Aggregatable` pattern.
 - By default, PostSharp considers that items of child collections will be child objects of the collection. If the collection itself is a child but its items should be considered as references, set the `ChildAttributeItemsRelationship` property to `RelationshipKind.Reference`.

See [Working With Child Collections on page 249](#) for details.

- Optionally, add a field or property to link back from the child object to the parent, and add the `ParentAttribute` to this field/property. PostSharp will automatically update this field or property to make sure it refers to the parent object.

TIP

For better encapsulation, setters of parent properties should have `private` visibility. In case of parent fields, the `private` visibility is preferred. User code should not manually set a parent field or property.

Example

In the following examples, an `Invoice` object owns several instances of the `InvoiceLine` class, therefore both classes must be annotated with `AggregatableAttribute`. However, the `Invoice` does not own the `Customer` to which it is associated, so the `Customer` class does not need the custom attribute.

Note that in the constructor of the `Invoice` class, we assign an `AdvisableCollectionT` to the `Lines` field instead of a `List`.

```
[Aggregatable]
publicclass Invoice
{
    public Invoice()
    {
        this.Lines = new AdvisableCollection<InvoiceLine>();
    }

    [Reference]
    public Customer Customer { get; set; }

    [Child]
    public IList<InvoiceLine> Lines { get; privateset; }

    [Child]
    public Address DeliveryAddress { get; set; }
}

[Aggregatable]
publicclass InvoiceLine
{
    [Reference]
    private Product product;

    publicdecimal Amount { get; set; }

    [Parent]
    public Invoice ParentInvoice { get; privateset; }
}

[Aggregatable]
publicclass Address
{
}

publicclass Customer
{
}
```

39.1. Rule-Based Annotation

There are times where you cannot or don't want to add custom attributes manually.

For instance, you cannot add any custom attribute to an auto-generated field backing a XAML element. The WPF designer generates C# code from your XAML files and stores them in *i.g.cs* files hidden in the *obj* folder. You cannot modify these files directly.

Another scenario is when you have a large amount of fields and don't want to annotate each of them individually.

Field rules allow you to annotate a field as a child or a reference programmatically, without adding a custom attribute to each field manually.

In the following example, let's consider a class where PostSharp shows an error message *COM002: "Field/property InvoiceLine.product must be annotated with a custom attribute [Child], [Reference] or [Parent]."*

```
[Aggregatable]
publicclass InvoiceLine
{
    private Product product;

    publicdecimal Amount { get; set; }
}
```

To automatically mark all fields as references by default:

1. Create a class inherited from the `FieldRule` class.

```
publicclass ReferenceFieldRule : FieldRule
```

2. Override the `GetRelationshipInfo(FieldInfo)` method. PostSharp calls the `GetRelationshipInfo(FieldInfo)` method for each field that is not annotated with the `ChildAttribute`, `ReferenceAttribute` or `ParentAttribute` custom attribute. The `GetRelationshipInfo(FieldInfo)` method allows you to specify the field relationship by returning a `RelationshipInfo` instance.

```
publicoverride RelationshipInfo? GetRelationshipInfo(FieldInfo field)
{
    returnnew RelationshipInfo(RelationshipKind.None, RelationshipKind.Reference);
}
```

3. Decorate your project's assembly with a `RegisterFieldRuleAttribute` custom attribute to activate your field rule.

```
[assembly: RegisterFieldRule(typeof(ReferenceFieldRule))]
```

CAUTION NOTE

You have to mark each project assembly to make your `FieldRule` active in the whole solution.

Now PostSharp considers all fields without any Parent-Child annotation as a reference and doesn't show any error for the product field.

NOTE

PostSharp has two built-in rules: one rule for auto-generated WinForms fields and one rule for auto-generated XAML fields.

```
<Windowx:Class="WpfApp.MainWindow"xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"xmlns:x="http://schemas.microsoft.com/wi
</Window>
```

In this example, the WPF designer generates a `myButton` field to a `MainWindow` class. Both the `myButton` field and the `MainWindow` class are `Control`⁴². The built-in rule annotates the `myButton` field as a reference automatically.

42. <https://msdn.microsoft.com/en-us/library/system.windows.controls.control.aspx>

CHAPTER 40

Enumerating Child Objects (Visitor)

After you have declared the structure of your object graph on page 241, you will want to make use of it.

Both the `ChildAttribute` and `ParentAttribute` can be used to declare parent-child relationships for other patterns such as Undo/Redo (`RecordableAttribute`) or threading models (`ImmutableAttribute`, `FreezableAttribute`, ...).

You can also use the `Aggregatable` pattern from your own code. The functionalities of this pattern are exposed by the `IAggregatable` interface, which all aggregatable object automatically implement. This interface allows you to execute a `Visitor` method against all child objects of a parent.

In the following example, we see how to implement recursive validation for an object model. We will assume that the `InvoiceLine` and `Address` line implement an `IValidatable` interface.

To enumerate all child objects of a parent:

1. Cast the parent object to the `IAggregatable` interface.

```
var invoice = new Invoice();
IAggregatable aggregatable = (IAggregatable) invoice;
```

NOTE

The `IAggregatable` interface will be injected into the `Invoice` class *after* compilation. Tools that are not aware of PostSharp may incorrectly report that the `Invoice` class does not implement the `IAggregatable` interface. Instead of using the cast operator, you can also use the `CastTSource, TTarget(TSource)` method. This method is faster and safer than the cast operator because it is verified and compiled by PostSharp at build time.

NOTE

If you are attempting to access `IAggregatable` members on either `AdvisableCollectionT` or `AdvisableDictionaryTKey, TValue` you will not be able to use the cast operator or the `CastTSource, TTarget(TSource)` method. Instead, you will have to use the **`QueryInterface`1(Object, Boolean)`** extension method.

2. Invoke the `VisitChildren(ChildVisitor, ChildVisitorOptions, Object)` method and pass a delegate to the method to be executed.

```
var invoice = new Invoice();
IAgregatable aggregatable = invoice.QueryInterface<IAgregatable>();
int errors = 0;
bool isValid = aggregatable.VisitChildren( (child, childInfo) =>
    {
        var validatable = child as IValidatable;
        if (validatable != null)
        {
            if ( !validatable.Validate() )
                errors++;
        }
        return true;
    });
```

NOTE

The visitor must return a true to continue the enumeration and false to stop the enumeration.

CHAPTER 41

Automatically Disposing Children Objects (Disposable)

When you are working with hierarchies of objects, you sometimes run into situations where you need to properly dispose of an object. Not only will you need to dispose of that object, but you likely will need to walk the object tree and recursively dispose of children of that object. To do this, we typically implement the `IDisposable` pattern and manually code the steps required to shut down the desired objects, and call the `Dispose` method on other children objects. This cascading of disposals takes a lot of effort and it is prone to mistakes and omissions.

The `DisposableAttribute` aspect relies on the `AggregatableAttribute` aspect and, as a result, is able to make use of the `VisitChildren(ChildVisitor, ChildVisitorOptions, Object)` method to cascade disposals through child objects.

This topic contains the following sections:

- [Disposing of object trees on page 247](#)
- [Customizing the Dispose logic on page 248](#)

Disposing of object trees

To automatically implement the `IDisposable` interface:

1. Add a reference to the `PostSharp.Patterns.Model` package.
2. On the top level object add the `DisposableAttribute`.
3. Annotate the object model as described in [Parent/Child, Visitor and Disposable on page 239](#).

NOTE

Fields that are marked as children but are assigned to an object that does not implement `IDisposable` (either manually or through `DisposableAttribute`) will simply be ignored during disposal.

NOTE

Items of child collections will be automatically disposed of as well unless items of child collections are considered as references. See [Working With Child Collections on page 249](#) for details.

Example

In this example, the `HomeMadeLogger` class has two fields, `_stream` and `_textWriter`, which should also be disposed of when the `HomeMadeLogger` is disposed of.

```
[Disposable]
publicclass HomeMadeLogger
{
    [Child]
    private TextWriter _textWriter;
    [Child]
```

Automatically Disposing Children Objects (Disposable)

```
private Stream _stream;
[Reference]
private MessageFormatter _formatter;

public HomeMadeLogger(MessageFormatter formatter)
{
    _formatter = formatter;
    _stream = new FileStream("our.log", FileMode.Append);
    _textWriter = new StreamWriter(_stream);
}

public void Debug(string message)
{
    _textWriter.WriteLine(_formatter.Format(message));
}
}
```

The `_stream` and `_textWriter` child objects will now have their `Dispose()` method called automatically when the `HomeMadeLogger` is disposed of. Since both the `_stream` and `_textWriter` objects are framework types that already implement `IDisposable`, adding the `DisposableAttribute` aspect to those object types is not necessary.

Customizing the Dispose logic

There will be times when you have objects that need custom disposal logic. At the same time, you may want to implement a parent child relationship and make use of the `DisposableAttribute`. PostSharp allows you to combine custom and automatic logic.

To add your own logic to the `Dispose` method, create a method with exactly the following signature:

```
protected virtual void Dispose( bool disposing )
{
}
```

CAUTION NOTE

The `DisposableAttribute` aspect does not automatically dispose of the object when it is garbage collected. That is, the aspect does not implement a destructor. If you need a destructor, you have to do it manually and invoke the `Dispose`.

Example

In the following example, we are customizing the `Dispose` pattern to expose the `IsDisposed` property:

```
[Disposable]
public class HomeMadeLogger
{
    public bool IsDisposed { get; private set; }

    protected virtual void Dispose( bool disposing )
    {
        this.IsDisposed = true;
    }
}
```

Once you have done this, PostSharp will properly run your custom `Dispose` logic as well as running any of the parent and child implementations of the `DisposableAttribute` that exist for the object.

CHAPTER 42

Working With Child Collections

It would not be possible to implement the Aggregatable pattern without support for collection classes. However, collections of the .NET base class libraries cannot be reliably extended to support the Aggregatable pattern. Therefore, code that implements the Aggregatable pattern must rely on collection classes defined by PostSharp, namely `AdvisableCollectionT`, `AdvisableDictionaryTKey`, `TValue`, `AdvisableKeyedCollectionTKey`, `TItem` and `AdvisableHashSetT`.

This topic contains the following sections:

- [Why yet other collection types? on page 249](#)
- [Replacing standard collections with advisable collections on page 250](#)
- [Casting advisable collections on page 250](#)
- [Parent surrogates on page 251](#)
- [Enumerating children and parent surrogates on page 252](#)
- [Collections of references on page 252](#)
- [Using immutable collections on page 252](#)

Why yet other collection types?

In the following example, an Invoice entity is composed of one instance of the Invoice class and several instances of the InvoiceLine class. The relationship between the Invoice and InvoiceLine classes is implemented using a collection.

```
[Aggregatable]
publicclass Invoice
{
    public Invoice()
    {
        this.Lines = new List<InvoiceLine>();
    }

    [Child]
    public IList<InvoiceLine> Lines { get; private set; }
}

[Aggregatable]
publicclass InvoiceLine
{
}
```

When we add a new element to the Lines collection, we also need to update the parent-child relationship between the corresponding invoice and invoice line. It is not possible to do this with the standard `ListT` class, so we need to build a specialized aggregatable collection class instead. However, we may later decide to apply another pattern to our object model, such as a threading model or undo/redo. This new pattern would, in turn, require support from the collection class. Creating new collection classes for each pattern (and potentially for each pattern combination) is clearly unmanageable.

Instead of providing a new collection class for each specific behavior we need to inject, PostSharp introduces the concept of *advisable collections*. Advisable collections are collection classes into which PostSharp can inject behavior dynamically, at run time, according to the field to which they are assigned. Advisable collections are a way to make the collection "inherit" the pattern of the parent class

Let's modify our previous example to work correctly with the `Aggregatable` aspect.

```
[Aggregatable]
public class Invoice
{
    public Invoice()
    {
        this.Lines = new AdvisableCollection<InvoiceLine>();
    }

    [Child]
    public IList<InvoiceLine> Lines { get; private set; }
}

[Aggregatable]
public class InvoiceLine
{
}
```

As you can see, the only change we made is using `AdvisableCollectionT` class instead of `ListT`. The `Aggregatable` aspect applied to the `Invoice` class detects that the child property is an advisable collection and applies dynamic `Aggregatable` advice to the collection instance at run time. This turns our collection of invoice lines into an aggregatable collection. If we apply another aspect to the `Invoice` class later, it can add new behaviors to this collection in the same way.

Replacing standard collections with advisable collections

The `PostSharp.Patterns.Collections` namespace defines advisable collection classes that are highly compatible with the collection types of the .NET base class libraries.

The following table shows how advisable collections map to standard collections.

Advisable collection	Replacement for
<code>AdvisableCollectionT</code>	<code>Array</code> , <code>ListT</code> , <code>CollectionT</code> , <code>ObservableCollectionT</code>
<code>AdvisableDictionaryTKey, TValue</code>	<code>DictionaryTKey, TValue</code>
<code>AdvisableKeyedCollectionTKey, TItem</code>	<code>KeyedCollectionTKey, TItem</code>

CAUTION NOTE

Interfaces `IReadOnlyListT` and `IReadOnlyCollectionT` are not implemented.

Casting advisable collections

Patterns such as `Aggregatable`, `Recordable` or `Threading Models` dynamically inject advices into advisable collections. These advices typically expose an interface, respectively `IAggregatable`, `IRecordable` and `IThreadAware`. Because interfaces are introduced at run-time and not at build-time, you cannot use the normal type casting constructs to access the interface members.

Instead of a normal cast, you can use the **QueryInterface`1(Object, Boolean)** extension method to access interfaces implemented by the given instance. This method will return the proper interface implementation irrespective how the interface is implemented: directly in the source code, introduced by PostSharp aspect at build time, or added dynamically at run time.

The following code snippet gets the `IAggregatable` interface of the `Lines` collection in the example above:

```
IAggregatable aggregatable = invoice.Lines.QueryInterface<IAggregatable>();
```

By default, the **QueryInterface`1(Object, Boolean)** method throws `InvalidCastException` if the given instance doesn't implement the queried interface. You can also safely check whether the interface is implemented by passing `false` as a method argument.

```
if ( collection.QueryInterface<IAgregatable>( false ) != null )
{
}
```

Parent surrogates

Collections play a special role in implementing the parent-child relationships between classes. Collections are often instruments instead of first-class entities of the object model. When enumerating children of a class, one generally wants to avoid the collections themselves to be returned, but only items of these collections. Additionally, the `Parent` property of a child object should typically refer to the parent entity and not to the collection that contains the child.

Consider the following example:

```
[Agregatable]
publicclass Invoice
{
    public Invoice()
    {
        this.Lines = new AdvisableCollection<InvoiceLine>();
    }

    [Child]
    public IList<InvoiceLine> Lines { get; privateset; }
}

[Agregatable]
publicclass InvoiceLine
{
    [Parent]
    public Invoice Invoice { get; privateset; }
}
```

The `Invoice` class contains a collection of `InvoiceLine` instances. We want each item of the `Lines` collection to be a child of the `Invoice` instance. However, the collection itself should not be considered a child of the `Invoice`. Additionally, we want the `InvoiceLine.Invoice` property to be set to the `Invoice`, not to the collection.

To implement this behavior, `PostSharp` needs to give a different status to collections than to other entities. This concept is named a *parent surrogate*, because the collection acts as a surrogate (or proxy) between the parent and its children.

Any aggregatable object can act as a parent surrogate, but only collections act as parent surrogates by default. You can override the default behavior by setting the `ChildAttributeIsParentSurrogate` property.

In the next example, the `Lines` collection will be treated as a first-class entity.

```
[Agregatable]
publicclass Invoice
{
    public Invoice()
    {
        this.Lines = new AdvisableCollection<InvoiceLine>();
    }

    [Child(IsParentSurrogate = false)]
    public IList<InvoiceLine> Lines { get; privateset; }
}

[Agregatable]
publicclass InvoiceLine
{
    [Parent]
```

```
    public IList<InvoiceLine> Parent { get; private set; }  
}
```

To cause a custom class to behave like a parent surrogate by default, set the `IsParentSurrogate` property of the `AggregateableAttribute` applied to your class to `true`. In this case, it's not allowed to override the value in the `[Child]` attributes applied to individual properties.

```
[AggregateableAttribute(IsParentSurrogate = false)]
```

Enumerating children and parent surrogates

The default behavior of the `VisitChildren(ChildVisitor, ChildVisitorOptions, Object)` method is to skip the surrogate collection itself and invoke the `ChildVisitor` delegate on each item of the collection. In our first example, calling the `VisitChildren(ChildVisitor, ChildVisitorOptions, Object)` method on the `Invoice` instance will invoke the visitor on the items of the `Lines` collection, but not on the collection instance itself.

You can customize this behavior by providing one or more flags for the `ChildVisitorOptions` parameter of the method. The `ChildVisitorOptions.IncludeParentSurrogates` flag will cause the visitor to be additionally invoked on the instances of the surrogate collections, while the `ChildVisitorOptions.ExcludeIndirectChildren` flag will exclude the items of such collection from being visited.

Collections of references

As we showed earlier, when you annotate the collection property with the `[Child]` attribute, collection items become children of the class instance.

In certain situations, you may want to have a collection of references. The collection itself is still marked with the `[Child]` custom attribute because it would make sense from the point of view of other patterns (for instance, changes in the collection must be recorded by the `Recordable` pattern). However, the collection items themselves must not be considered children of the entity.

To implement this requirement, you can set the `ChildAttributeItemsRelationship` property to `RelationshipKind.Reference`.

In the example below, the `RelatedOrders` collection is a child and therefore its changes are being recorded by the `Recordable` aspect. However, collection items are not children of the parent entity, because related orders do not belong to the invoice.

```
[Recordable]  
public class Invoice  
{  
    public Invoice()  
    {  
        this.Lines = new AdvisableCollection<InvoiceLine>();  
        this.RelatedOrders = new AdvisableCollection<Order>();  
    }  
  
    [Child]  
    public IList<InvoiceLine> Lines { get; private set; }  
  
    [Child(ItemsRelationship = RelationshipKind.Reference)]  
    public IList<Order> RelatedOrders { get; private set; }  
}
```

Using immutable collections

In section [Working With Child Collections](#) on page 249, we explained the need to replace standard .NET collections by special advisable collections of the `PostSharp.Patterns.Collections` namespace. These collections come with a significant inconvenient: they have a significant performance and memory overhead. In many situations, collections can be replaced by immutable collections. *Immutable collections* are collections whose content never changes after instan-

tiation. Adequate use of immutable collections can significantly improve application performance and simplify API design compared to mutable collections, whether standard or advisable.

Immutable collections are implemented in the `System.Collections.Immutable` namespace, contained in the `System.Collections.Immutable` NuGet package.

The Aggregatable pattern and threading models support immutable collections. When you assign an immutable collection to a child field of a parent object, items of the collection become children of the parent object. Immutable collections behave similarly than other types, so you still have to use the `ChildAttribute` and `ReferenceAttribute` custom attributes as usual.

PART 10

Undo/Redo

CHAPTER 43

Making Your Model Recordable

To make an object usable for undo/redo operations, you will need to add the `RecordableAttribute` aspect to the class. This aspect instruments changes to fields and records them into a `Recorder`. The aspect also instruments public methods to group field changes into logical operations.

To make the class recordable:

1. Add the `PostSharp.Patterns.Model` package to the project.
2. Add the `RecordableAttribute` to the class.
3. Annotate your object model for parent/child relationships as explained in [Parent/Child, Visitor and Disposable on page 239](#).

Example

The following example shows an object model that has been prepared for undo/redo.

```
[Recordable]
publicclass Invoice
{
    public Invoice()
    {
        this.Lines = new AdvisableCollection<InvoiceLine>();
    }

    [Reference]
    public Customer Customer { get; set; }

    [Child]
    public IList<InvoiceLine> Lines { get; private set; }

    [Child]
    public Address DeliveryAddress { get; set; }

    [NotRecorded]
    public boolean Persisted { get; set; }
}

[Recordable]
publicclass InvoiceLine
{
    [Reference]
    private Product product;

    publicdecimal Amount { get; set; }

    [Parent]
    public Invoice ParentInvoice { get; private set; }
}

[Recordable]
publicclass Address
{
}
```

Making Your Model Recordable

```
[Recordable]
publicclass Customer
{
}
```

By adding the `RecordableAttribute` aspect to the classes, all modifications to properties of these classes (including modifications to child collections) will be recorded.

Excluding a property from undo/redo

To exclude a property from participating in undo/redo, annotate that property with `NotRecordedAttribute`.

```
[Recordable]
publicclass Document
{
    publicstring Text { get; set; }

    [NotRecorded]
    public boolean AnyChangesMade { get; set; }
}
```

CHAPTER 44

Adding Undo/Redo to the User Interface

The Undo/Redo functionality that you added to your codebase needs to be made available to the users. Users will want to have the ability to move forward and backward through the stack of recorded operations.

This topic contains the following sections:

- [Using the ready-made WPF controls on page 259](#)
- [Clearing the initial history on page 260](#)
- [Creating custom undo/redo controls on page 260](#)

Using the ready-made WPF controls

PostSharp includes two button controls **UndoButton** and **RedoButton** that you can add to your application.

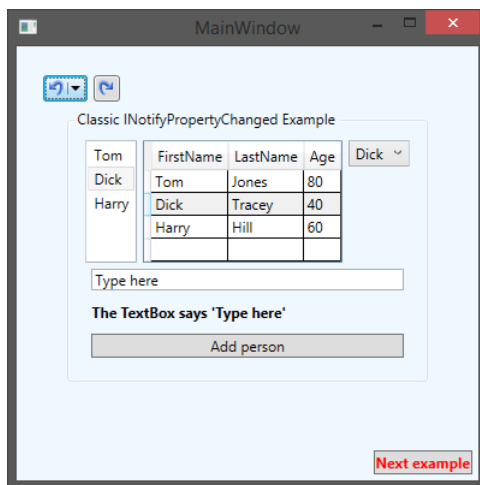
To add Undo/Redo to a WPF window:

1. Install the *PostSharp.Patterns.Xaml* package using NuGet.
2. Add the following namespace declaration to the root element of your XAML file:

```
xmlns:model="clr-namespace:PostSharp.Patterns.Model.Controls;assembly=PostSharp.Patterns.Xaml"
```

3. To add Undo and Redo buttons to the user interface, including the following two lines of XAML.

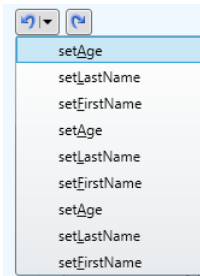
```
<model:UndoButton HorizontalAlignment="Left" Margin="22,24,0,0" VerticalAlignment="Top" />
<model:RedoButton HorizontalAlignment="Left" Margin="64,24,0,0" VerticalAlignment="Top" />
```



Your users are now able to make a changes in the user interface and Undo and/or Redo those changes at any point that they want.

Clearing the initial history

If we were to open the Customer management screen you would notice that the Undo button has a number of actions listed under it.



Those actions are listing the changes that were taken when the different Person instances were loaded and their properties were set. Most users will only want to see actions that they have manually taken in the screen. As such, you will need to manually interact with the Recorder to ensure that the Undo button list is empty when the window opens.

To provide an empty list of recorded actions when the window is initially opened, open the `ViewMode1Main` class and find the constructor. Add the following as the last line in the constructor:

```
RecordingServices.DefaultRecorder.Clear();
```

The `Recorder` class is accessed through the `RecordingServicesDefaultRecorder` property. This property contains the current `Recorder` instance that is being used by the `RecordableAttribute` aspect. The `Recorder` class has two collections, `UndoOperations` and `RedoOperations`, which contain all of the past operations that can be undone and redone. The `Clear` method removes all operations from both of those collections.

Now when you open the Customer management screen both the Undo and Redo buttons will show no history. This is the simplest type of Undo/Redo implementation that you can do. It will record each property change operation separately in the Undo and Redo UI buttons which probably isn't what you, or your users, will want to see. Read [Customizing Undo/Redo Operation Names on page 261](#) to learn how to record groupings of operations that make sense to your business users.

Creating custom undo/redo controls

If the buttons provided by PostSharp don't meet your requirements, you can create your own controls for WPF, Windows Phone, WinRT, or any other platform.

Custom controls will typically provide a front-end to the global `Recorder` exposed by the `RecordingServicesDefaultRecorder` property. The controls could show the contents of the `UndoOperations` and `RedoOperations` collections. Controls typically use the `RecorderUndo` and `RecorderRedo` methods.

CHAPTER 45

Customizing Undo/Redo Operation Names

The example of previous sections displays the list of operations appearing in the two UI buttons. That list of operations references the setters on the different individual properties in a very technical manner, for instance the operation of setting the first name is named `set_FirstName`, according to the name of the property in source code.

End users will want to see the operations described in meaningful business terms, not technical ones. This article will show you how to explicitly name the recording operations that will take place in your code.

This topic contains the following sections:

- [Default operation naming algorithm on page 261](#)
- [Setting the operation name declaratively on page 262](#)
- [Setting the operation scope and name dynamically on page 262](#)
- [Using the `OperationFormatter` class on page 263](#)

Default operation naming algorithm

From the end user's perspective, the undo/redo feature exposes a flat list of operations that can be undone or redone. From a system perspective, an operation is composed of changes to individual fields and collections. For instance, moving a picture on a design surface is seen as a single operation **Move** by the user, but it is composed of two changes in fields `x` and `y`.

Let's see this in a code example:

```
[Recordable]
publicclass Picture
{
    privatedouble x, y;

    publicdouble X
    {
        get { return x; }
        set { x = value; }
    }

    publicdouble Y
    {
        get { return y; }
        set { y = value; }
    }

    publicvoid Move( double x, double y )
    {
        this.X = x;
        this.Y = y;
    }
}

publicstaticclass Program
{
    publicstaticvoid Main()
    {
        var picture = new Picture();
    }
}
```

Customizing Undo/Redo Operation Names

```
    picture.Move( 10, 10 );  
  
    // 1 undo operation at this point: Move.  
  
    picture.X = 20;  
  
    // 2 undo operations at this point: set_X, Move.  
  
    picture.Y = 20;  
  
    // 3 undo operations at this point: set_Y, set_X, Move.  
  }  
}
```

By default, the `RecordableAttribute` aspect will automatically open a new operation for any public method unless the current `Recorder` already has an open operation. Therefore, invoking the `Move` method results in a single operation, even if it modifies two fields. Note that the `Move` method invokes the setters of public properties `X` and `Y`, which are themselves public methods, but they do not open new operations since they run from within the `Move` method. However, when properties `X` and `Y` are accessed from outside of the `Picture` class, new operations are created for the `set_X` and `set_Y` methods.

Setting the operation name declaratively

By default, the name of an operation is set to the name of the method. There are various ways to customize this name, and the easiest is to add a `RecordingScopeAttribute` custom attribute to the public method.

In the following example, we're declaring a different name for the `Move` method:

```
[Recordable]  
publicclass Picture  
{  
    private double x, y;  
  
    [RecordingScope("Moving the picture")]  
    publicvoid Move( double x, double y )  
    {  
        this.x = x;  
        this.y = y;  
    }  
}
```

With that `RecordingScopeAttribute` added, the recorded operation will now have a name of `Moving the picture` instead of just `Move`.

Setting the operation scope and name dynamically

Setting the operation name declaratively is convenient but relatively rigid. When more flexibility is needed, you can use the `RecorderOpenScope(String, RecordingScopeOption)` method to control the creation and naming of scopes.

In the following example, we will modify the `Move` method to include the target position in the operation description.

To dynamically name an operation:

1. Add the `[RecordingScope(RecordingScopeOption.Skip)]` custom attribute to the method, so that the method does not automatically define a new operation.

NOTE

This step is not required if you are starting the operation from a non-recordable object.

However, if you do not to add this custom attribute to a method of a recordable object, the `RecordableAttribute` aspect will automatically create a new scope to execute the method, and your call of the `OpenScope(String, RecordingScopeOption)` method will be ignored.

2. Invoke the `OpenScope(String, RecordingScopeOption)` method and wrap the code you want to record in a `using` block.

Example

```
[Recordable]
publicclass Picture
{
    privatedouble x, y;

    [RecordingScope(RecordingScopeOption.Skip)]
    publicvoid Move( double x, double y )
    {
        string scopeName = string.Format( "Moving to ({0}, {1})", x, y );

        using (RecordingScope scope = RecordingServices.DefaultRecorder.OpenScope(scopeName))
        {
            this.x = x;
            this.y = y;
        }
    }
}
```

Using the OperationFormatter class

Explicitly declaring the name for every operation would be a large and tedious task. It is possible to write your own naming engine and apply that set of naming rules across the entire application. To achieve this, derive your own implementation from the `OperationFormatter` class. to the

In the following example, we will create a custom formatter that reads the operation name from the `DisplayNameAttribute` custom attribute and display the value to which a property has been set.

To create and register a custom OperationFormatter:

1. Create a new class and inherit from the `OperationFormatter` class.
2. Create a constructor for the new formatter class.

3. Override the `FormatOperationDescriptor(IOperationDescriptor)` method and write your custom logic for generating a custom operation name.

NOTE

Formatters create a chain of responsibility. If one formatter is unable to provide a name it will ask the next formatter in the chain to attempt to provide a name. To make the hand-off occur the `FormatOperationDescriptor(IOperationDescriptor)` method needs to return `null`. If it returns anything else the chain is broken and the returned value is used as a name.

4. Finally, you need to add your custom name formatter into the chain of responsibility.

```
RecordingServices.OperationFormatter = new MyOperationFormatter(RecordingServices.OperationFormatter);
```

Because the `RecordingServices` is making use of a chain of responsibility, you are able to insert as many custom name formatters as you want. You are also able to determine their order of execution based on the order that you insert them into the chain of responsibility.

Example

```
class MyOperationFormatter : OperationFormatter
{
    public MyOperationFormatter( OperationFormatter next ) : base( next )
    {
    }

    protected override string FormatOperationDescriptor( IOperationDescriptor operation )
    {
        if ( operation.OperationKind != OperationKind.Method )
            return null;

        var descriptor = (MethodExecutionOperationDescriptor) operation;

        if ( descriptor.Method.IsSpecialName && descriptor.Method.Name.StartsWith( "set_" ) )
        {
            // We have a property setter.
            var property = descriptor.Method.DeclaringType.GetProperty(
                descriptor.Method.Name.Substring( 4 ),
                BindingFlags.Instance | BindingFlags.Public | BindingFlags.NonPublic );

            var attributes =
                (DisplayNameAttribute[]) property.GetCustomAttributes( typeof( DisplayNameAttribute ), false );

            if ( attributes.Length > 0 )
                return string.Format( "Set {0} to {1}", attributes[0].DisplayName, descriptor.Arguments[0] ?? "null" );
        }
        else
        {
            // We have another method.
            var attributes = (DisplayNameAttribute[])
                descriptor.Method.GetCustomAttributes( typeof( DisplayNameAttribute ), false );

            if ( attributes.Length > 0 )
                return attributes[0].DisplayName;
        }

        return null;
    }
}
```


CHAPTER 46

Assigning Recorders Manually

By default, all recordable objects are attached to the global Recorder exposed on the `RecordingServicesDefaultRecorder` property. There is nothing you have to do to make this happen. There may be circumstances where you want to create and assign your own recorder to the undo/redo process. There are two different ways that you can accomplish this.

This topic contains the following sections:

- [Overriding the default RecorderProvider on page 265](#)
- [Attaching a recorder manually on page 266](#)

Overriding the default RecorderProvider

By default, the `RecordableAttribute` aspect attaches an object to a Recorder as soon as its constructor exits. To determine which Recorder should be used, the aspect uses the `RecordingServicesRecorderProvider` service. By default, this service always serves the global instance that is also exposed on the `RecordingServicesDefaultRecorder` property.

You can override this automatic assignment to inject your own `RecorderProvider` to into the process.

To use a custom RecorderProvider:

1. Create a class inherited from the `RecorderProvider` class.
2. Implement the chaining constructor. The `RecorderProvider` that you inherited from requires a `RecorderProvider` as a constructor parameter. This constructor parameter facilitates the chain of responsibility for providers that can be run when a Recorder is requested. To keep the chain of responsibility intact your custom `RecorderProvider` will need to accept a `RecorderProvider` in its constructor and pass that to the base constructor.
3. Override the `GetRecorderCore(Object)` method.
4. Insert an instance of your custom `RecorderProvider` class into the chain of responsibility by assigning it to the `RecordingServicesRecorderProvider`.

```
RecordingServices.RecorderProvider = new MyProvider(RecordingServices.RecorderProvider);
```

NOTE

`RecorderProvider` is a chain of responsibility. As such, if a `GetRecorderCore(Object)` method returns `null` then the chain will move on to the next `RecorderProvider` and attempt to get a Recorder to use.

By overriding the default `RecorderProvider` you are able to assign a custom Recorder across the entire application.

Example

```
public class ThreadStaticRecorderProvider : RecorderProvider
{
    private static Recorder _recorder;
```

Assigning Recorders Manually

```
public ThreadStaticRecorderProvider(RecorderProvider next) : base(next)
{
}

public Recorder GetRecorderImpl(object obj)
{
    if ( _recorder == null )
    {
        _recorder = new Recorder();
    }

    return _recorder;
}
}
```

Attaching a recorder manually

The second way that you can add a Recorder to objects is to manually assign them when, and where, they are needed.

To manually assign a Recorder to an object:

1. Set the `RecordableAttributeAutoRecord` property to false for that class.

```
[Recordable(AutoRecord = false)]
public class Invoice
{
}
```

NOTE

By disabling `AutoRecord` you are telling the `RecordingServices` that this object should not be included in recordings unless the recording is explicitly declared in your code.

2. Create a new instance of a Recorder and attach the object to it using the `Attach(Object)` method.

```
var invoice = new Invoice();

var recorder = new Recorder();
recorder.Attach(invoice);
```

You can then use the `Detach(Object)` method to remove the Recorder from the object in question.

NOTE

An object must always have the same Recorder as its parent has unless the parent has no Recorder assigned. Because of this, whenever a Recorder is assigned to an object, all of the child objects will have that same Recorder assigned to them. However, if you detach a child object from its parent the child object's assigned Recorder will not be detached. For more information about parent-child relationships, see [Parent/Child, Visitor and Disposable on page 239](#).

CHAPTER 47

Adding Callbacks on Undo and Redo

You may run into situations where you will want to execute some code before or after an object is being modified by an Undo or Redo operation. This capability is provided through the `IRecordableCallback` interface.

In the following example, we output a message each time an undo or redo operation executes.

```
[Recordable]
publicclass Invoice : IRecordableCallback
{
    publicvoid OnReplaying(ReplayKind kind, ReplayContext context)
    {
        if (kind == ReplayKind.Redo) {
            Console.WriteLine("I will now redo a previously undone change to the shipping date.");
        }
    }

    publicvoid OnReplayed(ReplayKind kind, ReplayContext context)
    {
        if (kind == ReplayKind.Undo) {
            Console.WriteLine("A change to the shipping date is now undone.");
        }
    }

    public DateTime ShippingDate { get; set; }
}
```

For more information, see the reference documentation for the `IRecordableCallback` interface.

CHAPTER 48

Understanding the Recordable Aspect

This section describes how the `RecordableAttribute` aspect is implemented. It helps developers and architects to understand the behavior and limitations of the aspect.

This topic contains the following sections:

- [Overview on page 269](#)
- [Scopes and Logical Operations on page 269](#)
- [Atomic Operations on page 270](#)
- [Primitive Operations on page 270](#)
- [Restore Points on page 271](#)
- [Implementing `!EditableObject` on page 271](#)
- [Callback Methods on page 271](#)
- [Memory Consumption on page 271](#)

Overview

When the `RecordableAttribute` aspect is applied to a class, the aspect records changes performed on instances of this class. Changes are represented as instances of the `Operation` class. For instance, the `FieldOperationT` class represents the operation of changing the value to a field. All operations implement the `Undo(ReplayContext)` and `Redo(ReplayContext)` methods. For instance, the `FieldOperationT` class stores both the new and old value so that the operation can be undone and redone.

The changes are recorded into the `Recorder` object. The `Recorder` maintains two collections of operations: `UndoOperations` and `RedoOperations`. The `RecorderUndo` method takes the last operation from the `UndoOperations` collection, invokes `OperationUndo(ReplayContext)` for this operation, and moves the operation to the `RedoOperations` collection. The `RecorderRedo` method works symmetrically.

It would not be safe, however, to allow users to undo changes in the object model back to any arbitrary point in history. Users don't want to undo primitive changes to an object model, but to undo whole operations understood from a user's perspective. This is why the `UndoOperations` and `RedoOperations` collections don't expose primitive changes on the object model but logical operations.

By default, logical operations are automatically opened when calling a public or internal method of a recordable object, and closed when the same method exits. The principal use case of scopes is to define user-friendly operation names.

There is typically a single instance of the `Recorder` class per application, but there could be many if needed (for instance in a multi-document application). The default single instance is accessible from the `RecordingServicesDefaultRecorder` property. By default, recordable objects are attached to the default recorder immediately after completion of the constructor. See [Assigning Recorders Manually on page 265](#) to learn how to customize this behavior.

Scopes and Logical Operations

Scopes are a mechanism to aggregate several primitive operations into logical operations that make sense for the end-user. Logical operations are represented by the `CompositeOperation` class.

In general, logical operations form a flat structure: the `UndoOperations` and `RedoOperations` collections are flat double linked lists, and each `CompositeOperation` typically contains primitive operations such as a field value change.

Scopes define boundaries of logical operations. Scopes can be opened using the `RecorderOpenScope(RecordingScopeOption)` method, which returns an object of type `RecordingScope`. This class implements the `IDisposable` interface, making it convenient to define scopes with the `using` statement.

By default, the `RecordableAttribute` aspect encloses all instance public and internal methods with an implicit scope. That is, by default, public and internal methods define boundaries of logical operations.

Unlike logical operations, scopes are generally nested. Scope nesting typically happens when a public method directly or indirectly invokes another public method. In general, only the outermost scope results in creating a logical operation. This is why, in general, logical operations form a flat structure.

Because they are visible to users, logical operations must be given a user-friendly name. PostSharp defines default names that are not user-friendly. The responsibility of generating operation names is implemented by the `OperationFormatter` class. You can provide your own `OperationFormatter` to generate operations names on demand, or you can set the name explicitly in source code for each operation.

Scope names can be declaratively defined using the `RecordingScopeAttribute` custom attribute, or programmatically using the `RecorderOpenScope(String, RecordingScopeOption)` method. To learn more about operation names, see [Customizing Undo/Redo Operation Names](#) on page 261.

Atomic Operations

Atomic scopes are scopes whose changes are automatically rolled back when it does not complete successfully, typically when an exception occurs. The rollback is implemented using the undo mechanism. Atomic scopes are a concept similar to transactions, but multithreading is not taken into account. Therefore, other threads may see changes that have not been "committed", because the `Recordable` pattern does not have a notion of transaction isolation.

Atomic scopes cause composite operations to have a tree structure. However, the concept of atomic structure does not surface to the users. Therefore, from a user's perspective, the `UndoOperations` and `RedoOperations` collections still present linear lists of logical operations.

Scope defined declaratively using the `RecordingScopeAttribute` custom attribute, or programmatically using the `RecorderOpenScope(RecordingScopeOption)` method.

Primitive Operations

The following table lists the primitive operations that are automatically appended to the `Recorder` object by the `RecordableAttribute` aspect.

Class	Description
<code>FieldOperationT</code>	Represents the operation of setting a field to a different value.
<code>CollectionOperationT</code>	Represents operations on collections.
<code>DictionaryOperationTKey, TValue</code>	Represents operations on dictionaries.
<code>RecorderOperation</code>	Represents the operation of attaching or detaching an object to or from a <code>Recorder</code> .

Additionally to these system-defined operations, it is possible to implement custom operations by deriving from the `Operation` abstract class. You can then use the `RecorderAddOperation(Operation)` method to append the custom operation to the `Recorder`.

Logical operations, which are presented to the end user, are typically represented as instances of the `CompositeOperation` class.

Restore Points

Restore points act like bookmarks in the list of operations. They allow to undo or redo operations up to a specific point. You can use the `RecorderAddRestorePoint(String)` method to create a restore point. The method returns an instance of the `RestorePoint` class, which derives from the `Operation` class. Unlike other operations, you can safely remove a restore point from the history thanks to the `Remove` method.

Implementing IEditableObject

You can use the `EditableObjectAttribute` custom attribute to automatically implement the `IEditableObject` interface. The implementation is based on the `RecordableAttribute` aspect. It creates a `RestorePoint` when the `BeginEdit` method is invoked, removes the restore point upon `EndEdit`, and undoes changes up to the restore point when `CancelEdit` is called.

Because of this implementation strategy, it is possible that `CancelEdit` actually cancels changes done to other objects that share the same `Recorder`.

Callback Methods

The `Recorder` will invoke the `OnReplaying(ReplayKind, ReplayContext)` and `OnReplayed(ReplayKind, ReplayContext)` methods of any recordable object implementing the `IRecordableCallback` interface, whenever the object is affected by an undo or redo operation.

The order in which these methods are ordered on several objects is nondeterministic; in particular, the aggregation structure is not respected.

From callback methods, it is not allowed:

- to perform a change that would be recorded, e.g. to set a field that has not been waived from recording with the `NotRecordedAttribute` custom attributes.
- to invoke methods `Undo`, `Redo` or `AddRestorePoint` of the `Recorder` class.

Memory Consumption

The `UndoOperations` and `RedoOperations` collections hold strong references to all objects that have changes that can be undone or redone. This means that these objects cannot be garbage-collected and will remain in memory.

You can define the maximal number of operations available for undo thanks to the `RecorderMaximumOperationsCount` property.

PART 11

Caching

CHAPTER 49

Caching Method Return Values

Suppose you have a time-consuming method that always returns the same return value when called with the same arguments. You decided to cache it. This topic describes how to proceed.

This topic contains the following sections:

- [Caching the return value of a method on page 275](#)
- [Configuring the cache behavior on page 276](#)
- [Configuring caching with custom attributes on page 276](#)
- [Configuring caching with caching profiles on page 277](#)
- [Disabling caching at run time on page 278](#)

Caching the return value of a method

To make a return value of a method being cached:

1. Add a reference to the `PostSharp.Patterns.Caching` package.
2. Add the `CacheAttribute` custom attribute on the method which should be cached. Such method is called the *cached method*.
3. Now you need to specify which caching framework or caching server is to be used by the `CacheAttribute` aspect. We call this the *caching backend*. You must specify the caching backend by setting the `CachingServicesDefaultBackend` property. See [Caching Back-Ends on page 293](#) for a list of caching backends.

IMPORTANT NOTE

The caching backend has to be set before any cached method is called for the first time.

4. Unless all method parameters are intrinsic types such as `int` or `string`, you need to ensure that the parameter types generate a meaningful cache key. See [Customizing Cache Keys on page 289](#) for details.

Example

In this example, the `GetNumber` method return value is cached.

```
using System;
using System.Threading;
using PostSharp.Patterns.Caching;
using PostSharp.Patterns.Caching.Backends;

namespace PostSharp.Samples.Caching.MethodResults
{
    class Program
    {
        static void Main( string[] args )
        {
            CachingServices.DefaultBackend = new MemoryCachingBackend();

            Console.WriteLine( "Retrieving value of 1 for the 1st time should hit the database." );
            Console.WriteLine( "Retrieved: " + GetNumber( 1 ) );
        }
    }
}
```

Caching Method Return Values

```
        Console.WriteLine( "Retrieving value of 1 for the 2nd time should NOT hit the database." );
        Console.WriteLine( "Retrieved: " + GetNumber( 1 ) );

        Console.WriteLine( "Retrieving value of 2 for the 1st time should hit the database." );
        Console.WriteLine( "Retrieved: " + GetNumber( 2 ) );

        Console.WriteLine( "Retrieving value of 2 for the 2nd time should NOT hit the database." );
        Console.WriteLine( "Retrieved: " + GetNumber( 2 ) );
    }

    [Cache]
    static int GetNumber( int id )
    {
        Console.WriteLine( $">> Retrieving {id} from the database..." );
        Thread.Sleep( 1000 );
        return id;
    }
}
}
```

The output of this sample is:

```
Retrieving value of 1 for the 1st time should hit the database.
>> Retrieving 1 from the database...
Retrieved: 1
Retrieving value of 1 for the 2nd time should NOT hit the database.
Retrieved: 1
Retrieving value of 2 for the 1st time should hit the database.
>> Retrieving 2 from the database...
Retrieved: 2
Retrieving value of 2 for the 2nd time should NOT hit the database.
Retrieved: 2
```

Configuring the cache behavior

The following elements of the `CacheAttribute` aspect behavior can be configured:

- expiration (absolute and sliding),
- priority,
- auto-reload, and
- enabled/disabled.

Configuring caching with custom attributes

You can configure the `CacheAttribute` aspect by setting the properties of the `CacheAttribute` custom attribute. The inconvenience of this approach is that you have to repeat the configuration for each cached method. To configure several methods in a single line of code, you can add the `CacheConfigurationAttribute` custom attribute to the declaring type, a parent of the declaring type, or the declaring assembly.

When PostSharp processes the `CacheAttribute` aspect on a given method, it looks for configuration in the following order.

1. the `CacheAttribute` itself,
2. a `CacheConfigurationAttribute` attribute on the declaring class of the cached method,
3. a `CacheConfigurationAttribute` attribute any parent class of the cached method (starting from the declaring class to the parent class),
4. a `CacheConfigurationAttribute` added to the assembly declaring the cached method. Note that all `CacheConfigurationAttribute` assembly-level attributes defined in a different assembly than the current one are ignored.

5. the caching profile (see below).

Configuration is defined on a per-property basis. For each property of the cache aspect, the value set by the highest item in the preceding list wins.

Example

In the following account, the absolute expiration of cache items is set to 60 seconds for methods of the `AccountServices` class, but to 20 seconds for the `GetAccountsOfCustomer` method.

```
using System;
using System.Collections.Generic;
using PostSharp.Patterns.Caching;

namespace PostSharp.Samples.Caching
{
    [CacheConfiguration( AbsoluteExpiration = 60 )]
    class AccountServices
    {
        [Cache]
        publicstatic Account GetAccount(int id)
        {
            // Detailed skipped.
        }

        [Cache( AbsoluteExpiration = 20 )]
        publicstatic IEnumerable<Account> GetAccountsOfCustomer(int customerId)
        {
            // Detailed skipped.
        }

        publicstaticvoid UpdateAccount(Account account)
        {
            // Detailed skipped.
        }
    }
}
```

Configuring caching with caching profiles

Using custom attributes to configure caching has two major inconveniences: it is hard to share caching configuration between several classes that don't derive from the same parent, and the configuration cannot be modified at run time. To work around these limitations, you can use caching profiles.

Caching profiles are useful in the following scenarios:

- to centralize the configuration of several cached methods (which may belong to different type hierarchies) into a single location;
- to modify the configuration of cached methods (such as expiration settings) at run-time; and
- to completely disable or re-enable caching at run-time.

Caching profiles are represented by the `CachingProfile` class. They are exposed on the `CachingServicesProfiles` property, which is a collection indexed by the profile name. The default caching profile is accessible via the `CachingServices.Profiles.Default` property.

NOTE

Configuration specified thanks to the `CacheAttribute` aspect and the `CacheConfigurationAttribute` custom attribute have priority over the configuration of the `CachingProfile`.

To use a caching profile:

1. Set the `ProfileName` property of the `CacheAttribute` aspect or the `CacheConfigurationAttribute` custom attribute.
2. Configure the `CachingProfile` object exposed on the `CachingServicesProfiles` collection.

Example

The following code snippet sets the profile name to `Account` for all methods of the `AccountServices` class.

```
using System;
using System.Collections.Generic;
using System.Threading;
using PostSharp.Patterns.Caching;

namespace PostSharp.Samples.Caching
{
    [CacheConfiguration( ProfileName = "Account" )]
    class AccountServices
    {
        [Cache]
        publicstatic Account GetAccount(int id)
        {
            // Details skipped.
        }

        [Cache]
        publicstatic IEnumerable<Account> GetAccountsOfCustomer(int customerId)
        {
            // Details skipped.
        }

        publicstaticvoid UpdateAccount(Account account)
        {
            // Details skipped.
        }
    }
}
```

The following code snippet sets the absolute expiration to 60 seconds for all methods using the `Account` profile.

```
CachingServices.Profiles["Account"].AbsoluteExpiration = TimeSpan.FromSeconds(60);
```

Disabling caching at run time

You can disable caching at run time by setting the `IsEnabled` property of a caching profile to `false`, for instance:

```
CachingServices.Profiles.Default.IsEnabled = false;
CachingServices.Profiles["Account"].IsEnabled = false;
```

CHAPTER 50

Removing Items From the Cache

When a method updates an entity, it must also remove any cache item that depends on this entity. One way to achieve this goal is to require the update methods to know precisely which cached methods are dependent on the entity that has been modified, and to remove these methods (with proper arguments) from the cache. We call this scenario *direct* cache invalidation.

The benefit of direct invalidation is that it does not require a lot of resources on the caching backend. However, this approach has a big disadvantage: it exhibits an imperfect separation of concerns. Update methods need to have a precise knowledge of cached methods (typically read methods), therefore update methods need to be modified whenever a read method is added. For an approach with better abstraction, see [Working with Cache Dependencies on page 283](#).

This topic contains the following sections:

- [Invalidating cache items using the \[InvalidateCache\] aspect on page 279](#)
- [Invalidating cache items imperatively on page 280](#)

Invalidating cache items using the [InvalidateCache] aspect

You can add the `InvalidateCacheAttribute` aspect to a method (called the *invalidating method*) to cause any call to this method to remove from the cache the value of one or more other methods. Parameters of both methods are matched by name and type. If any parameter of the cached method cannot be matched with a parameter of the invalidating method, you will get a build error (unless the parameter has the `NotCacheKeyAttribute` custom attribute). The order of parameters is not considered.

NOTE

By default, the `InvalidateCacheAttribute` aspect will look for the cached method in the current type. You can specify a different type using the alternative constructor of the custom attribute. When you invalidate a non-static method (unless instance has been excluded from the cache key by setting the `IgnoreThisParameter` to `true`), you can do it only from a non-static method of a derived type.

If there are more invalidated methods of the same name for one invalidating method, a build error is emitted. To enable invalidation of all the matching overloads by the one invalidating methods, set the property `AllowMultipleOverloads` to `.`

Example

In this example, the `Update` method invalidates the cached return value of the `GetValue` method.

```
using System;
using System.Collections.Generic;
using System.Threading;
using PostSharp.Patterns.Caching;
using PostSharp.Patterns.Caching.Backends;

namespace PostSharp.Samples.Caching.Invalidation
{
    class Database
    {
        private Dictionary<int, string> data = new Dictionary<int, string>();
    }
}
```

Removing Items From the Cache

```
[Cache]
publicstring GetValue( int id )
{
    Console.WriteLine( $">> Retrieving {id} from the database..." );
    Thread.Sleep( 1000 );
    returnthis.data[id];
}

[InvalidateCache( nameof(GetValue) )]
publicvoid Update( int id, stringvalue )
{
    this.data[id] = value;
}
}

class Program
{
    staticvoid Main( string[] args )
    {
        CachingServices.DefaultBackend = new MemoryCachingBackend();

        Database db = new Database();

        db.Update( 1, "first" );

        Console.WriteLine( "Retrieving value of 1 for the 1st time should hit the database." );
        Console.WriteLine( "Retrieved: " + db.GetValue( 1 ) );

        Console.WriteLine( "Retrieving value of 1 for the 2nd time should NOT hit the database." );
        Console.WriteLine( "Retrieved: " + db.GetValue( 1 ) );

        db.Update( 1, "second" );

        Console.WriteLine( "Retrieving updated value of 1 for the 1st time should hit the database." );
        Console.WriteLine( "Retrieved: " + db.GetValue( 1 ) );
    }
}
}
```

The output of this sample is:

```
Retrieving value of 1for the 1st time should hit the database.
>> Retrieving 1from the database...
Retrieved: first
Retrieving value of 1for the 2nd time should NOT hit the database.
Retrieved: first
Retrieving updated value of 1for the 1st time should hit the database.
>> Retrieving 1from the database...
Retrieved: second
```

Invalidating cache items imperatively

Instead of annotating invalidating methods with a custom attribute, you can call to one of the overloads of the `CachingServices.Invalidation.Invalidate` method.

Example

In this example, the cached return value of the `GetValue` method is invalidated by calling one of the overloads of the `CachingServices.Invalidation.Invalidate` method.

```
using System;
using System.Collections.Generic;
using System.Threading;
using PostSharp.Patterns.Caching;
using PostSharp.Patterns.Caching.Backends;

namespace PostSharp.Samples.Caching.ImperativeInvalidation
{
```



```

class Database
{
    private Dictionary<int, string> data = new Dictionary<int, string>();

    [Cache]
    publicstring GetValue( int id )
    {
        Console.WriteLine( $">> Retrieving {id} from the database..." );
        Thread.Sleep( 1000 );
        returnthis.data[id];
    }

    publicvoid Update( int id, stringvalue )
    {
        this.data[id] = value;
    }
}

class Program
{
    staticvoid Main( string[] args )
    {
        CachingServices.DefaultBackend = new MemoryCachingBackend();

        Database db = new Database();

        db.Update( 1, "first" );

        Console.WriteLine( "Retrieving value of 1 for the 1st time should hit the database." );
        Console.WriteLine( "Retrieved: " + db.GetValue( 1 ) );

        Console.WriteLine( "Retrieving value of 1 for the 2nd time should NOT hit the database." );
        Console.WriteLine( "Retrieved: " + db.GetValue( 1 ) );

        db.Update( 1, "second" );
        CachingServices.Invalidation.Invalidate( db.GetValue, 1 );

        Console.WriteLine( "Retrieving updated value of 1 for the 1st time should hit the database." );
        Console.WriteLine( "Retrieved: " + db.GetValue( 1 ) );
    }
}

```

The output of this sample is:

```

Retrieving value of 1for the 1st time should hit the database.
>> Retrieving 1from the database...
Retrieved: first
Retrieving value of 1for the 2nd time should NOT hit the database.
Retrieved: first
Retrieving updated value of 1for the 1st time should hit the database.
>> Retrieving 1from the database...
Retrieved: second

```


CHAPTER 51

Working with Cache Dependencies

Cache dependencies have two major use cases. First, dependencies can act as a middle layer between the cached methods (typically the read methods) and the invalidating methods (typically the update methods) and therefore reduce the coupling between the read and update methods. Second, cache dependencies can be used to represent external dependencies, such as file system dependencies or SQL dependencies.

Compared to direct invalidation, using dependencies exhibits lower performance and higher resource consumption in the caching backend because of the need to store and synchronize the graph of dependencies. For details about direct invalidation, see [Removing Items From the Cache on page 279](#).

This topic contains the following sections:

- [Adding string dependencies on page 283](#)
- [Adding object-oriented dependencies through the `ICacheDependency` interface on page 284](#)
- [Adding object-oriented dependencies through a formatter on page 286](#)
- [Suspending the collection of cache dependencies on page 286](#)

Adding string dependencies

Eventually, all dependencies are represented as strings. Although we recommend using one of the strongly-typed approaches described below, it is good to understand how string dependencies work.

To assign a string dependency to a cached return value of a method and to invalidate it:

1. Add a call to the `CachingServices.CurrentContext.AddDependency` method to the cached method.
2. Add a call to the `CachingServices.Invalidation.Invalidate` method to the invalidating method.

NOTE

Dependencies properly work with recursive method calls. If a cached method A calls another cached method B, all dependencies of B are automatically dependencies of A, even if A was cached when A was being evaluated.

Example

In this example, the `GetValue` method assigns a string dependency to its cached return value. The `Update` method invalidates the dependency. This causes the related cached return value to be invalidated as well.

```
using System;
using System.Collections.Generic;
using System.Threading;
using PostSharp.Patterns.Caching;
using PostSharp.Patterns.Caching.Backends;

namespace PostSharp.Samples.Caching.StringDependencies
{
    class Database
    {
        private Dictionary<int, string> data = new Dictionary<int, string>();

        private static string GetValueDependencyString( int id ) => $"value:{id}";
    }
}
```

```

[Cache]
publicstring GetValue( int id )
{
    Console.WriteLine( $">> Retrieving {id} from the database..." );
    Thread.Sleep( 1000 );
    CachingServices.CurrentContext.AddDependency( GetValueDependencyString( id ) );
    returnthis.data[id];
}

publicvoid Update( int id, stringvalue )
{
    this.data[id] = value;
    CachingServices.Invalidation.Invalidate( GetValueDependencyString( id ) );
}
}

class Program
{
    staticvoid Main( string[] args )
    {
        CachingServices.DefaultBackend = new MemoryCachingBackend();

        Database db = new Database();

        db.Update( 1, "first" );

        Console.WriteLine( "Retrieving value of 1 for the 1st time should hit the database." );
        Console.WriteLine( "Retrieved: " + db.GetValue( 1 ) );

        Console.WriteLine( "Retrieving value of 1 for the 2nd time should NOT hit the database." );
        Console.WriteLine( "Retrieved: " + db.GetValue( 1 ) );

        db.Update( 1, "second" );

        Console.WriteLine( "Retrieving updated value of 1 for the 1st time should hit the database." );
        Console.WriteLine( "Retrieved: " + db.GetValue( 1 ) );
    }
}
}

```

The output of this sample is:

```

Retrieving value of 1for the 1st time should hit the database.
>> Retrieving 1from the database...
Retrieved: first
Retrieving value of 1for the 2nd time should NOT hit the database.
Retrieved: first
Retrieving updated value of 1for the 1st time should hit the database.
>> Retrieving 1from the database...
Retrieved: second

```

Adding object-oriented dependencies through the ICacheDependency interface

Working with string dependencies can be error-prone because the code generating the string is duplicated in the invalidated and the invalidating method. A better approach is to encapsulate the cache key generation logic, i.e. to represent the cache dependency as an object, and add some key-generation logic to this object.

If you own the source code of the class you want to use as a cache dependency, the easiest approach is to implement the ICacheDependency interface.

NOTE

This approach can be used to implement support for other kinds of dependencies, like file system dependencies or SQL dependencies.

Example

In the following example, the `Customer` class represents a business entity. Instances of this class are being cached. At the same time, they serve as object dependencies, therefore the `Customer` class implements the `ICacheDependency` interface. The `GetValue` method assigns an object dependency of type `Customer` to its cached return value. The `Update` method invalidates the dependency. This causes the related cached return value to be invalidated as well.

```
using System;
using System.Collections.Generic;
using System.Threading;
using PostSharp.Patterns.Caching;
using PostSharp.Patterns.Caching.Backends;
using PostSharp.Patterns.Caching.Dependencies;

namespace PostSharp.Samples.Caching.ICacheDependencies
{
    class Customer : ICacheDependency
    {
        public int Id { get; set; }

        public string Name { get; set; }

        public bool Equals( ICacheDependency other )
        {
            Customer otherCustomer = other as Customer;
            return otherCustomer != null && this.Id == otherCustomer.Id;
        }

        public string GetCacheKey() => $"{nameof(Customer)}:{this.Id}";
    }

    class Database
    {
        private Dictionary<int, Customer> customers = new Dictionary<int, Customer>();

        [Cache]
        public Customer GetCustomer( int id )
        {
            Console.WriteLine( $">> Retrieving {id} from the database..." );
            Thread.Sleep( 1000 );
            Customer customer = this.customers[id];
            CachingServices.CurrentContext.AddDependency( customer );
            return customer;
        }

        public void Update( Customer customer )
        {
            this.customers[customer.Id] = customer;
            CachingServices.Invalidation.Invalidate( customer );
        }
    }

    class Program
    {
        static void Main( string[] args )
        {
            CachingServices.DefaultBackend = new MemoryCachingBackend();

            Database db = new Database();

            db.Update( new Customer() {Id = 1, Name = "Alice"} );

            Console.WriteLine( "Retrieving value of 1 for the 1st time should hit the database." );
            Console.WriteLine( "Retrieved: " + db.GetCustomer( 1 ).Name );

            Console.WriteLine( "Retrieving value of 1 for the 2nd time should NOT hit the database." );
            Console.WriteLine( "Retrieved: " + db.GetCustomer( 1 ).Name );

            db.Update( new Customer() {Id = 1, Name = "Bob"} );
        }
    }
}
```

```
        Console.WriteLine( "Retrieving updated value of 1 for the 1st time should hit the database." );
        Console.WriteLine( "Retrieved: " + db.GetCustomer( 1 ).Name );
    }
}
}
```

The output of this sample is:

```
Retrieving value of 1for the 1st time should hit the database.
>> Retrieving 1from the database...
Retrieved: Alice
Retrieving value of 1for the 2nd time should NOT hit the database.
Retrieved: Alice
Retrieving updated value of 1for the 1st time should hit the database.
>> Retrieving 1from the database...
Retrieved: Bob
```

Adding object-oriented dependencies through a formatter

The previous approach requires implementing an interface in the source code of the business entity. If you cannot modify the source code of a dependency class, the best approach is to implement a formatter for this class and to register it.

See [Customizing Cache Keys](#) on page 289 for details.

Suspending the collection of cache dependencies

A new caching context, accessible through the `CachingServicesCurrentContext`, is created for each cached method. The caching context is propagated along all invoked methods. It is implemented using **AsyncLocal** on platforms that support it, otherwise it is implemented using `LogicalGetData(String)`.

When a parent cached method calls a child cached method, the dependencies of the child methods are automatically added to the parent method, even if the child method was actually not executed because its result was found in cache. Therefore, invalidating a child method automatically invalidates the parent method, which is most of the times an intuitive and desirable behavior.

There are cases where propagating the caching context from the parent to the child methods (and therefore the collection of child dependencies into the parent context) is not desirable. For instance, if the parent method runs an asynchronous child task using `Task.Run` and does not wait for its completion, then it is likely that the dependencies of methods called in the child task should not be propagated to the parent (the child task could be considered a side effect of the parent method, and should not affect caching). Undesired dependencies would not break the program correctness, but it would make it less efficient.

To suspend the collection of dependencies in the current context and in all children contexts, you can use the `CachingServicesSuspendDependencyPropagation` method with a `using` construct.

Example

In the next example, the dependencies of `ChildMethod` (a side-effect method calling the cached method `ToString`) are not propagated to the parent `CachedMethod`.

```
[Cache]
int CachedMethod()
{
    using ( CachingServices.SuspendDependencyPropagation() )
    {
        Task.Run( ChildMethod );
    }

    return 0;
}

void ChildMethod()
{
    Console.WriteLine( "ChildMethod:" + this.ToString() );
}
```

```
}  
[Cache]  
public override string ToString()  
{  
    CachingServices.CurrentContext.AddDependency( "MyDependency" );  
    return "{MyObject}";  
}
```


CHAPTER 52

Customizing Cache Keys

Each value returned by a cached method is indexed by a cache key. By default, the cache key is composed of the full method name, the parameter types, and the parameter values. By default, parameters are formatted using the `ToString` method. PostSharp allows you to completely customize how the cache key is generated.

This topic contains the following sections:

- [Default cache key generation logic on page 289](#)
- [Overriding the `ToString` method on page 289](#)
- [Implementing a custom cache key formatter on page 289](#)
- [Excluding parameters from a cache key on page 290](#)
- [Changing the maximal length of a cache key on page 291](#)
- [Implementing a custom cache key builder on page 291](#)

Default cache key generation logic

By default, the key is composed of the following elements of the method call:

- the full name of the declaring type (including generic parameters, if any),
- the method name,
- the method generic parameters, if any,
- the formatted caller object (unless the method is static),
- a comma-separated list of all method arguments including the full type of the parameter and the formatted parameter value,
- in case that the backend supports it, a global prefix that allows using the same caching server with several applications (see e.g. `KeyPrefix`).

CAUTION NOTE

By default, the cache key of a value is built using `ToString` method, but the default implementation of the `ToString` method does not return a unique string for custom types. You **must** override the `ToString` method or implement a formatter for all types of parameters used in a cached method.

Overriding the `ToString` method

By default, the cache key of a value is built using the `ToString` method. You can override the `ToString` method so that it returns a distinct value for each distinct instance of the type.

Implementing a custom cache key formatter

Since the `ToString` method is used in different contexts than just caching, using it as the cache key might be inappropriate in your case. In this situation, you can implement the `IFormattable` interface or the `FormatterT` class.

See [Implementing a Custom Formatter on page 171](#) for details.

Excluding parameters from a cache key

Using [NotCacheKey]

To exclude a method parameter from being a part of a cache key, add the `NotCacheKeyAttribute` custom attribute on the parameter to be excluded.

Using [IgnoreThisParameter]

To exclude the value of the `this` parameter of an instance method from being a part of a cache key, set the `IgnoreThisParameter` parameter of the aspect to `true`.

Example

In this example, the `this` and `callId` parameters of the `GetNumber` method are not part of the cache key.

```
using System;
using System.Threading;
using PostSharp.Patterns.Caching;
using PostSharp.Patterns.Caching.Backends;

namespace PostSharp.Samples.Caching.ExcludingParameters
{
    class Database
    {
        [Cache( IgnoreThisParameter = true )]
        public int GetNumber( int id, [NotCacheKey] int callId )
        {
            Console.WriteLine( $">> Retrieving {id} from the database, call ID {callId}..." );
            Thread.Sleep( 1000 );
            return id;
        }
    }

    class Program
    {
        static void Main( string[] args )
        {
            CachingServices.DefaultBackend = new MemoryCachingBackend();

            int callId = 0;

            Database db1 = new Database();
            Database db2 = new Database();

            Console.WriteLine( "Retrieving value of 1 for the 1st time from DB 1 should hit the database." );
            Console.WriteLine( "Retrieved: " + db1.GetNumber( 1, callId++ ) );

            Console.WriteLine( "Retrieving value of 1 for the 2nd time from DB 1 passing different call ID should NOT hit" );
            Console.WriteLine( "Retrieved: " + db1.GetNumber( 1, callId++ ) );

            Console.WriteLine( "Retrieving value of 1 for the 1st time from DB 2 should NOT hit the database." );
            Console.WriteLine( "Retrieved: " + db2.GetNumber( 1, callId++ ) );
        }
    }
}
```

The output of this sample is:

```
Retrieving value of 1 for the 1st time from DB 1 should hit the database.
>> Retrieving 1 from the database, call ID 0...
Retrieved: 1
Retrieving value of 1 for the 2nd time from DB 1 passing different call ID should NOT hit the database.
Retrieved: 1
Retrieving value of 1 for the 1st time from DB 2 should NOT hit the database.
Retrieved: 1
```

Changing the maximal length of a cache key

The maximal length of a cache key is 2048 characters by default.

To change the maximal length of a cache key, create a new instance of the `CacheKeyBuilder` class passing the new value of the maximal length to the constructor and assign the new instance to the `CachingServicesDefaultKeyBuilder` property.

CAUTION NOTE

If you need large cache keys, we suggest you also hash the cache key before submitting to the caching backend. To hash the cache key, implement a custom cache key builder (see below). The MD5 algorithm is generally a good choice given its speed, its collision probability, and the fact that its cryptographic strength is irrelevant in this case.

The following example sets the maximal length of a cache key to 4096 characters:

```
CachingServices.DefaultKeyBuilder = new CacheKeyBuilder(4096);
```

Implementing a custom cache key builder

All the options described above modify the behavior of formatting of a parameter value. To customize the other parts of the cache key, you can override the methods of the `CacheKeyBuilder` class.

To override the cache building logic

1. Create a new class and derive it from the `CacheKeyBuilder` class.
2. Override the virtual methods as needed.

To completely override the key building process, override the `BuildMethodKey(MethodInfo, IListObject, Object)` and/or `BuildDependencyKey(Object)` methods. The default implementation of these two methods uses `AppendType(UnsafeStringBuilder, Type)`, `AppendMethod(UnsafeStringBuilder, MethodInfo)`, `AppendArgument(UnsafeStringBuilder, Type, Object)` and `AppendObject(UnsafeStringBuilder, Object)` helper methods. If you need to override how a type, a method, an argument or an object is formatted, you can override just some of these methods.

3. Assign an instance of your new cache key builder to the `CachingServicesDefaultKeyBuilder` property.

CHAPTER 53

Caching Back-Ends

The `CacheAttribute` can be used with different caching frameworks or caching servers. This concept is called a *caching backend*. Caching backends are represented by the `CachingBackend` abstract class. You can use an existing implementation or implement your own caching backend.

In this chapter

Section	Description
Using In-Memory Cache on page 293	This article shows how to store cached values using <code>MemoryCache</code> class.
Using Redis Cache on page 293	This article shows how to store cached values using Redis and using a combination of Redis and <code>MemoryCache</code> class.

53.1. Using In-Memory Cache

For local, in-memory caching, PostSharp relies on the `MemoryCache` class of the .NET Framework.

To use the `MemoryCache` class to store cached values in memory, assign an instance of a `MemoryCachingBackend` class to `CachingServices.DefaultBackend` property.

```
CachingServices.DefaultBackend = new MemoryCachingBackend();
```

By default, the `MemoryCacheDefault` is used. To use other instance of the `MemoryCache` than the default one, an instance of the `MemoryCache` class can be passed to the constructor of the `MemoryCachingBackend` class.

```
MemoryCache cache = new MemoryCache( "myCache" );
CachingServices.DefaultBackend = new MemoryCachingBackend( cache );
```

See [MSDN](#)⁴³ for details on the `MemoryCache` class.

53.2. Using Redis Cache

[Redis](#)⁴⁴ is a popular choice for distributed, in-memory caching.

43. [https://msdn.microsoft.com/en-us/library/system.runtime.caching.memorycache\(v=vs.110\).aspx](https://msdn.microsoft.com/en-us/library/system.runtime.caching.memorycache(v=vs.110).aspx)

44. <https://redis.io/>

Our implementation uses [StackExchange.Redis library](#)⁴⁵ internally and is compatible with on-premises instances of Redis Cache as well as with the [Azure Redis Cache](#)⁴⁶ cloud service.

This topic contains the following sections:

- [Configuring the Redis server on page 294](#)
- [Configuring the caching backend in PostSharp on page 294](#)
- [Adding a local in-memory cache to a remote Redis server on page 294](#)
- [Using dependencies with the Redis caching backend on page 295](#)

Configuring the Redis server

To prepare your Redis server for use with PostSharp caching:

1. Set up the eviction policy to `volatile-lru` or `volatile-random`. See <https://redis.io/topics/lru-cache#eviction-policies> for details.

CAUTION NOTE

Other eviction policies than `volatile-lru` or `volatile-random` are not supported.

2. Set up the key-space notification to include the AKE events. See <https://redis.io/topics/notifications#configuration> for details.

Configuring the caching backend in PostSharp

To set up PostSharp to use Redis for caching:

1. Add a reference to the `PostSharp.Patterns.Caching.Redis` package.
2. Create an instance of `StackExchange.Redis.ConnectionMultiplexer`⁴⁷.
3. Create an instance of the `RedisCachingBackend` class using the `RedisCachingBackend.Create(IConnectionMultiplexer, RedisCachingBackendConfiguration)` factory method and assign the instance to `CachingServices.DefaultBackend`.

IMPORTANT NOTE

The caching backend has to be set before any cached method is called for the first time.

Example

```
string connectionConfiguration = "localhost";
ConnectionMultiplexer connection = ConnectionMultiplexer.Connect( connectionConfiguration );
RedisCachingBackendConfiguration redisCachingConfiguration = new RedisCachingBackendConfiguration();
CachingServices.DefaultBackend = RedisCachingBackend.Create( connection, redisCachingConfiguration );
```

Adding a local in-memory cache to a remote Redis server

For higher performance, you can add an additional, in-process layer of caching between your application and the remote Redis server. To enable the local cache, set the `RedisCachingBackendConfiguration.IsLocallyCached` property to `true`.

45. <https://stackexchange.github.io/StackExchange.Redis/>

46. <https://azure.microsoft.com/en-us/services/cache/>

47. <https://stackexchange.github.io/StackExchange.Redis/Configuration>

The benefit of using local caching is to decrease latency between the application and the Redis server, and to decrease CPU load due to the deserialization of objects. The inconvenience is that there is that distributed local caches are synchronized asynchronously, therefore different application instances may see different values of cache items during a few milliseconds. However, the application instance initiating the change will have a consistent view of the cache.

Using dependencies with the Redis caching backend

Support for dependencies is disabled by default with the Redis caching backend because it has an important performance and deployment impact. From a performance point of view, the cache dependencies need to be stored in Redis (therefore consuming memory) and handled in a transactional way (therefore consuming processing power). As for deployment, the problem is that the cache GC process, which cleans up dependencies when cache items are expired from the cache, needs to run continuously, even when the application is not running.

If you choose to enable dependencies with Redis, you need to make sure that there is at least one instance of the cache GC process is running. It is legal to have several instances of this process running, but since all instances will compete to process the same messages, it is better to ensure that only a small number of instances (ideally one) is running.

To use dependencies with the Redis caching backend:

- Make sure that at least one instance of the `RedisCacheDependencyGarbageCollector` class is alive at any moment (whenever the application is running or not). If several instances of your application use the same Redis server, a single instance of the `RedisCacheDependencyGarbageCollector` class is required. You may package the `RedisCacheDependencyGarbageCollector` into a separate application of cloud service.
- In case of an outage of the service running the GC process, execute the `PerformFullCollectionAsync(RedisCachingBackend, CancellationToken)` method.
- Set the `RedisCachingBackendConfigurationSupportsDependencies` property to `true`.

CHAPTER 54

Synchronizing Local In-Memory Caches for Multiple Servers

Caching in distributed applications can be a tricky problem. When there are several instances of an application running simultaneously (typically web sites or web services deployed into the cloud or web farms), you have to make sure that the cache is properly invalidated for all instances of the application.

A typical answer to this issue is to use a centralized cache server (or a cluster of cache servers) that solves this problem for you. For instance, you can use a Redis server or a Redis cluster. However, running a cache server, even more a cache cluster, comes with a cost, and it does not always pay off for medium applications such as the website of a small business.

An alternative solution to the problem of distributed caching is to have a local in-memory cache in each instance in the application. Instead of using a shared distributed cache, each application instance caches its own data into its own local cache. However, when one instance of the application modifies a piece of data, it needs to make sure that all instances remove the relevant items from their local cache. This is called *distributed cache invalidation*. It can be achieved easily and cheaply with a publish/subscribe (Pub/Sub) message bus such as Azure Service Bus, much less expensive than a cache cluster.

PostSharp allows you to easily add sub/sub cache invalidation to your existing PostSharp caching.

The principal inconvenience of pub/sub invalidation is that there is some latency in the invalidation mechanism, i.e. different instances of the application can see different data during a few dozens of milliseconds.

This topic contains the following sections:

- [Using Azure Service Bus pub/sub for distributed invalidation on page 297](#)
- [Using Redis pub/sub for distributed invalidation on page 298](#)

Using Azure Service Bus pub/sub for distributed invalidation

To use Azure Service Bus pub/sub for distributed invalidation:

1. Add in-memory local caching to your application as described in [Using In-Memory Cache on page 293](#).
2. Go to a Microsoft Azure portal, open the **Service Bus** panel and create a new **Topic**. Choose a small value for the time-to-live setting, for instance 30 seconds. See [Microsoft Azure website](#)⁴⁸ for details.
3. In the Microsoft Azure portal, create a **Shared access policy** and include the **Manage** right. This policy will be used by your application.
4. Go to the properties of the newly created policy and copy the primary or secondary connection string to the clipboard.
5. Go back to your source code and find the place where the `MemoryCachingBackend` is initialized.
6. Create an instance of `AzureCacheInvalidatorOptions` class and specify the connection string to the shared access policy you just created.

48. <https://azure.microsoft.com/en-us/services/service-bus/>

7. Create an instance of the `AzureCacheInvalidator` class using the `AzureCacheInvalidator.Create(CachingBackend, AzureCacheInvalidatorOptions)` factory method passing the existing instance of the `MemoryCachingBackend` and `AzureCacheInvalidatorOptions`. Assign the new `AzureCacheInvalidator` to the `CachingServices.DefaultBackend` property.

Example

This example shows how to initialize an in-memory caching backend to let it invalidate and be invalidated using Azure Service Bus Pub/Sub.

```
var localCache = new MemoryCachingBackend();

string connectionString = "Endpoint=sb://yourServiceNamespace.servicebus.windows.net/;EntityPath=yourTopic;SharedAccessKeyI

var azureCacheInvalidatorOptions = new AzureCacheInvalidatorOptions
    {
        ConnectionString = connectionString
    };

CachingServices.DefaultBackend = AzureCacheInvalidator.Create( localCache, azureCacheInvalidatorOptions );
```

Using Redis pub/sub for distributed invalidation

If you are already using Redis for PostSharp caching, it is useless to add another layer of invalidation because this is already taken care of by the `RedisCachingBackend` class. However, if you already have a Redis cluster but you don't want to use it for caching, you can still use it for cache invalidation. An example situation is when the latency of your Redis server is too high for caching but sufficient for cache invalidation.

To use Redis Pub/Sub for distributed invalidation:

1. Add in-memory local caching to your application as described in [Using In-Memory Cache on page 293](#).
2. Create an instance of `StackExchange.Redis.ConnectionMultiplexer`⁴⁹. See [Redis Pub/Sub documentation](#)⁵⁰ for details.
3. Create and configure an instance of `RedisCacheInvalidatorOptions` class.
4. Create an instance of the `RedisCacheInvalidator` class using the `RedisCacheInvalidator.Create(CachingBackend, IConnectionMultiplexer, RedisCacheInvalidatorOptions)` factory method passing your instance of `MemoryCachingBackend` and `RedisCacheInvalidatorOptions`. Assign the instance to the `CachingServices.DefaultBackend` property.

Example

This example shows how to initialize an in-memory caching backend to let it invalidate and be invalidated using Redis Pub/Sub.

```
var localCache = new MemoryCachingBackend();

string connectionConfiguration = "localhost";
string channelName = "myCahnnel";

ConnectionMultiplexer connection = ConnectionMultiplexer.Connect( connectionConfiguration );

var redisCacheInvalidatorOptions = new RedisCacheInvalidatorOptions
    {
        ChannelName = channelName
    };

CachingServices.DefaultBackend = RedisCacheInvalidator.Create( localCache, connection, redisCacheInvalidatorOptions );
```

49. <https://stackexchange.github.io/StackExchange.Redis/Configuration>

50. <https://redis.io/topics/pubsub/>

CHAPTER 55

Caching Special Types with Value Adapters

Some types, for instance the `IEnumeratorT` interface or the `Stream` class cannot be directly cached because the position of the enumerator or stream can be changed by the caller. Some other interfaces like `IEnumerableT` cannot be cached because the real value may be a LINQ expression, and it is not useful to cache the LINQ expression itself. However, you may still want to cache methods returning these types. If you wrote the code manually, you would simply cache a different type, for instance a list or an array of bytes.

PostSharp addresses this problem by the concept of a *value adapter*. A value adapter allows you to store another type than the one than the return type of the cached method. The method return value is called the *exposed value* because this is the value exposed by your API. The exposed value must be type-compatible with the method return type. The value that is actually stored in cache is called the *stored value*. For instance, for a method returning a `Stream`, the stored value is an array of bytes and the exposed value is a `MemoryStream`.

This topic contains the following sections:

- [Standard value adapters on page 299](#)
- [Implementing a custom value adapter on page 299](#)

Standard value adapters

The following value adapters are used automatically by default:

Return type	Stored type	Exposed type	Comments
<code>IEnumerableT</code>	<code>ListT</code>	<code>ListT</code>	
<code>IEnumeratorT</code>	<code>ListT</code>	<code>System.Collections.Generic.ListTEnumerator</code>	The <code>IEnumerator.Reset</code> method is not supported by the exposed value.
<code>Stream</code>	<code>Byte[]</code>	<code>MemoryStream</code>	

Implementing a custom value adapter

To implement a custom value adapter:

1. Add a reference to the `PostSharp.Patterns.Caching` package.
2. Create a class implementing the `IValueAdapterT` interface or the `IValueAdapter` interface.
3. Register the class from the previous step using one of the overloads of the `ValueAdapterFactoryRegister` method.

Each caching backend has its own instance of the `PostSharp.Patterns.Caching.ValueAdaptersValueAdapterFactory` class available via the `ValueAdapters` property.

NOTE

Null values are handled automatically outside of the value adapters.

Example

In this example, we create the `EnumerableValueAdapter`1` class, which transforms instances of the `IEnumerable<T>` interface into an array.

```
using System;
using System.Collections.Generic;
using System.Linq;
using PostSharp.Patterns.Caching;
using PostSharp.Patterns.Caching.Backends;
using PostSharp.Patterns.Caching.ValueAdapters;

namespace PostSharp.Samples.Caching.ValueAdapters
{
    class EnumerableValueAdapter<T> : ValueAdapter<IEnumerable<T>>
    {
        public override IEnumerable<T> GetExposedValue(object storedValue)
        {
            return (IEnumerable<T>)storedValue;
        }

        public override object GetStoredValue(IEnumerable<T> value)
        {
            Console.WriteLine("Caching enumerable.");
            return value.ToArray();
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            MemoryCachingBackend memoryCachingBackend = new MemoryCachingBackend();
            memoryCachingBackend.ValueAdapters.Register(typeof(IEnumerable<>), typeof(EnumerableValueAdapter<>));
            CachingServices.DefaultBackend = memoryCachingBackend;

            Console.WriteLine("Cache miss:");
            foreach (int value in GetValues())
            {
                Console.WriteLine("Value {0} obtained.", value);
            }

            Console.WriteLine("Cache hit:");
            foreach (int value in GetValues())
            {
                Console.WriteLine("Value {0} obtained.", value);
            }

            Console.WriteLine("Cache miss for null:");
            Console.WriteLine(GetNull() == null);

            Console.WriteLine("Cache hit for null:");
            Console.WriteLine(GetNull() == null);
        }

        [Cache]
        static IEnumerable<int> GetValues()
        {
            for (int value = 0; value < 3; value++)
            {
                Console.WriteLine("Returning value {0}.", value);
                yield return value;
            }
        }

        [Cache]
        static IEnumerable<int> GetNull()
        {
            return null;
        }
    }
}
```

```
    }  
}
```

The output of this sample is:

```
Cache miss:  
Caching enumerable.  
Returning value0.  
Returning value1.  
Returning value2.  
Value 0 obtained.  
Value 1 obtained.  
Value 2 obtained.  
Cache hit:  
Value 0 obtained.  
Value 1 obtained.  
Value 2 obtained.  
Cache miss fornull:  
True  
Cache hit fornull:  
True
```


CHAPTER 56

Preventing Concurrent Execution of Cached Methods

When the evaluation of a method consumes significant resources or time, you may want to prevent a situation where several threads, processes or machines are evaluating the same method with the same parameters at the same time. You can achieve it by instructing PostSharp to use a lock manager. PostSharp implements two lock managers: the default `NullLockManager`, and `LocalLockManager`.

This topic contains the following sections:

- [Preventing concurrent execution in the current process on page 303](#)
- [Handling lock timeouts on page 303](#)
- [Implementing a distributed lock manager on page 304](#)

Preventing concurrent execution in the current process

By default, the caching aspect allows concurrent execution of the same method with the same arguments.

The `LocalLockManager` class implements that is able to prevent execution of methods running in the current process (or `AppDomain`, to be exact).

To configure the lock manager, you have to set the `CachingProfileLockManager` property. Each caching profile must be set up separately.

The following code shows how to configure locking for two profiles:

```
CachingServices.Profiles.Default.LockManager = new LocalLockManager();
CachingServices.Profiles["MyProfile"].LockManager = new LocalLockManager();
```

NOTE

Each instance of the `LocalLockManager` class maintains its own set of locks. However, whether several profiles use the same or a different instance of the `LocalLockManager` does not matter because each method is associated with one and only one profile.

Handling lock timeouts

By default (unless you use the default `NullLockManager`), the caching aspect will wait for a lock during an infinite amount of time. Suppose that the thread that evaluates the method gets stuck (e.g. it is involved in a deadlock). Because of the locking mechanism, all threads evaluating the same method will also get stuck. To avoid this situation, you can choose to implement a timeout behavior.

Two properties influence the timeout behavior:

Property	Description
CachingProfileAcquireLockTimeout	Maximum time that the caching aspect will wait for the lock manager to acquire a lock. To specify an infinite waiting time, set this property to <code>TimeSpan.FromMilliseconds(-1)</code> . The default behavior is to wait infinitely.
CachingProfileAcquireLockTimeoutStrategy	Implements the logic executed when the caching aspect could not acquire a lock because of a timeout. The default behavior is to throw a <code>TimeoutException</code> . You can implement your own strategy by implementing the <code>IAcquireLockTimeoutStrategy</code> interface.

NOTE

This section only covers the time it takes to acquire a lock. It does not cover the execution time of the method that has already acquired the lock.

The following code shows how to set a 10-second timeout and ignore any timeout situation.

```
CachingServices.Profiles.Default.AcquireLockTimeout = TimeSpan.FromSeconds(10);
CachingServices.Profiles.Default.AcquireLockTimeoutStrategy = new IgnoreLockStrategy();
```

Here is the code of the `IgnoreLockStrategy` class:

```
publicclass IgnoreLockStrategy : IAcquireLockTimeoutStrategy
{
    publicvoid OnTimeout( string key )
    {
        // The cacheable method will be evaluated regardless of our inability to acquire a lock, // unless we throw an excep
    }
}
```

Implementing a distributed lock manager

Implementing a distributed locking algorithm is a highly complex task and we at PostSharp decided not to get involved in this business (just as we do not provide the implementation of a cache itself). However, PostSharp gives you the ability to use any third-party implementation.

To create make your lock manager work with the caching aspect, you should implement the `ILockManager` and `ILockHandle` interfaces.

CHAPTER 57

Troubleshooting Caching

If you need to troubleshoot the caching aspect, you can enable logging for this feature. The procedure is different whether or not you are using PostSharp Logging in your application.

This topic contains the following sections:

- [Enabling logging of caching with PostSharp Logging on page 305](#)
- [Enabling logging of caching with System.Diagnostics on page 305](#)

Enabling logging of caching with PostSharp Logging

If you use PostSharp Logging for logging, you can enable detailed logging of the caching aspect by enabling the Caching logging role for the whole application or a specific type or namespace.

The following code shows how to log the details of the caching aspect to the system console for the whole application:

```
LoggingServices.DefaultBackend = new ConsoleLoggingBackend();  
LoggingServices.DefaultBackend.DefaultVerbosity.SetMinimalLevel(LogLevel.Debug, LoggingRoles.Caching);
```

Enabling logging of caching with System.Diagnostics

If you are not using PostSharp Logging, you can enable logging of the caching aspect by configuring the TraceSource named PostSharp.Cache in you app.config or web.config file:

```
<?xmlversion="1.0"encoding="utf-8"?><configuration><startup><supportedRuntimeversion="v4.0"sku=".NETFramework,Version=v4.7
```


PART 12

Multithreading

CHAPTER 58

Writing Thread-Safe Code with Threading Models

A threading model is a design pattern that gives guarantees that your code executes safely on a multithreaded computer. Threading models both define coding rules (for instance: all fields must be private) and add new behaviors to existing code (for instance: acquiring a lock before method execution). Coding rules are typically enforced at build time or at run time; violations result in build-time errors or run-time exceptions. Threading models may also require the use of custom attributes in source code, for instance to indicate that a method requires read access to the object.

TIP

We recommend assigning a threading model to every class whose instances can be shared between different threads.

Threading models raise the level of abstraction at which multi threading is addressed. Compared to working directly with locks and other low-level threading primitives, using threading models has the following benefits:

- Threading models are **named solutions** to a recurring problem. Threading models are specific types of design patterns, and have the same benefits. When team members discuss the multithreaded behavior of a class, they just need to know which threading model this class uses. They don't need to know the very details of its implementation. Since the human short-term memory seems to be limited to 5-9 elements, it is important to think in terms of larger conceptual blocks whenever we can.
- Much of the code required to implement the threading model can be **automatically generated**, which decreases the number of lines of code, and therefore the number of defects. It also reduces development and maintenance costs.
- Your source code can be **automatically verified** against the selected threading model, both at build time and at run time. This makes the discovery of defect much more deterministic. Without verifications, threading defects usually show up randomly and provoke data structure corruption instead of immediate exceptions. Run-time verification would be too labor-intensive to implement without compiler support, so would be most likely omitted.

Available threading models

PostSharp Threading Library provides an implementation for the following threading models:

Threading Model	Aspect Type	Description
Thread-Unsafe Threading Model on page 328	ThreadUnsafeAttribute	These objects may never be accessed concurrently by several threads.
Thread Affine Threading Model on page 316	ThreadAffineAttribute	These objects must be accessed from the thread that instantiated them.

Threading Model	Aspect Type	Description
Synchronized Threading Model on page 317	SynchronizedAttribute	Synchronized objects can be accessed by a single thread at a time. Other threads will wait until the object is available.
Reader/Writer Synchronized Threading Model on page 319	ReaderWriterSynchronizedAttribute	These objects that can be read concurrently by several threads, but write access requires exclusivity. Public methods of this object must specify which kind of access they require (read or write, typically).
Actor Threading Model on page 325	ActorAttribute	These objects communicate with their clients using an asynchronous communication pattern. All accesses to the object are queued and then processed in a single thread. However, queuing is transparent to clients, which just call standard void or methods.
Freezable Threading Model on page 314	FreezableAttribute	These objects can be set to a state where their property values can no longer be changed. Unlike immutable objects, the developer dictates the time and place in their code where changes to the object's state will no longer be accepted.
Immutable Threading Model on page 310	ImmutableAttribute	These objects cannot have their state changed after their constructor has finished executing.

Other topics

Article	Description
Making a Whole Project or Solution Thread Safe on page 329	This article describes how to get compiler warnings when you forget to assign a threading model to a type.
Opting In and Out From Thread Safety on page 331	This article shows how to disable the enforcement of the threading model for specific fields or methods.
Compatibility of Threading Models on page 332	This article lists compatibility of threading models when they are applied to objects that are in a parent-child relationship.
Enabling and Disabling Runtime Verification on page 333	This article explains when runtime verification is enabled or disabled and how to customize the default behavior.

Conceptual documentation

Please read [this technical white paper](#)⁵¹ for details about the concepts and architecture of PostSharp Threading Models.

58.1. Immutable Threading Model

There are times when you want certain objects in your codebase to retain their post creation state without the possibility of it ever changing. These objects are said to be immutable. Immutable objects are useful in multithreaded applications

51. <https://www.postsharp.net/links/threading-model-white-paper>

because they can be safely accessed by several threads concurrently, without the need for locking or other synchronization. PostSharp offers the `ImmutableAttribute` aspect that allows you enforce this pattern on your objects.

Changes in an object with the `ImmutableAttribute` aspect will be forbidden as soon as the object constructor exits. Any further attempt to modify the object will result in an `ObjectReadOnlyException`.

NOTE

The Immutable pattern can be too strong for some common object-oriented scenarios, for instance with serializable classes. In some cases, the `Freezable` object is a better choice. For details, see [Freezable Threading Model on page 314](#).

This topic contains the following sections:

- [Making a class immutable on page 311](#)
- [Rules enforced by the Immutable aspect on page 311](#)
- [Immutable object trees on page 311](#)
- [Immutable vs readonly on page 312](#)
- [Constructor execution on page 313](#)
- [Immutable collections on page 313](#)

Making a class immutable

To add the Immutable pattern manually:

1. Add a reference to the `PostSharp.Patterns.Threading` package to your project.
2. Add the `ImmutableAttribute` custom attribute to your class.
3. Annotate your object model for parent/child relationships as described in [Annotating an Object Model for Parent/Child Relationships \(Aggregatable\) on page 241](#).

Rules enforced by the Immutable aspect

An immutable object will throw the following exceptions at run-time:

- `ThreadMismatchException` if both following conditions are simultaneously true:
 - you access the object from a different thread than the one that created it, and
 - the constructor has not completed yet.
- `ObjectReadOnlyException` if a field or property is being modified after the constructor has completed.

Immutable object trees

Because the Immutable pattern is an implementation of the `Aggregatable` pattern, all of the same behaviors of the `AggregatableAttribute` are available. As a result, you can create both immutable classes and immutable object trees. For more information regarding object trees, read [Parent/Child, Visitor and Disposable on page 239](#).

NOTE

Children of immutable objects must be marked as `Immutable` or `Freezable` themselves. Adding `ImmutableAttribute` or `FreezableAttribute` to the child classes will accomplish this. `Freezable` children will be automatically frozen when the constructor of the parent completes.

Immutable vs readonly

Many C# developers make use of the `readonly` keyword in an attempt to make their objects immutable. The `readonly` keyword doesn't guarantee immutability though. Using `readonly` only ensures that no method other than the object's constructor can alter the variable's value. It doesn't, however, prevent you from altering values on complex objects outside of the constructor.

The `readonly` keyword may be too strict

In the following code sample, the `_id` variable is a primitive type and can't be altered outside the constructor. This is enforced at compile time and an error would be displayed where the `SetIdentifier` method attempts to change the `_id` field value. The compiler does not see that the `SetIdentifier` method is called only from the constructor. In this example, the `readonly` keyword is too strong even if the class is legitimately immutable.

```
publicclass Invoice
{
    publicreadonlylong _id;

    public Invoice(long id)
    {
        SetIdentifier(id);
    }

    privatevoid SetIdentifier(long id)
    {
        // Will cause compilation error.
        _id = id;
    }
}
```

The `readonly` keyword may also be too loose

When you have complex entities composed of several objects, immutability is a characteristic of the whole entity, not of a single object. However, this does not fit with the `readonly` keyword.

In the following example, the `Invoice` is not immutable even if the `_invoiceHeader` field is `readonly`.

```
publicclass Invoice
{
    publicreadonly InvoiceHeader _invoiceHeader;

    public Invoice()
    {
        _invoiceHeader = new InvoiceHeader();
    }

    publicvoid Refresh()
    {
        // Valid but not immutable.
        _invoiceHeader.CustomerName = "Jim";
        _invoiceHeader.CustomerPhone = "555-123-9876";
    }
}
```

The same type of change to object state can happen with collections. You can not reinitialize a `readonly` collection, but you can freely `Add`, `Remove`, `Clear` and do other operations that the collection itself exposes. Additionally, if the collection contains complex types you are able to change values on each instance that the collection contains.

```
publicclass Invoice
{
    publicreadonly IList<Item> _items;
    public Invoice()
    {
        _items = new List<Item>();
    }
}
```



```

public void Refresh()
{
    //will cause a compilation error
    _items = new List<Item>();

    //valid but not immutable
    _items.Add(new Item());
    _items[0].Price = 3.50f;
    _items.RemoveAt(0);
}
}

```

As you can see there is no way to use the `readonly` keyword to make complex object graphs immutable. Combining the `ImmutableAttribute`, `ChildAttribute`, `AdvisableCollectionT` and the `AdvisableDictionaryTKey, TValue` types allows you to make immutable objects that guarantee no changes to primitive or complex objects after constructor execution has completed.

Constructor execution

Objects are not frozen until the last constructor has finished executing. Because of this, you can use the constructor to set up the state of the parent instance through its own constructor as well as chained, or inherited object, constructors. You're also able to make changes to child object instances through their constructors at this time.

```

[Immutable]
public class Invoice : Document
{
    public Invoice(long id) : base(id)
    {
        Items = new AdvisableCollection<Item>();
        Items.Add(new Item("widget"));
    }

    [Child]
    public AdvisableCollection<Item> Items { get; set; }
}

[Immutable]
public class Document
{
    private long _id;
    public Document(long id)
    {
        _id = id;
    }
}

[Immutable]
public class Item
{
    public Item(string name)
    {
        Name = name;
    }
    public string Name { get; set; }
}

```

In this example, the constructors finish executing in the order of `Document`, `Item` and finally `Invoice`. It is not until after the `Invoice` constructor finishes executing that the object graph is made immutable.

Immutable collections

When authoring immutable object models, immutable collections are a good replacement for advisable collections. Immutable collections are implemented in the `System.Collections.Immutable` namespace, contained in the `System.Collections.Immutable` NuGet package.

58.2. Freezable Threading Model

When you need to prevent changes to an instance of an object most of the time, but not all of the time, the `Immutable` pattern (implemented by the `ImmutableAttribute` aspect) will be too aggressive for you. In these situations, you need a pattern that allows you to define the point in time where immutability begins. To accomplish this you can make use of the `FreezableAttribute` aspect.

Changes in an object with the `ImmutableAttribute` aspect will be forbidden as soon as you call the `Freeze` method. Any further attempt to modify the object will result in an `ObjectReadOnlyException`. Any attempt to share the object before you call the `Freeze` method will result in a `ThreadMismatchException`.

This topic contains the following sections:

- [Making an object freezable on page 314](#)
- [Freezing an object on page 314](#)
- [Rules enforced by the Freezable aspect on page 315](#)
- [Determining whether an object is in frozen state on page 315](#)
- [Freezable object trees on page 315](#)

To make an object freezable, all you need to do is add the `FreezableAttribute` attribute to the class in question.

Making an object freezable

To make an object freezable:

1. Add the `PostSharp.Patterns.Threading` package to your project using NuGet.
2. Add the `FreezableAttribute` custom attribute to your class.

```
using PostSharp.Patterns.Threading;

[Freezable]
publicclass Invoice
{
    publiclong Id { get; set; }
}
```

3. Annotate your object model for parent/child relationships as described in [Annotating an Object Model for Parent/Child Relationships \(Aggregatable\) on page 241](#).

Freezing an object

To freeze an object, you will first have to cast the object to the `IFreezable` interface. After that, you will be able to call the `Freeze` method.

```
var invoice = new Invoice();
invoice.Id = 123456;

((IFreezable)invoice).Freeze();
```

NOTE

The `IFreezable` interface will be injected into the `Invoice` class *after* compilation. Tools that are not aware of PostSharp may incorrectly report that the `Invoice` class does not implement the `IFreezable` interface.

Instead of using the cast operator, you can also use the `CastTSource, TTarget(TSource)` method. This method is faster and safer than the cast operator because it is verified and compiled by PostSharp at build time.

NOTE

If you are attempting to freeze either `AdvisableCollectionT` or `AdvisableDictionaryTKey, TValue` you will not be able to use the cast operator or the `CastTSource, TTarget(TSource)` method. Instead, you will have to use the **QueryInterface`1(Object, Boolean)** extension method.

Once you've called the `Freeze` method on an object instance the code will no longer be able to change the property values on that instance. If a value change is attempted the code will throw an `ObjectReadOnlyException`.

```
var invoice = new Invoice();
invoice.Id = 123456;

((IFreezable)invoice).Freeze();

// This will throw an exception.
invoice.Id = 345678;
```

Rules enforced by the Freezable aspect

A freezable object will throw the following exceptions at run-time:

- `ThreadMismatchException` if both following conditions are simultaneously true:
 - you access the object from a different thread than the one that created it, and
 - the `Freeze` method has not yet been called.
- `ObjectReadOnlyException` if a field or property is being modified after the `Freeze` method has been called.

Determining whether an object is in frozen state

To determine whether an object has been frozen, cast it to `IThreadAware` and get the `readonly` value from `IsReadOnly` via the `ConcurrencyController` property.

```
var invoice = new Invoice();
invoice.Id = 123456;

((IFreezable)invoice).Freeze();

// The 'frozen' property will be set to 'true'.bool frozen = ((IThreadAware)invoice).ConcurrencyController.IsReadOnly;
```

Freezable object trees

The `Freezable` pattern relies on the `Aggregatable` pattern. The `AggregatableAttribute` aspect will be implicitly added to the target class. Therefore, you can not only create freezable classes, but also freezable object trees. Read the [Parent/Child, Visitor and Disposable on page 239](#) for more information on how to establish object trees.

IMPORTANT NOTE

Children of freezable objects must be either freezable or immutable. Therefore, children classes must be annotated with the `FreezableAttribute` or `ImmutableAttribute` custom attribute. Collection types must be derived from `AdvisableCollectionT` or `AdvisableDictionaryTKey, TValue`. Arrays cannot be used.

58.3. Thread Affine Threading Model

One of the simplest ways to consider threading is to limit object instance access to the thread that created the instance. This is how the Thread Affine threading model works.

This topic contains the following sections:

- [Adding the Thread Affine model to a class on page 316](#)
- [Rules enforced by the Actor aspect on page 316](#)
- [Working with object trees on page 317](#)

Adding the Thread Affine model to a class

To apply the Thread-Affine threading model to a class:

1. Add the *PostSharp.Patterns.Threading* package to your project.
2. Add `using PostSharp.Patterns.Threading` namespace to your file.
3. Add the `ThreadAffineAttribute` to the class.
4. Annotate your object model for parent/child relationships as described in [Annotating an Object Model for Parent/Child Relationships \(Aggregatable\) on page 241](#).

Example

In the example below the `ThreadAffineAttribute` has been added to the class.

```
[ThreadAffine]
publicclass OrderService
{
    publicvoid Process(int sequence)
    {
        Console.WriteLine("sequence {0}", sequence);
        Console.WriteLine("sleeping for 10s");

        Thread.Sleep(new TimeSpan(0,0,10));
    }
}
```

Rules enforced by the Actor aspect

The `ThreadAffineAttribute` aspect does not verify your code at build-time. Instead, it injects code that enforces the model at run time. If it detects that the object is being accessed from a different thread than the one that created it, the aspect will throw a `ThreadMismatchException` exception.

To test this the thread-affine `OrderService` class, we can run the following code:

```
publicvoid Main()
{
    var orderService = new OrderService();

    orderService.Process(1);

    var backgroundWorker = new BackgroundWorker();
    backgroundWorker.DoWork += (sender, args) =>
    {
        try
        {
            orderService.Process(2);
        }
        catch (Exception ex)
        {
            Console.WriteLine(ex.ToString());
        }
    }
}
```

```

    }
};
backgroundWorker.RunWorkerAsync();
}

```

The above code will execute the `orderService.Process(1)` method and output the following to the console.

```

sequence 1
sleeping for 10s

```

That code successfully executed because the `orderService` instance was both created (via the `new` keyword) and executed on the same thread.

After the 10 second sleep period, a `BackgroundWorker` thread is opened and it is set to execute the `orderService.Process(2)` method. If an exception is thrown that will be output to the console. When this piece of code executes you will see the following console output.

```

PostSharp.Patterns.Threading.ThreadMismatchException: An attempt was made to access the object from an invalid thread.
   at PostSharp.Patterns.Threading.Controllers.ThreadAffineController.AcquireAccessCore(ObjectAccessLevel objectAccessLevel, ConcurrentAccessToken& concurrentAccessToken)
   at PostSharp.Patterns.Threading.Controllers.ConcurrencyController.AcquireAccess(ObjectAccessLevel objectAccessLevel, ConcurrentAccessToken& concurrencyAccessToken)
   at PostSharp.Patterns.Threading.ThreadAwareAttribute.OnMethodEntry(MethodExecutionArgs args)
   at Postsharp.OrderService.Process(Int32 sequence) in e:\working\postsharp3.1\writing\test\OrderService.cs:line 16
   at Postsharp.Program.<>c__DisplayClass1.<Synchronized>b__0(Object sender, EventArgs args) in e:\working\postsharp3.1\writing\test\Program.cs:line 34

```

As you can see that a `ThreadMismatchException` exception was thrown. This happened because the `orderService` instance was created on the main thread and the `BackgroundWorker` thread attempted to execute it. Because the `OrderService` class has been marked with the `ThreadAffineAttribute` attribute only the thread that creates an instance of it can access that instance.

Working with object trees

Because the Thread-Affine model is an implementation of the `Aggregatable` pattern, all of the same behaviors of the `AggregatableAttribute` are available. For more information regarding object trees, read [Parent/Child, Visitor and Disposable](#) on page 239.

NOTE

Once you have established your parent-child relationships you will need to apply compatible threading models to the child classes. You will want to refer to the [Compatibility of Threading Models on page 332](#) article to determine which threading model will work for the children of the Thread-Affine object.

58.4. Synchronized Threading Model

A common way to avoid data races is to enclose all public instance methods of a class with a `lock(this)` statement. This is basically what the Synchronized model does

This article describes how to use the Synchronized model and how it differs from the use of `lock(this)` statement.

This topic contains the following sections:

- [Comparing with lock\(this\) on page 318](#)
- [Applying the Synchronized model to a class on page 318](#)

- [Rules enforced by the Synchronized aspect on page 319](#)
- [Working with object trees on page 319](#)

Comparing with lock(this)

Traditionally, the C# keyword `lock(this)` statement has been used to synchronize access of several threads to a single object. When an object is locked by one thread, any other object that attempts to access that object will have its execution blocked.

```
private object myLockingObject = new Object();

public void DoSomething()
{
    lock(myLockingObject)
    {
        //some code that does something in one thread at a time
    }
}
```

In this example, the `myLockingObject` member variable is used as a locking object. Once a thread runs the `lock(myLockingObject)` line, all other threads that enter the `DoSomething` method will stop executing, or be blocked, until the original thread has exited the `lock(myLockingObject)` code block.

The `Synchronized` model is similar to using the `lock` statement around every single public method, but it has the following differences:

- It is not technically equivalent to locking the current instance (`this`). Another object is actually being locked.
- Locking is automatic for all public and internal instance methods. You cannot forget it.
- If a thread attempts to access a field without having first acquired access to the object (by invoking a public or internal method), an exception will be thrown.
- The pattern also works with entities composed of several objects organized in a tree.

Applying the Synchronized model to a class

To apply the `Synchronized` threading model to a class:

1. Add the `PostSharp.Patterns.Threading` package to your project.
2. Add `using PostSharp.Patterns.Threading` namespace to your file.
3. Add the `SynchronizedAttribute` to the class.
4. Annotate your object model for parent/child relationships as described in [Annotating an Object Model for Parent/Child Relationships \(Aggregatable\) on page 241](#).

Example

In the example below the `SynchronizedAttribute` has been added to the class.

```
[Synchronized]
public class OrderService
{
    public void Process(int sequence)
    {
        Console.WriteLine("sequence {0}", sequence);
        Console.WriteLine("sleeping for 10s");

        Thread.Sleep(new TimeSpan(0,0,10));
    }
}
```

To test this we can run the following code.

```
public void Main()
{
    var orderService = new OrderService();

    var backgroundWorker = new BackgroundWorker();
    backgroundWorker.DoWork += (sender, args) => orderService.Process(1);
    backgroundWorker.RunWorkerAsync();

    orderService.Process(2);
}
```

The code above will attempt to execute the Process method on two different threads; the main thread and a background worker thread. Because these two threads are trying to access the same instance of the OrderService the first thread to access it will block the second. As a result, when you run the program you will first see the following.

```
sequence 2
sleeping for 10s
```

Because the OrderService.Process method has a Thread.Sleep call, the first thread accessing that method will block the second for 10 seconds. After those 10 seconds have passed the second thread will no longer be blocked and it will be able to continue its execution.

```
sequence 2
sleeping for 10s
sequence 1
sleeping for 10s
```

Rules enforced by the Synchronized aspect

The SynchronizedAttribute aspect emits build-time errors in the following situations:

- The class contains a public or internal field.

A synchronized object will throw a ThreadAccessException whenever some code tries to access a field from a thread that does not own the correct lock, i.e. the call stack does not contain a public or internal method of this method (e.g. a private delegate call).

Working with object trees

Because the Synchronized model is an implementation of the Aggregatable pattern, all of the same behaviors of the AggregatableAttribute are available. For more information regarding object trees, read [Parent/Child, Visitor and Disposable on page 239](#).

NOTE

Once you have established your parent-child relationships you will need to apply compatible threading models to the child classes. You will want to refer to the [Compatibility of Threading Models on page 332](#) article to determine which threading model will work for the children of the Synchronized object.

58.5. Reader/Writer Synchronized Threading Model

When a class instance is concurrently used by multiple threads, accesses must be synchronized to prevent data races, which typically result in data inconsistencies and corruption of data structures. The Reader/Writer Synchronized Threading Model

uses locks to allow several read-only methods to execute simultaneously on one instance, but guarantee that writer methods have exclusive access.

The Reader/Writer Synchronized Threading Model is implemented by the `ReaderWriterSynchronizedAttribute` aspect. It requires you to annotate all public methods of your synchronized classes with the `ReaderAttribute` and `WriterAttribute` custom attributes.

This topic contains the following sections:

- [Why use the reader-writer synchronized pattern? on page 320](#)
- [Making a class reader-writer synchronized on page 321](#)
- [Rules enforced by the ReaderWriterSynchronized aspect on page 322](#)
- [Raising synchronous events on page 322](#)
- [Executing long-running write methods on page 323](#)
- [Working with object trees on page 324](#)

Why use the reader-writer synchronized pattern?

Problems without locks

Consider the following example of an `Order` class which stores an amount and a discount:

```
class Order
{
    int Amount { get; private set; }
    int Discount { get; private set; }

    public int AmountAfterDiscount
    {
        get { return this.Amount - this.Discount; }
    }

    public void Set(int amount, int discount)
    {
        if (amount < discount)
            throw new InvalidOperationException();

        this.Amount = amount;
        this.Discount = discount;
    }
}
```

In this example, the `Set` method writes to the `Amount` and `Discount` members, while the `AmountAfterDiscount` property reads these members. In a single-threaded program, the `AmountAfterDiscount` property is guaranteed to be positive or zero. However, in a multithreaded program, the `AmountAfterDiscount` property could be evaluated in the middle of the `Set` operation, and return an inconsistent result.

Problems of the lock keyword

The easiest way to synchronize accesses to a class in C# is to use the `lock` keyword. However, this practice cannot be generalized for two reasons:

- The use of exclusive locks often results in high contention and therefore low performance because many threads queue to access the same resource;
- Applications relying on exclusive locks are prone to deadlocks because of cyclic waiting dependencies.

Problems of reader-writer locks

Reader-writer locks take advantage of the fact that most applications involve much fewer writes than reads, and that concurrent reads are always safe. Reader-writer locks ensure that no other thread is accessing the object when it is being written. Reader-writer locks are normally implemented by the .NET classes `ReaderWriterLock` or `ReaderWriterLockSlim`. The following example shows how `ReaderWriterLockSlim` would be used to control reads and writes in the `Order` class:


```

class Order
{
    private ReaderWriterLockSlim orderLock = new ReaderWriterLockSlim();

    public decimal Amount { get; private set; }
    public decimal Discount { get; private set; }

    public decimal AmountAfterDiscount
    {
        get
        {
            orderLock.EnterReadLock();
            decimal result = this.Amount - this.Discount;
            orderLock.ExitReadLock();
            return result;
        }
    }

    public void Set(decimal amount, decimal discount)
    {
        if (amount < discount)
        {
            throw new InvalidOperationException();
        }

        orderLock.EnterWriteLock();
        this.Amount = amount;
        this.Discount = discount;
        orderLock.ExitWriteLock();
    }
}

```

However, working directly with the `ReaderWriterLock` and `ReaderWriterLockSlim` classes has disadvantages:

- It is cumbersome because a lot of code is required.
- It is unreliable because it is too easy to forget to acquire the right type of lock, and these errors are not detectable by the compiler or by unit tests.

So, not only the direct use of locks results in more lines of code, but it won't reliably prevent nondeterministic data structure corruptions.

Making a class reader-writer synchronized

PostSharp Threading Pattern Library has been designed to eliminate nondeterministic data corruptions while reducing the size of thread synchronization code to the absolute minimum (but not less).

The `ReaderWriterSynchronizedAttribute` aspect implements the threading model (or threading pattern) based on the reader-writer lock, with the following principles:

- At any time, the object can be open for reading or closed for reading.
- Methods define their required access level using `ReaderAttribute` and `WriterAttribute` custom attributes (other access levels exist for advanced scenarios)
- An error will be emitted at build-time or runtime, but deterministically, whenever an object field is being accessed by a method that does not have the required access level on the object.

To apply the `ReaderWriterSynchronized` threading model to a class:

1. Add the `PostSharp.Patterns.Threading` package your project.
2. Add using `PostSharp.Patterns.Threading` namespace to your file.
3. Add the custom attribute `[ReaderWriterSynchronizedAttribute]` to the class.

4. Annotate your object model for parent/child relationships as described in [Annotating an Object Model for Parent/Child Relationships \(Aggregatable\)](#) on page 241.
5. Add the custom attribute `[ReaderAttribute]` or `[WriterAttribute]` to the public and internal methods. Note that it is not necessary to put these attributes on property getters and setters or on events.

The `ReaderAttribute` attribute causes PostSharp to acquire a lock on the instance whenever the method is invoked. While this lock is held, other threads can also read properties or invoke read-only methods of that instance, but calls to properties or methods marked with `WriterAttribute` will be blocked until all reads are complete.

Likewise, invoking methods marked with `WriterAttribute` will lock the instance causing all reads and writes to block until the write has completed and the write lock has been released.

Example

The following code shows the `Order` class, synchronized with the reader-writer threading pattern:

```
[ReaderWriterSynchronized]
class Order
{
    decimal Amount { get; private set; }
    decimal Discount { get; private set; }

    public decimal AmountAfterDiscount
    {
        get { return this.Amount - this.Discount; }
    }

    [Writer]
    public void Set(decimal amount, decimal discount)
    {
        if (amount < discount)
            throw new InvalidOperationException();

        this.Amount = amount;
        this.Discount = discount;
    }
}
```

Rules enforced by the `ReaderWriterSynchronized` aspect

The `ReaderWriterSynchronizedAttribute` aspect emits build-time errors in the following situations:

- The class contains a public or internal field.
- The class contains a public method is missing a `ReaderAttribute` or `WriterAttribute` custom attribute, or another attribute derived from `AccessLevelAttribute`. Note that property getters and setters and event accessors do not need to be annotated.

The reader/writer synchronized object will throw a `ThreadAccessException` whenever some code tries to access a field from a thread that does not own the correct lock, i.e. when a reader method tries to write a field, or when a non-annotated method (e.g. a delegate call) tries to read or write a field.

Raising synchronous events

In some situations, a method with write access needs to allow other threads to read the object before another write is performed on the object. The implementation of `INotifyCollectionChanged` gives a typical example of this situation. The `INotifyCollectionChanged` event defined by this interface is typically raised from a write method but is consumed from the user interface thread. The object cannot have changed between the moment the event is raised and it is processed by the UI thread, because the event arguments contain data that relates to the current state of the object. Using only `WriterAttribute` and `ReaderAttribute` would either result in deadlocks or in inconsistencies, respectively.

The solution to this problem is to use the `YielderAttribute` custom attribute, which allows read access from other threads but prevents any other thread from acquiring a writer lock.

Example

In the following example, `OrderCollection` is a collection of `Order` objects. In this example, the `Add()` and `Remove()` methods are marked with the `WriterAttribute` attributes. Listeners can be notified about these changes by subscribing to the **CollectionChanged** event which is exposed through the implementation of `INotifyCollectionChanged`

Since listeners can be on other threads (e.g. a UI thread), this event is invoked by the `Add()` and `Remove()` methods via a method called `OnCollectionChanged()` which has been marked with the `YielderAttribute` attribute. This lock ensures that the listener (which may be in another thread space) can read the current state of the collection without the collection being modified by another invocation of the `Add()` or `Remove()` operations from another thread.

```
[ReaderWriterSynchronized]
class OrderCollection : ICollection, INotifyCollectionChanged
{
    ArrayList list = new ArrayList();

    // Details skipped.

    [Reader]
    public int Count
    {
        get
        {
            return list.Count;
        }
    }

    [Writer]
    public void Add(Order o)
    {
        list.Add(o);
        NotifyCollectionChangedEventArgs changedArgs = new NotifyCollectionChangedEventArgs(NotifyCollectionChangedAction.Add, o);
        OnCollectionChanged(changedArgs);
    }

    [Writer]
    public void Remove(int index)
    {
        NotifyCollectionChangedEventArgs changedArgs = new NotifyCollectionChangedEventArgs(NotifyCollectionChangedAction.Remove, list[index]);
        list.RemoveAt(index);
        OnCollectionChanged(changedArgs);
    }

    [Yielder]
    private void OnCollectionChanged(NotifyCollectionChangedEventArgs changedArgs)
    {
        CollectionChanged(this, changedArgs);
    }

    [Reader]
    public Order Get(int index)
    {
        return (Order)list[index];
    }

    public event NotifyCollectionChangedEventHandler CollectionChanged;
}
}
```

Executing long-running write methods

Since write methods require exclusive access to the object, they should complete as quickly as possible. However, this is not always possible. Some long-running write methods really do a lot of write operations (or rely on slow external services) which make them inappropriate for the reader-writer-synchronized model. However, many write methods are

actually composed of a lot of read operations but just a few write operations at the end. In this case, it is possible to use a combination of the `UpgradeableReaderAttribute` and `WriterAttribute` attributes. The `UpgradeableReaderAttribute` attribute ensures that no other thread than the current one will be able to acquire a writer lock on the object, so it gives the guarantee that the object is not going to be modified during the method's execution. A method that holds an upgradeable reader lock can then invoke a method with the `WriterAttribute` attributes custom attribute. Note that it is important that the writer methods leave the object in a consistent state before exiting, because other threads will be allowed to read the object.

The following example builds on that in the section where the `Order` class contains a collection of `Line` objects which make up the order. In the example below, a new method called `Recalculate()` has been added to `Order` which iterates through each `Line` in the collection, tallies up the amount from each, and then stores the total in `Amount`.

Since the `Recalculate` method performs a series of reads followed by a write operation (to store the total in `Amount`), it is marked with the `UpgradeableReaderAttribute` attribute which ensures that all of the orders that it reads remain locked so that it calculates and writes out the correct total. In addition to this, the set accessor of the `Order`'s `Amount` property has been marked with `WriterAttribute`:

```
[ReaderWriterSynchronized]
class Order
{
    // Other details skipped for brevity.publicdecimal Amount
    {
        // The [Reader] attribute optional here is optional because the method is a public getter.get;

        // The [Writer] attribute is required because, although the method is a setter, this setter is private, // therefore
        [Writer] privateset;
    }

    [UpgradeableReader]
    publicvoid Recalculate()
    {
        decimal total = 0;

        for (int i = 0; i < lines.Count; ++i)
        {
            total += lines[i].Amount;
        }

        this.Amount = total;
    }
}
```

Working with object trees

Because the Reader/Writer Synchronized model is an implementation of the `Aggregatable` pattern, all of the same behaviors of the `AggregatableAttribute` are available. For more information regarding object trees, read [Parent/Child, Visitor and Disposable on page 239](#).

NOTE

Once you have established your parent-child relationships you will need to apply compatible threading models to the child classes. You will want to refer to the [Compatibility of Threading Models on page 332](#) article to determine which threading model will work for the children of the Read/Writer Synchronized object.

When a `ReaderWriterSynchronized` object becomes the child of a `Synchronized` object, it effectively becomes fully `Synchronized` itself. From that point on, even its Reader methods will require the full lock and will act as though they were Writer methods. When it stops being a child of a `Synchronized` object, you will again become able to run multiple Reader methods at the same time.

58.6. Actor Threading Model

Given the complexity of trying to coordinate accesses to an object from several threads, sometimes it makes more sense to avoid multi threading altogether. The Actor model avoids the need for thread safety on class instances by routing method calls from each instance to a single message queue which is processed, in order, by a single thread.

Since the processing for each instance takes place in a single thread, multithreading is avoided altogether and the object is guaranteed to be free of data races. Calls are processed asynchronously in the order in which they were added to the message queue. Because all calls to an actor are asynchronous, it is recommended that the `async/await` feature of C# 5.0 be used.

Additionally to providing a race-free programming model, the Actor pattern has the benefit of transparently distributing the computing load to all available CPUs without additional logic. Note that PostSharp's implementation does not assign a new thread to each actor instance but uses a thread pool instead, so it is possible to have a very large number of actors with relatively low overhead.

This topic contains the following sections:

- [Applying the Actor pattern on page 325](#)
- [Rules enforced by the Actor aspect on page 327](#)
- [Working with a complex state on page 327](#)
- [Dealing with constraints of the Actor model on page 327](#)

Applying the Actor pattern

To apply the Actor threading model:

1. Add the `PostSharp.Patterns.Threading` package to your project.
2. Add `using PostSharp.Patterns.Threading` namespace to your file.
3. Add the `ActorAttribute` to the class.
4. Annotate your object model for parent/child relationships as described in [Annotating an Object Model for Parent/Child Relationships \(Aggregatable\) on page 241](#).
5. It is recommended, but not required, that you change all methods `async` methods, and modify the code that calls them.

Example

Consider the following example of an `AverageCalculator` class. The code is not thread-safe because incrementing the `count` has four operations (read and write) that must all be performed atomically.

```
class AverageCalculator
{
    float sum;
    int count;

    public void AddSample(float n)
    {
        this.count++;
        this.sum += n;
    }

    public float GetAverage()
    {
        return this.sum / this.count;
    }
}
```

We could use the Synchronized or Reader-Writer Synchronized threading model to make sure that the calling thread will wait if the object is currently being accessed by another thread. Another solution in this situation is to avoid concurrency altogether using the Actor pattern and asynchronous methods.

In the reworked example below, the `AverageCalculator` class has had the `ActorAttribute` added and the `GetAverage` method has been changed into asynchronous with `ReentrantAttribute` attribute. The `AddSample` method was also changed to an async method returning `Task` and `ReentrantAttribute` attribute was applied.

Note that we could keep the methods non-async, but it is a good practice to make the public API of all actors async methods.

```
[Actor]
class AverageCalculator
{
    float sum;
    int count;

    [Reentrant]
    public async Task AddSample(float n)
    {
        this.count++;
        this.sum += n;
    }

    [Reentrant]
    public async Task<float> GetAverage()
    {
        return this.sum / this.count;
    }
}
```

You can now use the same `AverageCalculator` from two concurrent threads.

```
class Program
{
    static void Main(string[] args)
    {
        MainAsync().GetAwaiter().GetResult();
    }

    static async Task MainAsync()
    {
        AverageCalculator averageCalculator = new AverageCalculator();

        SampleObserver observer = new SampleObserver(averageCalculator);
        DataSources.Source1.Subscribe(observer);
        DataSources.Source2.Subscribe(observer);

        Console.ReadKey();

        float average = await averageCalculator.GetAverage();

        Console.WriteLine("Average: {0}", average);
    }
}

class SampleObserver : IObservable<float>
{
    AverageCalculator calculator;

    public void OnNext(float value)
    {
        // Each of the data sources can call us from a different thread and concurrently. // But we don't have to care since
    }

    // Details skipped.
}
```

Behind the scenes, each invocation of `AverageCalculator.AddSample` is added to the message queue by the `ActorAttribute`, which then processes each call sequentially in the order it was added to the queue. This gives us the guarantee that an instance of the `AverageCalculator` class is never being accessed concurrently by two threads, and eliminates the need to make take multithreading into account.

Rules enforced by the Actor aspect

At build time, the Actor aspect emits an error in the following situations: if your class has public or internal instance fields.

- The class has async methods that are not annotated with the `ReentrantAttribute` attribute (non-reentrant async methods are not yet supported in actors).
- The class has public or internal instance fields.

At run-time, an actor will throw a `ThreadMismatchException` if some code attempts to access a field from a thread that does not currently have access to the object. This typically happens when you schedule a background task or register to an event handler, and you do not mark this method with the `EntryPointAttribute` custom attribute.

Working with a complex state

`PostSharp` generates code that prevents the fields of an actor class to be accessed from an invalid context. For instance, trying to read an actor field from a background task would result in a `ThreadAccessException`. However, very often, the state is more complex than fields of simple types like `int` or `string`. The state can be composed of several objects and collections.

To prevent state corruption, it is important that `PostSharp` generates code that enforces the Actor model at run time even for child objects of the actor.

To add complex state to actor classes:

1. Declare the Parent-Child relationship on the property using the `ChildAttribute` custom attribute.
2. Add the `PrivateThreadAwareAttribute` attribute to the child class.

For more information regarding parent-child relationships in threading models, see also [Parent/Child, Visitor and Disposable on page 239](#).

Example

```
[Actor]
class AverageCalculator
{
    float sum;
    int count;

    [Child]
    private CounterInfo counterInfo;

    // Other details skipped for brevity
}

[PrivateThreadAware]
publicclass CounterInfo
{
    publicstring Name { get; set; }
}
```

Dealing with constraints of the Actor model

Per definition of the Actor model, all methods are executed asynchronously. Methods that have no return value (void methods) can be executed asynchronously without syntactic changes. However, methods that do have a return value need to be made asynchronous using the `async` keyword.

In some situations, the application of the `async` keyword and the corresponding dispatching of the method may be unnecessary. For instance, a method that returns immutable information is always thread-safe and does not need to be dispatched. For more information on excluding methods from dispatching, see [Opting In and Out From Thread Safety on page 331](#).

58.7. Thread-Unsafe Threading Model

When you are dealing with multithreaded code you will run into situations where some objects are not safe for concurrent use by several threads. Although these objects should theoretically not be accessed concurrently, it is very hard to prove that it never happens. And when it does happen, thread-unsafe data structures get corrupted, and symptoms may appear much later. These issues are typically very difficult to debug. So instead of relying on hope, it would be nice if the object threw an exception whenever it is accessed simultaneously by several threads. This is why we have the thread-unsafe threading model.

This topic contains the following sections:

- [Applying the Thread-Unsafe model to a class on page 328](#)
- [Rules enforced by the Thread-Unsafe aspect on page 328](#)

Applying the Thread-Unsafe model to a class

To apply the Thread-Unsafe threading model to a class:

1. Add the `PostSharp.Patterns.Threading` package to your project.
2. Add `using PostSharp.Patterns.Threading` namespace to your file.
3. Add the `ThreadUnsafeAttribute` to the class.
4. Annotate your object model for parent/child relationships as described in [Annotating an Object Model for Parent/Child Relationships \(Aggregatable\) on page 241](#).

Rules enforced by the Thread-Unsafe aspect

The `ThreadUnsafeAttribute` aspect emits build-time errors in the following situations:

- The class contains a public or internal field.

Internally, the Thread-Unsafe model is implemented by a lock. The lock is automatically acquired by public and internal methods, just like the `Synchronized` model.

A thread-unsafe object will throw the following exceptions:

- A `ThreadAccessException` whenever some code tries to access a field from a thread that does not own the correct lock, i.e. the call stack does not contain a public or internal method of this method (e.g. a private delegate call).
- A `ConcurrentAccessException` when two public or internal methods execute at the same time on the same object (i.e. whenever the lock cannot be acquired without waiting).

58.8. Making a Whole Project or Solution Thread Safe

When you want to make a large application thread-safe with PostSharp threading models, it can become difficult to remember to assign a threading model to every single class. In this situation, you can add the thread-safety policy to your project or solution.

The thread-safety policy emits warnings in two situations:

- classes that are not assigned to a threading model,
- static fields that are not read-only or not of a thread-safe type.

IMPORTANT NOTE

The thread-safety policy does not make your application thread-safe by itself. What the thread-safety policy does is to remind you to use threading models in your code. It is the use of threading models that makes your application thread-safe.

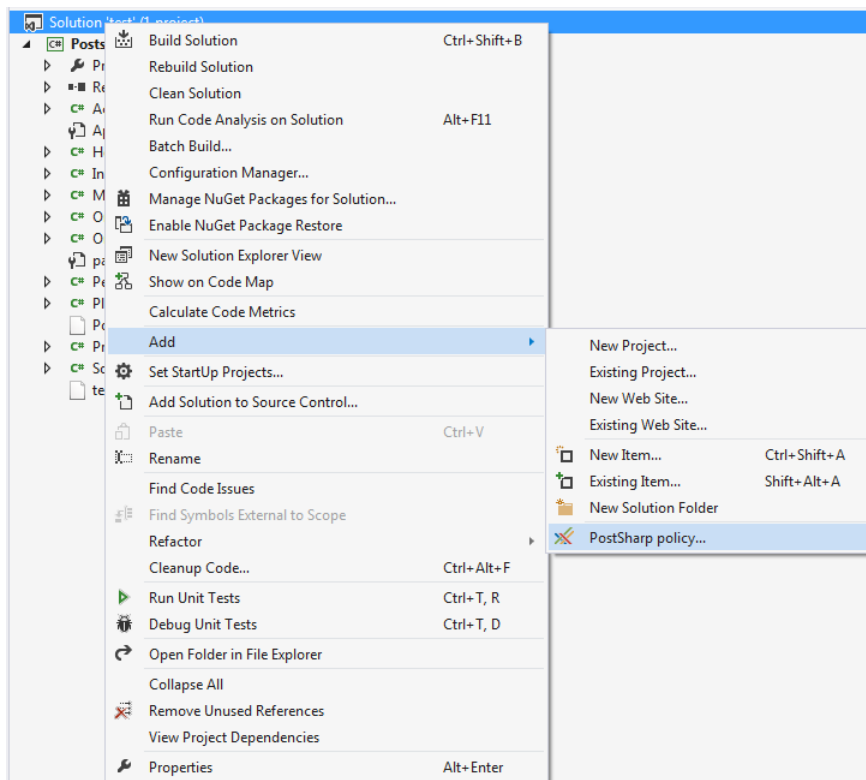
This topic contains the following sections:

- [Adding the thread-safety policy using PostSharp Tools for Visual Studio. on page 329](#)
- [Adding the thread-safety policy to a project manually. on page 330](#)
- [Adding the thread-safety policy to a whole solution manually. on page 330](#)

Adding the thread-safety policy using PostSharp Tools for Visual Studio.

To apply the thread-safety policy to your application with PostSharp Tools for Visual Studio:

1. Right click on your solution or your project in **Solution Explorer**, select **Add** followed by **PostSharp Policy...**



2. In the **Add PostSharp policy** wizard, expand **Threading** and select **Thread Safety**.
3. If you clicked on the solution, select the projects that you would like to add the policy to.
4. Review the configuration that you have selected and click **Next**.
5. Close the wizard when the process had completed by clicking **Finish**.

If you added the policy to the whole solution, the result of running this wizard is that a *pssln* file has been added to your project. The *pssln* file contains an entry that enables deadlock detection across all projects in your solution.

```
<Projectxmlns="http://schemas.postsharp.org/1.0/configuration"xmlns:t="clr-namespace:PostSharp.Patterns.Threading;assembly
```

Adding the thread-safety policy to a project manually.

To add the thread-safety policy to a project manually:

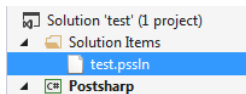
1. Add the *PostSharp.Patterns.Threading* NuGet package to the project.
2. Add the *ThreadSafetyPolicy* any C# file. We recommend you add it to a new file named *GlobalAspects.cs*.

```
using PostSharp.Patterns.Threading;  
[assembly: ThreadSafetyPolicy]
```

Adding the thread-safety policy to a whole solution manually.

To manually add the thread-safety policy to a whole solution:

1. Open the solution's *pssln* file. This can be found under the Solution Items folder in Visual Studio's Solution Explorer.



If the *pssln* file doesn't exist manually add the file at the solution level. Name the file with the same name as your solution and the *pssln* file extension.

2. If you had to create the *pssln* file and add it to your solution add the following XML to it. If the *pssln* file already existed in your project proceed to the next step.

```
<?xmlversion="1.0"encoding="utf-8"?><Projectxmlns="http://schemas.postsharp.org/1.0/configuration"xmlns:t="clr-n
```

3. Add a multicast attribute to the Project element that will add *ThreadSafetyPolicy* to all the projects in the solution.

```
<?xmlversion="1.0"encoding="utf-8"?><Projectxmlns="http://schemas.postsharp.org/1.0/configuration"xmlns:t="clr-n
```

4. Add the *PostSharp.Patterns.Threading* NuGet package to all projects in the solution.

Once you save the *pssln* file you will have added thread-safety policy to all projects in your solution.

58.9. Opting In and Out From Thread Safety

By default, PostSharp enforces thread safety for all instance fields and all public and internal methods of any class to which you applied a threading model.

However, there are times when you want to opt-out from this mechanism for a specific field or method. A typical reason is that access to the field is synchronized manually using a different mechanism.

This section shows how to override the default thread safety implemented by PostSharp.

This topic contains the following sections:

- [Opting out from thread-safety verification for a method on page 331](#)
- [Opting out from thread-safety for a field on page 331](#)
- [Opting in for thread safety for callback methods on page 332](#)

Opting out from thread-safety verification for a method

To disable enforcement of the class-level threading model for a specific method, add the `ExplicitlySynchronizedAttribute` attribute to that method.

In the following example, this custom attribute allows us to implement the `ToString` in a class that respects the Actor model. Without the custom attribute, this would not have been possible because non-void public methods must have the `async` keyword.

```
[Actor]
class Player
{
    private readonly string name;

    [ExplicitlySynchronized]
    public override string ToString()
    {
        return this.name;
    }
}
```

When used on a method, the `ExplicitlySynchronizedAttribute` attribute has several effects:

1. Lock-based aspects such as `SynchronizedAttribute` or `ReaderWriterSynchronizedAttribute` will not attempt to acquire a lock before executing this method.
2. Accesses to fields are not verified during the whole execution of the method (for the current thread).
3. All build-time verifications are disabled for this method.

CAUTION NOTE

By using the `ExplicitlySynchronizedAttribute` custom attribute, you are significantly increasing the risk that multithreading defects in user code go undetected by PostSharp. Code using `ExplicitlySynchronizedAttribute` should be more carefully covered by reviews and tests.

Opting out from thread-safety for a field

To disable enforcement of the class-level threading model for a specific field, add the `ExplicitlySynchronizedAttribute` attribute to the field:

```
[Actor]
class MyActor
{
```

```
[ExplicitlySynchronized]
int counter;

public void FooBar()
{
    // This line would throw an exception without [ExplicitlySynchronized].
    Task.Factory.StartNew(() => Interlocked.Increment( ref this.counter ));
}
}
```

When used on a field, the `ExplicitlySynchronizedAttribute` attribute has several effects:

1. Accesses to the field are never verified
2. All build-time verifications are disabled for this method.

Opting in for thread safety for callback methods

By default, thread safety is ensured when a thread first invokes a public or internal method of an object. The underlying motivation is that public and internal methods are the primary way how a thread can enter an object. Another way is to enter an object through a delegate call to a private method. By default, PostSharp does not ensure thread safety for private methods. If you register a callback method, you need to add the `EntryPointAttribute` custom attribute on this method.

In the following code snippet, the `OnCreated` method is invoked from a background thread by the `FileSystemWatcher` class. The `InputQueueWatcher` is thread-safe thanks to the `SynchronizedAttribute` aspect.

```
[Synchronized]
class InputQueueWatcher
{
    FileSystemWatcher watcher;

    [Child]
    AdvisableCollection<string> files = new AdvisableCollection<string>();

    public InputQueueWatcher(string path)
    {
        this.watcher = new FileSystemWatcher();
        this.watcher.Path = path;
        this.watcher.NotifyFilter = NotifyFilters.LastWrite | NotifyFilters.FileName | NotifyFilters.DirectoryName;
        this.watcher.Filter = "*.xml";
        this.watcher.Created += new FileSystemEventHandler(OnCreated);
    }

    [EntryPoint]
    private void OnCreated(object source, FileSystemEventArgs e)
    {
        // Without [EntryPoint], the following line would throw ThreadAccessException.this.files.Add(e.FullPath);
    }

    public ICollection Files { get { return this.files; } }
}
}
```

58.10. Compatibility of Threading Models

This table shows which threading models can you use as children based on the model of the parent.

Compatibility Matrix

Parent↓ Child→	Actor	Freezable	Immutable	Private	Reader-Writer Synchronized	Synchronized	Thread Affine	Thread Unsafe
Actor	No	Yes	Yes	Yes	No	No	No	Yes
Freezable	No	Yes	Yes	Yes	No	No	No	No
Immutable	No	Yes	Yes	Yes	No	No	No	No
Reader-Writer Synchronized	Yes (Own)	Yes	Yes	Yes	Yes (Shared)	No	No	No
Synchronized	Yes (Own)	Yes	Yes	Yes	Yes (Shared)	Yes (Shared)	No	No
Thread Affine	Yes (Own)	Yes	Yes	Yes	No	No	Yes	Yes (Shared)
Thread Unsafe	Yes (Own)	Yes	Yes	Yes	No	No	Yes	Yes (Shared)
Private								

When you assign an object with a threading model to another object with a threading model as a child, the assignment statement will block until the thread has full access to both objects. For a Synchronized object, that means it has the lock. For a ReaderWriterSynchronized object, that means it has the write lock. The same is true for an assignment which removes the parent-child link from two objects.

58.11. Enabling and Disabling Runtime Verification

When you apply a threading model to a class, PostSharp adds two kinds of behaviors: behaviors that are necessary to implement the semantic of the threading model (for instance acquiring a lock or dispatching a method call) and behaviors that validate that the source code is valid against the chosen threading model (for instance that no field is written if the current method does not have write access). The second set of behaviors are called *runtime verifications*. By default, runtime verifications are enabled in the Debug build and disabled in the Release build.

This section explains how to enable or disable runtime verification.

This topic contains the following sections:

- [Understanding the default configuration on page 333](#)
- [Enabling or disabling runtime verification for a whole project on page 333](#)
- [Enabling and disabling runtime verification for a specific class on page 335](#)

Understanding the default configuration

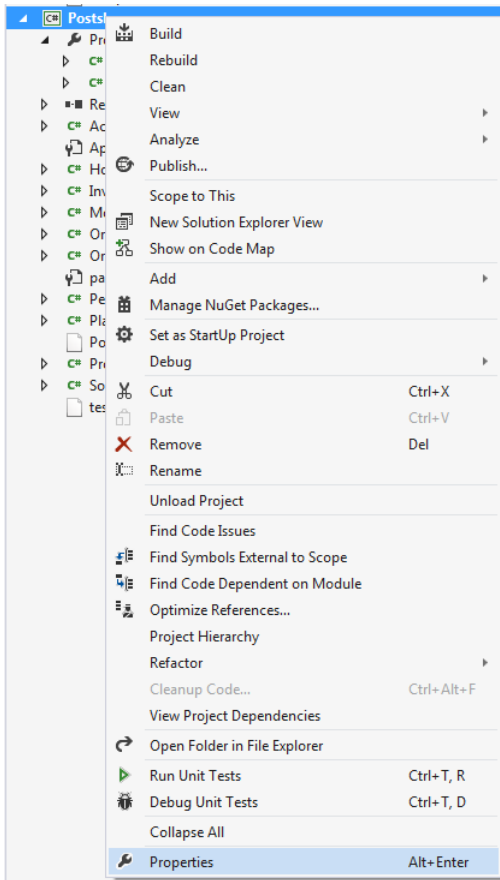
By default, runtime verification is disabled if the **Optimize Code** compiler flag is enabled. Therefore, runtime verification is enabled by default in the Debug build and disabled in the Release build.

Enabling or disabling runtime verification for a whole project

Perform the following steps to enable runtime verification by using the Project Settings dialog

Enabling Runtime Verification in Project Properties

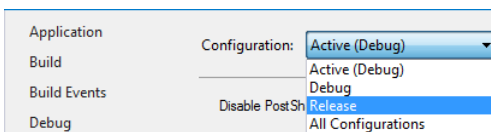
1. Open the project's Properties window.



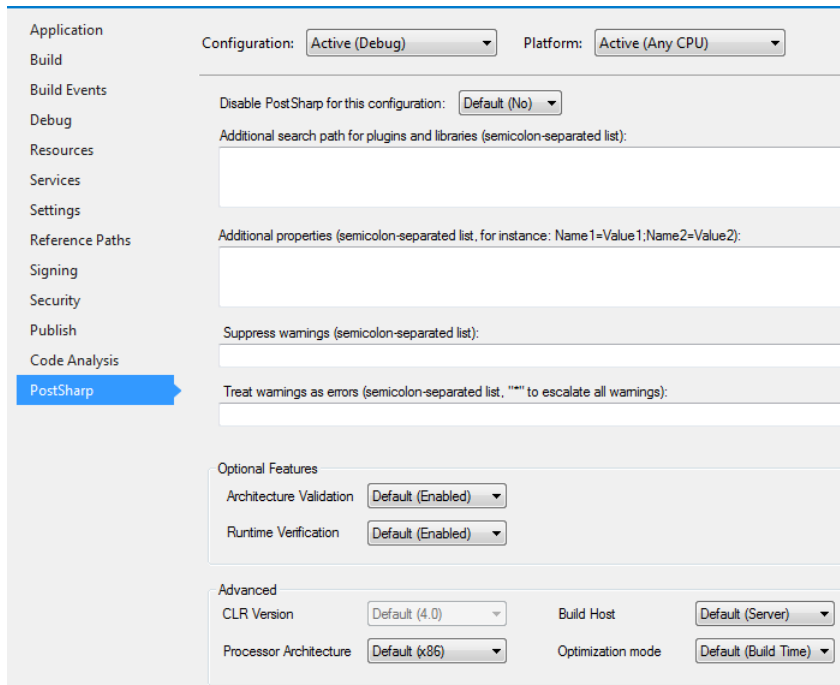
2. Select the build configuration that you want to enable runtime verification on.

NOTE

By default, projects have two different build configurations: Debug and Release. Each build configuration can, and by default does, have a different behavior for runtime verification.

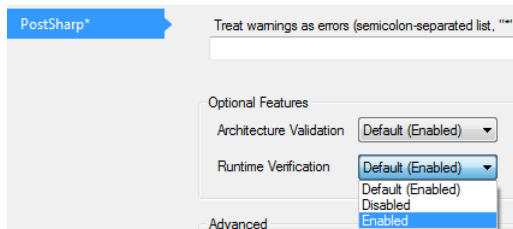


- Open the PostSharp tab.



- In the Optional Features section there is a Runtime Verification dropdown. The dropdown has three options in it; Default, Disabled, and Enabled.

The Default option will include either (Enabled) or (Disabled) after it. This value will change based on the Optimize Code compiler flag setting. If the compiler flag is disabled the drop down option will read Default (Enabled) and if the Optimize Code flag is enabled the dropdown option will read Default (Disabled).



Enabling and disabling runtime verification for a specific class

You can override the project-level configuration of the runtime verification setting by setting the `RuntimeVerificationEnabled` property of the threading model custom attribute. This property is defined by the `ThreadAwareAttribute` class, from which all threading model attributes derive.

```
[ThreadAffine(RuntimeVerificationEnabled = true)]
```

If the property is not manually set it derives its value from the setting on the project properties page. If you want to override the default value all you need to do is set the value of the `RuntimeVerificationEnabled` to `true` or `false`.

58.12. Run-Time Performance of Threading Model

When runtime verification of threading models is disabled (see [Enabling and Disabling Runtime Verification on page 333](#)), there is almost no runtime overhead of using PostSharp threading models compared to implementing thread synchronization manually, at least after the object model has been instantiated.

However, PostSharp threading models come at a high memory cost, and there may be a significant performance overhead when instantiating large object graphs unless care is taken.

This topic contains the following sections:

- [Memory Consumption on page 336](#)
- [Instantiation of Large Object Trees on page 336](#)
- [Assigning the Concurrency Controller Manually on page 337](#)

Memory Consumption

As most complex aspects implemented with PostSharp, threading models have a high memory cost. Several object instances are needed for each instance of a thread-safe class. If memory consumption is a concern, you should not use threading models on classes that have a very high number of instances.

Instantiation of Large Object Trees

Threading models can have a significant impact on the cost of creating large object trees. In some situations, the cost of instantiating the tree can become $O(n^2)$ instead of $O(n)$. The performance issue affects only the following threading models:

- Synchronized,
- Reader-Writer Synchronized and
- Thread-Unsafe.

The performance issue stems from the fact that each root node in a tree needs its own instance of its concurrency controller.

Consider the scenario when you build a tree using a depth-first approach. That means that you would first instantiate the leaves of the tree, then the parent of the leaves, then the parent of the parent, and so on until you reach the root. Depth-first tree instantiation is a common strategy when you instantiate immutable trees. Note however that the immutable model is not affected by this issue.

When you start instantiating the leaves, and until the leaf is assigned to a parent, every leaf is the root of its own tree. This means that an instance of the concurrency controller may be created if needed. When you instantiate the first-level parents, a new concurrency controller is created for each first-level parent. When the leaf is assigned to its parent, the concurrency controllers of the leaves will be replaced by the concurrency controller of the immediate parent.

The same phenomenon occurs at each level of the tree. Whenever you assign a sub-tree to a parent, the concurrency controller of whole subtree is reassigned. Replacing the concurrency controller of a subtree is an $O(n)$ operation, and it should be achieved for each of the n nodes, which means that totally the concurrency controllers will be reassigned $O(n^2)$ times.

During the operation of instantiating the tree, $O(n)$ controllers may be instantiated. However, at the end of the operation, a single controller will remain in memory.

To prevent PostSharp from allocating $O(n)$ controllers and performing $O(n^2)$ reassignments, you need to manually assign newly-created objects to a concurrency controller.

Assigning the Concurrency Controller Manually

To avoid excessive creation and assignment of concurrency controllers, you can use the `WithConcurrencyController(IConcurrencyController)` method to set the default controller for newly-created objects.

The following code snippet illustrates the use of `WithConcurrencyController(IConcurrencyController)`. Thanks to this method, a single concurrency controller instance is created, and each node is assigned only once to this concurrency controller, amounting to 3 assignments for 3 nodes. Without the use of this method, 3 instances of the concurrency controller would have been created, and totally 5 assignments would be done.

```
var useDeadlockDetection = DeadlockDetectionPolicy.IsEnabled(typeof(SynchronizedObject).Assembly);

using ( ThreadingServices.WithConcurrencyController( ConcurrencyControllerFactory.CreateSynchronizedController(useDeadlockI
{
    var child1 = new SynchronizedObject();
    var child2 = new SynchronizedObject();

    var parent = new SynchronizedObject();
    parent.Children.Add( child1 );
    parent.Children.Add( child2 );
}
```


CHAPTER 59

Dispatching a Method to Background

Long running processes will block the further execution of code while the system waits for them to complete. When you are building applications it's common to push long running processes to the background so that other processes can continue without waiting. Two common ways of doing this are with asynchronous processing and the `BackgroundWorker`. Both require a lot of boiler plate code to push execution to another thread.

PostSharp provides you with the ability to push execution of a method to a background thread without having to worry about all of the boiler plate code.

To add execute a method in the background:

1. Add the `PostSharp.Patterns.Threading` package to your project.
2. Add `using PostSharp.Patterns.Threading` namespace to your file.
3. Add the `BackgroundAttribute` to the method that you want to push to the background for execution. The method must have `void`, `Task` or `TaskTResult` return type.

Those simple steps are all that is required for you to declare that a method should be executed in a background thread.

If the method is `void`, it will be executed in the background, and the caller code will not wait until the background method completes its execution. If the method returns a `Task`, the method will be fully executed in the background (even the first segment of the method, before the first `await` keyword).

Methods annotated with this attribute are run in the managed thread pool, unless you use the attribute's `IsLongRunning` property, in which case it is run as a new separate background thread.

CHAPTER 60

Dispatching a Method to the UI Thread

When you are building desktop or mobile user interfaces, parts of your code may execute on background threads. However, the user interface itself can be accessed only from the UI thread. Therefore, it is often necessary to dispatch execution of code from a background thread to the foreground thread.

Traditionally, thread dispatching has been implemented using the `Invoke(Delegate)` method in WinForms or the `Dispatcher` class in XAML. However, this results in a large amount of boilerplate, making the code unreadable.

The `DispatchedAttribute` aspect addresses the issue of thread dispatching by forcing a method to execute on the thread that created the object (typically the foreground thread).

This topic contains the following sections:

- [Forcing a method to execute on the foreground thread on page 341](#)
- [Executing a method asynchronously on page 342](#)
- [Executing async methods in the foreground thread on page 342](#)

Forcing a method to execute on the foreground thread

To force a method to execute on the foreground thread:

1. Add the `PostSharp.Patterns.Threading` package to your project.
2. Add `using PostSharp.Patterns.Threading` namespace to your file.
3. Add the `DispatchedAttribute` to the method that you want to be executed on the foreground thread, regardless of where it's called from.

Example

The following example shows how to use both `BackgroundAttribute` and `DispatchedAttribute` to specify on which threads different methods of the class are executed.

```
[Background]
privatevoid SaveButton_Click( object sender, RoutedEventArgs e )
{
    using ( var file = File.CreateText( this.path ) )
    {
        this.model.SaveTo( file );
    }

    this.SaveCompleted();
}

[Dispatched]
privatevoid SaveCompleted()
{
    this.StatusLabel.Text = "Finished Saving";
}
```

Executing a method asynchronously

By default, the `DispatchedAttribute` forces the target method to execute synchronously on the foreground thread, which means that the background thread will wait until the method execution has completed. This waiting causes some performance overhead. Additionally, synchronous execution is not always useful. If the method has no return value and no side effect of interest for the calling thread, the method could be safely executed asynchronously, which means the calling thread would not need to wait for the method execution to complete on the foreground thread, so that the calling thread would continue its execution immediately after having enqueued the call to the foreground thread.

You can enable asynchronous execution of a dispatched method by passing the true value to the parameter of the `DispatchedAttribute(Boolean)` constructor, for instance:

```
[Dispatched(true)]
privatevoid SaveCompleted()
{
    this.StatusLabel.Text = "Finished Saving";
}
```

Executing async methods in the foreground thread

When you use the `DispatchedAttribute` aspect on asynchronous methods (`async` keyword in C#), the method is guaranteed to execute on the foreground thread even when it is invoked from a background thread.

CHAPTER 61

Detecting Deadlocks at Runtime

A common problem that is found in multithreaded code is that multiple threads enter a situation where they are waiting for each other to finish. This is a deadlock situation and neither thread will complete executing in this situation. Because the threads are waiting on each other, neither is capable of providing diagnostic information to aid in debugging the situation. The `DeadlockDetectionPolicy` helps provide this information.

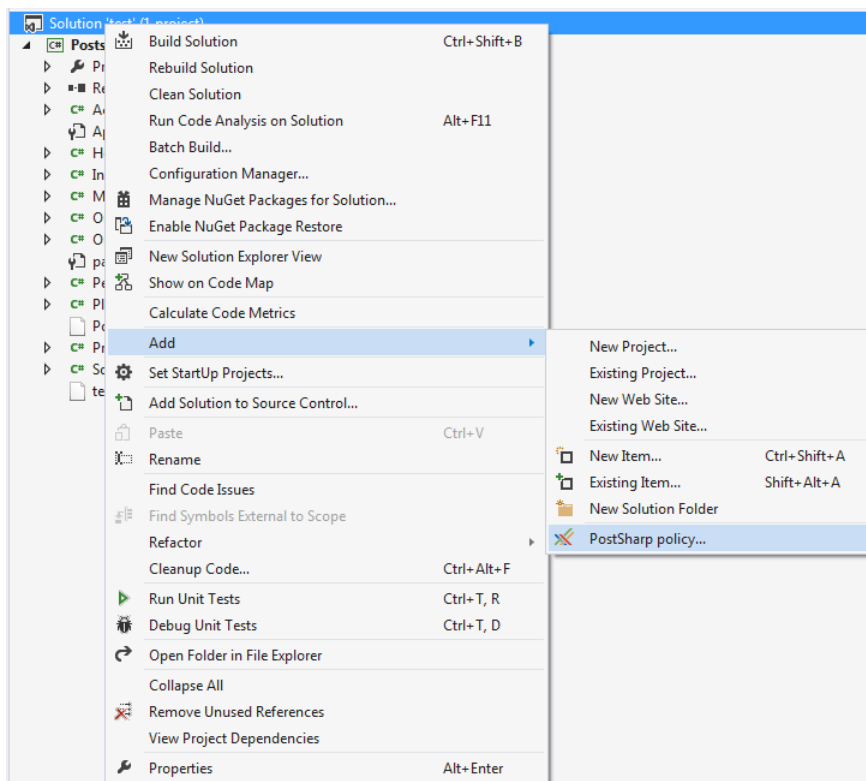
This topic contains the following sections:

- [Adding deadlock detection using PostSharp Tools for Visual Studio on page 343](#)
- [Manually adding deadlock detection to a project on page 347](#)
- [Manually adding deadlock detection to the whole solution on page 347](#)
- [Deadlock detection on page 348](#)

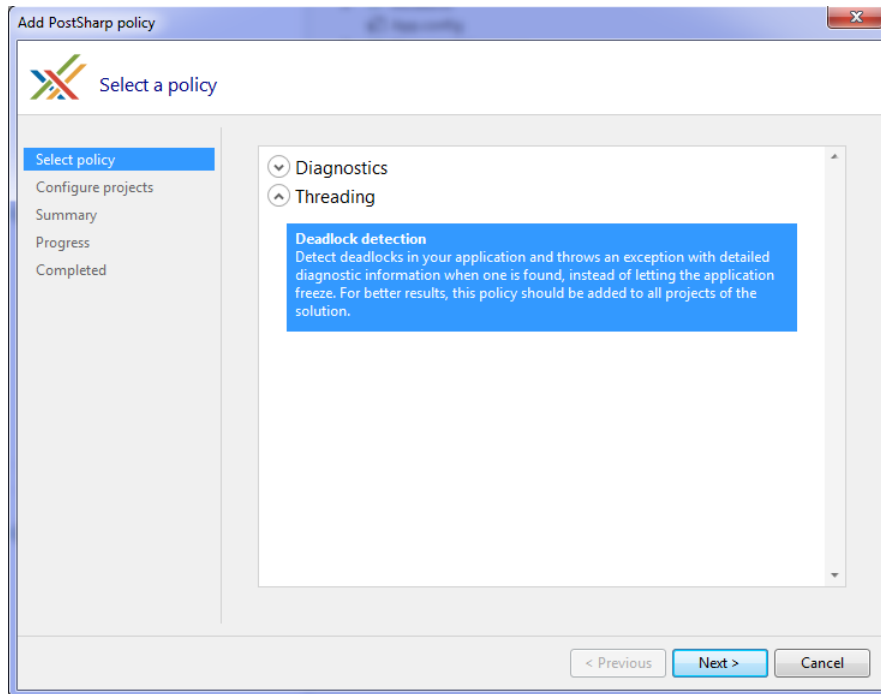
Adding deadlock detection using PostSharp Tools for Visual Studio

To apply the deadlock detection to your application with PostSharp Tools for Visual Studio:

1. Right click on your solution in Solution Explorer, select Add followed by PostSharp Policy...



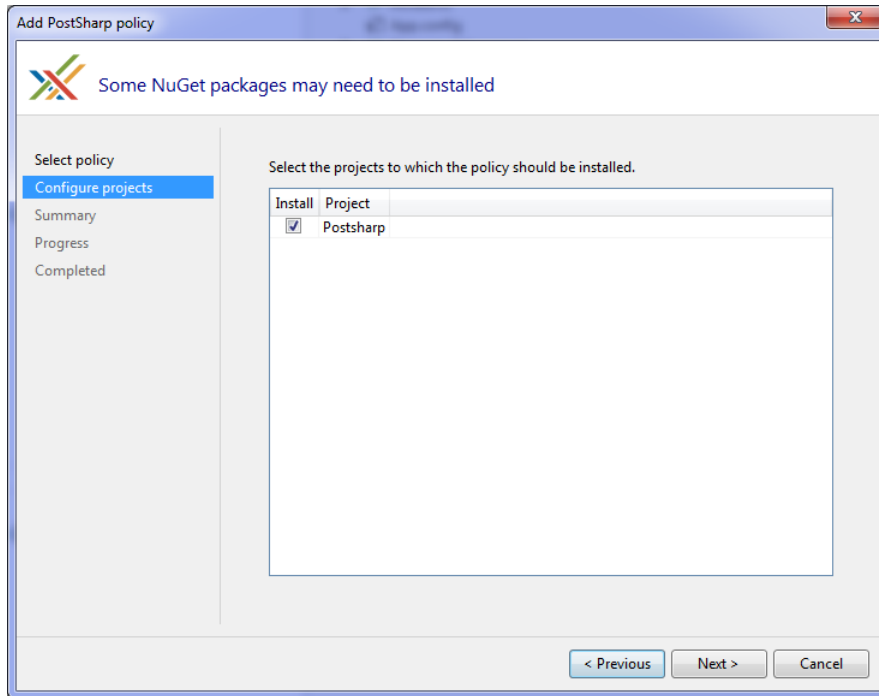
2. In the Add PostSharp policy wizard, expand Threading and select Deadlock detection.



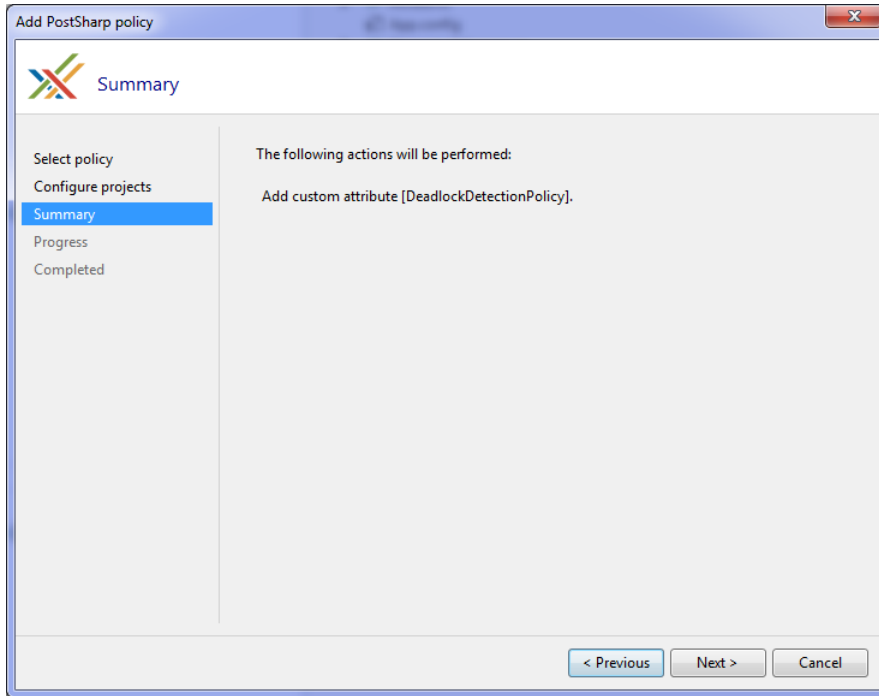
3. Select the projects that you would like to add deadlock detection to.

NOTE

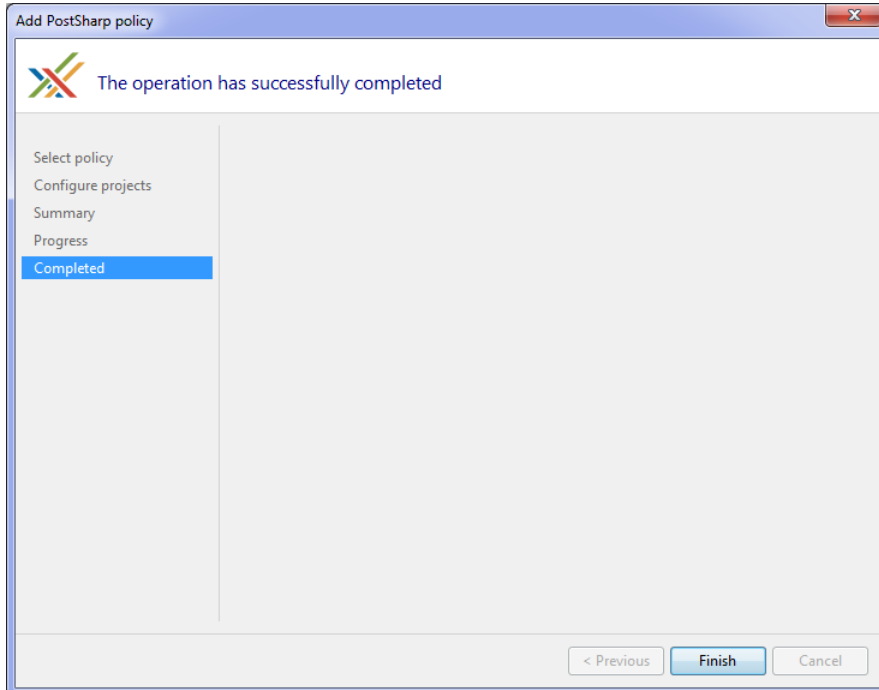
You will need to add this to every project in your application. Excluding projects could cause your application to fail.



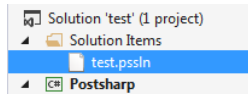
- Review the configuration that you have selected and click Next.



- Close the wizard when the process had completed by clicking Finish.



The result of running this wizard is that a *pssln* file has been added to your project.



The *pssl* file contains an entry that enables deadlock detection across all projects in your solution.

```
<Projectxmlns="http://schemas.postsharp.org/1.0/configuration"xmlns:t="clr-namespace:PostSharp.Patterns.Threading;assembly
```

Manually adding deadlock detection to a project

To manually add deadlock detection to a project:

1. Add the *PostSharp.Patterns.Threading* NuGet package to the project.
2. Add the `DeadlockDetectionPolicy` custom attribute to in any C# file. We recommend you add it to a new file named *GlobalAspects.cs*.

```
[assembly: DeadlockDetectionPolicy]
```

NOTE

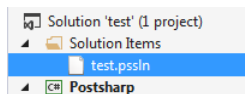
You will need to add this to every project in your application. Excluding projects could cause your application to fail.

Manually adding deadlock detection to the whole solution

Adding deadlock detection at the solution level can also be done manually. This can be done by adding an entry to the *pssl* file in the solution.

To manually add deadlock detection to a solution:

1. Open the solution's *pssl* file. This can be found under the Solution Items folder in Visual Studio's Solution Explorer.



If the *pssl* file doesn't exist manually add the file at the solution level. Name the file with the same name as your solution and the *pssl* file extension.

2. If you had to create the *pssl* file and add it to your solution add the following XML to it. If the *pssl* file already existed in your project proceed to the next step.

```
<?xmlversion="1.0"encoding="utf-8"?><Projectxmlns="http://schemas.postsharp.org/1.0/configuration"xmlns:t="clr-n
```

3. Add a multicast attribute to the Project element that will add `DeadlockDetectionPolicy` to all the projects in the solution.

```
<?xmlversion="1.0"encoding="utf-8"?><Projectxmlns="http://schemas.postsharp.org/1.0/configuration"xmlns:t="clr-n
```

4. Add the *PostSharp.Patterns.Threading* NuGet package to all projects in the solution.

Once you save the *pssl* file you will have added deadlock detection to all projects in your solution.

Deadlock detection

When a deadlock is detected a `DeadLockException` is thrown. The exception will include a detailed report of all the threads and locks involved in the deadlock. Here is an example of that.

```
PostSharp.Patterns.Threading.DeadLockException: Deadlock detected. The following
synchronization elements form a cycle: #0={{Thread 9, Name=""}}; #1={{System.Object:Lock}}; #2={{Thread 10, Name=""}}; #3={{System.Object:Lock}}

-- start of stack trace of thread 10 (Name=""):
   at System.Threading.Monitor.ReliableEnterTimeout(Object obj, Int32 timeout, Boolean& lockTaken)
   at System.Threading.Monitor.TryEnter(Object obj, Int32 millisecondsTimeout, Boolean& lockTaken)
   at PostSharp.Patterns.Threading.DeadlockDetectionPolicy.MonitorEnhancements.<>c__DisplayClass3f.<EnterInternal>b__3b(Nothing _, Int32 timeout)
   at PostSharp.Patterns.Threading.DeadlockDetectionPolicy.Helper.NoTimeoutAcquire[TResult,TContext](Func`1 enterWaiting, Func`3 waitAndCheckIfFinished, Action`1 convertWaitingToAcquired, Action`1 exitWaiting, Func`2 getResult)
   at PostSharp.Patterns.Threading.DeadlockDetectionPolicy.MonitorEnhancements.<>c__DisplayClass3f.<EnterInternal>b__39()
   at PostSharp.Patterns.Threading.DeadlockDetection.DeadlockMonitor.Execute[T](Func`1 callback)
   at PostSharp.Patterns.Threading.DeadlockDetectionPolicy.MonitorEnhancements.EnterInternal(Object obj, Boolean& lockTaken)
   at PostSharp.Patterns.Threading.DeadlockDetectionPolicy.MonitorEnhancements.EnterObject_Boolean(MethodInterceptionArgs args)
   at PostSharp.ImplementationDetails_c23235bd.<>z__Aspects.<System.Threading.Monitor.Enter>b__(Object obj, Boolean& lockTaken) in :line 0
   at PostSharp.MonthEndProcessing.ProcessSalaries() in e:\working\postsharp3.l\writing\test\MonthEndProcessing.cs:line 15
   at PostSharp.Program.<>c__DisplayClass1.<Deadlock>b__0(Object sender, DoWorkEventArgs args) in e:\working\postsharp3.l\writing\test\Program.cs:line 31
   at System.ComponentModel.BackgroundWorker.OnDoWork(DoWorkEventArgs e)
   at System.ComponentModel.BackgroundWorker.WorkerThreadStart(Object argument)
   at System.Runtime.Remoting.Messaging.StackBuilderSink._PrivateProcessMessage(IntPtr md, Object[] args, Object server, Object[]& outArgs)
   at System.Runtime.Remoting.Messaging.StackBuilderSink.AsyncProcessMessage(IMessage msg, IMessageSink replySink)
   at System.Runtime.Remoting.Proxies.AgileAsyncWorkerItem.ThreadPoolCallback(Object o)
   at System.Threading.QueueUserWorkItemCallback.WaitCallback_Context(Object state)
   at System.Threading.ExecutionContext.RunInternal(ExecutionContext executionContext, ContextCallback callback, Object state, Boolean preserveSyncCtx)
   at System.Threading.ExecutionContext.Run(ExecutionContext executionContext, ContextCallback callback, Object state, Boolean preserveSyncCtx)
   at System.Threading.QueueUserWorkItemCallback.System.Threading.IThreadPoolWorkItem.ExecuteWorkItem()
   at System.Threading.ThreadPoolWorkQueue.Dispatch()
   at System.Threading._ThreadPoolWaitCallback.PerformWaitCallback()

-- end of stack trace of thread 10

-- start of stack trace of thread 9 (Name=""):
   at PostSharp.Patterns.Threading.DeadlockDetection.DeadlockMonitor.AnalyzeDeadlockCycle(IEnumerable`1 cycle, StringBuilder messageBuilder, Thread[]& threadsInDeadlock, StackTrace[]& stackTraces)
   at PostSharp.Patterns.Threading.DeadlockDetection.DeadlockMonitor.ProcessDeadlock(IEnumerable`1 cycle)
   at PostSharp.Patterns.Threading.DeadlockDetection.DeadlockMonitor.DetectDeadlocksInternal(Thread startThread)
   at PostSharp.Patterns.Threading.DeadlockDetectionPolicy.Helper.NoTimeoutAcquire[TResult,TContext](Func`1 enterWaiting, Func`3 waitAndCheckIfFinished, Action`1 convertWaitingToAcquired, Action`1 exitWaiting, Func`2 getResult)
   at PostSharp.Patterns.Threading.DeadlockDetectionPolicy.MonitorEnhancements.<>c__DisplayClass3f.<EnterInternal>b__39()
   at PostSharp.Patterns.Threading.DeadlockDetection.DeadlockMonitor.Execute[T](Func`1 callback)
   at PostSharp.Patterns.Threading.DeadlockDetectionPolicy.MonitorEnhancements.EnterInternal(Object obj, Boolean& lockTaken)
   at PostSharp.Patterns.Threading.DeadlockDetectionPolicy.MonitorEnhancements.EnterObject_Boolean(MethodInterceptionArgs args)
   at PostSharp.ImplementationDetails_c23235bd.<>z__Aspects.<System.Threading.Monitor.Enter>b__(Object obj, Boolean& lockTaken) in :line 0
   at PostSharp.MonthEndProcessing.ProcessBonuses() in e:\working\postsharp3.l\writing\test\MonthEndProcessing.cs:line 26
   at PostSharp.Program.Deadlock() in e:\working\postsharp3.l\writing\test\Program.cs:line 34
```

PART 13

Developing Custom Aspects

CHAPTER 62

Developing Simple Aspects

Simple aspects are aspects that are composed of a single transformation. Developing a simple aspect in PostSharp is straightforward: you just have to create a new class, derive it from a primitive aspect class, and override some special methods named *advices*.

If your aspect cannot be implemented as a single transformation, see [Developing Composite Aspects on page 397](#).

In this section:

Article	Description
Injecting Behaviors Before and After Method Execution on page 351	This article shows how to execute code when a method starts, succeeds, or fails, around <code>await</code> operators of async methods, or <code>yield</code> return statements of iterator methods.
Handling Exceptions on page 359	This article describes how to write an aspect that handles exceptions.
Intercepting Methods on page 364	This article explains how to intercept the execution of methods.
Intercepting Properties and Fields on page 368	This article shows how to intercept read and write accesses to fields and properties.
Intercepting Events on page 374	This article describes how to intercept the action of adding a delegate to an event, removing a delegate from an event, and invoking a delegate by raising the event.
Introducing Custom Attributes on page 377	This article shows how to add a custom attribute to any element of code.
Introducing Managed Resources on page 381	This article shows how to add a managed resource to the current assembly.

62.1. Injecting Behaviors Before and After Method Execution

The `OnMethodBoundaryAspect` aspect implements the so-called *decorator* pattern: it allows you to execute logic before and after the execution of a target method.

You may want to use method decorators to perform logging, monitor performance, initialize database transactions or any one of many other infrastructure related tasks. PostSharp provides you with an easy to use framework for all of these tasks in the form of the `OnMethodBoundaryAspect`.

This topic contains the following sections:

- [Executing code before and after method execution on page 352](#)
- [Accessing the current execution context on page 353](#)
- [Returning without executing the method on page 354](#)

- [Handling exceptions on page 355](#)
- [Sharing state between advices on page 355](#)
- [Working with async methods on page 355](#)
- [Working with iterator methods on page 357](#)

Executing code before and after method execution

When you are decorating methods, there are different locations that you may wish to inject functionality to. You may want to perform a task prior to the method executing or just before it finishes execution. There are situations where you may want to inject functionality only when the method has successfully executed or when it has thrown an exception. All of these injection points are structured and available to you in the `OnMethodBoundaryAspect` class as virtual methods (called *advices*) that you can implement if you need them.

The following table shows the advice methods available in the `OnMethodBoundaryAspect` class (see below for more advice methods available on `async` and `iterator` methods).

Advice	Description
<code>OnEntry(MethodExecutionArgs)</code>	When the method execution starts, before any user code.
<code>OnSuccess(MethodExecutionArgs)</code>	When the method execution succeeds (i.e. returns without an exception), after any user code.
<code>OnException(MethodExecutionArgs)</code>	When the method execution fails with an exception, after any user code. It is equivalent to a <code>catch</code> block.
<code>OnExit(MethodExecutionArgs)</code>	When the method execution exits, whether successfully or with an exception. This advice runs after any user code and after the <code>OnSuccess(MethodExecutionArgs)</code> or <code>OnException(MethodExecutionArgs)</code> method of the current aspect. It is equivalent to a <code>finally</code> block.

To create a simple aspect that writes some text whenever a method enters, succeeds, or fails:

1. Add a reference to the *PostSharp* package to your project.
2. Create an aspect class and inherit `OnMethodBoundaryAspect`.
3. Annotate the class with the `[SerializableAttribute]` custom attribute.
4. Override the `OnEntry(MethodExecutionArgs)`, `OnSuccess(MethodExecutionArgs)`, `OnException(MethodExecutionArgs)` and/or `OnExit(MethodExecutionArgs)` as needed, and code the logic that needs to be executed at these points.
5. Add the aspect to one or more methods. Since `OnMethodBoundaryAspect` derives from the `Attribute` class, you can just add the aspect custom attribute to the methods you need. If you need to add the aspect to more methods (for instance all public methods in a namespace), you can learn about more advanced techniques in [Adding Aspects to Code on page 109](#).

Example

The following code snippet shows a simple aspect based on `OnMethodBoundaryAspect` which writes a line to the console during each of the four events. The aspect is applied to the `Program.Main` method.

```
[Serializable]
public class LoggingAspect : OnMethodBoundaryAspect
{
    public override void OnEntry(MethodExecutionArgs args)
    {
        Console.WriteLine("The {0} method has been entered.", args.Method.Name);
    }
}
```



```

public override void OnSuccess(MethodExecutionArgs args)
{
    Console.WriteLine("The {0} method executed successfully.", args.Method.Name);
}

public override void OnExit(MethodExecutionArgs args)
{
    Console.WriteLine("The {0} method has exited.", args.Method.Name);
}

public override void OnException(MethodExecutionArgs args)
{
    Console.WriteLine("An exception was thrown in {0}.", args.Method.Name);
}
}

static class Program
{
    [LoggingAspect]
    static void Main()
    {
        Console.WriteLine("Hello, world.");
    }
}

```

Executing the program prints the following lines to the console:

```

The Main method has been entered.
Hello, world.
The Main method executed successfully.
The Main method has exited.

```

Accessing the current execution context

As illustrated in the example above, you can access information about the method being intercepted from the property `MethodExecutionArgs.Method`, which gives you a reflection object `MethodBase`. This object gives you access to parameters, return type, declaring type, and other characteristics. In case of generic methods or generic types, `Method` gives you the proper generic method instance, so you can use this object to get generic parameters.

The `MethodExecutionArgs` object contains more information about the current execution context, as illustrated in the following table:

Property	Description
Method	The method or constructor being executed (in case of generic methods, this property is set to the proper generic instance of the method).
Arguments	The arguments passed to the method. In case of out and ref arguments, the argument values can be modified by the implementation of the <code>OnSuccess(MethodExecutionArgs)</code> or <code>OnExit(MethodExecutionArgs)</code> advices.
Instance	The object on which the method is being executed, i.e. the value of the <code>this</code> keyword.
ReturnValue	The return value of the method. This property can be modified by the aspect.
Exception	The Exception thrown by the method. This value can be modified (see below).

NOTE

The properties of the `MethodExecutionArgs` class cannot be directly viewed in the debugger. Because of optimizations, the properties must be referenced in your source code in order to be viewable in the debugger.

Example

The following program illustrates how to consume the current context from the `MethodExecutionArgs` parameter:

```
[Serializable]
publicclass LoggingAspect : OnMethodBoundaryAspect
{
    publicoverridevoid OnEntry(MethodExecutionArgs args)
    {
        Console.WriteLine("Method {0}({1}) started.", args.Method.Name, string.Join( " ", args.Arguments ));
    }

    publicoverridevoid OnSuccess(MethodExecutionArgs args)
    {
        Console.WriteLine("Method {0}({1}) returned {2}.", args.Method.Name, string.Join( " ", args.Arguments ), args.ReturnValue);
    }
}

staticclass Program
{
    staticvoid Main()
    {
        Foo( 1, 2 );
    }

    staticint Foo(int a, int b)
    {
        Console.WriteLine("Hello, world.");
        return3;
    }
}
```

When this program executes, it prints the following output:

```
Method Foo(1, 2) started.
Hello, world.
Method Foo(1, 2) returned 3.
```

Returning without executing the method

When your aspect is interacting with the target code, there are situations where you will need to alter the execution flow behavior. For example, your `OnEntry(MethodExecutionArgs)` advice may want to prevent the target method from being executed. PostSharp offers this ability through the use of the `MethodExecutionArgs.FlowBehavior` property. Unless the target method is void or is an iterator method, you will also need to set the `MethodExecutionArgs.ReturnValue` property.

```
publicoverridevoid OnEntry(MethodExecutionArgs args)
{
    if (args.Arguments.Count > 0 && args.Arguments[0] == null)
    {
        args.FlowBehavior = FlowBehavior.Return;
        args.ReturnValue = -1;
    }

    Console.WriteLine("The {0} method was entered with the parameter values: {1}",
        args.Method.Name, argValues.ToString());
}
```

As you can see, all that is needed to exit the execution of the target code is setting the `FlowBehavior` property on the `MethodExecutionArgs` to `Return`.

Managing execution flow control when dealing with exceptions there are two primary situations that you need to consider: re-throwing the exception and throwing a new exception.

Handling exceptions

When you implement the `OnException(MethodExecutionArgs)` class, PostSharp will generate a `try/catch` block and invoke `OnException(MethodExecutionArgs)` from the catch block.

The default behavior of the `OnException(MethodExecutionArgs)` advice is to rethrow the exception after the execution of the advice. You can also choose to ignore the exception, to replace it with another. For details, see [Handling Exceptions on page 359](#).

Sharing state between advices

When you are working with multiple advices on a single aspect, you will encounter the need to share state between these advices. For example, if you have created an aspect that times the execution of a method, you will need to track the starting time at `OnEntry(MethodExecutionArgs)` and share that with `OnExit(MethodExecutionArgs)` to calculate the duration of the call.

To do this we use the `MethodExecutionTag` property on the `MethodExecutionArgs` parameter in each of the advices. Because `MethodExecutionTag` is an object type, you will need to cast the value stored in it while retrieving it and before using it.

```
[Serializable]
publicclass ProfilingAspect : OnMethodBoundaryAspect
{
    publicoverridevoid OnEntry(MethodExecutionArgs args)
    {
        args.MethodExecutionTag = Stopwatch.StartNew();
    }

    publicoverridevoid OnExit(MethodExecutionArgs args)
    {
        var sw = (Stopwatch)args.MethodExecutionTag;
        sw.Stop();

        System.Diagnostics.Debug.WriteLine("{0} executed in {1} seconds", args.Method.Name,
                                           sw.ElapsedMilliseconds / 1000);
    }
}
```

NOTE

The value stored in `MethodExecutionTag` will not be shared between different instances of the aspect. If the aspect is attached to two different pieces of target code, each attachment will have its own unshared `MethodExecutionTag` for state storage.

Working with async methods

The specificity of `async` methods is that their execution can be suspended while they are awaiting a dependent operation (typically another `async` method or a `Task`). While an `async` method is suspended, it does not block any thread. When the dependent operation has completed, the execution of the `async` method can be resumed, possibly on a different thread than the one the method was previously executing on.

There are many situations in which you may want to execute some logic when an `async` method is being suspended or resumed. For instance, a profiling aspect may exclude the time when the method is waiting for a dependency. You can achieve this by overriding the `OnYield(MethodExecutionArgs)` and `OnResume(MethodExecutionArgs)` methods of the `OnMethodBoundaryAspect` class.

The following table shows the advices that are specific to `async` methods (and iterator methods).

Advice	Description
OnYield(MethodExecutionArgs)	When the method execution being suspended, i.e. when the operand of the <code>await</code> operator is a task that has not yet completed.
OnResume(MethodExecutionArgs)	When the method execution being resumed, i.e. when the operand of the <code>await</code> operator has completed.

NOTE

When the operand of the `await` is a task that has already completed, the `OnYield(MethodExecutionArgs)` and `OnResume(MethodExecutionArgs)` methods are not invoked.

See [Semantic Advising of Iterator and Async Methods on page 382](#) for more details regarding the behavior `OnMethodBoundaryAspect` aspect on asynchronous methods.

Example

In this example, we will reuse the `ProfilingAttribute` class from the previous section and will extend it to exclude the time spent while waiting for dependent operations.

```
[Serializable]
public class ProfilingAspect : OnMethodBoundaryAspect
{
    public override void OnEntry(MethodExecutionArgs args)
    {
        args.MethodExecutionTag = Stopwatch.StartNew();
    }

    public override void OnExit(MethodExecutionArgs args)
    {
        var sw = (Stopwatch)args.MethodExecutionTag;
        sw.Stop();

        System.Diagnostics.Debug.WriteLine("{0} executed in {1} seconds", args.Method.Name,
                                           sw.ElapsedMilliseconds / 1000);
    }

    public override void OnYield( MethodExecutionArgs args )
    {
        Stopwatch sw = (Stopwatch) args.MethodExecutionTag;
        sw.Stop();
    }

    public override void OnResume( MethodExecutionArgs args )
    {
        Stopwatch sw = (Stopwatch) args.MethodExecutionTag;
        sw.Start();
    }
}
```

Let's apply the `[Profiling]` attribute to the `TestProfiling` method.

```
[Profiling]
public async Task TestProfiling()
{
    await Task.Delay( 3000 );
    Thread.Sleep( 1000 );
}
```

During the code execution, the stopwatch will start upon entering the `TestProfiling` method. It will stop before the `await` statement and resume when the task awaiting is done. Finally, the time measuring is stopped again before exiting the `TestProfiling` method and the result is written to the console.

```
Method ProfilingTest executed for 1007ms.
```

Working with iterator methods

Iterator methods are methods that contain the `yield` keyword. Under the hood, the C# or VB compiler transforms the iterator method into a state machine class that implements the `IEnumerableT` and `IEnumeratorT` interfaces. Calling the `MoveNext` method causes the method to execute until the next `yield` keyword. The keyword causes the method execution to be suspended, and it is resumed by the next call to `MoveNext`.

Just like with async methods, you can use the `OnYield(MethodExecutionArgs)` and `OnResume(MethodExecutionArgs)` methods to inject behaviors when an iterator method is suspended or resumed.

The following table explains the behavior of the different advices of the `OnMethodBoundaryAspect` aspects in the context of iterator methods.

Advice	Description
<code>OnEntry(MethodExecutionArgs)</code>	When the method execution starts, i.e. during the <i>first</i> call of the <code>MoveNext</code> method, before any user code.
<code>OnYield(MethodExecutionArgs)</code>	When the iterator method emits a result using the <code>yield</code> return statement (and the iterator execution is therefore suspended). The operand of the <code>yield</code> return statement (i.e. the value of the <code>IEnumeratorCurrent</code> property) can be read from the <code>YieldValue</code> property of the <code>MethodExecutionArgs</code> parameter.
<code>OnResume(MethodExecutionArgs)</code>	When the method execution being resumed by a call to <code>MoveNext</code> . Note, however, that the <i>first</i> call to <code>MoveNext</code> results in a call to <code>OnEntry(MethodExecutionArgs)</code> .
<code>OnSuccess(MethodExecutionArgs)</code>	When the iterator method execution succeeds (i.e. returns without an exception, causing the <code>MoveNext</code> method to return <code>false</code>) or is interrupted (by disposing of the enumerator), after any user code.
<code>OnException(MethodExecutionArgs)</code>	When the method execution fails with an exception, after any user code. It is equivalent to a <code>catch</code> block around the whole iterator method.
<code>OnExit(MethodExecutionArgs)</code>	When the iterator method execution successfully (when <code>MoveNext</code> method returns <code>false</code>), is interrupted (by disposing of the enumerator) or fails with an exception. This advice runs after any user code and after the <code>OnSuccess(MethodExecutionArgs)</code> or <code>OnException(MethodExecutionArgs)</code> method of the current aspect. It is equivalent to a <code>finally</code> block around the whole iterator.

See [Semantic Advising of Iterator and Async Methods](#) on page 382 for more details regarding the behavior `OnMethodBoundaryAspect` aspect on iterator methods.

Example

The following program illustrates the timing of different advices in the context of an iterator.

```
[Serializable]
class MyAspect : OnMethodBoundaryAspect
{
    public override void OnEntry(MethodExecutionArgs args)
    {
```

Developing Simple Aspects

```
        Console.WriteLine("! entry");
    }

    public override void OnResume(MethodExecutionArgs args)
    {
        Console.WriteLine("! resume");
    }
    public override void OnYield(MethodExecutionArgs args)
    {
        Console.WriteLine($"! yield return {args.YieldValue}");
    }

    public override void OnSuccess(MethodExecutionArgs args)
    {
        Console.WriteLine("! success");
    }

    public override void OnExit(MethodExecutionArgs args)
    {
        Console.WriteLine("! exit");
    }
}

class Program
{
    static void Main(string[] args)
    {
        foreach (var i in Foo())
        {
            Console.WriteLine($"# received {i}");
        }

        Console.WriteLine("# done");
    }

    [MyAspect]
    static IEnumerable<int> Foo()
    {
        Console.WriteLine("@ part 1");
        yield return 1;
        Console.WriteLine("@ part 2");
        yield return 2;
        Console.WriteLine("@ part 3");
        yield return 3;
        Console.WriteLine("@ part 4");
    }
}
}
```

Executing the program prints the following output:

```
! entry
@ part 1
! yield return 1
# received 1
! resume
@ part 2
! yield return 2
# received 2
! resume
@ part 3
! yield return 3
# received 3
! resume
@ part 4
! success
! exit
# done
```

Working with other enumerable methods

The behavior described above is also true when there is a method interception aspect ordered after the method boundary aspect. In that case, the method interception aspect could return any enumerable object, not just the iterator from the target iterator method. This means that the `OnYield` advice behaves differently:

Advice	Description
<code>OnYield(MethodExecutionArgs)</code>	At the end of the returned enumerator's <code>MoveNext</code> method

You can also enable this behavior for other methods with the return type `IEnumerableT` or `IEnumeratorT` using semantic advising. See [Semantic Advising of Iterator and Async Methods on page 382](#) for more details.

Technical details

When there is a non-semantically-advised method boundary aspect or a method interception aspect ordered after a semantically-advised method boundary aspect on an iterator method, or when a semantically-advised method boundary aspect is applied to a method that returns `IEnumerableT`, `IEnumeratorT` or their non-generic variants, Post-Sharp transforms the target method equivalently to this example:

Suppose the original target method is:

```
[MySemanticBoundaryAspect, MyInterceptionAspect] // in that order
static IEnumerable<int> Return2()
{
    yieldreturn 2;
}
```

Then that piece of code is equivalent to this code:

```
[MySemanticBoundaryAspect]
static IEnumerable<int> Return2()
{
    var innerEnumerable = Return2__OriginalMethod();
    var innerEnumerator = innerEnumerable?.GetEnumerator();
    if (innerEnumerator == null)
    {
        yieldbreak;
    }
    foreach(var element in innerEnumerator)
    {
        yieldreturn element;
    }
}
[MyInterceptionAspect]
static IEnumerable<int> Return2__OriginalMethod()
{
    yieldreturn 2;
}
```

62.2. Handling Exceptions

Adding exception handlers to code requires the addition of `try/catch` statements which can quickly pollute code.

Exception handling implemented this way is also not reusable, requiring the same logic to be implemented over and over wherever exceptions must be dealt with. Raw exceptions also present cryptic information and can often expose too much information to the user.

PostSharp provides a solution to these problems by allowing custom exception handling logic to be encapsulated into a reusable class, which is then easily applied as an attribute to all methods and properties where exceptions are to be dealt with.

This topic contains the following sections:

- [Intercepting an exception on page 360](#)
- [Specifying the type of handled exceptions on page 361](#)
- [Ignoring \("swallowing"\) exceptions on page 361](#)
- [Replacing or wrapping exceptions on page 362](#)
- [Accessing the current execution context on page 363](#)

Intercepting an exception

PostSharp provides the `OnExceptionAspect` class which is the base class from which exception handlers are to be derived from.

If you also need to execute logic before and upon success of the target method, you can derive your aspect from the `OnMethodBoundaryAspect` class. Both classes behave almost identically as far as exception handling is concerned.

Both classes define a virtual method named `OnException(MethodExecutionArgs)` method: this is the method where the exception handling logic (i.e. what would normally be in a `catch` statement) goes. A `MethodExecutionArgs` parameter is passed into this method by PostSharp; it contains information about the exception.

To create an exception handling aspect:

1. Add a reference to the *PostSharp* package to your project.
2. Derive a class from `OnExceptionAspect`.
3. Apply the `SerializableAttribute` to the class.
4. Override `OnException(MethodExecutionArgs)` and implement your exception handling logic in this class. The `Exception` object is available on the `Exception` property of the `MethodExecutionArgs` parameter.
5. Add the aspect to one or more methods. Since `OnExceptionAspect` derives from the `Attribute` class, you can just add the aspect custom attribute to the methods you need. If you need to add the aspect to more methods (for instance all public methods in a namespace), you can learn about more advanced techniques in [Adding Aspects to Code on page 109](#).

Example

The following snippet shows an example of an exception handler which watches for exceptions of any type, and then writes a message to the console when an exception occurs:

```
[Serializable]
publicclass PrintExceptionAttribute : OnExceptionAspect
{
    publicoverridevoid OnException(MethodExecutionArgs args)
    {
        Console.WriteLine(args.Exception.Message);
    }
}
```

Once created, apply the derived class to all methods and/or properties for which the exception handling logic is to be used, as shown in the following example:

```
class Customer
{
    publicstring FirstName { get; set; }
    publicstring LastName { get; set; }

    [PrintException]
    publicvoid StoreName(string path)
    {
        File.WriteAllText( path, string.Format( "{0} {1}", this.FirstName, this.LastName ) );
    }
}
```



```
}

```

Here `PrintException` will output a message when an exception occurs in trying to write text to a file.

Specifying the type of handled exceptions

The `GetType(MethodBase)` method can be used to return the type of the exception which is to be handled by this aspect. Otherwise, all exceptions will be caught and handled by this class. Note that the `GetType(MethodBase)` method is evaluated at build time.

If the aspect needs to handle several types of exception, the `GetType` should return a common base type, and the `OnException` implementation should be modified to dynamically handle different types of exception.

Example

In the following snippet, we updated the `PrintExceptionAttribute` aspect and added the possibility to specify from the custom attribute constructor which type of exception should be traced.

```
[Serializable]
publicclass PrintExceptionAttribute : OnExceptionAspect
{
    Type type;

    public PrintExceptionAttribute(Type type)
    {
        this.type = type;
    }

    // Method invoked at build time.// Should return the type of exceptions to be handled. publicoverride Type GetType
    {
        returnthis.type;
    }

    publicoverridevoid OnException(MethodExecutionArgs args)
    {
        Console.WriteLine(args.Exception.Message);
    }
}

class Customer
{
    publicstring FirstName { get; set; }
    publicstring LastName { get; set; }

    [PrintException(typeof(IOException)]
    publicvoid StoreName(string path)
    {
        File.WriteAllText( path, string.Format( "{0} {1}", this.FirstName, this.LastName ) );
    }
}

```

Ignoring ("swallowing") exceptions

The `FlowBehavior` member of `MethodExecutionArgs` in the exception handler's `OnException(MethodExecutionArgs)` method, can be set to ignore an exception. Note however that ignoring exceptions is generally dangerous and not recommended. In practice, it's only safe to ignore exceptions in event handlers (e.g. to display a message in a WPF form) and in thread entry points.

Exceptions can be ignored by setting the `FlowBehavior` property to `Return`. You must then set the return value of the method by setting the `ReturnValue` property.

Example

The following aspect catches all exceptions flowing from the methods the aspect is applied to, prints the exception to the console, and makes the target method return -1 instead of failing with an exception.

```
[Serializable]
publicclass PrintAndIgnoreExceptionAttribute : OnExceptionAspect
{
    publicoverridevoid OnException(MethodExecutionArgs args)
    {
        Console.WriteLine(args.Exception.Message);
        args.FlowBehavior = FlowBehavior.Return;
        args.ReturnValue = -1;
    }
}

publicclass Customer
{
    [PrintException(typeof(IOException))]
    publicint GetDataLength(string path)
    {
        return File.ReadAllText(path).Length;
    }
}
```

Replacing or wrapping exceptions

Many times, the original exception must be hidden from the user or the client of the service, and should be replaced by another exception. This can be done by setting the `MethodExecutionArgsFlowBehavior` property to `FlowBehavior.ThrowException` and the `MethodExecutionArgsException` to the new exception.

- `FlowBehavior.RethrowException`: rethrows the original exception after the exception handler exits. This is the default behavior for the `OnException(MethodExecutionArgs)` advice.
- `FlowBehavior.ThrowException`: throws a new exception once the exception handler exits. This is useful when details of the original exception should be hidden from the user or when a more meaningful exception is to be shown instead. When throwing a new exception, a new exception object must be assigned to the `Exception` member of `MethodExecutionArgs`. The following snippet shows the creation of a new `BusinessExceptionAttribute` which throws a `BusinessException` containing a description of the cause:

```
[Serializable]
publicsealedclass BusinessExceptionAttribute : OnExceptionAspect
{
    publicoverridevoid OnException(MethodExecutionArgs args)
    {
        args.FlowBehavior = FlowBehavior.ThrowException;
        args.Exception = new BusinessException("Bad Arguments", new Exception("One or more arguments were null. l
    }
}
```

NOTE

You can also throw a new exception from the `OnException(MethodExecutionArgs)` method, but the exception call stack will show the aspect method itself, while with `MethodExecutionArgsFlowBehavior`, the exception call stack will originate from the target method (unless there is an interception aspect on the target method, in which case the call stack will originate from an intermediate method).

Example

The following aspect handles all exceptions in the `BusinessServices` class by generating a GUID for it, writing all details to the trace file and then throwing a `BusinessException` showing just the incident GUID and hiding other details.

```
[Serializable]
publicsealedclass HandleExceptionsAttribute : OnExceptionAspect
{
    publicoverridevoid OnException(MethodExecutionArgs args)
    {
        Guid guid = Guid.NewGuid();

        // In a real-world app, we would file the exception in the QA database.
        Trace.WriteLine( #"Exception {guid}:");
        Trace.WriteLine(args.Exception.ToString());

        args.FlowBehavior = FlowBehavior.ThrowException;
        args.Exception = new BusinessException( $"The service failed unexpectedly. Please report the incident to the QA te.
    }
}

[HandleExceptions]
publicclass BusinessServices
{
    // Dozens of methods here.
}

publicclass BusinessException : Exception
{
    public BusinessException(string message) : base(message)
    {
    }
}
```

Accessing the current execution context

The `OnException(MethodExecutionArgs)` method requires one argument of type `MethodExecutionArgs`. This object gives access to the exception being handled, the identity of the method being executed, its arguments, and the current object.

The following table lists the pieces of context made available by the `MethodExecutionArgs` class.

Property	Description
Method	The method or constructor being executed (in case of generic methods, this property is set to the proper generic instance of the method). This is not necessarily equal to the method that originally threw the exception.
Arguments	The arguments passed to the method. In case of out and ref arguments, the argument values can be modified by the aspect.
Instance	The object on which the method is being executed, i.e. the value of the <code>this</code> keyword.
Exception	The <code>Exception</code> thrown by the method. This value can be modified (see below).

NOTE

The properties of the `MethodExecutionArgs` class cannot be directly viewed in the debugger. Because optimizations, the properties must be referenced in your source code in order to be viewable in the debugger.

Example

The following code improves the previous example by adding more context information to the print-out of the exception details.

```

[Serializable]
public sealed class HandleExceptionsAttribute : OnExceptionAspect
{
    public override void OnException(MethodExecutionArgs args)
    {
        Guid guid = Guid.NewGuid();

        // In a real-world app, we would file the exception in the QA database.
        Trace.WriteLine( $"Exception {guid} when invoking the method {args.Method.DeclaringType.FullName}.{args.Method.Name}");
        Trace.WriteLine(args.Exception.ToString());

        args.FlowBehavior = FlowBehavior.ThrowException;
        args.Exception = new BusinessException( $"The service failed unexpectedly. Please report the incident to the QA team." );
    }
}

```

62.3. Intercepting Methods

It is often useful to be able to intercept the invocation of a method and invoke your own hook in its place. Common use cases for this capability include dispatching the method execution to a different thread, asynchronously executing the method at a later time, and retrying the method call when an exception is thrown.

PostSharp addresses these needs with the `MethodInterceptionAspect` aspect class which intercepts the invocation of a method before the method is executed. It also allows you to invoke the original method and access its arguments and return value.

The current article covers method *interception*, for another approach to injecting behaviors into methods, see [Injecting Behaviors Before and After Method Execution on page 351](#).

This topic contains the following sections:

- [Intercepting a method call on page 364](#)
- [Accessing the current execution context on page 366](#)
- [Intercepting methods returning a Task, including async methods on page 366](#)

Intercepting a method call

To create an aspect that retries a method call on exception:

1. Add a reference to the `PostSharp` package to your project.
2. Create an aspect class and inherit `MethodInterceptionAspect`. Annotate the class with the `[SerializableAttribute]` custom attribute.
3. Override and implement the `OnInvoke(MethodInterceptionArgs)` method and implement the interception logic. Call `args.Proceed()` to invoke the intercepted method.

NOTE

Calling `base.OnInvoke()` is equivalent to calling `args.Proceed()`.

4. Add the aspect to one or more methods. Since `MethodInterceptionAspect` derives from the `Attribute` class, you can just add the aspect custom attribute to the methods you need. If you need to add the aspect to more methods (for instance all public methods in a namespace), you can learn about more advanced techniques in [Adding Aspects to Code on page 109](#).

Example

Consider the following `CustomerService` class which has methods to load and save customer entities and relies on calls to a database or a web-service.

```
publicclass CustomerService
{
    publicvoid Save(Customer customer)
    {
        // Database or web-service call.
    }
}
```

Occasionally, the connection to the underlying store may become unreliable and the application user is presented with the error message. To improve the user experience you may want to retry the failing operation several times before displaying the error message. In the following steps, we'll create a method interception class which can be applied to repository methods and will retry the invocation whenever an exception is thrown by the original method.

The complete aspect code is as follows:

```
[Serializable]
publicclass RetryOnExceptionAttribute : MethodInterceptionAspect
{
    public RetryOnExceptionAttribute()
    {
        this.MaxRetries = 3;
    }

    publicint MaxRetries { get; set; }

    publicoverridevoid OnInvoke(MethodInterceptionArgs args)
    {
        int retriesCounter = 0;

        while (true)
        {
            try
            {
                args.Proceed();
                return;
            }
            catch (Exception e)
            {
                retriesCounter++;
                if (retriesCounter > this.MaxRetries) throw;

                Console.WriteLine(
                    "Exception during attempt {0} of calling method {1}.{2}: {3}",
                    retriesCounter, args.Method.DeclaringType, args.Method.Name, e.Message);
            }
        }
    }
}
```

Apply the `[RetryOnException]` custom attributes to all methods where the behavior is needed.

In the following snippet, this aspect is applied to the `CustomerService.Save` method:

```
publicclass CustomerService
{
    [RetryOnException(MaxRetries = 5)]
    publicvoid Save(Customer customer)
    {
        // Database or web-service call.
    }
}
```

Whenever the `CustomerService.Save` method is invoked, the `RetryOnExceptionAttribute.OnInvoke` method is called instead. The aspect method will invoke the original method and retry if necessary.

Accessing the current execution context

As illustrated in the example above, you can access information about the method being intercepted from the property `MethodInterceptionArgs.Method`, which gives you the `MethodBase` of the method that has been intercepted. This object gives you access to parameters, return type, declaring type, and other characteristics. In case of generic methods or generic types, `MethodInterceptionArgs.Method` gives you the proper generic method instance, so you can use this object to get generic parameters.

The `MethodInterceptionArgs` parameter gives you access to other pieces of information regarding the current execution context.

Property	Description
Method	The method being executed (in case of generic methods, this property is set to the proper generic instance of the method).
Arguments	The arguments passed to the method. If you modify the arguments and call <code>args.Proceed()</code> , the intercepted method will be invoked with the modified arguments.
Instance	The object on which the method is being executed, i.e. the value of the <code>this</code> keyword.
ReturnValue	The return value of the method. This property is populated after the aspect calls <code>args.Proceed</code> . The aspect can then modify the value of this property if it needs to return a different value than the one returned by the intercepted method.

Intercepting methods returning a Task, including async methods

The `Task` class in .NET represents operations that can execute asynchronously. Whenever you want to intercept a method that returns a `Task`, you have two options of how to define the target of the interception:

- Intercepting the logic that creates and returns a new `Task`. The logic of the asynchronous operation represented by the `Task` is not intercepted, and the status of `Task`, return value and thrown exception are not handled by an aspect. This is what happens when you intercept a `Task`-returning method with an aspect implementing only `OnInvoke(MethodInterceptionArgs)`.
- Intercepting both the logic that instantiates the `Task` and the logic of the `Task`. In this case, you intercept the asynchronous operation represented by the task. You can await for the completion of the task, and you can handle the return value of the task and thrown exception inside the aspect. This interception mode is called *semantic advising*.

To intercept the whole `Task` logic, your aspect must implement the `OnInvokeAsync(MethodInterceptionArgs)` method. Your implementation can call `args.ProceedAsync()` instead of `args.Proceed()` to invoke the intercepted method and execute the intercepted task.

When an aspect implements both `OnInvoke(MethodInterceptionArgs)` and `OnInvokeAsync(MethodInterceptionArgs)`, `PostSharp` automatically selects the proper method when the target method returns a `Task`. You can change this behavior by changing the value of the `SemanticallyAdvisedMethodKinds` aspect property. See [Semantic Advising of Iterator and Async Methods on page 382](#) for details regarding this property.

In this article, we will demonstrate how to control the semantic behavior of the `MethodInterceptionAspect` aspect when it is applied to a method returning a task or to an async method. For a more general information about using `MethodInterceptionAspect` see [Intercepting Methods on page 364](#).

Example

To demonstrate when the semantic approach to method interception can be useful, let's extend at the `RetryOnExceptionAttribute` aspect created in the previous example.

The previous version of `RetryOnExceptionAttribute` aspect intercepts the original method, but it does not intercept the async task returned by the method. As a result, the aspect cannot catch the exception thrown by the async task and the test method fails. In order to properly handle async methods, we need to add an implementation of the `OnInvokeAsync(MethodInterceptionArgs)` method to the `RetryOnExceptionAttribute` aspect:

```
[Serializable]
publicclass RetryOnExceptionAttribute : MethodInterceptionAspect
{
    public RetryOnExceptionAttribute()
    {
        this.MaxRetries = 3;
    }

    publicint MaxRetries { get; set; }

    publicoverridevoid OnInvoke(MethodInterceptionArgs args)
    {
        int retriesCounter = 0;

        while (true)
        {
            try
            {
                args.Proceed();
                return;
            }
            catch (Exception e)
            {
                retriesCounter++;
                if (retriesCounter > this.MaxRetries) throw;

                Console.WriteLine(
                    "Exception during attempt {0} of calling method {1}.{2}: {3}",
                    retriesCounter, args.Method.DeclaringType, args.Method.Name, e.Message);
            }
        }
    }

    publicoverrideasync Task OnInvokeAsync(MethodInterceptionArgs args)
    {
        int retriesCounter = 0;

        while (true)
        {
            try
            {
                await args.ProceedAsync();
                return;
            }
            catch (Exception e)
            {
                retriesCounter++;
                if (retriesCounter > this.MaxRetries) throw;

                Console.WriteLine(
                    "Exception during attempt {0} of calling method {1}.{2}: {3}",
                    retriesCounter, args.Method.DeclaringType, args.Method.Name, e.Message);
            }
        }
    }
}
```

We can now apply the `[RetryOnException]` aspect to an async method:

```
[RetryOnException]
private async Task AsyncThrow()
{
    await Task.Yield();
    if (--this.counter > 0)
    {
        throw new Exception();
    }
}
```

Whenever the `AsyncThrow` method is invoked, the `RetryOnExceptionAttribute.OnInvokeAsync` method is called instead. The aspect method will invoke the original method asynchronously and await for its completion. If the asynchronous task throws an exception, the aspect will catch the exception and either retry the asynchronous call or re-throw the exception.

Intercepting methods returning a null Task

The following example shows how to intercept methods that may return a null `Task`. There is no way for an aspect based on a semantically-advised `MethodInterceptionAspect` to return a null `Task`. If the aspect requires returning a null `Task`, it must use non-semantic advising. See [Semantic Advising of Iterator and Async Methods on page 382](#) for details.

```
[Serializable]
public sealed class MyAspect1 : MethodInterceptionAspect
{
    public override async Task OnInvokeAsync(MethodInterceptionArgs args)
    {
        object instance = args.Instance;
        Arguments arguments = args.Arguments;

        MethodBindingInvokeAwaitable bindingInvokeAwaitable = args.AsyncBinding.InvokeAsync(ref instance, arguments);

        Task task = bindingInvokeAwaitable.GetTask();

        if (task != null)
        {
            args.ReturnValue = await bindingInvokeAwaitable;
        }
        else
        {
            args.ReturnValue = "Some special value";
        }
    }
}
```

62.4. Intercepting Properties and Fields

In .NET, both fields and properties are "things" that can be set and get. You can intercept get and set operations using the `LocationInterceptionAspect`. It makes it possible to develop useful aspects, such as validation, filtering, change tracking, change notification, or property virtualization (where the property is backed by a registry value, for instance).

This topic contains the following sections:

- [Intercepting Get operations on page 369](#)
- [Intercepting Set operations on page 370](#)
- [Getting and setting the underlying property on page 371](#)
- [Intercepting fields on page 372](#)
- [Getting the property or property being accessed on page 372](#)
- [Reacting to initialization of instance fields and properties on page 373](#)

Intercepting Get operations

In this example, we will see how to create an aspect that filters the value read from a field or property.

To create an aspect that filters the value read from a field or property

1. Add a reference to the *PostSharp* package to your project.
2. Create an aspect that inherits from *LocationInterceptionAspect* and add the custom attribute *[SerializableAttribute]*.
3. Override the *OnGetValue(LocationInterceptionArgs)* method.

```
[Serializable]
publicclass StringCheckerAttribute : LocationInterceptionAspect
{
    publicoverridevoid OnGetValue(LocationInterceptionArgs args)
    {
        base.OnGetValue(args);
    }
}
```

4. Calling `base.OnGetValue` actually retrieves the value from the underlying field or property, and populates the **Value** property. Add some code to check if the property currently is set to `null`. If the current value is `null`, we want to return a predefined value. To do this we can set the **Value** property. Any time this property is requested, and it is set to `null`, the value "foo" will be returned.

```
publicoverridevoid OnGetValue(LocationInterceptionArgs args)
{
    base.OnGetValue(args);

    if (args.Value == null)
    {
        args.Value = "foo";
    }
}
```

5. Now that you have a complete getter interception aspect written you can attach it to the target code. Simply add an attribute to either properties or fields to have the interception attached.

```
publicclass Customer
{
    [StringChecker]
    privatereadonlystring _address;

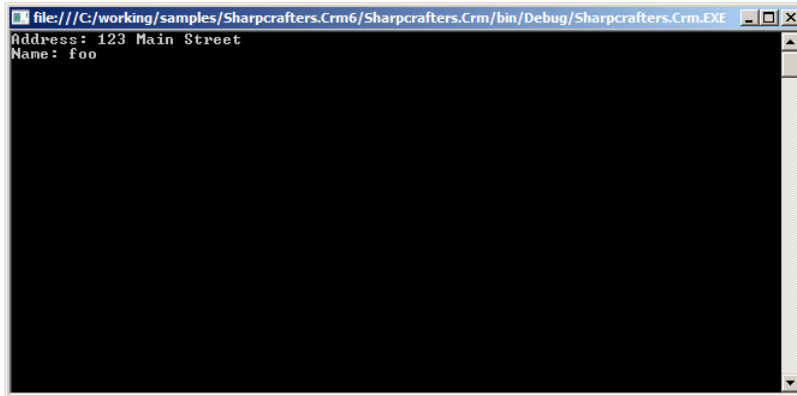
    public Customer(string address)
    {
        _address = address;
    }
    [StringChecker]
    publicstring Name { get; set; }
    publicstring Address { get { return _address; } }
}
```

NOTE

Adding aspects to target code one property or field at a time can be a tedious process. There are a number of techniques in the article [Adding Aspects to Multiple Declarations on page 112](#) that explain how to add aspects en mass.

- Now when you create an instance of a customer and immediately try to access the Name and Address values the get request will be intercepted and null values will be returned as "foo".

```
class Program
{
    static void Main(string[] args)
    {
        var customer = new Customer("123 Main Street");
        Console.WriteLine("Address: {0}", customer.Address);
        Console.WriteLine("Name: {0}", customer.Name);
        Console.ReadKey();
    }
}
```



Property and field interception is a simple and seamless task. Once you have intercepted your target you can act on the target or you can allow the original code to execute.

Intercepting Set operations

The previous section showed how to intercept a get accessor. Intercepting a set accessor is accomplished in a similar manner by implementing `OnSetValue(LocationInterceptionArgs)` in the `LocationInterceptionAspect`.

The following snippet shows the addition of `OnSetValue(LocationInterceptionArgs)` to the `StringCheckerAttribute` example:

```
[Serializable]
public class StringCheckerAttribute : LocationInterceptionAspect
{
    public override void OnGetValue(LocationInterceptionArgs args)
    {
        base.OnGetValue(args);
    }

    public override void OnSetValue(LocationInterceptionArgs args)
    {
        base.OnSetValue(args);
    }
}
```

When applied to a property with a set operator, `OnSetValue(LocationInterceptionArgs)` will intercept the set operation. In the `Customer` example shown below, `OnSetValue(LocationInterceptionArgs)` will be called whenever the `Name` property is set:

```
public class Customer
{
    .
    .
    .
}
```

```
[StringChecker]
publicstring Name { get; set; }
}
```

The `SetNewValue(Object)` method of `LocationInterceptionArgs` can be used instead of `base.OnSetValue()` to pass a different value in for the property. For example, `OnSetValue(LocationInterceptionArgs)` could be used to check for a null string, and then change the string to a non-null value:

```
[PSerializable]
publicclass StringCheckerAttribute : LocationInterceptionAspect
{
    .
    .
    .
    publicoverridevoid OnSetValue(LocationInterceptionArgs args)
    {
        if (args.Value == null)
        {
            args.Value = "Empty String";
        }

        args.ProceedSetValue();
    }
}
```

CAUTION NOTE

Inline initializers of instance fields and properties are not intercepted by `OnSetValue(LocationInterceptionArgs)`. To react to the initialization of an instance field or property, use the `OnInstanceLocationInitialized(LocationInitializationArgs)` advice. See [Reacting to initialization of instance fields and properties on page 373](#) for details.

Getting and setting the underlying property

PostSharp provides a mechanism to check a property's underlying value via the `LocationInterceptionArgs.GetCurrentValue` method. This can be useful to check the current property value when a setter is called and then take some appropriate action.

For example, the following snippet shows a modified `OnSetValue(LocationInterceptionArgs)` method which gets the current underlying property value and compares the (new) value passed into the setter against the current value. If current and new value don't match then some message is written:

```
publicoverridevoid OnSetValue(LocationInterceptionArgs args)
{
    // get the current underlying value
    string existingValue = (string)args.GetCurrentValue();

    if (((existingValue==null) && (args.Value != null)) || (!existingValue.Equals(args.Value)))
    {
        Console.WriteLine("Value changed.");
        args.ProceedSetValue();
    }
}
```

NOTE

`GetCurrentValue` will call the underlying property getter without going through `OnGetValue(LocationInterceptionArgs)`. If several aspects are applied to the property (and/or to the property setter), `GetCurrentValue` will go through the next aspect in the chain of invocation.

PostSharp also provides a mechanism to set the underlying property in a getter via the `SetNewValue(Object)` method of `LocationInterceptionArgs`. This could be used for example, to ensure that a default value is assigned to the underlying property if there is currently no value. The following snippet shows a modified `OnGetValue(LocationInterceptionArgs)` method which gets the current underlying value, and sets a default value if the current value is null:

```
public override void OnGetValue(LocationInterceptionArgs args)
{
    object o = args.GetCurrentValue();
    if (o == null)
    {
        args.SetNewValue("value not set");
    }

    base.OnGetValue(args);
}
```

Intercepting fields

One benefit to implementing a `LocationInterceptionAspect` is that it can be applied directly to fields, allowing for reads and writes to those fields to be intercepted, just like with properties.

Applying a `LocationInterceptionAspect` implementation to a field is simply a matter of setting it as an attribute on a field, just as it was done with a property:

```
public class Customer
{
    .
    :
    .
    [StringChecker]
    public string name;
}
```

With the attribute applied to the `name` field, all attempts to get and set that field will be intercepted by `StringChecker` in its `OnGetValue(LocationInterceptionArgs)` and `OnSetValue(LocationInterceptionArgs)` methods.

Note that when a `LocationInterceptionAspect` is added to a field, the field is replaced by a property of the same field and visibility. The field itself is renamed and made private.

Getting the property or property being accessed

Information about the property or field being intercepted can be obtained through the `LocationInterceptionArgs` via its **Location** property. The type of this property, `LocationInfo`, can represent a `FieldInfo`, a `PropertyInfo`, or a `ParameterInfo` (although `LocationInterceptionAspect` cannot be added to parameters).

One use for this is to reflect the property name whenever a property is changed. In the following example, we have an `Entity` class that implements `INotifyPropertyChanged` and a public `OnPropertyChanged` method which allows notifications to be made whenever a property is changed. The `Customer` class has been modified to derive from `Entity`.

```
class Entity : INotifyPropertyChanged
{
    public event PropertyChangedEventHandler PropertyChanged;

    public void OnPropertyChanged(string propertyName)
    {
        if (PropertyChanged != null)
            PropertyChanged(this, new PropertyChangedEventArgs(propertyName));
    }
}

class Customer : Entity
{
```

```
publicstring Name { get; set; }
}
```

With the ability to invoke an `OnPropertyChanged` event, we can create a `LocationInterceptionAspect` which invokes this event when setting a value and pass in the property name from the underlying `PropertyInfo` object:

```
[Serializable]
publicclass NotifyPropertyChangedAttribute : LocationInterceptionAspect
{
    publicoverridevoid OnSetValue(LocationInterceptionArgs args)
    {
        If ( args.Value != args.GetCurrentValue() )
        {
            args.Value = args.Value;
            args.ProceedSetValue();
            ((Entity)args.Instance).OnPropertyChanged(args.Location.Name);
        }
    }
}
```

NOTE

This example is a simplistic implementation of the `NotifyPropertyChangedAttribute` aspect. For a production-ready implementation, see the section `INotifyPropertyChanged` on page 203.

This aspect can then be applied to the `Customer` class:

```
[NotifyPropertyChangedAttribute]
class Customer : INotifyPropertyChanged
{
    publicstring Name { get; set; }
}
```

Now when the `Name` property is changed, `NotifyPropertyChangedAttribute` will invoke the `Entity.OnPropertyChanged` method passing in the property name retrieved from its underlying property.

Reacting to initialization of instance fields and properties

C# and VB.NET allow fields and properties to be initialized on the same line as their declaration instead of in the constructor. This is called the *initialization* of the field or property. While the initialization of *static* fields and properties can be intercepted by `OnSetValue(LocationInterceptionArgs)`, this method will not be called while initializing instance fields or properties. The reason for that limitation is that the `OnSetValue(LocationInterceptionArgs)` method requires a valid `this` object, which we don't have because inline initialization is performed before the base constructor has been called.

The initialization of instance fields or properties cannot be intercepted. However, you can react to their initial value by implementing the `OnInstanceLocationInitialized(LocationInitializationArgs)` advice. This advice will be invoked after the completion of the base constructor and before the execution of your own constructor.

Example

Suppose you have an item in an auction and you want to take an action each time the price of the item changes. You have the following class:

```
class AuctionItem
{
    [Report]
    publicint Price { get; set; } = 40000;
}
```

And you write your `LocationInterceptionAspect`:

```
[Serializable]
class ReportAttribute : LocationInterceptionAspect
{
    publicvoid OnSetValue(LocationInterceptionArgs args)
    {
        args.ProceedSetValue();
        Console.WriteLine("The " + args.LocationName + " of the item changed to " + args.Value);
    }

    publicvoid OnInstanceLocationInitialized(LocationInitializationArgs args)
    {
        Console.WriteLine("The " + args.LocationName + " of the item starts out at " + args.Value);
    }
}
```

Then the following code snippet:

```
AuctionItem item = new AuctionItem();
item.Price += 2000;
```

will have this output:

```
The Price of the item starts out at 40000
The Price of the item changed to 42000
```

See the Remarks section of `OnSetValue(LocationInterceptionArgs)` and `OnInstanceLocation-Initialized(LocationInitializationArgs)` for more details.

62.5. Intercepting Events

You interact with events in three primary ways: subscribing, unsubscribing and raising them. Like methods and properties, you may find yourself needing to intercept these three interactions. How do you execute code every time that an event is subscribed to? Or raised? Or unsubscribed? *PostSharp* provides you with a simple mechanism to accomplish this easily.

This topic contains the following sections:

- [Intercepting Add and Remove on page 374](#)
- [Intercepting Raise on page 375](#)
- [Accessing the current context on page 376](#)
- [Example: Removing offending event subscribers on page 376](#)
- [Intercepting the event initializer on page 376](#)

Intercepting Add and Remove

Throughout the life of an event, it is possible to have many different event handlers subscribe and unsubscribe. You may want to log each of these actions.

1. Add a reference to the *PostSharp* package to your project.
2. Create an aspect that inherits from `EventInterceptionAspect`. Add the `[PSerializableAttribute]` custom attribute.
3. Override the `OnAddHandler(EventInterceptionArgs)` method and add your logging code to the method body.

4. Add the `base.OnAddHandler` call to the body of the `OnAddHandler(EventInterceptionArgs)` method. If this is omitted, the original call to add a handler will not be executed. Unless you want to stop the addition of the handler, you will need to add this line of code.

```
[Serializable]
publicclass CustomEventHandling : EventInterceptionAspect
{
    publicoverridevoid OnAddHandler(EventInterceptionArgs args)
    {
        base.OnAddHandler(args);
        Console.WriteLine("A handler was added");
    }
}
```

5. To log the removal of an event handler, override the `OnRemoveHandler(EventInterceptionArgs)` method.
6. Add the logging you require to the method body.
7. Add the `base.OnRemoveHandler` call to the body of the `OnRemoveHandler(EventInterceptionArgs)` method. Like you saw when overriding the `OnAddHandler(EventInterceptionArgs)` method, if you omit this call, the original call to remove the handler will not occur.

```
publicoverridevoid OnRemoveHandler(EventInterceptionArgs args)
{
    base.OnRemoveHandler(args);
    Console.WriteLine("A handler was removed");
}
```

Once you have defined the interception points in the aspect, you will need to attach the aspect to the target code. The simplest way to do this is to add the attribute to the event handler definition.

```
publicclass Example
{
    [CustomEventHandling]
    publicevent EventHandler<EventArgs> SomeEvent;

    publicvoid DoSomething()
    {
        if (SomeEvent != null)
        {
            SomeEvent.Invoke(this, EventArgs.Empty);
        }
    }
}
```

Intercepting Raise

When you are intercepting events, you may also have situations where you want to execute additional code when the event is raised. Raising of an event can occur in many places and you will want to centralize this code to avoid repetition.

1. Override the `OnInvokeHandler(EventInterceptionArgs)` method in your aspect class and add the logging you require to the method body.
2. Add a call to `base.OnInvokeHandler` to ensure that the original invocation occurs.

```
publicoverridevoid OnInvokeHandler(EventInterceptionArgs args)
{
    base.OnInvokeHandler(args);
    Console.WriteLine("A handler was invoked");
}
```

By adding the attribute to the target event handler earlier in this process you have enabled intercepting of each raised event.

Accessing the current context

At any time, the `Handler` property is set to the delegate being added, removed, or invoked. You can read and write this property. If you write it, the delegate you assign must be compatible with the type of the event. The `Event` property gets you the `EventInfo` of the event being accessed.

Within `OnInvokeHandler(EventInterceptionArgs)`, the property `Arguments` gives access to the arguments with which the delegate was invoked.

These concepts will be illustrated in the following example.

Example: Removing offending event subscribers

When events are subscribed to, the component that raises the event has no way to ensure that the subscriber will behave properly when that event is raised. It's possible that the subscribing code will throw an exception when the event is raised and when that happens you may want to unsubscribe the handler to ensure that it doesn't continue to throw the exception. The `EventInterceptionAspect` can help you to accomplish this easily.

1. Override the `OnInvokeHandler(EventInterceptionArgs)` method in your aspect.
2. In the method body add a `try...catch` block.
3. In the `try` block add a call to `base.OnInvokeHandler` and in the `catch` block add a call to `RemoveHandler(Delegate)`

```
[PSerializable]
publicclass CustomEventHandling : EventInterceptionAspect
{
    publicoverridevoid OnInvokeHandler(EventInterceptionArgs args)
    {
        try
        {
            base.OnInvokeHandler(args);
        }
        catch (Exception e)
        {
            Console.WriteLine("Handler '{0}' invoked with arguments {1} failed with exception {2}.",
                args.Handler.Method,
                string.Join(", ", args.Arguments.Select(a => a == null ? "null" : a.ToString())),
                e.GetType().Name);

            args.RemoveHandler(args.Handler);
            throw;
        }
    }
}
```

Now, any time an exception is thrown during event execution, the offending event handler will be unsubscribed from the event.

Intercepting the event initializer

The `OnAddHandler(EventInterceptionArgs)` method does not intercept the initializer of a field-like event. If you want to intercept the adding of all handlers, do not use event initializers and instead add the initial handler in the constructor.

```
publicclass TargetClass
{
    [EventInterception]
    publicevent EventHandler FieldLikeEvent = EventHandler1; // will not be intercepted
    {
        this.FieldLikeEvent += EventHandler2; // will be intercepted
    }
}
```


62.6. Introducing Custom Attributes

Applying custom attributes to class members in C# is a powerful way to add metadata about those members at compile time.

PostSharp provides the ability to create a custom attribute class which when applied to another class, can iterate through those class members and automatically decorate them with custom attributes. This can be useful for example, to automatically apply custom attributes or groups of custom attributes when new class members are added, without having to remember to do it manually each time.

This topic contains the following sections:

- [Introducing new custom attributes on page 377](#)
- [Copying existing custom attributes on page 378](#)

Introducing new custom attributes

In the following example, we'll create an attribute decorator class which applies .NET's `DataContractAttribute` to a class and `DataMemberAttribute` to members of a class at build time.

1. Start by creating a class called `AutoDataContractAttribute` which derives from `TypeLevelAspect`. `TypeLevelAspect` transforms the class into an attribute which can be applied to other classes. Also implement `IAспектProvider` which exposes the `ProvideAspects(Object)` method for iterating on class members. `ProvideAspects(Object)` will be called for each member in the target class and will contain the code for applying the attributes:

```
publicsealedclass AutoDataContractAttribute : TypeLevelAspect, IAspectProvider
{
    public IEnumerable<AspectInstance> ProvideAspects(object targetElement)
    {
    }
}
```

2. Implement the `ProvideAspects(Object)` method to cast the `targetElement` parameter to a `Type` object. Note that this method will be called at build time. Since `ProvideAspects(Object)` will be called for the class itself and for each member of the target class, the `Type` object can be used for inspecting each member and making decisions about when and how to apply custom attributes. In the following snippet, the implementation returns a new `AspectInstance` for the `Type` containing a new `DataContractAttribute` and then iterates through each property of the `Type` returning a new `AspectInstance` with the `DataMemberAttribute` for each. Note that both the `DataContractAttribute` and `DataMemberAttribute` are both wrapped in `CustomAttributeIntroductionAspect` objects:

```
publicsealedclass AutoDataContractAttribute : TypeLevelAspect, IAspectProvider
{
    // This method is called at build time and should just provide other aspects. public IEnumerable<AspectInstar
    {
        Type targetType = (Type) targetElement;

        CustomAttributeIntroductionAspect introduceDataContractAspect =
            new CustomAttributeIntroductionAspect(
                new ObjectConstruction(typeof (DataContractAttribute).GetConstructor(Type.EmptyTypes)));
        CustomAttributeIntroductionAspect introduceDataMemberAspect =
            new CustomAttributeIntroductionAspect(
                new ObjectConstruction(typeof (DataMemberAttribute).GetConstructor(Type.EmptyTypes)));

        // Add the DataContract attribute to the type. yieldreturnnew AspectInstance(targetType, introduceDataCor
        // Add a DataMember attribute to every relevant property. foreach (PropertyInfo property in
            targetType.GetProperties(BindingFlags.Public | BindingFlags.DeclaredOnly | BindingFlags.Instance))
        {
            if (property.CanWrite)
                yieldreturnnew AspectInstance(property, introduceDataMemberAspect);
        }
    }
}
```

NOTE

Since the `ProvideAspects(Object)` method returns an `IEnumerable`, the `yield` keyword should be used to return aspects for PostSharp to apply.

3. Apply the `AutoDataContractAttribute` class. In the following example we apply it to a `Product` class where it will decorate `Product` with `DataContractAttribute` and each member with `DataMemberAttribute`:

```
[AutoDataContractAttribute]
publicclass Product
{
    publicint ID { get; set; }

    publicstring Name { get; set; }

    publicint RevisionNumber { get; set; }
}
```

Copying existing custom attributes

Another way to introduce attributes to class members is to copy them from another class. This is useful, for example, when distinct classes have members with the same names and are of the same types. In this case, attributes can be defined in one class and then that class can be used to decorate other similar classes with same attributes.

In the following snippet, `Product`'s `ID` and `Name` properties have both been modified to contain an additional attribute from the `System.ComponentModel.DataAnnotations` namespace – `Editable`, `Display`, and `Required` respectively.

Below Product is another class called ProductViewModel containing the same properties to which we want to copy the attributes to:

```
class Product
{
    [EditableAttribute(false)]
    [Required]
    public int Id { get; set; }

    [Display(Name = "The product's name")]
    [Required]
    public string Name { get; set; }
    public int RevisionNumber { get; set; }
}

class ProductViewModel
{
    public int Id { get; set; }
    public string Name { get; set; }
    public int RevisionNumber { get; set; }
}
```

To copy the attributes from the properties of Product to the corresponding properties of ProductViewModel, create an attribute class which can be applied to ProductViewModel to perform this copy process:

1. Create a `TypeLevelAspect` which implements `IAspectProvider`. In the snippet below our class is called `CopyCustomAttributesFrom`:

```
class CopyCustomAttributesFrom : TypeLevelAspect, IAspectProvider
{
}
```

2. Create a constructor to take in the class type from which the property attributes are to be copied from. This class type will be used in the next step to enumerate its properties:

```
class CopyCustomAttributesFrom : TypeLevelAspect, IAspectProvider
{
    private Type sourceType;

    public CopyCustomAttributesFrom(Type srcType)
    {
        sourceType = srcType;
    }
}
```

3. Implement ProvideAspects(Object):

```
class CopyCustomAttributesFrom : TypeLevelAspect, IAspectProvider
{
    // Details skipped.
    public IEnumerable<AspectInstance> ProvideAspects(object targetElement)
    {
        Type targetClassType = (Type)targetElement;

        //loop thru each property in target
        foreach (PropertyInfo targetPropertyInfo in targetClassType.GetProperties())
        {
            PropertyInfo sourcePropertyInfo = sourceType.GetProperty(targetPropertyInfo.Name);

            //loop thru all custom attributes for the source property and copy to the target property
            foreach (CustomAttributeData customAttributeData in sourcePropertyInfo.GetCustomAttributesData())
            {
                //filter out attributes that aren't DataAnnotations
                if (customAttributeData.AttributeType.Namespace != "System.ComponentModel.DataAnnotations")
                {
                    CustomAttributeIntroductionAspect customAttributeIntroductionAspect =
                        new CustomAttributeIntroductionAspect(new ObjectConstruction(customAttributeData));

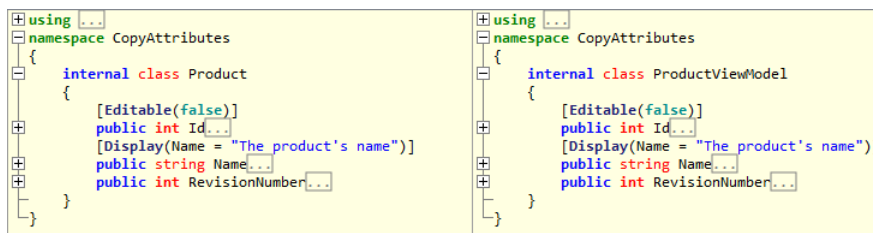
                    yield return new AspectInstance(targetPropertyInfo, customAttributeIntroductionAspect);
                }
            }
        }
    }
}
```

The ProvideAspects(Object) method iterates through each property of the target class and then gets the corresponding property from the source class. It then iterates through all custom attributes defined for the source property, copying each to the corresponding property of the target class. ProvideAspects(Object) also filters out attributes which aren't from the System.ComponentModel.DataAnnotations namespace to demonstrate how you may want to ignore some attributes during the copy process.

- Decorate the ProductViewModel class with the CopyCustomAttributesFrom attribute, specifying Product as the source type in the constructor. During compilation, CopyCustomAttributesFrom's ProvideAspects(Object) method will then perform the copy process from Product to ProductViewModel:

```
[CopyCustomAttributesFrom(typeof(Product))]
class ProductViewModel
{
    // Details skipped.
}
```

The following screenshot shows the Product and ProductViewModel classes reflected from an assembly. Here we can see that the Editable and Display attributes were copied from Product to ProductViewModel using CopyCustomAttributesAttribute at build time:



NOTE

It is not possible to delete or replace an existing custom attribute.

62.7. Introducing Managed Resources

Embedding resources in .NET allows for data to be packaged together with your code in an assembly. Resources are normally specified at design time and then embedded by the compiler during build time.

PostSharp's `AssemblyLevelAspect` adds additional flexibility by allowing you to programmatically add resources at compile time. In doing so you can add logic and therefore flexibility in determining which resources get embedded and how. For example, you could use this feature to encrypt a resource just before embedding it into your assembly.

Introducing resources

In the following example, we'll create an assembly decorator which retrieves the current date and time during compilation, and then stores that information in the current assembly as a resource. The example will then show that that information can be retrieved from the assembly at run time.

1. Start by creating a class called `AddBuildInfoAspect` which derives from `AssemblyLevelAspect`. Also implement `IAспектProvider` which exposes the `ProvideAspects(Object)` method. The `ProvideAspects(Object)` method will be called once by PostSharp, providing access to assembly information and allowing for a resource to be programmatically added to the assembly:

```
publicsealedclass AddBuildInfoAspect : AssemblyLevelAspect, IAspectProvider
{
    public IEnumerable<AspectInstance> ProvideAspects(object targetElement)
    {
    }
}
```

2. Implement the `ProvideAspects(Object)` method:

```
publicsealedclass AddBuildInfoAspect : AssemblyLevelAspect, IAspectProvider
{
    public IEnumerable<AspectInstance> ProvideAspects(object targetElement)
    {
        Assembly assembly = (Assembly)targetElement;

        byte[] userNameData = Encoding.ASCII.GetBytes(
            assembly.FullName + " was compiled by: " + Environment.UserName);
        ManagedResourceIntroductionAspect mria2 = new ManagedResourceIntroductionAspect("BuildUser", userNameData);

        yieldreturnnew AspectInstance(assembly, mria2);
    }
}
```

In this example, the `targetElement` object passed in is cast to an `Assembly` object from which the assembly named is retrieved. The code then gets the current date and time, concatenates it with the assembly name, and then converts this string to a byte array. The byte array is then stored along with a name for the data in PostSharp's `ManagedResourceIntroductionAspect` object, and returned via an `AspectInstance`. PostSharp then embeds the resource into the current assembly.

3. Open your project's `AssemblyInfo.cs` file and add a line to include the `AddBuildInfoAspect` class:

```
[assembly:AddBuildInfoAspect]
```

With this code in place, the assembly will now embed the date and time as a resource into itself during compilation.

The following code demonstrates how to retrieve the data at run time:

```
class Program
{
    staticvoid Main(string[] args)
    {
        Assembly a = Assembly.GetExecutingAssembly();
```

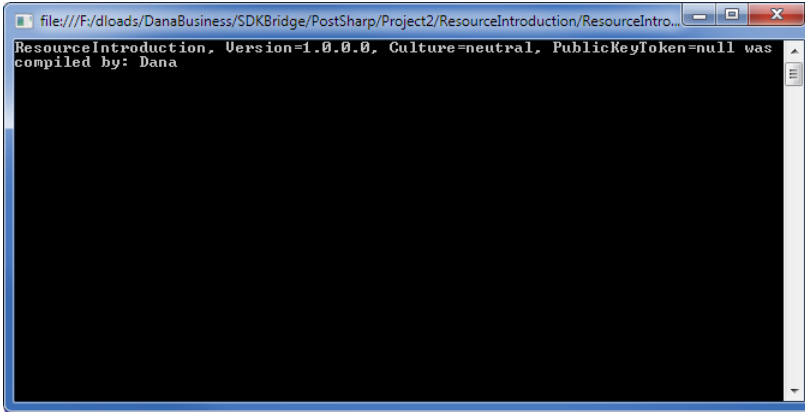
```

Stream stream = a.GetManifestResourceStream("BuildUser");

byte[] bytesRead = newbyte[stream.Length];
stream.Read(bytesRead, 0, (int)stream.Length);
stringValue = Encoding.ASCII.GetString(bytesRead);
Console.WriteLine(value);
    }
}

```

This will display the following line in the console window:



62.8. Semantic Advising of Iterator and Async Methods

In most C# and VB methods, the source code is very similar to the way how the method is actually executed by the .NET runtime. However, with async and iterator methods, mapping between source code and assembly code is far from being straightforward. The compiler performs a complex transformation of the source code and generates a state machine type. If you disassemble an async or iterator method, you would just find some instructions that instantiate this state machine. When you apply an aspect to an async or iterator method, it leads to an ambiguity whether the aspect should be applied *semantically* at the abstraction level of the source code, or whether it should be applied *non-semantically* at the abstraction level of the assembly code.

By default, PostSharp applies *semantic advising* for async and iterator methods. It also uses semantic advising for all methods returning a Task.

This article discusses all details of semantic advising.

This topic contains the following sections:

- [Semantic advising for asynchronous code on page 382](#)
- [Semantic advising for iterators on page 384](#)
- [Semantic advising vs non-semantic advising on page 384](#)
- [Supported and default advising modes on page 385](#)
- [Enabling and disabling semantic advising on page 386](#)
- [Coping with situations where semantic advising is not available on page 387](#)

Semantic advising for asynchronous code

Consider the following code snippet:

```

publicclass FlowerService
{
    [MyAspect]

```

```

public Task<Flower> GetFlowerAsync1( int flowerId, string connectionString )
{
    var connection = ConnectionManager.GetConnection( connectionString );
    returnthis.GetFlowerAsync2( flowerId, connection );
}

[MyAspect]
public async Task<Flower> GetFlowerAsync2( int flowerId, Connection connection )
{
    var flowerData = await connection.GetFlowerAsync( flowerId );
    var familyData = await connection.GetFlowerFamilyAsync( flowerData.FamilyId );
    returnnew Flower( flowerId, flowerData.Name, familyData.Name );
}
}

```

The `GetFlowerAsync1` method returns a `Task` but is not `async` (it would be useless and hurt performance to make it `async`). The `GetFlowerAsync2` method both returns a `Task` and is `async`.

Both methods are enhanced by `[MyAspect]`. For different behaviors implemented by `MyAspect`, how would you expect `MyAspect` to work?

- If `MyAspect` was an exception handler, you would probably expect all exceptions to be caught by `MyAspect`, including exceptions thrown by the `GetFlowerAsync` and `GetFlowerFamilyAsync` methods and the constructor of the `Flower` class. You would be deceived to realize that the aspect only handles exceptions thrown in the process of instantiating `Task<Flower>` and execute the part of the task that can run synchronously.
- If `MyAspect` was a profiling aspect, you would probably want to measure the time taken by the whole method to execute. That is, you will probably be interested in the time of the whole `Task<Flower>` to run to completion, not just the time to instantiate it and run to the first waiting point.
- If `MyAspect` was a caching aspect, you would probably want to cache the `Flower` object, not the `Task<Flower>` itself.

That is, most of the time, you want the aspect to apply to the *semantic* of the method, not to its *implementation* (how it is implemented in MSIL and executed by the .NET runtime).

We use the term *semantic advising* when an aspect or advice is applied to the level of abstraction of the programming language (C# or VB). Non-semantic advising or low-level advising means that PostSharp applies the aspect to the level of abstraction of MSIL.

Semantic advising is the default behavior for all methods returning a `Task` (or any other awaitable type such as `ValueTask`) and all `async` methods.

Synchronization context

In method boundary aspects, all advices are executed in the synchronization context of the caller of the advised method. For example, if you await an `async` method from the event handler of an event in a Windows Forms application, all advices (such as `OnEntry(MethodExecutionArgs)` and `OnExit(MethodExecutionArgs)`) are also executed on the Windows Forms UI thread.

It is generally a good choice to execute the advices in the current synchronization context. However, it can cause a deadlock in a situation where the synchronization context is blocked until the advised method completes.

Example of the deadlock:

```

[OnMethodBoundaryAspect1]
Task Return4()
{
    return Task.Delay(500);
}
void button1_Clicked(object sender, EventArgs args)
{

```

```
Return4().Wait(); // blocks the Windows Forms UI thread// without PostSharp, the thread would get unblocked when the del:
}
```

If you do not want to execute advices in the current synchronization context, use `MethodInterceptionAspect` instead of `OnMethodBoundaryAspect`. The interception aspect gives you more control over the synchronization context.

Semantic advising for iterators

The notion of semantic advising also applies to iterator methods, i.e. methods that include the `yield return` statement.

Consider the following code snippet:

```
public class PostcardService
{
    [MyAspect]
    public IEnumerable<Postcard> GetPostcards1( )
    {
        return new [] { new Postcard("Hello from Alaska"), new Postcard("Hello from Siberia") };
    }

    [MyAspect]
    public IEnumerable<Postcard> GetPostcards2( int flowerId, Connection connection )
    {
        yield return new Postcard("Hello from Alaska");
        yield return new Postcard("Hello from Siberia");
    }
}
```

The `GetPostcards2` method requires special attention. Under the hood, the C# or VB compiler generates a new class implementing the `IEnumerableT` and `IEnumeratorT` interfaces, called the enumerator class. At runtime, calling the `GetPostcards2` method only instantiates the enumerator class. The initial logic of `GetPostcards2` is moved to the `MoveNext` method of the enumerator.

Let's do the same exercise as for asynchronous methods. For different behaviors implemented by `MyAspect`, how would you expect `MyAspect` to work?

- If `MyAspect` was an exception handler, you would probably want the aspect to catch any exception thrown by the C# code that you can see. That is, in `GetPostcards2`, you actually want to catch exceptions in the `MoveNext` method of the enumerator class. In this case, you need semantic advising.
- If `MyAspect` was a profiling aspect, you may want to only measure the time when `GetPostcards2` is actually executing, but exclude the time when the caller is processing the data returned by the enumerator. Therefore, you will also want to add behaviors to the `MoveNext` method of the enumerator class. In this case again, you need semantic advising.
- If `MyAspect` was a caching aspect, however, you will want to cache a copy of the enumerator itself, therefore you will need to enhance the `GetPostcards2` method and not the `MoveNext` method. In this case, you don't need semantic advising.

Semantic advising vs non-semantic advising

The following table compares semantic advising with non-semantic advising in several situations.

	Semantic Advising	Non-Semantic Advising
Async methods:		
Code covered or intercepted by the aspect	The whole async method.	The part of the async method before the first <code>await</code> operator whose operand (typically a <code>Task</code>) has not yet completed.

	Semantic Advising	Non-Semantic Advising
Return value	The operand of the return statement, i.e. <code>TaskTResultResult</code> .	The <code>TaskTResult</code> object itself.
Non-async methods returning a Task:		
Code covered or intercepted by the aspect	Both the code that instantiates or gets the <code>Task</code> and the whole execution of the <code>Task</code> .	The compiler-generated code that instantiates or gets the <code>Task</code> (and if the <code>Task</code> represents an async method, plus the first segment of the method that runs synchronously).
Return value	The value of the <code>TaskTResultResult</code> property.	The <code>TaskTResult</code> object itself.
Iterator methods:		
Code covered or intercepted by the aspect	The whole method.	The compiler-generated code that instantiates the enumerator class (no user code is covered).
Return value	None.	The enumerator.
Non-iterator methods returning IEnumerableT:		
Code covered or intercepted by the aspect	The returned enumerator's <code>MoveNext</code> method.	The method that is being enhanced by the aspect.
Return value	None.	The enumerator or enumerable object.
Async iterator methods:		
Code covered or intercepted by the aspect	(not supported)	The compiler-generated code that instantiates the async enumerator class (no user code is covered).
Return value	(not supported)	The async enumerator.

Supported and default advising modes

Semantic advising is available for the `OnMethodBoundaryAspect`, `OnExceptionAspect` and `MethodInterceptionAspect` aspects. Whenever semantic advising makes sense, it is the default advising mode, except for normal methods returning `IEnumerableT` or `IEnumeratorT` which are not semantically advised by default because we expect that non-semantic advising is more often what you need for those methods.

The following table specifies where semantic advising is supported and where it is the default advising mode.

Target methods	Advising modes supported by On-MethodBoundaryAspect and On-ExceptionAspect	Advising modes supported by MethodInterception-Aspect	Default advising mode
async method returning void	Semantic and non-semantic	Non-semantic	Semantic
async method returning a Task	Semantic and non-semantic	Semantic and non-semantic	Semantic
async method returning something else than void or a Task	Semantic and non-semantic	Non-semantic	Semantic
Iterator method	Semantic and non-semantic	Non-semantic	Semantic
Normal method returning IEnumerableT or IEnumeratorT	Semantic and non-semantic	Non-semantic	Non-semantic
Async iterator method (C# 8)	Non-semantic	Non-semantic	Semantic

CAUTION NOTE

With `MethodInterceptionAspect`, the default advising mode is *always* non-semantic when the `OnInvokeAsync(MethodInterceptionArgs)` advice is not implemented by the aspect.

The default mode is always semantic, even in situations where semantic advising is not available. This design allows us to implement support for semantic advising in future versions of PostSharp without breaking backward compatibility. However, PostSharp will emit a build-time error if you try to use semantic advising on a method that is not supported.

The next sections explain how to opt out from semantic advising and how to cope with situations when semantic advising is not available.

Enabling and disabling semantic advising

You can disable or enable semantic advising by setting the `SemanticallyAdvisedMethodKinds` property of your aspect. You would typically set this property in the constructor or in the `CompileTimeInitialize(MethodBase, AspectInfo)` method.

If you want to disable semantic advising in all situations, set the `SemanticallyAdvisedMethodKinds` property to `None`. Otherwise, you can select individual situations in which semantic advising should be applied by setting the property to a bitwise combination of the values of the `SemanticallyAdvisedMethodKinds` enumeration. For instance, the `Async | Iterator` value instructs PostSharp to use semantic advising for `async` methods and iterators but not for other methods returning a `Task` or an enumerable.

Example

The following code snippet shows how to configure a caching aspect so that semantic advising is used for methods returning a `Task` but not for methods returning an enumerable.

```
[Serializable]
publicclass CacheAttribute : MethodInterceptionAspect
{
    public CacheAttribute()
    {
        this.SemanticallyAdvisedMethodKinds = SemanticallyAdvisedMethodKinds.ReturnsAwaitable;
    }
}
```

```
// Detailed skipped.  
}
```

Coping with situations where semantic advising is not available

By default, PostSharp will emit a build-time error if you're applying a semantically advising aspect to a method that does not support it (for instance, an asynchronous `MethodInterceptionAspect` cannot be applied to an `async void` method).

Instead of failing with an error, you can change the behavior by setting the `UnsupportedTargetAction` aspect property. The default value is `Fail`. You can choose `Ignore` to silently skip applying the aspect or advice to the target method, or `Fallback` to apply non-semantic advising.

If you are using composite aspects, you can change the attribute property `UnsupportedTargetAction` and similarly named properties on other advices.

CHAPTER 63

Understanding Aspect Lifetime and Scope

An original feature of PostSharp is that aspects are instantiated at compile time. Most other frameworks instantiate aspects at run time.

Persistence of aspects between compile time and run time is achieved by serializing aspect instances into a binary resource stored in the transformed assembly. Therefore, you should carefully mark all aspect classes with the `PSerializableAttribute` custom attribute, and distinguish between serialized fields (typically initialized at compile-time and used at run-time) and non-serialized fields (typically used at run-time only or at compile-time only).

This topic contains the following sections:

- Scope of Aspects
- Steps in the Lifetime of an Aspect Instance

Scope of Aspects

PostSharp offers two kinds of aspect scopes: static (per-class) and per-instance.

Statically Scoped Aspects

With statically-scoped aspects, PostSharp creates one aspect instance for each element of code to which the aspect applies. The aspect instance is stored in a static field and is shared among all instances of the target class.

In generic types, the aspect instance has not exactly the same scope as static fields. Consider the following piece of code:

```
publicclass GenericClass<T>
{
    static T f;

    [Trace]
    publicvoidvoid SetField(T value) { f = value; }
}

publicclass Program
{
    publicstaticvoid Main()
    {
        GenericClass<int>.SetField(1);
        GenericClass<long>.SetField(2);
    }
}
```

In this program, there are two instances of the static field `f` (one for `GenericClass<int>`, the second for `GenericClass<long>`) but only a single instance of the aspect `Trace`.

Instance-Scoped Aspects

Instance-scoped aspects have the same scope (instance or static) as the element of code to which they are applied. If an instance-scoped aspect is applied to a static member, it will have static scope. However, if it is applied to an instance member or to a class, it will have the same lifetime as the class instance: an aspect instance will be created whenever the class is instantiated, and the aspect instance will be garbage-collectable at the same time as the class instance.

Instance-scoped aspects are implemented according to the *“prototype pattern”*: the aspect instance created at compile time serves as a prototype, and is cloned at run-time whenever the target class is instantiated.

Instance-scoped aspects must implement the interface `IInstanceScopedAspect`. Any aspect may be made instance-scoped. The following code is a typical implementation of the interface `IInstanceScopedAspect`:

```
object IInstanceScopedAspect.CreateInstance( AdviceArgs adviceArgs )
{
    return this.MemberwiseClone();
}

void IInstanceScopedAspect.RuntimeInitializeInstance()
{
}
```

Steps in the Lifetime of an Aspect Instance

The following table summarizes the different steps of the aspect instance lifetime:

Phase	Step	Description
Compile-Time	Instantiation	PostSharp creates a new instance of the aspect for every target to which it applies. If the aspect has been applied using a multicast custom attribute (<code>MulticastAttribute</code>), there will be one aspect instance for each matching element of code. When the aspect is given as a custom attribute or a multicast custom attribute, each custom attribute instance is instantiated using the same mechanism as the Common Language Runtime (CLR) does: PostSharp calls the appropriate constructor and sets the properties and/or fields with the appropriate values. For instance, when you use the construction <code>[Trace(Category="FileManager")]</code> , PostSharp calls the default constructor and the <code>Category</code> property setter.
	Validation	PostSharp validates the aspect by calling the <code>CompileTimeValidate</code> aspect method. See Validating Aspect Usage on page 393 for details.
	Compile-Time Initialization	PostSharp invokes the <code>CompileTimeInitialize</code> aspect method. This method may be overridden by concrete aspect classes in order to perform some expensive computations that do not depend on runtime conditions. The name of the element to which the custom attribute instance is applied is always passed to this method.
Run-Time	Serialization	After the aspect instances have all been created and initialized, PostSharp serializes them into a binary stream. This stream is stored inside the new assembly as a managed resource.
	Deserialization	Before the first aspect must be executed, the aspect framework deserializes the binary stream that has been stored in a managed resource during post-compilation. At this point, there is still one aspect instance per target class.
	Per-Class Runtime Initialization	Once all custom attribute instances are deserialized, we call for each of them the <code>RuntimeInitialize</code> method. But this time we pass as an argument the real <code>System.Reflection</code> object to which it is applied.
	Per-Instance Runtime Initialization	This step applies only to instance-scoped aspects when they have been applied to an instance member. When a class is instantiated, the aspect framework creates an aspect instance by invoking the method <code>CreateInstance(AdviceArgs)</code> of the prototype aspect instance. After the new aspect instance has been set up, the aspect framework invokes the <code>RuntimeInitializeInstance</code> .
	Advice Execution	Finally, advices (methods such as <code>OnEntry(MethodExecutionArgs)</code>) are executed.

CHAPTER 64

Initializing Aspects

As explained in the section [Understanding Aspect Lifetime and Scope](#) on page 389, a different aspect instance is associated with every element of code it is applied to. Aspect instances are created at compile time, serialized into the assembly as a managed resource, and deserialized at run time. If the aspect is instance-scoped, instances are duplicated from the prototype and initialized.

Therefore, you can override one of the following three methods to handle aspect initializations:

1. The method `CompileTimeInitialize` is invoked at compile time, and should initialize only serializable fields of the aspect, so that the value of these fields will be available at run time. The argument of this method is the `System.Reflection` object representing the element of code to which this aspect instance has been applied. Therefore, this method can already perform expensive computations that depend only on metadata.
2. The method `RuntimeInitialize` is invoked at run time. Note that the aspect constructor itself is not invoked at run time. Therefore, overriding `RuntimeInitialize` is the only way to perform initialization tasks at run time. If the aspect is instance-scoped, this method is executed on the prototype instance.
3. The methods `IInstanceScopedAspectCreateInstance(AdviceArgs)` and `IInstanceScopedAspectRuntimeInitializeInstance` is invoked only for instance-scoped aspects. They initialize the aspect instance itself, as `RuntimeInitialize` was invoked on the prototype.

TIP

Initializing an aspect at compile time is useful when you need to compute a difficult result that depends only on metadata -- that is, it does not depend on any runtime information. An example is to build the strings that need to be printed by a tracing aspect. It is rather expensive to build strings that contain the full type name, the method name, and eventually placeholders for generic parameters and parameters. However, all required pieces of information are available at compile time. So compile time is the best moment to compute these strings.

64.1. Coping with Custom Object Serializers

Some aspects need to be initialized when a new instance of the class to which they are applied is created. For instance, instance-scoped aspect must be cloned from the prototype; members imported into the through `ImportMemberAttribute` must be bound to aspect fields.

PostSharp enhances every constructor of every enhanced class so that aspects are properly initialized.

However, it is possible to create new instances of classes by *bypassing* the constructor. This happens, for instance, when classes are deserialized by the `BinaryFormatter` or the `DataContractSerializer`. These formatters use the method `FormatterServices.GetUninitializedObject(Type)` to create new instances, but this method bypasses all constructors.

PostSharp implements a workaround for the deserializers `BinaryFormatter` and `DataContractSerializer`: it creates or modifies a method annotated by the custom attribute `OnDeserializingAttribute`, so that aspects are initialized properly.

However, if you are using a custom deserializer, or for any reason create instances using the method `FormatterServices.GetUninitializedObject(Type)`, you will have to initialize aspects manually.

Initializing Aspects Manually

There are two possible ways to initialize an aspect from user code.

By Defining a Method `InitializeAspects`

You can define in your classes (typically in one of the root classes of your class hierarchy) a method with the following name and signature:

```
protectedvirtualvoid InitializeAspects();
```

When PostSharp discovers this method, it will insert its own initialization logic at the beginning of the `InitializeAspects` method. The original logic is not deleted. This method can safely have an empty implementation.

The following constraints apply:

- The method should be `virtual` unless the class is sealed.
- The method should be `protected` or `public` unless the class is `internal`.

For instance, the following class would enable aspects (applied to this class or on derived classes) to be initialized after deserialization (note that PostSharp automatically generates this code for `BinaryFormatter` and `DataContractSerializer`; you only need to do it manually for a custom serializer).

```
[DataContract]
publicabstractclass BaseClass
{
    protectedvirtualvoid InitializeAspects()
    {
    }

    [OnDeserializing]
    privatevoid OnDeserializingInitializeAspects()
    {
        this.InitializeAspects();
    }
}
```

By Invoking `AspectUtilities.InitializeCurrentAspects`

Instead of providing an empty method `InitializeAspects`, it is possible to invoke the method `AspectUtilities.InitializeCurrentAspects`. A call to this method will be translated into a call to `InitializeAspects`. It has to be invoked from a non-static method of an enhanced class.

If the class from which `InitializeCurrentAspects` is invoked has not been enhanced by an aspect requiring initialization, the call to this method is simply ignored.

NOTE

Using this approach may be brittle in some situations: calls to `InitializeCurrentAspects` will have no effect if aspects are applied to derived classes, but not to the calling class. In this scenario, it is preferable to define the method `InitializeAspects`.

CHAPTER 65

Validating Aspect Usage

Some aspects make sense only on a specific subset of targets. For instance, an aspect may require being applied to non-static methods only. Another aspect may not be compatible with methods that have ref or out parameters. If these constraints are not respected, these aspects will fail at run time. However, defects detected by the compiler are always cheaper to fix than ones detected later. So, as the developer of an aspect, you should ensure that the build will fail if your aspect is being used on an invalid target.

This topic contains the following sections:

- [Using \[MulticastAttributeUsage\] on page 393](#)
- [Implementing CompileTimeValidate on page 393](#)
- [Using Message Sources on page 394](#)
- [Validating Attributes That Are Not Aspects on page 395](#)

Using [MulticastAttributeUsage]

The first level of protection is to configure multicasting properly with [MulticastAttributeUsageAttribute], as described in the article [Adding Aspects Declaratively Using Attributes on page 111](#). However, this approach can only filter based on characteristics that are supported by the multicasting component.

Implementing CompileTimeValidate

The best way to validate aspect usage is to override the `CompileTimeValidate(Object)` method of your aspect class.

In this example, we will show how an aspect `RequirePermissionAttribute` can require being applied only to methods of types that implement the `ISecurable` interface.

1. Inherit from one of the pre-built aspects. In this case, `OnMethodBoundaryAspect`.

```
publicclass RequirePermissionAttribute: OnMethodBoundaryAspect
```

2. Override the `CompileTimeValidate(Object)` method.

```
publicoverridebool CompileTimeValidate(MethodBase target)
{
```

3. Perform a check to see if the target class implements the interface in question.

```
    Type targetType = target.DeclaringType;
    if (!typeof(ISecurable).IsAssignableFrom(targetType))
    {
    }
}
```

4. If the target does not implement the interface you must signal the compilation process that this target should not have the aspect applied to it. There are two ways to do this. The first option is to throw an `InvalidAnnotationException`.

```
if (!typeof(ISecurable).IsAssignableFrom(targetType))
{
    thrownew InvalidAnnotationException("The target type does not implement ISecurable.");
}
```

5. The second option is to emit an error message to the compilation process.

```
if (!typeof(ISecurable).IsAssignableFrom(targetType))
{
    Message.Write(SeverityType.Error, "Custom01",
        "The target type does not implement ISecurable.", target);
    returnfalse;
}
```

NOTE

You may have noticed that `CompileTimeValidate(Object)` returns a boolean value. If you only return `false` from this method the compilation process will silently ignore it. You must either throw the `InvalidAnnotationException` or emit an error message to not silently ignore the `false` return value.

Making use of the `CompileTimeValidate(Object)` method is a great way to encode custom rules for applying aspects to target code. While it could be used to duplicate the functionality of the `AttributeTargetTypeAttributes` or `AttributeTargetMemberAttributes`, its real power is to go beyond those filtering techniques. By using `CompileTimeValidate(Object)` you are able to filter aspect application in any manner that you can interrogate your codebase using reflection.

Using Message Sources

If you plan to raise many messages, you may prefer to define your own `MessageSource`. A `MessageSource` is backed by a managed resource mapping error codes to error messages.

In order to create your own `MessageSource`, you should:

1. Create an implementation of the `IMessageDispenser`. Typically, implement the `GetMessage(String)` method using a large switch statement. To each message will correspond a string

2. Create a static instance of the `MessageSource` class for your message source.

For instance, the following code defines a message source based on a message dispenser:

```
internalclass ArchitectureMessageSource : MessageSource
{
    publicstaticreadonly ArchitectureMessageSource Instance = new ArchitectureMessageSource();

    private ArchitectureMessageSource() : base( "PostSharp.Architecture", new Dispenser() )
    {
    }

    privateclass Dispenser : MessageDispenser
    {
        public Dispenser() : base( "CUS" )
        {
        }

        protectedoverridestring GetMessage( int number)
        {
            switch ( number )
            {
                case1:
                    return"Interface {0} cannot be implemented by {1} because of the [InternalImplement] constrai

                case2:
                    return"{0} {1} cannot be referenced from {2} {3} because of the [ComponentInternal] constrair

                case3:
                    return"Cannot use [ComponentInternal] on {0} {1} because the {0} is not internal.";

                case4:
                    return"Cannot use [Internal] on {0} {1} because the {0} is not public.";

                default:
                    returnnull;
            }
        }
    }
}
```

3. Then you can use a convenient set of methods on your `MessageSource` object:

```
MyMessageSource.Instance.Write( classType, SeverityType.Error, "CUS001", newobject[] { interfaceType, classType }
```

NOTE

You can also emit information and warning messages.

TIP

Use `ReflectionSearch` to perform complex queries over `System.Reflection`.

Validating Attributes That Are Not Aspects

You can validate any attribute derived from `Attribute` by implementing the interface `IValidableAnnotation`.

CHAPTER 66

Developing Composite Aspects

PostSharp offers two approaches to aspect-oriented development. The first, as explained in section [Developing Simple Aspects on page 351](#), is very similar to object-oriented programming. It requires the aspect developer to override virtual methods or implement interfaces. This approach is very efficient for simple problems.

One way to grow in complexity with the first approach is to use the interface `IAAspectProvider` (see [Adding Aspects Dynamically on page 408](#)). However, even this technique has its limitations.

This chapter documents the second approach, closer to the classic paradigm of aspect-oriented programming introduced by AspectJ. This approach allows developers to implement more complex design patterns using aspects. We call the aspects developed with this approach *composite aspects*, because they are freely composed of different elements named *advices* and *pointcuts*.

An *advice* is anything that adds a behavior or a structural element to an element of code. For instance, introducing a method into a class, intercepting a property setter, or catching exceptions, are advices.

A *pointcut* is a function returning a set of elements of code to which advices apply. For instance, a function returning the set of properties annotated with the custom attribute `DataMember` is a pointcut.

Classes supporting advices and pointcuts are available in the namespace `PostSharp.Aspects.Advices`.

A composite aspect generally derives from a class that does not define its own advices: `AssemblyLevelAspect`, `TypeLevelAspect`, `InstanceLevelAspect`, `MethodLevelAspect`, `LocationLevelAspect` or `EventLevelAspect`. As such, these aspects have no functionality. You can add functionalities by adding advices to the aspect.

Advices are covered in the following sections:

Section	Description
Adding Behaviors to Existing Members on page 398	Advices with equivalent functionality as <code>OnMethodBoundaryAspect</code> , <code>MethodInterceptionAspect</code> , <code>LocationInterceptionAspect</code> , and <code>EventInterceptionAspect</code> .
Introducing Interfaces, Methods, Properties and Events on page 402	Make the aspect introduce an interface into the target class. The interface is implemented by the aspect itself.
Accessing Members of the Target Class on page 406	Make the aspect introduce a new method, property or event into the target class. The new member is implemented by the aspect itself. Conversely, the aspect can import a member of the target so that it can invoke it through a delegate.

66.1. Adding Behaviors to Existing Members

In order to add new behaviors to (i.e. modify) existing members (methods, fields, properties, or events), two questions must be addressed:

- **What** transformation should be performed? The answer lays in the *advice*. This advice is a method of your advice, annotated with a custom attribute determining in which situation the method should be invoked. You can freely choose the name of the method, but its signature must match the one expected by the advice type.
- **Where** should it be performed, i.e. on which elements of code? The answer lays in the *pointcut*, another custom attribute expected on the method providing the transformation.

This topic contains the following sections:

- [How to Add a Behavior to an Existing Member](#)
- [Advice Kinds on page 399](#)
- [Pointcuts Kinds on page 399](#)
- [Grouping Advices on page 400](#)

How to Add a Behavior to an Existing Member

1. Start with an empty aspect class deriving `AssemblyLevelAspect`, `TypeLevelAspect`, `InstanceLevelAspect`, `MethodLevelAspect`, `LocationLevelAspect` or `EventLevelAspect`. Mark it as serializable.
2. Choose an advice type in the list below. For instance: `OnMethodEntryAdvice`.
3. Create a method. The signature of this method should match exactly the signature matched by this advice type.
4. Annotate this method with a custom attribute of the advice type you chose. For instance: `[OnMethodEntryAdvice]`.
5. Choose a pointcut type in the list below. For instance: `SelfPointcut`. Annotate the advice method with that custom attribute. For instance: `[SelfPointcut]`.

Example

The following code shows a simple tracing aspect implemented with an advice and a pointcut. This aspect is exactly equivalent to a class derived from `OnMethodBoundaryAspect` where only the method `OnEntry(MethodExecutionArgs)` has been overwritten. The example is a method-level aspect and `SelfPointcut` means that the advice applies to the same target as the method itself.

```
using System;
using PostSharp.Aspects;
using PostSharp.Aspects.Advices;
using PostSharp.Serialization;

namespace Samples6
{
    [PSerializable]
    public sealed class TraceAttribute : MethodLevelAspect
    {
        [OnMethodEntryAdvice, SelfPointcut]
        public void OnEntry(MethodExecutionArgs args)
        {
            Console.WriteLine("Entering {0}.{1}", args.Method.DeclaringType.Name, args.Method.Name);
        }
    }
}
```

Advice Kinds

The following table lists all types of advices that can transform existing members. Note that all these advices are available as a part of a simple aspect (for instance `OnMethodEntryAdvice` corresponds to `OnMethodBoundaryAspectOn-Entry(MethodExecutionArgs)`). For a complete documentation of the advice, see the documentation of the corresponding simple aspect.

Advice Type	Targets	Description
<code>OnMethodEntryAdvice</code> <code>OnMethodSuccessAdvice</code> <code>OnMethodExceptionAdvice</code> <code>OnMethodExitAdvice</code>	Methods	These advices are equivalent to the advices of the aspect <code>OnMethodBoundaryAspect</code> . The target method to be wrapped by a <code>try/catch/finally</code> construct.
<code>OnMethodInvokeAdvice</code>	Methods	This advice is equivalent to the aspect <code>MethodInterceptionAspect</code> . Calls to the target methods are replaced to calls to the advice.
<code>OnLocationGetValueAdvice</code> <code>OnLocationSetValueAdvice</code>	Fields, Properties	These advices are equivalent to the advices of the aspect <code>LocationInterceptionAspect</code> . Fields are changed into properties, and calls to the accessors are replaced to calls to the proper advice.
<code>LocationValidationAdvice</code>	Fields, Properties, Parameters	This advice is equivalent to the <code>ValidateValue(T, String, LocationKind, LocationValidationContext)</code> method of the <code>ILocationValidationAspectT</code> aspect interface. It validates values assigned to their targets and throws an exception in case of error.
<code>OnEventAddHandlerAdvice</code> <code>OnEventRemoveHandlerAdvice</code> <code>OnEventInvokeHandlerAdvice</code>	Events	These advices are equivalent to the advices of the aspect <code>EventInterceptionAspect</code> . Calls to add and remove semantics are replaced by calls to advices. When the event is fired, the <code>OnEventInvokeHandler</code> is invoked for each handler, instead of the handler itself.

Pointcuts Kinds

Pointcuts determine *where* the transformation provided by the advice should be applied.

From a logical point of view, pointcuts are functions that return a set of code elements. A pointcut can only select elements of code that are inside the target of the aspect itself. For instance, if an aspect has been applied to a class `A`, the pointcut can select the class `A` itself, members of `A`, but not different classes or members of different classes.

Multicast Pointcut

The pointcut type `MulticastPointcut` allows expressing a pointcut in a purely declarative way, using a single custom attribute. It works in a very similar way as `MulticastAttribute` (see [Adding Aspects Declaratively Using Attributes on page 111](#)) the kind of code elements being selected, their name and attributes can be filtered using properties of this custom attribute.

For instance, the following code applies the `OnPropertySet` advice to all non-abstract properties of the class to which the aspect has been applied.

```
[OnLocationSetValueAdvice,
MulticastPointcut( Targets = MulticastTargets.Property,
                  Attributes = MulticastAttributes.Instance | MulticastAttributes.NonAbstract)]
publicvoid OnPropertySet( LocationInterceptionArgs args )
{
    // Details skipped.
}
```

Method Pointcut

The pointcut type `MethodPointcut` allows expressing a pointcut imperatively, using a C# or VB method. The argument of the custom attribute should contain the name of the method implementing the pointcut.

The only parameter of this method should be type-compatible with the kind of elements of code to which the *aspect* applies. The return value of the pointcut method should be a collection (`IEnumerable<T>`) of objects that are type-compatible with the kind of elements of code to which the *advice* applies.

For instance, the following code applies the `OnPropertySet` advice to all writable properties that are not annotated with the `IgnorePropertyChanged` custom attribute.

```
private IEnumerable<PropertyInfo> SelectProperties( Type type )
{
    const BindingFlags bindingFlags = BindingFlags.Instance |
        BindingFlags.DeclaredOnly | BindingFlags.Public;

    return from property
        in type.GetProperties( bindingFlags )
        where property.CanWrite && !property.IsDefined(typeof(IgnorePropertyChanged))
        select property;
}

[OnLocationSetValueAdvice, MethodPointcut( "SelectProperties" )]
public void OnPropertySet( LocationInterceptionArgs args )
{
    // Details skipped.
}
```

As you can see in this example, pointcut methods can use the power of LINQ to query `System.Reflection`.

Self Pointcut

The pointcut type `SelfPointcut` simply selects the target of the aspect.

Grouping Advices

The table of above shows advice types grouped in families. Advices of different type but of the same family can be grouped into a single logical *filter*, so they are considered as single transformation.

Why Grouping Advices

Consider for instance three advices of the family `OnMethodBoundaryAspect`: `OnMethodEntryAdvice`, `OnMethodSuccessAdvice` and `OnMethodExceptionAdvice`. The way how they are ordered is important, as it results in a different generation of the `try/catch/finally` block.

The following table compares advice ordering strategies. In the left column, advices are executed in the order: `OnEntry`, `OnExit`, `OnException`. In the right column, advices are grouped together.


```

void Method()
{
  try
  {
    OnEntry();

    try
    {
      // Original method body.
    }
    finally
    {
      OnExit();
    }
  }
  catch
  {
    OnException();
    throw;
  }
}

void Method()
{
  OnEntry();

  try
  {
    // Original method body.
  }
  catch
  {
    OnException();
    throw;
  }
  finally
  {
    OnExit();
  }
}

```

The code in the left column may make sense in some situations, but it is not consistent with the code generated by `OnMethodBoundaryAspect`. Note that the advices may have been ordered differently: the order `OnEntry`, `OnException`, `OnExit` would have generated the same code as in the right column. However, you would have had to use custom attributes to specify order relationships between advices (see [Ordering Advices on page 412](#)). Grouping advices is a much easier way to ensure consistency.

Additionally, when advices of the `OnMethodBoundaryAspect` family are grouped together, it will be possible to share information among them using `MethodExecutionTag`.

The reasons to group advices of the family `LocationInterceptionAspect` and `EventInterceptionAspect` are similar: advices grouped together behave consistently as a single filter (see [\[interception-aspects\]](#)).

How to Group Advices

To group several advices into a single filter:

1. Choose a *master advice*. The choice of the master advice is arbitrary. All other advices of the group are called *slave advices*.
2. Annotate the master advice method with one advice custom attribute (see [Available Advices on page 399](#) and one pointcut custom attribute (see [Available Pointcuts on page 399](#)), as usual.
3. Annotate all slave advices with one advice custom attribute. Set the property `Master` of the custom attribute to the name of the master advice method.
4. Do not specify any pointcut on slave advice methods.

The following code shows how two advices of type `OnMethodEntryAdvice` and `OnMethodExitAdvice` can be grouped into a single filter:

```

[OnMethodEntryAdvice, MulticastPointcut]
public void OnEntry(MethodExecutionArgs args)
{
}

```

```

[OnMethodExitAdvice(Master="OnEntry")]
public void OnExit(MethodExecutionArgs args)
{
}

```

66.2. Introducing Interfaces, Methods, Properties and Events

Some design patterns require you to add properties, methods or interfaces to your target code. If many components in your codebase need to represent the same construct, repetitively adding those constructs flies in the face of the DRY (Don't Repeat Yourself) principle. So how can you add code constructs to your target code without it becoming repetitive?

PostSharp offers a number of ways for you to add different code constructs to your codebase in a controlled and consistent manner. Let's take a look at those techniques.

This topic contains the following sections:

- [Introducing interfaces on page 402](#)
- [Introducing methods on page 403](#)
- [Introducing properties on page 404](#)
- [Controlling the visibility of introduced members on page 406](#)
- [Overriding members or interfaces on page 406](#)

Introducing interfaces

One of the common situations that you will encounter is the need to implement a specific interface on a large number of classes. This may be `INotifyPropertyChanged`, `IDisposable`, `IEquatableT` or some custom interface that you have created. If the implementation of the interface is consistent across all of the targets then there is no reason that we shouldn't centralize its implementation. So how do we go about adding that interface to a class at compile time?

1. Let's add the `IIDentifiable` interface to the target code.

```
publicinterface IIDentifiable
{
    Guid Id { get; }
}
```

2. Create an aspect that inherits from `InstanceLevelAspect` and add the custom attribute `[PSerializableAttribute]`.
3. The key to adding an interface to target code is that you must implement that interface on your aspect. Let's implement the `IIDentifiable` interface on our aspect. It's this implementation of the interface that will be added to the target code, so anything that you include in method or property bodies will be added to the target code as you have declared it in the aspect.

```
[PSerializable]
publicclass IdentifiableAspect : InstanceLevelAspect, IIDentifiable
{
    public Guid Id { get; private set; }
}
```

4. Add the `IntroduceInterfaceAttribute` attribute to the aspect and include the interface type that you want to add to the target code.

```
[IntroduceInterface(typeof(IIDentifiable))]
[PSerializable]
publicclass IdentifiableAspect : InstanceLevelAspect, IIDentifiable
{
    public Guid Id { get; private set; }
}
```

5. Finally you need to declare where this aspect should be applied to the codebase. In this example, let's add it, as an attribute, to a class.

```
[IdentifiableAspect]
public class Customer
{
    public string Name { get; set; }
    public string Address { get; set; }
}
```

6. After compilation, you can decompile the target code and see that the interface has been added to it.

```
public class Customer : IIdentifiable
{
    [System.Diagnostics.DebuggerNonUserCode, System.Runtime.CompilerServices.CompilerGenerated]
    internal sealed class <z__Aspects...
    [System.NonSerialized]
    private IdentifiableAspect <z__aspect0;
    public string Name...
    public string Address...
    public Customer()...
    [System.Runtime.CompilerServices.CompilerGenerated]
    System.Guid IIdentifiable.get_Id()
    {
        return ((IIdentifiable)this.<z__aspect0).Id;
    }
    [System.Runtime.CompilerServices.CompilerGenerated]
    protected virtual void InitializeAspects()...
}
```

As you can see in the decompiled code, interfaces are implemented explicitly on the target code. It is also possible to introduce public members to target code. This is covered below.

NOTE

Interfaces and members introduced by PostSharp are not visible at compile time. To access the dynamically applied interface you must make use of a special PostSharp feature; the `CastTSource, TTarget(TSource)` pseudo-operator. The `CastTSource, TTarget(TSource)` method will allow you to safely cast the target code to the interface type that was dynamically applied. Once that call has been done, you are able to make use of the instance through the interface constructs.

There is no way to access a dynamically-inserted method, property or event, other than through reflection or the dynamic keyword.

NOTE

When you start adding code constructs to your target code, you need to determine how to initialize them correctly. Because these code constructs are not available for you to work with at compile time you need to figure out how to deal with them some other way. To see more about initializing code constructs that you introduce via aspects, please see the section [Initializing Aspects on page 391](#).

Introducing methods

The introduction of methods to your target code is very similar to introducing interfaces. The biggest difference is that you will be introducing code at a much more granular level.

1. Create an aspect that inherits from `InstanceLevelAspect` and add the custom attribute `[SerializableAttribute]`.

2. Add to the aspect the method you want to introduce to the target code.

```
[PSerializable]
publicclass OurCustomAspect : InstanceLevelAspect
{
    publicvoid TheMethodYouWantToUse(string aValue)
    {
        Console.WriteLine("Inside a method that was introduced {0}", aValue);
    }
}
```

NOTE

The method that you declare must be marked as public. If it is not you will see an error at compile time.

3. Decorate the method with the `IntroduceMemberAttribute` attribute.

```
[IntroduceMember]
publicvoid TheMethodYouWantToUse(string aValue)
{
    Console.WriteLine("Inside a method that was introduced {0}", aValue);
}
```

4. Finally, declare where you want this aspect to be applied in the codebase.

```
[OurCustomAspect]
publicclass Customer
{
    publicstring Name { get; set; }
}
```

5. After compilation, you can decompile the target code and see that the method has been added.

```
public class Customer
{
    [DebuggerNonUserCode, CompilerGenerated]
    internal sealed class <>z__Aspects...
    [NonSerialized]
    private OurCustomAspect <>z__aspect0;
    public string Name...
    public Customer()...
    public void TheMethodYouWantToUse(string aValue)
    {
        this.<>z__aspect0.TheMethodYouWantToUse(aValue);
    }
    [CompilerGenerated]
    protected virtual void InitializeAspects()...
}
```

Introducing properties

The introduction of properties is almost exactly the same as the introduction of methods. Like introducing a method you will use the `IntroduceMemberAttribute` attribute. Let's take a look at the details.

1. Create an aspect that inherits from `InstanceLevelAspect` and add the custom attribute `[PSerializableAttribute]`.

2. Add the property you want to introduce to the aspect.

```
[Serializable]
publicclass OurCustomAspect : InstanceLevelAspect
{
    publicstring Name { get; set; }
}
```

NOTE

The property that you declare must be marked as public. If it is not you will see a compiler error.

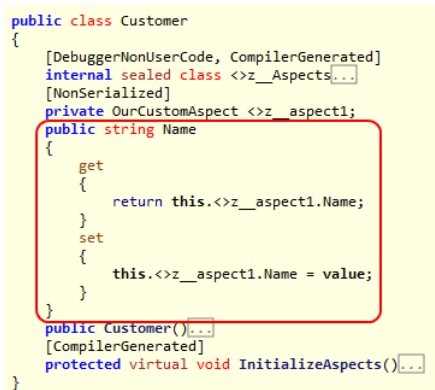
3. Decorate the property with the `IntroduceMemberAttribute` attribute.

```
[IntroduceMember]
publicstring Name { get; set; }
```

4. Add the aspect attribute to the target code where the aspect should be applied.

```
[OurCustomAspect]
publicclass Customer
{
}
```

5. After you have compiled the codebase you can decompile the target code and see that the property has been added.



```
public class Customer
{
    [DebuggerNonUserCode, CompilerGenerated]
    internal sealed class <z__Aspects...
    [NonSerialized]
    private OurCustomAspect <z__aspect1;
    public string Name
    {
        get
        {
            return this.<z__aspect1.Name;
        }
        set
        {
            this.<z__aspect1.Name = value;
        }
    }
    public Customer(...
    [CompilerGenerated]
    protected virtual void InitializeAspects(...
}
```

As noted for both the introduction of methods and properties, the code being introduced must be declared as public. This is needed to ensure that PostSharp can function. If you look closely at the decompiled targets you will see that the introduced members are actually calling the methods/properties that were declared on the aspect. If the method/property on the aspect is not public, the target code will not be able to call it as it should.

NOTE

It is possible to introduce properties to target code, but it is not possible to introduce fields to your target code. The reason is that all members are introduced by delegation: the actual implementation of the member always resides in the aspect.

Controlling the visibility of introduced members

You may not want the introduced member to have public visibility once it has been introduced to the target code. PostSharp allows you to control the visibility of the introduced member through the use of the `Visibility` property on the aspect. To declare that a member should be introduced with private visibility, all you have to do is declare it as such.

```
[IntroduceMember(Visibility = Visibility.Private)]
publicstring Name { get; set; }
```

You have the ability to introduce members with a number of different visibilities including public, private, assembly (internal in C#) and others. You also have the ability to mark an introduction so that it will be declared as virtual if you set the `IsVirtual` property to true.

```
[IntroduceMember(Visibility = Visibility.Private, IsVirtual = true)]
publicstring Name { get; set; }
```

Overriding members or interfaces

One thing you need to be aware of is the situation where you are introducing a member that may already exist in the scope of the target code. Perhaps the method you are trying to introduce is available on the target code through inheritance. It's possible that the method is explicitly declared on the target code as well. The introduction of a member via an aspect needs to take these situations into account. PostSharp allows you to take these situations into account through the use of the `OverrideAction` property.

The `OverrideAction` property allows you to declare a rule for how the introduction of a member or interface should behave if the member or interface is already implemented on the target code. This property allows you to declare rules such as `Fail` (any conflict situation will throw a compile time error), `Ignore` (continue on without trying to introduce the member/interface), `OverrideOrFail` or `OverrideOrIgnore`. It's important to understand how you want to apply your introduced members/interfaces in situations where that member/interface may already exist.

```
[IntroduceMember(OverrideAction = MemberOverrideAction.Fail)]
publicstring Name { get; set; }
```

66.3. Accessing Members of the Target Class

PostSharp makes it possible to import a delegate of a target class method, property or event into the aspect class, so that the aspect can invoke this member.

These mechanisms allow developers to encapsulate more design patterns using aspects.

This topic contains the following sections:

- Importing Members of the Target Class
- Interactions Between Several Member Introductions and Imports

Importing Members of the Target Class

Importing a member into an aspect allows this aspect to invoke the member. An aspect can import methods, properties, or fields.

To import a member of the target type into the aspect class:

1. Define a field into the aspect class, of the following type:

Member Kind	Field Type
Method	A typed <code>Delegate</code> , typically one of the variants of <code>Action</code> or <code>FuncTResult</code> . The delegate signature should exactly match the signature of the imported method.
Property	<code>PropertyTValue</code> , where the generic argument is the type of the property.
Collection Indexer	<code>PropertyTValue</code> , <code>TIndex</code> , where the first generic argument is the type of the property value and the second is the type of the index parameter. Indexers with more than one parameter are not supported.
Event	<code>EventTDelegate</code> , where the generic argument is the type of the event delegate (for instance <code>EventHandler</code>).

2. Make this field public. The field cannot be static.
3. Add the custom attribute `ImportMemberAttribute` to the field. As the constructor argument, pass the name of the member to be imported.

At runtime, the field is set to a delegate of the imported member. Properties and events are imported as set of delegates (`PropertyTValueGet`, `PropertyTValueSet`; `EventTDelegateAdd`, `EventTDelegateRemove`). These delegates can be invoked by the aspect as any delegate.

The property `ImportMemberAttributeIsRequired` determines what happens if the member could not be found in the target class or in its parent. By default, the field will simply have the `null` value if it could not be bound to a member. If the property `IsRequired` is set to `true`, a compile-time error will be emitted.

Interactions Between Several Member Introductions and Imports

Although member introduction and import may seem simple advices at first sight, things become more complex when several advices try to introduce or import the same member. PostSharp handles these situations in a robust and predictable way. For this purpose, it is important to process classes, aspects and advices in a consistent order.

PostSharp enforces the following order:

1. Base classes are processed first, derived classes after. Therefore, when a class is being processed, all parent classes have already been fully processed.
2. Aspects targeting the same class are sorted (see [Coping with Several Aspects on the Same Target on page 409](#)) and executed.
3. Advices of the same aspect are sorted and executed in the following order:
 - a. Member imports which have the property `ImportMemberAttributeOrder` set to `Before-Introductions`.
 - b. Member introductions.
 - c. Members imports which have the property `ImportMemberAttributeOrder` set to `After-Introductions` (this is the default value).

Based on this well-defined order, the advices behave as follow:

Advice	Precondition	Behavior
ImportMemberAttribute	No member, or private member defined in a parent class.	Error if <code>ImportMemberAttributeIsRequired</code> is true, ignored otherwise (by default).
	Non-virtual member defined.	Member imported.
	Virtual member defined.	If <code>ImportMemberAttributeOrder</code> is <code>BeforeIntroductions</code> , the overridden member is imported. This similar to calling a method with the base prefix in C#. Otherwise (and by default), the member is dynamically resolved using the virtual table of the target object.
IntroduceMemberAttribute	No member, or private member defined in a parent class.	Member introduced.
	Non-virtual member defined in a parent class	Ignored if the property <code>IntroduceMemberAttributeOverrideAction</code> is <code>Ignore</code> or <code>OverrideOrIgnore</code> , otherwise fail (by default).
	Virtual member defined in a parent class	Introduce a new override method if the property <code>IntroduceMemberAttributeOverrideAction</code> is <code>OverrideOrFail</code> or <code>OverrideOrIgnore</code> , ignore if the property is <code>Ignore</code> , otherwise fail (by default).
	Member defined in the target class (virtual or not)	Fail by default or if the property <code>IntroduceMemberAttributeOverrideAction</code> is <code>Fail</code> . Otherwise: <ol style="list-style-type: none"> 1. Move the previous method body to a new method so that the previous implementation can be imported by advices <code>ImportMemberAttribute</code> with the property <code>Order</code> set to <code>BeforeIntroductions</code>. 2. Override the method with the imported method.

66.4. Adding Aspects Dynamically

Additionally to providing advices, an aspect can provide other aspects dynamically using `IAAspectProvider`. This allows aspect developers to address situations where it is not possible to add aspects declaratively (using custom attributes) to the source code; aspects can be provided on the basis of a complex analysis of the target assembly using `System.Reflection`, or by reading an XML file, for instance.

For details about `IAAspectProvider`, see [Adding Aspects Programmatically using `IAAspectProvider` on page 131](#).

CHAPTER 67

Coping with Several Aspects on the Same Target

As the team learns aspect-oriented programming and starts adding more aspect to projects, chances raise that several aspects are added to the same element of code. This could be a major source of troubles if PostSharp did not provide a robust framework to detect and prevent conflicts between aspects:

- Most aspects need to **be ordered**. For instance, an authorization aspect must be executed *before* a caching aspect.
- Even if some aspects don't care to be ordered, it's good to have them applied in **predictable order**. Otherwise, some code that works today may be broken tomorrow -- just because aspects were applied in a different order.
- Some aspects **conflict**; they cannot be together on the same aspect, or not in a given order. For instance, it does not make sense to persist an object using two different aspects: one would persist to the database, the other to the registry.
- Some aspects **require** other aspects to be applied. For instance, an aspect changing the mouse pointer to an hourglass requires the method to execute asynchronously, otherwise the pointer shape will never be updated.

PostSharp addresses these issues by making it possible to add dependencies between aspects. The aspect dependency framework is implemented in the namespace `PostSharp.Aspects.Dependencies`.

NOTE

The aspect dependency framework is not related to the notion of dependency injection.

Aspect Dependency Custom Attributes

You can express dependencies of an aspect by annotating the aspect class with custom attributes derived from the type `AspectDependencyAttribute`. Several derived types are available; every type matches other aspects according to different criteria.

Attribute Type	Description
<code>AspectTypeDependencyAttribute</code>	This custom attribute expresses a dependency with a well-known aspect class.
<code>AspectRoleDependencyAttribute</code>	This custom attribute expresses a dependency with any aspect classes enrolled in a given role. Its dual is <code>ProvideAspectRoleAttribute</code> : this custom attribute enrolls an aspect class into a role. A role is simply a string. Whenever possible, consider using one of the roles defined in the class <code>StandardRoles</code> .

Attribute Type	Description
AspectEffectDependencyAttribute	This custom attribute expresses a dependency with any aspect that has a specific effect on the source code or the control flow. Effects are represented as a string, whose valid values are listed in the type <code>StandardEffects</code> . Effects are provisioned by the aspect weaver on the basis of a rough analysis of what the aspect may do; aspect developers cannot assign new effects to aspects. However, they can waive effects by using the custom attribute <code>WaiveAspectEffectAttribute</code> . For instance, an aspect developer can specify that a trace attribute has no effect at all; this aspect will commute with any other aspect (see below).

All these custom attributes have similar structure and members. The first parameter of their constructor, of type `AspectDependencyAction`, determines the kind of dependency relationship added between the current aspect and the aspects matched by the custom attribute.

PostSharp supports the following kinds of relationships:

Action	Description
Order	The dependency expresses an order relationship. The second constructor of the custom attribute, of type <code>AspectDependencyPosition</code> (with values <code>Before</code> or <code>After</code>), must be specified. The custom attributes determine the position of the current aspect with respect to matched aspects.
Require	The dependency expresses a requirement. PostSharp will issue a compile-time error if the requirement is not satisfied for any target of the current aspect. The second constructor of the custom attribute, of type <code>AspectDependencyPosition</code> , is optional. If specified, an aspect matching the dependency should be present before or after the current aspect.
Conflict	The dependency expresses a conflict. PostSharp will issue a compile-time error if any aspect matching the dependency rule is present on any target of the current aspect. The second constructor of the custom attribute, of type <code>AspectDependencyPosition</code> , is optional. If specified, an error is issued only if a matching aspect is present before or after the current aspect.
Commute	The dependency specifies that the current aspect is commutable with any matching aspect. When aspects are commutable, PostSharp does not issue any warning if they are not strongly ordered.

Custom attribute types and values of the enumeration `AspectDependencyAction` are orthogonal; they can be freely combined.

Examples

Using role-based dependencies

The following code shows how three aspects can be ordered without having explicit knowledge of each other. Each aspect provides a different role, and defines dependencies with respect to other roles.

```
[ProvideAspectRole( StandardRoles.Threading )]
[AspectRoleDependency(AspectDependencyAction.Order, AspectDependencyPosition.Before, "UI")]
publicsealedclass BackgroundAttribute : MethodInterceptionAspect
{
    // Details skipped
}

[ProvideAspectRole( StandardRoles.ExceptionHandling )]
[AspectRoleDependency( AspectDependencyAction.Order, AspectDependencyPosition.After, StandardRoles.Threading )]
[AspectRoleDependency(AspectDependencyAction.Order, AspectDependencyPosition.After, "UI")]
publicsealedclass ExceptionDialogAttribute : OnExceptionAspect
{
    // Details skipped
}
```

```
[ProvideAspectRole("UI")]
publicsealedclass StatusTextAttribute : OnMethodBoundaryAspect
{
    // Details skipped
}
```

Using effect-based dependencies

The following code shows how to protect an authorization aspect to be executed after an aspect which may change the control flow and skip the execution of the method, such as a caching aspect. Then, it shows how the aspect `BackgroundAttribute` can opt out from this effect, because the aspect developer knows that does aspect does not skip the execution of the method, but only defers it.

```
[AspectEffectDependency( AspectDependencyAction.Conflict, AspectDependencyPosition.Before,
                        StandardEffects.ChangeControlFlow )]
publicsealedclass AuthorizationAttribute : OnMethodBoundaryAspect
{
    // Details skipped.
}

[WaiveAspectEffect(StandardEffects.ChangeControlFlow)]
publicsealedclass BackgroundAttribute : MethodInterceptionAspect
{
    // Details skipped
}
```

Deferring Ordering to Aspect Users

By adding dependencies to the aspect class, the aspect developer specifies the order of execution of aspects in a fully static way. The same order is used for every element of code to which aspects apply. While this behavior is most of the time desirable, there may be situations where we want to defer ordering to users of our aspects.

Aspect users can influence the order of execution of an aspect by setting the aspect property `AspectPriority`, typically when using the aspect custom attribute (the same property is available in the configuration object as `Aspect-ConfigurationAspectPriority`, see [Aspect Configuration on page 415](#)).

Setting the `AspectPriority` results to an aspect in adding an ordering dependency between this aspect and all other aspects where the same property has been set. Therefore, aspect priorities complement, and do not replace, other ordering dependencies. The aspect developer may specify vital aspect dependencies (that is, under-specify aspect ordering), and let it to the aspect user to complete the ordering with priorities.

CAUTION NOTE

Do not confuse the property `AspectPriority` with `AttributePriority`. The latter determines an order in which several custom attributes of the same type are processed by the `MulticastAttribute` engine. The first determines in which order the aspects are executed at run time.

Adding Dependencies to Third-Party Aspects

If you are using aspects provided by several third-party vendors who don't know about each other, you may need to solve conflicts on your own.

You can do that by adding any custom attribute derived from `AspectDependencyAttribute` at assembly level, and use the property `TargetType` to specify to which aspect class the dependency applies.

Here is an example:

```
[assembly: AspectTypeDependency( AspectDependencyAction.Order, AspectDependencyPosition.Before,
                                typeof(Vendor1.TraceAspect), TargetType = typeof(Vendor2.ExceptionHandlingAspect) )]
```

67.1. Ordering Advices

The section [Coping with Several Aspects on the Same Target on page 409](#) talks in terms of *aspect dependencies* and *aspect ordering*. Most of what has been said there is also valid to advices. When we talk of the order of execution of aspects, we actually mean the execution of advices ("aspects" themselves, *stricto sensu*, are never executed).

Dependencies defined at aspect level implicitly apply to all advices. When developing a composite aspect (see [Developing Composite Aspects on page 397](#)), it is possible to add dependencies directly to advice methods by annotating them with custom attributes of the namespace `PostSharp.Aspects.Dependencies`.

Note that all advices provided by an aspect are ordered in a single block. Suppose that a method is the target of advices `Aspect1.MethodA`, `Aspect1.MethodB` and `Aspect2.MethodC`. The next table shows valid and invalid orders:

Valid Orders	Invalid Orders
<code>Aspect1.MethodA</code> , <code>Aspect1.MethodB</code> , <code>Aspect2.MethodC</code>	<code>Aspect1.MethodA</code> , <code>Aspect2.MethodC</code> , <code>Aspect1.MethodB</code>
<code>Aspect1.MethodB</code> , <code>Aspect1.MethodA</code> , <code>Aspect2.MethodC</code>	<code>Aspect1.MethodB</code> , <code>Aspect2.MethodC</code> , <code>Aspect1.MethodA</code>
<code>Aspect2.MethodC</code> , <code>Aspect1.MethodA</code> , <code>Aspect1.MethodB</code>	
<code>Aspect2.MethodC</code> , <code>Aspect1.MethodB</code> , <code>Aspect1.MethodA</code>	

Ordering Advices of the Same Aspect

Advices of the same aspect can be used using any custom attribute derived from `AspectDependencyAttribute`.

Because advices of the same aspect instance are necessarily ordered in block, it is appropriate to specify dependencies between aspect classes extensively, and specify ordering of advices only in the scope of the current aspect instance. The most appropriate dependency custom attribute for this purpose is `AdviceDependencyAttribute`, which accepts the name of the advice method as a parameter.

CHAPTER 68

Understanding Aspect Serialization

As explained in section [Understanding Aspect Lifetime and Scope](#) on page 389, aspects are first instantiated at build time by the weaver, are then initialized by the `CompileTimeInitialize` method, and serialized and stored in the assembly as a managed resource. Aspects are then deserialized at run time, before being executed.

Because of the aspect life cycle, aspect classes must be made serializable as described in this section.

This topic contains the following sections:

- [Default serialization strategy](#) on page 413
- [Fallback serialization strategy](#) on page 413
- [Aspects without serialization](#) on page 414

Default serialization strategy

Typically, aspects can be made serializable by adding a custom attribute to the class, which causes all fields of the class to be serialized. Fields that do not need to be serialized must be annotated with an opt-out custom attribute. PostSharp chooses the serialization strategy according to these custom attributes. The serialization strategy is implemented in classes derived from the abstract `AspectSerializer` class. The default serialization strategy is implemented in the `PortableAspectSerializer` class, that is backed by `PortableFormatter`.

This is how you can apply default serialization strategy to your aspect:

- To make the class serializable, annotate the class with the `[PSerializableAttribute]` custom attribute.
- To exclude the field from the serialization, annotate the field with the `[PNonSerializedAttribute]` custom attribute.

Fallback serialization strategy

In some cases, the default serialization strategy implemented by the `PortableAspectSerializer` class may not be appropriate for your aspects. For example, the data structures used in your classes may not be supported by the `PortableFormatter` implementation or you may need your code to be backward compatible with PostSharp 4.2 and earlier. In versions 4.2 and earlier the default serialization strategy was implemented in the `BinaryAspectSerializer` class, that was backed by `BinaryFormatter`. You can still use `BinaryAspectSerializer` as a fallback serialization strategy in PostSharp 4.3 and later.

To apply fallback serialization strategy to your aspects, use `[SerializableAttribute]` custom attribute instead of `[PSerializableAttribute]`, and use `[NonSerializedAttribute]` custom attribute instead of `[PNonSerializedAttribute]`.

NOTE

The `BinaryAspectSerializer` class is supported only in projects that target the .NET Framework with full trust.

Aspects without serialization

In some situations, serializing and deserializing the aspect may be a suboptimal solution. In case aspect field values are a pure function of constructor arguments and properties, it may be more efficient to emit code that instantiates these aspects at run time instead of serializing-deserializing them. This is the case, typically, if the aspect does not implement the `CompileTimeInitialize` method.

In this situation, it is better to use a different serializer: `MsilAspectSerializer`.

NOTE

`MsilAspectSerializer` is actually **not** a serializer. When you use this implementation instead of a real serializer, the aspect is **not** serialized, but the weaver generates MSIL instructions to build the aspect instance at run time, by calling the aspect class constructor and by setting its fields and properties.

You can specify which serializer should be used for a specific aspect class by setting the property `Aspect-ConfigurationSerializerType` of the configuration of this aspect class or instance.

See section [Aspect Configuration](#) on page 415 for details.

The following code shows how to choose the serializer type for an `OnMethodBoundaryAspect`:

```
[OnMethodBoundaryAspectConfiguration(SerializerType=typeof(MsilAspectSerializer))]  
publicsealed MyAspect : OnMethodBoundaryAspect
```

CHAPTER 69

Aspect Configuration

Configuration settings of aspects determine how they should be processed by their weaver. Configuration settings are always evaluated at build time. Most aspects have one or many of them. For instance, the aspect type `OnExceptionAspect` has a configuration setting determining the type of exceptions handled with this aspect.

There are two ways to configure an aspect: declarative and imperative.

Declarative Configuration

You can configure an aspect declaratively by applying the appropriate custom attribute on the aspect class. Aspect configuration attributes are in the namespace `PostSharp.Aspects.Configuration`. Every aspect type has its corresponding type of configuration attribute. The name of the custom attribute starts with the name of the aspect and has the suffix `ConfigurationAttribute`. For instance, the configuration attribute of the aspect class `OnExceptionAspect` is `OnExceptionAspectConfigurationAttribute`.

Declarative configuration has always precedence over imperative configuration: if some property of the configuration custom attribute is set on the aspect class, or on any parent, the corresponding imperative semantic will not be evaluated.

Once a configuration property has been set in a parent class, it cannot be overwritten in a child class.

Note that these restrictions are enforced at the level of properties. If a property of a configuration custom attribute is not set in a parent class, it can still be overwritten in a child class or by an imperative semantic.

Imperative Configuration

A second way to configure an aspect class is to override its configuration methods or set its configuration property.

NOTE

Imperative configuration is only available when you target the full .NET Framework. It is not available for Silverlight or the Compact Framework.

Benefits of Imperative Configuration

The advantage of imperative configuration is that it can be arbitrarily complex (since the code of the configuration method is executed inside the weaver). Specifically, it allows the configuration to be dependent on how the aspect is actually used, for instance the configuration can depend on the value of a property of the aspect custom attribute.

Implementation Note

Under the hood, aspects implement the method `IAAspectBuildSemanticsGetAspectConfiguration(Object)`. This method should return a configuration object, derived from the class `AspectConfiguration`. Every aspect class has its own aspect configuration class. For instance, the configuration attribute of the aspect class `OnExceptionAspect` is `OnExceptionAspectConfiguration`. The aspect type `OnExceptionAspect` implements `IAAspectBuildSemanticsGetAspectConfiguration(Object)` by creating an instance of `OnExceptionAspectConfiguration`, then it invokes the method `OnExceptionAspectGetExceptionType(MethodBase)` and copies the return value of this method to the property `OnExceptionAspectConfigurationExceptionType`. Therefore, there are two ways to configure an aspect:

either by overriding configuration methods and setting configuration properties (these methods and properties are provided by the framework for convenience only), or by implementing the method `IAAspectBuildSemanticsGetAspect-Configuration(Object)`. If your aspect does not derive from the aspect class `OnExceptionAspect`, but directly implements the aspect interface `IOnExceptionAspect`, you can use only the later method.

CHAPTER 70

Customizing Aspect Appearance in Visual Studio

This chapter explains how to configure how your custom aspects appear in PostSharp Tools for Visual Studio. It contains the following topics:

Section	Description
Customizing Aspect Description in Tooltips on page 417.	This topic describes how to change the description of custom aspects in PostSharp-generated tooltips in Visual Studio.
Estimating Code Savings on page 418	This topic explains how to give hints so that PostSharp can better estimate how many lines of code are saved thanks to your aspect.

70.1. Customizing Aspect Description in Tooltips

When you position the mouse cursor over a declaration that has been enhanced by an aspect, PostSharp Tools adds a description of the aspect to the Intellisense tooltip. The description that PostSharp generates by default is sometimes little helpful. To make the Intellisense description of your aspect more understandable for its users, you should override the default description.

Simple Aspects

Simple aspects are aspects built by deriving from a base class such as `OnMethodBoundaryAspect` and overriding virtual methods of the base class, such as `OnEntry(MethodExecutionArgs)`. They are described in the section [Developing Simple Aspects on page 351](#).

To set the description of a simple aspect, add the `AspectDescriptionAttribute` custom attribute to the aspect class.

This is illustrated in the following code snippet.

```
[PSerializable]
[AspectDescription("Applies the exception handling policy")]
publicsealedclass ExceptionHandlerAttribute : OnExceptionAspect
{
    publicoverridevoid OnException( MethodExecutionArgs eventArgs )
    {
        if ( !ExceptionHandler.OnException( eventArgs.Exception ) )
        {
            eventArgs.FlowBehavior = FlowBehavior.Continue;
        }
    }
}
```

Composite Aspects

Composite aspects are aspects where advices are not overridden from the base class, but are added using advice and pointcut custom attributes such as `OnMethodEntryAdvice` and `MethodPointcut`. Composite aspects are described in section [Developing Composite Aspects on page 397](#). Unlike simple aspects, composite aspects can have several advices,

With composite aspects, you should add a description to every advice. You can do that by setting the `Description` property of the advice custom attribute.

The following code snippet illustrates how to set the description of the advice. This description will appear in the Intellisense tooltip of each property affected by this advice.

```
[OnLocationSetValueAdvice( Description="Persists the property to disk." ),
MulticastPointcut( Targets = MulticastTargets.Property,
                   Attributes = MulticastAttributes.Instance | MulticastAttributes.NonAbstract)]
publicvoid OnPropertySet( LocationInterceptionArgs args )
{
    // Details skipped.
}
```

70.2. Estimating Code Savings

During build, PostSharp attempts to estimate how many lines of handwritten code were avoided thanks to aspects. By default, PostSharp considers that 2 lines of code are saved every time an advice is applied to a target. This is of course a very rough estimate. You can add information to your aspects and advices to make the estimate more accurate.

TIP

When adding code saving estimate, ask yourself the following question: how much code would an intelligent developer have written if she has to implement the same feature without PostSharp, using the best possible strategy? Do not assume that the strategy you took to implement the feature with an aspect would be the same as the strategy for handwritten code.

This topic contains the following sections:

- [Simple aspects on page 418](#)
- [Composite aspects on page 419](#)
- [Adding code saving hints programmatically on page 419](#)

Simple aspects

Simple aspects are aspects built by deriving from a base class such as `OnMethodBoundaryAspect` and overriding virtual methods of the base class, such as `OnEntry(MethodExecutionArgs)`. They are described in the section [Developing Simple Aspects on page 351](#).

By default, PostSharp estimates that 2 lines of handwritten code are avoided for each advice method that you override, every time the aspect is applied to a target.

To override the default value, add the `LinesOfCodeAvoidedAttribute` custom attribute to the aspect class. The argument of the custom attribute constructor must be set to the number of lines of handwritten code avoided every time the aspect is applied to a target.

The following code snippet shows how to specify that 4 lines of code are avoided every time the aspect is applied. If the aspect is applied to 100 methods, PostSharp will estimate that 400 lines of handwritten code have been avoided.

```
[PSerializable]
[LinesOfCodeAvoided(4)]
publicsealedclass ExceptionHandlerAttribute : OnExceptionAspect
{
    publicoverridevoid OnException( MethodExecutionArgs eventArgs )
    {
        if ( !ExceptionHandler.OnException( eventArgs.Exception ) )
        {
            eventArgs.FlowBehavior = FlowBehavior.Continue;
        }
    }
}
```

Composite aspects

Composite aspects are aspects where advices are not overridden from the base class, but are added using advice and pointcut custom attributes such as `OnMethodEntryAdvice` and `MethodPointcut`. Composite aspects are described in section [Developing Composite Aspects on page 397](#). Unlike simple aspects, composite aspects can have several advices,

By default, PostSharp estimates that 2 lines of handwritten code are avoided for each advice, every time the advice is applied to a target. Some advices may have different default values. For instance, the `IntroduceInterfaceAttribute` advice shall count 2 lines of code per introduced interface method.

You can still use the aspect-level `LinesOfCodeAvoidedAttribute` custom attribute. It will increment the estimated number of avoided lines of code every time the *aspect* is applied to a target. However, to provide more relevant estimates, you need to provide code saving information at *advice* level.

To specify how many lines of handwritten code are avoided every time an advice is applied to a target, specify the `Advice` property of the advice custom attribute.

The following code snippet shows how to specify that 1 line of code is avoided every time the advice is applied. Suppose that the aspect is applied to 100 classes and each class has in average 5 instance non-abstract properties. In this situation, PostSharp will estimate that 500 lines of handwritten code have been avoided.

```
[OnLocationSetValueAdvice( LinesOfCodeAvoided = 1 ),
MulticastPointcut( Targets = MulticastTargets.Property,
                  Attributes = MulticastAttributes.Instance | MulticastAttributes.NonAbstract)]
publicvoid OnPropertySet( LocationInterceptionArgs args )
{
    // Details skipped.
}
```

Adding code saving hints programmatically

In the previous sections, we described how to add code saving hints declaratively using custom attributes. Sometimes declarative estimations are not accurate enough. To learn how to add programmatic hints, see [Pushing Information to PostSharp Tools Programmatically on page 419](#).

70.3. Pushing Information to PostSharp Tools Programmatically

The `IWeavingSymbolsService` service allows you to push information from your aspect, at build time, to PostSharp Tools for Visual Studio.

This service can be used in the following scenarios:

- Adding some text to the Intellisense tooltip of a declaration.

- Adding some code saving information.
- Add some annotation that means that PostSharp Tools should consider that a declaration has been decorated with a custom attribute. This annotation is then taken into account by the analytic engine that powers the real-time quick actions and diagnostics of PostSharp Tools. For instance, the `FieldRule` facility uses this feature.

To get an instance of this service, use the `GetServiceT(Boolean)` method from `PostSharpEnvironment.CurrentProject.GetService`.

CHAPTER 71

Consuming Dependencies from an Aspect

Aspects, as other components, may have dependencies to other application services. Aspects may be bound to the abstract interface to this service, and may need to resolve the dependency at run time.

However, two reasons prevent us from the following approaches that are usual with dependency injection containers:

- Aspects are instantiated at build time, and dependency-injection containers only exist at run-time.
- Aspects typically have a static scope. Unless they implement the `IInstanceScopedAspect`, aspect instances are stored in static fields, even when applied to instance members.

These characteristics are not an obstacle to using service containers, but different patterns must be followed.

This section presents several ways to consume dependencies from an aspect:

- [Using a Global Composition Container on page 421](#)
- [Using a Global Service Locator on page 424](#)
- [Using Dynamic Dependency Resolution on page 426](#)
- [Using Contextual Dependency Resolution on page 429](#)
- [Importing Dependencies from the Target Object on page 432](#)

71.1. Using a Global Composition Container

Although the aspect cannot be instantiated by the dependency injection container, it is possible to initialize the aspect from an *ambient container* at run time. An ambient container is one that is exposed as a static member and that is global to the whole application.

Dependency injection containers typically offer methods to initialize objects that have been instantiated externally. For instance, the Managed Extensibility Framework offers the `SatisfyImportsOnce(ComposablePart)` method.

The dependency injection method can be invoked from the `RuntimeInitialize(MethodBase)` method.

NOTE

User code has no control over the time when and the thread on which an aspect is initialized. Therefore, using `ThreadStaticAttribute` to make the container local to the current thread is not a reliable approach.

IMPORTANT NOTE

The service container must be initialized before the execution of any class that is enhanced by the aspect. It means that it is not possible to use the aspect on test classes themselves. To relax this constraint, it is possible to initialize the dependency lazily, when the first advice is hit.

Example: testable logging aspect with a global MEF service container

The following code snippet shows a logging aspect and how it could be used in production code:

```
using System;
using System.ComponentModel.Composition;
using System.ComponentModel.Composition.Hosting;
using System.ComponentModel.Composition.Primitives;
using System.Reflection;
using PostSharp.Aspects;
using PostSharp.Extensibility;
using PostSharp.Serialization;

namespace DependencyResolution.GlobalServiceContainer
{
    publicinterface ILogger
    {
        void Log(string message);
    }

    publicstaticclass AspectServiceInjector
    {
        privatestatic CompositionContainer container;

        publicstaticvoid Initialize(ComposablePartCatalog catalog)
        {
            container = new CompositionContainer(catalog);
        }

        publicstaticvoid BuildObject(object o)
        {
            if (container == null)
                thrownew InvalidOperationException();

            container.SatisfyImportsOnce(o);
        }
    }

    [PSerializable]
    publicclass LogAspect : OnMethodBoundaryAspect
    {
        [Import] private ILogger logger;

        publicoverridevoid RuntimeInitialize(MethodBase method)
        {
            AspectServiceInjector.BuildObject(this);
        }

        publicoverridevoid OnEntry(MethodExecutionArgs args)
        {
            logger.Log("OnEntry");
        }
    }

    internalclass Program
    {
        privatestaticvoid Main(string[] args)
        {
            AspectServiceInjector.Initialize(new TypeCatalog(typeof (ConsoleLogger)));

            // The static constructor of LogAspect is called before the static constructor of the type// containing target
        }
    }
}
```

```

        Foo.LoggedMethod();
    }
}

internalclass Foo
{
    [LogAspect]
    publicstaticvoid LoggedMethod()
    {
        Console.WriteLine("Hello, world.");
    }
}

[Export(typeof (ILogger))]
internalclass ConsoleLogger : ILogger
{
    publicvoid Log(string message)
    {
        Console.WriteLine(message);
    }
}
}

```

The following code snippet shows how the logging aspect can be tested:

```

using System;
using System.ComponentModel.Composition;
using System.ComponentModel.Composition.Hosting;
using System.Text;
using Microsoft.VisualStudio.TestTools.UnitTesting;

namespace DependencyResolution.GlobalServiceContainer.Test
{
    [TestClass]
    publicclass TestLogAspect
    {
        static TestLogAspect()
        {
            AspectServiceInjector.Initialize(new TypeCatalog(typeof (TestLogger)));
        }

        [TestMethod]
        publicvoid TestMethod()
        {
            TestLogger.Clear();
            new TargetClass().TargetMethod();
            Assert.AreEqual("OnEntry" + Environment.NewLine, TestLogger.GetLog());
        }

        privateclass TargetClass
        {
            [LogAspect]
            publicvoid TargetMethod()
            {
            }
        }
    }

    [Export(typeof (ILogger))]
    internalclass TestLogger : ILogger
    {
        publicstaticreadonly StringBuilder stringBuilder = new StringBuilder();

        publicvoid Log(string message)
        {
            stringBuilder.AppendLine(message);
        }

        publicstaticstring GetLog()
        {
            return stringBuilder.ToString();
        }
    }
}

```

```

    }

    publicstaticvoid Clear()
    {
        stringBuilder.Clear();
    }
}
}
}

```

71.2. Using a Global Service Locator

If all aspect instances are using the same global dependency injection container, it is likely that dependencies of all instances will resolve to the same service implementation. Therefore, storing dependencies in an instance field may be a waste of memory, especially for aspects that are applied to a very high number of code elements.

Alternatively, dependencies can be stored in static fields and initialized in the aspect static constructor.

TIP

Use the `PostSharpEnvironment.IsPostSharpRunning` property to make sure that this part of the static constructor is executed at run time only, when PostSharp is *not* running.

In this case, dependency injection method such as `SatisfyImportsOnce(ComposablePart)` cannot be used. Instead, the container must be used as a service locator. For instance, MEF exposes the method `ExportProvider.GetExport`.

IMPORTANT NOTE

The service locator must be initialized before the execution of any class that is enhanced by the aspect. It means that it is not possible to use the aspect on the entry-point class (Program or App, typically). To relax this constraint, it is possible to initialize the dependency on demand, for instance using the LazyT construct.

Example: testable aspect with a global MEF service locator

The following code snippet shows a logging aspect and how it could be used in production code:

```

using System;
using System.ComponentModel.Composition;
using System.ComponentModel.Composition.Hosting;
using System.ComponentModel.Composition.Primitives;
using PostSharp.Aspects;
using PostSharp.Extensibility;
using PostSharp.Serialization;

namespace DependencyResolution.GlobalServiceLocator
{
    publicinterface ILogger
    {
        void Log(string message);
    }

    publicstaticclass AspectServiceLocator
    {
        privatestatic CompositionContainer container;

        publicstaticvoid Initialize(ComposablePartCatalog catalog)
        {

```



```

        container = new CompositionContainer(catalog);
    }

    public static Lazy<T> GetService<T>() where T : class
    {
        return new Lazy<T>(GetServiceImpl<T>);
    }

    private static T GetServiceImpl<T>()
    {
        if (container == null)
            throw new InvalidOperationException();

        return container.GetExport<T>().Value;
    }
}

[PSerializable]
public class LogAspect : OnMethodBoundaryAspect
{
    private static readonly Lazy<ILogger> logger;

    static LogAspect()
    {
        if (!PostSharpEnvironment.IsPostSharpRunning)
        {
            logger = AspectServiceLocator.GetService<ILogger>();
        }
    }

    public override void OnEntry(MethodExecutionArgs args)
    {
        logger.Value.Log("OnEntry");
    }
}

internal class Program
{
    private static void Main(string[] args)
    {
        AspectServiceLocator.Initialize(new TypeCatalog(typeof (ConsoleLogger)));

        LoggedMethod();
    }

    [LogAspect]
    public static void LoggedMethod()
    {
        Console.WriteLine("Hello, world.");
    }
}

[Export(typeof (ILogger))]
internal class ConsoleLogger : ILogger
{
    public void Log(string message)
    {
        Console.WriteLine(message);
    }
}
}

```

The following code snippet shows how the logging aspect can be tested:

```

using System;
using System.ComponentModel.Composition;
using System.ComponentModel.Composition.Hosting;
using System.Text;
using Microsoft.VisualStudio.TestTools.UnitTesting;

```

```

namespace DependencyResolution.GlobalServiceLocator.Test
{
    [TestClass]
    publicclass TestLogAspect
    {
        static TestLogAspect()
        {
            AspectServiceLocator.Initialize(new TypeCatalog(typeof (TestLogger)));
        }

        [TestMethod]
        publicvoid TestMethod()
        {
            TestLogger.Clear();
            TargetMethod();
            Assert.AreEqual("OnEntry" + Environment.NewLine, TestLogger.GetLog());
        }

        [LogAspect]
        privatevoid TargetMethod()
        {
        }
    }

    [Export(typeof (ILogger))]
    internalclass TestLogger : ILogger
    {
        publicstaticreadonly StringBuilder stringBuilder = new StringBuilder();

        publicvoid Log(string message)
        {
            stringBuilder.AppendLine(message);
        }

        publicstaticstring GetLog()
        {
            return stringBuilder.ToString();
        }

        publicstaticvoid Clear()
        {
            stringBuilder.Clear();
        }
    }
}

```

71.3. Using Dynamic Dependency Resolution

Both previous approaches have a static dependency resolution strategy: it cannot be changed over time. Therefore, these strategies could be unsuitable in cases where several tests need different configurations of the dependency container.

A possible solution is to resolve dependencies dynamically each time they are needed, and not only at aspect initialization. Although this solution is ideal for the sake of testing, it may be too inefficient for production. Therefore, the solution would still need to provide dependency caching for production mode. Caching would neutralize the dynamic characteristics of dependency resolution.

This solution would be based on the following elements:

1. The service locator can be initialized in two modes: production (the resolution strategy is immutable) and testing (the resolution strategy can be modified).

2. The service locator returns a delegate (Func<T>, where *T* is the dependency type), instead of the dependency itself (T or Lazy<T>).
3. The aspect calls the service locator during aspect initialization and stores the delegate.
4. The aspect calls the delegate at run time.

Example: testable logging aspect with a global MEF service container with dynamic resolution

The following code snippet shows a logging aspect and how it could be used in production code:

```
using System;
using System.ComponentModel.Composition;
using System.ComponentModel.Composition.Hosting;
using System.ComponentModel.Composition.Primitives;
using PostSharp.Aspects;
using PostSharp.Extensibility;
using PostSharp.Serialization;

namespace DependencyResolution.Dynamic
{
    publicinterface ILogger
    {
        void Log(string message);
    }

    publicstaticclass AspectServiceLocator
    {
        privatestatic CompositionContainer container;
        privatestaticbool isCacheable;

        publicstaticvoid Initialize(ComposablePartCatalog catalog, bool isCacheable)
        {
            if (AspectServiceLocator.isCacheable && container != null)
                thrownew InvalidOperationException();

            container = new CompositionContainer(catalog);
            AspectServiceLocator.isCacheable = isCacheable;
        }

        publicstatic Func<T> GetService<T>() where T : class
        {
            if (isCacheable)
            {
                return () => new Lazy<T>(GetServiceImpl<T>).Value;
            }
            else
            {
                return GetServiceImpl<T>;
            }
        }

        privatestatic T GetServiceImpl<T>()
        {
            if (container == null)
                thrownew InvalidOperationException();

            return container.GetExport<T>().Value;
        }
    }

    [PSerializable]
    publicclass LogAspect : OnMethodBoundaryAspect
    {
        privatestaticreadonly Func<ILogger> logger;

        static LogAspect()
        {
            if (!PostSharpEnvironment.IsPostSharpRunning)

```

Consuming Dependencies from an Aspect

```
        {
            logger = AspectServiceLocator.GetService<ILogger>();
        }
    }

    public override void OnEntry(MethodExecutionArgs args)
    {
        logger().Log("OnEntry");
    }
}

internal class Program
{
    private static void Main(string[] args)
    {
        AspectServiceLocator.Initialize(new TypeCatalog(typeof (ConsoleLogger)), true);

        LoggedMethod();
    }

    [LogAspect]
    public static void LoggedMethod()
    {
        Console.WriteLine("Hello, world.");
    }
}

[Export(typeof (ILogger))]
internal class ConsoleLogger : ILogger
{
    public void Log(string message)
    {
        Console.WriteLine(message);
    }
}
}
```

The following code snippet shows how the logging aspect can be tested:

```
using System;
using System.ComponentModel.Composition;
using System.ComponentModel.Composition.Hosting;
using System.Text;
using Microsoft.VisualStudio.TestTools.UnitTesting;

namespace DependencyResolution.Dynamic.Test
{
    [TestClass]
    public class TestLogAspect
    {
        [TestMethod]
        public void TestMethod()
        {
            // The ServiceLocator can be initialized for each test.
            AspectServiceLocator.Initialize(new TypeCatalog(typeof (TestLogger)), false);

            TestLogger.Clear();
            TargetMethod();
            Assert.AreEqual("OnEntry" + Environment.NewLine, TestLogger.GetLog());
        }

        [LogAspect]
        private void TargetMethod()
        {
        }
    }

    [Export(typeof (ILogger))]
}
```

```

internalclass TestLogger : ILogger
{
    publicstaticreadonly StringBuilder stringBuilder = new StringBuilder();

    publicvoid Log(string message)
    {
        stringBuilder.AppendLine(message);
    }

    publicstaticstring GetLog()
    {
        return stringBuilder.ToString();
    }

    publicstaticvoid Clear()
    {
        stringBuilder.Clear();
    }
}
}

```

71.4. Using Contextual Dependency Resolution

The dependency resolution strategy does not necessarily need to resolve to the same service implementation for all occurrences of the dependency. It is possible to design a strategy that depends on the context. For instance, the service locator could accept the aspect type and the target element of code as parameters. Test code could configure the service locator to resolve dependencies to specific implementations for a given context.

Evaluating context-sensitive rules may be CPU-intensive, but it needs to be done only during testing. In production mode, dependency resolution can be delegated to a global service catalog.

Example: testable logging aspect with contextual dependency resolution

The following code snippet shows a logging aspect and how it could be used in production code:

```

using System;
using System.Collections.Generic;
using System.ComponentModel.Composition;
using System.ComponentModel.Composition.Hosting;
using System.ComponentModel.Composition.Primitives;
using System.Reflection;
using PostSharp.Aspects;
using PostSharp.Extensibility;
using PostSharp.Serialization;

namespace DependencyResolution.Contextual
{
    publicinterface ILogger
    {
        void Log(string message);
    }

    publicstaticclass AspectServiceLocator
    {
        privatestatic CompositionContainer container;
        privatestatic HashSet<object> rules = new HashSet<object>();

        publicstaticvoid Initialize(ComposablePartCatalog catalog)
        {
            container = new CompositionContainer(catalog);
        }

        publicstatic Lazy<T> GetService<T>(Type aspectType, MemberInfo targetElement) where T : class
        {

```

Consuming Dependencies from an Aspect

```
        return new Lazy<T>(() => GetServiceImpl<T>(aspectType, targetElement));
    }

    private static T GetServiceImpl<T>(Type aspectType, MemberInfo targetElement) where T : class
    {
        // The rule implementation is naive but this is for testing purpose only.
        foreach (object rule in rules)
        {
            DependencyRule<T> typedRule = rule as DependencyRule<T>;
            if (typedRule == null) continue;

            T service = typedRule.Rule(aspectType, targetElement);
            if (service != null) return service;
        }

        if (container == null)
            throw new InvalidOperationException();

        // Fallback to the container, which should be the default and production behavior.
        return container.GetExport<T>();
    }

    public static IDisposable AddRule<T>(Func<Type, MemberInfo, T> rule)
    {
        DependencyRule<T> dependencyRule = new DependencyRule<T>(rule);
        rules.Add(dependencyRule);
        return dependencyRule;
    }

    private class DependencyRule<T> : IDisposable
    {
        public DependencyRule(Func<Type, MemberInfo, T> rule)
        {
            this.Rule = rule;
        }

        public Func<Type, MemberInfo, T> Rule { get; private set; }

        public void Dispose()
        {
            rules.Remove(this);
        }
    }
}

[Serializable]
public class LogAspect : OnMethodBoundaryAspect
{
    private Lazy<ILogger> logger;

    public override void RuntimeInitialize(MethodBase method)
    {
        logger = AspectServiceLocator.GetService<ILogger>(this.GetType(), method);
    }

    public override void OnEntry(MethodExecutionArgs args)
    {
        logger.Value.Log("OnEntry");
    }
}

internal class Program
{
    private static void Main(string[] args)
    {
        AspectServiceLocator.Initialize(new TypeCatalog(typeof(ConsoleLogger)));

        LoggedMethod();
    }

    [LogAspect]
    public static void LoggedMethod()
    {
    }
}
```

```

    {
        Console.WriteLine("Hello, world.");
    }
}

[Export(typeof (ILogger))]
internalclass ConsoleLogger : ILogger
{
    publicvoid Log(string message)
    {
        Console.WriteLine(message);
    }
}
}

```

The following code snippet shows how the logging aspect can be tested:

```

using System;
using System.Text;
using Microsoft.VisualStudio.TestTools.UnitTesting;

namespace DependencyResolution.Contextual.Test
{
    [TestClass]
    publicclass TestLogAspect
    {
        [TestMethod]
        publicvoid TestMethod()
        {
            // The ServiceLocator can be initialized for each test.using (
            AspectServiceLocator.AddRule<ILogger>(
                (type, member) =>
                    type == typeof (LogAspect) && member.Name == "TargetMethod" ? new TestLogger() : null)
            )
            {
                TestLogger.Clear();
                TargetMethod();
                Assert.AreEqual("OnEntry" + Environment.NewLine, TestLogger.GetLog());
            }
        }

        [LogAspect]
        publicvoid TargetMethod()
        {
        }
    }

    internalclass TestLogger : ILogger
    {
        publicstaticreadonly StringBuilder stringBuilder = new StringBuilder();

        publicvoid Log(string message)
        {
            stringBuilder.AppendLine(message);
        }

        publicstaticstring GetLog()
        {
            return stringBuilder.ToString();
        }

        publicstaticvoid Clear()
        {
            stringBuilder.Clear();
        }
    }
}

```

71.5. Importing Dependencies from the Target Object

The principal reason why aspects are believed to be difficult to test is that they are statically scoped by default, i.e. aspect objects are stored in static fields. However, any aspect can be made instance-scoped if it implements the `IInstanceScopedAspect` interface. See [Understanding Aspect Lifetime and Scope on page 389](#) for more information about aspect scopes.

Instance-scoped aspects can consume dependencies from the objects to which they are applied. They can also add dependencies to the target objects.

For instance, an aspect can consume a service `ILogger` using the following procedure:

To consume a service from an instance-scoped aspect:

1. Add a public property of name `Logger` and type `ILogger` to the aspect and add the `IntroduceMemberAttribute` custom attribute. This will cause the aspect to add a property to the target class. Use the parameter `MemberOverrideAction.Ignore` to ignore the property if it already exists in the target type or if it has been added by another aspect.
2. Add two custom attributes `ImportAttribute` and `CopyCustomAttributesAttribute` to the `Logger` property. This will cause the aspect to add the `[Import]` custom attribute to the `Logger` property added to the target class.
3. Add a public field of name `LoggerProperty` and type `Property<ILogger>` to the aspect class and add the `ImportMemberAttribute` custom attribute to this field, with `"Logger"` as the parameter value. This will allow the aspect to read the `Logger` property even if it has been defined from outside the aspect.
4. The aspect can now consume the dependency by calling `this.LoggerProperty.Get()`.

The procedure is illustrated in the next example.

Example: testable logging aspect that consumes the dependency from the target object

The following code snippet shows a logging aspect and how it could be used in production code:

```
using System;
using System.Collections.Generic;
using System.ComponentModel.Composition;
using System.ComponentModel.Composition.Hosting;
using System.ComponentModel.Composition.Primitives;
using System.ComponentModel.Design;
using System.Reflection;
using PostSharp.Aspects;
using PostSharp.Aspects.Advices;
using PostSharp.Extensibility;
using PostSharp.Reflection;
using PostSharp.Serialization;

namespace DependencyResolution.InstanceScoped
{
    publicinterface ILogger
    {
        void Log(string message);
    }

    [PSerializable]
    publicclass LogAspect : OnMethodBoundaryAspect, IInstanceScopedAspect
    {
        [IntroduceMember(Visibility = Visibility.Family, OverrideAction = MemberOverrideAction.Ignore)]
        [CopyCustomAttributes(typeof(ImportAttribute))]
        [Import(typeof(ILogger))]
        public ILogger Logger { get; set; }
    }
}
```



```

[ImportMember("Logger", IsRequired = true)]
public Property<ILogger> LoggerProperty;

public override void OnEntry(MethodExecutionArgs args)
{
    this.LoggerProperty.Get().Log("OnEntry");
}

object IInstanceScopedAspect.CreateInstance(AdviceArgs adviceArgs)
{
    return this.MemberwiseClone();
}

void IInstanceScopedAspect.RuntimeInitializeInstance()
{
}
}

[Export(typeof (MyServiceImpl))]
internal class MyServiceImpl
{
    [LogAspect]
    public void LoggedMethod()
    {
        Console.WriteLine("Hello, world.");
    }
}

internal class Program
{
    private static void Main(string[] args)
    {
        AssemblyCatalog catalog = new AssemblyCatalog(typeof (Program).Assembly);
        CompositionContainer container = new CompositionContainer(catalog);
        MyServiceImpl service = container.GetExport<MyServiceImpl>().Value;
        service.LoggedMethod();
    }
}

[Export(typeof (ILogger))]
internal class ConsoleLogger : ILogger
{
    public void Log(string message)
    {
        Console.WriteLine(message);
    }
}
}

```

The following code snippet shows how the logging aspect can be tested:

```

using System;
using System.ComponentModel.Composition;
using System.ComponentModel.Composition.Hosting;
using System.Text;
using Microsoft.VisualStudio.TestTools.UnitTesting;

namespace DependencyResolution.InstanceScoped.Test
{
    [TestClass]
    public class TestLogAspect
    {
        [TestMethod]
        public void TestMethod()
        {
            TypeCatalog catalog = new TypeCatalog(typeof (TestLogger), typeof (TestImpl));
            CompositionContainer container = new CompositionContainer(catalog);
            TestImpl service = container.GetExport<TestImpl>().Value;
            TestLogger.Clear();
        }
    }
}

```

Consuming Dependencies from an Aspect

```
        service.TargetMethod();
        Assert.AreEqual("OnEntry" + Environment.NewLine, TestLogger.GetLog());
    }

    [Export(typeof (TestImpl))]
    privateclass TestImpl
    {
        [LogAspect]
        publicvoid TargetMethod()
        {
        }
    }
}

[Export(typeof (ILogger))]
internalclass TestLogger : ILogger
{
    publicstaticreadonly StringBuilder stringBuilder = new StringBuilder();

    publicvoid Log(string message)
    {
        stringBuilder.AppendLine(message);
    }

    publicstaticstring GetLog()
    {
        return stringBuilder.ToString();
    }

    publicstaticvoid Clear()
    {
        stringBuilder.Clear();
    }
}
}
```

PART 14

Validating Architecture

CHAPTER 72

Restricting Interface Implementation

Under some circumstances, you may want to restrict users of an API to implement an interface. You may want to allow them to consume the interface but not to implement it in their own classes, so that, later, you can add new members to this interface without breaking the user's code. If retaining the interface as a public artifact is required, the programming language does not give you any option to enforce the desired restriction. Enter the `InternalImplementAttribute` from PostSharp.

This topic contains the following sections:

- [Adding the constraint to the interface on page 437](#)
- [Emitting an error instead of a warning on page 439](#)
- [Ignoring warnings on page 440](#)

Adding the constraint to the interface

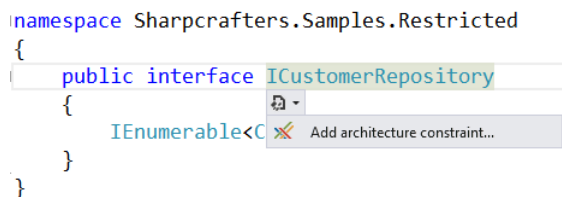
To restrict implementation of publicly declared interfaces you simply need to add `[InternalImplementAttribute]` to that interface.

NOTE

This procedure requires [PostSharp Tools for Visual Studio](#)⁵² to be installed on your machine. You can however achieve the same results by editing the code and the project manually.

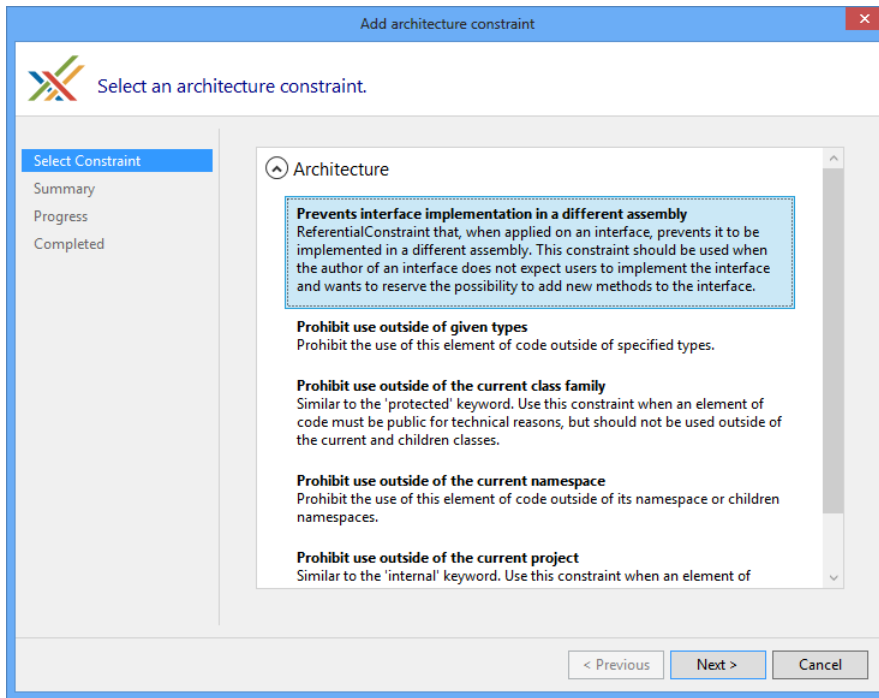
1. Place the caret over the interface that you want to add the attribute select the "Add architectural constraint...".

```
namespace Sharpcrafters.Samples.Restricted
{
    public interface ICustomerRepository
    {
        IEnumerable<C
    }
}
```

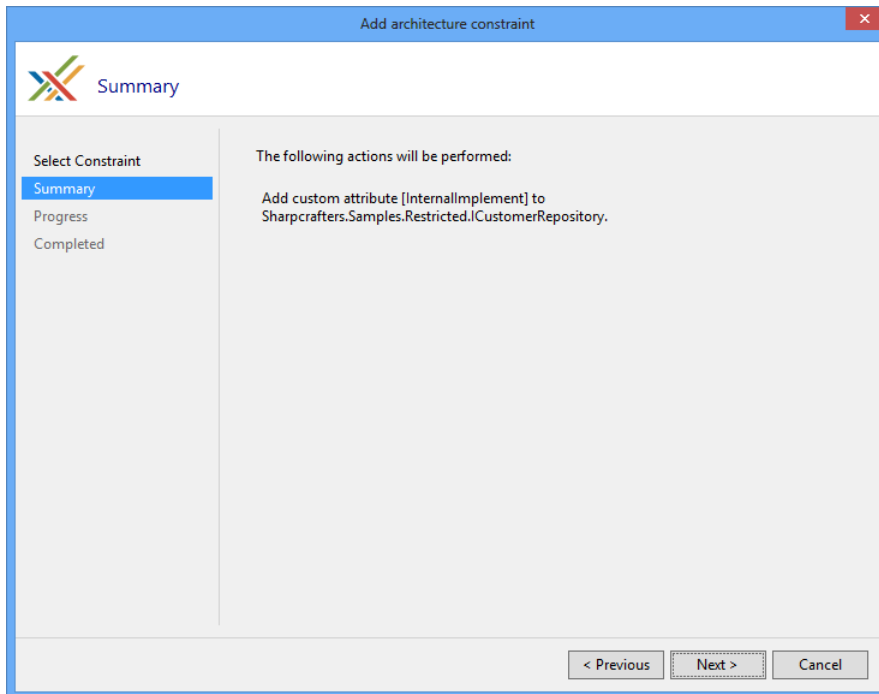


52. <https://visualstudiogallery.msdn.microsoft.com/a058d5d3-e654-43f8-a308-c3bdfdd0be4a>

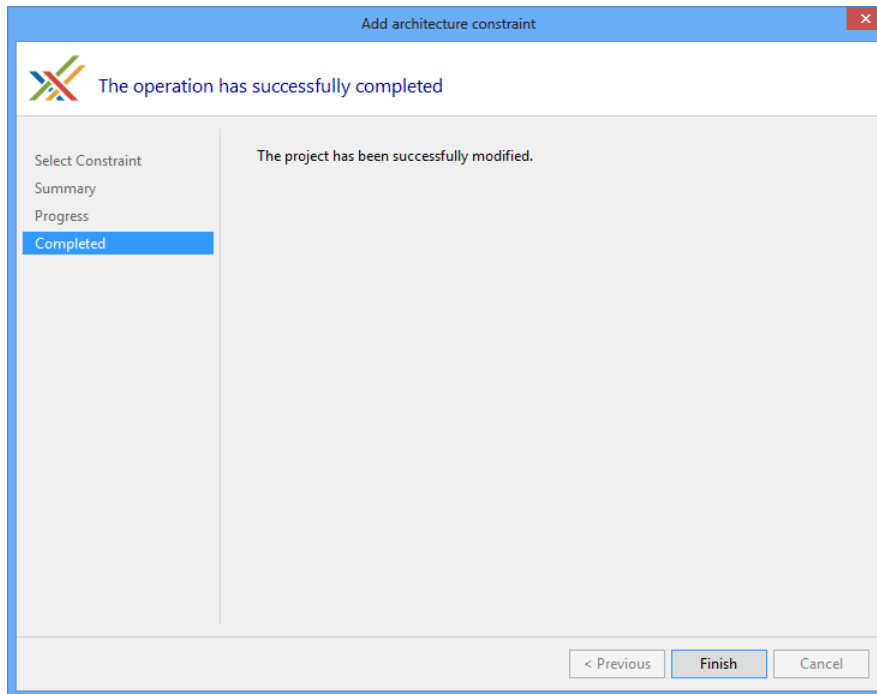
2. Select "Prevent interface implementation in a different assembly" and select **Next**.



3. Verify that you will be adding the `InternalImplementAttribute` attribute to the correct piece of code.



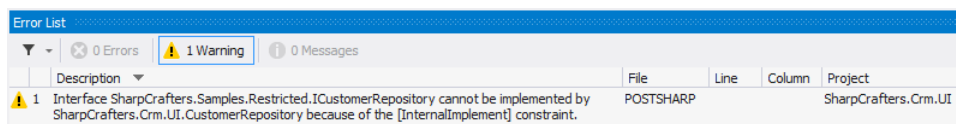
- Once the download, installation and configuration of PostSharp have finished you can close the wizard and look at the changes that were made to your codebase.



- You'll notice that the only thing that has changed in the code is the addition of the `[InternalImplementAttribute]` attribute.

```
[InternalImplement]
publicinterface ICustomerRepository
{
    IEnumerable<Customer> FetchAll();
}
```

Once that is done, implementing the interface that was decorated with the `InternalImplementAttribute` from another assembly will create a compile time warning.



NOTE

To perform this architectural validation the project that is trying to implement the interface will need to be processed by PostSharp.

Emitting an error instead of a warning

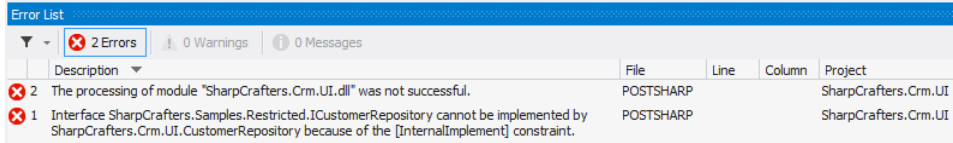
If a warning isn't strong enough for your environment you can change the output to a compile time error by setting the `InternalImplementAttribute` to have a `Severity` type of `Error`.

```
[InternalImplement(Severity = SeverityType.Error)]
publicinterface ICustomerRepository
```

Restricting Interface Implementation

```
{  
    IEnumerable<Customer> FetchAll();  
}
```

Now any reference to the decorated interface from another assembly will generate an error and fail the compilation of your project.



Ignoring warnings

If you are trying to implement a constrained interface in a separate assembly and you want to override the warning being generated there is a solution available for you. The **IgnoreWarningAttribute** attribute can be applied to stop warnings from being generated.

NOTE

The **IgnoreWarningAttribute** attribute will only suppress warnings. If you have escalated the warnings to be errors, those errors will still be generated even if the **IgnoreWarningAttribute** attribute is present.

To suppress warnings all that you need to do is add the **IgnoreWarningAttribute** attribute to the offending piece of code. In this example, we would suppress the warning being generated by adding the attribute to the class that is implementing the constrained interface. Once we have done that, the warning generated for that specific implementation would be suppressed. All other locations that are implementing this interface will continue to generate their warnings.

NOTE

You may wonder where the identifier AR0101 comes from. **IgnoreWarningAttribute** actually works with any PostSharp warning and not just this one. Any build error, whether from MSBuild, C# or PostSharp, has an identifier. To see error identifiers in Visual Studio, open the View menu and click on the Output item, select "Show output from: Build". You will see warnings including their identifiers.

```
[IgnoreWarning("AR0101")]  
publicclass PreferredCustomerRepository : ICustomerRepository  
{  
    public IEnumerable<Customer> FetchAll()  
    {  
        returnnull;  
    }  
}
```


CHAPTER 73

Controlling Component Visibility Beyond Private and Internal

When you are working on applications it's common to run across situations where you want to restrict access to a component you have written. Usually, you control this access using the `private` and/or `internal` keywords when defining the component. A class marked as `internal` can be accessed by any other class in the same assembly, but that may not be the level of restriction needed within the codebase. Access to a `private` class is restricted to those components that are inside the same class or struct that contains the `private` class, which prevents any other classes from accessing it. In one situation we are restricting access to the component to only the class or struct that contains it. In the other situation, we are allowing access to the component from any other component that is in the same assembly. What if needed something in between?

PostSharp offers the ability to define component access rules that exist between the scope of the `internal` and `private` keywords. This gives us the opportunity to restrict access to a component only from other components in the same namespace. We can also restrict access to a select few other components.

As an example let's look at a data access related class. As a precaution against developer's circumventing our data access structure we want to limit access to this repository class.

This topic contains the following sections:

- [Restricting access to specific namespaces on page 441](#)
- [Restricting access to specific types on page 444](#)
- [Controlling component visibility outside of the containing assembly on page 445](#)
- [Emitting errors instead of warnings on page 448](#)
- [Ignoring warnings on page 448](#)

Restricting access to specific namespaces

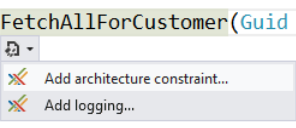
NOTE

This procedure requires [PostSharp Tools for Visual Studio](#)⁵³ to be installed on your machine. You can however achieve the same results by editing the code and the project manually.

To limit access of a class only to other classes within the validation namespace:

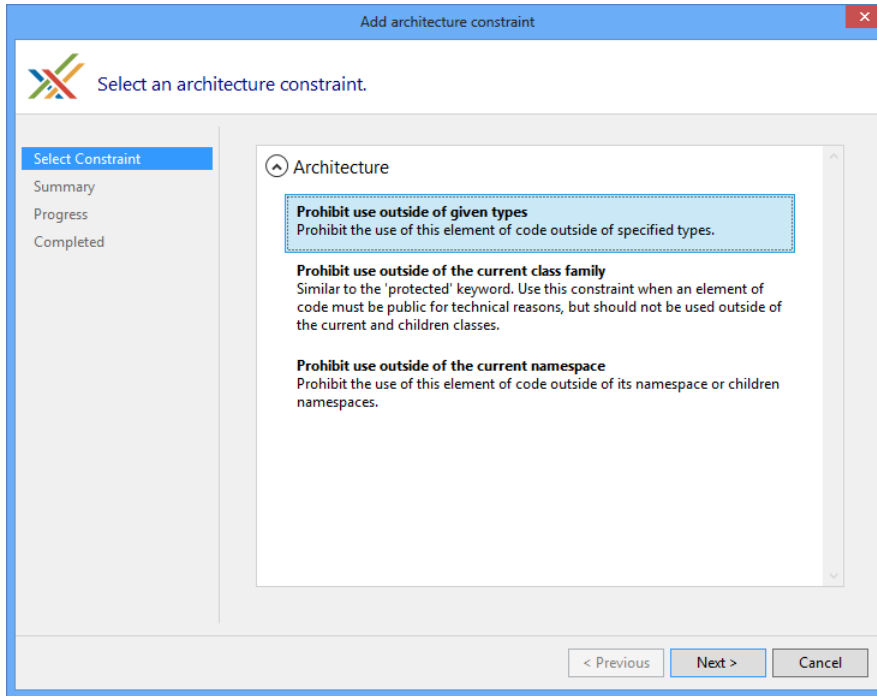
1. Put the caret on the `internal` class that should have restricted access. Select "Add architectural constraint..." from the smart tag options.

```
internal IEnumerable<Invoice> FetchAllForCustomer(Guid id)
{
    //dostuff
    return null;
}
```

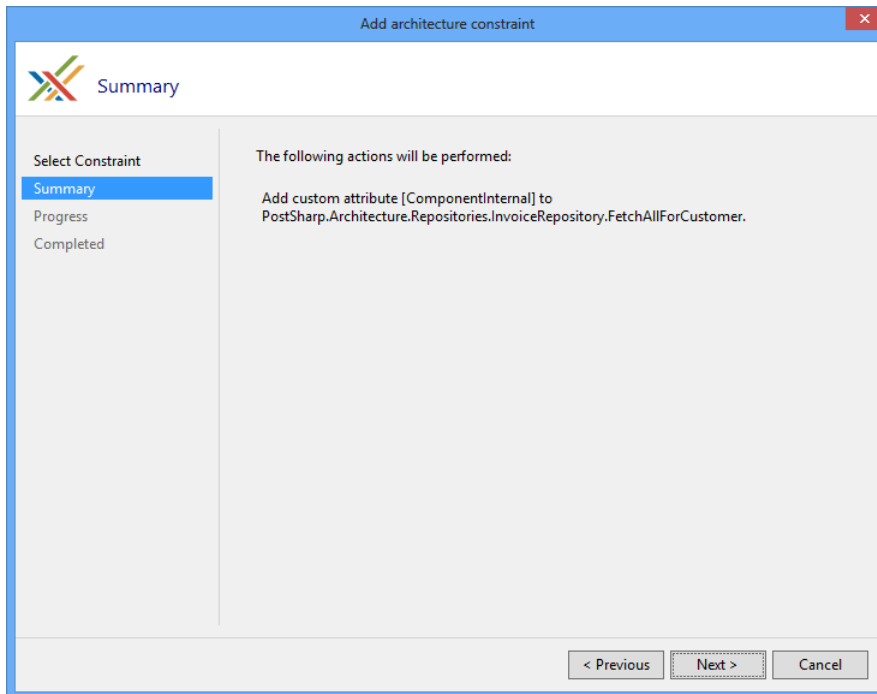


53. <https://visualstudiogallery.msdn.microsoft.com/a058d5d3-e654-43f8-a308-c3bdfdd0be4a>

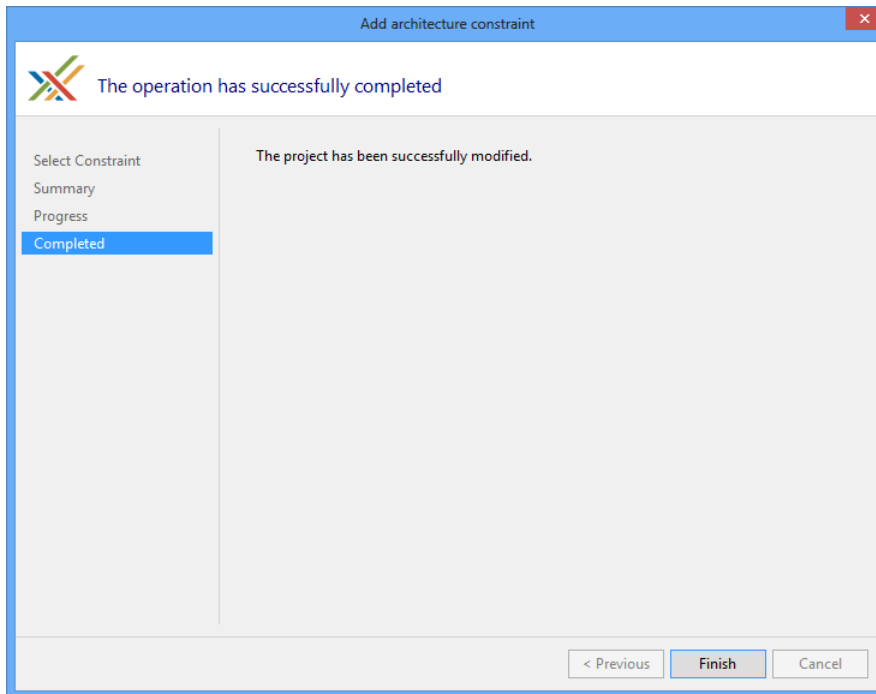
2. Select "Prohibit use outside of given types" from the list of options.



3. Verify that you will be adding the ComponentInternalAttribute attribute to the correct piece of code.



- Once the download, installation and configuration of PostSharp have finished you can close the wizard and look at the changes that were made to your codebase.



- You'll notice that the only thing that has changed in the code is the addition of the `[ComponentInternalAttribute]` attribute.

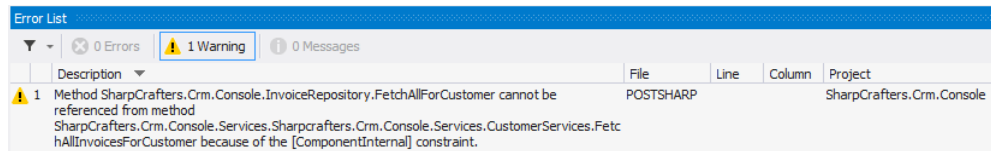
```
namespace Sharpcrafters.Crm.Console.Repositories
{
    publicclass InvoiceRepository
    {
        [ComponentInternal]
        internal IEnumerable<Invoice> FetchAllForCustomer(Guid id)
        {
            //dostuff returnnull;
        }
    }
}
```

- The `[ComponentInternalAttribute]` attribute is templated to accept a string for the namespace that should be able to access this method. There are two options that you could use. The first is to pass the attribute an array of `typeof(...)` values that represents the types that can access this method. The second option is to pass in an array of strings that contain the namespaces of the code that should be able to access this method. For our example, replace the `typeof(TODO)` with a string for the validation namespace.

7. If you try to access this component from a namespace that hasn't been granted access you will see a compile time warning in the Output window.

```
namespace Sharpcrafters.Crm.Console.Services
{
    publicclass InvoiceServices
    {
        public IEnumerable<InvoiceForList> FetchAllInvoicesForCustomer(Guid id)
        {
            var invoiceRepository = new InvoiceRepository();

            var allInvoices = invoiceRepository.FetchAllForCustomer(id);
            return
                allInvoices.Where(x => !x.PaidInFull).Select(
                    x => new InvoiceForList
                    {
                        PurchaseDate = x.PurchaseDate,
                        ShipDate = x.ShipDate,
                        TotalAmount = x.Total
                    });
        }
    }
}
```



NOTE

If you are trying to access the component from a namespace that is in a different project you will need Post-Sharp to process that project for the validation to occur.

Restricting access to specific types

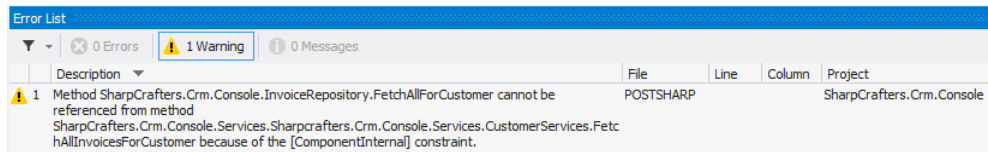
Under some circumstances, namespace level restrictions may not be tight enough for your needs. In that situation, you have the ability to apply this constraint at a type level.

1. To restrict access at a component type level you need to explicitly define which component types will have access. This is done by passing types into the constructor of the `ComponentInternalAttribute` attribute's constructor. The construct accepts an array of `Type` which allows you to define many different component types that should be granted access.

```
publicclass InvoiceRepository
{
    [ComponentInternal(typeof(Sharpcrafters.Crm.Console.Services.InvoiceServices))]
    internal IEnumerable<Invoice> FetchAllForCustomer(Guid id)
    {
        //dostuff returnnull;
    }
}
```

- Now if you try to access this component from a type that hasn't been granted access you will see a compile time warning in the Output window.

```
public class CustomerServices
{
    public IEnumerable<Customer> FetchAll()
    {
        var invoiceRepository = new InvoiceRepository();
        var allInvoices = invoiceRepository.FetchAllForCustomer(Guid.NewGuid());
    }
}
```



Controlling component visibility outside of the containing assembly

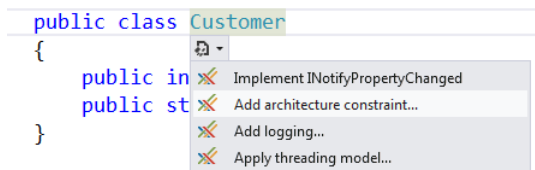
Because of framework limitations or automated testing requirements you sometimes need to declare components as public so that you can perform the desired tasks or testing. For some of those components, you probably don't want external applications accessing them. For instance, WPF controls need a default constructor for use in the designer, but sometimes you want another constructor to be used at run time, so you want to prevent the default constructor to be used from code.

PostSharp offers you the ability to decorate a publically declared component in such a way that it is not accessible by applications that reference its assembly. All you need to do is apply the `InternalAttribute` attribute.

- Let's mark the `Customer` class so that it can only be accessed from the assembly it resides in.

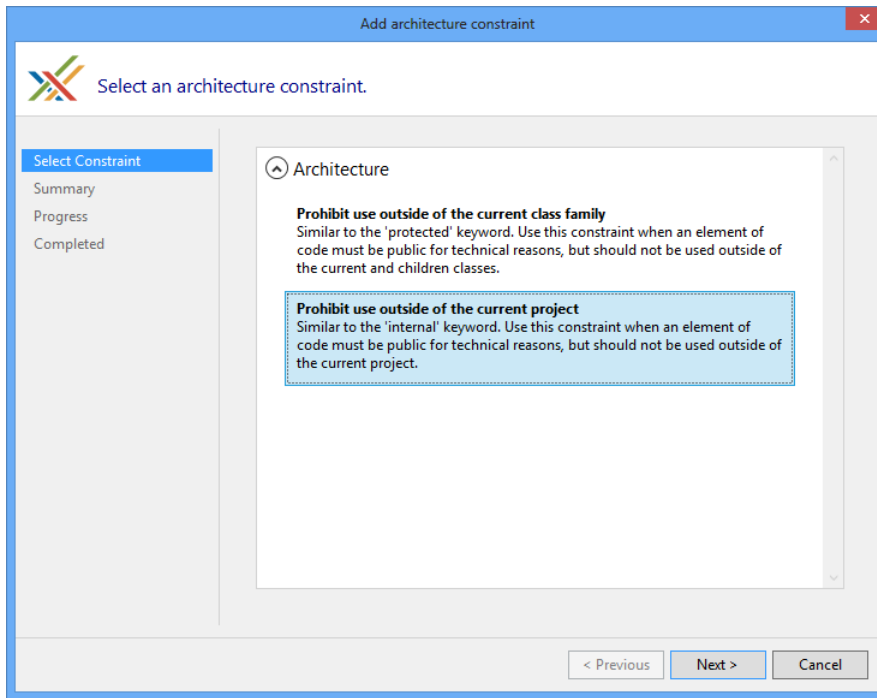
```
namespace Sharpcrafters.Crm.Core
{
    public class Customer
    {
        public int Id { get; set; }
        public string Name { get; set; }
    }
}
```

- Place the caret on the publically declared component that you want to restrict external access to and expand the smart tag. Select "Add architectural constraint" This procedure requires [PostSharp Tools for Visual Studio](https://visualstudiogallery.msdn.microsoft.com/a058d5d3-e654-43f8-a308-c3bdfdd0be4a)⁵⁴ to be installed on your machine. You can however achieve the same results by editing the code and the project manually..

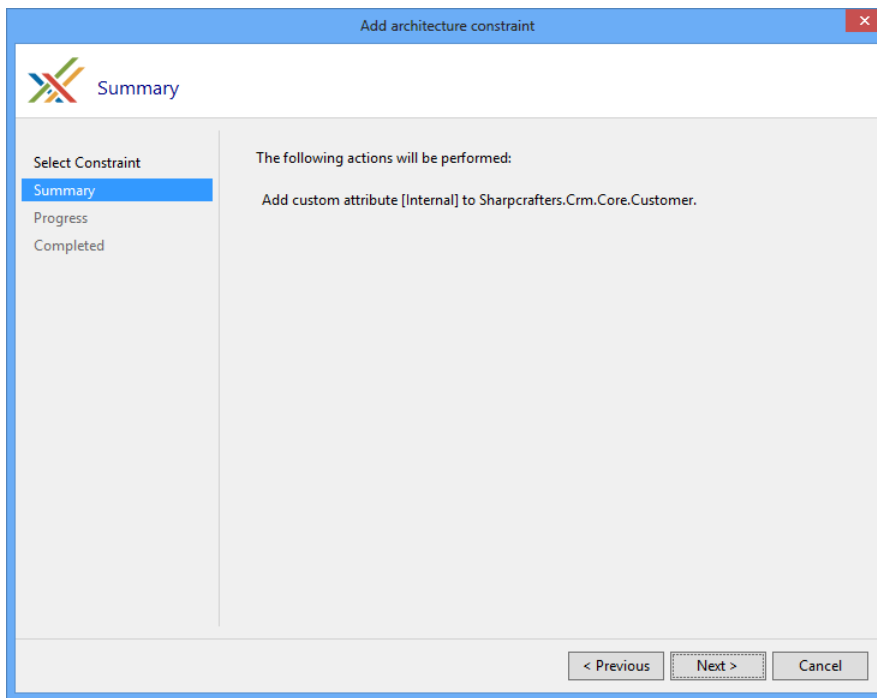


54. <https://visualstudiogallery.msdn.microsoft.com/a058d5d3-e654-43f8-a308-c3bdfdd0be4a>

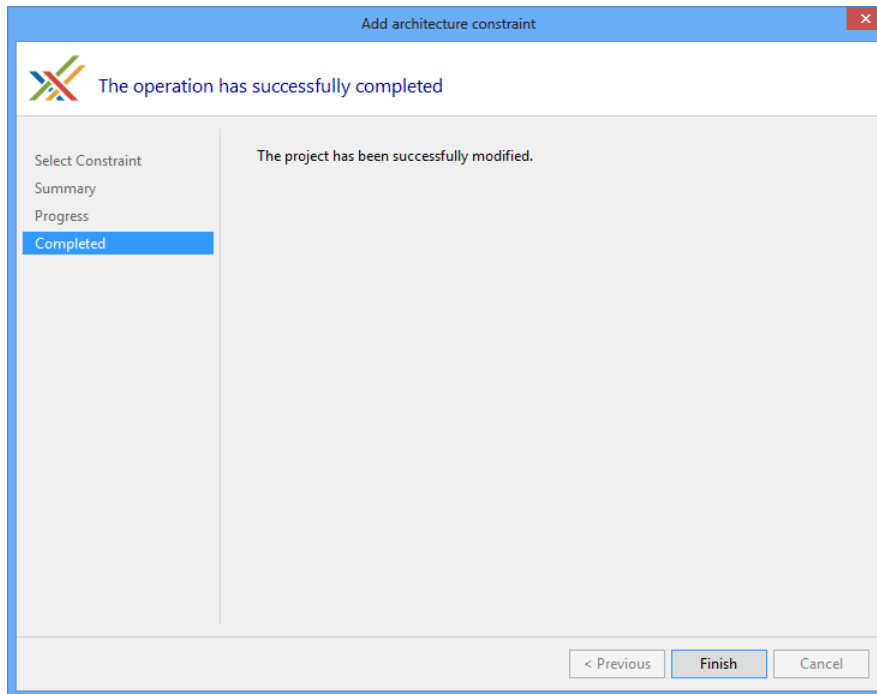
- When prompted to select a constraint, choose to "Prohibit use outside of the project".



- The summary page gives you the opportunity to review the selections that you have made. If you notice that the configuration is not what you wanted you can click the **Previous** button and adjust your selections. If the configuration meets your needs click **Next**. In this demo, you will see that the [InternalAttribute] attribute is being added to the Customer class.



- Once the download, installation and configuration of PostSharp have finished you can close the wizard and look at the changes that were made to your codebase.



- You'll notice that the only thing that has changed in the code is the addition of the `[InternalAttribute]` attribute.

```
namespace Sharpcrafters.Crm.Core
{
    [Internal]
    publicclass Customer
    {
        publicint Id { get; set; }
        publicstring Name { get; set; }
    }
}
```

- When you attempt to make use of that public component in a different assembly a compile time warning will appear in the Output window.

```
namespace Sharpcrafters.Crm.Console.Repositories
{
    publicclass CustomerRepository:ICustomerRepository
    {
        public IEnumerable<Customer> FetchAll()
        {
            returnnew List<Customer>{new Customer{Id=1,Name="Joe Johnson"}};
        }
    }
}
```

NOTE

The assembly that is attempting to use the public component will need to reference PostSharp for this validation to occur.

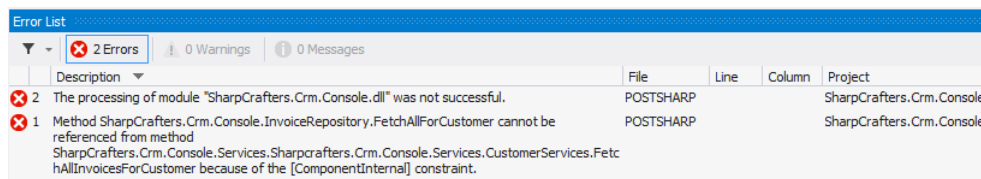
Emitting errors instead of warnings

By default, any situation that breaks the access rules defined by the application of the `ComponentInternalAttribute` or `InternalAttribute` attribute will generate a compile time warning. It's possible to escalate this warning to the error level.

1. Changing the output warning to an error requires you to set the `Severity` level.

```
[ComponentInternal(typeof (InvoiceServices), Severity = SeverityType.Error)]
public IEnumerable<Invoice> FetchAllForCustomer(Guid id)
{
    //dostuff returnnull;
}
```

2. Now when you try to access the component when access hasn't been granted the Output window will display an error message.



Ignoring warnings

There may be specific situations where you want to suppress the warning message that is being generated at compile time. In those cases, you can apply the `IgnoreWarningAttribute` attribute to the locations where you want to allow access to the component.

NOTE

The `IgnoreWarningAttribute` attribute will only suppress warnings. If you have escalated the warnings to be errors, those errors will still be generated even if the `IgnoreWarningAttribute` attribute is present.

If you wanted to allow access to the constrained component in a specific method you could add the `IgnoreWarningAttribute` attribute to that method.

```
publicclass CustomerServices
{
    [IgnoreWarning("AR0102")]
    public IEnumerable<Customer> FetchAll()
    {
        var invoiceRepository = new InvoiceRepository();
        var allInvoices = invoiceRepository.FetchAllForCustomer(Guid.NewGuid());
    }
}
```

NOTE

AR0102 is the identifier of the warning emitted by `ComponentInternalAttribute`. To ignore warnings emitted by `Internal`, use the identifier `AR0104`.

You may wonder where these identifiers come from. `IgnoreWarningAttribute` actually works with any PostSharp warning and not just this one. Any build error, whether from MSBuild, C# or PostSharp, has an identifier. To see error identifiers in Visual Studio, open the View menu and click on the Output item, select "Show output from: Build". You will see warnings including their identifiers.

If you wanted to allow access in an entire class you could add the **IgnoreWarningAttribute** attribute at the class level. Any access to the constrained component within the class would have its warning suppressed.

```
[IgnoreWarning("AR0102")]
public class CustomerServices
{
    public IEnumerable<Customer> FetchAll()
    {
        var invoiceRepository = new InvoiceRepository();
        var allInvoices = invoiceRepository.FetchAllForCustomer(Guid.NewGuid());
    }
}
```


CHAPTER 74

Developing Custom Architectural Constraints

When you are creating your applications it is common to adopt custom design patterns that must be respected across all modules. Custom design patterns have the same benefits as standard ones, but they are specific to your application. For instance, the team could decide that every class derived from `BusinessRule` must have a nested class named `Factory`, derived from `BusinessRulesFactory`, with a public default constructor.

Even performing line-by-line code reviews can miss violations of the pattern. Is there a better way to ensure that this doesn't happen? PostSharp offers the ability create custom architectural constraints. The constraints that you write are able to verify anything that you can query using reflection.

There are two kinds of constraints: *scalar constraints* and *referential constraints*.

This topic contains the following sections:

- [Creating a scalar constraint on page 451](#)
- [Creating a referential constraint on page 454](#)
- [Validating the constraint itself on page 456](#)
- [Ignoring warnings on page 456](#)

Creating a scalar constraint

Scalar constraints typically validate an element of code, while referential constraints validate how an element of code is being used.

Let's start with a scalar constraint and create a constraint that verifies the first condition our `BusinessRule` design pattern: that any class derived from `BusinessRule` must have a nested class named `Factory`. We can model this condition as a scalar constraint that applies to any class derived from `BusinessRule`. Therefore, we will create a type-level scalar constraint, apply it to the `BusinessRule` class, and use attribute inheritance to have the constraint automatically applied to all derived classes.

1. Create a class that inherits from the `ScalarConstraint` class in PostSharp.

```
using System;
publicclass BusinessRulePatternValidation : ScalarConstraint
{
}
```

2. Designate what code construct type this validation aspect should work for by adding the `MulticastAttributeUsageAttribute` attribute. In this case, we want the validation to occur on types only, and we want to enable inheritance.

```
[MulticastAttributeUsage(MulticastTargets.Class, Inheritance = MulticastInheritance.Strict)]
publicclass BusinessRulePatternValidation : ScalarConstraint
{
}
```

3. Override the `ValidateCode(Object)` method.

```
[MulticastAttributeUsage(MulticastTargets.Class, Inheritance = MulticastInheritance.Strict)]
publicclass BusinessRulePatternValidation : ScalarConstraint
{
    publicoverridevoid ValidateCode(object target)
    {
    }
}
```

4. Create a rule that checks that there's a nested type called `Factory`. You'll note that the `target` parameter for the `ValidateCode(Object)` method is an object type. Depending on which target type you declare in the `MulticastAttributeUsageAttribute` attribute, the value passed through this parameter will change. For `MulticastTargets.Type` the type passed is `Type`. To make use of the target for validation you must cast to that type first.

```
[MulticastAttributeUsage(MulticastTargets.Class, Inheritance = MulticastInheritance.Strict)]
publicclass BusinessRulePatternValidation : ScalarConstraint
{
    publicoverridevoid ValidateCode(object target)
    {
        var targetType = (Type) target;

        if ( targetType.GetNestedType("Factory") == null )
        {
            // Error
        }
    }
}
```

NOTE

Valid types for the `target` parameter of the `ValidateCode(Object)` method include `Assembly`, `Type`, `MethodInfo`, `ConstructorInfo`, `PropertyInfo`, `EventInfo`, `FieldInfo`, and `ParameterInfo`.

5. Write a warning about the rule being broken to the Output window in Visual Studio.

```
[MulticastAttributeUsage(MulticastTargets.Class, Inheritance = MulticastInheritance.Strict)]
publicclass BusinessRulePatternValidation : ScalarConstraint
{
    publicoverridevoid ValidateCode(object target)
    {
        var targetType = (Type)target;

        if (targetType.GetNestedType("Factory") == null)
        {
            Message.Write(
                targetType, SeverityType.Warning,
                "2001",
                "The {0} type does not have a nested type named 'Factory'.",
                targetType.DeclaringType,
                targetType.Name);
        }
    }
}
```

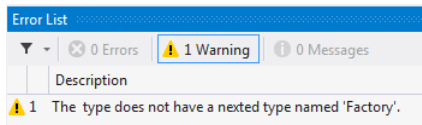
- Attach the rule to the code that needs to be protected. For this example we want to add this rule to the `BusinessRule` class.

```
[BusinessRulePatternValidation]
publicclass BusinessRule
{
    // No Factory class here.
}
```

NOTE

This example shows applying the constraint to only one class. If you want to apply a constraint to large portions of your codebase, read the section on [Adding Aspects to Multiple Declarations on page 112](#)

- Now if you compile the project you will see an error in the Output window of Visual Studio when you run a build.



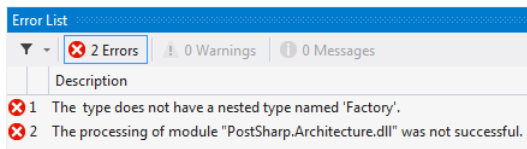
NOTE

Due to a bug, the error code will not be shown in the Error list.

- In some circumstances you may determine that a warning isn't aggressive enough. We can alter the rule that you have created so that it outputs a compile time error instead. All that you need to do is change the `SeverityType` in the `Message.Write` to `Error`.

```
[MulticastAttributeUsage(MulticastTargets.Class, Inheritance = MulticastInheritance.Strict)]
publicclass BusinessRulePatternValidation : ScalarConstraint
{
    publicoverridevoid ValidateCode(object target)
    {
        var targetType = (Type)target;

        if (targetType.GetNestedType("Factory") == null)
        {
            Message.Write(
                targetType, SeverityType.Error,
                "2001",
                "The {0} type does not have a nested type named 'Factory'.",
                targetType.DeclaringType,
                targetType.Name);
        }
    }
}
```



Using this technique it is possible to create rules or restrictions based on a number of different criteria and implement validation for several design patterns.

When you are working on projects you need to ensure that they adhere to the ideals and principles that our project teams hold dear. As with any process in software development, manual verification is guaranteed to fail at some point in time. As you do in other areas of the development process, you should look to automate the verification and enforcement of our ideals. The ability to create custom architectural constraints provides both the flexibility and verification that you need to achieve this goal.

Creating a referential constraint

Now let's create a referential constraint that verifies the second condition our `BusinessRule` design pattern: that the `BusinessRule` class can only be used in the `Controllers` namespace. You can model this condition as a referential constraint and apply the constraint to any class in your codebase. If you apply this constraint to the entirety of your codebase you will ensure that the `BusinessRule` design pattern is only referenced in the `Controllers` namespace.

1. Create a class that inherits from the `ReferentialConstraint` class in PostSharp.

```
publicclass BusinessRuleUseValidation : ReferentialConstraint
{
}
```

2. Declare that this aspect should work only on types by adding the `MulticastAttributeUsageAttribute` attribute to the class.

```
[MulticastAttributeUsage(MulticastTargets.Class, Inheritance = MulticastInheritance.Strict)]
publicclass BusinessRuleUseValidation : ReferentialConstraint
{
}
```

3. Override the `ValidateCode(Object, Assembly)` method.

```
[MulticastAttributeUsage(MulticastTargets.Class, Inheritance = MulticastInheritance.Strict)]
BusinessRuleUseValidation : ReferentialConstraint
{
    publicoverridevoid ValidateCode(object target, Assembly assembly)
    {
    }
}
```

4. Create the rule that checks for the use of the `BusinessRule` type in the target code.

```
[MulticastAttributeUsage(MulticastTargets.Class, Inheritance = MulticastInheritance.Strict)]
publicclass BusinessRulePatternValidation : ReferentialConstraint
{
    publicoverridevoid ValidateCode(object target, Assembly assembly)
    {
        var targetType = (Type) target;
        var usages = ReflectionSearch
            .GetMethodsUsingDeclaration(typeof (BusinessRule));

        if (usages !=null)
        {
            // Warning
        }
    }
}
```

NOTE

The rule here makes use of the `ReflectionSearch` helper class that is provided by the PostSharp framework. This class, along with others, is an extension to the built in reflection functionality of .NET and can be used outside of aspects as well.

5. Write a warning message to be included in the Output window of Visual Studio.

```
[MulticastAttributeUsage(MulticastTargets.Class, Inheritance = MulticastInheritance.Strict)]
publicclass BusinessRulePatternValidation : ReferentialConstraint
{
    publicoverridevoid ValidateCode(object target, Assembly assembly)
    {
        var targetType = (Type) target;
        var usages = ReflectionSearch
            .GetMethodsUsingDeclaration(typeof (BusinessRule));

        if (usages !=null)
        {
            Message.Write(
                targetType, SeverityType.Warning,
                "2002",
                "The {0} type contains a reference to 'BusinessRule'" +
                "which should only be referenced from Controllers.",
                targetType.Name);
        }
    }
}
```

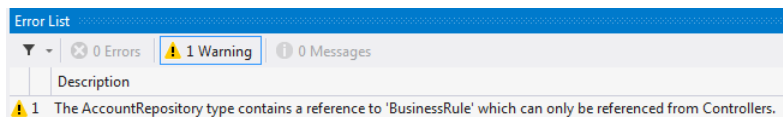
6. Attach the referential constraint that you created to any code that needs to be checked. In this example, add an attribute to the AccountRepository class.

```
namespace PostSharp.Architecture.Repositories
{
    [BusinessRuleUseValidation]
    publicclass AccountRepository
    {
        publicvoid AddAccount(string name)
        {
            var businessRule = new BusinessRule();
            businessRule.DoStuff();
        }
    }
}
```

NOTE

This example shows applying the constraint to only one class. If you want to apply this constraint to a larger portion of your codebase, read the section on [Adding Aspects to Multiple Declarations on page 112](#).

7. Now when you compile the project you will see a warning in the Output window in Visual Studio.



NOTE

If using a warning isn't aggressive enough you can change the `SeverityType` to `Error`. Now when the rule is broken an error will appear in the Output window of Visual Studio and the build will not be successful.

CAUTION NOTE

PostSharp constraints operate at the lowest level. For instance, checking relationships of a type with the rest of the code does not implicitly check the relationships of the methods of this type. Also, checking relationships of namespaces is not possible.

Custom attribute multicasting can be used to apply a constraint to a large number of types, for instance all types of a namespace. But this would result in one constraint instance for every type, method and field on this namespace. Although this has no impact on run time, it could severely affect build time. For this reason, the current version of PostSharp Constraints is not suitable to check isolation (layering) of namespaces at large scale.

Referential constraints provide you with the ability to declare architectural design patterns right in your code. By documenting these patterns right in the codebase you are able to provide easy access for the development team as well as continual verification that your desired design patterns are being adhered to.

Validating the constraint itself

Now that you have created scalar and referential constraints you can be assured that certain architectural rules are being consistently implemented in your codebase. There is one thing that is missing though.

With what you have done thus far, it is possible to attach your architectural constraints to any code element in your projects. This may not be appropriate. For example, the scalar constraint that you created to perform the `BusinessRulePatternValidation` may be a valid constraint only on classes that exist in the `Models` namespace.

Let's look at how we can ensure that this constraint is only enforced on classes that exist in the `Models` namespace.

1. Open the `BusinessRulePatternValidation` class that you created earlier.
2. Override the `ValidateConstraint(Object)` method.
3. Write the validation logic to ensure that this constraint is only applied to classes in the `Models` namespace.

NOTE

When the `ValidateConstraint(Object)` method returns `true`, it tells PostSharp that the constraint should be applied to that target code element. When the `ValidateConstraint(Object)` method returns `false` PostSharp will not apply the constraint to the target code element.

Now, when the `BusinessRulePatternValidation` attribute is applied to a class that is not in the `Models` namespace of your project, there will be no warning or error added to the Visual Studio Output window.

When the attribute is applied to a class in the `Models` namespace and that class doesn't pass the constraint's rules you will continue to see the warning or error indicating this architectural failure.

Ignoring warnings

There will be situations where a constraint is generating a warning that is of no concern. In these exceptional circumstances, it is best if you remove the warning from the Visual Studio Output window.

To ignore these unnecessary warnings, find the target code that is responsible for generating the warning. Add the **IgnoreWarningAttribute** attribute to the target code entering the **MessageId** of the warning that you want to suppress.

The **MessageId** can be found in your constraint where you issue the `Message.Write` command. The **Reason** value performs no function during the suppression of the warning. It exists so that you can provide clear communication as to why the warning is being ignored.

NOTE

The **IgnoreWarningAttribute** attribute will only suppress the issuance of Message.Write statements that are assigned a SeverityType of Warning. If the SeverityType is set to Error the **IgnoreWarningAttribute** attribute will have no suppression effect on that statement.

PART 15

Testing and Debugging

CHAPTER 75

Debugging Run-Time Logic

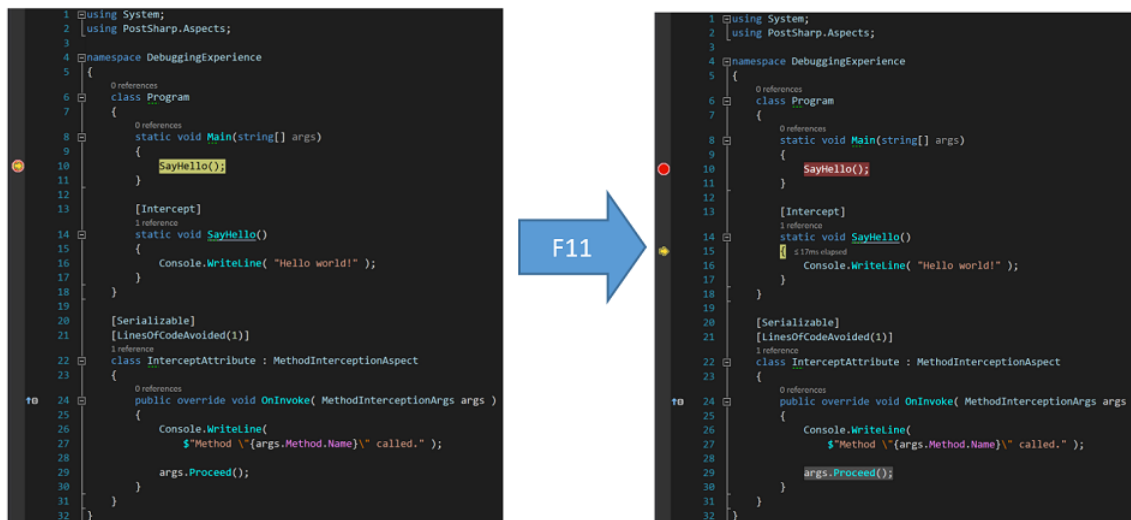
Enhancing your code with aspects gives you a new dimension to your debugging experience. With PostSharp, patterns are implemented in classes that are separate from the business logic. Most of the time, you will want to debug just the business logic. But sometimes, you will want to debug the aspects: aspect code will now be skipped by default during step-into sessions and in the call stack window.

This topic contains the following sections:

- [Stepping into aspect code on page 461](#)
- [Showing aspects code in the call stack window on page 461](#)
- [Disabling debugger enhancements on page 462](#)

Stepping into aspect code

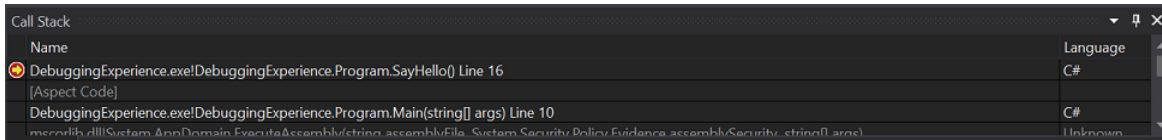
Suppose you have a `SayHello` method intercepted by an `Intercept` aspect. You are about to step into the `SayHello` method. By default, the debugger steps over the code of the `Intercept` aspect and then breaks in the beginning of the `SayHello` method. When `Step Into Aspects` is enabled, the debugger will step into the `Intercept` aspect.



The `Step Into Aspects` feature is disabled by default. To turn it on, go to the menu **PostSharp / Options**, then to the **General** tab and the **Debugging** section, and check the **Step Into Aspects** check box. Now you can step into aspects using the **Step Into (F11)** command of the debugger.

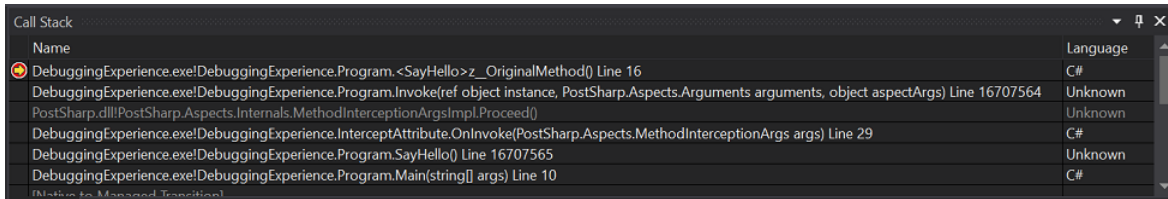
Showing aspects code in the call stack window

By default, all the calls to the methods of the `Intercept` aspect are hidden behind one stack frame named `[Aspect Code]`. Suppose you have the `SayHello` method intercepted by the `Intercept` aspect like in the example above. You are inside the `SayHello` method. In the call stack, all the methods introduced by PostSharp are hidden.



Sometimes you may need to see the real call stack that includes all intermediate method calls generated by PostSharp. This is the purpose of the **Show Aspects Code in Call Stack** feature.

To turn it on, go to the menu **PostSharp / Options**, then to the **General** tab and the **Debugging** section, and check the **Show Aspects Code in Call Stack** check box. Now, the call stack includes all the methods introduced by PostSharp.

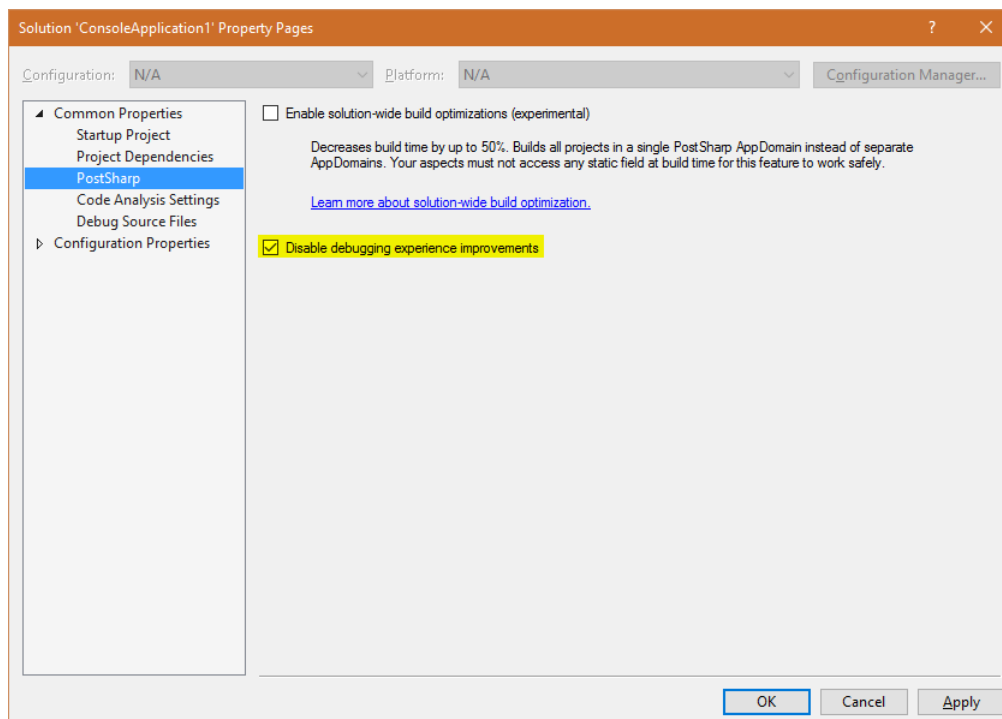


Disabling debugger enhancements

PostSharp improves your debugging experience by installing extensions for the Visual Studio Debugger and by enhancing PDB files during the build. In some use cases you may want to disable these PostSharp debugger extensions and revert back to the default debugging behavior in Visual Studio (e.g. building for a new or unsupported target framework, debugging code on unsupported devices, working around bugs, etc.).

To disable PostSharp debugger extensions:

1. Right-click on your solution in the **Solution Explorer** and then click **Properties**
2. On the **PostSharp** page of the displayed solution property pages dialog, select the check box **Disable debugging experience improvements**.



3. Confirm the change by clicking **OK** and then rebuild your solution.

NOTE

Debugging extensions are automatically disabled when you build your project using MSBuild from outside Visual Studio.

CAUTION NOTE

If you build a solution with debugging extensions enabled, you must debug the solution with an instance of Visual Studio where debugging extensions are enabled, otherwise your debugging experience will be frustrating.

CHAPTER 76

Debugging Build-Time Logic

It may seem unusual to debug compile-time logic, but like any process, it is perfectly legal and even simple to debug the build process!

Basically, what you will do is to attach a debugger to the PostSharp process. If you use the standard MSBuild targets for PostSharp, define the constant `PostSharpAttachDebugger=True`.

The trick is easier to explain when you have compile-time logic (your aspect, for instance) and the transformed assembly in different Visual Studio projects.

Suppose you have your aspects logic `MyAspects.csproj` and unit tests (i.e. the code to be transformed) in `MyAspects.Test.csproj`. The easiest way to debug `MyAspects.csproj` is the following:

To debug the build-time logic of an aspect:

1. Open Visual Studio and load the solution containing `MyAspects.csproj`.
2. Open the Visual Studio Command Prompt and go to the directory containing `MyAspects.Test.csproj`.
3. Build `MyAspects.csproj` using Visual Studio as usual .
4. From the command prompt, type:

```
msbuild MyAspects.Test.csproj /T:Rebuild /P:PostSharpAttachDebugger=True
```
5. The build process will hit a break point. When it happens, attach the instance of `MyAspects.csproj` Visual Studio.

NOTE

Because of a bug in Visual Studio, you need to use the **mixed debugging engine**. To do that, check the option **Manually choose the debugging engines** in the Visual Studio Just-In-Time Debugger and select both the managed and the native engines.

6. Set up break points in your code and continue the program execution.

CHAPTER 77

Testing that an Aspect has been Applied

In the previous section, we have seen how to test the aspect behavior itself. Now, let's see how we can test that the aspect has been applied to the expected set of targets. This can also be called *testing the pointcut*.

Why test that the aspect has been properly applied?

You may need to test whether an aspect has been applied to specific targets for one of the following reasons:

- The aspect is applied using non-trivial regular expressions with `MulticastAttribute`.
- The aspect is silently filtered out using `CompileTimeValidate(MethodBase)`.
- The aspect is applied using an `IAAspectProvider`.

Testing that the aspect behavior is exhibited

The most obvious way to test that the aspect has been applied to an element of code is to execute that code and ensure that the code actually exhibits the aspect behavior. This approach does not differ from the one described in section [Testing Run-Time Logic on page 469](#).

Testing that the aspect custom attribute is present

You can check that an aspect has been applied to a target by reflecting the custom attributes present on this element of code.

However, custom attributes representing aspects are stripped by default. If you want PostSharp to emit custom attributes, follow instructions of section [Reflecting Aspect Instances at Runtime on page 126](#).

NOTE

Aspects added by `IAAspectProvider` are not represented by custom attributes, so their presence cannot be tested by this approach.

Parsing the PostSharp symbol file

PostSharp generates a symbol file named `bin\Debug\MyAssembly.pssym`, where `MyAssembly` is the name of the assembly. In theory, you could use this file to determine which elements of code have been modified by aspects in your project.

CAUTION NOTE

The PostSharp symbol file format is undocumented and unsupported. It means that PostSharp support team cannot answer questions related to this file format.

Testing that an Aspect has been Applied

CHAPTER 78

Testing Run-Time Logic

When designing a test strategy for aspects, it is fundamental to understand that aspects cannot be used in isolation. They are always used in the context of the code artifact to which it has been applied. Therefore, when writing an aspect, two kinds of test artifacts must be written:

- *Test target code* to which the aspect will be applied.
- *Test invocation code* that invokes the target code and verifies that the combination of the aspect and the target code exhibits the intended behavior.

Achieving large test coverage

As with other code, you have to test the aspect with input context that varies enough to produce a large code coverage.

In the case of aspects, the input context is composed of the following items:

- *Arguments of the aspect itself*, i.e. constructor arguments and property values. If the aspect behavior depends on aspect arguments, high code coverage of the aspect requires varying aspect arguments.
- *Target code* can be considered as conceptually being a part of the input arguments of the aspect. For instance, if an aspect contains logic that depends on the method being static or non-static, you should test the aspect against both static and non-static methods.
- *Arguments of the target code* can affect the run-time behavior of the aspect. For instance, a buggy aspects may incorrectly handle null arguments.

Example: testing a caching aspect

The following example demonstrates how to test a caching aspect. High code coverage is achieved by varying the target code and testing with null and non-null parameters.

```
using System;
using System.Threading.Tasks;
using Microsoft.VisualStudio.TestTools.UnitTesting;

namespace Samples
{
    [TestClass]
    public class TestCacheAspect
    {
        private static int invocations;

        // Instance method without parameters

        [TestMethod]
        public void TestInstanceMethodWithoutParameter()
        {
            int call1 = this.InstanceMethodWithoutParameter();
            int call2 = this.InstanceMethodWithoutParameter();

            Assert.AreEqual(call1, call2);
        }

        [Cache]
    }
}
```

Testing Run-Time Logic

```
private int InstanceMethodWithoutParameter()
{
    return invocations++;
}

// Static method without parameters

[TestMethod]
public void TestStaticMethodWithoutParameter()
{
    int call1 = StaticMethodWithoutParameter();
    int call2 = StaticMethodWithoutParameter();

    Assert.AreEqual(call1, call2);
}

[Cache]
private static int StaticMethodWithoutParameter()
{
    return invocations++;
}

// Instance method with parameters

[TestMethod]
public void TestInstanceMethodWithParameter()
{
    int call1a = this.InstanceMethodWithParameter("foo");
    int call2a = this.InstanceMethodWithParameter(null);
    int call1b = this.InstanceMethodWithParameter("foo");
    int call2b = this.InstanceMethodWithParameter(null);

    Assert.AreEqual(call1a, call1b);
    Assert.AreEqual(call2a, call2b);
    Assert.AreNotEqual(call1a, call2a);
}

[Cache]
private int InstanceMethodWithParameter(string param)
{
    return invocations++;
}

[TestMethod]
public void TestInstanceTaskMethodWithParameters()
{
    int invocationsOld = invocations;

    int call11a = this.InstanceTaskMethodWithParameters(1, 1).Result;
    int call11b = this.InstanceTaskMethodWithParameters(1, 1).Result;

    Assert.AreEqual(call11a, call11b);
    Assert.AreEqual(1, invocations - invocationsOld);
}

#region InstanceTaskMethodWithParameters

[Cache]
private Task<int> InstanceTaskMethodWithParameters(int a, int b)
{
    return Task.Run(() =>
    {
        invocations++;
        return a + b;
    });
}

#endregion
}
}
```

CHAPTER 79

Testing Build-Time Logic

Testing build-time logic of aspects has specific challenges:

- Aspects can emit errors and warnings, which cannot be tested using a run-time testing framework. We need a mechanism to test error messages themselves.
- When a project contains a large number of test cases (which are all compiled at the same time), it is difficult to isolate one specific case when the debugger is attached to the build process (see [Debugging Build-Time Logic on page 465](#)). We need a mechanism to run the build process on a single test case.

Therefore, we built a test framework specifically for the purpose of testing aspects.

This topic contains the following sections:

- [Creating an aspect unit test project on page 471](#)
- [Executing a single test on page 472](#)
- [Executing all tests from a directory on page 472](#)
- [Executing all tests in the project directory on page 472](#)
- [Test that messages are emitted on page 472](#)
- [Allow unsafe code on page 472](#)
- [Creating a reference assembly on page 472](#)

Creating an aspect unit test project

To create an aspect unit test project:

1. Create a console project and add all required references to it.
2. Add PostSharp to this project
3. Edit the project file using a text editor. The project file must import *PostSharp.BuildTests.targets* before *Microsoft.CSharp.targets* ([download⁵⁵](#)). File *PostSharp.targets* also needs to be included (which is the case if the PostSharp NuGet package is added to the project).
4. Implement each test case as a standalone file having its own Program class and Main method. To avoid naming conflicts, every file should have a distinct namespace.

A test is considered successful in the following situations:

- the test compiles using the C# or VB compiler, and
- the test compiles using PostSharp without any unexpected message (see below), and
- the output exe is valid according to **PEVERIFY**, and
- the output exe executes successfully and returns the exit code 0,

This default behavior can be altered by test directives, as described below.

55. <https://www.postsharp.net/downloads/samples/3.0/PostSharp.BuildTests.targets>

Executing a single test

Execute the following line from the command prompt:

```
msbuild /t:TestOne /p:Source=MyFile.cs
```

Executing all tests from a directory

Execute the following line from the command prompt:

```
msbuild /t:Test /p:SourceDir=MyDirectory
```

Executing all tests in the project directory

Execute the following line from the command prompt:

```
msbuild /t:Test
```

Test that messages are emitted

If the test is expected to emit a message (error, warning, information), insert the text `@ExpectedMessage(PS0001)` in the test file as a comment line.

If this directive is present, the test will be valid if and only if all expected messages, and no other, have been emitted.

Allow unsafe code

To enable unsafe code and disable verification by **PEVERIFY**, insert the text `@Unsafe` in the test file as a comment line.

Creating a reference assembly

In case that a test requires a dependency assembly (typically, for tests that require two assemblies, for instance testing aspect inheritance that crosses assembly boundaries), you can create a second file named *MyTest.Dependency.cs*, if the first file is named *MyTest.cs*. This will create an assembly *MyTest.Dependency.dll*, and main test will have a reference to this assembly.

PART 16

Hacking

CHAPTER 80

Executing Code Just After the Assembly is Loaded

Visual Basic has a concept of module. The module is a special class that gets initialized immediately when the assembly is loaded. This feature is implemented by the CLR, but is not exposed to the C# language. The `ModuleInitializerAttribute` attribute allows you to have module initializers in C#.

To add a module initializer to your project:

- Create a public or internal method that has no parameter and no return value. The type declaring the method cannot have generic parameters.
- Add the `ModuleInitializerAttribute` attribute to this method.

You can add several module initializers a project. Module initializers will be executed in the order you specified in the attribute constructor.