PostSharp 4.0

User Manual

Table of Contents

PART 1

Introduction

CHAPTER 1

PostSharp Overview

This chapter gives a high-lever overview of PostSharp. It is composed of the following topics:

Topic	Description
What Is PostSharp [7]	This topic describes how PostSharp can help your development team to deliver better software in less time.
PostSharp Components [11]	This topic describes all components of PostSharp (Visual Studio extension, NuGet packages).
How Does PostSharp Work [11]	This topic gives a high-level understanding of how PostSharp integrates with your build process.
Key Technologies [13]	This topic positions PostSharp in the realm of metaprogramming, aspect-oriented programming, static analysis and dynamic analysis.

1.1. What Is PostSharp

PostSharp is a 100%-compatible extension to the C# or VB languages that adds support for design patterns and thread safety. Teams that use PostSharp are able to deliver features with fewer lines of code and fewer defects.

In conventional object-oriented programming, technical concerns like threading, INotifyPropertyChanged or logging generally result in a large amount of boilerplate code. This boilerplate code is tangled with business code, and makes it more difficult to understand and modify the business meaning of the source code.

Boilerplate code is in fact instances of source code patterns. As any pattern, their instances exhibit a great amount of regularity and predictability. Any decent software developer would be able to explain to their colleague how to properly implement INotifyPropertyChanged. These implementation guidelines would form some sort of algorithm, expressed in natural language, that the colleague would execute. Most of the work would be repetitive and would only require a limited amount of creativity.

Algorithmic work is exactly what machines are good at, so why not offload it the compiler? This is exactly what PostSharp has been designed for. Traditional languages have concepts like classes, methods, fields, but they don't have any construct to represent patterns like the implementation of INotifyPropertyChanged.

PostSharp adds support for patterns into the C# and VB languages, allowing developers to work more productively at a higher level of abstraction and to avoid the bugs that stem from working with a large amount of technical details.

This section gives you an overview of the tools that PostSharp give you to automate the implementation and validation of patterns in your code:

- Standard patterns [8]
- Thread safety patterns [8]
- Implementation of custom patterns [9]
- Validation of custom patterns [10]

Standard patterns

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

- INotifyPropertyChanged: see INotifyPropertyChanged [77].

- Parent/child relationships: see Parent/Child Relationships [91].

- Undo/redo: see Undo/Redo [105].

- Code contracts: see Contracts [121].

- Logging: see Walkthrough: Adding Detailed Tracing to a Code Base [129].

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]
public class CustomerViewModel
{
    [Required]
    public Customer Customer { get; set; }

    public string FullName { get { return this.Customer.FirstName + " " + this.Customer.LastName; } }
}

[NotifyPropertyChanged]
[Recordable]
public class Customer
{
    public string FirstName { get; set; }
    public string LastName { get; set; }

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

    [Url]
    public string HomePage { get; set; }

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

[NotifyPropertyChanged]
[Recordable]
public class Address
{
    [Parent]
    public Customer Parent { get; private set; }

    public string Line1 { get; set; }
}
```

Thread safety patterns

Multi-threading 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 multi-threading. 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 [174].

- Freezable: see Freezable Threading Model [171].

- Actor: see Actor Threading Model [178].

- Reader/Writer Synchronized: see Reader/Writer Synchronized Threading Model [183].

- Synchronized: see Synchronized Threading Model [188].

- Thread Unsafe: see Thread-Unsafe Threading Model [192].

- Thread Affine: see Thread Affine Threading Model [195].

- Thread Dispatching: see Dispatching a Method to Background [205] and Dispatching a Method to the UI Thread [207].

- Deadlock Detection: see Detecting Deadlocks at Runtime [209].

Example

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

```
[Freezable]
public class Customer
{
    public string Name { get; set; }

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

}

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

    public string Line1 { get; set; }
}

public class Program
{
    public static void 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 terms 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 [217].

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 hand-written 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 solving common design problems. For instance, the `InternalImplementAttribute` constraint prevents an interface to be implemented in an external assembly.

See Validating Architecture [319] 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 decide to create an architectural constraint that will validate the code against the project-specific *Business Rule Factory* pattern.

```
[MulticastAttributeUsage(MulticastTargets.Class, Inheritance = MulticastInheritance.Strict)]
public class BusinessRulePatternValidation : ScalarConstraint
{
    public override void 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 );
        }

        // ...
    }
}

[BusinessRulePatternBalidation]
public abstract BusinessRule
{
  // ...
}
```

1.2. 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 in the build process and post-processes the output of the compiler.

This topic contains the following sections.

- MSBuild Integration [11]
- MSIL Rewriting [11]

MSBuild Integration

PostSharp integrates itself in the build process thanks to *PostSharp.targets*, which is imported into each project using Post-Sharp 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 [61] for details.

MSIL Rewriting

PostSharp post-processes the compiler output by reading and disassembling the intermediate assembly, execute the required transformations and validations, and rewriting the final assembly 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.

1.3. PostSharp Components

PostSharp is composed of the following components:

8

Component Name	Component Kind	Description
PostSharp Tools for Visual Studio	Visual Studio Extension	This is the user interface of PostSharp. It extends the Visual Studio editor and provides a new menu, option pages, toolbox windows, and smart tags. PostSharp Tools for Visual Studio must be installed on each developer workstation but is not required on build servers.
PostSharp	NuGet Package	This is the core of PostSharp. It contains the PostSharp compiler, the MSBuild integration (the *PostSharp.targets* file is added to your projects upon installation of the PostSharp package), and the facade library *PostSharp.dll*, which exposes features to developers: • *PostSharp Aspect Framework* allows you to automate the *implementation* of other code patterns and address code repetitions that are specific to your own applications, or simply that are not available off-the-shelf in a pattern library. For more information, see sections Developing Custom Aspects [217] and Adding Aspects to Code [145]. • *PostSharp Architecture Framework* allows you to automate the *validation* of design pattern implementations, to enforce design intend, or simply to verify coding guidelines. The framework allows you to create constraint classes that encapsulate the validation logic and that can be applied to code artifacts. The framework provides tools to analyze the relationships between code artifacts and have access to the AST of method bodies. For more information, see section Validating Architecture [319].
PostSharp Common Pattern Library	NuGet Package	This package contains definitions and aspects that are shared by other pattern libraries, including: • Aggregatable pattern: see Parent/Child Relationships [91] for details. • Basic definitions for threading aspects.
PostSharp Model Pattern Library	NuGet Package	This package contains the following features: • NotifyPropertyChanged aspect: see INotifyPropertyChanged [77] for details. • Recordable pattern for undo/redo: see Undo/Redo [105] for details. • Code contracts: see Contracts [121] for details.
PostSharp Model Pattern Library (Controls)	NuGet Package	This package contains the undo/redo buttons for WPF. See Adding Undo/Redo to the User Interface [107] for details.

Component Name	Component Kind	Description
PostSharp Threading Pattern Library	NuGet Package	This package contains all threading aspects and their infrastructure, including: • All threading models: see Writing Thread-Safe Code with Threading Models [169] for details. • Thread dispatching aspects: see Dispatching a Method to Background [205] and Dispatching a Method to the UI Thread [207] for details • Deadlock detection: see Detecting Deadlocks at Runtime [209] for details.
PostSharp Diagnostics Pattern Library	NuGet Package	This package contains the logging aspect. See Walkthrough: Adding Detailed Tracing to a Code Base [129] for details.

1.4. Key Technologies

PostSharp combines several technologies to leverage design pattern automation.

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 cross-cut 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 [217].

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 of 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 [319] 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 runtime, 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 runtime. 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 [169].

Another example of use of dynamic program analysis in PostSharp is deadlock detection. For details, see Detecting Deadlocks at Runtime [209].

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

CHAPTER 2
Getting Started with 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
Getting Started: Selecting and Creating Aspects [15]	In the first phase, the team learns about the concepts and abilities of PostSharp, and identify 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.
	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.
Getting Started: Installing and Deploying PostSharp [17]	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.
Getting Started: Using Aspects [17]	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 *use* 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.

2.1. Getting Started: 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 you 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	PostSharp Overview [7]
	Requirements and Compatibility [31]
More About Design Pattern Automation	Article: Design Pattern Automation[1]
More About Aspect-Oriented Programming	Aspect-Oriented Programming in Microsoft .NET[2]
	White Paper: Producing High-Quality Software with Aspect-Oriented Programming[3]

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 [121]
	Parent/Child Relationships [91]
User interface patterns	Understanding the NotifyPropertyChanged Aspect [86]
	Understanding the Recordable Aspect [116]
Multi-threading	White Paper: Threading Models for Object-Oriented Programming[4]
	Detecting Deadlocks at Runtime [209]
	Dispatching a Method to the UI Thread [207]
	Dispatching a Method to Background [205]
Diagnostics	Logging [129]

---

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

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.

Topic	Articles
Aspects	Developing Custom Aspects [217]
Architecture Validation	Validating Architecture [319]

2.2. Getting Started: Installing and Deploying PostSharp

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

Topic	Articles
Deploying to the Development Environment	Installing PostSharp Tools for Visual Studio [34]
	Installing PostSharp Into a Project [34]
	Deploying License Keys [35]
	Uninstalling PostSharp [51]
Deploying to the Build Infrastructure	Using PostSharp on a Build Server [46]
	Restoring Packages at Build Time [47]
	Using PostSharp with Visual Studio Online [48]
Deploying to Production or End-User Devices	Deploying PostSharp to End-User Devices [57]
Deploying to Large or Regulated Development Environments	License Audit [38]
	Using PostSharp License Server [38]
	Installing PostSharp Unattended [0]
	Upgrading Large Repositories from a Previous Version of PostSharp [49]

2.3. Getting Started: 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

You can of course 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 [31]
	Installing PostSharp Tools for Visual Studio [34]
Add PostSharp to a project and keep it up-to-date	Installing PostSharp Into a Project [34]
	Upgrading from a Previous Version of PostSharp [49]

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 [129]
Code Contracts	Contracts [121]
INotifyPropertyChanged	INotifyPropertyChanged [77]
Aggregatable	Parent/Child Relationships [91]
Disposable	Walkthrough: Automatically Disposing Children Objects [96]
Undo and Redo	Undo/Redo [105]
Threading Models	Writing Thread-Safe Code with Threading Models [169]
	Freezable [171], Immutable [174], Actor [178], Reader/Writer Synchronized [183], Synchronized [188], Thread-Unsafe [192], Thread Affine [195]
	Compatibility of Threading Models [200]
	Opting In and Out From Thread Safety [198]
Deadlock Detection	Detecting Deadlocks at Runtime [209]
Dispatching Threads	Dispatching a Method to the UI Thread [207]
	Dispatching a Method to Background [205]

Topic	Articles
Architecture Validation	Restricting Interface Implementation [319]
	Controlling Component Visibility Beyond Private and Internal [322]

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 [146]
Resolving Errors	Working with Errors, Warnings, and Messages [73]

CHAPTER 3

What's New in PostSharp

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:

3.1. 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 [22]
- Disposable pattern [22]
- Immutable threading model [22]
- Freezable threading model [22]
- Synchronized threading model [22]
- Redesign of reader-writer-synchronized, actor, and thread-unsafe threading models [22]
- Recordable pattern (undo/redo) [22]
- Dynamic location imports [22]
- Aspect repository [22]
- OnInstanceConstructed advice [22]
- InitializeAspectInstance advice [23]
- NotifyPropertyChanged optimization [23]

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 Relationships [91].

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 Walkthrough: Automatically Disposing Children Objects [96].

Immutable threading model

The Immutable patterns made functional languages popular for its great usefulness in multi-threaded 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 [174].

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 [171].

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 [188].

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 is 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 `ImportMember-Attribute`, we added the `IAdviceProvider` interface and the `ImportLocationAdviceInstance` class.

Aspect repository

The new `IAspectRepositoryService` service exposes the list of all aspects added to the code model, both using custom attributes or `IAspectProvider`, 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 `Runtime-InitializeInstance` but passes information about the reason why the aspect is initialized (constructor, clone, deserialization).

NotifyPropertyChanged optimization

Our `NotifyPropertyChangedAttribute` is four times faster at runtime on average.

3.2. 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 [23]
- Improved configuration system [23]
- Build-time performance improvement [23]
- Resolution of file and line of error messages [24]
- Indentation in logging [24]
- Separate licensing of Pattern Libraries [24]

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 [65] 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 [71] 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 Diagnostics 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.

3.3. 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 [24]
- Diagnostics Pattern Library [24]
- Threading Pattern Library [24]
- Smart tags and wizards in Visual Studio [25]
- Better platform support through Portable Class Libraries [25]
- Unified deployment through NuGet and Visual Studio Gallery [25]
- Transparency to obfuscators [25]
- Deprecation of old platforms [25]

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 runtime, 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 multi-threading is being addressed. It provides three threading models: actors (`Actor`), reader-writer synchronized (`ReaderWriterSynchronized-Attribute`) 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 `Binary-Formatter`. 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.

3.4. 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 [25]
- Support for NuGet and improved no-setup experience [25]
- Compatibility with obfuscators [26]
- Extended reflection API [26]
- Architectural validation [26]
- Compatibility with Code Contracts [26]
- Support for Silverlight 5.0 [26]
- License server [26]

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[5]. 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.

---

5. http://www.nuget.org/List/Packages/PostSharp

Compatibility with obfuscators

PostSharp can now be used jointly, and without limitation of features, with some obfuscators. See **[obfuscators]** for details.

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 [319] 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.

3.5. 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 [26]
- Composite aspects (advices and pointcuts) [27]
- Adaptive code generation [27]
- Interception aspect for fields and properties [27]
- Interception aspect for events [27]
- Aspect dependencies [27]
- Instance-scoped aspects [27]
- Support for new platforms [27]
- Build performance improvements [27]

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 to 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 runtime, 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 Post-Sharp) 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

3.6. 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 [28]
- Reading assemblies without loading them in the CLR [28]
- Lazy loading of assemblies [28]
- Build-time performance improvement [28]
- Support for Mono [28]
- Support for Silverlight 2.0 and the Compact Framework 2.0 [28]
- Pluggable aspect serializer & partial trust [28]

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 4
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.

PostSharp is composed of the following components:

- *PostSharp Tools for Visual Studio* is the user interface of PostSharp. It extends the Visual Studio editor and provides a new menu, option pages, and toolbox windows, and smart tags. PostSharp Tools for Visual Studio must be installed on each developer workstation but is not required on build servers.

- *PostSharp Compiler* contains the build-time components and the run-time libraries. PostSharp Compiler is distributed only as a NuGet package. It is typically included in the source repository or is restored before build from the package repository.

- *PostSharp Pattern Libraries* provide ready-made implementations of the most common design patterns. They are distributed as NuGet package and depend on the *PostSharp* package.

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

4.1. Requirements and Compatibility

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

This topic contains the following sections.

Requirements on development workstations and build servers

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

- Microsoft Visual Studio 2010, 2012 or 2013, except Express editions, but including Community Edition (Visual Studio is not required on build servers).

- .NET Framework 4.5.

- Windows Vista SP2, Windows 7 SP1, Windows 8, Windows 8.1, Windows Server 2003 SP2, Windows Server 2003 R2 SP2, Windows Server 2008 SP2, Windows Server 2008 R2 SP2, Windows Server 2012, Windows Server 2012 R2.

- NuGet Package Manager 2.2 or later.

NOTE

The latest version of NuGet Package Manager will be installed automatically by PostSharp if NuGet 2.2 is not already installed. This operation requires administrative privileges.

CAUTION NOTE

NuGet Package Manager needs to be configured manually in order to match the requirements of some corporate environments, especially in situations with a large number of Visual Studio solutions. Please contact our technical support if this is a concern for your team.

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.

PostSharp Component	.NET Framework	Silverlight	Windows Phone (Silverlight)	Windows Phone (WinRT)	Windows (WinRT)
Aspect Framework	3.5 SP1, 4.0, 4.5	4, 5	7, 8	8.1	8, 8.1
Architecture Framework	3.5 SP1, 4.0, 4.5	4, 5	7, 8	8.1	8, 8.1
Diagnostics Pattern Library	4.0, 4.5	-	-	-	-
Model Pattern Library	4.0, 4.5	-	8	8.1	8, 8.1
Threading Pattern Library	4.0, 4.5	-	-	-	-

NOTE

PostSharp supports *Portable Class Library* projects that target frameworks shown in the table.

Compatibility with VB

Although Visual Basic is fully supported by the PostSharp compiler, several limitations will affect the developer experience:

- PostSharp Tools for Visual Studio are not available for VB.

- PostSharp will not resolve the file and line/column of errors and warnings.

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.

Additionally, ASP.NET "vNext", which has a different project system than MSBuild, 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 request the Code Analysis to execute on the output of PostSharp by setting the `ExecuteCodeAnalysisOnPostSharp-Output` MSBuild property to `True`. For more information, see Configuring Projects Using MSBuild [61].

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

Starting from version 3, PostSharp generates assemblies that are theoretically compatible with all obfuscators.

CAUTION NOTE

PostSharp 3 generates 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.

Compatibility with ILMerge

You can use **ILMerge** to combine *PostSharp.dll*, or assemblies processed by PostSharp, into a large assembly. No special action is required to support ILMerge.

Known Incompatibilities

PostSharp is not compatible with the following products or features:

Product or Feature	Reason	Workaround
ILMerge	Bug in ILMerge	Use another product (such as 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 or Silverlight 5
Silverlight 4	No longer under Microsoft mainstream support	Use PostSharp 3.1 or Silverlight 5
.NET Compact Framework	No support for PCL	Use PostSharp 2.1 or Windows Phone 8
.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	Target Windows Phone 8 or use PostSharp 3.1
Mono	Not Supported	Compile on Windows using MSBuild

Product or Feature	Reason	Workaround
Visual Studio Express	Microsoft's licensing policy	Use Visual Studio Community Edition
Delayed strong-name signing on cloud build servers	No way to unregister verification of strong names	Use normal (non-delayed) strong-name signing or use build servers where you have administrative access.
ASP.NET Web Sites	Not built using MSBuild	Convert the ASP.NET Web Site to an ASP.NET Web Application.

4.2. 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 [34] for details.

To install PostSharp Tools for Visual Studio:

1. Download the file *PostSharp-X.X.X.X.vsix* from http://www.postsharp.net/download.

2. Open the file *PostSharp-X.X.X.X.vsix*.

3. Start Visual Studio.

4. Complete the 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 2.1.

4.3. 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 [34]
- Including files in source control [35]

Adding PostSharp to a project

To add PostSharp to a project:

1. Open the **Solution Explorer** in Visual Studio.

2. Right-click on the project.

3. Click on **Add PostSharp**.

TIP
Remember that adding PostSharp to a project just means adding the *PostSharp* NuGet package. If you want to add Post-Sharp to several projects in a solution, it may be easier to use NuGet to manage packages at 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[6] and NuGet Configuration Settings[7] for more information.

Including files in source control

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

- *packages.config*
- **.psproj*
- **.pssln*

Optionally, you can also include the *packages* folder in source control. Note that some people advise against this practice. If you choose not to include the *packages* folder in source control, read Restoring Packages at Build Time [47].

Once you have all of these files include 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.

4.4. 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 [35]
- Installing the license key in source control [37]

Registering a license key using the user interface

Registering a license key using the user interface is the prefered procedures for individual developers and small teams.
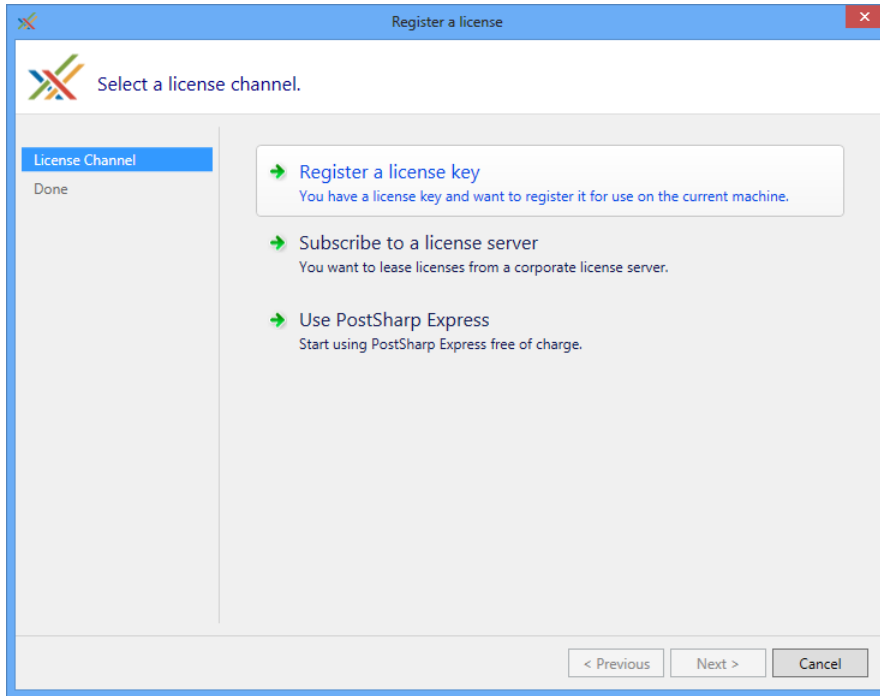
To register a license key using the user interface:

1. Open Visual Studio.

2. Click on menu **PostSharp**, then **Options**.

3. Open the **License** option page.

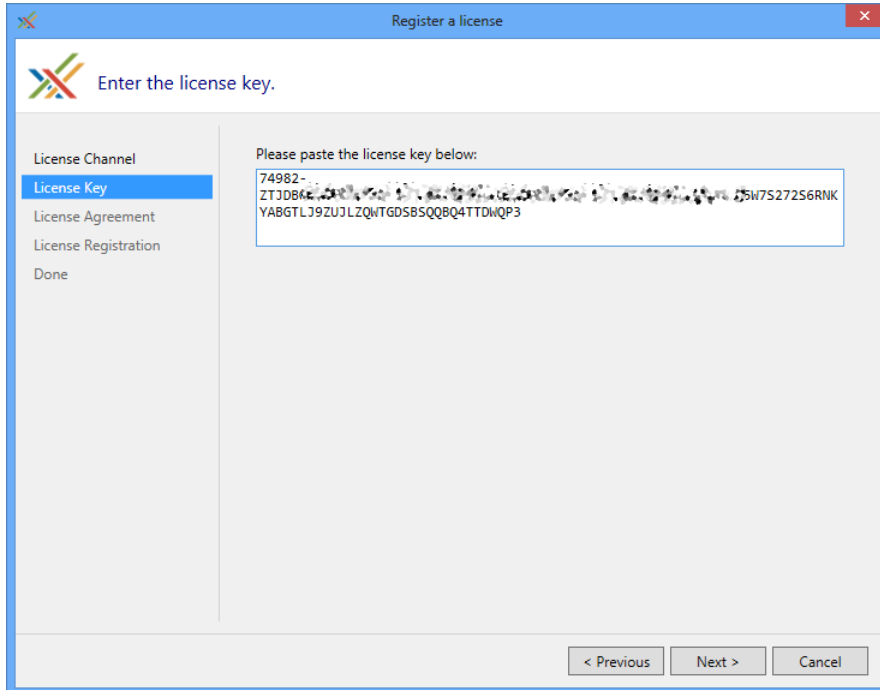4. Click on the **Register a license** link.

---

6. http://docs.nuget.org/docs/reference/nuget-config-file
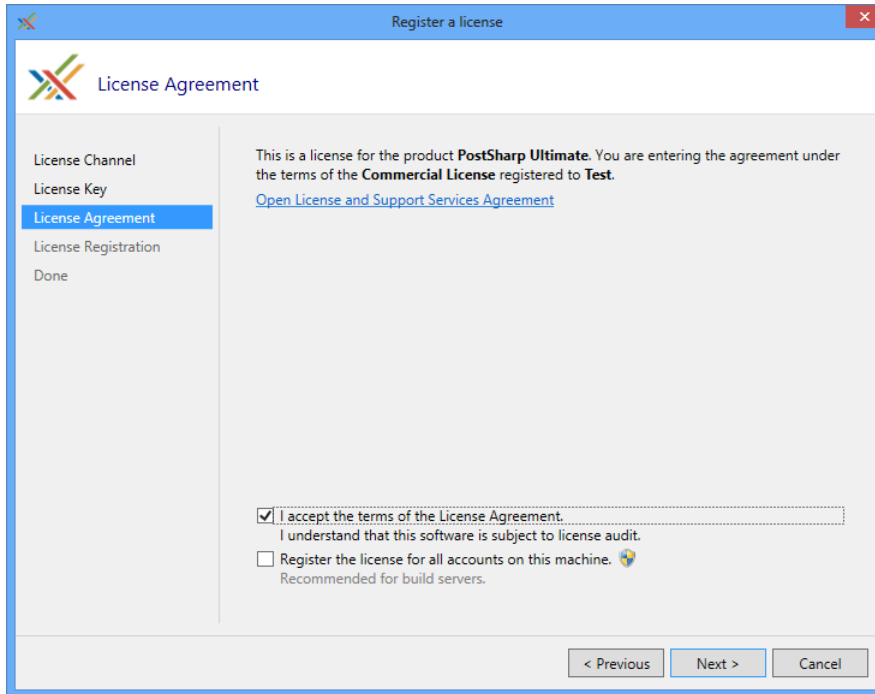7. http://docs.nuget.org/docs/reference/nuget-config-settings

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 source control

It is possible to install the license key in 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:

```
<Project xmlns="http://schemas.postsharp.org/1.0/configuration">
  <License Value="000-AAAAAAAAAAAAAA"/>
</Project>
```

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

See Working with PostSharp Configuration Files [65] for details about this configuration file.

4.4.1. 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:

- 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 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 industry, 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 [38].

4.4.2. 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 [38]
- Design Principles [39]
- Subscribing to the license server [39]
- Installing the license settings in source control [42]

If you are a system administrator or license administrator, see also Installing and Servicing PostSharp License Server [42].

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 [38] 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 lead 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 use 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.
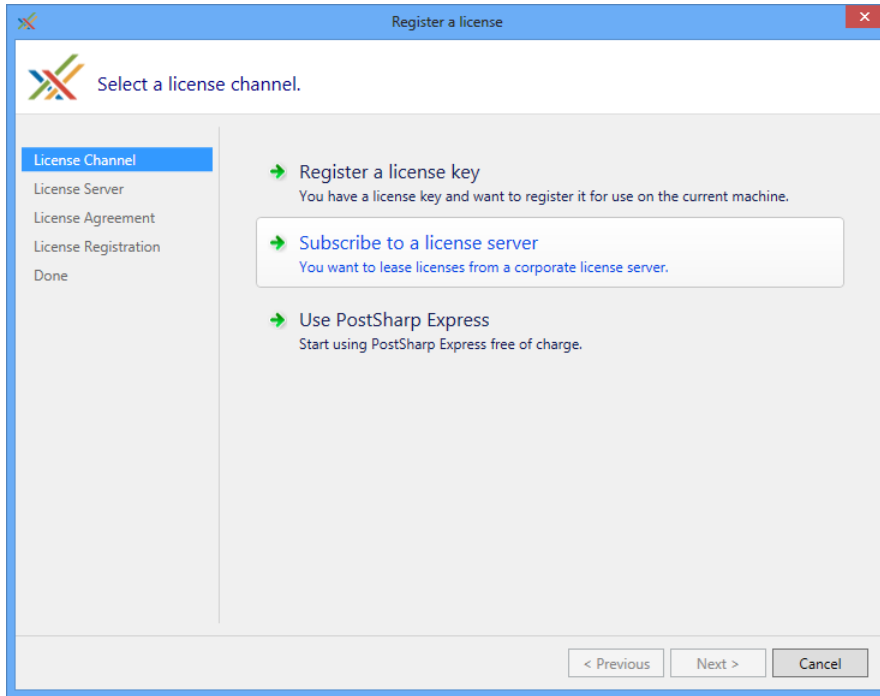
5. Paste the URL of the license server. You can click on the **Open** hyperlink to verify that the URL is correct and that you have access to it. Click **Next**.
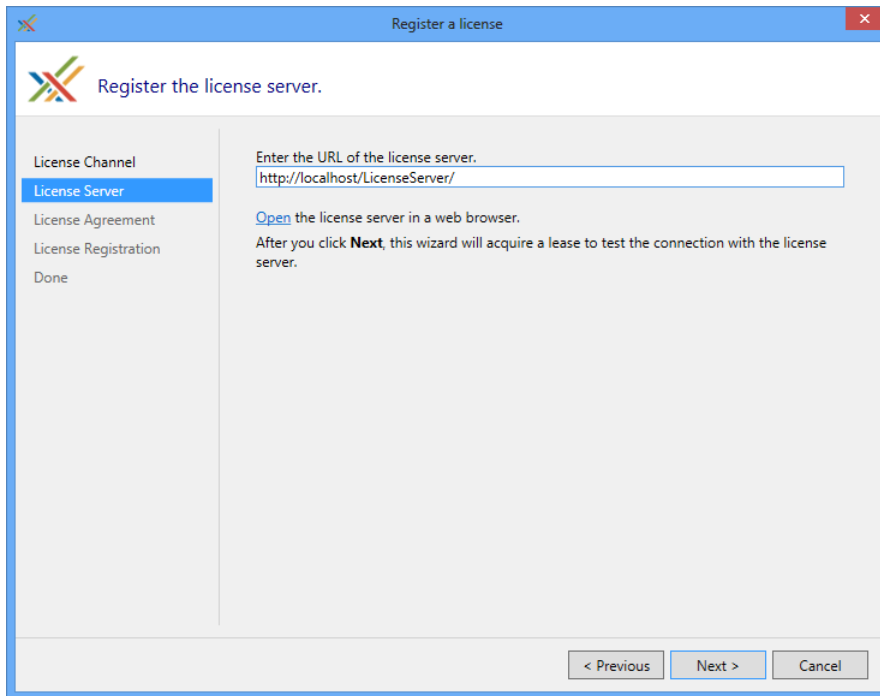
6. Read the license agreement and check the option **I accept**. Click on **Next**.

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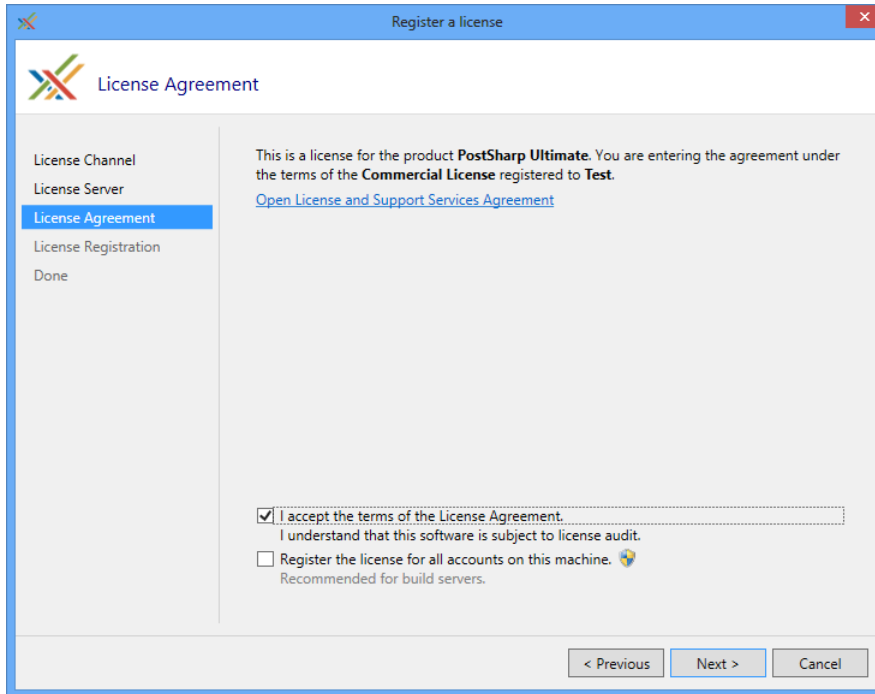
If you are registering the license server on a build server, also check the option **Register the license for all accounts on this machine.**

7. You are done.

Installing the license settings in source control

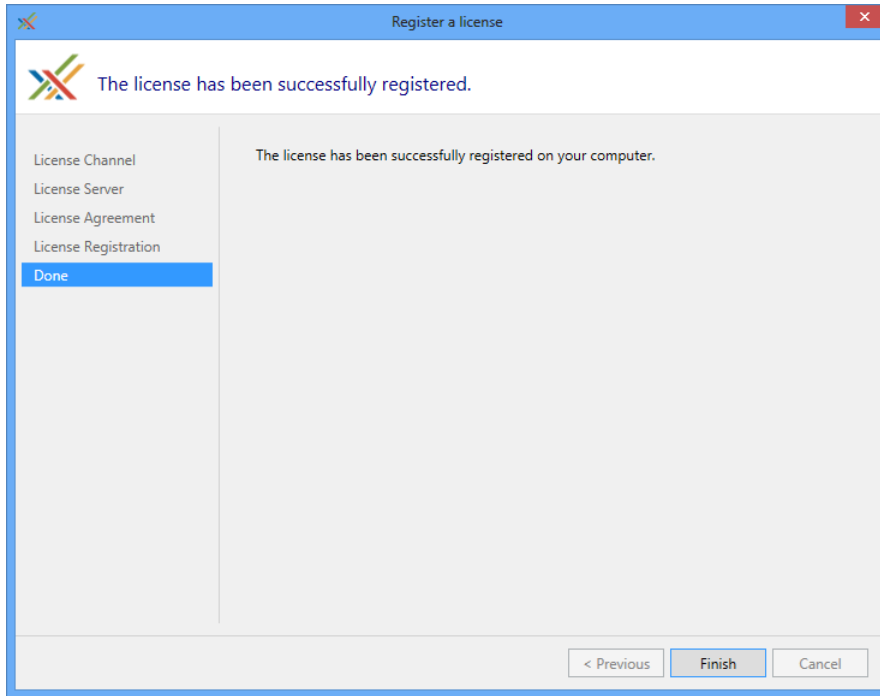It is possible to subscribe to the license server using a file stored in source control by using the exact same mechanism as to register a license key.

To install license settings 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:

```
<Project xmlns="http://schemas.postsharp.org/1.0/configuration" x:xmlns="http://schemas.postsharp.org/1.0/configuratio
  <License Value="http://server/path"/>
</Project>
```

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

See Working with PostSharp Configuration Files [65] for details about this configuration file.

4.4.3. 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 not obfuscated, 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 [43]

System Requirements

PostSharp License Server is an ASP.NET 4.0 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 the PostSharp website. For version `3.1.44`, the URL is *http://www.postsharp.net/downloads/postsharp-3.1/v3.1.44/SharpCraftersLicenseServer-3.1.44.zip*. You can open the download manager at http://www.postsharp.net/downloads/ and browse to the version you are interested in.

   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 4.0.

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 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 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**.

3. Paste the license key and click on button **Add**.
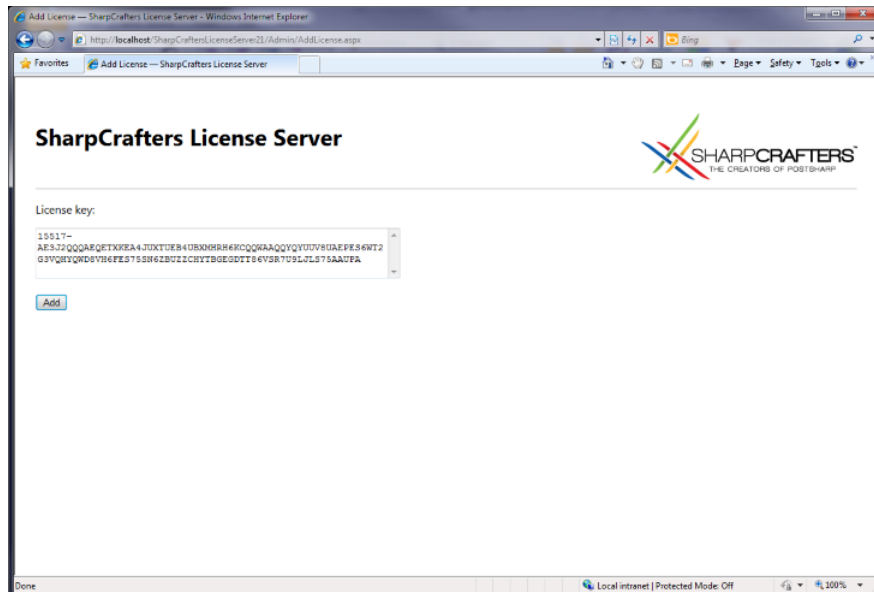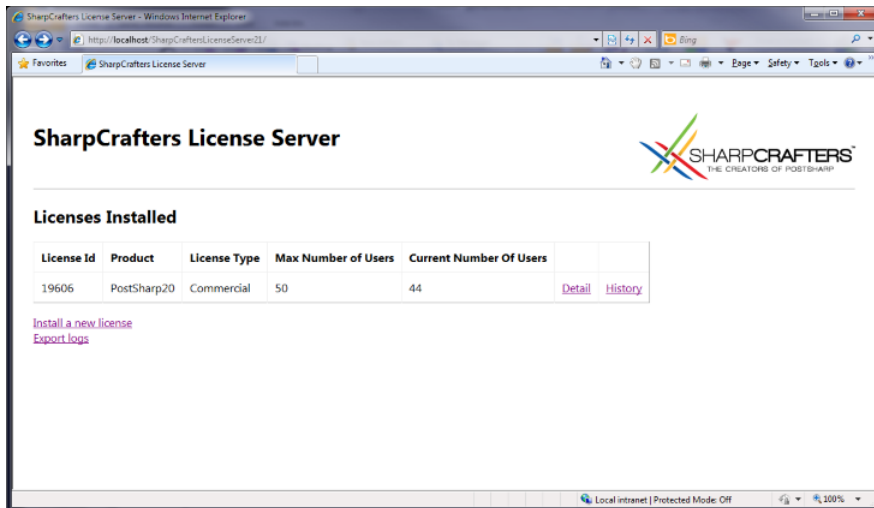
Testing the license server

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

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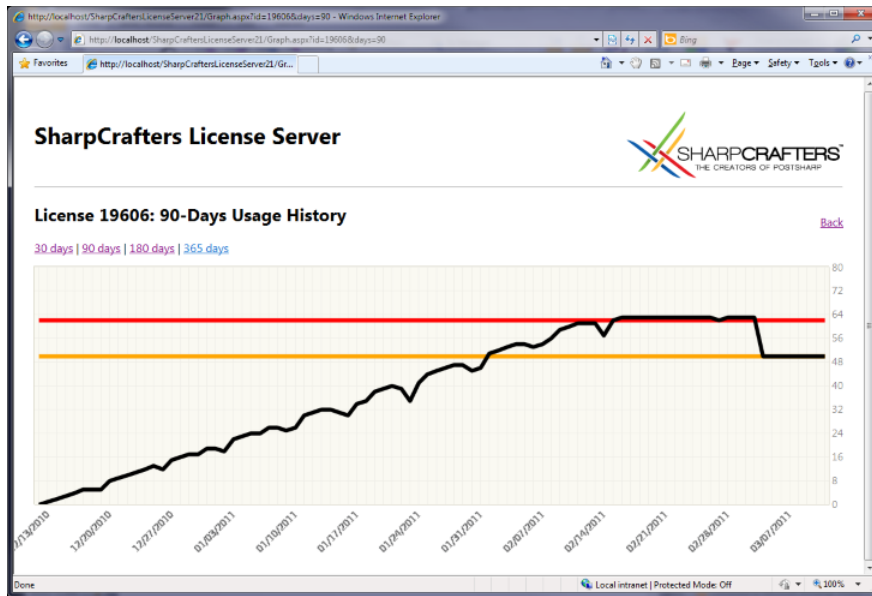
Canceling leases

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 num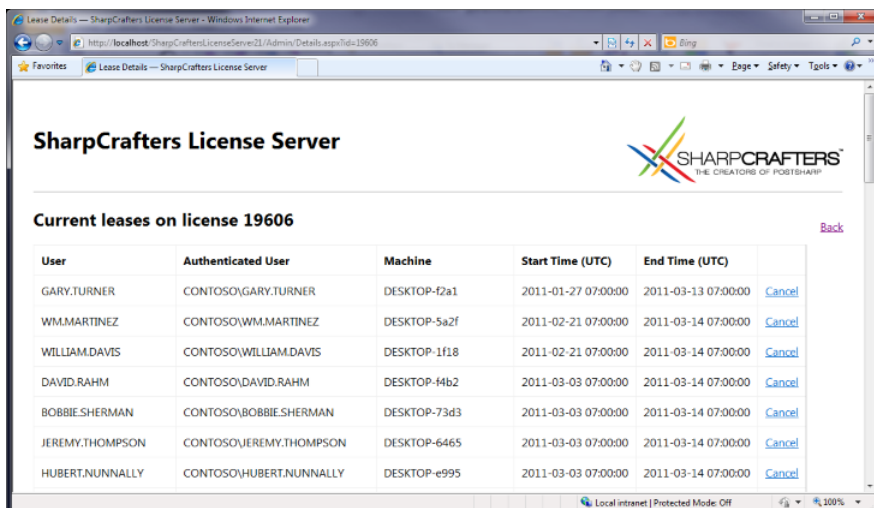ber of users and lease duration. It is safe to delete all records from the `Lease` table at any time, unless these records are necessary for accounting purposes.

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

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

4.5. 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 [47].

Installing a License on the Build Server

The License Agreement specifies that build servers don't need their own license. PostSharp will not attempt to enforce licensing if it detects that it runs in unattended mode. PostSharp uses several heuristics to detect whether it is running unattended. These heuristics include the use of `Environment.UserInteractive`, checking `Process.SessionId` (from Windows Vista, all processes running in session 0 are unattended), or checking the parent process.

If this check does not work for any reason, you may use the license key of any licensed user for the build server. This will not be considered a license infringement. However, it is better to report the issue to our technical support so that we can fix the detection algorithms.

It is recommended to include the license key in the source control. See Deploying License Keys [35] for details.

If you are using the license server, it is possible to configure it to provide license to build servers "for free", without holding a license from the pool. See the configuration options of the license server in Installing and Servicing PostSharp License Server [42] for details.

4.5.1. 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 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 PostSharp modifies the project file (*csproj* or *vbproj*, typically) to include the file *PostSharp.targets*. This 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 fail, and subsequent builds will succeed.

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

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, where *MySolution.sln* is the solution for which packages have to be restored.

Please look at the NuGet Command-Line Reference[8] 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
```

In this command, where *SolutionDirectory\packages* is the directory where the NuGet packages should be installed.

Please look at the NuGet Command-Line Reference[9] for details.

TIP

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

---

8. http://docs.nuget.org/docs/reference/command-line-reference
9. http://docs.nuget.org/docs/reference/command-line-reference

4.5.2. 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 [34] section. You will also need to configure a build definition as outlined in the Create or edit build definition[10] 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.
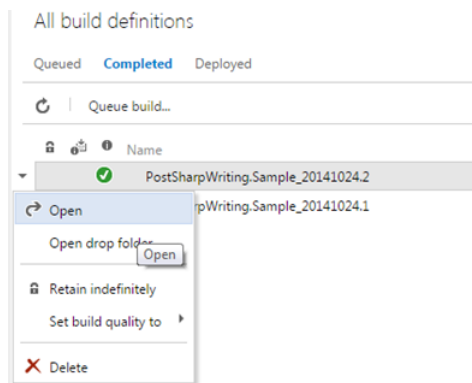
2. Select the Diagnostics tab.

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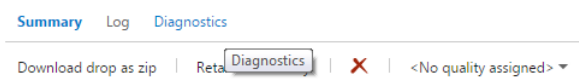
3. Ensure that the installation of PostSharp and any PostSharp patterns libraries that you used.

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.

4.6. 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.

> **TIP**
>
> Other sections of this chapter, specifically Installing PostSharp Tools for Visual Studio [34], Deploying License Keys [35] and Using PostSharp on a Build Server [46], 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 Post-Sharp.

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 http://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 `PSerializable-Attribute` 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.

4.7. 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

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 hand-written 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 give procedural advise on 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
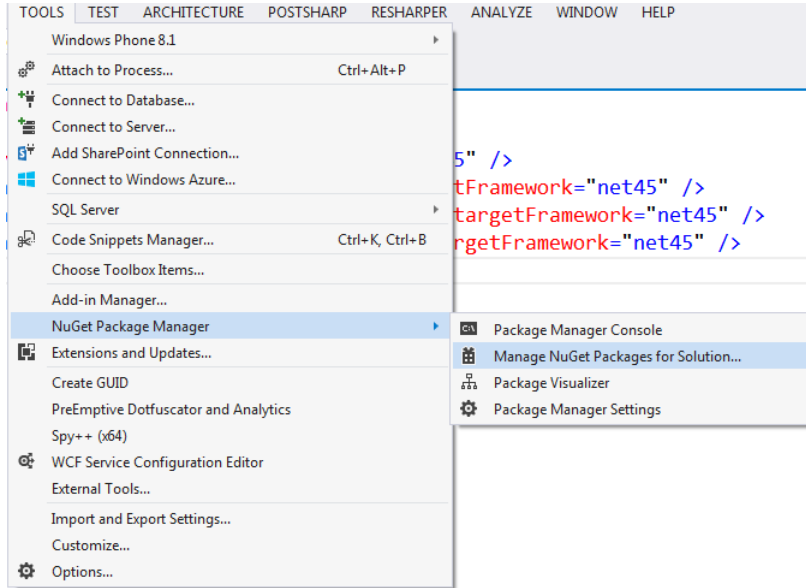
2. Select the All tab from the left side of the window.
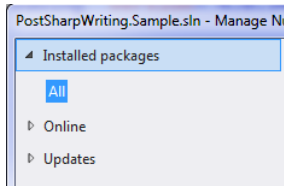
3. Find the PostSharp packages in the list and select one of the PostSharp Library packages. Click the Manage button.
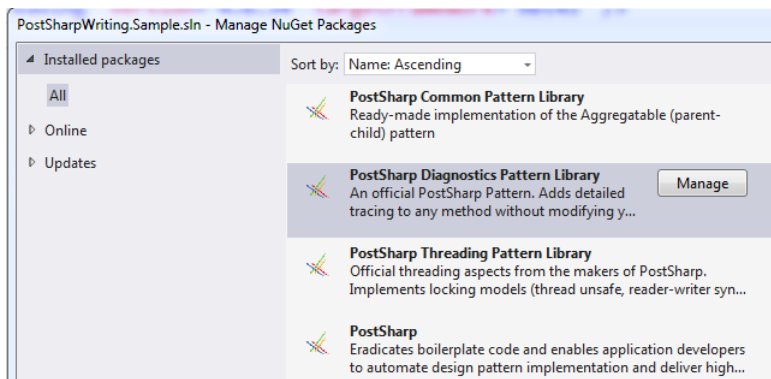
NOTE

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.

4. Ensure that you uncheck all of the projects listed in the window and click OK.
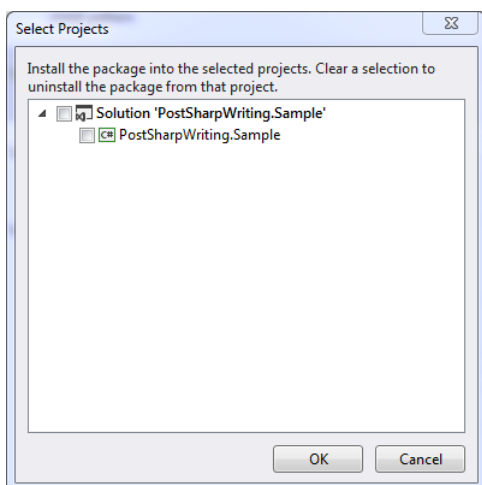
5. Repeat steps 3 and 4 for each of the PostSharp packages that show in the Packages Manager for Solution window.

6. 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 yoru 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.
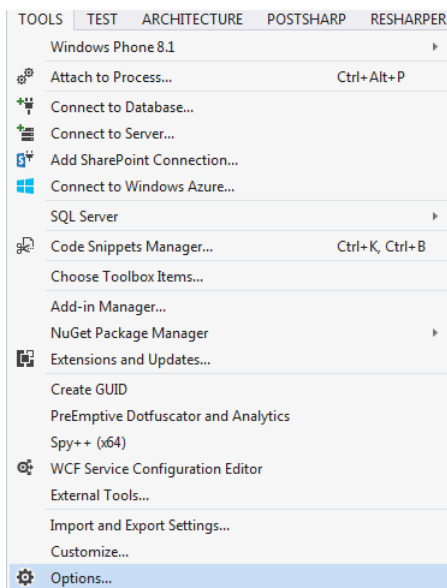
2. Expand the PostSharp node in the tree and select the Compiled Images node.
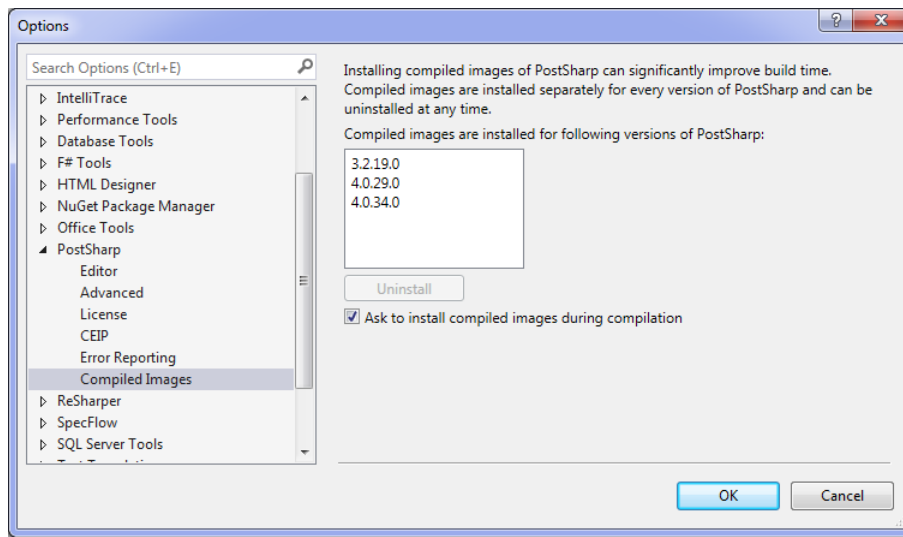
3. Select an entry in the listbox and click Uninstall.
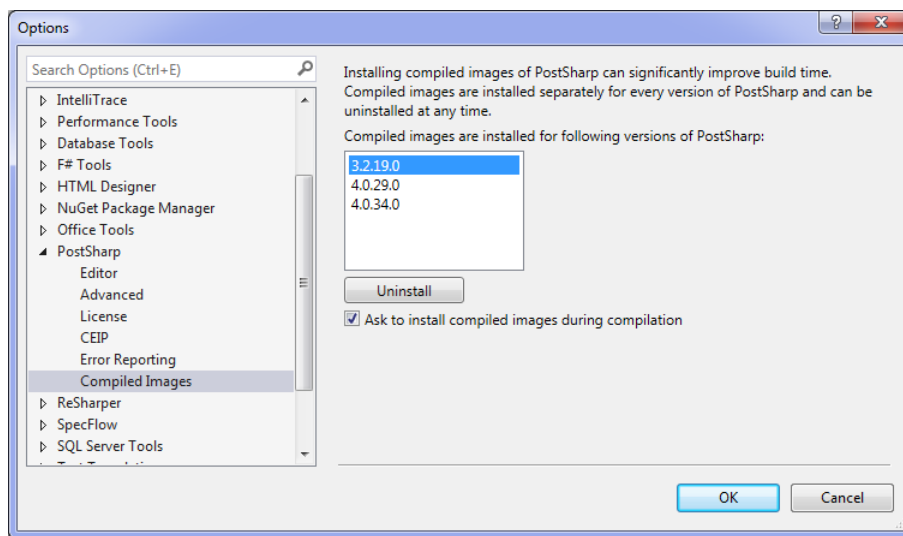
4. Follow the wizard to uninstall the compiled images for the selection you made. There are no choices to be made, simply click Next and Finish until the wizard is completed.

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.
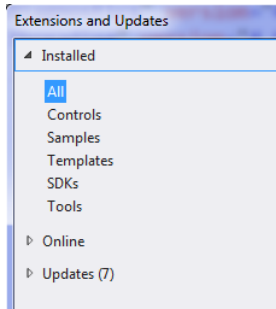
3. Find the PostSharp entry in the middle of the window, select it and click the Uninstall button.
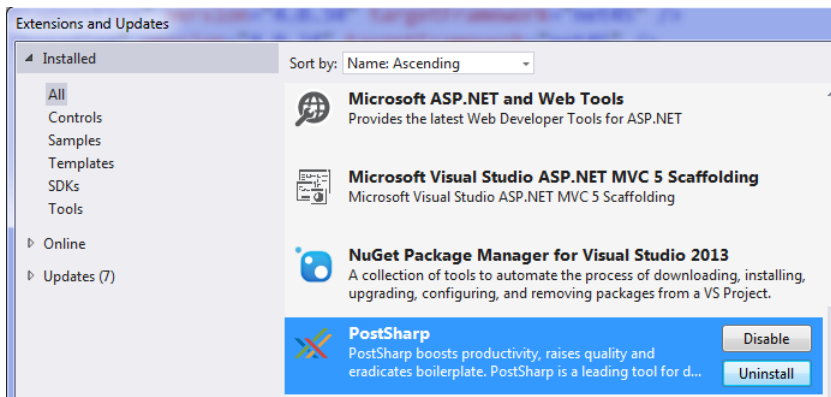
4. Click 'Yes' to confirm that you want to uninstall the PostSharp extension.
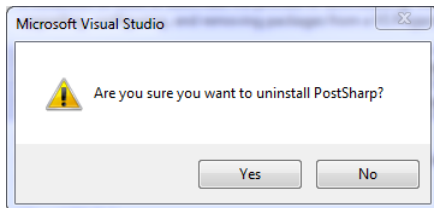
NOTE

You will need to restart Visual Studio to complete uinstalling the PostSharp extension.

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

1. Open Windows Explorer and navigate to *C:\ProgramData\PostSharp*.
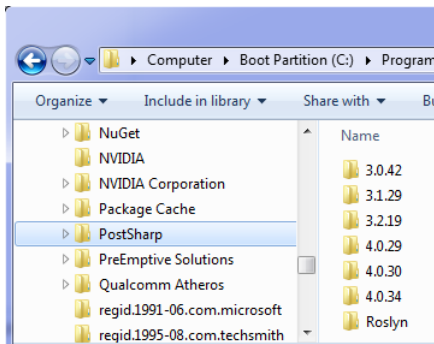
2. Select the *C:\ProgramData\PostSharp* folder and delete it.

4.8. 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. These libraries are the ones included in the *lib* sub-directory of the NuGet packages for the relevant target framework.

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.

Deployment

CHAPTER 5

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 is 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.

5.1. 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

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 Visual Studio
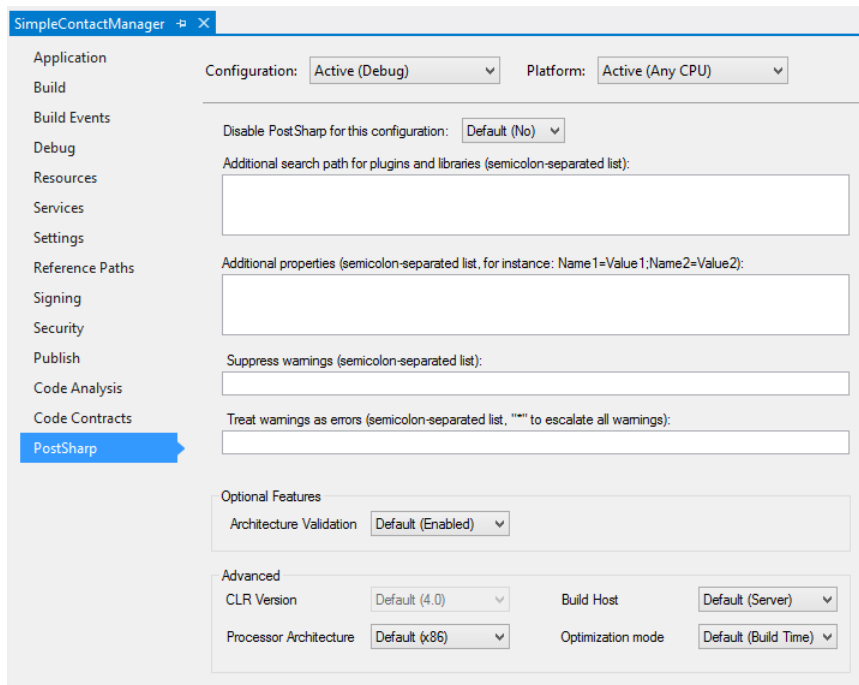
Understanding configuration settings

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: `..\MyWeaver\bin\$(Configuration)`. 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: `Name1=Value1;Name2=Value2`. This property can reference other MSBuild properties, for instance: `RootNamespace=$(RootNamespace)`. Several properties are defined by default; for details, see Well-Known PostSharp Properties [69]. This setting maps to the MSBuild property PostSharpProperties
Suppress warnings	A semicolon-separated list of warning identifiers that must be ignored, or * 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 * 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 [319].
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 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 PostSharpTargetProcessor.
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 PostSharpHost. For details, see Configuring Projects Using MSBuild [61].
Optimization Mode	When set to Build Time, PostSharp will use a faster algorithm to generate the final assembly. When set to Size, PostSharp will use a slower algorithm that generates smaller assemblies. The default value is Build Time by default, or Size when the C# or Visual Basic compiler is set to generate optimal code (typically, in release builds). This settings maps to the MSBuild property PostSharpOptimizationMode.

5.2. 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 [61]
- Configuring several projects at a time [62]
- Setting MSBuild properties from the command line [62]
- List of properties [62]

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 Visual Basic 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:

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

   See Configuring Projects Using MSBuild [61] 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[11].

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:

```xml
<?xml version="1.0" encoding="utf-8"?>
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <PropertyGroup>
    <PropertyName>PropertyValue</PropertyName>
  </PropertyGroup>
</Project>
```

See Configuring Projects Using MSBuild [61] for the list of MSBuild properties used by PostSharp.

> **NOTE**
>
> For more information regarding MSBuild project files, see MSDN Documentation[12].

Setting MSBuild properties from the command line

When an MSBuild property does not need to be set permanently, it is convenient to set is 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).

---

11. http://msdn.microsoft.com/en-us/library/vstudio/ms171458.aspx
12. http://msdn.microsoft.com/en-us/library/vstudio/dd637714.aspx

Property Name	Description
`SkipPostSharp`	True if PostSharp should not be executed.
`PostSharpOptimizationMode`	When set to OptimizeForBuildTime, PostSharp will use a faster algorithm to generate the final assembly. The other possible value is OptimizeForSize. The default value of the `PostSharpOptimizationMode` property is OptimizeForBuildTime by default and OptimizeForSize when the C# or Visual Basic compiler is set to generate optimal code (typically, in release builds).
`PostSharpDisabledMessages`	Comma-separated list of warnings and messages that should be ignored.
`PostSharpEscalatedMessages`	Comma-separated list of warnings that should be escalated to errors. Use * to escalate all warnings.
`PostSharpLicense`	License key or URL of the license server.
`PostSharpProperties`	Additional properties passed to the PostSharp project, in format `Name1=Value1;Name2=Value2`. See Well-Known PostSharp Properties [69].
`PostSharpConstraintVerificationEnabled`	Determines whether verification of architecture constraints is enabled. The default value is `True`.

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 to influence the choice of the PostSharp host process.

Property Name	Description
`PostSharpTargetFrameworkVersion`	Version of the CLR that hosts the PostSharp process. The only valid value is 4.0.
`PostSharpTargetProcessor`	Processor architecture of the PostSharp hosting process. Valid values are x86 and x64. Since PostSharp needs to execute the current project during 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.
`PostSharpHost`	Kind of process hosting PostSharp. This setting is usually changed for troubleshooting only. The following values are supported: • PipeServer 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. • Native 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. • Managed 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 `app.config` or `web.config`.

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 Attaching a Debugger at Build Time [316].
PostSharpTrace	A semicolon-separated list of trace categories to be enabled. The property is effective only when PostSharp runs in diagnostic mode (see property `PostSharpBuild` here above). Additionally, the MSBuild verbosity should be set to detailed at least. For details, see Attaching a Debugger at Build Time [316].
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 `ExitCode` output property of the `PostSharp` MSBuild task to take action.
PostSharpFailOnUnexpectedMessage	This property should be used jointly with `PostSharpExpectedMessages`. 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 (*.psproj) 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 PostSharp.dll 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 output of PostSharp. By default, the analysis is done on the input 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 *obj\Debug\Before-PostSharp* 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 ExecuteCodeAnalysisOnPostSharpOutput property has been set to True.

5.3. 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 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 [65]
- Sharing configuration between projects [66]
- Order of processing of configuration files [66]

PostSharp configuration files

The configuration system of PostSharp is based on configuration files.

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

See Configuration File Schema Reference [67]for details about the format of this file.

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

```xml
<Project xmlns="http://schemas.postsharp.org/1.0/configuration">

  <Property Name="LoggingBackend" Value="nlog"/>

  <Using File="..\..\Build\bin\{$Configuration}\PostSharp.Patterns.Diagnostics.Weaver.dll"/>
  <Using File="..\..\Build\bin\{$Configuration}\PostSharp.Patterns.Diagnostics.Weaver.NLog.dll"/>

  <Multicast>
    <LogAttribute xmlns="clr-namespace:PostSharp.Patterns.Diagnostics;assembly:PostSharp.Patterns.Diagnostics" AttributeTargetTy
  </Multicast>

</Project>
```

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

- Adding license keys. See the License [67] configuration element for details.

    Configuring properties. See the Property [67] configuration element for details.

    Including a plug-in. See the Using [67] configuration element for details.

    Adding aspects or constraints without modifying source code. See Adding Aspects Using XML [162] for details.

    Editing logging profiles (TODO).

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 [67] 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 `My-Project.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 [67] 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 [67] elements are loaded.

2. Property [67] elements are loaded. Properties are evaluated at this moment, unless they are marked for deferred evaluation.

3. SearchPath [67] elements are loaded.

4. Using [67] elements are loaded and referenced plug-ins and configuration files are immediately loaded.

5. SectionType [67] elements are loaded.

6. Service [67] 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.

5.3.1. Configuration File Schema Reference

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

```xml
<Projectxmlns="http://schemas.postsharp.org/1.0/configuration"x:xmlns="http://schemas.postsharp.org/1.0/configuration">

  <!-- The following elements must appear in the proper order. -->

  <LicenseValue="[<license-key>|<url>](;[<license-key>|<url>])*"/>

  <SearchPathDirectory="<expression:string>(;<expression:string>)*"ReferenceDirectory="<expression:string>"Condition="<express
  <SearchPathFile="<expression:string>(;<expression:string>)*"ReferenceDirectory="<expression:string>"Condition="<expression:b

  <UsingFile="<expression:string>"ProjectName="<expression:string>"Condition="<expression:bool>"/>

  <SectionTypeLocalName="<string>"Namespace="<string>"/>

  <PropertyName="<string>"Value="<expression:string>"Overwrite="[true|false]"Sealed="[true|false]"Deferred="[true|false]"Condi

  <ServiceTypeName="<string>"AssemblyFile="<string>"Condition="<expression:bool>"/>

  <!-- The rest of the file contains extension elements defined using <SectionType /> elements.
       PostSharp itself defines the following extension elements: -->

  <Multicast>
    <MyMulticastAspectMyAttributeName="<value>"xmlns="clr-namespace:<namespace>;assembly:<assembly>"x:Condition="<expression:b
  </Multicast>

  <LoggingProfilesxmlns="clr-namespace:PostSharp.Patterns.Diagnostics;assembly:PostSharp.Patterns.Diagnostics"x:Condition="<ex
    <LoggingProfileName="<string>"OnEntryLevel="<LogLevel>"OnSuccessLevel="<LogLevel>"OnExceptionLevel="<LogLevel>"OnEntryOpti
  </LoggingProfiles>

</Project>
```

Schema elements

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

Project

This element is the root of the configuration file.

License

This element allows to load 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
File	string expression	Optional (either File or Directory is required). A semicolon-separated list of files that must be added to the path.
Directory	string expression	Optional (either File or Directory is required). A semicolon-separated list of directories that must be added to the path.
ReferenceDirectory	string expression	Optional. The directory from which relative paths in File or Directory are resolved.
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 *psplugin* or *dll* will be searched.
ProjectName	string	Optional. In case that a single *dll* includes several configurations, specifies which configuration should be loaded.
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.

Attribute	Type	Description
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 `IService` 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 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 [70].

5.3.2. 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.

Property Name	MSBuild Property Name	Description
`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`.

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.

5.3.3. Including CLR Objects in Configuration

PostSharp includes a basic facility to describe CLR objects using XML. This facility is used to implement the Multicast [67] and LoggingProfiles [67] sections of the configuration file, and can be used to define custom sections.

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

This topic contains the following sections.

- Basic rules [70]
- Formatting of attributes [70]
- Specific rules for the Multicast section [71]

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

Specific rules for the Multicast section

The following additional rules apply to the Multicast [67] 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.

5.3.4. 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:

```
<Project xmlns="http://schemas.postsharp.org/1.0/configuration">
  <Property Name="LoggingEnabled" Value="{has-plugin('PostSharp.Patterns.Diagnostics')}" Deferred="true"/>
  <Multicast>
    <When Condition="{$LoggingEnabled}">
      <d:Log/>
    </When>
  </Multicast>
</Project>
```

Please check the MSDN documentation[13] 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 o the value of the *LoggingEnabled* property.

---

13. http://msdn.microsoft.com/en-us/library/ms256138(v=vs.110).aspx

Using operators and functions

You can use any XPath function and operators.

Additionally to standard XPath 1.0 functions[14], PostSharp defines the following functions:

Function	Description
has-plugin(name)	Evaluates to true if the given plug-in is loaded, otherwise false.
environment(variable)	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:

```
<Project xmlns="http://schemas.postsharp.org/1.0/configuration">
  <Property Name="A" Value="A"/>            <!-- Evaluates to A -->
  <Property Name="B" Value="B;{$A}"/>       <!-- Evaluates to B;A -->
  <Property Name="C" Value="C;{$B};{$A}"/>  <!-- Evaluates to C;B;A;A -->
</Project>
```

5.4. 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

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:

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

For details regarding expressions, see Using Expressions in Configuration Files [71].

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 [65] element.

---

14. http://msdn.microsoft.com/en-us/library/ms256138(v=vs.110).aspx

5.5. 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 [73]
- Emitting Errors, Warnings, and Messages [74].

TIP

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

5.5.1. 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 [59] for details.
- By defining the `PostSharpDisabledMessages` or `PostSharpEscalatedMessages` MSBuild properties. See Configuration [59] and Configuring Projects Using MSBuild [61] for details.
- By using the **DisablePostSharpMessageAttribute** or `EscalatePostSharpMessageAttribute` custom attribute at assembly level. This approach is considered obsolete.

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).

To ignore warnings emitted by constraints, it is preferable to use the **IgnoreConstraintWarningAttribute** custom attribute. Indeed, this attribute is conditional to the compilation symbol POSTSHARP_CONSTRAINTS, so it will be ignored by the compiler unless constraint verification is enabled for the current project and build configuration.

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

5.5.2. 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 back-end of a message dispenser, comes with a non-negligible performance penalty because of the cost of instantiating the `ResourceManager`.

Standard Patterns

CHAPTER 6

INotifyPropertyChanged

Binding objects to the UI is a large and tedious task. You must implement `INotifyPropertyChanged` on every property that needs to be bound. You need to ensure that the underlying property setter correctly raises events so that the View knows that changes have occurred. The larger your codebase, the more work there is. You can partially eliminate all of this repetitive code by pushing some of the functionality to a base class that each Model class inherits from. It still doesn't eliminate all of the repetition though.

PostSharp can completely eliminate all of that repetition for you. All you have to do is make use of the Model Pattern Library's `NotifyPropertyChangedAttribute` aspect.

In this chapter

Topic	Description
	This section shows how to automatically implement the `INotifyPropertyChanged` aspect in a class thanks to the `NotifyPropertyChangedAttribute` aspect.
	This section describes how to handle dependencies that cross several objects, as when a view-model object is dependent on properties of a model object.
	This section documents how to cope with the cases that cannot be automatically handled by the `NotifyPropertyChangedAttribute` aspect.
	This section describes the principles and concepts on which the `NotifyPropertyChangedAttribute` aspect relies.

6.1. Walkthrough: Automatically Implementing INotifyPropertyChanged

This section shows how to make your class automatically implements the `INotifyPropertyChanged` interface `Notify-PropertyChangedAttribute` aspect.

Let's start with a simple class that has two simple properties and one composite property:

```csharp
public class CustomerForEditing
{
    public string FirstName { get; set; }
    public string LastName { get; set; }

    public string FullName
    {
        get { return string.Format("{0} {1}", this.FirstName, this.LastName);}
```

```
    }
}
```

This topic contains the following sections.

- Adding the NotifyPropertyChanged aspect with the UI [78]
- Adding the NotifyPropertyChanged aspect manually [80]
- See Also [0]

Adding the NotifyPropertyChanged aspect with the UI

To add INotifyPropertyChanged aspect with the UI:

1. Put the caret on the class name and expand the Smart Tag. From the list select "Implement INotifyProperty-Changed".
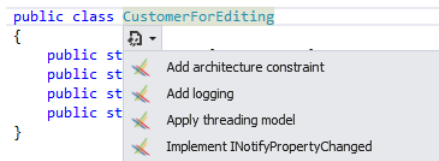
2. If you haven't previously added the Model Pattern Library to the current project, PostSharp will inform you that it will be doing this as well as adding an attribute to the target class.
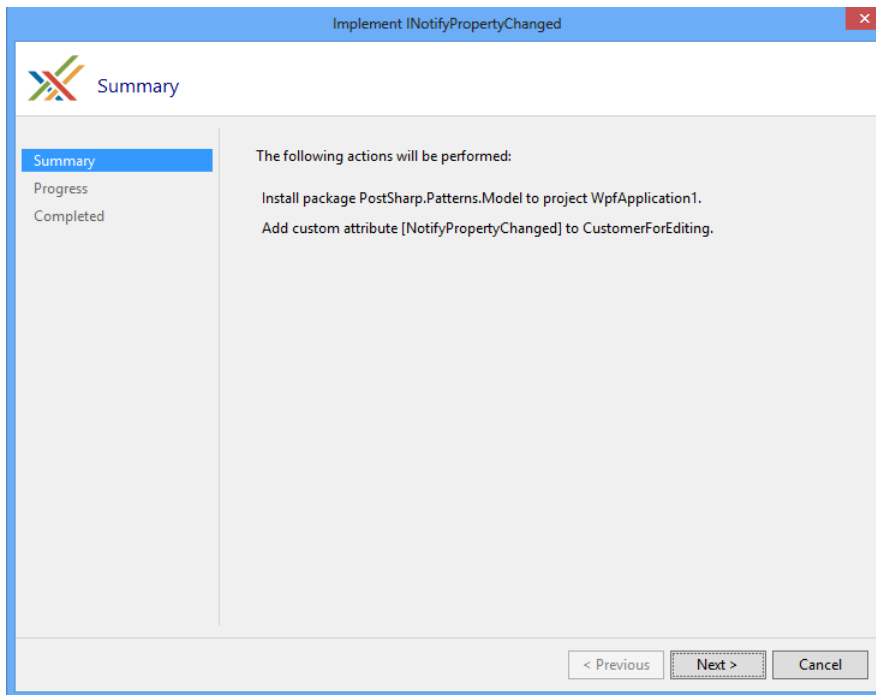
3. PostSharp will download the Model Pattern Library and add the attribute.
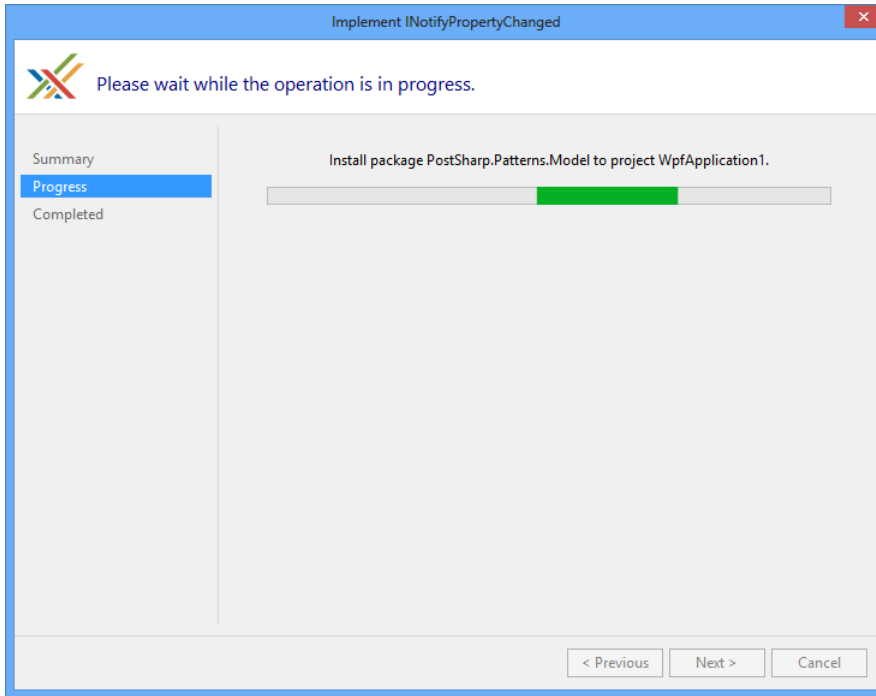
4. Once the download, installation and configuration of the Model Pattern Library and the addition of the attribute has finished you can close the wizard and look at the changes that were made to your codebase.
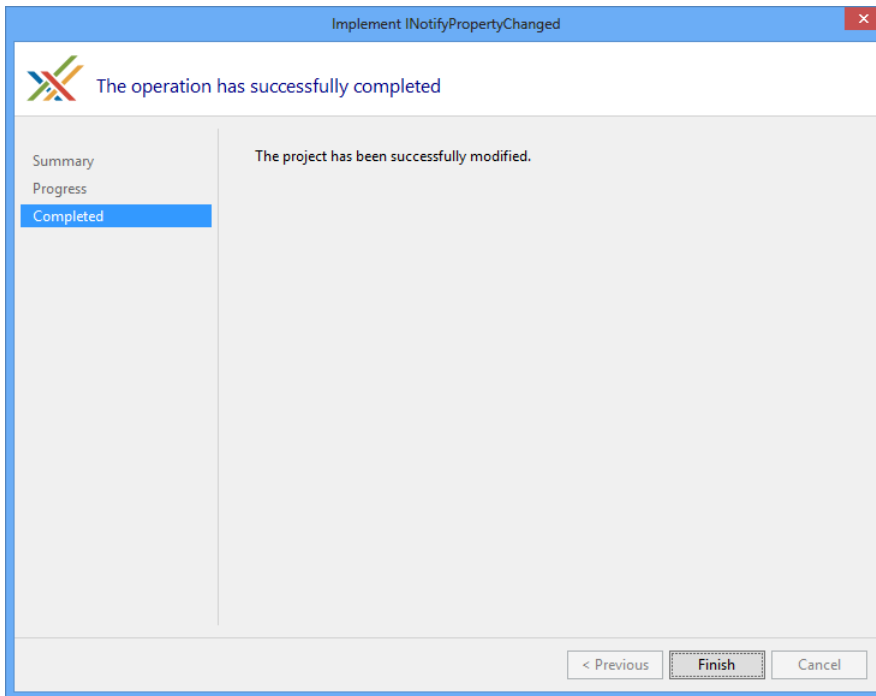
5. You'll notice that the code you added `NotifyPropertyChangedAttribute` to has only been slightly modified. PostSharp has added a `NotifyPropertyChangedAttribute` attribute to the class. This class level attribute will add the implementation of `NotifyPropertyChangedAttribute` to the class as well as the plumbing code in each property that makes it work.

```
[NotifyPropertyChanged]
public class CustomerForEditing
{
    public string FirstName { get; set; }
    public string LastName { get; set; }

    public string FullName
    {
        get { return string.Format("{0} {1}", this.FirstName, this.LastName); }
    }

}
```

> **NOTE**
>
> This example has added `NotifyPropertyChangedAttribute` to one class. If you need to implement `Notify-PropertyChangedAttribute` to many different classes in your codebase you will want to read about using aspect multicasting. See the section Adding Aspects to Multiple Declarations [147].

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.

Adding the NotifyPropertyChanged aspect manually

The wizard does nothing more than installing a NuGet package and adding a custom attribute. You can achieve the same manually.

To add INotifyPropertyChanged aspect manually:

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.

6.2. Walkthrough: 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's Model Pattern Library easily handles transitive dependencies. Simply add the `NotifyPropertyChanged-Attribute` class attribute to each class in the dependency chain. This will ensure that property change notifications are propagated up and down the dependency chain. The Model Pattern Library takes care of the rest and will even handle circular dependencies.

In the following set of steps, 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 `Last-Name` properties.

1. Add the `CustomerModel` class to your project ensuring that the `NotifyPropertyChangedAttribute` attribute is included:

```
[NotifyPropertyChanged]
public class CustomerModel
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string Phone { get; set; }
    public string Mobile { get; set; }
    public string Email { get; set; }
}
```

2. Setup a view-model class which contains a reference to a `CustomerModel` object, add properties to get/set the name related fields. References to properties of the `CustomersForEditing` object should be in the form of `this.field.Property` (or `this.Property.Property`), otherwise PostSharp won't be able to discover the dependencies from your source code.

```
class CustomerViewModel
{
    Customer model;

    public CustomerViewModel(Customer m)
    {
        this.model = m;
    }

    public string FirstName { get { return this.model.FirstName; } set { this.model.FirstName = value;}}

    public string LastName { get { return this.model.LastName; } set { this.model.LastName = value; }}

}
```

3. Add the `FullName` property and use the same rule as described in the previous step to reference dependent properties:

```
class CustomerViewModel
{
    Customer model;

    public CustomerViewModel(Customer m)
    {
        this.model = m;
    }
    public string FirstName { get { return this.model.FirstName; } set { this.model.FirstName = value;}}
    public string LastName { get { return this.model.LastName; } set { this.model.LastName = value; }}

    public string FullName { get {
      return string.Format("{0} {1}", this.model.FirstName, this.model.LastName);
    } }

  }
```

4. Add the `NotifyPropertyChangedAttribute` attribute to the class:

```
[NotifyPropertyChanged]
class CustomerViewModel
{
    Customer model;

    public CustomerViewModel(Customer m)
    {
        this.model = m;
    }

    public string FirstName { get { return this.model.FirstName; } set { this.model.FirstName = value;}}
    public string LastName { get { return this.model.LastName; } set { this.model.LastName = value; }}

    public string FullName { get {
      return string.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.

NOTE

Read the article Customizing the NotifyPropertyChanged Aspect [82] to learn about referencing properties without using the `this.field.Property` form.

6.3. Customizing the NotifyPropertyChanged Aspect

Postsharp includes a number of attributes for customizing the Model Pattern's behaviour and for handling special dependencies.

This topic contains the following sections.

- Ignoring Changes to Properties [82]
- Handling Virtual Calls, References, and Delegates in a Get Accessor [83]
- Handling Local Variables [84]
- Handling Dependencies on Pure Methods [85]

Ignoring Changes to Properties

Use the `IgnoreAutoChangeNotificationAttribute` class attribute to prevent an `OnPropertyChanged` event from being invoked when setting a property. For example, the `CustomerModel` class contains a `Country` property amongst others:

```
[NotifyPropertyChanged]
public class CustomerModel
{
    public string FirstName { get; set; }
    public string LastName { get; set; }

    public string Phone { get; set; }
    public string Mobile { get; set; }
    public string Email { get; set; }
    public string Country { get; set;}
}
```

To prevent a property notification from being invoked when the `Country`'s value is set, simply place the `IgnoreAuto-ChangeNotificationAttribute` attribute above the property:

```
[NotifyPropertyChanged]
public class CustomerModel
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string Phone { get; set; }
    public string Mobile { get; set; }
    public string Email { get; set; }

    [IgnoreAutoChangeNotification]
    public string Country { get; set;}
}
```

Handling Virtual Calls, References, and Delegates in a Get Accessor

If a get accessor 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. To suppress this error, you can add the [`SafeForDependencyAnalysisAttribute`] custom attribute to the property accessor (or in any method used by the property accessor). This custom attribute instructs PostSharp that the property accessor is "safe" – in other words, it contains only dependencies in the canonical form `this.field.Property`.

For example, say `CustomerModel` contains a virtual method called `ValidateCountry()` which is used by the get accessor of its `Country` property:

```
[NotifyPropertyChanged]
public class CustomerModel
{
  // Details skipped.

protected virtual bool ValidateCountry(string s)
{
  if (s!=null)
    return true;
  else
    return false;
}

  public string Country
  {
    get
    {
      if(this.ValidateCountry(value))
        return value;
      else
        return null;
    }
    set;
  }
}
```

In this situation the property relies on a virtual method which PostSharp cannot resolve at build time, so the `SafeForDependencyAnalysisAttribute` attribute can be placed on the `Country` property suppress this error:

```
[NotifyPropertyChanged]
public class CustomerModel
{
    // Details skipped.

  public virtual bool Test(string s)
  {
    if (s!=null)
      return true;
```

```
    else
      return false;
  }

  [SafeForDependencyAnalysisAttribute]
  public string Country
  {
    get
    {
      if(this.test(value) == true)
        return value;
      else
       return null;
    }
    set;
  }
}
```

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 `On` construct (see the next section). 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.

Handling Local Variables

Properties may depend on a property of another object, and sometimes this object must be stored in a local variable. PostSharp is not able to analyze chains of dependencies in properties that are dependent on a property of a local variable.

For example, consider the following version of `CustomerModel` which contains properties for primary and secondary contact phone numbers, each of which is of type `Contact`, as well a string property called `ValidPhoneNumber` which attempts to return a non-null phone number:

```
public class Contact
{
  public string Phone {get; set;}
}

[NotifyPropertyChanged]
public class CustomerModel
{
  public Contact PrimaryContact{get; set;}
  public Contact SecondaryContact{get; set;}

  public string ValidPhoneNumber
  {
    Contact contact = null;
    if(PrimaryContact != null)
      contact = PrimaryContact;
    else
      contact = SecondaryContact;

    if(contact != null)
      return contact.Phone;
    else
      return null;
  }
}
```

In this situation the local variable `contact` cannot be analyzed, so the dependency must be explicitly specified using the `DependsOn` method:

```
[NotifyPropertyChanged]
public class CustomerModel
{
  public Contact PrimaryContact{get; set;}
  public Contact SecondaryContact{get; set;}

  [SafeForDependencyAnalysis]
  public string ValidPhoneNumber
  {
    Depends.On(this.PrimaryContact.Phone, this.SecondaryContact.Phone);
    Contact contact = null;
    if(PrimaryContact != null)
      contact = PrimaryContact;
    else
      contact = SecondaryContact;

    if(contact != null)
      return contact.Phone;
    else
      return null;
  }
}
```

NOTE

The `SafeForDependencyAnalysisAttribute` attribute is still required in order to suppress the error about the dependency on a local variable.

Handling Dependencies on Pure Methods

Often times an object will depend on a method which is solely dependent on its input parameters to produce an output (e. g. a static method). 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`:

```
public class ContactHelper
{
  [Pure]
  public static string GetValidPhoneNumber(string firstPhoneNumber, string secondPhoneNumber)
  {
    if(firstPhoneNumber != null)
      return firstPhoneNumber;
    else if (secondPhoneNumber != null)
      return secondPhoneNumber;
    else
      return null;
  }
}

[NotifyPropertyChanged]
public class CustomerModel
{
  public Contact PrimaryContact{get; set;}
  public Contact SecondaryContact{get; set;}

  public string ValidPhoneNumber
  {
    get {
      return ContactHelper.GetValidPhoneNumber(this.PrimaryContact.Phone, this.SecondaryContact.Phone);
    }
  }
}
```

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.

6.4. 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

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 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 runtime 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 hand-written 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 `INotifyChildProperty-Changed`. 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 `Pure-Attribute` custom attribute, or unless the method is a property getter in a supported dependency chain.

- Valuations of properties of local variables or method return values are not supported. Only properties of fields or properties are supported.

See Customizing the NotifyPropertyChanged Aspect [82] 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
{
  public decimal Amount { get; private set; }
  public decimal Tax { get; private set; }
  public decimal Total { get; private set; }

  public void 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 exists 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-PropertyChangedServicesResumeEvents` methods.

Remarks

The `NotifyPropertyChangedAttribute` aspect never evaluates property getters at runtime. 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.

INotifyPropertyChanged

CHAPTER 7
Parent/Child Relationships

The parent-child relationship is a foundational concept of object oriented design. There are three kinds of object relationships in the UML specification:

- *Aggregation* is the parent-child (also named whole-part) relationship. It is implemented in PostSharp by the `AggregatableAttribute` aspect described in Walkthrough: Annotating an Object Model for Parent-Child Relationships [92].

- *Composition* is an aggregation relationship where the parent controls the lifetime their children. It is implemented in PostSharp by the `DisposableAttribute` aspect pattern, which relies on the `AggregatableAttribute` aspect. For details, see Walkthrough: Automatically Disposing Children Objects [96].

- *Association* is a simple reference between two objects.

Despite its importance, C# and VB have no keyword to represent aggregation. All C# and VB object references correspond to an association. Therefore, most applications and frameworks tend to re-implement the aggregation relationship, resulting in boilerplate code and defects. For instance, UI frameworks such as WinForms and WPF rely on a parent-child structure.

PostSharp implements the Aggregatable pattern thanks to the `AggregatableAttribute` aspect, together with the `ChildAttribute`, `ReferenceAttribute` and `ParentAttribute` custom attributes.

The Aggregatable pattern is used by other PostSharp aspects, including all threading models (`ThreadAwareAttribute`), `DisposableAttribute` and `RecordableAttribute`. You can also use the aspect to automatically implement a parent-child relationship in your own code.

In this chapter

Section	Description
Walkthrough: Annotating an Object Model for Parent-Child Relationships [92]	This section shows how to prepare a class so that it can participate in a parent-child relationship.
Walkthrough: Enumerating Child Objects [95]	This section describes how to enumerate the children of an object thanks to the visitor pattern.
Walkthrough: Automatically Disposing Children Objects [96]	This section shows how to automatically implement the `IDisposable` interface so that children objects are disposed when the parent object is disposed.
Working With Collections [100]	This section covers advanced topics related to collections in aggregatable object models.

7.1. Walkthrough: Annotating an Object Model for Parent-Child Relationships

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.

This topic contains the following sections.

- Applying through the UI [92]
- Applying manually [93]

Applying through the UI

When applying any of the threading model patterns using the wizard you may encounter the Relationships page.
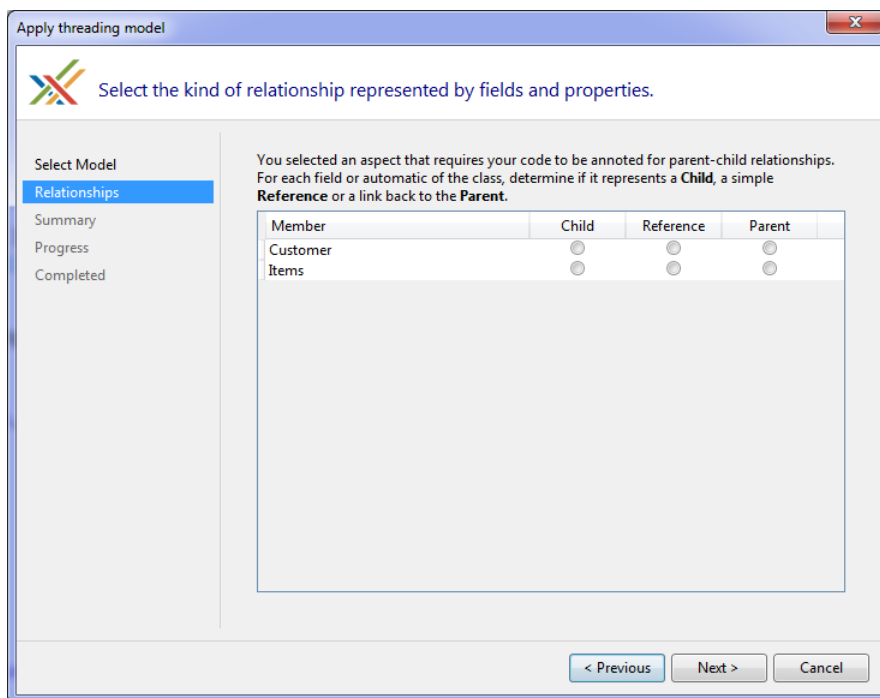
This page provides you with the ability to establish the relationships between objects in an object tree. Simply select the option desired for each complex object and the wizard will add these attributes for you.

> **NOTE**
>
> The wizard process will not address or change collections in the object parent-child relationships. You will need to change collection types by hand.

Applying manually

To apply the Aggregatable to an object model:

1. Add the `AggregatableAttribute` attribute to the parent and children classes. 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**
>
> 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).

```csharp
[Aggregatable]
public class Invoice
{
  public Invoice()
  {
    this.Lines = new List<InvoiceLine>();
  }

    public Customer Customer { get; set; }
    public IList<InvoiceLine> Lines { get; set; }
  public Address DeliveryAddress { get; set; }
}

[Aggregatable]
public class InvoiceLine
{
  private Product product;
  public decimal Amount { get; set; }
}

[Aggregatable]
public class Address
{
}
```

2. Annotate fields and automatic properties of all aggregatable classes with the `ChildAttribute` or `Reference-Attribute` custom attribute. Fields or properties of a value type must not be annotated.

```csharp
[Aggregatable]
public class Invoice
{
  public Invoice()
  {
      this.Lines = new List<InvoiceLine>();
  }

    [Reference]
    public Customer Customer { get; set; }

    [Child]
    public IList<InvoiceLine> Lines { get; private set; }

  [Child]
  public Address DeliveryAddress { get; set; }
}

[Aggregatable]
public class InvoiceLine
{
  [Reference]
  private Product product;

  public decimal Amount { get; set; }
}

[Aggregatable]
public class Address
{
}
```

3. Modify the code to use `AdvisableCollectionT` and `AdvisableDictionaryTKey, TValue` 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.

   In our code example, we need to modify the constructor of the `Invoice` class and assign an `Advisable-CollectionT` to the `Lines` field instead of a `List`.

```csharp
public Invoice()
{
    this.Lines = new AdvisableCollection<InvoiceLine>();
}
```

4. 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.

   In this example, we are adding a `ParentInvoice` property to the `InvoiceLine` class to link back to the `Invoice` class.

   ```
   [Aggregatable]
   public class InvoiceLine
   {
     [Reference]
     private Product product;

     public decimal Amount { get; set; }

     [Parent]
     public Invoice ParentInvoice { get; private set; }
   }
   ```

   > **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.

7.2. Walkthrough: Enumerating Child Objects

After you have declared the structure of your object graph [92], 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 `CastSourceType, TargetType(SourceType)` 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 `Advisable-DictionaryTKey, TValue` you will not be able to use the cast operator or the `CastSourceType, Target-Type(SourceType)` method. Instead, you will have to use the `QueryInterfaceT(Object, Boolean)` extension method.

2. Invoke the **VisitChildren(ChildVisitor, ChildVisitorOptions)** method and pass a delegate to the method to be executed.

```
var invoice = new Invoice();
IAggregatable aggregatable = invoice.QueryInterface<IAggregatable>();
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.

7.3. Walkthrough: Automatically Disposing Children Objects

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 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 **AggregatableAttribue** aspect and, as a result, is able to make use of the **VisitChildren(ChildVisitor, ChildVisitorOptions)** method to cascade disposals through child objects.

This topic contains the following sections.

Disposing of object graphs

Adding Disposable to an object

1. On the top level object add the `DisposableAttribute`.

```csharp
[Disposable]
public class HomeMadeLogger
{
  private TextWriter _textWriter;
  private Stream _stream;
  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));
  }
}
```

At this point the `IDisposable` interface will be implemented on the `HomeMadeLogger` class. Now any code making use of `HomeMadeLogger` is able to execute the `HomeMadeLogger.Dispose()` method.

2. Identify the fields and properties that represents references to child objects. To do this, add the `ChildAttribute` to the fields and properties that expose those child objects, and the `ParentAttribute` to fields and properties that represent plain references. In the case of the `HomeMadeLogger` there are two objects, the `_stream` and the `_textWriter` fields, which should also be disposed when the `HomeMadeLogger` is disposed.

```
[Disposable]
public class HomeMadeLogger
{
  [Child]
  private TextWriter _textWriter;
  [Child]
  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. 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.

NOTE

Fields that are marked as children but are assigned to a object that does not implement `IDisposable` (either manually or through `DisposableAttribute`) will simply be ignored during disposal.

Disposing of child collections

Child objects that need to be disposed of don't always exist in a one-to-one relationship with the parent object. It's common to see collections of child objects that need disposal as well. In that case you need to dispose of each entry in a collection.

Adding Disposable to collections

1. Use a collection type that supports the `IAggregatable` interface. We do this by making use of `Advisable-CollectionT` or `AdvisableDictionaryTKey, TValue` class.

```
[Disposable]
public class HomeMadeLogger
{
    [Child]
    public AdvisableCollection<Context> LoggingContexts { get; set; }
}
```

2. To make the `LoggingContexts` collection automatically dispose when the `HomeMadeLogger` object disposes you need to ensure that the `Contexct` class has the `DisposableAttribute` aspect on it.

```
[Disposable]
public class Context
{
    //...
}
```

With that, the `HomeMadeLogging` class and all of the child objects in the LoggingContexts collection are hooked together and they will all be disposed of in an orderly fashion when an instance of the `HomeMadeLogging` object is disposed.

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`.

To add your own logic to the Dispose method:

1. Create a method with exactly the following signature:

```
protected virtual void Dispose( bool disposing )
```

2. Include a call to `base.Dispose(disposing)`.

3. Include your own logic.

In the following example, we are customizing the Dispose pattern to expose the `IsDispose` property:

```
[Disposable]
public class MessageFormatter : Formatter
{

  [Child]
  MessageSink sink;

  public bool IsDisposed { get; private set; }

  protected virtual void Dispose( bool disposing )
  {
    base.Dispose( 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.

CAUTION NOTE

The `DisposableAttribute` aspect does not automatically dispose 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`.

7.4. Working With 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` and `AdvisableKeyedCollectionTKey, TItem`.

This topic contains the following sections.

Understanding the need for specific collections

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]
public class Invoice
{
    public Invoice()
    {
        this.Lines = new List<InvoiceLine>();
    }

    [Child]
    public IList<InvoiceLine> Lines { get; private set; }
}

[Aggregatable]
public class 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 runtime, 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 runtime. 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
`AdvisableCollectionT`	`Array`, `ListT`, `CollectionT`, `ObservableCollectionT`
`AdvisableDictionaryTKey, TValue`	`DictionaryTKey, TValue`
`AdvisableKeyedCollectionTKey, TItem`	`KeyedCollectionTKey, TItem`

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 advice 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 `QueryInterfaceT(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 `QueryInterfaceT(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<IAggregatable>( false ) != null )
{
}
```

Controlling the status of collections in the parent-child relationship

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:

```
[Aggregatable]
public class Invoice
{
    public Invoice()
    {
      this.Lines = new AdvisableCollection<InvoiceLine>();
    }

    [Child]
    public IList<InvoiceLine> Lines { get; private set; }

}

[Aggregatable]
public class InvoiceLine
{
    [Parent]
    public Invoice Invoice { get; private set; }
}
```

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.

```
[Aggregatable]
public class Invoice
{
    public Invoice()
    {
      this.Lines = new AdvisableCollection<InvoiceLine>();
    }

  [Child(IsParentSurrogate = false)]
    public IList<InvoiceLine> Lines { get; private set; }

}

[Aggregatable]
public class 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 `AggregatableAttribute` applied on your class to `true`. In this case it's not allowed to override the value in the `[Child]` attributes applied to individual properties.

```
[AggregatableAttribute(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; }

}
```

CHAPTER 8

Undo/Redo

Most business application users are familiar with applications that have the ability to undo and redo changes that they have made. It's not common to see this functionality in custom built applications because it is quite difficult to do. Despite this difficulty, undo/redo is consistently mentioned on the top of users' wish list.

The `RecordableAttribute` aspect makes it much easier to add undo/redo to your application by automatically appending changes done on your object model to a `Recorder` that you can them bind to your user interface. Unlike other approaches to undo/redo, the `RecordableAttribute` aspect only requires minimal changes to your source code.

In this chapter

Section	Description
Making Your Model Recordable [105]	The first step is to make your model classes This section shows how to add the `RecordableAttribute` aspect to the model classes to enable the undo/redo functionality.
Adding Undo/Redo to the User Interface [107]	This section describes how to expose the undo/redo functionality to the application's users.
Customizing Undo/Redo Operation Names [109]	This section shows how to group changes into logical operations and give them a name that is meaningful to the application's users.
Assigning Recorders Manually [114]	This section explains how to customize the assignment of recordable objects to recorders.
Adding Callbacks on Undo and Redo [115]	This section shows how to execute custom logic when undo/redo operations occur in a recordable object.
Understanding the Recordable Aspect [116]	This section describes the concepts and architecture of the `RecordableAttribute` aspect.

8.1. 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 record them into a `Recorder`. The aspect also instruments public methods to group field changes into logical operations.

This topic contains the following sections.

- Making a class recordable using the UI [106]
- Making a class recordable manually [107]
- Working with object graphs and collections [107]

Making a class recordable using the UI

To make a class recordable using the UI:

1. First place the caret on the name of the object that you want to make recordable. The smart tag will appear below the object name. Expand it and select "Add another aspect..."

2. In the Add Another Aspect wizard expand the Model section, select "Enable undo/redo", and click Next.
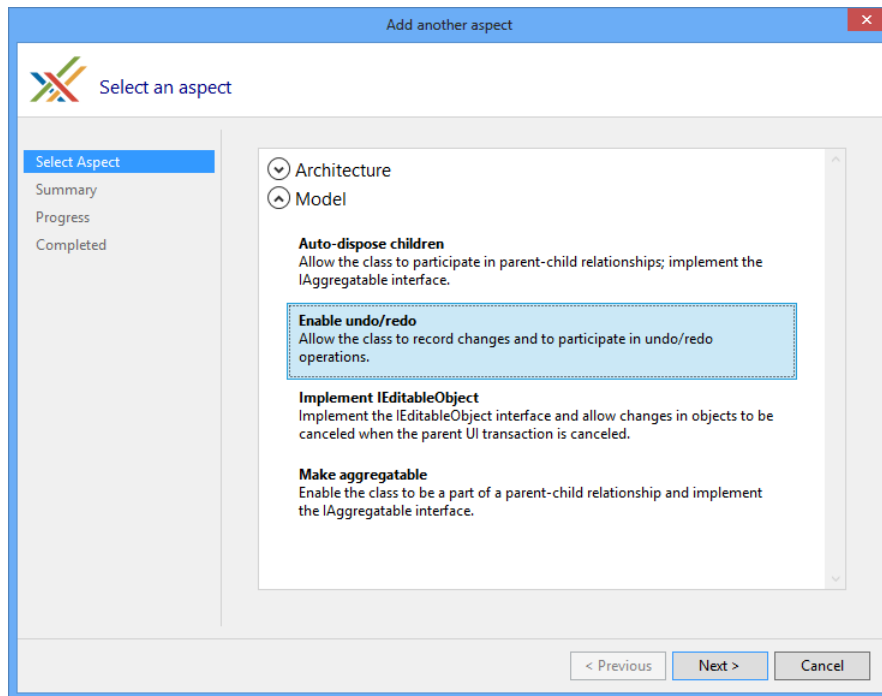
If you have not added a reference to the threading model assembly the Apply Threading Model wizard will download it from Nuget and add the reference.

3. Once the wizard has completed the object will now be flagged as recordable.

```
[NotifyPropertyChanged]
[Recordable]
public class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public int Age { get; set; }
}
```

By adding the `RecordableAttribute` aspect to the `Person` class all of the properties that are primitive types will be recorded when they change.

Making a class recordable manually

To make the class recordable manually:

1. Install the *PostSharp.Patterns.Model* package using NuGet.

2. Add the `RecordableAttribute` to the class.

Working with object graphs and collections

In the example above, the `Invoice` class just had a few fields of a primitive type (`int`, `string`, ...). In real-world application, objects are not isolated entities, but are parts of larger structures named object graphs.

The `RecordableAttribute` aspect needs to understand the parent-child structure of your object graphs. It relies on the `AggregatableAttribute` aspect for this purpose.

Therefore, when adding the `RecordableAttribute` aspect to a class, you need to complete a few more steps:

- Annotate fields with the `ChildAttribute`, `ReferenceAttribute` or `ParentAttribute` custom attribute.

- Replace arrays and collections with instances of the `AdvisableCollectionT` and `AdvisableDictionaryTKey, TValue` classes.

See Parent/Child Relationships [91] for more information.

8.2. 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 forwards and backwards through the operations that they too and have been recorded.

This topic contains the following sections.

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.Model.Controls* 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.Model.Controls"
```

3. To add Undo and Redo buttons to the user interface, include 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"/>
```
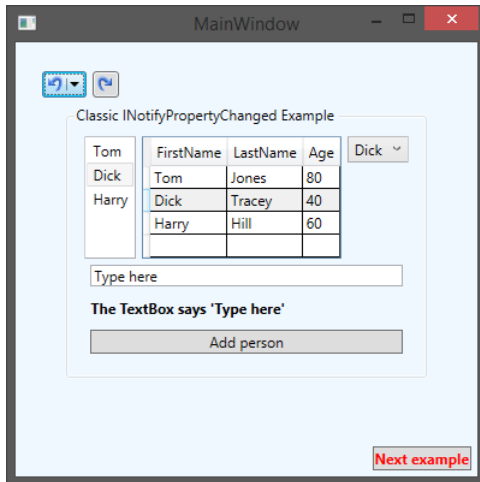

Your users are now able to make a changes in the user interface and Undo and/or Redo those changes at any point that they want.

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.
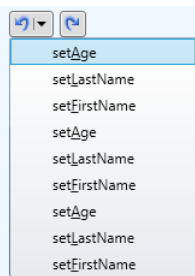
Those actions are listing the changes that were taken when the different `Person` instances were loaded and their properties were set. Most users will only want to see actions that they have manually taken in the screen. As such, you will need to manually interact with the `Recorder` to ensure that the Undo button list is empty when the window opens.

To provide an empty list of recorded actions when the windows is initially opened, open the `ViewModelMain` 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 [109] 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 or WinRT.

Custom controls will typically provide a front-end to the global `Recorder` exposed on the `RecordingServicesDefault-Recorder` property, and we provide a view for the `UndoOperations` and `RedoOperations` collections. Controls typically use the `RecorderUndo` and `RecorderRedo` methods.

8.3. 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.

- Understanding the default operation naming mechanism [109]
- Setting operation names declaratively [110]
- Setting operation names dynamically [111]
- Using the OperationFormatter class [111]

Understanding the default operation naming mechanism

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]
public class Picture
{
    private double x, y;

    public double X
    {
        get { return x; }
        set { x = value; }
    }

    public double Y
    {
```

```
      get { return y; }
      set { y = value; }
   }

   public void Move( double x, double y )
   {
      this.X = x;
      this.Y = y;
   }

}

public static class Program
{
   public static void Main()
   {
      var picture = new Picture();

      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 operation names 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]
public class Picture
{
   private double x, y;

   [RecordingScope("Moving the picture")]
   public void 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 operation names 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. Exclude from the `Recorder` the method which contains the block of code that you wish to encapsulate in a recording.

```
[RecordingScope(RecordingScopeOption.Skip)]
public void Move( double x, double y )
```

> **NOTE**
>
> This step is not required if you are starting the operation from a non-recordable object.

2. Invoke the `OpenScope(String, RecordingScopeOption)` method and wrap the code you want to record in a `using` block.

```
[Recordable]
public class Picture
{
    private double x, y;

    [RecordingScope(RecordingScopeOption.Skip)]
    public void 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;
        }
    }
}
```

> **NOTE**
>
> If you do not to add this custom attribute to the method, 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.

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.

   ```csharp
   public class MyOperationFormatter : OperationFormatter
   {
   }
   ```

2. Create a constructor for the new formatter class.

   ```csharp
   public class MyOperationFormatter : OperationFormatter
   {
       public MyOperationFormatter(OperationFormatter next) : base(next)
       {
       }
   }
   ```

3. Next you need to override the `FormatOperationDescriptor(IOperationDescriptor)` method and write your custom logic for generating a custom operation name.

```csharp
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;
    }
}
```

> **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 `FormatOperation-Descriptor(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.

```csharp
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.

8.4. 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 [114]
- Attaching a recorder manually [115]

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.

   ```
   public class MyProvider : RecorderProvider
   {
   }
   ```

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 it's constructor and pass that to the base constructor.

   ```
   public class MyProvider : RecorderProvider
   {
     public MyProvider(RecorderProvider next) : base (next)
     {
     }
   }
   ```

3. Override the **GetRecorderImpl(Object)** method.

   ```
   public class MyProvider : RecorderProvider
   {
     public MyProvider(RecorderProvider next) : base(next)
     {
     }

     public Recorder GetRecorderImpl(object obj)
     {
       //where you will write code to create a new Recorder instance
       throw new NotImplementedException();
     }
   }
   ```

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);
   ```

RecorderProvider is a chain of responsibility. As such, if a **GetRecorderImpl(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.

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 it's parent the child object's assigned `Recorder` will not be detached. For more information about parent-child relationships, see Parent/Child Relationships [91].

8.5. 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 will show how to integrate the `RecordableAttribute` aspect with a custom implementation of `INotifyPropertyChanged` (the standard `NotifyPropertyChangedAttribute` aspect is already integrated with the `RecordableAttribute` aspect so you don't need to worry).

```
[Recordable]
public class Invoice : INotifyPropertyChanged, IRecordableCallback
```

```
{
  public void OnReplaying(ReplayKind kind, ReplayContext context)
  {
  }

  public void OnReplayed(ReplayKind kind, ReplayContext context)
  {
    OnPropertyChanged("ShippingDate");
  }

  private DateTime _shippingDate;
  public DateTime ShippingDate
  {
    get { return _shippingDate; }
    set
    {
      _shippingDate = value;
       OnPropertyChanged("ShippingDate");
    }
  }

  public event PropertyChangedEventHandler PropertyChanged;

  protected virtual void OnPropertyChanged(string propertyName = null)
  {
      var handler = PropertyChanged;
      if (handler != null) handler(this, new PropertyChangedEventArgs(propertyName));
  }
}
```

For more information, see the reference documentation for the `IRecordableCallback` interface.

8.6. 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

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 `RecorderUndo` 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 this 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 [114] 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(RecordingScope-Option)` 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 [109].

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 similar concept than transactions, but multi-threading 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
`FieldOperationT`	Represents the operation of setting a field to a different value.
`CollectionOperationT`	Represents operations on collections.
`DictionaryOperationTKey, TValue`	Represents operations on dictionaries.
`RecorderOperation`	Represents the operation of attaching or detaching an object to or from a `Recorder`.

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 than `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 non-deterministic; in particular, the aggregation structure is not respected.

It is not allowed, from a callback methods:

- 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.

Undo/Redo

CHAPTER 9
Contracts

This topic contains the following sections.

Contracts

Throwing exceptions upon detecting a bad or unexpected value is good programming practice. However, writing the same checks over and over in different areas of the code base is tedious, error prone, and difficult to maintain.

Consider the following method which checks if a valid string has been passed in:

```
public class CustomerModel
{
public void SetFullName(string firstName, string lastName)
{
    if(firstName == null)
      throw NullReferenceException();

    if(lastName == null)
      throw NullReferenceException();

    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:

```
public class CustomerModel
{
    public void 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 inline with other code.

A contract can also be used in a property as shown in the following example:

```
public class 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:

```
public class CustomerModel
{
    [Required]
    private string mFirstName = "Not filled in yet";

    public void 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:

- from the UI
- manually

This section contains the following subsections.

Adding Contracts from the UI

PostSharp's Visual Studio integration provides a smart tag popup which can be used to select and apply a contract to a parameter, field, or property.
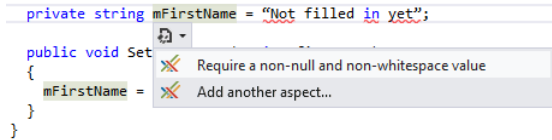
To add contract using the UI:

1. Click on the parameter, field, or property for which the contract is to be applied. While hovering the mouse over this item, a smart tag drop-down will appear:

2. Click on the smart tag drop-down to reveal the contracts available:

```
public class CustomerModel
{
    private string mFirstName = "Not filled in yet";

    public void Set
    {                  ⚔  Require a non-null and non-whitespace value
        mFirstName =  ⚔  Add another aspect...
    }
}
```

3. Select a contract from the list or select **Add another aspect** to display the aspect selection dialog:
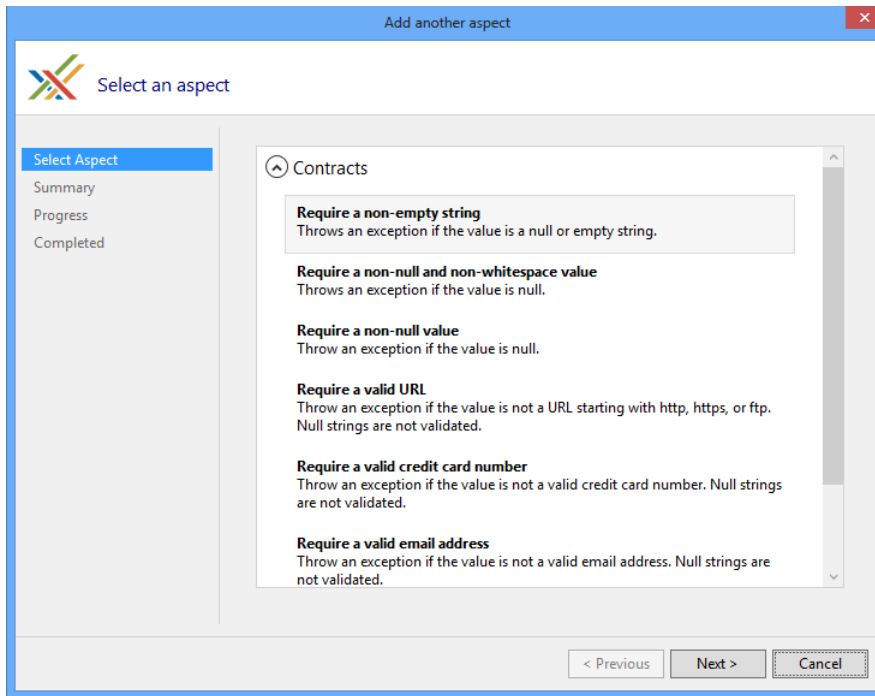
4. Select a contract and click **Next**.

5. Confirm the addition of the contract and click **Next**:
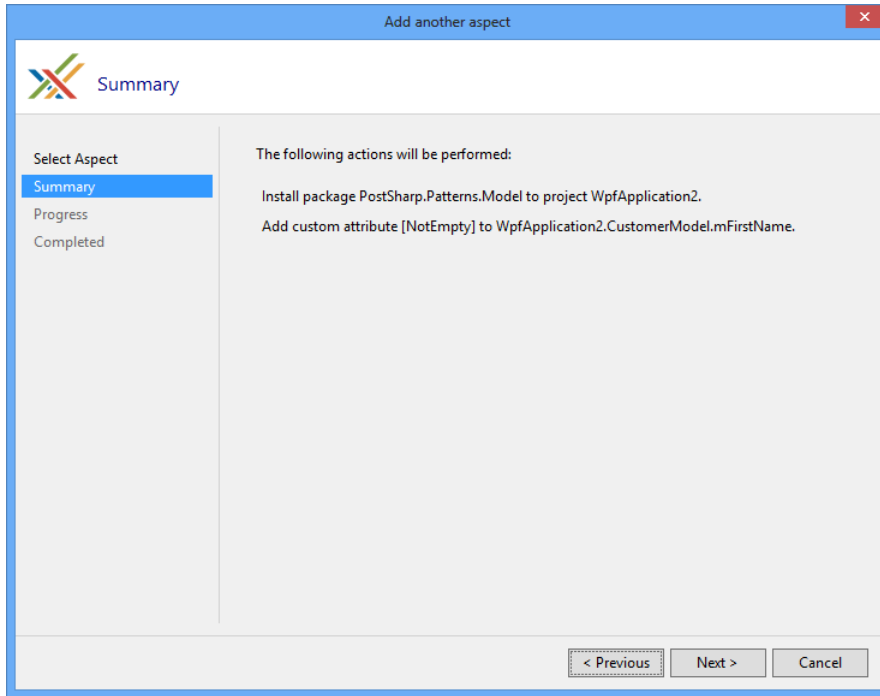
6. Click **Finish** when the dialog indicates that the operation completed:
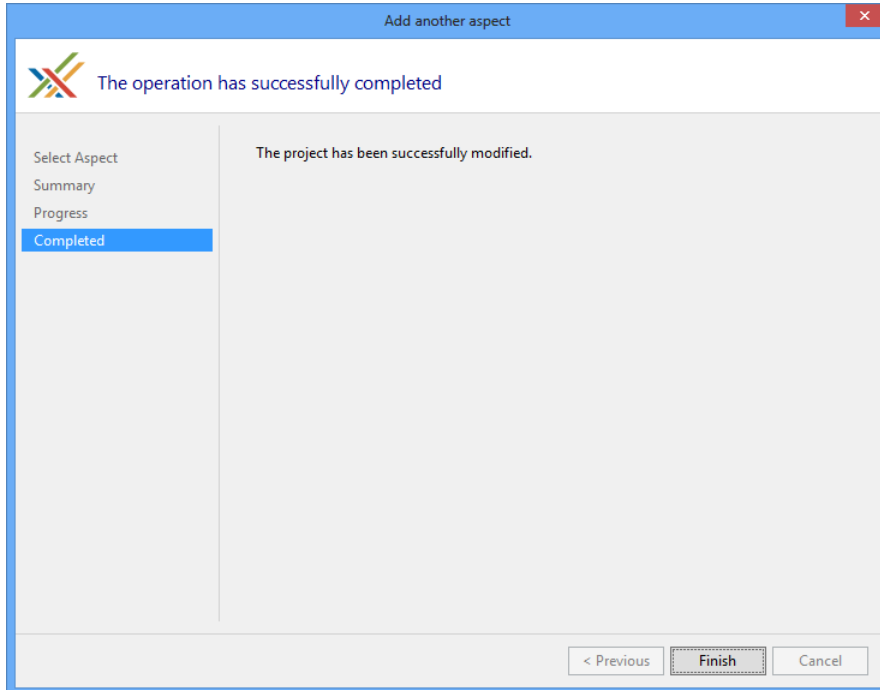
The aspect has now been added in code:

```
public class CustomerModel
{

  [NotEmpty]
  private string mFirstName = "Not filled in yet";

  public void SetFirstName(string firstName)
  {
    mFirstName = firstName;
  }
}
```

Adding Contracts Manually

To add contract manually:

1. Add the assembly: `PostSharp.Patterns.Model` to your project.

2. Add the namespace: `PostSharp.Patterns.Contracts`.

3. Add the attribute before the parameter name for example:

```
public void SetFullName([Required] string firstName, [Required] string lastName)
```

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` and set the `ErrorMessage` property:

```
public class NonZeroAttribute : LocationContractAttribute
{
    public NonZeroAttribute()
      : base()
    {
        ErrorMessage = "The {2} must have a non-zero value";
    }
}
```

> **NOTE**
>
> The `ErrorMessage` property can be set to a formatting string that contains placeholders. See the documentation for the `ErrorMessage` property for more information.

3. Implement the `ILocationValidationAspect` interface in the new contract class which exposes the `Validate-Value(T, String, LocationKind)` 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:

```csharp
public class NonZeroAttribute : LocationContractAttribute, ILocationValidationAspect<int>, ILocationValidationAspect
{
    public NonZeroAttribute()
      : base()
    {
        ErrorMessage = "Expected some type";
    }

    public Exception ValidateValue(int value, string name, LocationKind locationKind)
    {
        if (value == 0)
            return this.CreateArgumentOutOfRangeException(value, name, locationKind);
        else
            return null;
    }

    public Exception ValidateValue(uint value, string name, LocationKind locationKind)
    {
        if (value == 0)
            return this.CreateArgumentOutOfRangeException(value, name, locationKind);
        else
            return null;
    }
};
```

`ValidateValue(T, String, LocationKind)` 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 return an exception if a check fails, or null or if no exception is to be raised.

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:

```csharp
bool Mod([NonZero] int number, [NonZero] int dividend)
{
    return ((number % dividend) == 0);
}

bool Mod([NonZero] uint number, [NonZero] uint dividend)
{
    return ((number % dividend) == 0);
}
```

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:

```csharp
public interface ICustomerModel
{
    void SetFullName([Required] string firstName, [Required] string lastName);
}

public class CustomerModel : ICustomerModel
{
    public void 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 Post-Sharp and PostSharp Model pattern assembly.

Contracts

CHAPTER 10

Logging

The Diagnostics Pattern Library enables you to configure where logging should be performed and to keep your log entries in sync as you add, remove and refactor your codebase. Currently, the library provides a single aspect: `LogAttribute`.

In this chapter

10.1. Walkthrough: Adding Detailed Tracing to a Code Base

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 maybe 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.

PostSharp offers a solution to all of these problems. The logging pattern library allows you to configure where logging should be performed and the pattern library takes over the task of keeping your log entries in sync as you add, remove and refactor your codebase. Let's take a look at how you can add trace logging for the start and completion of method calls.

To add trace logging for the start and completion of method calls:

1. Let's add logging to our `Save` method.

```
public void Save(string firstName, string lastName, string streetAddress, string city)
{
    var customerRepository = new CustomerRepository();
    customerRepository.Save(firstName, lastName, streetAddress, city);
}
```

2. Put the caret on the `Save` method name and expand the Smart Tag. From the list select "Add logging".

3. The first option that you need to select is the Logging Profile. For this example we will take the default provided: it logs the method enters and exits, and includes parameter values and return value. Select the "Default" profile and click **Next**.

4. The next page of the wizard gives you the opportunity to choose the logging back-end that you want to use. For this example select "System.Diagnostics.Trace" and click **Next**.

5. 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**.

6. The progress page shows a progress bar and summary of what actions PostSharp is taking to add the selected logging configuration to your codebase. It's at this point that PostSharp and the logging pattern library will be downloaded from NuGet and added as references to your codebase.

7. Once the download, installation and configuration of PostSharp and the logging pattern library has finished you can close the wizard and look at the changes that were made to your codebase.

8. You'll notice that the code you added the logging to has changed slightly. PostSharp has added a `LogAttribute` attribute to the method. Since we chose the default logging profile, there is no argument to the `LogAttribute` attribute.

```
[Log]
public void Save(string firstName, string lastName, string streetAddress, string city)
{
    var customerRepository = new CustomerRepository();
    customerRepository.Save(firstName, lastName, streetAddress, city);
}
```

> **NOTE**
>
> This example has added a single attribute to one method. If you plan on adding this logging to many different locations in your codebase you will want to read about using the MulticastAttribute: Adding Aspects to Multiple Declarations [147].

9. If you were to run this method the trace logging that you added would output a log message when entering the method and an entry when leaving the method. Note that the parameter values are automatically included in the log message.
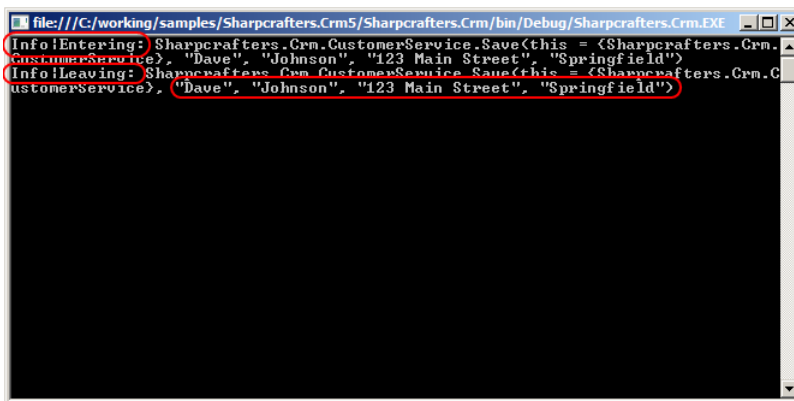
Now that you have logging added to the `Save` method you are able to change the method's name as well as add and remove parameters with the confidence that your log entries will be kept in sync with each of those changes. In combination with attribute multicasting (the article Adding Aspects to Multiple Declarations [147]), adding logging to your codebase and maintaining it becomes a very easy task.

10.2. Walkthrough: Customizing Logging

When adding logging to your codebase, you may need to set up logging options 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*. Logging profiles are stored in the solution-wide configuration file, and each profile specifies what information you want to be included in the log output. You apply a given logging profile by providing its name as an argument for the `LogAttribute` attribute's constructor. Let's take a look at how you can create your own logging profile and use it in your code.

To create a custom logging profile:

1. Let's add customized logging to our Save method.

```
public void Save(string firstName, string lastName, string streetAddress, string city)
{
    var customerRepository = new CustomerRepository();
    customerRepository.Save(firstName, lastName, streetAddress, city);
}
```
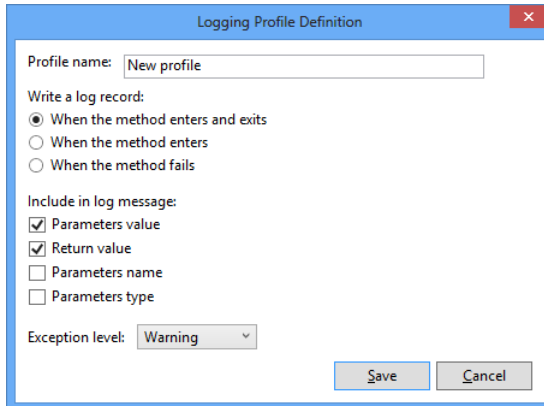
2. Put the caret on the Save method name and expand the Smart Tag. From the list select "Add logging".
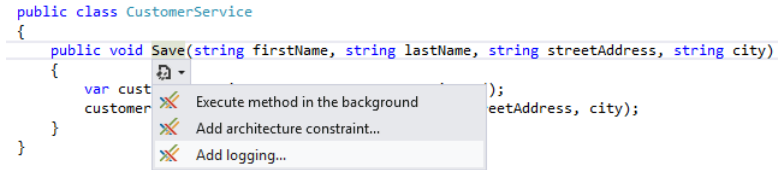
3. The first page of the wizard is the logging profile selection dialogue. In this dialogue you can select one of the predefined profiles, edit an existing profile or create a new one. For this example click on the **"New logging profile..."** link to create a new logging profile.
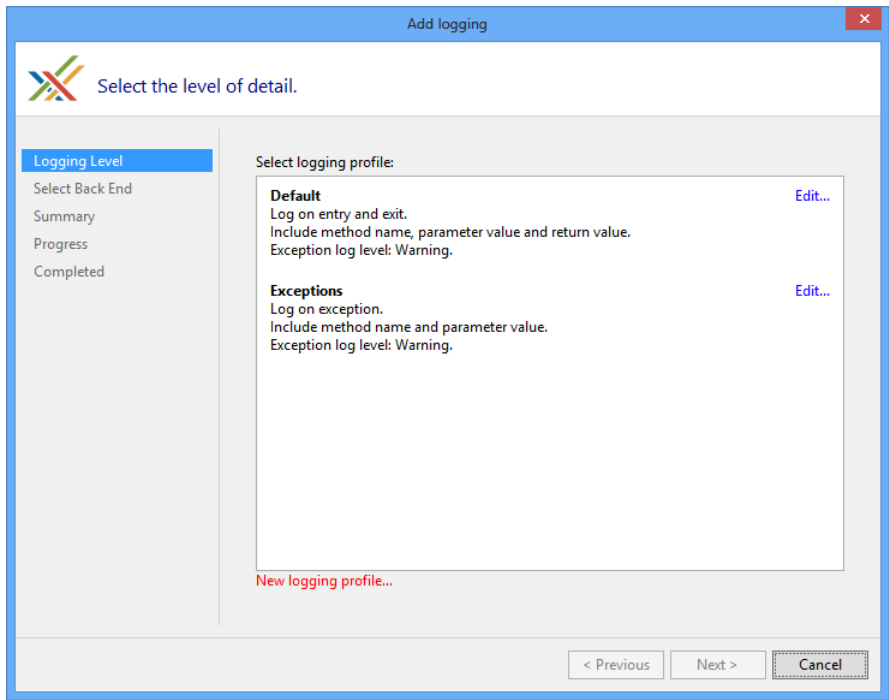
4. Provide a name for your newly created logging profile in the opened profile customization dialogue.
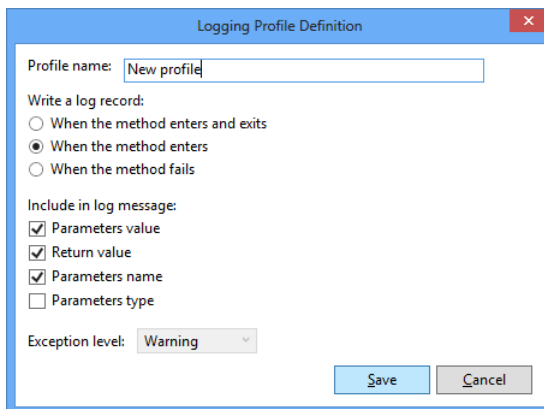
5. Choose when to write a new message to the log. You can select from one of these options: when entering and exiting the method, only when entering the method, only when exception is thrown inside the method.

6. Choose what information to include in each log message. The available options are: parameter values, names, and types, and return value of the method.

7. Finally, select the logging level to use for the exception log messages.

8. When you're done with customizing your new logging profile, click **Save** button.

9. Back in the profile selection dialogue select your profile and click **Next**.
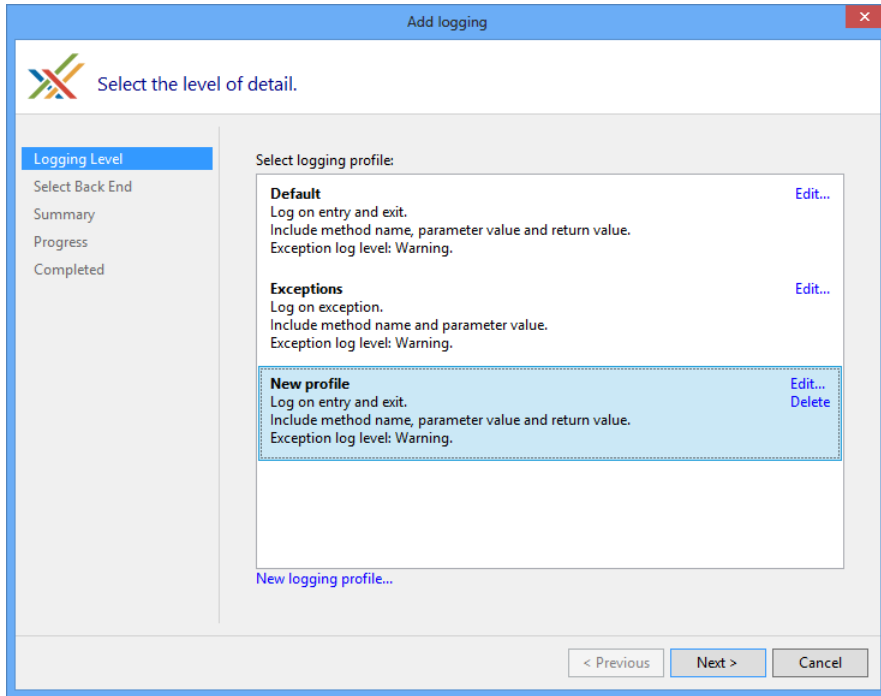
10. The next page of the wizard gives you the opportunity to choose the logging back-end that you want to use. For this example select "System.Diagnostics.Trace" and click **Next**.
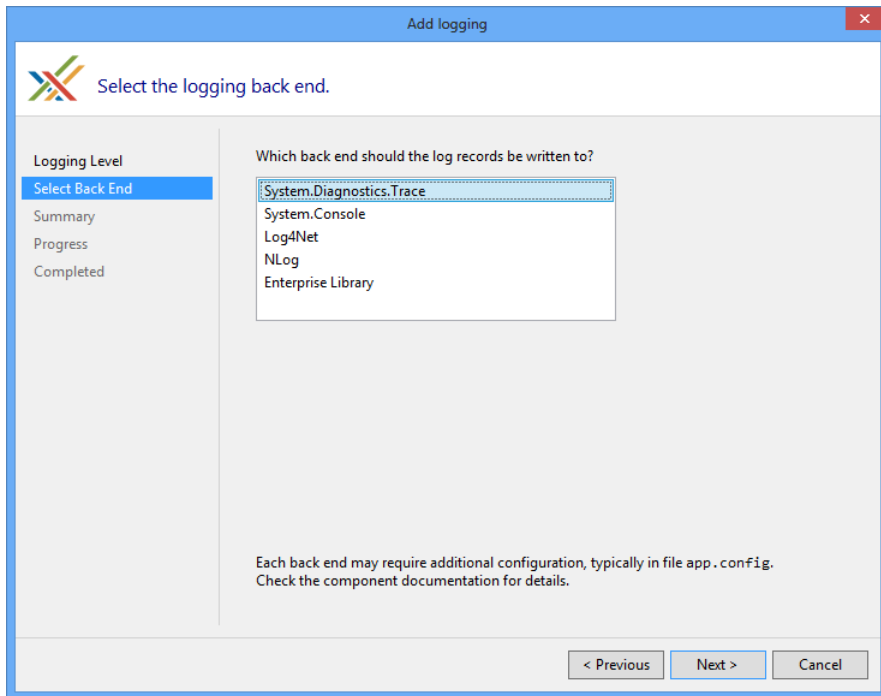
11. The summary page gives you the opportunity to review the selections that you have made. If you notice that the configuration is not what you wanted you can click the **Previous** button and adjust your selections. If the configuration meets your needs click **Next**.
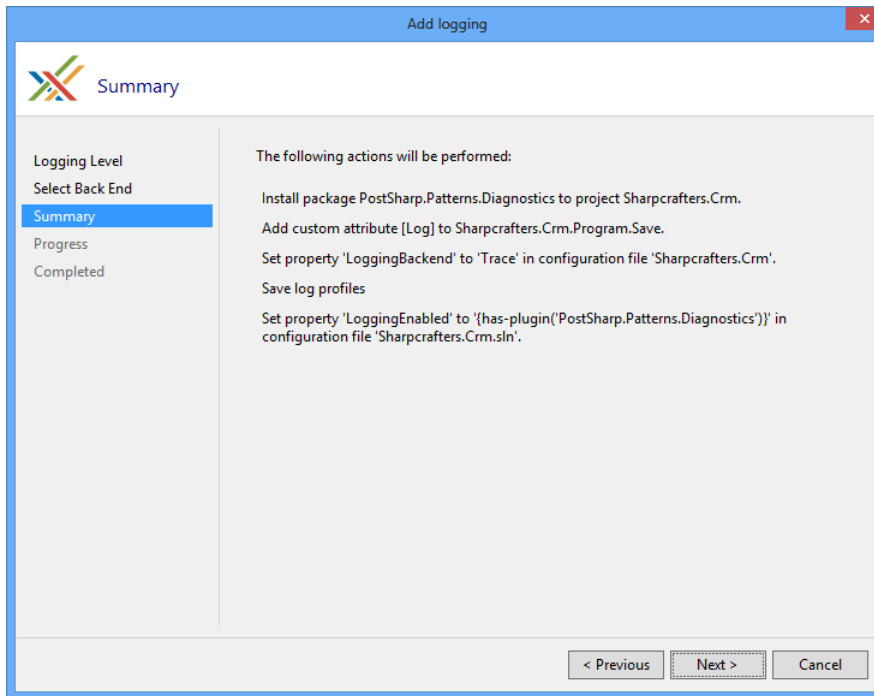
12. The progress page shows a progress bar and summary of what actions PostSharp is taking to add the selected logging configuration to your codebase. It's at this point that PostSharp and the logging pattern library will be downloaded from NuGet and added as references to your codebase.
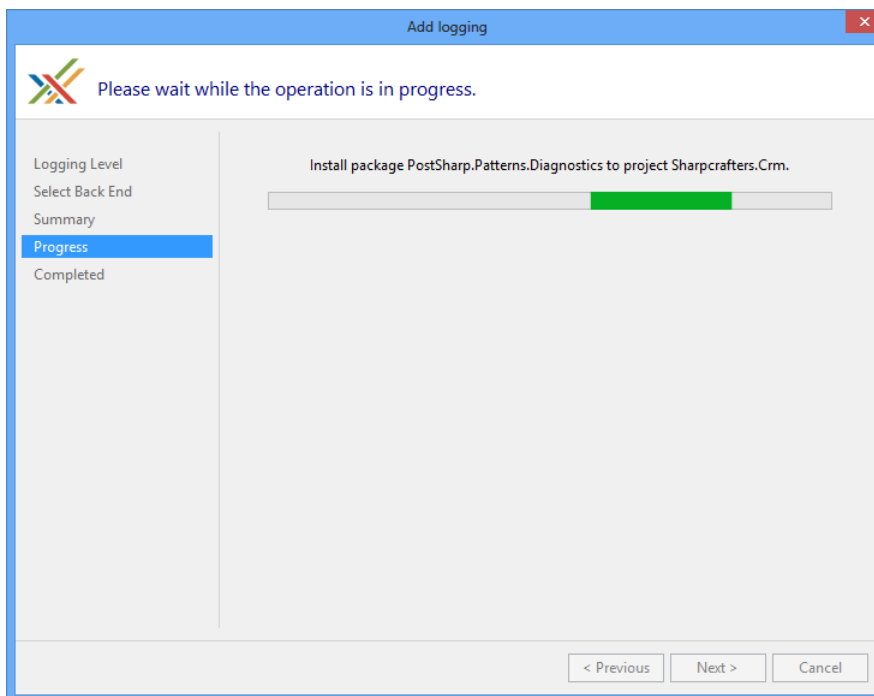
13. Once the download, installation and configuration of PostSharp and the logging pattern library has finished you can close the wizard and look at the changes that were made to your codebase.
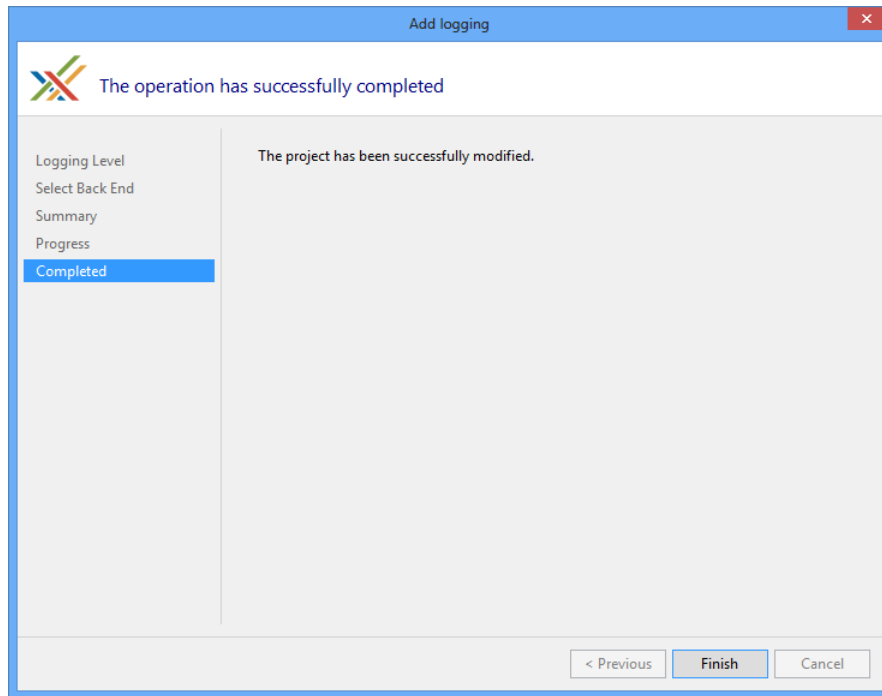
14. First of all, the code window with **SolutionName.pssln** file will be shown. This file stores all your custom logging profiles and other solution-level information.

```xml
<?xml version="1.0" encoding="utf-8"?>
<Project xmlns="http://schemas.postsharp.org/1.0/configuration" xmlns:d="clr-namespace:PostSharp.Patterns.Diagnostics;
  <Property Name="LoggingEnabled" Value="{has-plugin('PostSharp.Patterns.Diagnostics')}" Deferred="true"/>
  <d:LoggingProfiles p:Condition="{$LoggingEnabled}">
    <d:LoggingProfile Name="New profile" OnEntryOptions="IncludeParameterValue | IncludeReturnValue | IncludeThisArgum
  </d:LoggingProfiles>
</Project>
```

15. You'll also notice that the code you added the logging to has changed slightly. PostSharp has added a `Log-Attribute` attribute to the method and the attribute has the name of your logging profile specified as an argument.

```csharp
[Log( "New profile" )]
public void Save(string firstName, string lastName, string streetAddress, string city)
{
    var customerRepository = new CustomerRepository();
    customerRepository.Save(firstName, lastName, streetAddress, city);
}
```

> **NOTE**
>
> This example has added a single attribute to one method. If you plan on adding this logging to many different locations in your codebase you will want to read about using the MulticastAttribute: Adding Aspects to Multiple Declarations [147].

16. If you were to run this method the trace logging that you added would output a log message according to the options you've specified in your logging profile.
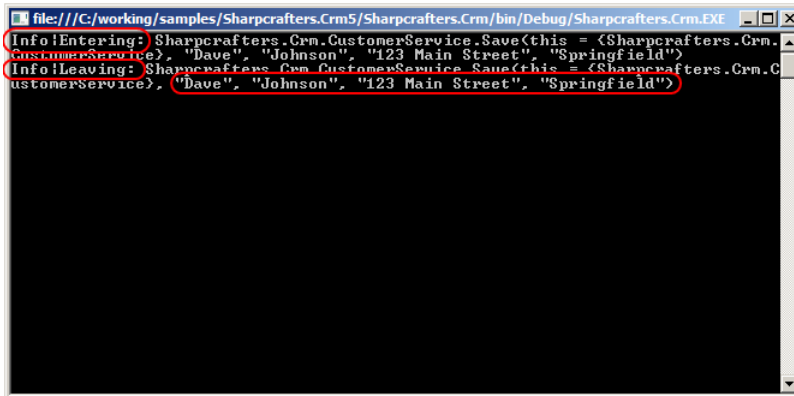
Now that you have logging added to the `Save` method you are able to change the method's name as well as add and remove parameters with the confidence that your log entries will be kept in sync with each of those changes. In combination with attribute multicasting (the article Adding Aspects to Multiple Declarations [147]), adding logging to your codebase and maintaining it becomes a very easy task.

10.3. Walkthrough: Tracing Parameter Values Upon Exception

When you're working with your codebase it's common to need to add logging of exceptions 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 that were passed to the method where the exception is being caught and logged. 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 maybe removed. Along with performing these refactorings you have to remember to update the exception 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.

PostSharp offers a solution to all of these problems. The logging pattern library allows you to configure where logging should be performed and the pattern library takes over the task of keeping your log entries in sync as you add, remove and refactor your codebase. Let's take a look at how you can add trace logging of exceptions that includes the parameter values that were passed to the method that is being logged.

To add trace logging of exceptions that includes the parameter values:
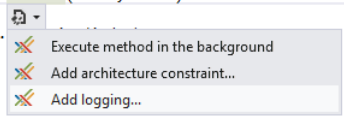
1. Let's add logging to our `DoStuff` method.

```csharp
public void DoStuff(int i, int x)
{
    Console.WriteLine(i/x);
}
```

2. Put the caret on the `DoStuff` method name and expand the Smart Tag. From the list select "Add logging".

3. The first option that you need to select is the Logging Profile. Here we want to choose the "Exceptions" profile and accept its default values. Click **Next**.

4. The next page of the wizard gives you the opportunity to choose the logging back-end that you want to use. For this example select "System.Diagnostics.Trace" and click **Next**.

5. 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**.

6. The progress page shows a progress bar and summary of what actions PostSharp is taking to add the selected logging configuration to your codebase. It's at this point that PostSharp and the logging pattern library will be downloaded from Nuget and added as references to your codebase.

7. Once the download, installation and configuration of PostSharp and the logging pattern library has finished you can close the wizard and look at the changes that were made to your codebase.

8. You'll notice that the code you added the logging to has changed slightly - PostSharp has added a `Log-ExceptionAttribute` attribute to the method.

```
[LogException]
public void DoStuff(int i, int x)
{
    Console.WriteLine(i/x);
}
```

> **NOTE**
>
> This example has added a single attribute to one method. If you plan on adding this logging to many different locations in your codebase you will want to read about using attribute multicasting. See Adding Aspects Declaratively Using Attributes [146].

9. If you were to run this method from a console application and pass in a value of zero for the second parameter it would generate a `DivideByZeroException`. The trace logging that you added would output a log message plus the exception's stack trace to the console. Note that the parameter values are automatically included in the log message.
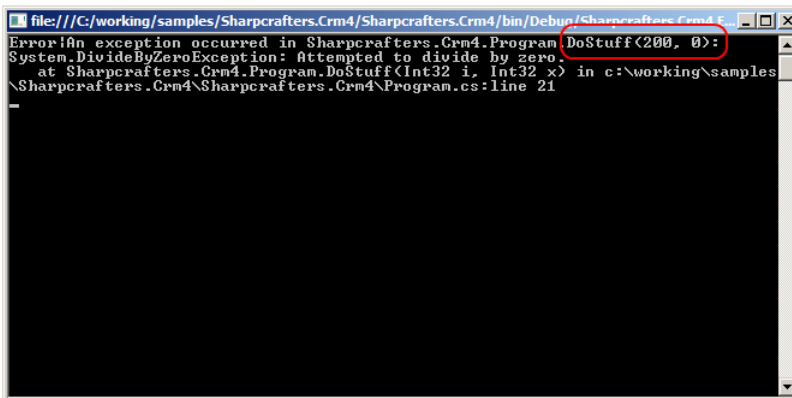
10. For those of you interested in what is happening behind the scenes we can decompile the method and observe what PostSharp has done to our codebase. You'll notice two significant things when you look at the decompiled code. First, PostSharp added in a `try...catch` block that wraps the entirety of the original methods contents. The second thing you'll notice is that the catch block logs the exception and re-throws it. This ensures that your code execution paths will remain unchanged after you've added the logging.

```
public static void DoStuff(int i, int x)
{
    try
    {
        Console.WriteLine(i / x);
    }
    catch (Exception arg)
    {
        <>z__LoggingImplementationDetails.WriteLine("Error|An exception occurred in Sharpcrafters.Crm4.Program.DoStuff({0}, {1}):\n{2}", i, x, arg);
        throw;
    }
}
```

Now that you have logging added to the `DoStuff` method you are able to change the method's name as well as add and remove parameters with the confidence that your log entries will be kept in sync with each of those changes. In combination with the attribute multicasting (See the section Adding Aspects Declaratively Using Attributes [146]), adding logging to your codebase and maintaining it becomes a very easy task.

10.4. Walkthrough: Changing the Logging Back-End

This section describes how to change the logging back-end for a project or solution.

To change the logging back-end:

1. Remove the NuGet package containing the previous back-end implementation, if any.

2. Add the NuGet package containing the new back-end implementation, if any.

   The following table shows the identifier of the back-ends and the package in which they are implemented:

Name	NuGet Package	Description
Console	PostSharp.Patterns.Diagnostics	Logging using `ConsoleWriteLine(String)`
Trace	PostSharp.Patterns.Diagnostics	Logging using `Trace`
Log4Net	PostSharp.Patterns.Diagnostics.Log4Net	
NLog	PostSharp.Patterns.Diagnostics.NLog	
EnterpriseLibrary	PostSharp.Patterns.Diagnostics.EnterpriseLibrary	

3. If the new NuGet package contains several implementations, set the `LoggingBackend` property in the PostSharp project file (*MyProject.psproj*) to the name of the chosen logging back-end. Below is an example of PostSharp project file with the `LoggingBackend` set to "Trace".

   ```xml
   <?xml version="1.0" encoding="utf-8"?>
   <Project xmlns="http://schemas.postsharp.org/1.0/configuration" OverridesDefaultProject="false">
     <Property Name="LoggingBackend" Value="Trace"/>
   </Project>
   ```

CHAPTER 11
Adding Aspects to Code

An aspect has no effect until it is applied to some element of code. PostSharp provides multiple ways to add aspects to your code.

Applying Aspects to Multiple Elements of Code Declaratively

In many situations, you want to apply the same aspect to many elements of code. For instance, you may need to add tracing or performance monitoring to all public methods of a namespace. Since there may be hundreds of affected methods, you don't want to add a custom attribute to all of them.

Thanks to an extension of semantics of custom attributes named *"multicast custom attribute"* (`MulticastAttribute`), it is easy to apply an aspect to multiple elements of code using a single line of code.

For details, see Adding Aspects Declaratively Using Attributes [146] and Understanding Aspect Inheritance [161].

Applying Aspects to Multiple Elements of Code Imperatively

If declarative features of `MulticastAttribute` are not sufficient for your case, you can select elements of code imperatively. For instance, you can develop complex filters based on `System.Reflection` or read information from an XML file.

There are two ways you can implement imperative selection aspect targets:

Filtering Out Using CompileTimeValidate

To filter out elements of codes that have been selected by `MulticastAttribute`, you can implement the method `Compile-TimeValidate(Object)` of your aspect and silently return `false` if the candidate target is not appropriate. .

For instance, the following aspect will apply only on security-critical methods.

```csharp
using System;
using System.Reflection;
using PostSharp.Aspects;

namespace Samples2
{
    [Serializable]
    public sealed class TraceSecurityCriticalAttribute : OnMethodBoundaryAspect
    {
        // Select only security-critical methods.
        public override bool CompileTimeValidate(MethodBase method)
        {
            return method.IsSecurityCritical;
        }

        public override void OnEntry(MethodExecutionArgs args)
        {
            Console.WriteLine("On Entry");
        }
    }
}
```

See Validating Aspect Usage [262] for details.

Adding Aspect Instances Using IAspectProvider

If you have to implement more complex rules to select the target of aspects, you can create another aspect that will do nothing else than adding aspect instances to your code. This aspect must implement the interface `IAspectProvider` and will typically derive from `AssemblyLevelAspect` or `TypeLevelAspect`.

> **TIP**
>
> Use `ReflectionSearch` to perform complex queries over `System.Reflection`.

11.1. 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 of 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 it 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 needs 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 [147]
- Adding Aspects to Multiple Declarations [147]
- Adding Aspects to Derived Classes and Methods [149]
- Overriding and Removing Aspect Instances [154]
- Reflecting Aspect Instances at Runtime [157]

For a conceptual overview of this feature, see:.

- Understanding Attribute Multicasting [158]

-

11.1.1. Adding Aspects to a Single Declaration

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

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

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

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

11.1.2. Adding Aspects to Multiple Declarations

Once have 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 [162]) and dynamic aspect providers (see more in the section Adding Aspects Programmatically using IAspectProvider [163]).

This topic contains the following sections.

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.

```
[OurLoggingAspect]
public class CustomerServices
```

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's an unsustainable proposition. Thankfully, there is a way to make this easier. Instead of applying your aspect on 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.

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 Post-Sharp'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 applied.

3. Add the `AttributeTargetTypes` property to the aspects's 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: OurLoggingAsepct(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 [154] 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`'s 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 a 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 `AttributeTargetTypeAttributes` property to the `MulticastAttribute`'s constructor.

2. Set the `AttributeTargetTypeAttributes` value to `Virtual`Virtual.

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

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 emitting any error, and the candidate target will be ignored. See the section Validating Aspect Usage [262] for details.

The second approach is opt-in. See the section Adding Aspects Programmatically using IAspectProvider [163] for details.

11.1.3. 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.

NOTE

PostSharp Professional or higher edition is required for aspect inheritance.

This topic contains the following sections.

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 `IAspectProvider`. In this example we start with the `AutoDataContractAttribute` class which was introduced in the section Example: Automatically Adding DataContract and DataMember Attributes [294].

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]
   public sealed class AutoDataContractAttribute : TypeLevelAspect, IAspectProvider
   {
       // Details skipped.
   }
   ```

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

   ```
   [AutoDataContractAttribute]
   class Document
   {
       public string Title { get; set; }
       public string 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 that hard-coding it into the custom attribute. In the following snippet, we've removed the `MulticastAttributeUsageAttribute` attribute from `Auto-DataContractAttribute`:

```
// MulticastAttributeUsage(Inheritance = MulticastInheritance.Strict)]
[Serializable]
public sealed class 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:

```
[TraceMethodAttribute(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:

```
[Serializable]
public sealed class TraceMethodAttribute : OnMethodBoundaryAspect
{
    public override void OnEntry(MethodExecutionArgs args)
    {
        Console.WriteLine(string.Format("Entering {0}.{1}.", args.Method.DeclaringType.Name, args.Method.Name));
    }

    public override void 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.
    [TraceMethodAttribute(AttributeInheritance = MulticastInheritance.Strict)]
    public virtual void RenderHtml(StringBuilder html)
    {
        html.AppendLine( this.Title );
      html.AppendLine( this.Author );
    }
}

class MultiPageArticle: Document
{

  // This method will be traced.
```

```csharp
    public override void RenderHtml(StringBuilder html)
    {
      base.RenderHtml(html);
      foreach ( ArticlePage page in this.Pages )
      {
        page.RenderHtml( html );
      }
    }

  // This method will NOT be traced.
      public void 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 `Multi-PageArcticle.RenderHtml` method as shown here:

```
Entering MultiPageArcticle.RenderHtml
Entering Document.RenderHtml
Leaving Document.RenderHtml
Leaving MultiPageArcticle.RenderHtml
```

NOTE

Aspect inheritance works with virtual, abstract and interface methods and their parameters.

We would get the similar result by adding the `TraceMethod` attribute to the `Document` class. Indeed, by virtue of attribute multicasting (see section Adding Aspects to Multiple Declarations [147] for more details), adding a method-level attribute to a class implicitly adds it to all method of this class.

```csharp
[TraceMethodAttribute(AttributeInheritance = MulticastInheritance.Strict)]
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 NOT 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 NOT be traced.
        public void 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 MultiPageArcticle.RenderHtml
Entering Document.RenderHtml
Entering Document.get_Title
Leaving Document.get_Title
Entering Document.get_Author
Leaving Document.get_Author
Leaving Document.RenderHtml
Leaving MultiPageArcticle.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:

```
[TraceMethodAttribute(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.

11.1.4. Overriding and Removing Aspect Instances

Having multiple instances of the same aspect on the same element of code is sometimes a 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 [277] 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 of 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 `MulticastAttributeUsageAttributeAllowMultiple` 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).

```csharp
using System;
using System.Diagnostics;
using PostSharp.Aspects;
using PostSharp.Extensibility;
using Samples3;

[assembly: Trace(AttributeTargetTypes = "Samples3.My*", Category = "A")]
[assembly: Trace(AttributeTargetTypes = "Samples3.My*",
    AttributeTargetMemberAttributes = MulticastAttributes.Public, Category = "B")]

namespace Samples3
{
    [Serializable]
    public sealed class TraceAttribute : OnMethodBoundaryAspect
    {
        public string Category { get; set; }

        public override void OnEntry(MethodExecutionArgs args)
        {
            Trace.WriteLine("Entering " +
                        args.Method.DeclaringType.FullName + "." + args.Method.Name, this.Category);
        }
    }


    public class MyClass
    {
        // This method will have 1 Trace aspect with Category set to A.
        private void Method1()
        {
        }

        // This method will have 2 Trace aspects with Category set to A, B
        public void Method2()
        {
        }

        // This method will have 3 Trace aspects with Category set to A, B, C.
        [Trace(Category = "C")]
        public void 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 on assembly level, but these aspects are replaced by a different one on `Method3`.

```csharp
using System;
using System.Diagnostics;
using PostSharp.Aspects;
using PostSharp.Extensibility;
using Samples5;
```

```
[assembly: Trace(AttributeTargetTypes = "Samples5.My*", Category = "A")]
[assembly: Trace(AttributeTargetTypes = "Samples5.My*",
    AttributeTargetMemberAttributes = MulticastAttributes.Public, Category = "B")]

namespace Samples5
{
    [Serializable]
    public sealed class TraceAttribute : OnMethodBoundaryAspect
    {
        public string Category { get; set; }

        public override void OnEntry(MethodExecutionArgs args)
        {
            Trace.WriteLine("Entering " +
                        args.Method.DeclaringType.FullName + "." + args.Method.Name, this.Category);
        }
    }


    public class MyClass
    {
        // This method will have 1 Trace aspect with Category set to A.
        private void Method1()
        {
        }

        // This method will have 2 Trace aspect with Category set to A, B.
        public void Method2()
        {
        }

        // This method will have 1 Trace aspects with Category set to C.
        [Trace(Category = "C", AttributeReplace = true)]
        public void 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 Samples4;

[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]
    public sealed class TraceAttribute : OnMethodBoundaryAspect
    {
        public string Category { get; set; }

        public override void OnEntry(MethodExecutionArgs args)
        {
            Trace.WriteLine("Entering " +
                        args.Method.DeclaringType.FullName + "." + args.Method.Name, this.Category);
```

```
        }
    }

    public class MyClass
    {
        // This method will have 1 Trace aspect with Category set to A.
        private void Method1()
        {
        }

        // This method will have 1 Trace aspects with Category set to B.
        public void Method2()
        {
        }

        // This method will have 1 Trace aspects with Category set to C.
        [Trace(Category = "C")]
        public void 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 )]
          public static class Helpers
          {

          }
        }
```

11.1.5. 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 )]
  public class TagAttribute : MulticastAttribute
  {
    public string Tag;
  }
```

157

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.

11.1.6. 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, or/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 on an assembly, it is implicitly applied to all fields of that assembly.

The general rule is: when a multicast attribute is applied on 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 maximalistic: 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 on 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]
public class GuiThreadAttribute : MethodInterceptionAspect
{
// Details skipped.
}
```

Note the presence of the `AttributeUsageAttribute` attribute in the sample above. It tells the C# or Visual Basic 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 on a type containing an abstract method, the aspect will not be multicast to this method, neither to its constructors.

TIP

Additionally 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.

As an aspect user, it is important to understand that you can only apply aspects to elements of codes 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 = MulticastAttributes
```

Filtering Properties

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

`MulticastAttribute` Property	`MulticastAttributeUsage-Attribute` Property	Description
`AttributeTargetElements`	`ValidOn`	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.

MulticastAttribute Property	MulticastAttributeUsage-Attribute Property	Description
AttributeTargetAssemblies		Wildcard expression or regular expression specifying to which assemblies the attribute is multicast.
	AllowExternalAssemblies	Determines whether the aspect can be applied to elements defined in a different assembly than the current one.
AttributeTargetTypes		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.
AttributeTargetTypeAttributes	TargetTypeAttributes	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.
AttributeTargetMembers		Wildcard expression or regular expression filtering by name the member to which the attribute is applied.
AttributeTargetMemberAttributes	TargetMemberAttributes	Restricts the attributes (visibility, virtuality, abstraction, literality, …) of the member to which the aspect is applied.
AttributeTargetParameters		Wildcard expression or regular expression specifying to which parameter the attribute is multicast.
AttributeTargetParameterAttributes	TargetParameterAttributes	Restricts the attributes (in/out/ref) of the parameter to which the aspect is applied.
AttributeInheritance	Inheritance	Specifies whether the aspect is propagated along the lines of inheritance of the target interface, class, method, or parameter (see Understanding Aspect Inheritance [161]).

CAUTION NOTE

Whenever possible, do not rely on naming conventions to apply aspects (properties `AttributeTargetTypes`, `Attribute-TargetMembers` 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 `MulticastAttribute-UsageAttribute`. 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: she can restrict the set of possible targets supported by the aspect, but cannot enlarge it.

11.1.7. 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 of 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: these 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 [146] for details).

The difference between strict and multicasting 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.
```

Adding Aspects to Code

```
  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.Method`.

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 [154] for details.

11.2. 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

This example is based on the `AutoDataContractAttribute` explained in the section Example: Automatically Adding Data-Contract and DataMember Attributes [294].

```
namespace MyCustomAttributes
{
    // We set up multicast inheritance so  the aspect is automatically added to children types.
    [MulticastAttributeUsage(MulticastTargets.Class, Inheritance = MulticastInheritance.Strict)]
    [Serializable]
    public sealed class AutoDataContractAttribute : TypeLevelAspect, IAspectProvider
    {
        // Details skipped.
    }
}
```

Normally `AutoDataContractAttribute` would be applied to `Customer` in code as follows:

```
namespace MyNamespace
{
    [AutoDataContractAttribute]
    class Customer
    {
        public string FirstName {get; set;}
        public string 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-namespace:MyCustomAttributes;assembly:MyAssembly">
        <my:AutoDataContractAttributeAttributeTargetTypes=" MyNamespace.Customer"/>
    </Multicast>

</Project>
```

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 [147] for details about these properties.

11.3. 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. In 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 => return new AspectInstance(targetElement, 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 Customizing the NotifyPropertyChanged Aspect [82].

NOTE

PostSharp does not automatically initialize the aspects provided by `IAspectProvider`, even if the method `CompileTime-Initialize` 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 runtime just like normal aspects using the `RunTimeInitialize` method.

Creating Graphs of Aspects

It is common that aspects provided by `IAspectProvider` (children aspects) form an 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 runtime. In other words, if you store in a child aspect a reference to another aspect, you will be able to use this reference at runtime.

Threading Patterns

CHAPTER 12

Writing Thread-Safe Code with Threading Models

A threading model is a design pattern that gives guarantees that your code executes safely on a multi-threaded 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 to assign 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 multi-threaded 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 [192]	`ThreadUnsafeAttribute`	These objects may never be accessed concurrently by several threads.

Threading Model	Aspect Type	Description
Thread Affine Threading Model [195]	`ThreadAffineAttribute`	These objects must be accessed from a the thread that instantiated them.
Synchronized Threading Model [188]	`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 [183]	`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 [178]	`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 [171]	`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 [174]	`ImmutableAttribute`	These objects cannot have their state changed after their constructor has finished executing.

Other topics

Article	Description
Opting In and Out From Thread Safety [198]	This article shows how to disable the enforcement of the threading model for specific fields or methods.
Compatibility of Threading Models [200]	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 [201]	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[15] for details about the concepts and architecture of PostSharp Threading Models.

---

15. http://www.postsharp.net/links/threading-model-white-paper

12.1. 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.

This topic contains the following sections.

Making an object freezable using the UI

To make an object freezable all you need to do is add the `FreezableAttribute` attribute to the class in question.

NOTE

This example shows freezing a class that only has primitive data types. Information on working with complex objects can be found the Freezable Object Trees section later in this article.

---

1. First place the caret on the name of the object that you want to make freezable. The smart tag will appear below the object name. Expand it and select "Apply Threading Model...".

2. In the Apply Threading Model wizard select "Apply freezable threading model" and click Next.
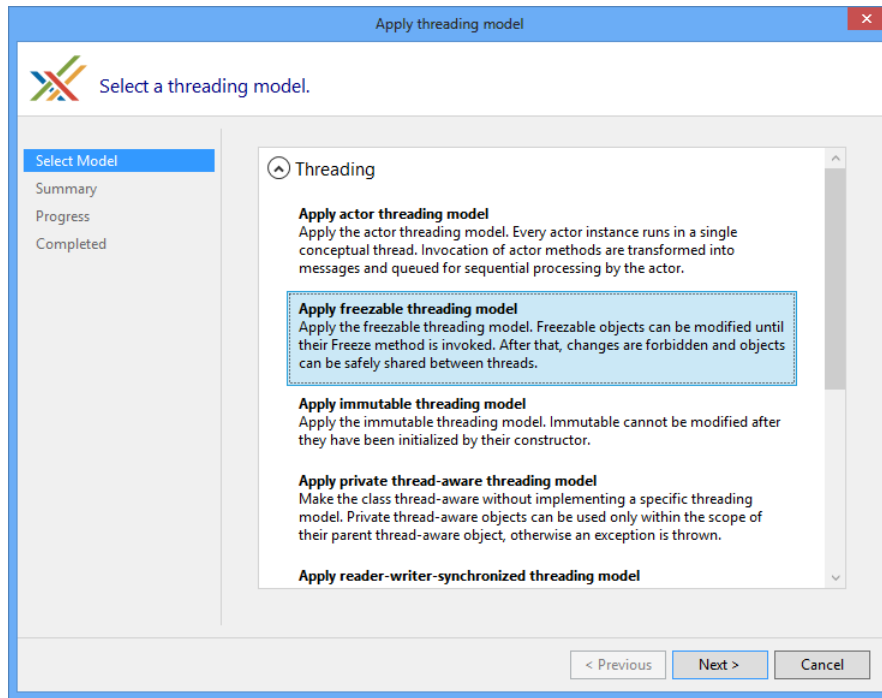
If you have not added a reference to the threading model assembly the Apply Threading Model wizard will download it from NuGet and add the reference.

3. Once the wizard has completed the object will now be flagged as freezable and the Customer and Item properties have had Parent-Child relationships established on them.

```csharp
using PostSharp.Patterns.Threading;

[Freezable]
public class Invoice
{
    public long Id { get; set; }
}
```

Making an object freezable manually

NOTE

This example shows freezing a class that only has primitive data types. To see how to work with complex objects see the **[]** section later in this article.

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]
public class Invoice
{
    public long Id { get; set; }
}
```

Freezing an object

To freeze an object, use will first have to case the object to the `IFreezable` interface. After that you are 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 `CastSourceType, TargetType(SourceType)` 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 `CastSourceType, TargetType(SourceType)` method. Instead, you will have to use the `QueryInterfaceT(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;
```

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 Relationships [91] for more information on how to establish object trees.

IMPORTANT NOTE

Children of freezable objects must be either freezable either immutable. Therefore, children classes must be annotated with the `FreezableAttribute` or `ImmutableAttribute` custom attribute. Collection types must be derived from `AdvisableCollectionT` or `AdvisableDictionaryTKey, TValue`.

12.2. Immutable Threading Model

There are times when you want certain objects in your codebase to retain their post creation state without possibility of it ever changing. These objects are said to be immutable. Immutable objects are useful in multi-threaded applications because they can be safely accessed by several threads concurrently, without 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 [171].

This topic contains the following sections.

Making a class immutable

To make an object immutable all you need to do is add the `ImmutableAttribute` attribute to the class in question.

NOTE

This example shows freezing a class that only has primitive data types. Information on working with complex objects can be found the Working With Immutable Object Trees section later in this article.

To add the Immutable pattern using the UI:

1. First place the caret on the name of the class that you want to make immutable. The smart tag will appear below the object name. Expand it and select **Apply Threading Model**.
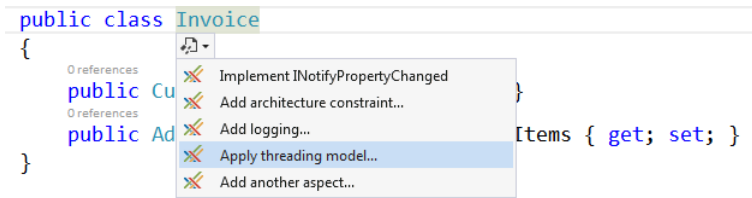
2. In the **Apply Threading Model** wizard select **Apply immutable threading model** and click **Next**.
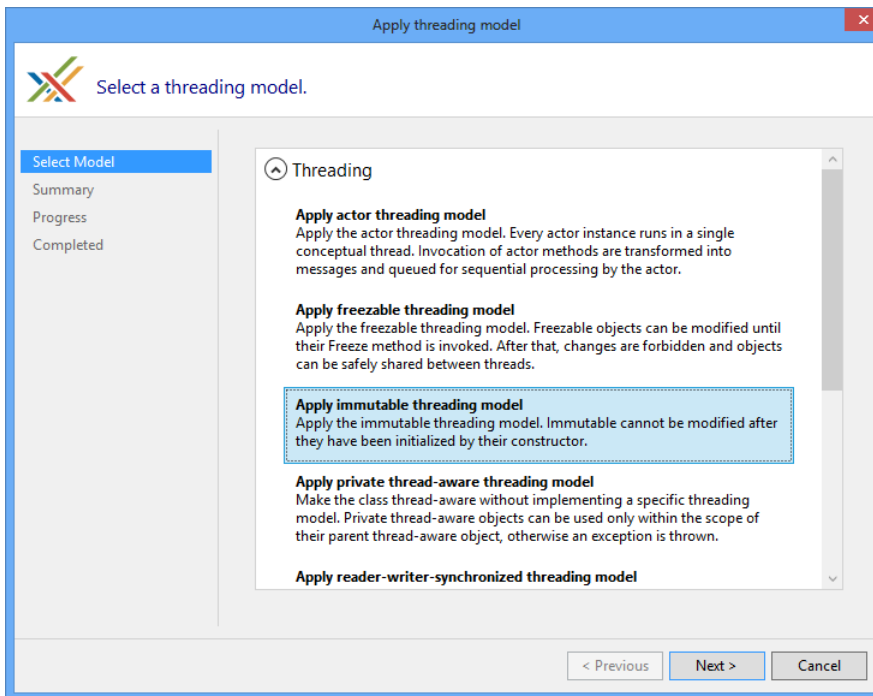
If you have not yet added the *PostSharp.Patterns.Threading* package to your project, the **Apply Threading Model** wizard will download it from NuGet and install it into your project.

3. Once the wizard has completed the class will now be flagged as immutable.

```csharp
using PostSharp.Patterns.Threading;

[Immutable]
public class Invoice
{
  public Invoice(long id)
  {
    Id = id;
  }
  public long Id { get; set; }
}
```

Now, once you've created an instance of the immutable object your 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 `ObjectReadOnly-Exception`.

```csharp
var invoice = new Invoice(12345);

// This will throw an exception.
invoice.Id = 345678;
```

To add the Immutable pattern manually:

1. Add the *PostSharp.Patterns.Threading* package to your project using NuGet.

2. Add the `ImmutableAttribute` custom attribute to your class.

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 Relationships [91].

NOTE

Children of immutable objects must be marked as immutable themselves. Adding the `ImmutableAttribute` to the child classes will accomplish this. Additionally, collection types must be derived from `AdvisableCollectionT` or `Advisable-DictionaryTKey, TValue`.

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.

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's value.

```csharp
public class Invoice
{
  public readonly long _id;
  public Invoice(long id)
  {
    SetIdentifier(id);
  }
```

```
  public void SetIdentifier(long id)
  {
    _id = id
  }
}
```

If you were to mark a complex object as `readonly` the same rule holds true as it does for primitive types. You are not able to change the complex object instance outside of the constructor. In the following example, initializing the `Customer` in the constructor is valid, but reinitializing it in the `Refresh` method will cause a compilation error.

```
public class Invoice
{
  public readonly Customer _customer;
  public Invoice()
  {
    _customer = new Customer();
  }

  public void Refresh()
  {
    //will cause a compilation error
    _customer = new Customer();
  }
}
```

What you can do when a complex object is marked as `readonly` is alter the values within that complex object. In the case of the `Customer` object you cannot reinitialize the instance but you can change the properties on the already created instance. What you see in the following `Refresh` methd is perfectly valid.

```
public class Invoice
{
  public readonly Customer _customer;
  public Invoice()
  {
    _customer = new Customer();
  }

  public void Refresh()
  {
    //valid but not immutable
    _customer.Name = "Jim";
    _customer.Phone = "555-123-9876";
  }
}
```

The same type of change to object state can happen with collections. You can not reinitializee 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.

```
public class Invoice
{
  public readonly 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 = 3.50;
    _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 Inovice(long id) : base(id)
  {
    Id = 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.

12.3. 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, multi-threading 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 provide 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.

A single-threaded example

Consider the following example of an `AverageCalculator` class. The code is not thread because incrementing the `count` has four operations (read and write) that must all be performed atomically.

```csharp
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.

Applying the Actor model using the UI

To apply the Actor threading model to your class with the UI:

1. Place the mouse cursor over your class name and select "Apply threading model..." from the drop-down.
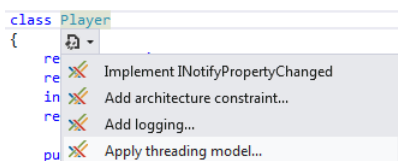
2. Select "Apply actor threading model" and click **Next**.
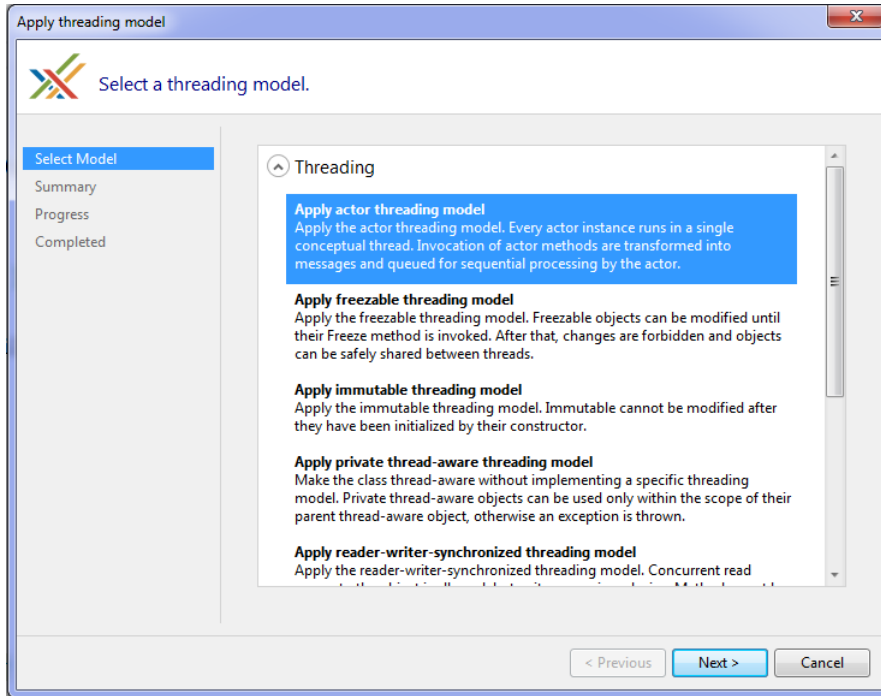
3. Verify the actions on the Summary screen and click **Next**.
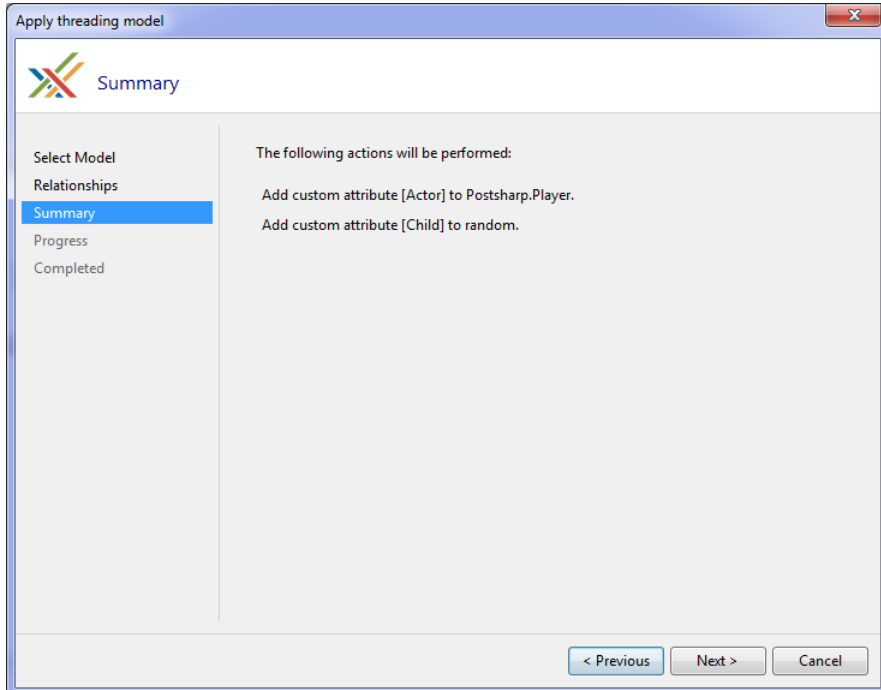
4. Click **Finish** after the installation completes:
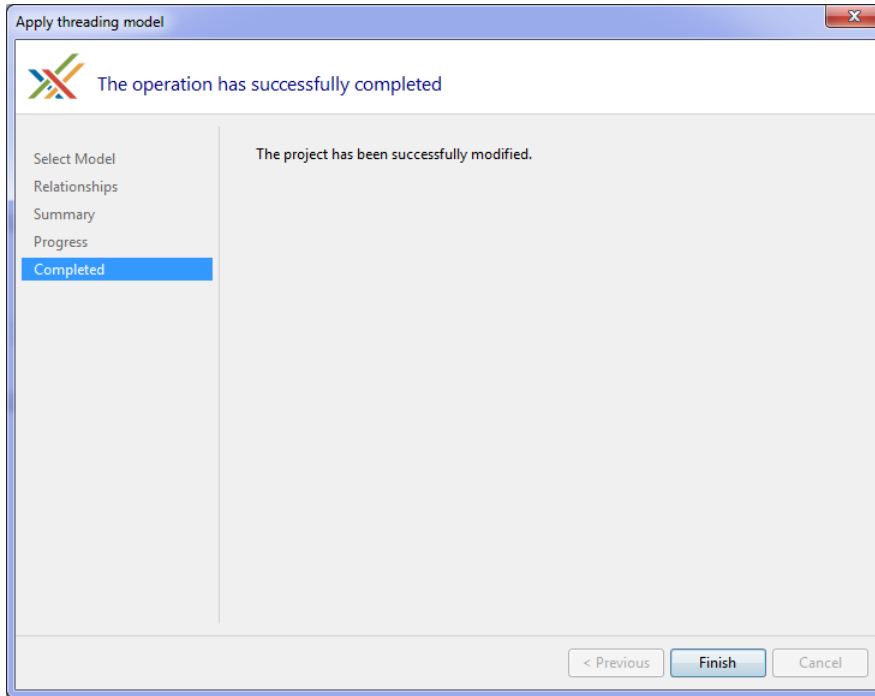
Your class will now have the `ActorAttribute` and all dependencies will have been added to the project.

5. Finally, you need to change all methods that have a return value to asynchronous methods, and modify the code that calls them.

Applying the Actor manually

To apply the Actor threading model manually:

1. Add the PostSharp.Patterns.Threading NuGet package to your project.

2. Add the `ActorAttribute` to the class.

3. Finally, you need to change all methods that have a return value to asynchronous methods, and modify the code that calls them.

In the reworked example below, the `AverageCalculator` class has had the `ActorAttribute` added and the `GetAverage` methods has been changed into asynchronous an methods. The `AddSample` method was not changed because it does not have a return value.

```csharp
[Actor]
class AverageCalculator
{
    float sum;
    int count;

    public void AddSample(float n)
    {
        this.count++;
        this.sum += n;
    }

    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 : IObserver<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 our calculator will enqueue method calls.
      this.calculator.AddSample( value );
    }

    // Details skipped.
}
```

Behind the scenes, each invocation of `AverageCalculator.AddSample` is added to the message queue by the `Actor-Attribute`, 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 multi-threading into account.

Working with 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, state is more complex than fields of simple type like **[int]** or **[string]**. 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 runtime 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:

```
[Actor]
class AverageCalculator
{
    float sum;
    int count;

    [Child]
    private CounterInfo counterInfo;

    // Other details skipped for brevity
}
```

2. Add the `PrivateThreadAwareAttribute` attribute to the child class.

```
[PrivateThreadAware]
public class CounterInfo
{
    public string Name { get; set; }
}
```

For more information regarding parent-child relationships in threading models, see also Parent/Child Relationships [91].

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 [198].

12.4. 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.

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 multi-threaded program, the `AmountAfterDiscount` property could be evaluated in the middle of the `Set` operation, and return an inconsistent result.

This topic contains the following sections.

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.

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:

```csharp
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 non-deterministic data structure corruptions.

Making a class reader-writer synchronized

PostSharp Threading Pattern Library has been designed to eliminate non-deterministic 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:
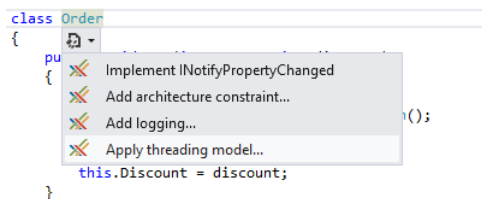
- At any time, the object can be open for reading or closed for reading.

- 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.

There are two ways to add the reader-writer-synchronized pattern to your class:

1. using the user interface
2. manually

To apply the Reader-Writer Synchronized pattern using the user interface:

1. Hover the mouse over the class name. This displays the smart tag dropdown.

2. Click the smart tag dropdown and click **Apply threading model**:

3. Select **Apply reader-writer-synchronized threading model** and click **Next**:
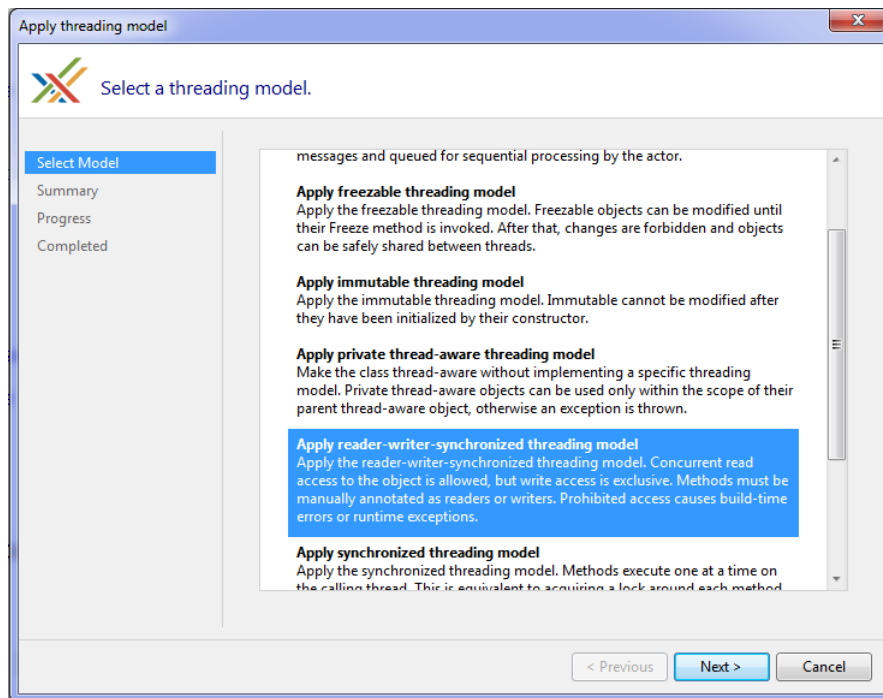
4. Click **Next** on the summary screen and the click on **Finished** to complete the process. The `ReaderWriter-SynchronizedAttribute` attribute is now applied to the class.

5. Once `ReaderWriterSynchronizedAttribute` is applied to the class, each method or property which accesses member data must then be marked as a reader or a writer. Position the caret on the name of the method or on the get or set keyword and choose "Acquire reader lock" or "Acquire writer lock" from the smart tag dropdown.

To apply ReaderWriterSynchronizedAttribute to a class manually:

1. Add NuGet Package `PostSharp.Patterns.Threading`.

2. Add `using PostSharp.Patterns.Threading`.

3. Add the custom attribute `[ReaderWriterSynchronizedAttribute]` to the class.

4. 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 following code shows the `Order` class , synchronized with the reader-writer threading pattern:

```csharp
[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;
    }
}
```

`ReaderAttribute` places a lock on the instance whenever the property or method is invoked. While this lock is held, other threads can also invoke a property or method of that instance which reads, but calls to properties or methods marked with `WriterAttribute` will be blocked until all reads are complete.

Likewise, invoking properties or methods marked with `WriterAttribute` will lock the instance causing reads to block until the write has completed and the write lock has been released.

Since `ReaderWriterSynchronizedAttribute` requires that all properties and methods which access member data be marked with `ReaderAttribute` or `WriterAttribute`, `ReaderWriterSynchronizedAttribute` throws an exception when an accessor does not have one of these attributes. This ensures that unsynchronized reads and writes are caught the instant they occur.

NOTE

Property getters are intrinsically thread-safe and don't need to be marked with a [`ReaderAttribute`] custom attribute.

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 as been marked with `WriterAttribute`:

```
[ReaderWriterSynchronized]
class Order
{
    // Other details skipped for brevity.

    public decimal 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 is does not acquire write access by default.
        [Writer] private set;
```

```
    }

    [UpgradeableReader]
    public void 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 Relationships [91].

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 [200] article to determine which threading model will work for the children of the Read/Writer Synchronized object.

12.5. 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) [188]
- Applying the Synchronized Model using the UI [189]
- Applying Synchronized manually [191]
- Working with object trees [191]

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(my-LockingObject)` 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 using the UI

To apply the Synchronized threading model to your class with the UI:

1. Place the cursor over your class name and select "Apply threading model..." from the drop down
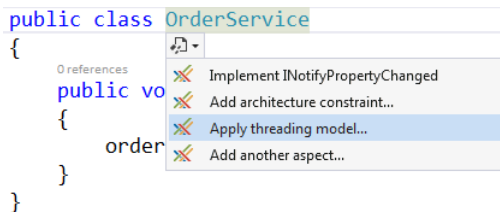
2. Select "Apply Synchronized threading model" from the Select Model tab and select "Next".
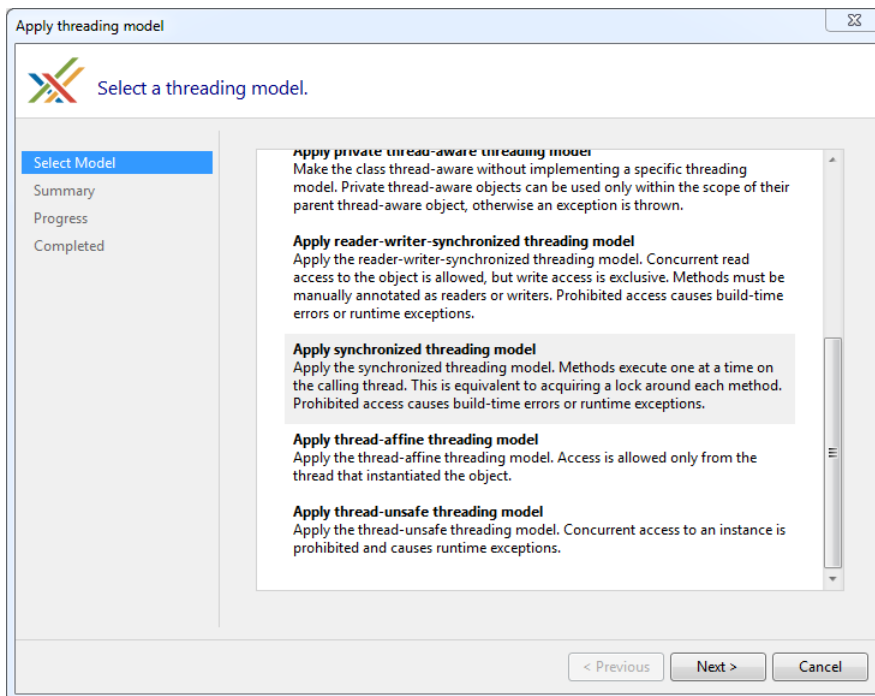
3. Confirm the actions on the Summary tab and select "Next".
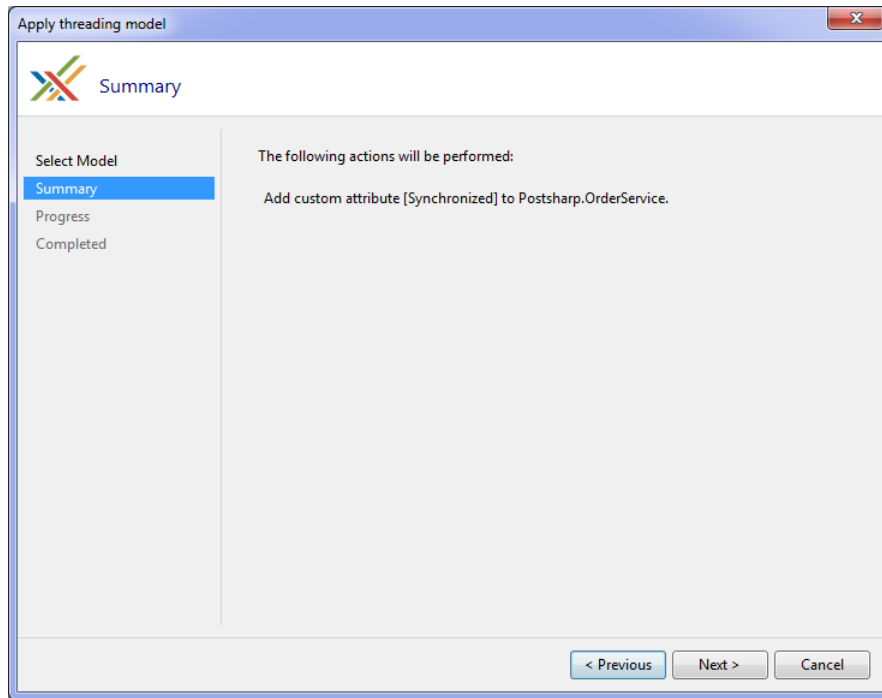
4. Click **Finish** after the installation completes.
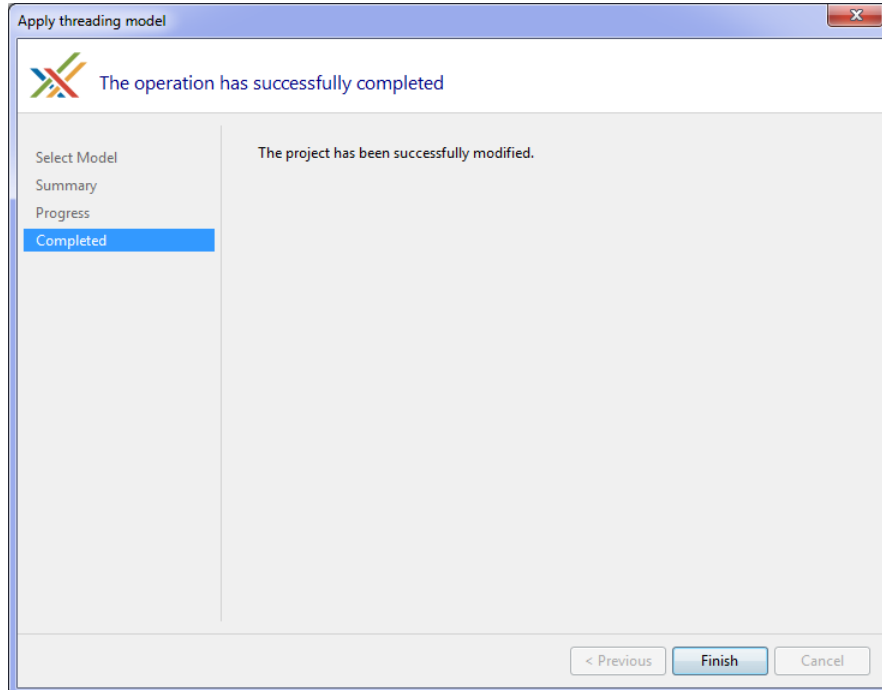
Your class will now have the `SynchronizedAttribute` and any previously missing PostSharp related references will have been added to the project.

Applying Synchronized manually

To apply the Synchronized threading model manually:

1. Add the PostSharp.Patterns.Threading NuGet package to your project.

2. Add the SynchronizedAttribute to the class.

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(string args[])
{
    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 follwing.

```
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
```

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 Relationships [91].

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 [200] article to determine which threading model will work for the children of the Synchronized object.

12.6. Thread-Unsafe Threading Model

When you are dealing with multi-threaded 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 proof 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.

- Marking an object as thread-unsafe [192]

Marking an object as thread-unsafe

To mark an object as thread-unsafe:

1. When you don't want multiple threads to access a single instance of a given class you will want to configure `InstanceLevelAspect` thread safety. To do this, select "Apply threading model" from the smart tag on the class.

2. Choose the "Apply thread-unsafe threading model" option.
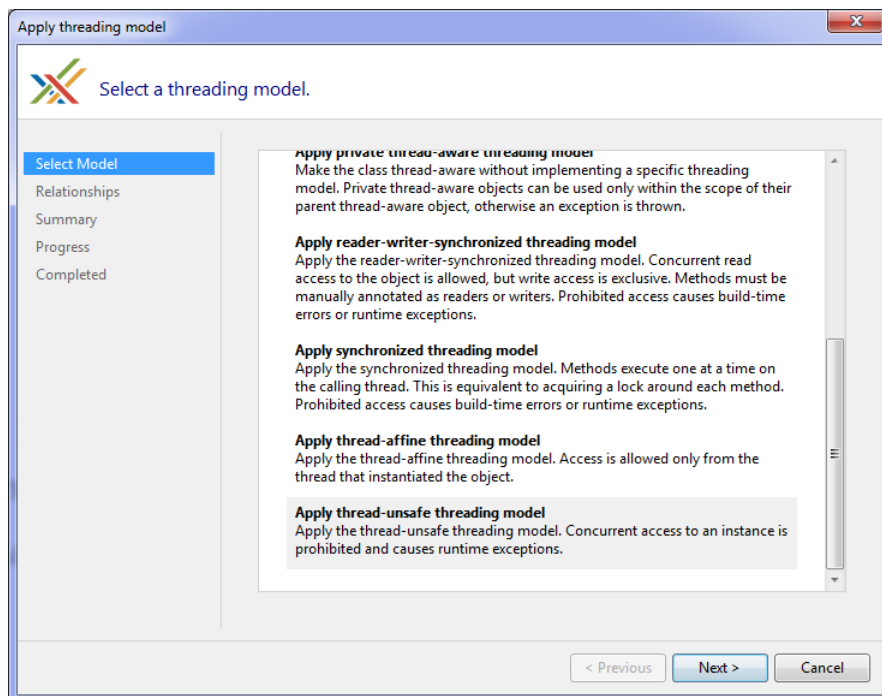
3. You will be prompted with a summary of the changes that will be made based on the configuration you selected in the wizard.
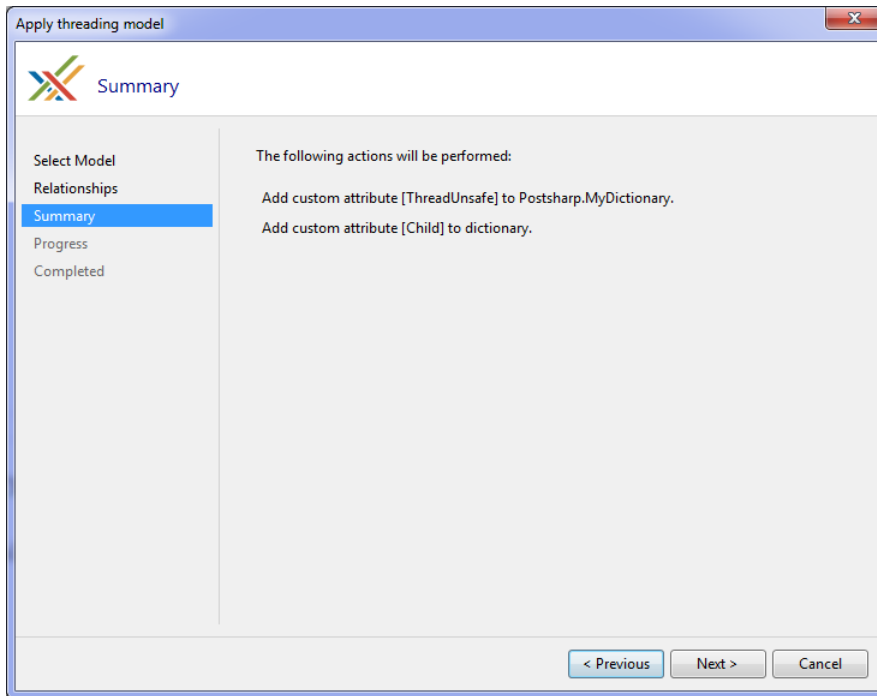
4. PostSharp will download the Threading Pattern Library and add it to your project if that hasn't been done yet.
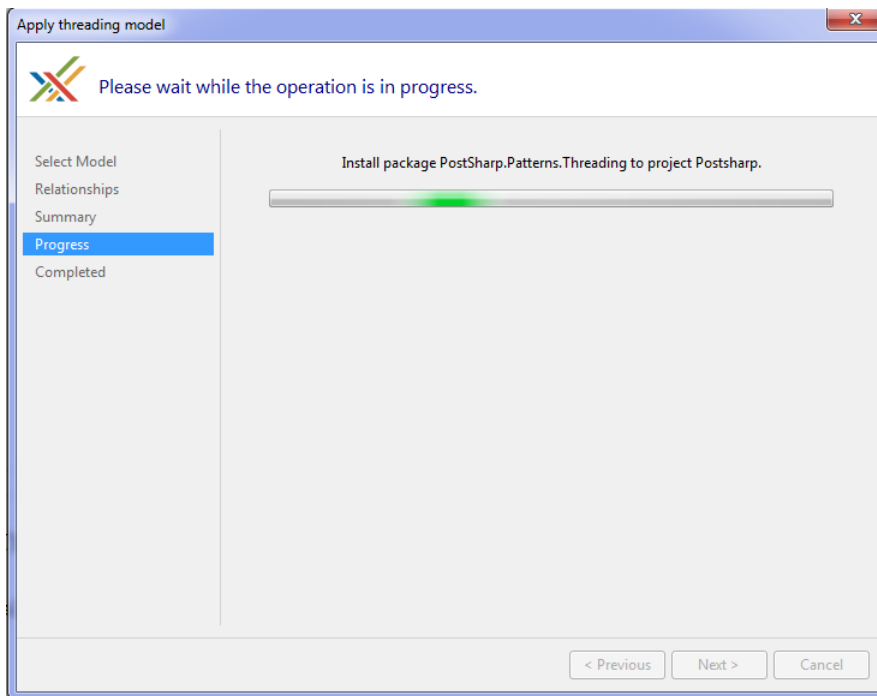
5. Once the process has completed successfully you'll be presented with the final page of the wizard.
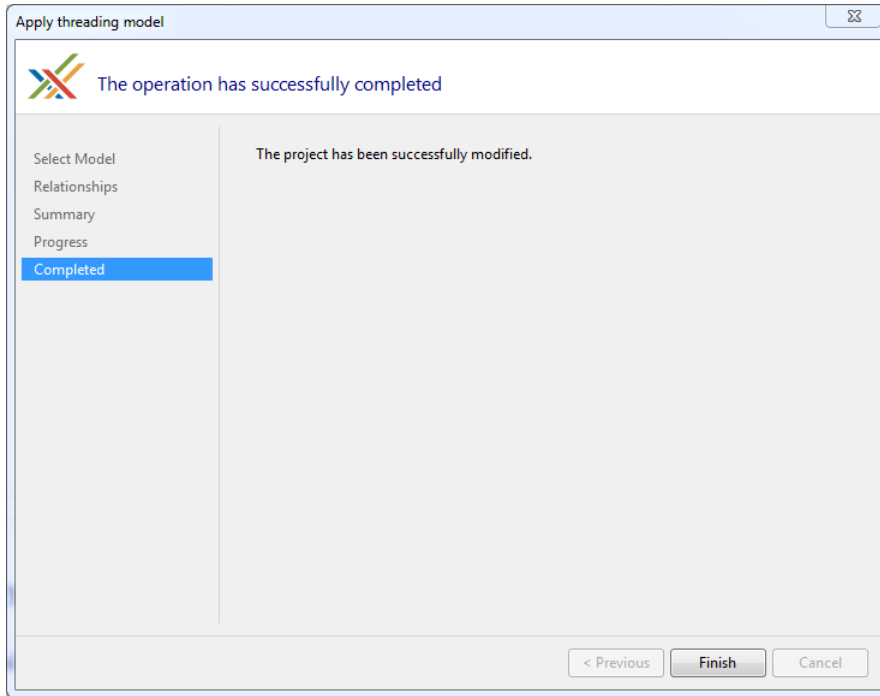
6. You'll notice that only one change was made to your codebase. The [`ThreadUnsafeAttribute`] attribute was added to the class you were targeting.

```
[ThreadUnsafe]
class AverageCalculator
{
    float sum;
    int count;

    public void AddSample(float n)
    {
        this.count++;
        this.sum += n;
    }

    public float GetAverage()
    {
        return this.sum / this.count;
    }
}
```

Now when your application executes no two threads will be able to access a single instance of the `AverageCalculator` class at the same time. If two threads attempt to do this, the second thread will receive a `ConcurrentAccessException`. Without the exception, there would be a slight chance that internal `sum` or `count` fields would have invalid values because the increment operations would not be atomic.

12.7. 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 using the UI [195]
- Adding Thread Affine manually [197]
- Understanding runtime enforcement [197]
- Working with object trees [198]

Adding the Thread Affine model using the UI

To apply the Thread Affine threading model to your class with the UI:

1. Place the cursor over your class name and select "Apply threading model..." from the drop down
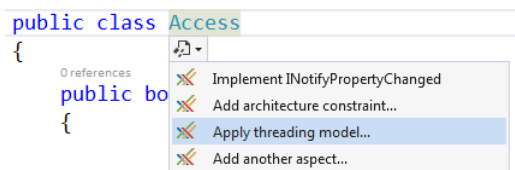
2. Select "Apply thread-affine threading model" from the Select Model tab and select "Next".
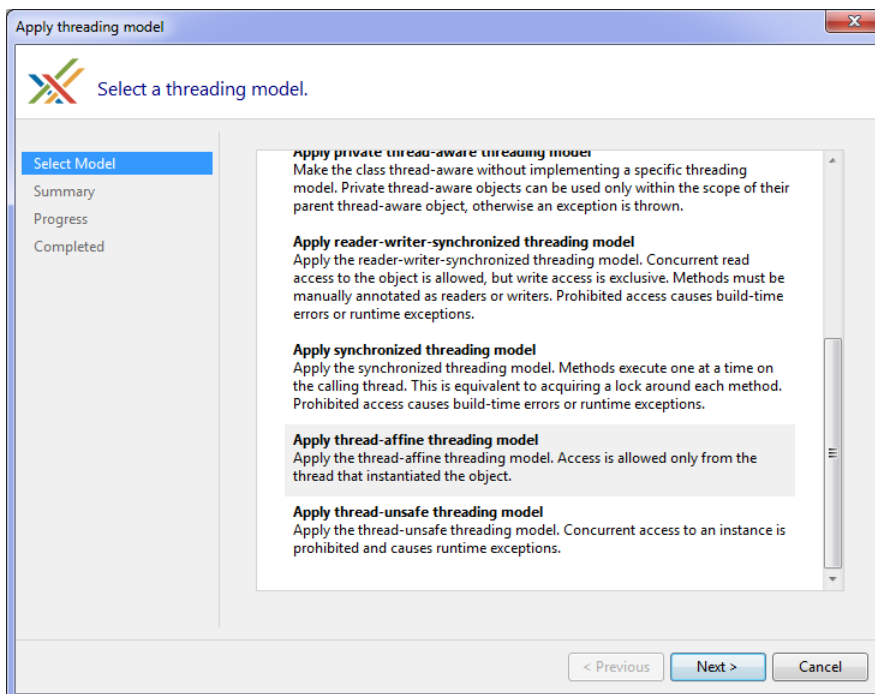
3. Confirm the actions on the Summary tab and select "Next".
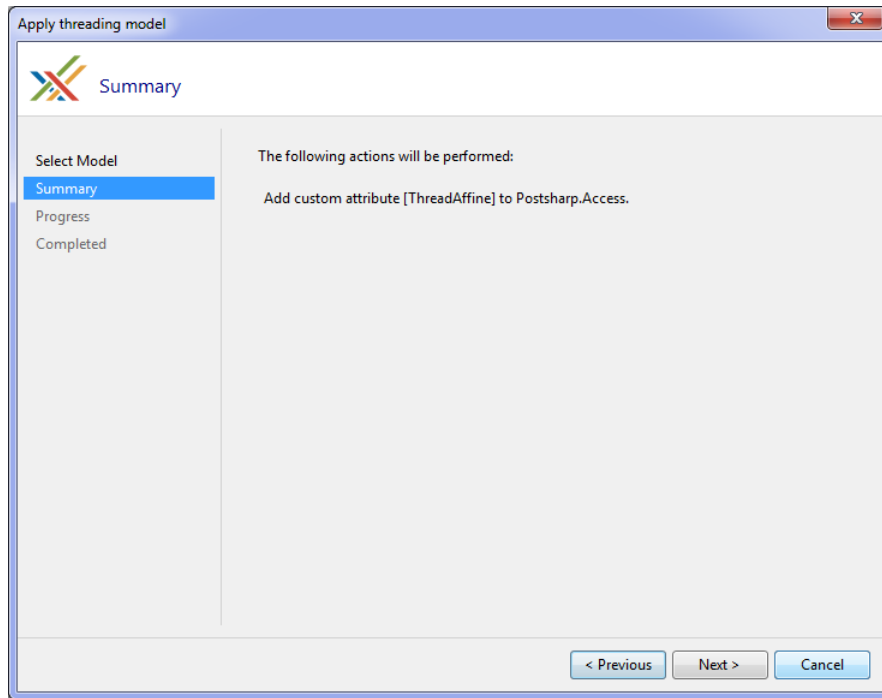
4. Click **Finish** after the installation completes.
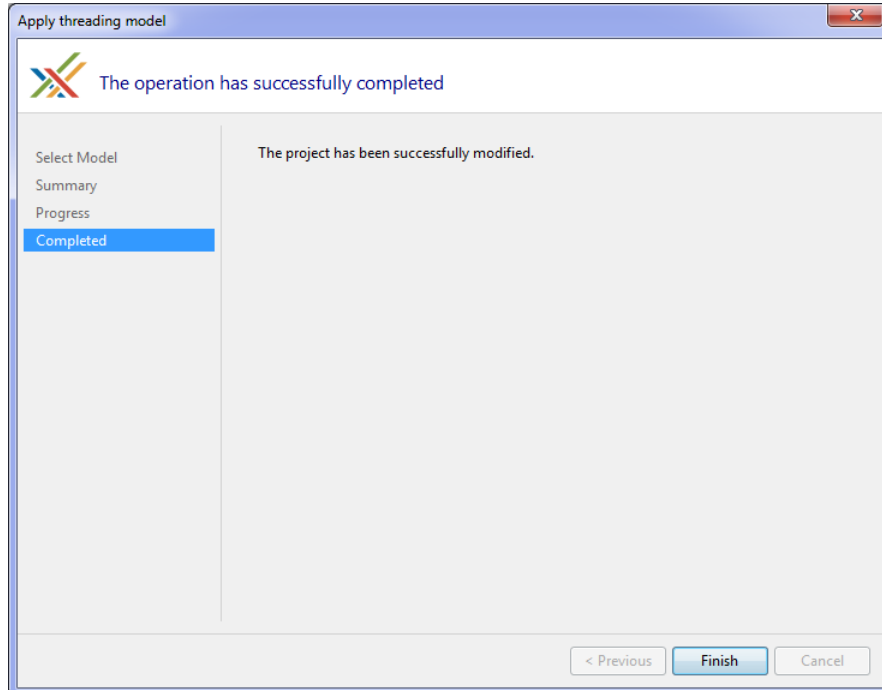
Your class will now have the `ThreadAffineAttribute` aspect and any previously missing PostSharp related references will have been added to the project.

Adding Thread Affine manually

To apply the Thread-Affine threading model manually:

1. Add the *PostSharp.Patterns.Threading* NuGet package to your project.

2. Add the `ThreadAffineAttribute` to the class.

In the example below the `ThreadAffineAttribute` has been added to the class.

```csharp
[ThreadAffine]
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));
    }
}
```

Understanding runtime enforcement

The `ThreadAffineAttribute` does not verify your code at build-time. Instead, it injects code that enforces the model at runtime. 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:

```csharp
public void Main(string args[])
{
    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.

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 Relationships [91].

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 [200] article to determine which threading model will work for the children of the Thread-Affine object.

12.8. 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 [198]
- Opting out from thread-safety for a field [199]
- Opting in for thread safety for callback methods [199]

Opting out from thread-safety verification for a method

To disable enforcement of the class-level threading model for a specific method, add the `ExplicitlySynchronized-Attribute` 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; } }

}
```

12.9. Compatibility of Threading Models

Required introduction

Compatibility Matrix

Parent	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								

12.10. 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

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.
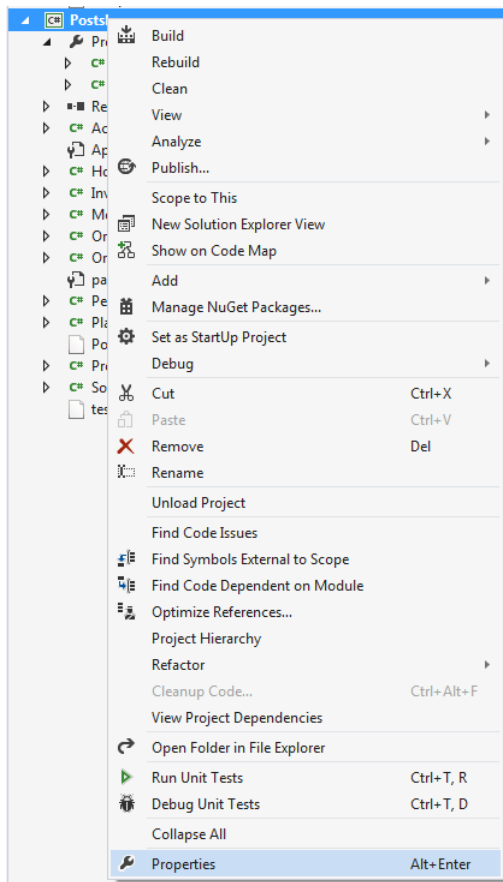
2. Select the build configuration that you want to enable runtime verification on.

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.

---
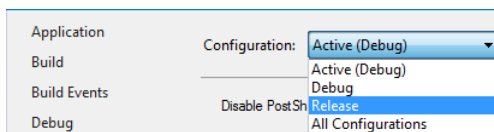
3. Open the PostSharp tab.
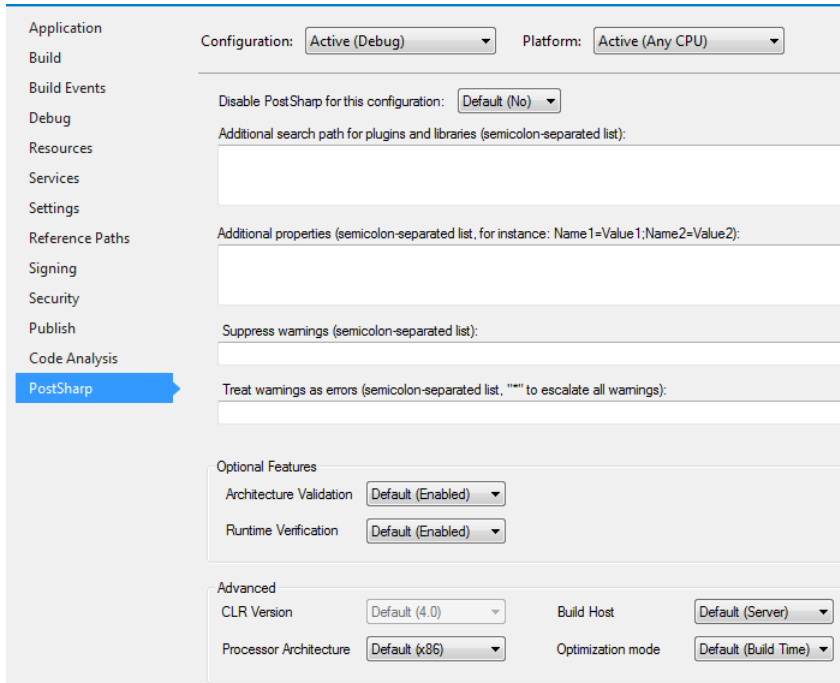
4. In the Optional Features section there is a Runtime Verification dropdown. The dropdown has three options in it; Default, Disabled, and Enabled.

   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 dropdown option will read Default (Enabled) and if the Optimize Code flag is enabled the dropdown option will read Default (Disabled).
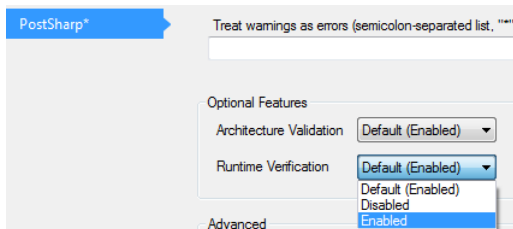
Enabling and disabling runtime verification for a specific class

You can override the project-level configuration of the runtime verification setting by setting the `RuntimeVerification-Enabled` 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`.

CHAPTER 13
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 asyncronous processing and the `BackgroundWorker`. Both require a lot of boiler plate code to push execution to another thread.
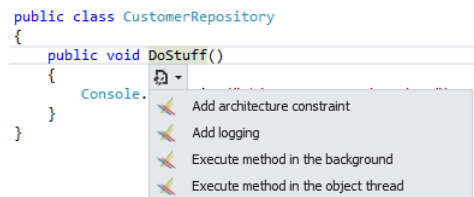
PostSharp provides you with the ability to push execution of a method to a background thread without having to worry about all of the boiler plate code.

To add [Background] attribute:

1. Find the method that you want to push to the background for execution.

```csharp
public class CustomerRepository
{
    public void DoStuff()
    {
        Console.WriteLine("Things are getting done");
    }
}
```

2. Select "Execute method in the background" from the Smart Tag available under the method name.

3. You'll notice that only one change was made to your codebase. The `[Background]` attribute was added to the class you were targeting. Now when this method executes in your application it will occur in another thread and will allow for the calling code to continue executing.

```csharp
public class CustomerRepository
{
    [Background]
    public void DoStuff()
    {
        Console.WriteLine("Things are getting done");
    }
}
```

Those simple steps are all that is required for you to declare that a method should be executed in a background thread.

CHAPTER 14
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

Once you've added background processing to the button click, you will need a way to have the `SaveCompleted` method run on the UI thread. If you don't do this you will get an `InvalidOperationException` because setting the `lblStatus.Text` property is a cross threading operation.
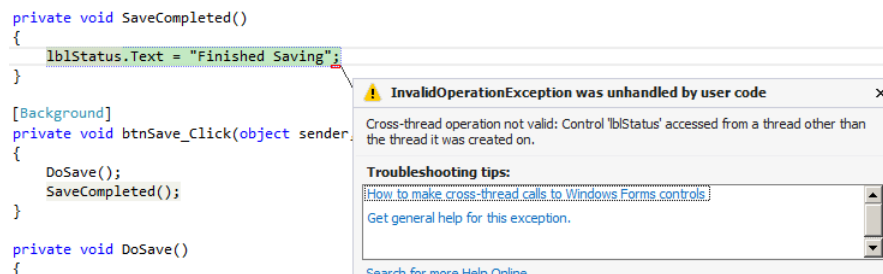
Here's how you can fix this:

1. Your code has encapsulated the UI thread interaction in the `SaveCompleted` method. Open the smart tag on that method and choose to "execute the method in the object thread". This tells PostSharp to configure this method to execute in the thread that the class containing the method is executing in.
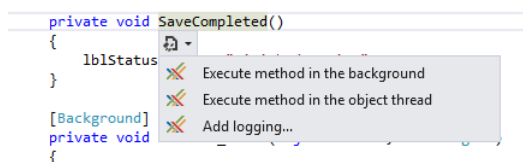
2. After selecting the smart tag option you will be returned to your code and you'll notice that the only change was the addition of the [DispatchedAttribute] attribute to the SaveCompleted method.

```
[Dispatched]
private void SaveCompleted()
{
    lblStatus.Text = "Finished Saving";
}
```

> **NOTE**
>
> Note, that [Dispatched] attribute can be applied only to instance methods of UI controls (WinForms or WPF), or to any class implementing the IDispatcherObject interface manually.

Now if you run your code you will no longer receive the InvalidOperationException and instead will see the label on the UI update. All of the Save functionality will occur in a separate thread which prevents the user interface from locking up while that is happening.

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)]
private void SaveCompleted()
{
    lblStatus.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 15
Detecting Deadlocks at Runtime

A common problem that is found in multi-threaded 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 the UI

To apply the deadlock detection to your application with the UI:

1. Right click on your solution in Solution Explorer, select Add followed by Postsharp Policy...
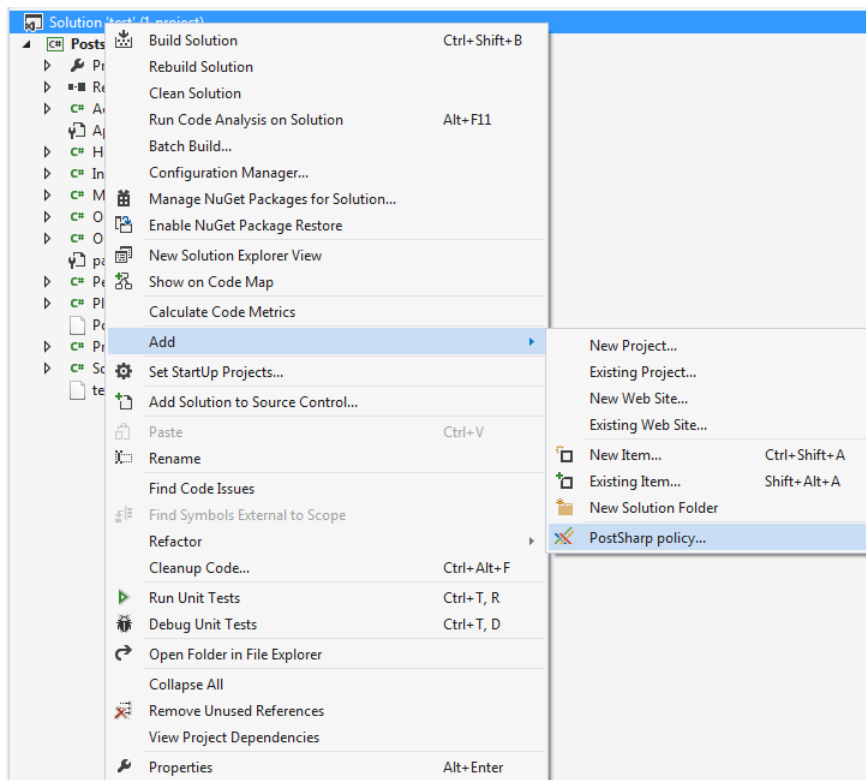
2. In the Add Postsharp policy wizard, expand Threading and select Deadlock detection.
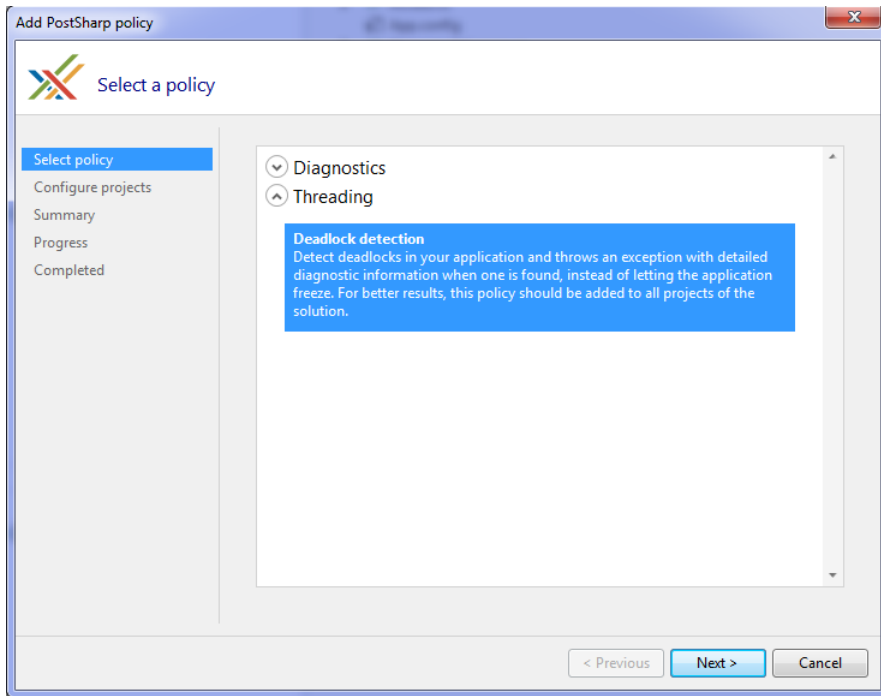
3. Select the projects that you would like to add Deadlock detection to.

NOTE

You will need to add this to every project in your application. Excluding projects could cause your application to fail.
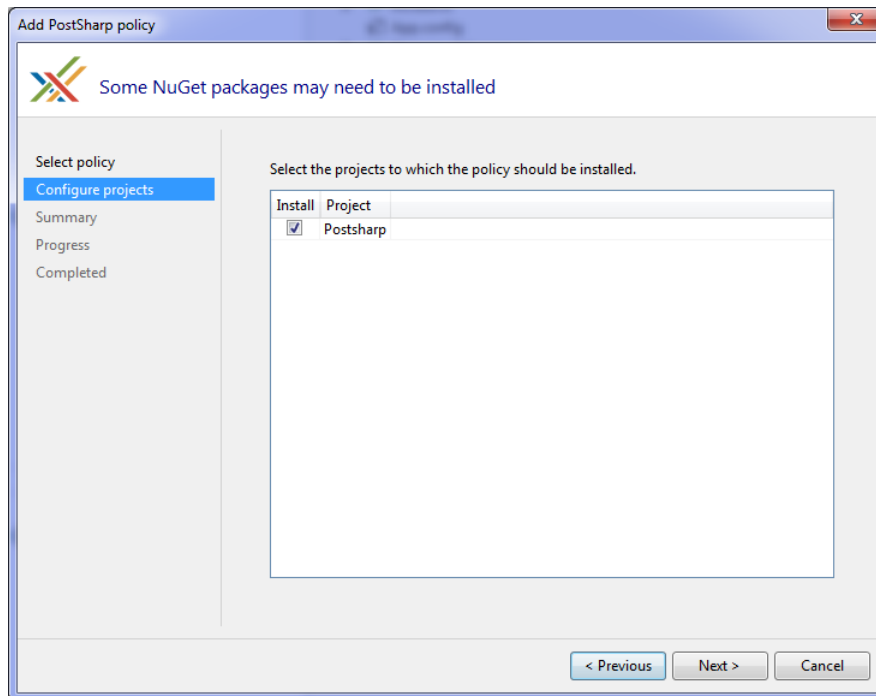
4. Review the configuration that you have selected and click Next.
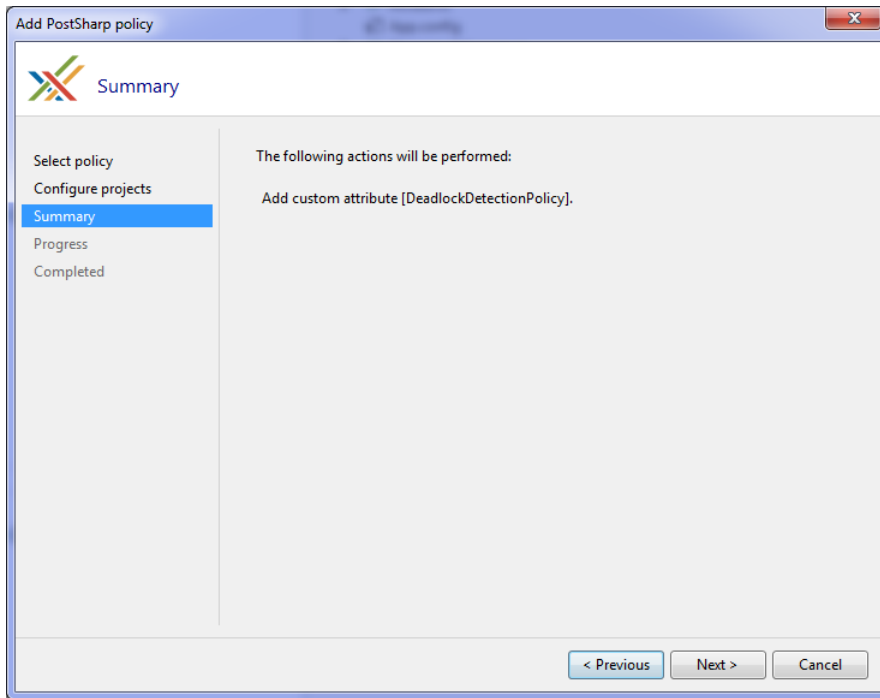
5. Close the wizard when the process had completed by clicking Finish.
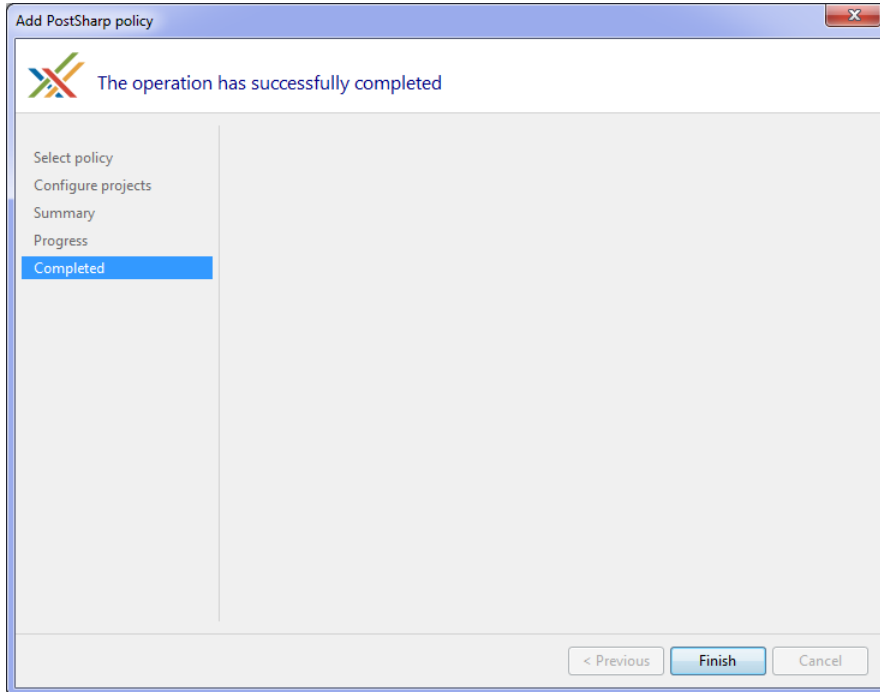
The result of running this wizard is that a pssln file has been added to your project.
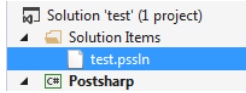
The pssln file contains an entry that enables deadlock detection across all projects in your solution.

```
<Projectxmlns="http://schemas.postsharp.org/1.0/configuration"xmlns:t="clr-namespace:PostSharp.Patterns.Threading;assembly:Pos
  <Multicast>
    <t:DeadlockDetectionPolicy/>
  </Multicast>
</Project>
```

Manually adding deadlock detection to a project

To manually add deadlock detection you will need to add an assembly level attribute to the project. This can be added anywhere in the project but it is most commonly added to the AssemblyInfo file.

```
[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 pssln file

Adding deadlock detection at a solution level can also be done manually. This can be done by adding an entry to the pssln file in the solution.

To manually add deadlock detection to a solution:

1. Open the solution's pssln file. This can be found under the Solution Items folder in Visual Studio's Solution Explorer.
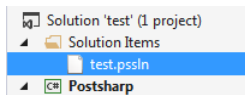
   If the pssln file doesn't exist manually add the file at the solution level. Name the file with the same name as your solution and the pssln file extension.

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 procede to the next step.

   ```
   <?xmlversion="1.0"encoding="utf-8"?>
   <Projectxmlns="http://schemas.postsharp.org/1.0/configuration"xmlns:t="clr-namespace:PostSharp.Patterns.Threading;as
   </Project>
   ```

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-namespace:PostSharp.Patterns.Threading;as
     <Multicast>
       <t:DeadlockDetectionPolicy/>
     </Multicast>
   </Project>
   ```

Once you save the pssln 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.Ob
ject: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, B
oolean& lockTaken)
    at System.Threading.Monitor.TryEnter(Object obj, Int32 millisecondsTimeout, B
oolean& lockTaken)
    at PostSharp.Patterns.Threading.DeadlockDetectionPolicy.MonitorEnhancements.<
>c__DisplayClass3f.<EnterInternal>b__3b(Nothing _, Int32 timeout)
    at PostSharp.Patterns.Threading.DeadlockDetectionPolicy.Helper.NoTimeoutAcqui
re[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.E
nterInternal(Object obj, Boolean& lockTaken)
    at PostSharp.Patterns.Threading.DeadlockDetectionPolicy.MonitorEnhancements.E
nter_Object_Bool(MethodInterceptionArgs args)
    at PostSharp.ImplementationDetails_c23235bd.<>z__Aspects.<System.Threading.Mo
nitor.Enter>b__(Object obj, Boolean& lockTaken) in :line 0
    at Postsharp.MonthEndProcessing.ProcessSalaries() in e:\working\postsharp3.1w
riting\test\MonthEndProcessing.cs:line 15
    at Postsharp.Program.<>c__DisplayClass1.<Deadlock>b__0(Object sender, DoWorkE
ventArgs args) in e:\working\postsharp3.1writing\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(IMe
ssage msg, IMessageSink replySink)
    at System.Runtime.Remoting.Proxies.AgileAsyncWorkerItem.ThreadPoolCallBack(Ob
ject o)
    at System.Threading.QueueUserWorkItemCallback.WaitCallback_Context(Object sta
te)
    at System.Threading.ExecutionContext.RunInternal(ExecutionContext executionCo
ntext, ContextCallback callback, Object state, Boolean preserveSyncCtx)
    at System.Threading.ExecutionContext.Run(ExecutionContext executionContext, C
ontextCallback callback, Object state, Boolean preserveSyncCtx)
    at System.Threading.QueueUserWorkItemCallback.System.Threading.IThreadPoolWor
kItem.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.AnalyzeDead
lockCycle(IEnumerable`1 cycle, StringBuilder messageBuilder, Thread[]& threadsIn
Deadlock, StackTrace[]& stackTraces)
    at PostSharp.Patterns.Threading.DeadlockDetection.DeadlockMonitor.ProcessDead
lock(IEnumerable`1 cycle)
    at PostSharp.Patterns.Threading.DeadlockDetection.DeadlockMonitor.DetectDeadl
ocksInternal(Thread startThread)
    at PostSharp.Patterns.Threading.DeadlockDetectionPolicy.Helper.NoTimeoutAcqui
re[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.E
nterInternal(Object obj, Boolean& lockTaken)
    at PostSharp.Patterns.Threading.DeadlockDetectionPolicy.MonitorEnhancements.E
nter_Object_Bool(MethodInterceptionArgs args)
    at PostSharp.ImplementationDetails_c23235bd.<>z__Aspects.<System.Threading.Mo
nitor.Enter>b__(Object obj, Boolean& lockTaken) in :line 0
    at Postsharp.MonthEndProcessing.ProcessBonuses() in e:\working\postsharp3.1wr
iting\test\MonthEndProcessing.cs:line 26
    at Postsharp.Program.Deadlock() in e:\working\postsharp3.1writing\test\Progra
m.cs:line 34
```

Custom Patterns

CHAPTER 16

Developing Custom Aspects

This chapter describes how to build your own aspect. It includes the following topics:

16.1. Developing Simple Aspects

In PostSharp, developing an aspect is as simple as deriving a primitive aspect class and overriding some special methods named *advice*. Aspects encapsulate a transformation of an element of code (such as a method or a property), and advices are the methods that are executed at runtime.

For instance, the effect of the aspect `OnMethodBoundaryAspect` is to wrap the target method into a `try`/`catch`/`finally` construct, and the advices of this aspect are `OnEntry(MethodExecutionArgs)`, `OnSuccess(MethodExecutionArgs)`, `OnException(MethodExecutionArgs)` and `OnExit(MethodExecutionArgs)`

By default, advices of primitive aspect types have an empty implementation, so the aspect has no effect until you override at least one advice.

To develop a simple aspect:

1. Add PostSharp to your project. See Installing PostSharp Tools for Visual Studio [34] for details.

2. Create a new class and make it derive from one of the primitive aspect classes (see below).

3. Annotate the class with the custom attribute `SerializableAttribute`, or `PSerializableAttribute` if your project targets anything else than the full .NET Framework. See Understanding Aspect Lifetime and Scope [259] to understand why.

4. Override one of the aspect advice methods.

Aspect Classes

The following table gives a list of available primitive aspect classes. Every aspect class is described in greater detailed in the class reference documentation.

Aspect Type	Targets	Description
`OnMethodBoundaryAspect`	Methods	Methods enhanced with an `OnMethodBoundaryAspect` are wrapped by a `try`/`catch`/`finally` construct. This aspect provides the advices `OnEntry(MethodExecutionArgs)`, `OnSuccess(MethodExecutionArgs)`, `OnException(MethodExecutionArgs)` and `OnExit(MethodExecutionArgs)`; these advices are invoked directly from the transformed method, the return value, and the exception (if applicable). This aspect is useful to implement tracing or transaction handling, for instance. For details, see Injecting Behaviors Before and After Method Execution [220].
`OnExceptionAspect`	Methods	Methods enhanced with an `OnExceptionAspect` are wrapped by a `try`/`catch` construct. This aspect provides the advice `OnException(MethodExecutionArgs)`; this advice is invoked from the `catch` block. This aspect is useful to implement exception handling policies. Contrarily to `OnMethodBoundaryAspect`, this aspect lets you define the type of caught exceptions by overriding the method `GetExceptionType(MethodBase)` For details, see Handling Exceptions [225].
`MethodInterceptionAspect`	Methods	When a method is enhanced by a `MethodInterceptionAspect`, all calls to this method are replaced by calls to `OnInvoke(MethodInterceptionArgs)`, the only advice of this aspect type. This aspect is useful when the execution of target method can be deferred (asynchronous calls), must be dispatched on a different thread. For details, see Intercepting Methods [240].

Aspect Type	Targets	Description
LocationInterceptionAspect	Fields, Properties	When a field or a property is enhanced by a LocationInterceptionAspect, all calls to its accessors are replaced by calls to advices OnGetValue(LocationInterceptionArgs) and OnSetValue(LocationInterceptionArgs). Fields are transparently replaced by properties. This aspect is useful to implement functionalities that need to get or set the location value, such as the observability design pattern (INotifyPropertyChanged). For details, see Intercepting Properties and Fields [245].
EventInterceptionAspect	Events	When an event is enhanced by an EventInterceptionAspect, all calls to its add and remove semantics are replaced by calls to advices OnAddHandler(EventInterceptionArgs) and OnRemoveHandler(EventInterceptionArgs). Additionally, when the event is fired, even of invoking directly the handlers that were added to the event, the advice OnInvokeHandler(EventInterceptionArgs) is called instead. This aspect is useful to add functionalities to events, such as implementing asynchronous events or materialized list of subscribers. For details, see Intercepting Events [250].
CompositionAspect	Types	This aspect introduces an interface into a type by composition. The interface is introduced statically; the aspect method GetPublicInterfaces(Type) should return the type of introduced interfaces. However, the object implementing the interface is created dynamically at runtime by the implementation of the method CreateImplementationObject(AdviceArgs). For details, see Introducing Interfaces [252].
CustomAttributeIntroductionAspect	Any	This aspect introduces a custom attribute on any element of code. A custom attribute can be represented as a CustomAttributeData or a ObjectConstruction. For details, see Introducing Custom Attributes [254].
ManagedResourceIntroductionAspect	Assemblies	This aspect introduces a managed resource into the current assembly. For details, see Introducing Managed Resources [258].
ILocationValidationAspect	Fields, Properties, Parameters	This aspect causes any new value assigned to its target to be validated. If the aspect determines the value is invalid, an exception is thrown. The aspects of the PostSharp.Patterns.Contracts namespace are built on the top of this interface aspect. For details, see Contracts [121].

TIP

The implementation of aspects OnMethodBoundaryAspect and OnExceptionAspect is very efficient; they should be preferred over other aspects whenever it makes sense.

Using Aspect Interfaces

The primitive aspect classes listed above only exist for convenience. In reality, PostSharp only understands interfaces. Every of these aspect classes implements a pair of interfaces. For instance, the class `OnMethodBoundaryAspect` implements the interfaces `IOnMethodBoundaryAspect` and `IMethodLevelAspectBuildSemantics`.

The aspect classes are more convenient because they derive from `MulticastAttribute`, which extends `SystemAttribute` with multicasting capability. See Adding Aspects to Multiple Declarations [147] for details.

If you do not need or want the capabilities of `MulticastAttribute` (for instance because the aspect is not used as a custom attribute, see `IAspectProvider`), you can implement the aspect interface manually. An aspect class must implement an interface derived from `IAspect`, and may implement an interface derived from `IAspectBuildSemantics`. Please refer to the documentation of the aspect class to get information about the corresponding aspect interface.

Additionally to the aspect interface corresponding to an aspect class, you can define the following interfaces on aspect classes:

Aspect Interface	Description
`IAspectProvider`	This interface defines a single method `ProvideAspects(Object)`, returning a collection of `AspectInstance`. The method allows an aspect to dynamically provide other aspects to the weaver.
`IInstanceScopedAspect`	By default, aspects have static scope: there is one instance of the aspect per target class. Implementing the `IInstanceScopedAspect` makes the aspect instance-scoped: there will be one instance of this aspect per *instance* of the target class.

16.1.1. Injecting Behaviors Before and After Method Execution

There are two ways to inject behaviors into methods. The first is the *method decorator*: it allows you to add instructions before and after method execution. The second is method *interception*: the hook gets invoked instead of the method. Decorators are faster than interceptors, but interceptors are more powerful. The current article covers decorators. For the other aspect, see Intercepting Methods [240].

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`.

Injection points

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`.

To create a simple aspect that writes some text whenever a method enters, succeeds, or fails:

1. Create an aspect class and inherit `OnMethodBoundaryAspect`. Annotate the class with the [`Serializable-Attribute`] custom attribute.

> **NOTE**
>
> Use [`PSerializableAttribute`] instead of [`SerializableAttribute`] if your project targets Silverlight, Windows Phone, Windows Store, or runs with partial trust.

2. To add functionality prior to the execution of the target method, override the method and code the functionality you desire.

```
[Serializable]
public class LoggingAspect : OnMethodBoundaryAspect
{
  public override void OnEntry(MethodExecutionArgs args)
  {
    Console.WriteLine("The {0} method has been entered.", args.Method.Name);
  }
}
```

3. Inject functionality immediately after the method executes by overriding the `OnExit(MethodExecutionArgs)` method.

> **NOTE**
>
> It's important to remember that the `OnExit(MethodExecutionArgs)` method will execute every time that the target method completes its execution regardless of if the target method completed successfully or threw an exception.

```
public override void OnExit(MethodExecutionArgs args)
{
  Console.WriteLine("The {0} method has exited", args.Method.Name);
}
```

4. To add functionality that only executes when the target method has completed successfully, you will override the `OnSuccess(MethodExecutionArgs)` method in your aspect. The `OnSuccess(MethodExecutionArgs)` method will be executed every time that the target method completes successfully. If the target method throws an exception `OnSuccess(MethodExecutionArgs)` will not execute.

```
public override void OnSuccess(MethodExecutionArgs args)
{
  Console.WriteLine("The {0} method executed successfully.", args.Method.Name);
}
```

5. The final location that you can intercept requires you to override the `OnException(MethodExecutionArgs)` method. As the name of the overrode method suggests, this is where you can inject functionality that should execute when the target method throws and exception.

```
public override void OnException(MethodExecutionArgs args)
{
  Console.WriteLine("An exception was thrown in {0}.", args.Method.Name);
}
```

The four methods (`OnEntry(MethodExecutionArgs)`, `OnExit(MethodExecutionArgs)`, `OnSuccess(MethodExecutionArgs)` and `OnException(MethodExecutionArgs)`) that you overrode are the locations that you are able to intercept method execution. Between these four location you are able to implement many different infrastructure patterns with minimal effort.

Accessing the method

As illustrated in the examples above, you can access information about the method being intercepted from the property `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.

Accessing parameters

It's rare that you will intercept method execution and not interact with the parameters that were passed to the target method. For example, when you implement method interception for logging you will probably want to log the parameter values that were passed to the target method.

Each of the interception locations that were outlined earlier has access to that information. If you look at the `On-Entry(MethodExecutionArgs)` method in your aspect you will see that it has a `MethodExecutionArgs` parameter. That parameter is used for `OnExit(MethodExecutionArgs)`, `OnSuccess(MethodExecutionArgs)` and `OnException(Method-ExecutionArgs)` as well. The collection `Arguments` gives access to parameter values.

Let's modify the `OnEntry(MethodExecutionArgs)` method and include the parameter values in the log message.

To include argument values to the logged text:

1. Create a `foreach` loop to gather each of the parameter values in the `Arguments` property of the `args` parameter.

2. In the loop concatenate the parameter values into a string.

3. Pass that string of argument values to the logging tool.

```csharp
 public override void OnEntry(MethodExecutionArgs args)
{
  var argValues = new StringBuilder();
  foreach (var argument in args.Arguments)
  {
        argValues.Append(argument.ToString()).Append(",");
  }

  Console.WriteLine("The {0} method was entered with the parameter values: {1}",
               args.Method.Name, argValues.ToString());
}
```

> **NOTE**
>
> A production implementation of this aspect would need to take re-entrance into account. See the article Logging [129] for a ready-made logging aspect.

It's also possible to modify the parameter values inside your aspect methods. All you need to do is modify the value of the item in the `Arguments` collection. Remember that all items in the `Arguments` collection are object types so you will need to be careful with how you change values. If the value you are modifying was originally a string, you will want to ensure it stays a string type. It's especially true that when you change the parameter type in the `OnEntry(MethodExecutionArgs)` method you may cause the system to be unable to execute the target method due to a parameter type mismatch.

> **NOTE**
>
> The only parameter types that you can modify are those defined as either `out` or `ref`. If you need to modify input arguments, you should use Intercepting Methods [240].

Accessing the target objects

In combination with the parameters you will probably interact with the target code instance that the aspect is attached to. The `Instance` property provides you with the instance of the object that the aspect is currently operating against. It is an object type so you will need to cast it to the correct type to be able to interact with it. If you debug your aspect and that aspect doesn't make use of `Instance`, it will be set to null. It's also set to null if the target code is defined as static.

Accessing the return value

Like target method parameters you also have access to the return value for those target methods. It's possible to both read the return value as well as modify it. The return value can be found at `ReturnValue` in all four of the aspect methods covered earlier.

```
public override void OnExit(MethodExecutionArgs args)
{
    args.ReturnValue = false;
}
```

> **NOTE**
>
> If a target method is defined as void, the `ReturnValue` property will be set to null. `ReturnValue` is an object type so you must be careful how you modify the return value with respect to the return value type of the target code.

Changing execution flow

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, you may want to exit the execution of the target code at some point in the `OnEntry(Method-ExecutionArgs)` advice. PostSharp offers this ability through the use of `FlowBehavior`.

```
public override void OnEntry(MethodExecutionArgs args)
{
    if (args.Arguments.Count > 0 &amp;&amp; args.Arguments[0] == null)
    {
        args.FlowBehavior = FlowBehavior.Return;
    }

  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`.

> **NOTE**
>
> Using flow control to exit the target code execution will return back to the code that called the target code. As a result, you need to be considerate to the target code's return value. In the example above, the target code will always return null. This may or may not be the behavior that you want. If it isn't, you can set the `ReturnValue` on the `MethodExecutionArgs` and that value will be returned from the target code.

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.

Rethrowing an existing exception

To rethrow an existing exception, you will set the `FlowBehavior` property to `RethrowException`. Whatever exception that was caught in the `OnException(MethodExecutionArgs)` advice will be rethrown to the code that is calling the target code.

```
public override void OnException(MethodExecutionArgs args)
{
    if (args.Exception.GetType() == typeof(DivideByZeroException))
```

```
    {
        args.FlowBehavior = FlowBehavior.RethrowException;
    }
}
```

Throwing a new exception

To throw a new exception you will have to perform two tasks. First you will need to assign the new exception to the `Exception` property. This is the exception that will be thrown as part of the flow behavior. After that you will need to set the `FlowBehavior` property to `ThrowException`.

```
public override void OnException(MethodExecutionArgs args)
{
    if (args.Exception.GetType() == typeof(IndexOutOfRangeException))
    {
        args.Exception = new CustomArrayIndexException("This was thrown from an aspect",
                                                       args.Exception);
        args.FlowBehavior = FlowBehavior.ThrowException;
    }
}
```

> **NOTE**
>
> The remaining `FlowBehavior` enumeration value is `Continue`. In `OnException(MethodExecutionArgs)`, this behavior will not rethrow the caught exception. In `OnEntry(MethodExecutionArgs)`, `OnSuccess(MethodExecutionArgs)` and `OnExit(MethodExecutionArgs)` the target code execution will continue with no interruption.

> **NOTE**
>
> The default `FlowBehavior` value for `OnEntry(MethodExecutionArgs)`, `OnSuccess(MethodExecutionArgs)` and `OnExit(MethodExecutionArgs)` is `Continue`. For `OnException(MethodExecutionArgs)` the default value is `RethrowException`.

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]
public class ExecutionDurationAspect : 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);
```

```
    }
}
```

16.1.2. 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 where ever 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
- Specifying the type of handled exceptions
- Ignoring exceptions
- Replacing exceptions
- Displaying the method arguments on exception

Intercepting an exception

PostSharp provides the `OnExceptionAspect` class which is the base class from which exception handlers are to be derived from.

The key element of this class is the `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 OnExceptionAspect class:

1. Derive a class from `OnExceptionAspect`.

2. Apply the `SerializableAttribute` to the class.

3. Override `OnException(MethodExecutionArgs)` and implement your exception handling logic in this class.

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]
public class PrintExceptionAttribute : OnExceptionAspect
{

    public override void 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
{
    public string FirstName { get; set; }
    public string LastName { get; set; }

    [PrintException]
    public void 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.

Alternatively the attribute can be applied to the class itself as shown below, in which case the exception handler will handle exceptions for all methods and properties in the class:

```
[PrintExceptionAttribute(typeof(IOException))]
    class Customer
    {
      .
      .
      .
    }
```

See the section Adding Aspects to Multiple Declarations [147] for details about attribute multicasting.

Specifying the type of handled exceptions

The `GetExceptionType(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

The `GetExceptionType(MethodBase)` method is evaluated at build time.

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]
    public class PrintExceptionAttribute : OnExceptionAspect
    {
        Type type;

    public PrintExceptionAttribute() : this(typeof(Exception))
    {
    }

        public PrintExceptionAttribute (Type type)
            : base()
        {
            this.type = type;
        }

        // Method invoked at build time.
        // Should return the type of exceptions to be handled.
        public override Type GetExceptionType(MethodBase method)
        {
            return this.type;
        }
```

```
        public override void OnException(MethodExecutionArgs args)
        {
            Console.WriteLine(args.Exception.Message);
        }
    }
```

Example:

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

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

}
```

NOTE

If the aspect needs to handle several types of exception, the `GetExceptionType` should return a common base type, and the `OnException` implementation should be modified to dynamically handle different types of exception.

Ignoring 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` to `Return`:

```
[Serializable]
public class PrintAndIgnoreExceptionAttribute : OnExceptionAspect
{

    public override void OnException(MethodExecutionArgs args)
    {
        Console.WriteLine(args.Exception.Message);
        args.FlowBehavior = FlowBehavior.Return;
    }
 }
```

If a method returns a value then the `ReturnValue` member of *args* can be set to an object to return. For example, consider the following `GetDataLength` method in `Customer` which returns the number of characters read from a file:

```
 class Customer
{
    [PrintException(typeof(IOException))]
    public int GetDataLength(string path)
    {
return File.ReadAllText(path).Length;
    }

}
```

We can then modify the `OnException(MethodExecutionArgs)` method of `PrintAndIgnoreExceptionAttribute` to return an integer with a value of `-1`:

```
public override void OnException(MethodExecutionArgs args)
{
    Console.WriteLine(args.Exception.Message);
    args.FlowBehavior = FlowBehavior.Return;
    args.ReturnValue = -1;
}
```

Replacing exceptions

Many times an exception must be exposed to the user, either by allowing the original exception to be rethrown, or by throwing a new exception. This can be done by setting `FlowBehavior` as follows:

> `FlowBehavior.RethrowException`: rethrows the original exception after the exception handler exits. This is the default behavior for the `OnException(MethodExecutionArgs)` advise.

> `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 `BusinessException-Attribute` which throws a `BusinessException` containing a description of the cause:

```
[Serializable]
public sealed class BusinesssExceptionAttribute : OnExceptionAspect
{
    public override void OnException(MethodExecutionArgs args)
    {
        .
        .
        .
        args.FlowBehavior = FlowBehavior.ThrowException;
        args.Exception = new BusinessException("Bad Arguments", new Exception("One or more arguments were null. (
    }
}

class BusinessException : Exception
{
    public BusinessException(string message, Exception innerException) : base(message, innerException)
    {
    }
}
```

Displaying the method arguments on exception

When an exception is thrown, it can be useful to view and display the parameter values that were passed into the method where the exception occurred. These values can be retrieved by iterating through the `Arguments` property of the *args* parameter of the `OnException(MethodExecutionArgs)` method. In the following snippet, `OnException(MethodExecution-Args)` has been modified to iterate through all exception values, and to concatenate them into a string. If a null value is encountered, then the code embeds the word null into the string. This string is then displayed as the message of the `Null-ReferenceException` which is rethrown:

```
public override void OnException(MethodExecutionArgs args)
{
    string parameterValues = "";

    foreach (object arg in args.Arguments)
    {
        if (parameterValues.Length > 0)
        {
            parameterValues += ", ";
```

```
        }

        if (arg != null)
        {
            parameterValues += arg.ToString();
        }
        else
        {
            parameterValues += "null";
        }
    }

        Console.WriteLine( "Exception {0} in {1}.{2} invoked with arguments {3}", args.Exception.GetType().Name, args.Method
    }
}
```

> **NOTE**
>
> The `Arguments` field of args cannot be directly viewed in the debugger. The `Arguments` field must be referenced by another object in order to be viewable in the debugger.

16.1.3. Injecting Behaviors into Async Methods

Async methods are methods with the `async` keyword in C#. Unlike normal methods, execution of async methods can be paused and resumed. Execution is paused when the method depends on a task that is being executed asynchronously, and is resumed when the dependent task has completed. An async method provides a convenient way to do potentially long-running work without blocking the caller's thread.

Async methods return a value of the `Task` type. At build time, the compiler performs a complex transformation of the code. The original async logic is moved to a different type called here the *state machine type*, and the original method body is replaced by just a few lines of code instantiated this state machine.

By default, all aspects applied on an async method are actually applied to the method *instantiating* the state machine. However, there are times when you want to actually add an aspect to the state machine. We will see how this is possible using the `OnMethodBoundaryAspect` aspect. For a more general information about using `OnMethodBoundaryAspect` in non-async methods, see Injecting Behaviors Before and After Method Execution [220].

There are two options when applying `OnMethodBoundaryAspect` to an async method. The first option is to apply the aspect to the method that instantiates and returns the task instance. The second option is to apply the aspect to the actual code that implements the logic inside the task. When using the second option you can also inject behaviors before and after your `await` statements.

This topic contains the following sections.

Applying the aspect to the method instantiating the async task

You can control how the aspect is applied to the async method by setting the boolean `ApplyToStateMachine` property. Set this property to `false` to apply the aspect only to the code instantiating the async task. You can set the property in the constructor if you don't want to set it explicitly every time the aspect is used.

In the following procedure, we demonstrate how to inject behaviors into the method instantiating the async task. For this purpose, we create a simplified caching aspect that caches the result of the async method. This aspect can be applied to any async method returning `Task<object>`. For example, to the `TestCaching` method shown below:

```
public async Task<object> TestCaching()
{
    await Task.Wait(1000); // Some long-running operation.
    return await Task.FromResult( new {Name = "Test object"} );
}
```

To inject behaviors into the method instantiating the async task:

1. Create an aspect class and inherit `OnMethodBoundaryAspect`. Annotate the class with the [`Serializable-Attribute`] custom attribute.

   > **NOTE**
   >
   > Use [`PSerializableAttribute`] instead of [`SerializableAttribute`] if your project targets Silverlight, Windows Phone, Windows Store, or runs with partial trust.

2. Set the `ApplyToStateMachine` property to `false` in your constructor.

   ```
   [Serializable]
   public class CacheTaskResultAttribute : OnMethodBoundaryAspect
   {
       public CacheTaskResultAttribute()
       {
           ApplyToStateMachine = false;
       }
   }
   ```

   > **NOTE**
   >
   > The default value of the `ApplyToStateMachine` property is `false` unless the `OnYield(MethodExecutionArgs)` or `OnResume(MethodExecutionArgs)` advice is implemented. However, PostSharp will emit a warning whenever `On-MethodBoundaryAspect` is applied to an async method without setting the `ApplyToStateMachine` property.

3. Implement some of the advice methods of the OnMethodBoundaryAspect class, according to your requirements. For more information about these advice methods, see Injecting Behaviors Before and After Method Execution [220].

   For our CacheAttribute aspect, we need to implement two advises: OnEntry(MethodExecutionArgs) and On-Success(MethodExecutionArgs).

   ```
   [Serializable]
   public class CacheTaskResultAttribute : OnMethodBoundaryAspect
   {
       public CacheTaskResultAttribute()
       {
           ApplyToStateMachine = false;
       }

       public override void OnEntry( MethodExecutionArgs args )
       {
           object cachedValue = MemoryCache.Default[args.Method.Name];
           if ( cachedValue != null )
           {
               args.ReturnValue = Task.FromResult( cachedValue );
               args.FlowBehavior = FlowBehavior.Return;
           }
       }

       public override void OnSuccess( MethodExecutionArgs args )
       {
           var task = (Task<object>) args.ReturnValue;
           args.ReturnValue =
               task.ContinueWith(
                   t =>
                   {
                       MemoryCache.Default[args.Method.Name] = t.Result;
                       return t.Result;
                   });
       }
   }
   ```

4. Apply your custom attribute to the target methods. Below you can see the [Cache] attribute applied to the Test-Caching method.

   ```
   [Cache]
   public async Task<object> TestCaching()
   {
     await Task.Wait(1000); // Some long-running operation.
     return await Task.FromResult( new {Name = "Test object"} );
   }
   ```

   > **NOTE**
   >
   > You can also set the ApplyToStateMachine property value when applying the attribute to a method, for instance using the syntax [MyAspect(ApplyToStateMachine = false)]. This can be useful if it makes sense to apply this aspect to both the state machine or the instantiation method.

Thanks to the aspect, whenever the TestCaching method is called, the CacheAttribute.OnEntry method executes first. It checks whether the result value is already stored in the cache. If the value already exists, then a new task is created from this value and immediately returned back to the caller. Otherwise, the execution continues normally and a new task instance is created to run our async method.

The CacheAttribute.OnSuccess method is called before the newly created async task is returned back to the caller. The aspect's method creates a new continuation task and returns it back to the caller instead of the original one. Once the

original task completes, the continuation added by the aspect stores the resulting value in cache. Afterwards, the original caller proceeds with processing the task's result.

Applying the aspect to the async task itself

In many cases, we need to inject behaviors into the code of the async task itself. You can accomplish this by setting the `On-MethodBoundaryAspectApplyToStateMachine` property to `true`.

We'll see how to use this feature to create an aspect that measures the execution time of a method. In this section, we will create a simple profiling aspect and show how to apply it correctly to the async methods.

To test the profiling aspect, we will measure the execution time of the `TestProfiling` example method, shown below. The call to `Thread.Sleep` represents the work performed by the method. This is the time we want to measure. The call to `Task.Delay` represents the work performed outside the method, for example the asynchronous call to a web service or a database server.

```
public async Task TestProfiling()
{
    await Task.Delay( 1000 );
    Thread.Sleep( 1000 );
}
```

To inject behaviors into the async task itself:

1. Create an aspect class and inherit `OnMethodBoundaryAspect`. Annotate the class with the [`Serializable-Attribute`] custom attribute.

    ```
    [Serializable]
    public class ProfilingAttribute : OnMethodBoundaryAspect
    {
    }
    ```

    > **NOTE**
    >
    > Use [`PSerializableAttribute`] instead of [`SerializableAttribute`] if your project targets Silverlight, Windows Phone, Windows Store, or runs with partial trust.

2. Implement some of the advice methods of the `OnMethodBoundaryAspect` class, according to your requirements. For more information about these advice methods, see Injecting Behaviors Before and After Method Execution [220].

    For our profiling aspect, we need to implement two advices: `OnEntry(MethodExecutionArgs)` and `On-Exit(MethodExecutionArgs)`.

    ```
    [Serializable]
    public class ProfilingAttribute : OnMethodBoundaryAspect
    {
        public override void OnEntry( MethodExecutionArgs args )
        {
            Stopwatch sw = Stopwatch.StartNew();
            args.MethodExecutionTag = sw;
        }

        public override void OnExit( MethodExecutionArgs args )
        {
            Stopwatch sw = (Stopwatch) args.MethodExecutionTag;
            sw.Stop();
            Console.WriteLine( "Method {0} executed for {1}ms.",
                            args.Method.Name, sw.ElapsedMilliseconds );
        }
    }
    ```

3. Apply your custom attribute to the target methods. Set the `ApplyToStateMachine` property to `true` when applying the attribute to async methods. Below you can see the `[Profiling]` attribute applied to the `TestProfiling` method.

```
[Profiling( ApplyToStateMachine = true )]
public async Task TestProfiling()
{
    await Task.Delay( 1000 );
    Thread.Sleep( 1000 );
}
```

> **NOTE**
>
> You can set the `ApplyToStateMachine` property in the constructor if you don't want to set it explicitly every time the aspect is used.

Whenever the `TestProfiling` method is called, it creates a new async task instance. At some later point, the task starts executing and immediately the `ProfilingAttribute.OnEntry` method is invoked. The method starts the stopwatch and stores it for the future use. Then the execution of the async task continues.

The `ProfilingAttribute.OnExit` method is called just before the async task completes. This method stops measuring the time and outputs the result to the console.

This program produces the following output:

```
Method TestProfiling executed for 2044ms.
```

As you can see, the output shows that we're not only measuring the execution time of the user code in the `TestProfiling` method, but also the time spent waiting for the external tasks to complete. The next section of this article will show how the `ProfilingAttribute` class can be improved to measure only the time spent inside the `TestProfiling` method.

Adding behaviors around the "await" statement

When applying the aspect to the async task itself, you can also inject behaviors before and after the `await` statements in that task. To achieve that, you need to override the `OnYield(MethodExecutionArgs)` and `OnResume(MethodExecution-Args)` methods of the `OnMethodBoundaryAspect` class.

The `OnYield(MethodExecutionArgs)` method is invoked after the async task get paused and yields the control flow to the calling thread, as the result of the `await` statement.

The `OnResume(MethodExecutionArgs)` method is invoked when the async task resumes execution at the point after the `await` statement.

> **NOTE**
>
> Implementing `OnYield(MethodExecutionArgs)` or `OnResume(MethodExecutionArgs)` method also automatically sets the default value of `ApplyToStateMachine` property to `true` and suppresses the warning generated by PostSharp when the `ApplyToStateMachine` property is not set.

In the next steps we will add the `OnYield` and `OnResume` overrides to our `ProfilingAttribute` class. This will allow us to pause the stopwatch when the async method execution is paused, and to resume the stopwatch when the async method execution is resumed.

To inject behaviors around the "await" statement:

1. Create an aspect class and inherit `OnMethodBoundaryAspect`. Annotate the class with the [`Serializable-Attribute`] custom attribute. For this example, we will reuse the `ProfilingAttribute` class from the previous section.

> **NOTE**
>
> Use [`PSerializableAttribute`] instead of [`SerializableAttribute`] if your project targets Silverlight, Windows Phone, Windows Store, or runs with partial trust.

2. Override the `OnYield(MethodExecutionArgs)` method to add functionality at the point where the async task is paused and yields the control flow to the calling thread.

   The following code snippet uses `OnYield(MethodExecutionArgs)` method to pause the stopwatch when the execution of the async method is paused.

   ```
   public override void OnYield( MethodExecutionArgs args )
   {
       Stopwatch sw = (Stopwatch) args.MethodExecutionTag;
       sw.Stop();
   }
   ```

3. Override the `OnResume(MethodExecutionArgs)` method to add functionality at the point where the async task resumes its execution after the `await` statement. This method is invoked when another async task has completed and the control flow has returned to the original task right after the `await` statement.

   The following code snippet uses `OnYield(MethodExecutionArgs)` method to resume the stopwatch when the execution of the async method is resumed.

   ```
   public override void OnResume( MethodExecutionArgs args )
   {
       Stopwatch sw = (Stopwatch) args.MethodExecutionTag;
       sw.Start();
   }
   ```

4. Apply your custom attribute to the target methods. Below you can see the [`Profiling`] attribute applied to the `TestProfiling` method.

   ```
   [Profiling]
   public async Task TestProfiling()
   {
       await Task.Delay( 1000 );
       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.
```

16.1.4. Injecting Behaviors into Iterators

Iterator methods are methods that can return several values. In C#, values are produced using the `yield return` statement. Iterators return a value of type `IEnumerable` or `IEnumerator`. At build time, the compiler performs a complex transformation of the code. The original iterator logic is moved to a different type implementing the `IEnumerator` interface, and the original method body is replaced by just a few lines of code instantiated this class.

By default, all aspects applied on an iterator method are actually applied to the method *instantiating* the iterator class. However, there are times when you want to actually add an aspect to the iterator itself. We will see how this is possible using the `OnMethodBoundaryAspect` aspect. For a more general information about using `OnMethodBoundaryAspect` in non-async methods, see Injecting Behaviors Before and After Method Execution [220].

You have two options when applying `OnMethodBoundaryAspect` to an iterator method. The first option is to apply the aspect to the method that only creates and returns the iterator instance. The second option is to apply the aspect to the actual code that implements the logic inside the iterator. When using the second option you can also inject behaviors before and after your `yield return` statements and access the current yield value.

This topic contains the following sections.

Applying the aspect to the method instantiating the iterator

You can control how the aspect is applied to the iterator by setting the boolean `ApplyToStateMachine` property. Set this property to false on your aspect to apply the aspect only to the code instantiating the iterator. You can set the property in the constructor if you don't want to set it explicitly every time the aspect is used.

In the following procedure, we demonstrate how to inject behaviors into the method instantiating the iterator. For this purpose, we create a simplified caching aspect that stores all the values returned by iterator as a single array in cache. This aspect can be applied to any iterator returning `IEnumerable`.

To inject behaviors into the method instantiating the iterator:

1. Create an aspect class and inherit `OnMethodBoundaryAspect`. Annotate the class with the [`Serializable-Attribute`] custom attribute.

> **NOTE**
>
> Use [`PSerializableAttribute`] instead of [`SerializableAttribute`] if your project targets Silverlight, Windows Phone, Windows Store, or runs with partial trust.

2. Set the `ApplyToStateMachine` property to `false` in your constructor.

```
[Serializable]
public class CacheAttribute : OnMethodBoundaryAspect
{
    public CacheResultAttribute()
    {
        ApplyToStateMachine = false;
    }
}
```

> **NOTE**
>
> The default value of the `ApplyToStateMachine` property is `false` unless the `OnYield(MethodExecutionArgs)` or `OnResume(MethodExecutionArgs)` advice is implemented. However, PostSharp will emit a warning whenever `OnMethodBoundaryAspect` is applied to an iterator without setting the `ApplyToStateMachine` property.

3. Override some of the virtual methods of the `OnMethodBoundaryAspect` class, according to your requirements. For more information about the corresponding virtual methods, see Injecting Behaviors Before and After Method Execution [220].

The following snippet shows the overrides of `OnEntry(MethodExecutionArgs)` and `OnSuccess(MethodExecution-Args)` methods for the `CacheAttribute` class.

```
[Serializable]
public class CacheAttribute : OnMethodBoundaryAspect
{
    public CacheAttribute()
    {
        ApplyToStateMachine = false;
    }

    public override void OnEntry( MethodExecutionArgs args )
    {
        object cachedValue = MemoryCache.Default[args.Method.Name];
        if ( cachedValue != null )
        {
            args.ReturnValue = cachedValue;
            args.FlowBehavior = FlowBehavior.Return;
        }
    }

    public override void OnSuccess( MethodExecutionArgs args )
    {
        object[] elements = ( (IEnumerable) args.ReturnValue ).OfType<object>().ToArray();
        MemoryCache.Default[args.Method.Name] = elements;
        args.ReturnValue = elements;
    }
}
```

4. Apply your custom attribute to the target methods. Below you can see the `[Cache]` attribute applied to the `Test-Caching` method.

```
[Cache]
public IEnumerable TestCaching()
{
    yield return 1;
    yield return 2;
    yield return 3;
}
```

> **NOTE**
>
> You can also set the `ApplyToStateMachine` property value when applying the attribute to a method, for instance using the syntax `[MyAspect( ApplyToStateMachine = false )]`. This can be useful if it makes sense to apply this aspect to both the state machine or the instantiation method.

Thanks to the aspect, whenever the `TestCaching` method is called, the `CacheAttribute.OnEntry` method executes first. It checks whether the result value is already stored in the cache. If the value already exists, then it is immediately returned back to the caller. Otherwise, the execution continues normally and a new iterator instance is created.

The `CacheAttribute.OnSuccess` method is called before the newly created iterator instance is returned back to the caller. The aspect's method enumerates all the iterator values using `ToArray` method and stores the resulting array in cache. The array is also returned to the caller instead of the original iterator instance.

Applying the aspect to the iterator itself

In many cases it can be useful to inject behavior into the user code of the iterator implementation itself. You can accomplish this by setting the `ApplyToStateMachine` property to true on your `OnMethodBoundaryAspect`.

For example, you may want to write a log entry each time the enumeration of the iterator begins and ends instead of logging only the creation of the iterator instance. In this section, we will create a simple logging aspect and show how to apply it correctly to iterators.

To inject behaviors into the iterator itself:

1. Create an aspect class and inherit `OnMethodBoundaryAspect`. Annotate the class with the [`Serializable-Attribute`] custom attribute.

```
[Serializable]
public class LogIteratorAttribute : OnMethodBoundaryAspect
{
}
```

> **NOTE**
>
> Use [`PSerializableAttribute`] instead of [`SerializableAttribute`] if your project targets Silverlight, Windows Phone, Windows Store, or runs with partial trust.

2. Override some of the virtual methods of the `OnMethodBoundaryAspect` class, according to your requirements. For more information about the corresponding virtual methods, see Injecting Behaviors Before and After Method Execution [220].

The following snippet shows the overrides of `OnEntry(MethodExecutionArgs)` and `OnSuccess(MethodExecutionArgs)` methods for the `LogIteratorAttribute` class.

```
[Serializable]
public class LogIteratorAttribute : OnMethodBoundaryAspect
{
    public override void OnEntry(MethodExecutionArgs args)
    {
        Console.WriteLine("The enumeration of {0} started.", args.Method.Name);
    }

    public override void OnSuccess(MethodExecutionArgs args)
    {
        Console.WriteLine("The enumeration of {0} finished.", args.Method.Name);
    }
}
```

3. Apply your custom attribute to the target iterator methods and set the `ApplyToStateMachine` property to true. Below you can see the `[LogIterator]` attribute applied to the `TestLogging` method.

```
[LogIterator( ApplyToStateMachine = true )]
public IEnumerable<int> TestLogging()
{
    yield return 1;
    yield return 2;
    yield return 3;
}
```

> **NOTE**
>
> You can set the `ApplyToStateMachine` property in the constructor if you don't want to set it explicitly every time the aspect is used.

Whenever the `TestLogging` method is called, it creates a new iterator instance. Then the caller starts the enumeration of the iterator and immediately the `LogIteratorAttribute.OnEntry` method is invoked. The method writes the first log entry and the enumeration continues.

The `LogIteratorAttribute.OnSuccess` method is called after the enumeration of the iterator has completed. The method writes the second log entry and the control is given back to the caller.

The snippet below shows the enumeration of our sample iterator and the output written to the console.

```
foreach (int n in TestLogging()) Console.WriteLine(n);

The enumeration of TestLogging started.
1
2
3
The enumeration of TestLogging finished.
```

Adding behaviors around the "yield return" statement

When applying the aspect to the iterator itself, you can also inject behaviors before and after the `yield return` statements in that iterator. To achieve that, you need to override the `OnYield(MethodExecutionArgs)` and `OnResume(MethodExecutionArgs)` methods of the `OnMethodBoundaryAspect` class.

The `OnYield(MethodExecutionArgs)` method is invoked after the iterator yields the control flow, as the result of the `yield return` statement. In this method you can also access and modify the current yield value by using the `YieldValue` property.

The `OnResume(MethodExecutionArgs)` method is invoked when the iterator resumes execution at the point after the `yield return` statement.

NOTE

Note, that implementing `OnYield(MethodExecutionArgs)` or `OnResume(MethodExecutionArgs)` method also automatically sets the default value of `ApplyToStateMachine` property to true and quiets the warning generated by PostSharp.

In the next steps we will create a sample aspect that counts the number of the not-null elements returned by the iterator and writes the result to the console.

To inject behaviors around the "yield return" statement:

1. Create an aspect class and inherit `OnMethodBoundaryAspect`. Annotate the class with the [`Serializable-Attribute`] custom attribute.

   ```
   [Serializable]
   public class NotNullCounterAttribute : OnMethodBoundaryAspect
   {
   }
   ```

   NOTE

   Use [`PSerializableAttribute`] instead of [`SerializableAttribute`] if your project targets Silverlight, Windows Phone, Windows Store, or runs with partial trust.

   ---

2. Override some of the virtual methods of the `OnMethodBoundaryAspect` class, according to your requirements. For more information about the corresponding virtual methods, see Injecting Behaviors Before and After Method Execution [220].

   The following snippet shows the overrides of `OnEntry(MethodExecutionArgs)` and `OnExit(MethodExecution-Args)` methods for the `NotNullCounterAttribute` class.

   ```
   [Serializable]
   public class NotNullCounterAttribute : OnMethodBoundaryAspect
   {
       public override void OnEntry(MethodExecutionArgs args)
       {
           // Initialize and store the couner for use in other method.
           args.MethodExecutionTag = 0;
       }

       public override void OnExit(MethodExecutionArgs args)
       {
           Console.WriteLine("{0} returned {1} not-null values.", args.Method.Name, args.MethodExecutionTag);
       }
   }
   ```

3. Override the `OnYield(MethodExecutionArgs)` method to add functionality at the point where the iterator yields the next value. This method is invoked when the execution of the iterator is paused and the caller takes over the control flow.

The following code snippet shows the `OnYield(MethodExecutionArgs)` method override for the `NotNull-CounterAttribute` class.

```csharp
public override void OnYield(MethodExecutionArgs args)
{
    if (args.YieldValue != null)
    {
        args.MethodExecutionTag = ((int) args.MethodExecutionTag) + 1;
    }
}
```

4. Apply your custom attribute to the target methods. Below you can see the `[NotNullCounter]` attribute applied to the `TestLogging` method.

```csharp
[NotNullCounter]
public IEnumerable<string> TestLogging()
{
    yield return "One";
    yield return null;
    yield return "Two";
}
```

Thanks to the aspect, we will initialize the counter when the enumeration of the `TestLogging` starts, increase the counter each time a not-null value is returned, and finally write the result when the enumeration completes.

The snippet below shows the enumeration of our sample iterator and the output written to the console.

```csharp
foreach (string s in TestLogging()) Console.WriteLine(s ?? "<null>");
```

```
One
<null>
Two
TestLogging returned 2 not-null values.
```

16.1.5. 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 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 [220].

This topic contains the following sections.

Intercepting a method call

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.

```
public class CustomerService
{
    public void 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.

To create an aspect that retries a method call on exception:

1. Create an aspect class and inherit `MethodInterceptionAspect`. Annotate the class with the [`Serializable-Attribute`] custom attribute.

   ```
   public class RetryOnExceptionAttribute : MethodInterceptionAspect
   {
   }
   ```

   > **NOTE**
   >
   > Use [`PSerializableAttribute`] instead of [`SerializableAttribute`] if your project targets Silverlight, Windows Phone, Windows Store, or runs with partial trust.

2. Define a property `MaxRetries` and set its default value to 3 in the constructor.

   ```
   public int MaxRetries { get; set; }

   public RetryOnExceptionAttribute()
   {
       this.MaxRetries = 3;
   }
   ```

3. Override the `OnInvoke(MethodInterceptionArgs)` method.

   ```
   public override void OnInvoke(MethodInterceptionArgs args)
   {
       base.OnInvoke(args);
   }
   ```

4. Edit the `OnInvoke` method to catch the exception and retry the operation. Use `base.OnInvoke()` to invoke the intercepted method.

   The complete aspect code is as follows:

```csharp
[Serializable]
public class RetryOnExceptionAttribute : MethodInterceptionAspect
{
    public int MaxRetries { get; set; }

    public override void OnInvoke(MethodInterceptionArgs args)
    {
        int retriesCounter = 0;

        while ( true )
        {
            try
            {
                base.OnInvoke(args);
                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);
            }
        }
    }
}
```

> **NOTE**
>
> Calling `base.OnInvoke()` is equivalent to calling `args.Proceed()`.

5. 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:

```csharp
public class CustomerService
{
    [RetryOnException(MaxRetries = 5)]
    public void Save(Customer customer)
    {
        // Database or web-service call.
    }
}
```

Whenever the `CustomerService.Save` method will be invoked, the `RetryOnExceptionAttribute.OnInvoke` method will be called instead. It will invoke the original method and retry if necessary.

Accessing the identity of the intercepted method

As illustrated in the example above, you can access information about the method being intercepted from the property `MethodInterceptionArgsMethod`, 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-InterceptionArgsMethod` gives you the proper generic method instance, so you can use this object to get generic parameters.

Accessing the arguments and the return value

An implementation of `MethodInterceptionAspect` can read and modify the arguments of the intercepted method thanks to the `MethodInterceptionArgsArguments` property, and the return value thanks to the `MethodInterceptionArgsReturn-Value` property.

For example, consider a `UserService` class with a method that returns user permissions. The method would normally return `null` if no permissions have been assigned to this user.

```csharp
public class UserService
{
    public Permissions GetPermissions(string username)
    {
        // Database or web-service call.
    }
}
```

If the given user has no assigned permission, instead of returning `null`, we would like the method to return the permissions assigned to the user named DefaultUser.

We can implement these requirements using an aspect derived from `MethodInterceptionAspect`. The aspect first invokes the original method, then it checks whether it has returned a `null` value. If the value is `null`, the method is invoked again but with the arguments replaced by the default values.

To create an aspect that uses default arguments when a method returns null:

1. Create an aspect class and inherit `MethodInterceptionAspect`. Annotate the class with the [`Serializable-Attribute`] custom attribute.

    ```csharp
    [Serializable]
    public class RetryIfNullAttribute : MethodInterceptionAspect
    {
    }
    ```

    > **NOTE**
    >
    > Use [`PSerializableAttribute`] instead of [`SerializableAttribute`] if your project targets Silverlight, Windows Phone, Windows Store, or runs with partial trust.

2. Add a field `defaultArguments` to the class and set this field from the constructor.

    ```csharp
    private object[] defaultArguments;

    public RetryIfNullAttribute(params object[] defaultArguments)
    {
        this.defaultArguments = defaultArguments;
    }
    ```

3. Override the `OnInvoke(MethodInterceptionArgs)` method.

    ```csharp
    public override void OnInvoke(MethodInterceptionArgs args)
    {
        base.OnInvoke(args);
    }
    ```

4. Edit the `OnInvoke` method to implement the aspect logic. Check whether the `MethodInterceptionArgsReturnValue` property is `null` return value, and set the items of the `MethodInterceptionArgsArguments` property. Finally, invoke the intercepted method a second time by calling `base.OnInvoke()`.

The complete aspect code is as follows:

```
[Serializable]
public class RetryIfNullAttribute : MethodInterceptionAspect
{
    private object[] defaultArguments;

    public RetryIfNullAttribute(params object[] defaultArguments)
    {
        this.defaultArguments = defaultArguments;
    }


    public override void OnInvoke(MethodInterceptionArgs args)
    {
        base.OnInvoke(args);

        if (args.ReturnValue == null)
        {
            Console.WriteLine(
                "The method {0}.{1} has returned a null value." +
                " Retrying with the default arguments...",
                args.Method.DeclaringType, args.Method.Name);

            for (int i = 0; i < args.Arguments.Count && i < this.defaultArguments.Length; i++)
            {
                args.Arguments[i] = this.defaultArguments[i];
            }

            base.OnInvoke(args);
        }
    }
}
```

5. Apply this method interception aspect as an attribute to all methods where the implemented logic is needed. In the following snippet, the aspect is applied to the `UserService.GetPermissions` method:

```
public class UserService
{
    [RetryIfNull("DefaultUser")]
    public Permissions GetPermissions(string username)
    {
        // database or web-service call
    }
}
```

Thanks to the aspect, whenever the `UserService.GetPermissions` method is invoked, the `RetryIfNullAttribute.OnInvoke` method will be called instead. The `OnInvoke` method will then invoke the original `GetPermissions` method. If it returns `null` for the provided `username`, then the method will be invoked again, but this time with the value DefaultValue for the `username` parameter.

Accessing the target objects

In combination with the parameters you will probably interact with the target code instance that the aspect is attached to. The `Instance` property provides you with the instance of the object that the aspect is currently operating against. It is an object type so you will need to cast it to the correct type to be able to interact with it. If you debug your aspect and that aspect doesn't make use of `Instance`, it will be set to null. It's also set to null if the target code is defined as static.

16.1.6. 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

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. Create an aspect that inherits from `LocationInterceptionAspect` and add the custom attribute [`Serializable-Attribute`].

   > **NOTE**
   >
   > Use [`PSerializableAttribute`] instead of [`SerializableAttribute`] if your project targets Silverlight, Windows Phone, Windows Store, or runs with partial trust.

2. Override the `OnGetValue(LocationInterceptionArgs)` method.

   ```
   [Serializable]
   public class StringCheckerAttribute : LocationInterceptionAspect
   {
       public override void OnGetValue(LocationInterceptionArgs args)
       {
           base.OnGetValue(args);
       }

   }
   ```

3. 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.

   ```
    public override void OnGetValue(LocationInterceptionArgs args)
   {
       base.OnGetValue(args);

       if (args.Value == null)
       {
           args.Value = "foo";
       }
   }
   ```

4. 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.

```
public class Customer
{
    [StringChecker]
    private readonly string _address;

    public Customer(string address)
    {
        _address = address;
    }
    [StringChecker]
    public string Name { get; set; }
    public string 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 [147] that explain how to add aspects en mass.

5. 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();
    }
}
```
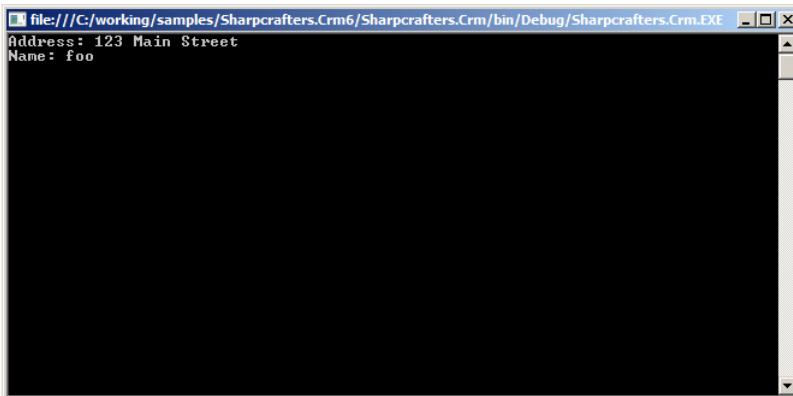

Property and field interception is a simple and seamless task. Once you have intercepted your target you can act on the target or you can allow the original code to execute.

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]
public string 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:

```
[Serializable]
public class StringCheckerAttribute : LocationInterceptionAspect
{
    .
    .
    .
public override void OnSetValue(LocationInterceptionArgs args)
{
if (args.Value == null)
{
args.Value = "Empty String";
}


args.ProceedSetValue();

}
}
```

Getting and setting the underlying property

PostSharp provides a mechanism to check a property's underlying value via the `LocationInterceptionArgsGetCurrent-Value` 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:

```
 public override void 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(LocationInterception-Args)`. 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 `Parameter-Info` (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
{
  public string 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]
public class NotifyPropertyChangedAttribute : LocationInterceptionAspect
{

public override void 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 [77].

This aspect can then be applied to the `Customer` class:

```
[NotifyPropertyChangedAttribute]
class Customer : INotifyPropertyChanged
{
    public string 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.

16.1.7. 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 everytime 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

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. Create an aspect that inherits from `EventInterceptionAspect`. Add the [`PSerializableAttribute`] custom attribute.

> **NOTE**
>
> Use [`PSerializableAttribute`] instead of [`SerializableAttribute`] if your project targets Silverlight, Windows Phone, Windows Store, or runs with partial trust.

2. Override the `OnAddHandler(EventInterceptionArgs)` method and add your logging code to the method body.

3. 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.

```
public class CustomEventing : EventInterceptionAspect
{
    public override void OnAddHandler(EventInterceptionArgs args)
    {
        base.OnAddHandler(args);
        Console.WriteLine("A handler was added");
    }
}
```

4. To log the removal of an event handler, override the `OnRemoveHandler(EventInterceptionArgs)` method.

5. Add the logging you require to the method body.

6. 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.

```
public override void 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.

```
 public class Example
{
    public EventHandler<EventArgs> SomeEvent;

    public void DoSomething()
    {
        if (SomeEvent != null)
        {
            SomeEvent.Invoke(this, EventArgs.Empty);
        }
    }
}
```

Intercepting Raise

When you are intercepting events you will also have situations where you will want to intercept the code execution when the event is raised. Raising an event can occur may places and you will want to centralize this code to save repetition.

1. Override the `OnInvokeHandler(EventInterceptionArgs)` method on 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.

```
 public override void OnInvokeHandler(EventInterceptionArgs args)
{
    base.OnInvokeHandler(args);
    Console.WriteLine("A handler was invoked");
}
```

By adding the attribute to the target code's 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` is powerful and can help you to accomplish this with ease.

1. Override the `OnInvokeHandler(EventInterceptionArgs)` method on 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 `Remove-Handler(Delegate)`

```csharp
 public class CustomEventing : EventInterceptionAspect
{
    public override void 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 when the event is executed, the offending event handler will be unsubscribed from the event.

16.1.8. Introducing Interfaces

When you create a `CompositionAspect` you are able to dynamically add interfaces to the target code at compile time and make use of that interface type at run time.

1. The first thing that you need to do is create an aspect that inherits from `CompositionAspect` and implements its members.

```csharp
[Serializable]
public class GeneralCompose : CompositionAspect
{
    public override object CreateImplementationObject(AdviceArgs args)
    {
        throw new System.NotImplementedException();
    }
}
```

2. Next, you need some way to tell the aspect what interface and concrete type you want to implement on the target code. To do that, create a constructor for your aspect that accepts two parameters; one for the interface type and one for the concrete implementation type. Assign those two constructor parameters to field level variables so we can make use of them in the aspect.

```csharp
[Serializable]
public class GeneralCompose : CompositionAspect
{
    private readonly Type _interfaceType;
    private readonly Type _implementationType;

    public GeneralCompose(Type interfaceType, Type implementationType)
    {
        _interfaceType = interfaceType;
        _implementationType = implementationType;
    }
```

Developing Simple Aspects

3. There are two methods that you need to implement to complete this aspect. The first is an override of the `GetPublicInterfaces(Type)` method. This method has a target type parameter which allows you to filter the application of the interface if you choose to. For this example, simply return an array that contains the interface type that was provided via the aspect's constructor.

```
protected override Type[] GetPublicInterfaces(Type targetType)
{
return new[] { _interfaceType };
}
```

> **NOTE**
>
> The interfaces that are returned from the `GetPublicInterfaces(Type)` method will be applied to the target code during compilation.

4. The second method that you need to override is `CreateImplementationObject(AdviceArgs)`. For this example you will return an instance of the concrete implementation that was provided in the aspect's constructor. The `CreateImplementationObject(AdviceArgs)` method doesn't return the type of the concrete implementation. It returns an instance of that type instead. To create the instance use the `CreateInstance(Type, ActivatorSecurityToken)` method.

```
public override object CreateImplementationObject(AdviceArgs args)
{
return Activator.CreateInstance(_implementationType);
}
```

> **NOTE**
>
> The `CreateImplementationObject(AdviceArgs)` method is invoked at the application's runtime.

5. Now that you have created a complete `CompositionAspect`, it will need to be applied to the target code. Add the aspect to the target code as an attribute. Provide the attribute with the interface and concrete types that you wish to implement.

```
[GeneralCompose(typeof(IList), typeof(ArrayList))]
public class Fruit
{
}
```

6. After compiling your application you will find that the target code now implements the assigned interfaces and exposes itself as a new instance of the concrete type you declared. The next question that needs addressing is how you will interact with the target code using that interface type.

To access the dynamically applied interface you must make use of a special PostSharp feature. The `CastSource-Type, TargetType(SourceType)` method will allow you to safely cast the target code to the interface type that you dynamically applied. Once that call has been done, you are able to make use of the instance through the interface constructs.

```
[GeneralCompose(typeof(IList), typeof(ArrayList))]
public class Fruit
{
    public Fruit()
    {
        IList list = Post.Cast<Fruit,IList>(this);
        list.Add("apple");
        list.Add("orange");
        list.Add("banana");
    }
}
```

16.1.9. 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
- Copying existing custom attributes

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`. `TypeLevel-Aspect` transforms the class into an attribute which can be applied to other classes. Also implement `IAspect-Provider` which exposes the `ProvideAspects(Object)` method for iterating on class members. `Provide-Aspects(Object)` will be called for each member in the target class and will contain the code for applying the attributes:

```
public sealed class 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 `CustomAttributeIntroduction-Aspect` objects:

```csharp
public sealed class AutoDataContractAttribute : TypeLevelAspect, IAspectProvider
{
    // This method is called at build time and should just provide other aspects.
    public IEnumerable<AspectInstance> ProvideAspects(object targetElement)
    {
        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.
        yield return new AspectInstance(targetType, introduceDataContractAspect);

        // Add a DataMember attribute to every relevant property.
        foreach (PropertyInfo property in
            targetType.GetProperties(BindingFlags.Public | BindingFlags.DeclaredOnly | BindingFlags.Instance))
        {
            if (property.CanWrite)
                yield return new 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 `DataContractAttribute` class. In the following example we apply it to a `Product` class where it will decorate `Product` with `DataContractAttribute` and each member with `DataMemberAttribute`:

```csharp
[DataContractAttribute]
public class Product
{
    public int ID { get; set; }

    public string Name { get; set; }

    public int RevisionNumber { get; set; }
}
```

See Example: Automatically Adding DataContract and DataMember Attributes [294] to learn how to have the `DataContract-Attribute` automatically applied to derived classes.

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 `Copy-CustomAttributesFrom`:

   ```
   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.Equals("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.

4. 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 `CopyCustomAttributes-Attribute` at build time:
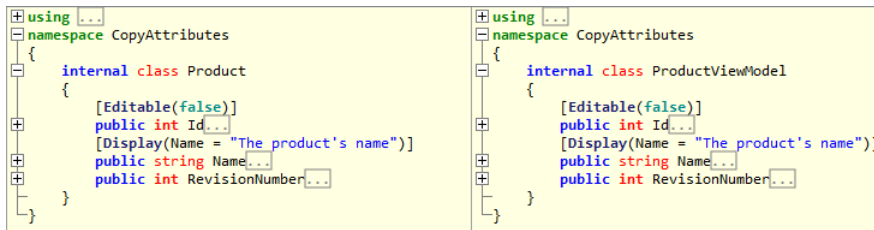
> **NOTE**
>
> It is not possible to delete or replace an existing custom attribute.

16.1.10. 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 runtime.

1. Start by creating a class called `AddBuildInfoAspect` which derives from `AssemblyLevelAspect`. Also implement `IAspectProvider` 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:

   ```csharp
   public sealed class AddBuildInfoAspect : AssemblyLevelAspect, IAspectProvider
   {
       public IEnumerable<AspectInstance> ProvideAspects(object targetElement)
       {
       }
   }
   ```

2. Implement the `ProvideAspects(Object)` method:

   ```csharp
   public sealed class 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);

           yield return new 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:

   ```csharp
   [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 runtime:

```
class Program
{
    static void Main(string[] args)
    {
        Assembly a = Assembly.GetExecutingAssembly();
        Stream stream = a.GetManifestResourceStream("BuildUser");

        byte[] bytesRead = new byte[stream.Length];
        stream.Read(bytesRead, 0, (int)stream.Length);
        string value = Encoding.ASCII.GetString(bytesRead);
        Console.WriteLine(value);
    }
}
```

This will display the following line in the console window:
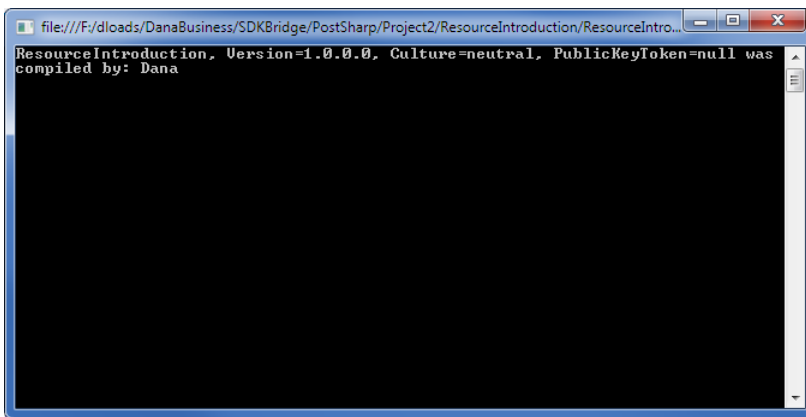
16.2. Understanding Aspect Lifetime and Scope

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 `Serializable-Attribute` 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).

NOTE

If your project targets Silverlight, Windows Phone, or the Compact Framework, aspects are initialized at run-time and all compile-time steps are skipped.

This topic contains the following sections.

- Scope of Aspects
- Steps in the Lifetime of an Aspect Instance
- Examples

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:

```csharp
public class GenericClass<T>
{
  static T f;

  [Trace]
  public void void SetField(T value) { f = value; }
}

public class Program
{
    public static void 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 `Generic-Class<long>`) but only a single instance of the aspect `Trace`.

Instance-Scoped Aspects

Instance-scoped aspect 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`:

```csharp
object IInstanceScopedAspect.CreateInstance( AdviceArgs adviceArgs )
{
  return this.MemberwiseClone();
}

object IInstanceScopedAspect.RuntimeInitializeInstance()
{
}
```

Steps in the Lifetime of an Aspect Instance

The following table summarizes the different steps of

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 (`MulticastAttribute`), 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 `[Trace(Category="FileManager")]`, PostSharp calls the default constructor and the `Category` property setter.
	Validation	PostSharp validates the aspect by calling the `CompileTimeValidate` aspect method. See Validating Aspect Usage [262] for details.
	Compile-Time Initialization	PostSharp invokes the `CompileTimeInitialize` aspect method. This method may, but must not, 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.
	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.
Run-Time	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 `RuntimeInitialize` method. But this time we pass as an argument the real `System.Reflection` 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 `CreateInstance(AdviceArgs)` of the prototype aspect instance. After the new aspect instance has been set up, the aspect framework invokes the `RuntimeInitializeInstance`.
	Advice Execution	Finally, advices (methods such as `OnEntry(MethodExecutionArgs)`) are executed.

Examples

Example: Raising an Event When the Object is Finalized [295]

16.3. Initializing Aspects

As explained in the section Understanding Aspect Lifetime and Scope [259], 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 runtime. 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 `IInstanceScopedAspectRuntime-InitializeInstance` 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.

16.4. Validating Aspect Usage

Some aspects make sense only on a specific subset of targets. For instance, an aspect may require to be applied on 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 runtime. 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]

The first level of protection is to configure multicasting properly with [`MulticastAttributeUsageAttribute`], as described in the article Adding Aspects Declaratively Using Attributes [146]. 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 to be applied only to methods of types that implement the `ISecurable` interface.

1. Inherit from one of the pre-built aspects. In this case `OnMethodBoundaryAspect`.

```
public class RequirePermissionAttribute: OnMethodBoundaryAspect
```

2. Override the `CompileTimeValidate(Object)` method.

```
 public override bool 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 `InvalidAnnotation-Exception`.

```
if (!typeof(ISecurable).IsAssignableFrom(targetType))
{
  throw new 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);
 return false;
}
```

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 `Attribute-TargetMemberAttributes`, its real power is to go beyond those filtering techniques. By using `CompileTime-Validate(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:

```csharp
internal class ArchitectureMessageSource : MessageSource
{
    public static readonly ArchitectureMessageSource Instance = new ArchitectureMessageSource();

    private ArchitectureMessageSource() : base( "PostSharp.Architecture", new Dispenser() )
    {
    }

    private class Dispenser : MessageDispenser
    {
        public Dispenser() : base( "CUS" )
        {
        }

        protected override string GetMessage( int number)
        {
            switch ( number )
            {
                case 1:
                    return "Interface {0} cannot be implemented by {1} because of the [InternalImplement] constraint

                case 2:
                    return "{0} {1} cannot be referenced from {2} {3} because of the [ComponentInternal] constraint.

                case 3:
                    return "Cannot use [ComponentInternal] on {0} {1} because the {0} is not internal.";

                case 4:
                    return "Cannot use [Internal] on {0} {1} because the {0} is not public.";

                default:
                    return null;
            }
        }
    }
}
```

3. Then you can use a convenient set of methods on your `MessageSource` object:

```csharp
MyMessageSource.Instance.Write( classType, SeverityType.Error, "CUS001", new object[] { 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`.

Examples

16.5. Developing Composite Aspects

PostSharp offers two approaches to aspect-oriented development. The first, as explained in section Developing Simple Aspects [217], 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 `IAspectProvider` (see Adding Aspects Dynamically [276]). 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 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 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 [265]	Advices with equivalent functionality as `OnMethodBoundaryAspect`, `MethodInterceptionAspect`, `LocationInterceptionAspect`, and `EventInterceptionAspect`.
Introducing Interfaces, Methods, Properties and Events [269]	Make the aspect introduce an interface into the target class. The interface is implemented by the aspect itself.
Accessing Members of the Target Class [274]	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.

16.5.1. 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 on 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
-
-
-

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: `[OnMethodEntry-Advice]`.

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;

namespace Samples6
{
    [Serializable]
    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 `OnMethodBoundaryAspectOnEntry(Method-ExecutionArgs)`. For a complete documentation of the advice, see the documentation of the corresponding simple aspect.

Advice Type	Targets	Description
OnMethodEntryAdvice OnMethodSuccessAdvice OnMethodExceptionAdvice OnMethodExitAdvice	Methods	These advices are equivalent to the advices of the aspect OnMethodBoundaryAspect. The target method to be wrapped by a try/catch/finally construct.
OnMethodInvokeAdvice	Methods	This advice is equivalent to the aspect MethodInterceptionAspect. Calls to the target methods are replaced to calls to the advice.
OnLocationGetValueAdvice OnLocationSetValueAdvice	Fields, Properties	These advices are equivalent to the advices of the aspect LocationInterceptionAspect. Fields are changed into properties, and calls to the accessors are replaced to calls to the proper advice.
LocationValidationAdvice	Fields, Properties, Parameters	This advice is equivalent to the ValidateValue(T, String, LocationKind)LocationInterceptionAspect method of the ILocationValidationAspectT aspect interface. It validates values assigned to their targets and throws an exception in case of error.
OnEventAddHandlerAdvice OnEventRemoveHandlerAdvice OnEventInvokeHandlerAdvice	Events	These advices are equivalent to the advices of the aspect EventInterceptionAspect. Calls to add and remove semantics are replaced by calls to advices. When the event is fired, the OnEventInvokeHandler 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 different classes or members of different classes.

Multicast Pointcut

The pointcut type MulticastPointcut allows to express 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 [146]) 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)]
public void OnPropertySet( LocationInterceptionArgs args )
{
  // Details skipped.
}
```

Method Pointcut

The pointcut type MethodPointcut allows to express a pointcut imperatively, using a C# or Visual Basic 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**) 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.

```csharp
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`, `OnMethodSuccess-Advice` and `OnMethodExceptionAdvice`. The way how they are ordered is important, as it results in different generation of `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.

```csharp
 void Method()
{
  try
  {
    OnEntry();

    try
    {
      // Original method body.
    }
    finally
    {
      OnExit();
    }
  }
  catch
  {
    OnException();
    throw;
  }
}
```

```csharp
 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 `On-MethodBoundaryAspect`. 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 [279]). 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 Understanding Interception Aspects [280]).

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 [266] and one pointcut custom attribute (see Available Pointcuts [267]), as usually.

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)
{
}
```

16.5.2. 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 [270]
- Introducing methods [271]
- Introducing properties [272]
- Controlling the visibility of introduced members [274]
- Overriding members or interfaces [274]

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`, **IEquatable** 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.

   ```
   public interface IIdentifiable
   {
       Guid Id { get; }
   }
   ```

2. Create an aspect that inherits from `InstanceLevelAspect` and add the custom attribute [`Serializable-Attribute`].

   > **NOTE**
   >
   > Use [`PSerializableAttribute`] instead of [`SerializableAttribute`] if your project targets Silverlight, Windows Phone, Windows Store, or runs with partial trust.

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.

   ```
   [Serializable]
   public class 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))]
   [Serializable]
   public class 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 `CastSourceType, TargetType(SourceType)` pseudo-operator. The `CastSourceType, TargetType(SourceType)` 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 construct 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 [261].

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 [`Serializable-Attribute`].

2. Add to the aspect the method you want to introduce to the target code.

```
[Serializable]
public class OurCustomAspect : InstanceLevelAspect
{
    public void 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]
public void 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]
public class Customer
{
    public string Name { get; set; }
}
```

5. After compilation you can decompile the target code and see that the method has been added.

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 [`Serializable-Attribute`].

2. Add the property you want to introduce to the aspect.

```
[Serializable]
public class OurCustomAspect : InstanceLevelAspect
{
    public string 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]
public string Name { get; set; }
```

4. Add the aspect attribute to the target code where the aspect should be applied.

```
[OurCustomAspect]
public class 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. Post-Sharp 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)]
public string 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 `Is-Virtual` property to `true`.

```
[IntroduceMember(Visibility = Visibility.Private, IsVirtual = true)]
public string 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 explicity 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)]
public string Name { get; set; }
```

16.5.3. 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
- Examples

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 `Delegate`, typically one of the variants of `Action` or `FuncTResult`. The delegate signature should exactly match the signature of the imported method.
Property	`PropertyTValue`, where the generic argument is the type of the property.
Collection Indexer	`PropertyTValue, TIndex`, 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	`EventTDelegate`, where the generic argument is the type of the event delegate (for instance `EventHandler`).

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 the 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 primordial 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 [277]) 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 `BeforeIntroductions`.

   b. Member introductions.

   c. Members imports which have the property `ImportMemberAttributeOrder` set to `AfterIntroductions` (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 `ImportMemberAttributeIsRequired` is `true`, ignored otherwise (by default).
	Non-virtual member defined.	Member imported.
	Virtual member defined.	If `ImportMemberAttributeOrder` is `BeforeIntroductions`, 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 `IntroduceMemberAttributeOverrideAction` is `Ignore` or `OverrideOrIgnore`, otherwise fail (by default).
	Virtual member defined in a parent class	Introduce a new `override` method if the property `IntroduceMemberAttributeOverrideAction` is `OverrideOrFail` or `OverrideOrIgnore`, ignore if the property is `Ignore`, otherwise fail (by default).
	Member defined in the target class (virtual or not)	Fail by default or if the property `IntroduceMemberAttributeOverrideAction` is `Fail`. Otherwise: 1. Move the previous method body to a new method so that the previous implementation can be imported by advices `ImportMemberAttribute` with the property `Order` set to `BeforeIntroductions`. 2. Override the method with the imported method.

Examples

16.5.4. Adding Aspects Dynamically

Additionally to providing advices, an aspect can provide other aspects dynamically using `IAspectProvider`. 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 `IAspectProvider`, see Adding Aspects Programmatically using IAspectProvider [163].

16.6. 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
`AspectTypeDependencyAttribute`	This custom attribute expresses a dependency with a well-known aspect class.
`AspectRoleDependencyAttribute`	This custom attribute expresses a dependency with any aspect classes enrolled in a given role. Its dual is `ProvideAspectRoleAttribute`: 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 `StandardRoles`.
`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 `StandardEffects`. 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 `WaiveAspectEffectAttribute`. 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).

Every of 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 `AspectDependencyPosition` (with values `Before` or `After`), 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 `AspectDependencyPosition`, 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 `AspectDependencyPosition`, 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")]
public sealed class AsyncAttribute : MethodInterceptionAspect
{
  // Details skipped
}

[ProvideAspectRole( StandardRoles.ExceptionHandling )]
[AspectRoleDependency( AspectDependencyAction.Order, AspectDependencyPosition.After, StandardRoles.Threading )]
[AspectRoleDependency(AspectDependencyAction.Order, AspectDependencyPosition.After, "UI")]
public sealed class ExceptionDialogAttribute : OnExceptionAspect
{
  // Details skipped
}

[ProvideAspectRole("UI")]
public sealed class 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 skipping the execution of the method, such as a caching aspect. Then, it shows how the aspect `Async-Attribute` 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 )]
public sealed class AuthorizationAttribute : OnMethodBoundaryAspect
{
  // Details skipped.
}

 [WaiveAspectEffect(StandardEffects.ChangeControlFlow)]
 public sealed class AsyncAttribute : 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 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 Configuring Aspects [284]).

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) ]
```

16.6.1. Ordering Advices

The section Coping with Several Aspects on the Same Target [277] 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 [265]), 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
`Aspect1.MethodA, Aspect1.MethodB, Aspect2.MethodC`	`Aspect1.MethodA, Aspect2.MethodC, Aspect1.MethodB`
`Aspect1.MethodB, Aspect1.MethodA, Aspect2.MethodC`	`Aspect1.MethodB, Aspect2.MethodC, Aspect1.MethodA`
`Aspect2.MethodC, Aspect1.MethodA, Aspect1.MethodB`	
`Aspect2.MethodC, Aspect1.MethodB, Aspect1.MethodA`	

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.

16.7. Understanding Interception Aspects

Aspect types `MethodInterceptionAspect`, `LocationInterceptionAspect` and `EventInterceptionAspect` are all based on the same principle: the aspect is invoked *instead of* the enhanced semantic. The aspect gets access to the intercepted semantic through methods prefixed by `Proceed`, or by other methods.

Things become more complex when several interception aspects are applied to the same element of code. Consider a method enhanced by three aspects `A`, `B` and `C`. When aspect `A` calls the method `Proceed`, it will actually invoke the method `OnInvoke(MethodInterceptionArgs)` of aspect `B`. Similarly, aspect `B` will invoke aspect `C`, and aspect `C` will eventually invoke the original method.

Chains of Invocation

Interception aspects form a *chain on invocation* where every aspect instance is a node in the chain, and the intercepted member is the last node.

An interception aspect can only invoke the next node in the chain. There is no way an aspect can invoke another node, or can access directly the intercepted member. This design ensures that aspects behave in a robust and consistent way in all situations.

Aspect types `LocationInterceptionAspect` and `EventInterceptionAspect` have several semantics (`Get` and `Set` for `LocationInterceptionAspect`; `Add`, `Remove` and `Invoke` for `EventInterceptionAspect`). All advices of the same aspect instance (one advice per semantic) logically belong to the same node in the chain of invocation. Therefore, when the implementation of the advice `LocationInterceptionAspectOnSetValue(LocationInterceptionArgs)` invokes the method `LocationInterceptionArgsGetCurrentValue`, it actually invokes the `Get` semantic of the next node in the chain. If the aspect had used `PropertyInfoGetValue(Object)` to get the value (as was usual in PostSharp 1.0), it would have invoked the *first* node in the chain!

Aspects as Filters: a Disciplined Approach to Aspect-Oriented Programming

In its early days, aspect-oriented programming (AOP) has been perceived as a dangerous technology. Aspects allowed to do anything with a program. Although AOP has been designed to improve the readability and maintainability of source code, it could actually have the opposite effect.

As goes the saying, with a sharp tool, one must pay greater attention.

PostSharp was designed to respect one of the most fundamental principles of software engineering: encapsulation. *Encapsulation* means the condition of being enclosed, as in a capsule. In object-oriented programming, the primary capsule is the class itself. Outside code communicates with the capsule through well-defined ports: public members. Outside code cannot modify what's inside the capsule. A well-designed capsule should check the validity of messages it receives or it sends - something called precondition and postcondition checking. The second level of encapsulation is the method: even inside a class, code should be designed so that the implementation of a method does not need to care about the implementation of another method.

Of course, it is possible to ignore the rules of encapsulation. But it would most probably result in poorly readable and maintainable code.

PostSharp actually allows you to break the first capsule: you can add advices to private members of a class. But it stops there: you cannot break the capsule of a method. Instead, you can enclose a method into a new capsule analog to a *filter*: the advice. When a method is enhanced by an advice, outside code seeking access to this method must go through its advice.

When a method is enhanced by several advices, every advice constitutes a filter that encloses not only the method, but all advices with lower priority.

Methods have a single semantic: `Invoke`. Properties, fields and events have many multiple semantics. These members can be considered as a single capsule, and their semantics as different ports in the capsule.

NOTE

Things can become more complex. Consider a property with a getter and a setter. The property is enhanced by an aspect of type `LocationInterceptionAspect`. The property setter is enhanced by a `MethodInterceptionAspect` with lower priority. From a logical point of view, the property is considered as a single capsule with two ports. The capsule is enclosed by two filters, one for each aspect. The aspect `LocationInterceptionAspect` filters both ports. However, `Method-InterceptionAspect` only filters the `Set` port. If the `LocationInterceptionAspect` invokes the `Get` semantic, it will be directed to the property getter, because there is no filter between the advice and the semantic. However, when the same aspect invokes the `Set` semantic, it will be directed to `LocationInterceptionAspect` as this filters lays in the way.

NOTE

The `Invoke` semantic of `EventInterceptionAspect` is executed in invert order. Indeed, the message originates inside the capsule is emitted outside. For all other semantics, the message always comes from outside and is directed to the capsule.

Aspect Bindings

When an advice is invoked, it receives an interface to the next node in the chain of invocation: an aspect binding. Every aspect type has its corresponding binding interface, exposed on a property of the advice argument object.

Aspect Type	Binding Interface	Exposed On
`MethodInterceptionAspect`	`IMethodBinding`	`MethodInterceptionArgsBinding`
`LocationInterceptionAspect`	`ILocationBinding`	`LocationInterceptionArgsBinding`
`EventInterceptionAspect`	`IEventBinding`	`EventInterceptionArgsBinding`

Binding objects are singletons. They are fully thread-safe and reentrant. They can be invoked in any situation. This contrasts with advice arguments, which may be shared among different advices and should not be used once the advice gave over control to the next node in the chain invocation.

NOTE

As objects of type `Arguments` may be shared among different advices, some of which may modify the arguments, it may be safe to clone the object before the advice gives over control.

NOTE

For run-time performance reasons, PostSharp does not access binding classes through their interface, but directly invokes their implementation. Implementation classes of binding interfaces are considered an implementation detail and should not be referred to from user code.

16.8. Understanding Aspect Serialization

As explained in section Understanding Aspect Lifetime and Scope [259], aspect 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 runtime, 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

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 these custom attributes. The serialization strategy is implemented in classes derived from the abstract class `AspectSerializer` according to the following table.

Target platform	Making the class serializable	Excluding fields	Aspect serializer
.NET Framework with full trust	`SerializableAttribute`	`NonSerializedAttribute`	`BinaryAspectSerializer`, backed by `BinaryFormatter`
Any platform	`PSerializableAttribute`	`PNonSerializedAttribute`	`PortableAspectSerializer`, backed by `PortableFormatter`

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 runtime 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 runtime, 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 Configuring Aspects [284] for details.

The following code shows how to choose the serializer type for an `OnMethodBoundaryAspect`:

```
[OnMethodBoundaryAspectConfiguration(SerializerType=typeof(MsilAspectSerializer))]
public sealed MyAspect : OnMethodBoundaryAspect
```

16.9. Advanced

16.9.1. 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.GetUnitializedObject(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 the method **FormatterServices.GetUnitializedObject(Type)**, you will have to initialize aspects manually.

Initializing Aspects Manually

There are many 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:

```
protected virtual void 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 on 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]
public abstract class BaseClass
{
  protected virtual void InitializeAspects()
  {
  }

  [OnDeserializing]
  private void OnDeserializingInitializeAspects()
  {
    this.InitializeAspects();
  }
}
```

By Invoking AspectUtilities.InitializeCurrentAspects

Instead of providing an empty method `InitializeAspects`, it is possible to invoke the method `Aspect-UtilitiesInitializeCurrentAspects`. 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`.

16.9.2. Configuring Aspects

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 `OnException-AspectConfigurationAttribute`.

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 `IAspectBuildSemanticsGetAspectConfiguration(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 `OnExceptionAspect-Configuration`. The aspect type `OnExceptionAspect` implements `IAspectBuildSemanticsGetAspect-Configuration(Object)` by creating an instance of `OnExceptionAspectConfiguration`, then it invokes the method `On-ExceptionAspectGetExceptionType(MethodBase)` and copies the return value of this method to the property `On-ExceptionAspectConfigurationExceptionType`. 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 `IAspectBuildSemanticsGetAspectConfiguration(Object)`. If your aspect does not derive from the aspect class `OnExceptionAspect`, but directly implements the aspect interface `IOn-ExceptionAspect`, you can use only the later method.

16.10. Examples

16.10.1. Tracing Method Execution

This code implements an aspect that writes a trace message before and after the execution of the methods to which the aspect is applied.

Requirements

PostSharp 2.0 Community Edition or higher

Demonstrates

The example demonstrates the use of `OnMethodBoundaryAspect`, and shows how to use `RuntimeInitialize(MethodBase)` to improve runtime performance.

Example

```csharp
using System;
using System.Diagnostics;
using System.Reflection;
using PostSharp.Aspects;

namespace Samples
{
    [Serializable]
    public sealed class TraceAttribute : OnMethodBoundaryAspect
    {
        // This field is initialized and serialized at build time, then deserialized at runtime.
        private readonly string category;

        // These fields are initialized at runtime. They do not need to be serialized.
        [NonSerialized] private string enteringMessage;
        [NonSerialized] private string exitingMessage;

        // Default constructor, invoked at build time.
        public TraceAttribute()
        {
        }

        // Constructor specifying the tracing category, invoked at build time.
        public TraceAttribute(string category)
        {
            this.category = category;
        }


        // Invoked only once at runtime from the static constructor of type declaring the target method.
        public override void RuntimeInitialize(MethodBase method)
        {
            string methodName = method.DeclaringType.FullName + method.Name;
            this.enteringMessage = "Entering " + methodName;
            this.exitingMessage = "Exiting " + methodName;
        }

        // Invoked at runtime before that target method is invoked.
        public override void OnEntry(MethodExecutionArgs args)
        {
            Trace.WriteLine(this.enteringMessage, this.category);
        }

        // Invoked at runtime after the target method is invoked (in a finally block).
        public override void OnExit(MethodExecutionArgs args)
        {
            Trace.WriteLine(this.exitingMessage, this.category);
        }
    }
}
```

Remarks

Note that fields `enteringMessage` and `exitingMessage` are initialized from method `RuntimeInitialize`. This method is invoked only once, before the aspect instance is used for the first time. It may have been possible to format the trace message from methods `OnEntry` and `OnExit`, but doing so would hurt performance for two reasons:

1. Getting the reflection object (`MethodBase`) is rather expensive.

2. Concatenating a string creates a new string instance, which causes an overhead to memory allocation and garbage collection.

The aspect results in instructions that can be inlined by the JIT/NGen compiler, which makes the aspect almost as fast as hand-written code.

16.10.2. Handling Exceptions

This code is an aspect that handles exceptions by opening a message box. It is useful in WPF applications to handle specific exceptions, for instance of type `IOException`

Requirements

PostSharp 2.0 Community Edition or higher

Demonstrates

This example demonstrates the aspect type `OnExceptionAspect`.

Example

```csharp
using System;
using System.Reflection;
using System.Windows;
using PostSharp.Aspects;

namespace Samples
{
    [Serializable]
    public sealed class ExceptionDialogAttribute : OnExceptionAspect
    {
        // We don't need this field at runtime, so we don't serialize it.
        [NonSerialized] private readonly Type exceptionType;

        public ExceptionDialogAttribute() : this(null)
        {
        }

        public ExceptionDialogAttribute(Type exceptionType)
        {
            this.exceptionType = exceptionType;

            // Set the default value for the dialog box.
            this.Message = "{0}";
            this.Caption = "Error";
        }

        public string Message { get; set; }
        public string Caption { get; set; }

        // Method invoked at build time. Should return the type of exceptions to be handled.
        public override Type GetExceptionType(MethodBase method)
        {
            return this.exceptionType;
        }

        // Method invoked at run time.
        public override void OnException(MethodExecutionArgs args)
        {
            // Format the exception message.
            string message = string.Format(this.Message, args.Exception.Message);

            // Finds the parent window of the dialog box.
            DependencyObject dependencyObject = args.Instance as DependencyObject;
            Window window = null;
            if (dependencyObject != null)
            {
                window = Window.GetWindow(dependencyObject);
            }

            if (window != null)
            {
```

```csharp
            // Display the error dialog with a parent window.
            MessageBox.Show(window, message, this.Caption, MessageBoxButton.OK, MessageBoxImage.Error);
        }
        else
        {
            // Display the error dialog without a parent window.
            MessageBox.Show(message, this.Caption, MessageBoxButton.OK, MessageBoxImage.Error);
        }

        // Do not rethrow the exception.
        args.FlowBehavior = FlowBehavior.Return;
    }
}
}
```

16.10.3. Caching the Result of a Method

The following class implements an aspect that caches the return value of methods to which it is applied.

Requirements

PostSharp 2.0 Community Edition or higher

Demonstrates

The example demonstrates the use of OnMethodBoundaryAspect, and shows how to use FlowBehavior to skip the execution of a method when its value is found in cache. The property MethodExecutionTag is used to store the cache key between the **OnEntry** and **OnSuccess** advices.

Example

```csharp
using System;
using System.Reflection;
using System.Text;
using System.Web;
using PostSharp.Aspects;

namespace Samples
{
    [Serializable]
    public sealed class CacheAttribute : OnMethodBoundaryAspect
    {
        // This field will be set by CompileTimeInitialize and serialized at build time,
        // then deserialized at runtime.
        private string methodName;

        // Method executed at build time.
        public override void CompileTimeInitialize(MethodBase method, AspectInfo aspectInfo)
        {
            this.methodName = method.DeclaringType.FullName + "." + method.Name;
        }

        private string GetCacheKey(object instance, Arguments arguments)
        {
            // If we have no argument, return just the method name so we don't uselessly allocate memory.
            if (instance == null && arguments.Count == 0)
                return this.methodName;

            // Add all arguments to the cache key. Note that generic arguments are not part of the cache
            // key, so method calls that differ only by generic arguments will have conflicting cache keys.
            StringBuilder stringBuilder = new StringBuilder(this.methodName);
            stringBuilder.Append('(');
            if (instance != null)
            {
                stringBuilder.Append(instance);
```

```
            stringBuilder.Append("; ");
        }

        for (int i = 0; i < arguments.Count; i++)
        {
            stringBuilder.Append(arguments.GetArgument(i) ?? "null");
            stringBuilder.Append(", ");
        }

        return stringBuilder.ToString();
    }

    // This method is executed before the execution of target methods of this aspect.
    public override void OnEntry(MethodExecutionArgs args)
    {
        // Compute the cache key.
        string cacheKey = GetCacheKey(args.Instance, args.Arguments);

        // Fetch the value from the cache.
        object value = HttpRuntime.Cache[cacheKey];

        if (value != null)
        {
            // The value was found in cache. Don't execute the method. Return immediately.
            args.ReturnValue = value;
            args.FlowBehavior = FlowBehavior.Return;
        }
        else
        {
            // The value was NOT found in cache. Continue with method execution, but store
            // the cache key so that we don't have to compute it in OnSuccess.
            args.MethodExecutionTag = cacheKey;
        }
    }

    // This method is executed upon successful completion of target methods of this aspect.
    public override void OnSuccess(MethodExecutionArgs args)
    {
        string cacheKey = (string) args.MethodExecutionTag;
        HttpRuntime.Cache[cacheKey] = args.ReturnValue;
    }
    }
}
```

Remarks

Note that the field `methodName` is initialized from method `CompileTimeInitialize(MethodBase, AspectInfo)`. This method is invoked at build time, then the value of the field is serialized into the assembly. Thanks to this mechanism, no reflection is needed at runtime.

16.10.4. Dispatching a Method Execution to the GUI Thread

This code implements an aspect that causes methods to which it is applied to be invoked on the GUI thread. Indeed, properties and methods of WPF objects can be accessed only from the thread from which the object was created. Other threads must use the Dispatcher of this object to dispatch the invocation of the method to the GUI thread. The `Asynchronous` property of this aspect allows the developer to specify that the method should be invoked asynchronously; in this case, the current thread will not wait until completion of the intercepted method.

Requirements

PostSharp 2.0 Community Edition or higher

Demonstrates

This example demonstrates the use of `MethodInterceptionAspect` to intercept invocations of method. Additionally, it shows bow to use `CompileTimeValidate(MethodBase)` to check that the aspect has been applied on a valid method.

Example

```csharp
using System;
using System.Linq;
using System.Reflection;
using System.Windows.Threading;
using PostSharp.Aspects;
using PostSharp.Extensibility;

namespace Samples
{
    [Serializable]
    [MulticastAttributeUsage(MulticastTargets.Method, TargetMemberAttributes = MulticastAttributes.Instance)]
    [AttributeUsage(AttributeTargets.Assembly | AttributeTargets.Class | AttributeTargets.Method, AllowMultiple = true)]
    public class GuiThreadAttribute : MethodInterceptionAspect
    {
        public bool Asynchronous { get; set; }

        // Method invoked at build time. It validates that the aspect has been applied to an acceptable method.
        public override bool CompileTimeValidate(MethodBase method)
        {
            // The method must be in a type derived from DispatcherObject.
            if (!typeof (DispatcherObject).IsAssignableFrom(method.DeclaringType))
            {
                Message.Write(SeverityType.Error, "CUSTOM02",
                            "Cannot apply [GuiThread] to method {0} because it is not a member of a type " +
                            "derived from DispatcherObject.", method);
                return false;
            }

            // If the call is asynchronous, there should not be any return value or ByRef parameter.
            if (this.Asynchronous)
            {
                if (((MethodInfo) method).ReturnType == typeof (void) ||
                    method.GetParameters().Any(parameter => parameter.ParameterType.IsByRef))
                {
                    Message.Write(SeverityType.Error, "CUSTOM02",
                                "Cannot apply [GuiThread(Asynchronous=true)] to method {0} because it is not a member " +
                                "of a type derived from DispatcherObject.", method);
                    return false;
                }
            }

            return true;
        }

        // Method invoked at run time _instead_ of the intercepted method.
        public override void OnInvoke(MethodInterceptionArgs args)
        {
            // Get the graphic object.
            DispatcherObject dispatcherObject = (DispatcherObject) args.Instance;

            // Check whether the current thread has access to this object.
            if (dispatcherObject.CheckAccess())
            {
                // We have access. Proceed with invocation.
                args.Proceed();
            }
            else
            {
                // We don't have access. Invoke the method synchronously.
                if (this.Asynchronous)
                {
                    dispatcherObject.Dispatcher.BeginInvoke(DispatcherPriority.Normal, new Action(args.Proceed));
```

```
                }
                else
                {
                    dispatcherObject.Dispatcher.Invoke(DispatcherPriority.Normal, new Action(args.Proceed));
                }
            }
        }
    }
}
```

16.10.5. Backing a Property with a Registry Value

This code is an aspect that binds a field or a property with a registry value. The first time the field or property is read, its value is retrieved from registry. Whenever the field or property is written, its value is written to registry.

Requirements

PostSharp 2.0 Community Edition or higher

Demonstrates

This example illustrates the use of the aspect type `LocationInterceptionAspect` in a situation where the location getter invokes the setter. It shows how the aspect can implement `IInstanceScopedAspect` to get the same scope (static or instance) than the location to which it is applied; thanks to this, we can use an aspect field (`fetchedFromRegistry`) to store the information whether the value has already been fetched from registry.

Example

```csharp
using System;
using Microsoft.Win32;
using PostSharp.Aspects;

namespace Samples
{
    public sealed class RegistryValueAttribute : LocationInterceptionAspect, IInstanceScopedAspect
    {
        // True if the the value has already been fetched from registry and stored in the field or property.
        // It is not serialize since we don't need it at build time.
        [NonSerialized] private bool fetchedFromRegistry;

        public RegistryValueAttribute(string keyName, string valueName)
        {
            this.KeyName = keyName;
            this.ValueName = valueName;
        }

        public string KeyName { get; private set; }
        public string ValueName { get; private set; }
        public object DefaultValue { get; set; }

        // Invoked at runtime whenever someone gets the value of the field or property.
        public override void OnGetValue(LocationInterceptionArgs args)
        {
            if (!this.fetchedFromRegistry)
            {
                // We have not fetched the value from registry. Do it now.
                object value = Registry.GetValue(this.KeyName, this.ValueName, this.DefaultValue);

                // Store this value in the target field/property.
                args.SetNewValue(value);

                // Return this value (we don't even need to call the underlying getter).
                args.Value = value;
```

```
            }
            else
            {
                // The value is already stored in the field/property, so just call the
                // underlying getter.
                args.ProceedGetValue();
            }
        }

        // Invoked at runtime whenever someone gets the value of the field or property.
        public override void OnSetValue(LocationInterceptionArgs args)
        {
            // Store the new value in registry if it has changed.
            if (Equals(args.Value, args.GetCurrentValue()))
            {
                Registry.SetValue(this.KeyName, this.ValueName, args.Value);
            }

            // Call the underlying setter.
            base.OnSetValue(args);

            // Sets that the underlying field/property already stores the field.
            this.fetchedFromRegistry = true;
        }

        #region Implementation of IInstanceScopedAspect

        object IInstanceScopedAspect.CreateInstance(AdviceArgs adviceArgs)
        {
            return this.MemberwiseClone();
        }

        void IInstanceScopedAspect.RuntimeInitializeInstance()
        {
        }

        #endregion
    }
}
```

16.10.6. Making an Event Asynchronous

This code is an aspect making an event asynchronous. When an event enhanced with this aspect is fired, subscribed handlers are invoked asynchronously. Whenever a subscribed handler fails with an exception, it is removed from the list of subscribers of this event.

Requirements

PostSharp 2.0 Professional Edition or higher

Demonstrates

This example illustrates the use of the aspect type `EventInterceptionAspect`.

Example

```
using System;
using System.Diagnostics;
using System.Threading.Tasks;
using PostSharp.Aspects;

namespace Samples
{
    [Serializable]
    public sealed class AsyncEventAttribute : EventInterceptionAspect
```

```
    {
        public override void OnInvokeHandler(EventInterceptionArgs args)
        {
            // Invoke the event handler asynchronously.
            Task.Factory.StartNew(() => Invoke(args)).Start();
        }

        private static void Invoke(EventInterceptionArgs args)
        {
            try
            {
                // Invoke the event handler.
                args.ProceedInvokeHandler();
            }
            catch (Exception e)
            {
                // Remove the event handler if it failed.
                Trace.TraceError(e.ToString());
                args.ProceedRemoveHandler();
            }
        }
    }
}
```

16.10.7. Dynamically Introducing an Interface

This aspect implement a very general way to compose types using a custom attribute. The custom attribute has two parameters: the type of the interface to be introduced, and the type of the class implementing the interface. This class should have a default constructor.

Requirements

PostSharp 2.0 Community Edition or higher

Demonstrates

This example demonstrates the use of `CompositionAspect`.

Example

```
using System;
using System.Collections;
using PostSharp;
using PostSharp.Aspects;

namespace Samples
{
    [Serializable]
    public sealed class GeneralComposeAttribute : CompositionAspect
    {
        // We don't need this field at runtime, so we mark it as non-serialized.
        [NonSerialized] private readonly Type interfaceType;

        private readonly Type implementationType;

        public GeneralComposeAttribute(Type interfaceType, Type implementationType)
        {
            this.interfaceType = interfaceType;
            this.implementationType = implementationType;
        }

        // Invoked at build time. We return the interface we want to implement.
        protected override Type[] GetPublicInterfaces(Type targetType)
        {
            return new[] {this.interfaceType};
```

```
        }

        // Invoked at run time.
        public override object CreateImplementationObject(AdviceArgs args)
        {
            return Activator.CreateInstance(this.implementationType);
        }
    }

    [GeneralCompose(typeof (IList), typeof (ArrayList))]
    internal class TestCompose
    {
        public TestCompose()
        {
            // Note the use of the Post.Cast method to get the implemented interface.
            IList list = Post.Cast<TestCompose, IList>(this);
            list.Add("apple");
            list.Add("orange");
            list.Add("banana");
        }
    }
}
```

16.10.8. Automatically Adding DataContract and DataMember Attributes

This aspect automatically adds custom attributes `DataContract` and `DataMember` to classes so they can be serialized by the WCF formatter. Properties that must not be serialized must be annotated with the custom attribute `NotDataMember-Attribute`.

Requirements

PostSharp 2.0 Professional Edition or higher

Demonstrates

This example demonstrates how `CustomAttributeIntroductionAspect` can be used together with `IAspectProvider`.

Example

```
using System;
using System.Collections.Generic;
using System.Reflection;
using System.Runtime.Serialization;
using PostSharp.Aspects;
using PostSharp.Extensibility;
using PostSharp.Reflection;

namespace Samples
{
    // We set up multicast inheritance so  the aspect is automatically added to children types.
    [MulticastAttributeUsage(MulticastTargets.Class, Inheritance = MulticastInheritance.Strict)]
    [Serializable]
    public sealed class AutoDataContractAttribute : TypeLevelAspect, IAspectProvider
    {
        // This method is called at build time and should just provide other aspects.
        public IEnumerable<AspectInstance> ProvideAspects(object targetElement)
        {
            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.
        yield return new AspectInstance(targetType, introduceDataContractAspect);

        // Add a DataMember attribute to every relevant property.
        foreach (PropertyInfo property in
            targetType.GetProperties(BindingFlags.Public | BindingFlags.DeclaredOnly | BindingFlags.Instance))
        {
            if (property.CanWrite && !property.IsDefined(typeof (NotDataMemberAttribute), false))
                yield return new AspectInstance(property, introduceDataMemberAspect);
        }
    }
}

[AttributeUsage(AttributeTargets.Property)]
public sealed class NotDataMemberAttribute : Attribute
{
}
}
```

16.10.9. Raising an Event When the Object is Finalized

This example shows an aspect that raises an event when instances of the target class is finalized (during garbage collection).

Requirements

PostSharp 2.0 Professional Edition or higher

Demonstrates

This example demonstrates how a composite aspect can be used to integrate an aspect with another implementation pattern, here `IDisposable`. It demonstrates that the lifetime of an instance-scoped aspect is bound to the lifetime of its target class.

Example

```
using System;
using PostSharp;
using PostSharp.Aspects;
using PostSharp.Aspects.Advices;
using PostSharp.Reflection;


namespace Samples
{
    // The aspect will introduce and implement this interface.
    public interface IObservableLifetime
    {
        // Event raised when the object is disposed.
        event EventHandler Disposed;

        // Event raised when the object is finialized.
        event EventHandler Finalized;
    }


    // The annotation IntroduceInterface specifies that the aspect introduces this interface.
    [Serializable]
    [IntroduceInterface(typeof (IObservableLifetime), OverrideAction = InterfaceOverrideAction.Fail)]
    public class ObservableLifetimeAttribute : InstanceLevelAspect, IObservableLifetime
    {
        // True if the aspect instance is a 'real' instance, false if it is the prototype instance.
        // We do not want to raise the Finalize event on prototype instances.
        [NonSerialized] private bool notPrototype;
```

```csharp
        // True if the aspect has already been invoked.
        [NonSerialized] private bool disposed;

        // Initializes the aspect instance.
        public override void RuntimeInitializeInstance()
        {
            this.notPrototype = true;
        }

        // At runtime, this field is set to a delegate of the method Dispose(bool) before we override it.
        [ImportMember("Dispose", IsRequired = true, Order = ImportMemberOrder.BeforeIntroductions)] public Action<bool>
            BaseDisposeMethod;

        // Overrides the method Dispose(bool) of the target type.
        [IntroduceMember(IsVirtual = true, OverrideAction = MemberOverrideAction.OverrideOrFail,
            Visibility = Visibility.Family)]
        public void Dispose(bool disposing)
        {
            // Ignore subsequent calls of this method.
            if (this.disposed)
                return;

            this.disposed = true;

            // Invoke the Dispose(bool) method of the base type.
            this.BaseDisposeMethod(disposing);


            // Raise the Disposed event.
            if (this.Disposed != null)
                this.Disposed(this.Instance, EventArgs.Empty);

            // Unlist this object from finalization.
            if (disposing)
            {
                GC.SuppressFinalize(this);
            }
        }

        // Introduces the event Disposed in the target type.
        [IntroduceMember(OverrideAction = MemberOverrideAction.Fail)]
        public event EventHandler Disposed;

        // Introduces the event Finalized in the target type.
        [IntroduceMember(OverrideAction = MemberOverrideAction.Fail)]
        public event EventHandler Finalized;

        // Finalizer.
        ~ObservableLifetimeAttribute()
        {
            // Ignore the finalizer if we are a prototype instance.
            if (!this.notPrototype)
                return;

            // Call the Dispose method of the target type.
            this.BaseDisposeMethod(false);

            // Raise the Finalized event.
            if (this.Finalized != null)
            {
                this.Finalized(this.Instance, EventArgs.Empty);
            }
        }
    }

    // A sample object.
    [ObservableLifetime]
    internal class DomainObject : IDisposable
    {
        private readonly string tag;
```

```csharp
        public DomainObject(string tag)
        {
            this.tag = tag;
        }

        protected virtual void Dispose(bool disposing)
        {
            Console.WriteLine("Dispose({0})", disposing);
        }

        public void Dispose()
        {
            this.Dispose(true);
        }

        public override string ToString()
        {
            return "{" + tag + "}";
        }
    }

    internal class Program
    {
        private static void Main(string[] args)
        {
            DomainObject f1 = new DomainObject("f1");
            IObservableLifetime fe1 = Post.Cast<DomainObject, IObservableLifetime>(f1);
            fe1.Finalized += OnFinalized;
            f1.Dispose();

            DomainObject f2 = new DomainObject("f2");
            IObservableLifetime fe2 = Post.Cast<DomainObject, IObservableLifetime>(f2);
            fe2.Finalized += OnFinalized;
            f2 = null;
            fe2 = null;
            GC.Collect();
            GC.WaitForPendingFinalizers();
        }

        private static void OnFinalized(object sender, EventArgs e)
        {
            Console.WriteLine("OnFinalized: " + sender);
        }
    }
}
```

The output of this program is:

```
Dispose(True)
Dispose(False)
OnFinalized: {f2}
```

Remarks

This example is made more complex by the fact that it understands the disposable pattern. The aspect requires the target class to implement the disposable pattern properly, i.e. to define the method `protected virtual void Dispose(bool)`. PostSharp will fail if the user tries to apply this aspect to a class that implements the pattern incorrectly.

The aspect overrides the `Dispose(bool)` method by defining a method of the same name and signature, and annotating it with the custom attribute `IntroduceMemberAttribute`.

In order to be able to invoke the original implementation of `Dispose(bool)`, the aspect defines the field `BaseDispose-Method`, whose type `Action<bool>` is a delegate compatible with the signature of the method `Dispose(bool)`. The custom attribute `ImportMemberAttribute` tells PostSharp to bind this field to the `Dispose` method.

Because the aspect class derives from `InstanceLevelAspect`, which implements `IInstanceScopedAspect`, aspect instances have the same lifetime as instances of the target class: both instances are garbage collected at the same time. Thanks to this, the aspect can reliably raise the `Finalized` event on the class instance when the aspect instance itself is collected.

CHAPTER 17

Testing and Debugging Aspects

Aspects should be tested as any piece of code. However, testing techniques for aspects differ from testing techniques for normal class libraries because of a number of reasons:

- Aspects instantiation is not user-controlled.

- Aspects partially execute at build time.

- Aspects can emit build errors. Logic that emits build errors should be tested too.

These characteristics are no obstacle to proper testing of aspects.

This chapter contains the following sections:

- Writing Simple Tests [299] explains how to test the behavior of an aspect.

- Testing that an Aspect has been Applied [301] shows how to test that an aspect has been applied to the expected set of code artifacts.

- Consuming Dependencies from the Aspect [302] describes several ways for aspects to consume dependencies from dependency-injection containers and service locators.

- Attaching a Debugger at Build Time [316] explains how to debug build-time logic.

17.1. Writing Simple Tests

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 artefact 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 the caching aspect illustrated in section Caching the Result of a Method [288]. High code coverage is achieved by varying the target code and testing with null and non-null parameters.

```csharp
using System;
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]
        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++;
        }
    }
}
```

17.2. 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 to test that the aspect has been property 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**.

- The aspect is applied using an `IAspectProvider`.

Testing that the aspect behavior is exhibited

The most obvious way to test that the aspect has been applied on 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 Writing Simple Tests [299].

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 [157].

NOTE

Aspects added by `IAspectProvider` 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.psssym*, 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.

17.3. Consuming Dependencies from the 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 runtime.

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:

17.3.1. 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 runtime. 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** method.

The dependency injection method can be invoked from the **RuntimeInitialize** method.

NOTE

User code has no control over the time when and the thread on which an aspect is initialized. Therefore, using `Thread-StaticAttribute` 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;
```

```csharp
using System.Reflection;
using PostSharp.Aspects;
using PostSharp.Extensibility;

namespace DependencyResolution.GlobalServiceContainer
{
    public interface ILogger
    {
        void Log(string message);
    }

    public static class AspectServiceInjector
    {
        private static CompositionContainer container;

        public static void Initialize(ComposablePartCatalog catalog)
        {
            container = new CompositionContainer(catalog);
        }

        public static void BuildObject(object o)
        {
            if (container == null)
                throw new InvalidOperationException();

            container.SatisfyImportsOnce(o);
        }
    }

    [Serializable]
    public class LogAspect : OnMethodBoundaryAspect
    {
        [Import] private ILogger logger;

        public override void RuntimeInitialize(MethodBase method)
        {
            AspectServiceInjector.BuildObject(this);
        }

        public override void OnEntry(MethodExecutionArgs args)
        {
            logger.Log("OnEntry");
        }
    }

    internal class Program
    {
        private static void 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 methods. This is why we cannot use the aspect in the Program class.
            Foo.LoggedMethod();
        }
    }

    internal class Foo
    {
        [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:

```csharp
using System;
using System.ComponentModel.Composition;
using System.ComponentModel.Composition.Hosting;
using System.Text;
using Microsoft.VisualStudio.TestTools.UnitTesting;

namespace DependencyResolution.GlobalServiceContainer.Test
{
    [TestClass]
    public class TestLogAspect
    {
        static TestLogAspect()
        {
            AspectServiceInjector.Initialize(new TypeCatalog(typeof (TestLogger)));
        }

        [TestMethod]
        public void TestMethod()
        {
            TestLogger.Clear();
            new TargetClass().TargetMethod();
            Assert.AreEqual("OnEntry" + Environment.NewLine, TestLogger.GetLog());
        }

        private class TargetClass
        {
            [LogAspect]
            public void TargetMethod()
            {
            }
        }
    }

    [Export(typeof (ILogger))]
    internal class TestLogger : ILogger
    {
        public static readonly StringBuilder stringBuilder = new StringBuilder();

        public void Log(string message)
        {
            stringBuilder.AppendLine(message);
        }

        public static string GetLog()
        {
            return stringBuilder.ToString();
        }

        public static void Clear()
        {
            stringBuilder.Clear();
        }
    }
}
```

17.3.2. 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 `PostSharpEnvironmentIsPostSharpRunning` property to make sure that this part of the static constructor is executed at runtime only, when PostSharp is *not* running.

In this case, dependency injection method such as **SatisfyImportsOnce** cannot be used. Instead, the container must be used as a service locator. For instance, MEF exposes the method `ExportProviderGetExport`.

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 **Lazy** 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:

```csharp
using System;
using System.ComponentModel.Composition;
using System.ComponentModel.Composition.Hosting;
using System.ComponentModel.Composition.Primitives;
using PostSharp.Aspects;
using PostSharp.Extensibility;

namespace DependencyResolution.GlobalServiceLocator
{
    public interface ILogger
    {
        void Log(string message);
    }

    public static class AspectServiceLocator
    {
        private static CompositionContainer container;

        public static void 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;
        }
    }

    [Serializable]
    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]
    public class TestLogAspect
    {
        static TestLogAspect()
        {
            AspectServiceLocator.Initialize(new TypeCatalog(typeof (TestLogger)));
        }

        [TestMethod]
        public void TestMethod()
```

```csharp
        {
            TestLogger.Clear();
            TargetMethod();
            Assert.AreEqual("OnEntry" + Environment.NewLine, TestLogger.GetLog());
        }

        [LogAspect]
        private void TargetMethod()
        {
        }
    }

    [Export(typeof (ILogger))]
    internal class TestLogger : ILogger
    {
        public static readonly StringBuilder stringBuilder = new StringBuilder();

        public void Log(string message)
        {
            stringBuilder.AppendLine(message);
        }

        public static string GetLog()
        {
            return stringBuilder.ToString();
        }

        public static void Clear()
        {
            stringBuilder.Clear();
        }
    }
}
```

17.3.3. 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 runtime.

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:

```csharp
using System;
using System.ComponentModel.Composition;
using System.ComponentModel.Composition.Hosting;
using System.ComponentModel.Composition.Primitives;
```

```csharp
using PostSharp.Aspects;
using PostSharp.Extensibility;

namespace DependencyResolution.Dynamic
{
    public interface ILogger
    {
        void Log(string message);
    }

    public static class AspectServiceLocator
    {
        private static CompositionContainer container;
        private static bool isCacheable;

        public static void Initialize(ComposablePartCatalog catalog, bool isCacheable)
        {
            if (AspectServiceLocator.isCacheable && container != null)
                throw new InvalidOperationException();

            container = new CompositionContainer(catalog);
            AspectServiceLocator.isCacheable = isCacheable;
        }

        public static Func<T> GetService<T>() where T : class
        {
            if (isCacheable)
            {
                return () => new Lazy<T>(GetServiceImpl<T>).Value;
            }
            else
            {
                return GetServiceImpl<T>;
            }
        }

        private static T GetServiceImpl<T>()
        {
            if (container == null)
                throw new InvalidOperationException();

            return container.GetExport<T>().Value;
        }
    }

    [Serializable]
    public class LogAspect : OnMethodBoundaryAspect
    {
        private static readonly Func<ILogger> logger;

        static LogAspect()
        {
            if (!PostSharpEnvironment.IsPostSharpRunning)
            {
                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))]
    internal class TestLogger : ILogger
    {
        public static readonly StringBuilder stringBuilder = new StringBuilder();

        public void Log(string message)
        {
            stringBuilder.AppendLine(message);
        }

        public static string GetLog()
        {
            return stringBuilder.ToString();
        }

        public static void Clear()
        {
            stringBuilder.Clear();
        }
```

```
    }
}
```

17.3.4. 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 maybe 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:

```csharp
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;

namespace DependencyResolution.Contextual
{
    public interface ILogger
    {
        void Log(string message);
    }

    public static class AspectServiceLocator
    {
        private static CompositionContainer container;
        private static HashSet<object> rules = new HashSet<object>();

        public static void Initialize(ComposablePartCatalog catalog)
        {
            container = new CompositionContainer(catalog);
        }

        public static Lazy<T> GetService<T>(Type aspectType, MemberInfo targetElement) where T : class
        {
            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>().Value;
        }
```

```csharp
        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))]
    internal class ConsoleLogger : ILogger
    {
        public void Log(string message)
        {
            Console.WriteLine(message);
        }
    }
}
```

The following code snippet shows how the logging aspect can be tested:

```csharp
using System;
using System.Text;
using Microsoft.VisualStudio.TestTools.UnitTesting;

namespace DependencyResolution.Contextual.Test
{
    [TestClass]
    public class TestLogAspect
    {
        [TestMethod]
        public void 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]
        public void TargetMethod()
        {
        }
    }

    internal class TestLogger : ILogger
    {
        public static readonly StringBuilder stringBuilder = new StringBuilder();

        public void Log(string message)
        {
            stringBuilder.AppendLine(message);
        }

        public static string GetLog()
        {
            return stringBuilder.ToString();
        }

        public static void Clear()
        {
            stringBuilder.Clear();
        }
    }
}
```

17.3.5. 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 `IInstance-ScopedAspect` interface. See Understanding Aspect Lifetime and Scope [259] 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 `Member-OverrideAction.Ignore` to ignore the property if it already exists in the target type of 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 parameter. 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:

```csharp
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;

namespace DependencyResolution.InstanceScoped
{
    public interface ILogger
    {
        void Log(string message);
    }


    [Serializable]
    public class 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();
            service.TargetMethod();
            Assert.AreEqual("OnEntry" + Environment.NewLine, TestLogger.GetLog());
        }

        [Export(typeof (TestImpl))]
        private class TestImpl
        {
            [LogAspect]
            public void TargetMethod()
            {
            }
        }
    }

    [Export(typeof (ILogger))]
    internal class TestLogger : ILogger
```

314

```
    {
        public static readonly StringBuilder stringBuilder = new StringBuilder();

        public void Log(string message)
        {
            stringBuilder.AppendLine(message);
        }

        public static string GetLog()
        {
            return stringBuilder.ToString();
        }

        public static void Clear()
        {
            stringBuilder.Clear();
        }
    }
}
```

17.4. 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 Attaching a Debugger at Build Time [316]). 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

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* (download[16]). 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.

---

16. http://www.postsharp.net/downloads/samples/3.0/PostSharp.BuildTests.targets

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 **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.

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 cross 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.

17.5. Attaching a Debugger at Build Time

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 to:

1. Open `MyAspects.csproj` and `MyAspects.Test.csproj` in two different instances of Visual Studio.

2. Open the Visual Studio Command Prompt and go to the directory containing `MyAspects.Test.csproj`.

3. Build `MyAspects.csproj` using Visual Studio as usually .

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. Set up break points in your code and continue the program execution.

CHAPTER 18
Validating Architecture

18.1. 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.
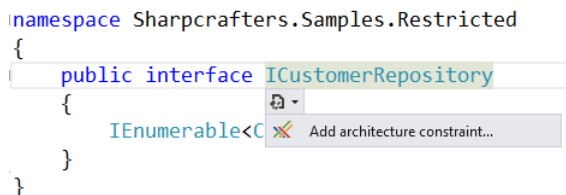
Adding the constraint to the interface

To restrict implementation of publicly declared interfaces you simply need to add [`InternalImplementAttribute`] to that interface.

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   Add architecture constraint...
    }
}
```

2. Select "Prevent interface implementation in a different assembly" and select **Next**.
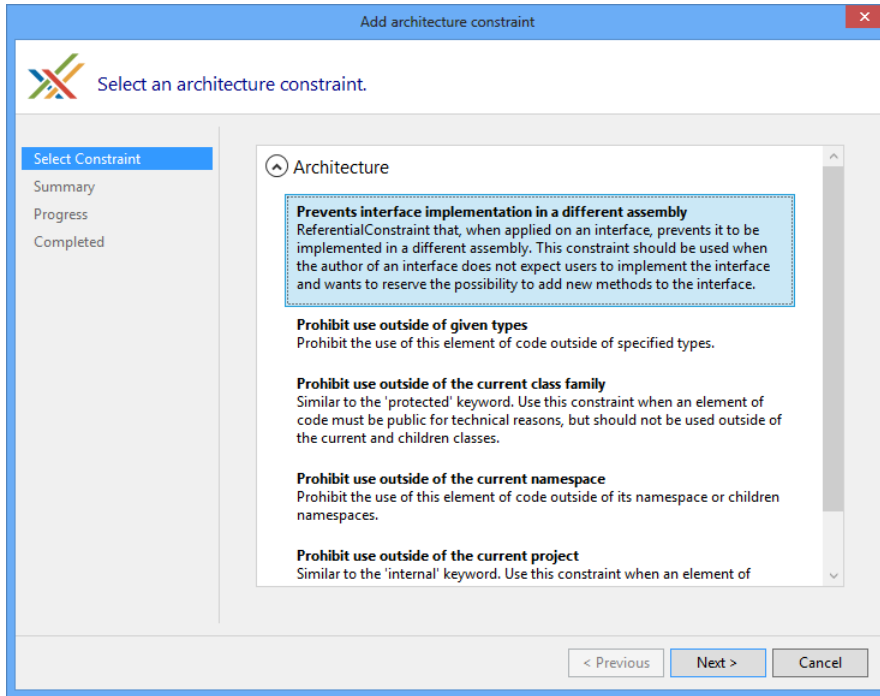
3. Verify that you will be adding the `InternalImplementAttribute` attribute to the correct piece of code.
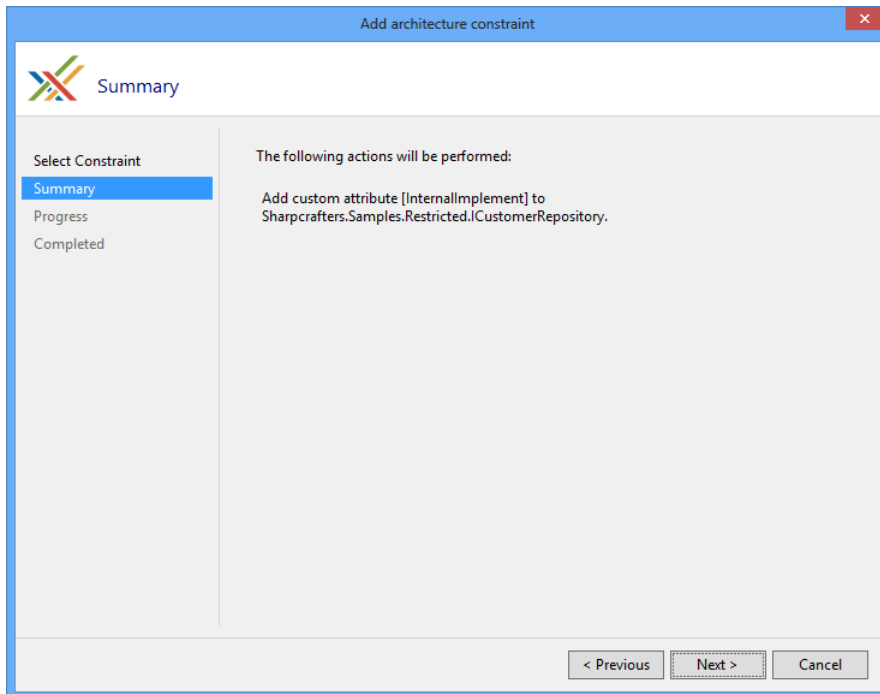
4. Once the download, installation and configuration of PostSharp has finished you can close the wizard and look at the changes that were made to your codebase.

5. You'll notice that the only thing that has changed in the code is the addition of the `[InternalImplement-Attribute]` attribute.

```
[InternalImplement]
public interface 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.
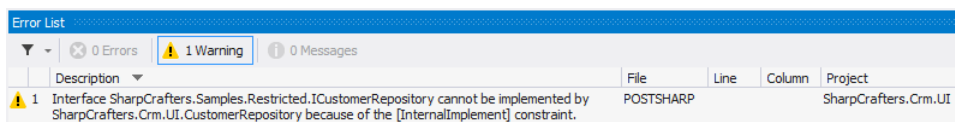
NOTE

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
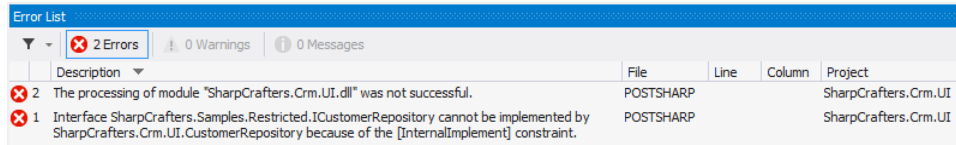
If a warning isn't strong enough for your environment you can change the output to a compile time error by setting the `InternalImplementAttribute` to have a `Severity type` of `Error`.

```
[InternalImplement(Severity = SeverityType.Error)]
public interface ICustomerRepository
```

```
{
    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")]
public class PreferredCustomerRepository : ICustomerRepository
{
    public IEnumerable<Customer> FetchAll()
    {
        return null;
    }
}
```

18.2. 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.
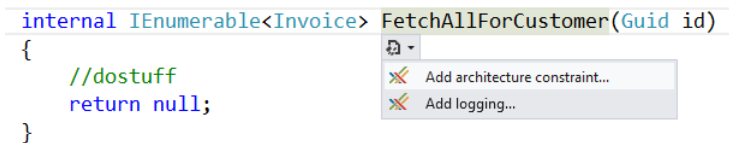
This topic contains the following sections.

Restricting access to specific namespaces

Our first step is to limit access to this 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.
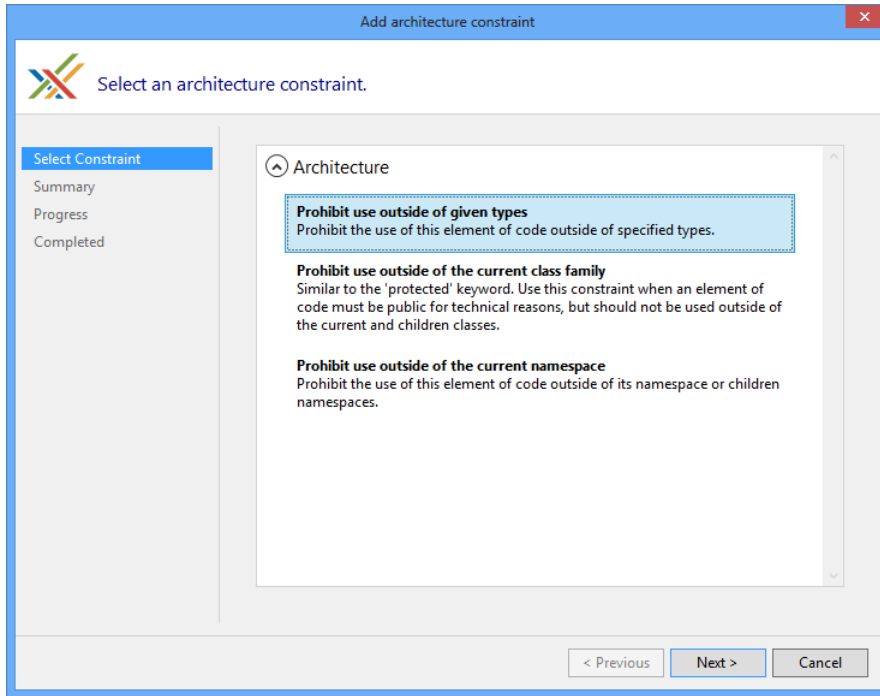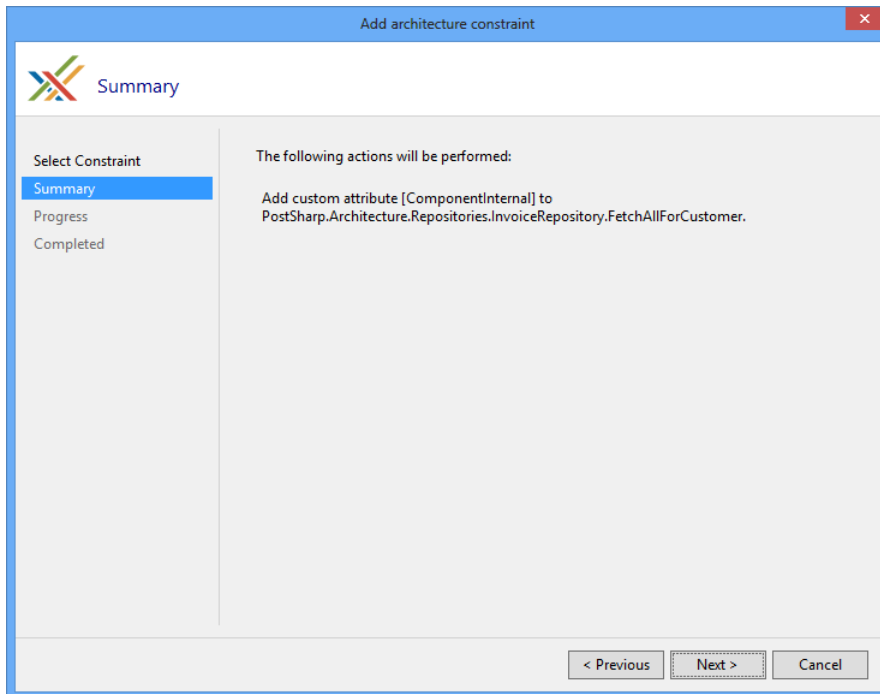
```csharp
internal IEnumerable<Invoice> FetchAllForCustomer(Guid id)
{
    //dostuff
    return null;
}
```

╳ Add architecture constraint...
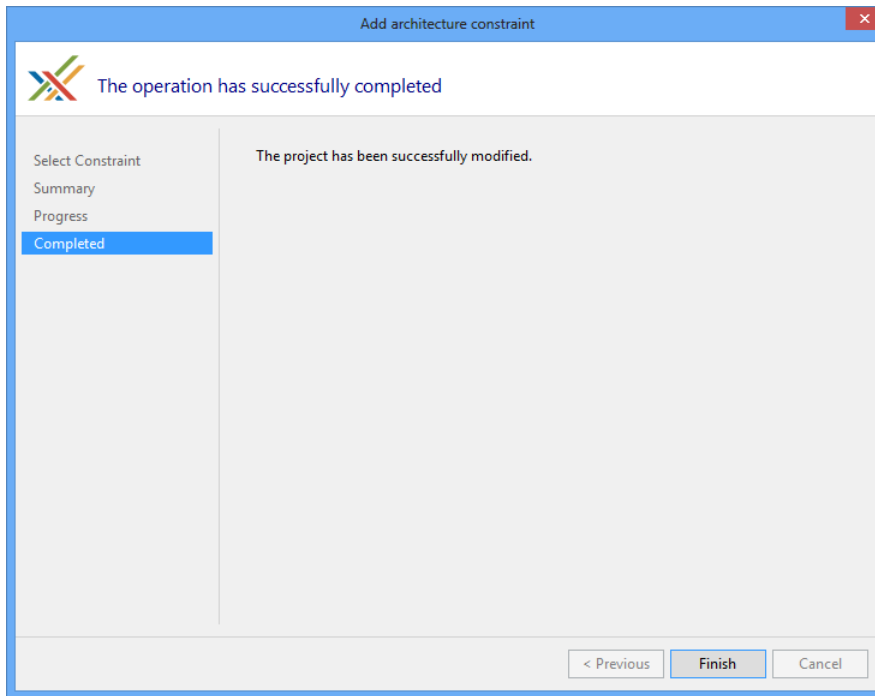╳ Add logging...

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.

4. Once the download, installation and configuration of PostSharp has finished you can close the wizard and look at the changes that were made to your codebase.

5. You'll notice that the only thing that has changed in the code is the addition of the [ComponentInternal-Attribute] attribute.

```
namespace Sharpcrafters.Crm.Console.Repositories
{
    public class InvoiceRepository
    {
        [ComponentInternal]
        internal IEnumerable<Invoice> FetchAllForCustomer(Guid id)
        {
            //dostuff
            return null;
        }
    }
}
```

6. 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.

```csharp
namespace Sharpcrafters.Crm.Console.Services
{
    public class 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
                    });
        }
    }
}
```

					File	Line	Column	Project
⚠	1	Method SharpCrafters.Crm.Console.InvoiceRepository.FetchAllForCustomer cannot be referenced from method SharpCrafters.Crm.Console.Services.Sharpcrafters.Crm.Console.Services.CustomerServices.FetchAllInvoicesForCustomer because of the [ComponentInternal] constraint.				POSTSHARP			SharpCrafters.Crm.Console

Error List — ❌ 0 Errors | ⚠ 1 Warning | ⓘ 0 Messages

> **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.

```csharp
public class InvoiceRepository
{
    [ComponentInternal(typeof(Sharpcrafters.Crm.Console.Services.InvoiceServices))]
    internal IEnumerable<Invoice> FetchAllForCustomer(Guid id)
    {
        //dostuff
        return null;
    }
}
```

2. 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());
    }
}
```

Error List						
▼ ▾	⊗ 0 Errors	⚠ 1 Warning	① 0 Messages			
	Description ▼		File	Line	Column	Project
⚠ 1	Method SharpCrafters.Crm.Console.InvoiceRepository.FetchAllForCustomer cannot be referenced from method SharpCrafters.Crm.Console.Services.Sharpcrafters.Crm.Console.Services.CustomerServices.FetchAllInvoicesForCustomer because of the [ComponentInternal] constraint.		POSTSHARP			SharpCrafters.Crm.Console
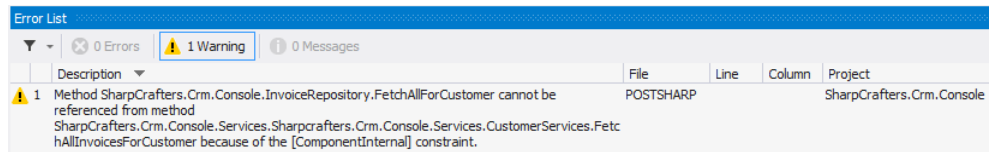
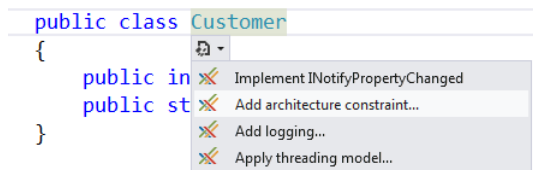## Controlling component visibility outside of the containing assembly

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 runtime, 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.

1. 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; }
    }
}
```

2. 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".
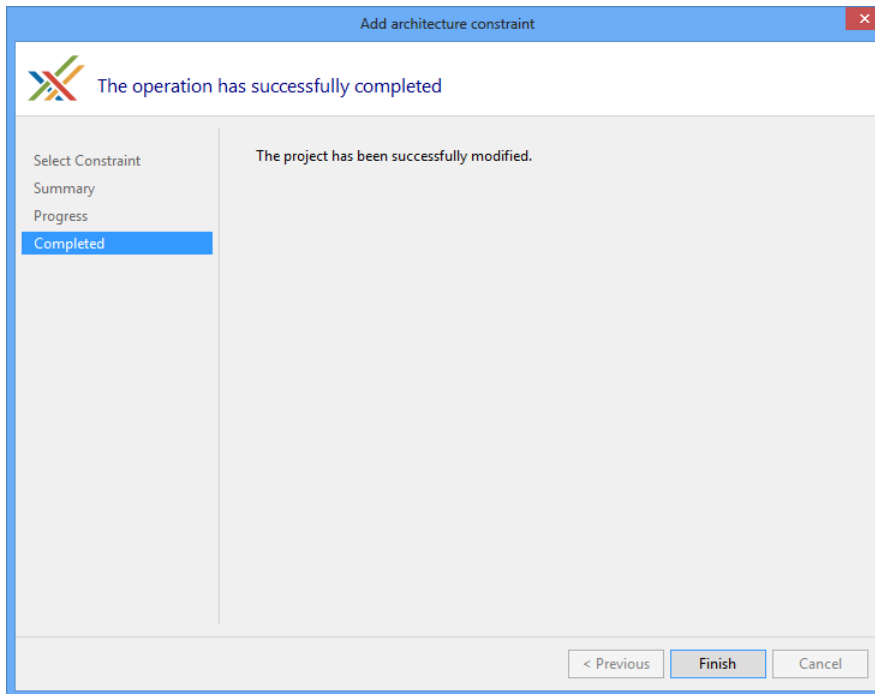
```
public class Customer
{
    public in ✕  Implement INotifyPropertyChanged
    public st ✕  Add architecture constraint...
}         ✕  Add logging...
          ✕  Apply threading model...
```

3. When prompted to select a constraint, choose to "Prohibit use outside of the project".
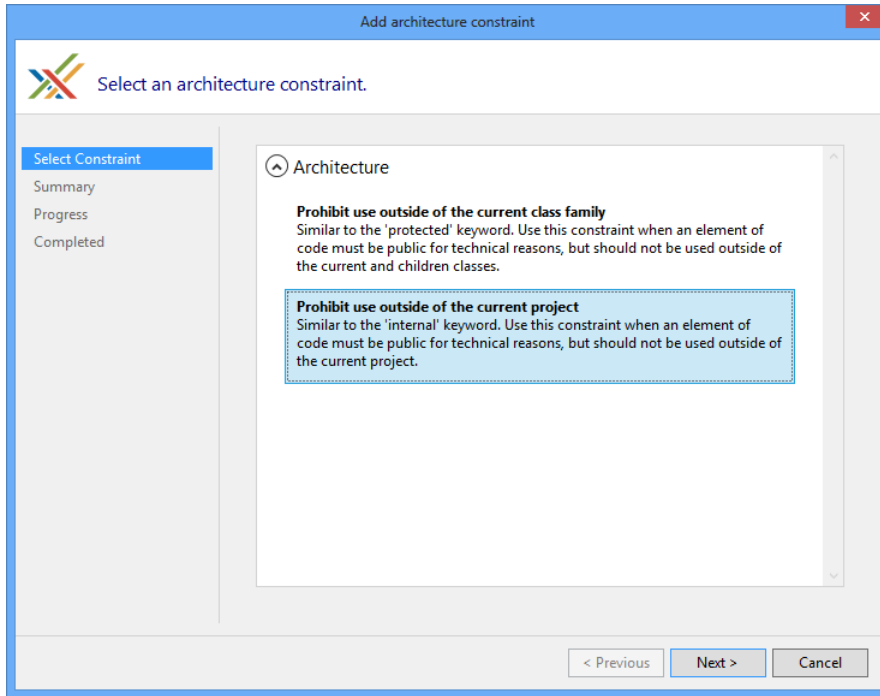
4. The summary page gives you the opportunity to review the selections that you have made. If you notice that the configuration is not what you wanted you can click the **Previous** button and adjust your selections. If the configuration meets your needs click **Next**. In this demo you will see that the [`InternalAttribute`] attribute is being added to the `Customer` class.
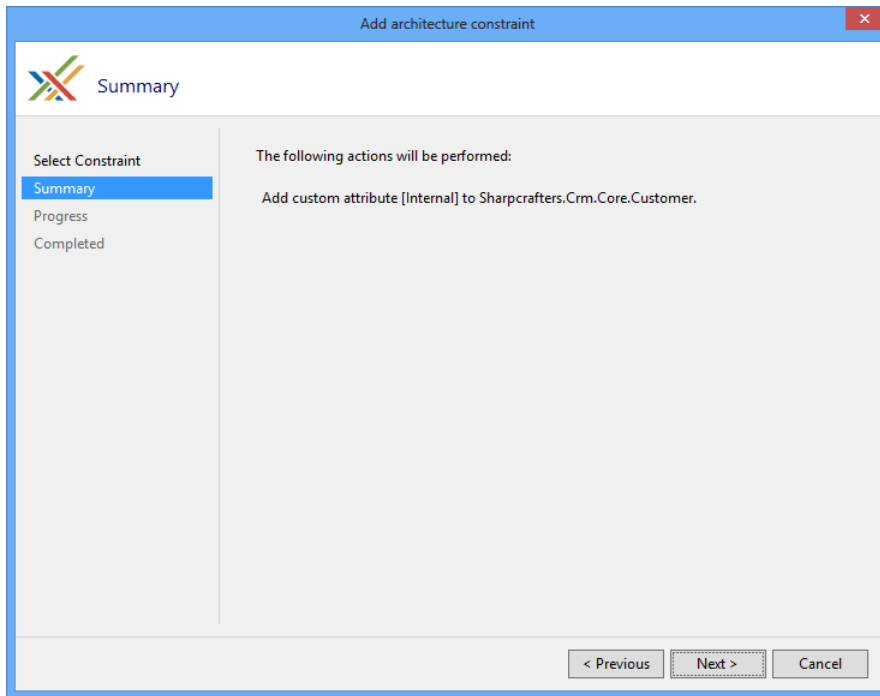
5. Once the download, installation and configuration of PostSharp has finished you can close the wizard and look at the changes that were made to your codebase.

6. 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]
    public class Customer
    {
        public int Id { get; set; }
        public string Name { get; set; }
    }
}
```

7. 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
{
    public class CustomerRepository:ICustomerRepository
    {
        public IEnumerable<Customer> FetchAll()
        {
            return new 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
    return null;
}
```

2. Now when you try to access the component when access hasn't been granted the Output window will display an error message.
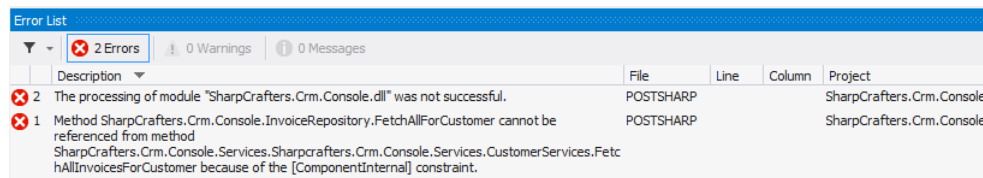
Ignoring warnings

There may be specific situations where you want to supress 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 `IgnoreWarning-Attribute` attribute to that method.

```
public class 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());
    }
}
```

18.3. 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

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;
   public class BusinessRulePatternValidation : ScalarConstraint
   {
   }
   ```

2. Designate what code construct type this validation aspect should work for by adding the `MulticastAttribute-UsageAttribute` 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)]
public class BusinessRulePatternValidation : ScalarConstraint
{
}
```

3. Override the `ValidateCode(Object)` method.

```
[MulticastAttributeUsage(MulticastTargets.Class, Inheritance = MulticastInheritance.Strict)]
public class BusinessRulePatternValidation : ScalarConstraint
{
    public override void 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)]
public class BusinessRulePatternValidation : ScalarConstraint
{
    public override void 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`, `Method-Info`, `ConstructorInfo`, `PropertyInfo`, `EventInfo`, `FieldInfo`, and `ParameterInfo`.

5. Write a warning that the rule being broken to the Output window in Visual Studio.

```
[MulticastAttributeUsage(MulticastTargets.Class, Inheritance = MulticastInheritance.Strict)]
public class BusinessRulePatternValidation : ScalarConstraint
{
    public override void 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);
        }
    }
}
```

6. 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]
public class 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 [147]

7. Now if you compile the project you will see an error in the Output window of Visual Studio when you run a build.
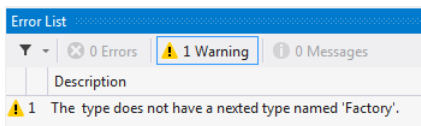
8. In some circumstances you may determine that a warning isn't aggressive enough. We can alter the rule that you have created so that it outputs a compile time error instead. All that you need to do is change the `SeverityType` in the `Message.Write` to `Error`.

```
[MulticastAttributeUsage(MulticastTargets.Class, Inheritance = MulticastInheritance.Strict)]
public class BusinessRulePatternValidation : ScalarConstraint
{
    public override void 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);
        }
    }
}
```
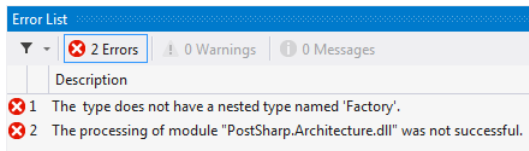

Using this technique it is possible to create rules or restrictions based on a number of different criteria and implement validation for several design patterns.

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.

```
 public class 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)]
public class BusinessRuleUseValidation : ReferentialConstraint
{
}
```

3. Override the `ValidateCode(Object, Assembly)` method.

```
[MulticastAttributeUsage(MulticastTargets.Class, Inheritance = MulticastInheritance.Strict)]
BusinessRuleUseValidation : ReferentialConstraint
{
    public override void 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)]
public class BusinessRulePatternValidation : ScalarConstraint
{
    public override void 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)]
public class BusinessRulePatternValidation : ScalarConstraint
{
    public override void 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]
    public class AccountRepository
    {
        public void 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 [147].

7. Now when you compile the project you will see a warning in the Output window in Visual Studio.
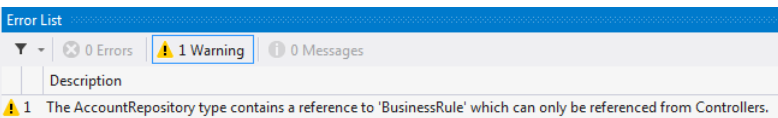
NOTE

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. Altough 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 `BusinessRule-PatternValidation` 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.