PostSharp 3.0

Reference Documentation

Table of Contents

Conceptual Documentation

PostSharp is a tool that allows development teams to achieve more with less code in Microsoft .NET:

1. **Software developers** will write clean, stable, efficient, and concise code. Fewer lines of code mean less time and fewer defects.
2. **Software architects** will be able to deliver partially or fully executable patterns, not just coding guidelines. After having carefully identified and selected design patterns, architects can code, in C# or Visual Basic, how the pattern should be implemented. Depending on the complexity of the pattern, PostSharp will automatically implement the pattern and/or will automatically validate that its manual implementation respects predefined validity rules.

This topic contains the following sections.

About Design Pattern Automation

PostSharp can be categorized as a *Design Pattern Automation* tool.

Design Pattern Automation is the use of tools to optimize productivity of development teams implementing software based on patterns. *Software patterns* are general reusable solutions to a commonly occurring problem. *Software design patterns* are patterns in the context of software design.

Typically, the architecture team would identify and select design patterns in an early phase of a project. Architects would then create designs that implement the design patterns. These designs typically include class diagrams, and instructions specifying how developers should implement the design. Design specifications are usually expressed in natural language. *Design Pattern Automation* refers to the ability to express designs in a formal language and use tools to help implementing the design.

Tools can help developers and architects in implementing designs derived from patterns in two ways:

- Tools can automatically implement some of the artefacts and behaviors required by the design, which then don't appear in source code. Design Pattern Automation therefore raises the level of abstraction of the source code, and makes the design intent more apparent in source code.

- Tools can validate hand-written code against design rules, from simple naming conventions to complex rules involving analysis of source code. Without tools, this activity would have only relied on code review.

Design Pattern Automation can extend to patterns that are not strictly considered *design patterns*:

- Design Pattern Automation extends to *application-specific patterns* (also named *custom patterns*), which are specific to the language, framework, and problem domain of an application. Althought these patterns are not general to the whole software engineering industry, their implementation can also be automatically generated and/or validated.
- Design Pattern Automation extends to *implementation patterns*, which are mere repetitions of code. Logging and exception handling belong to that category.

PostSharp components

PostSharp is composed of the following frameworks and libraries:

- *PostSharp Pattern Libraries* provide ready-made, commoditized implementations of some of the most common patterns in .NET. For more information, see section Working with Ready-Made Aspects at page 41.
- *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. 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. PostSharp contains the most advanced AOP framework for Microsoft .NET. For more information, see sections Developing Custom Aspects at page 141 and Adding Aspects to Code at page 113.
- *PostSharp Architecture Framework* is a static analysis tool that 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 artefacts. The framework provides tools to analyze the relationships between code artefacts and have access to the AST of method bodies. For more information, see section Enforcing Design Rules at page 253.

How does PostSharp work?

PostSharp inserts itself in the build process and enhances or validates the output of the C# or VB compiler. 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.

CHAPTER 1

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.

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.

Model Pattern Library

The NotifyPropertyChangedAttribute aspect is a ready-made implementation of the NotifyProperty-Changed 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 multithreading is being addressed. It provides three threading models: actors (Actor), reader-writer synchronized (ReaderWriterSynchronizedAttribute) and thread unsafe (ThreadUnsafeAttribute). Additionally, BackgroundAttribute and DispatchedAttribute allow you to easily dispatch a thread back and forth between a background and the UI thread.

Smart Tags and Wizards in Visual Studio

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

Better platform support through Portable Class Libraries

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

Unified deployment through NuGet and Visual Studio Gallery

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

Transparency to obfuscators

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

Deprecation of old platforms

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

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.

Better Build-Time Performance

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[1]. Local installation is no longer a requirement to use the Visual Studio Extension. However, because the setup program creates ngenned images, it still provides the faster experience.

Compatibility with Obfuscators

PostSharp can now be used jointly, and without limitation of features, with some obfuscators. See Obfuscation Tools at page 286 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 Enforcing Design Rules at page 253 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.

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

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.

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).

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: Location-InterceptionAspect (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 Time Improvements

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

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.

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 Enhancement

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

Support for Novell 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.

CHAPTER 2

Deploying and Configuring PostSharp

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. All components required for build are published as NuGet packages. These packages are typically stored in source control, or can be restored from a package repository before build. The user interface published as a Visual Studio extension (*vsix* 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

This chapter contains the following sections:

2.1. Requirements

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

- Microsoft Visual Studio 2010 or 2012, except Express editions.
- Windows XP with SP3 or later.
- 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

NuGet Package Manager is still considered unsuitable for some corporate environments, especially in situations with a large number of Visual Studio solutions. Concerns principally include package versioning management. Please contact our technical support if this is a concern for your team.

2.2. Installing PostSharp

PostSharp is composed of two components:

- The **Visual Studio Extension** is the user interface of PostSharp. It extends the Visual Studio editor and provides a new menu, option pages, and toolbox windows. The Visual Studio Extension must be installed on each developer workstation but is not required on build servers.
- The **NuGet package** contains the build-time components, which integrate into MSBuild, and the run-time libraries, which should be deployed with the customer's application. The NuGet package is typically included in the source repository or is restored before build from the package repository.

This topic contains the following sections.

Installing PostSharp

By *installing PostSharp*, we mean installing its user interface on a developer's computer. This procedure does not add PostSharp to any project.

To install PostSharp:

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.

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 PostSharp 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[2] and NuGet Configuration Settings[3] for more information.

Removing PostSharp from a project

To remove PostSharp from a project:

1. Open the **Solution Explorer** in Visual Studio.
2. Right-click on the project.
3. Click on **Manage NuGet Packages**.
4. Click on **Installed Packages**.
5. Find the *PostSharp* package and click on **Remove**.

2. http://docs.nuget.org/docs/reference/nuget-config-file
3. http://docs.nuget.org/docs/reference/nuget-config-settings

Uninstalling PostSharp

To uninstall PostSharp from Visual Studio:

1. Open Visual Studio.
2. Click on menu **Tools**, then **Extensions and Updates**.
3. Find the *PostSharp* extension and click on **Uninstall**.
4. Restart Visual Studio.

Updating PostSharp

When updating PostSharp, remember that it is composed of two kinds of components: the Visual Studio extension and the NuGet packages. These components can be updated separately. The Visual Studio extension is only loosely coupled to the NuGet packages, so the versions do not need to match.

To update the Visual Studio extension:

1. Open Visual Studio.
2. Click on menu **Tools**, then **Extensions and Updates**.
3. Click on menu **Updates**, then **Visual Studio Gallery**.
4. Find the *PostSharp* extension and click on **Update**.
5. Restart Visual Studio.

⚠ Caution

Updating the Visual Studio extension for PostSharp does not update the PostSharp NuGet packages.

To update the NuGet packages:

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**.

2.3. 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 at page 17
- Subscribing to a license server using the licensing wizard at page 20
- Installing the license settings in source control at page 23
- License activation and audit at page 24

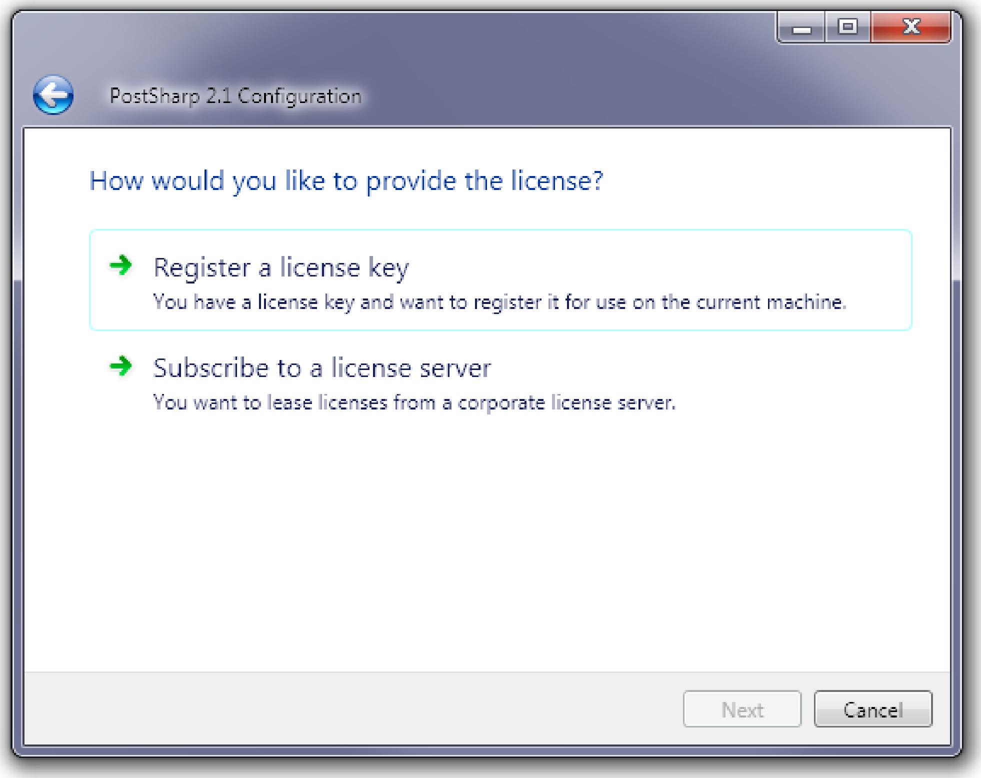
Registering a license key using the user interface

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.

5. Click on **Register a license**.

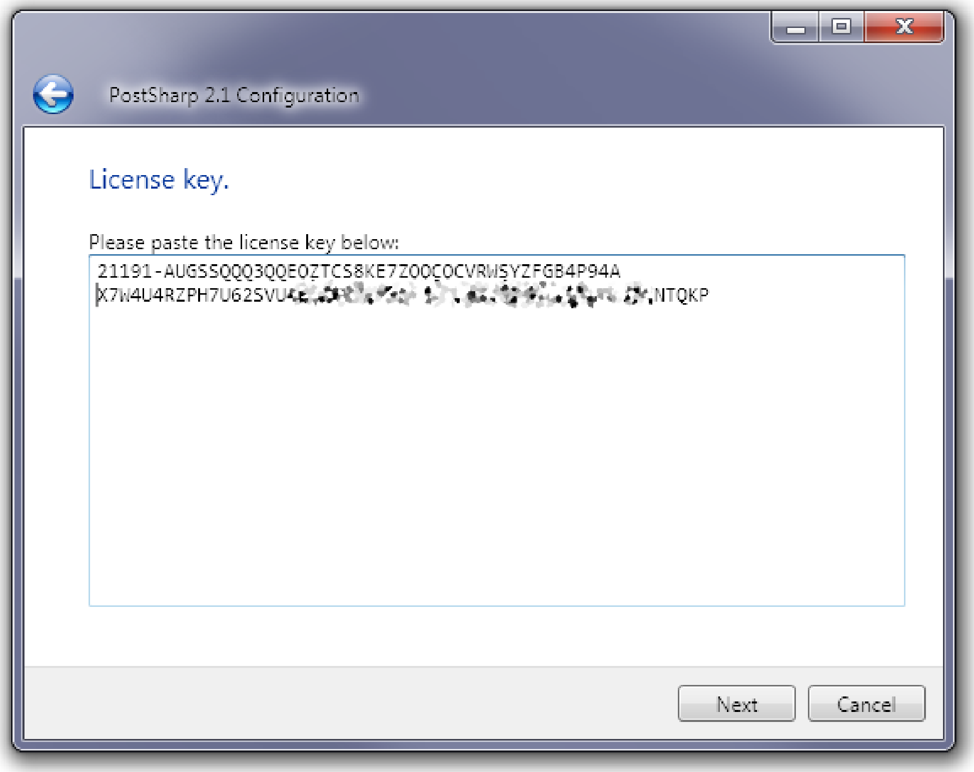


How would you like to provide the license?

Register a license key
You have a license key and want to register it for use on the current machine.

Subscribe to a license server
You want to lease licenses from a corporate license server.

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

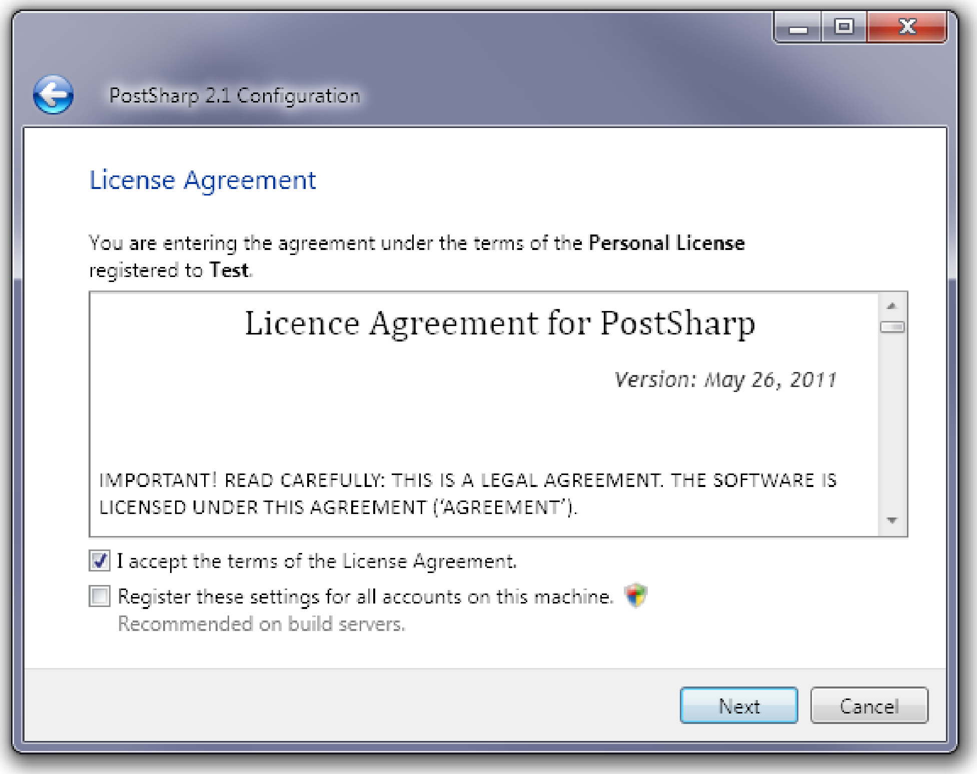


License key.

Please paste the license key below:

```
21191-AUGSSQQQ3QQEOZTCS8KE7ZOOCOCVRWSYZFGB4P94A
X7W4U4RZPH7U62SVU[redacted]NTQKP
```

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



License Agreement

You are entering the agreement under the terms of the **Personal License** registered to **Test**.

Licence Agreement for PostSharp

Version: May 26, 2011

IMPORTANT! READ CAREFULLY: THIS IS A LEGAL AGREEMENT. THE SOFTWARE IS LICENSED UNDER THIS AGREEMENT ('AGREEMENT').

[✓] I accept the terms of the License Agreement.

[ ] Register these settings for all accounts on this machine.
Recommended on build servers.

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.

Subscribing to a license server using the licensing wizard

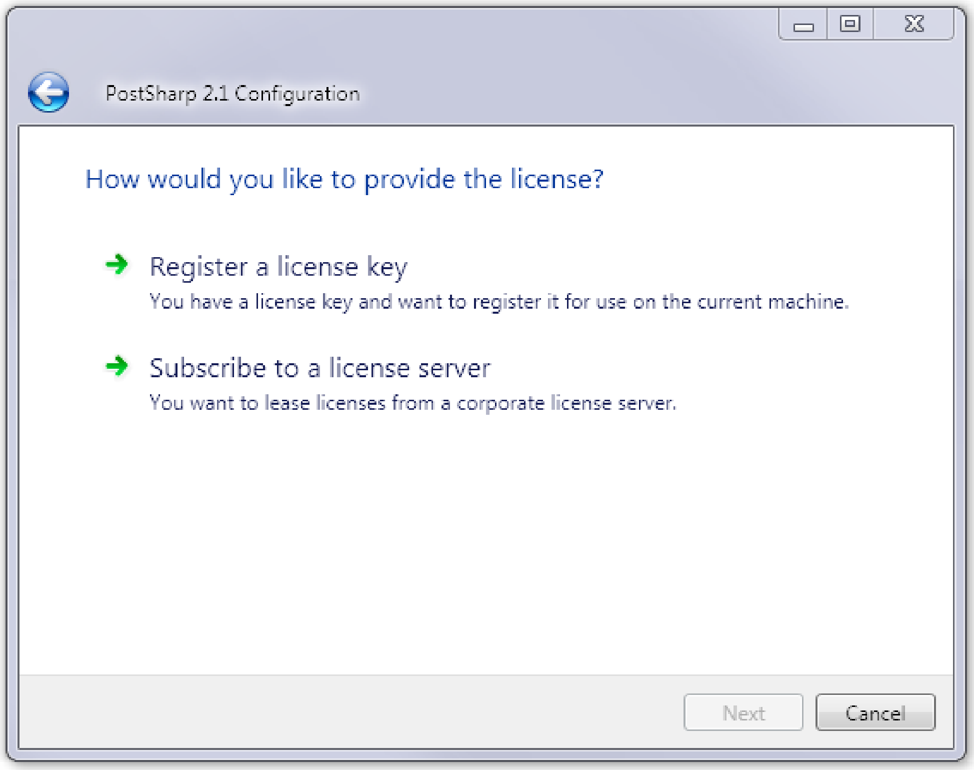
PostSharp Commercial Licenses are floating licenses. The license server is a product that provides license leases to developers and computes how many developers are using the product at any time. If the number of concurrent users exceeds the licensed number, the license administrator will receive an email, and excess 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.

Caution

Using a license server is substantially more cumbersome than other deployment options, becauses it requires your team to maintain an ASP.NET application running IIS as well as an SQL database. Remember that the use of the license server is purely optional and benevolent. Before deciding to use the license server, you may want to consider other options.

To subscribe to a license server using the licensing wizard:

1. Ask your administrator to download and install the license server. It consists in a MS SQL database and an ASP.NET application.
2. Open Visual Studio.
3. Click on menu **PostSharp**, then **Options**.
4. Open the **License** option page.
5. Click on the **Register a license** link.
6. Click **Subscribe to a license server**.

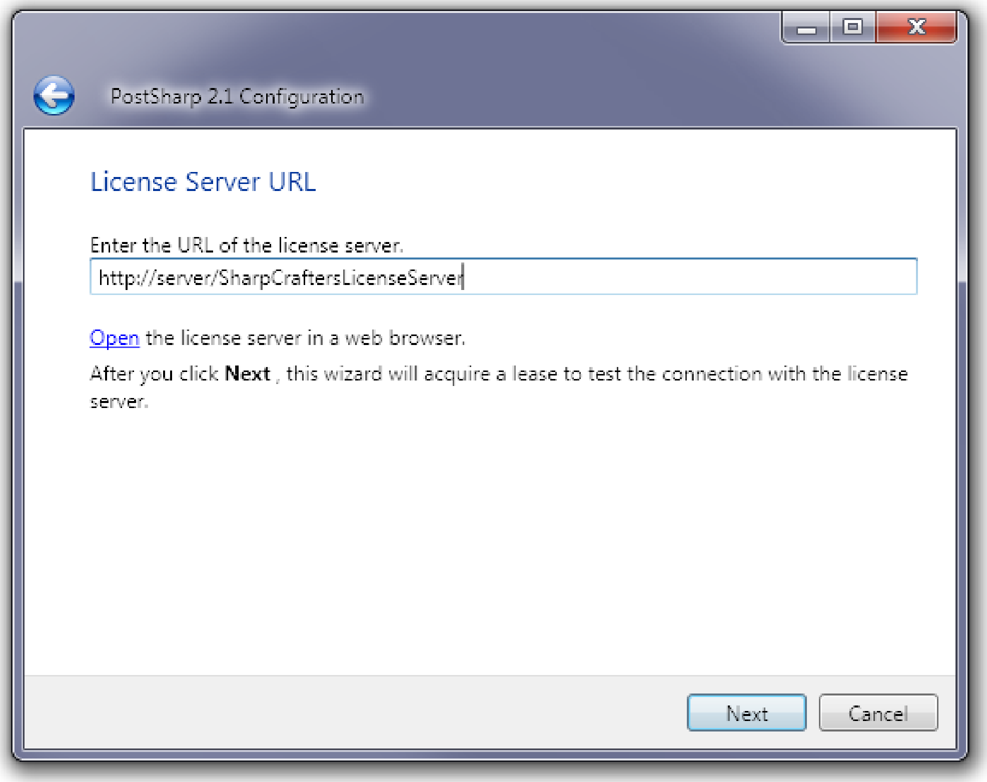


How would you like to provide the license?

Register a license key
You have a license key and want to register it for use on the current machine.

Subscribe to a license server
You want to lease licenses from a corporate license server.

7. Enter the URL where the license server has been installed. You should be able to access this URL with a browser. Click **Next**.



License Server URL

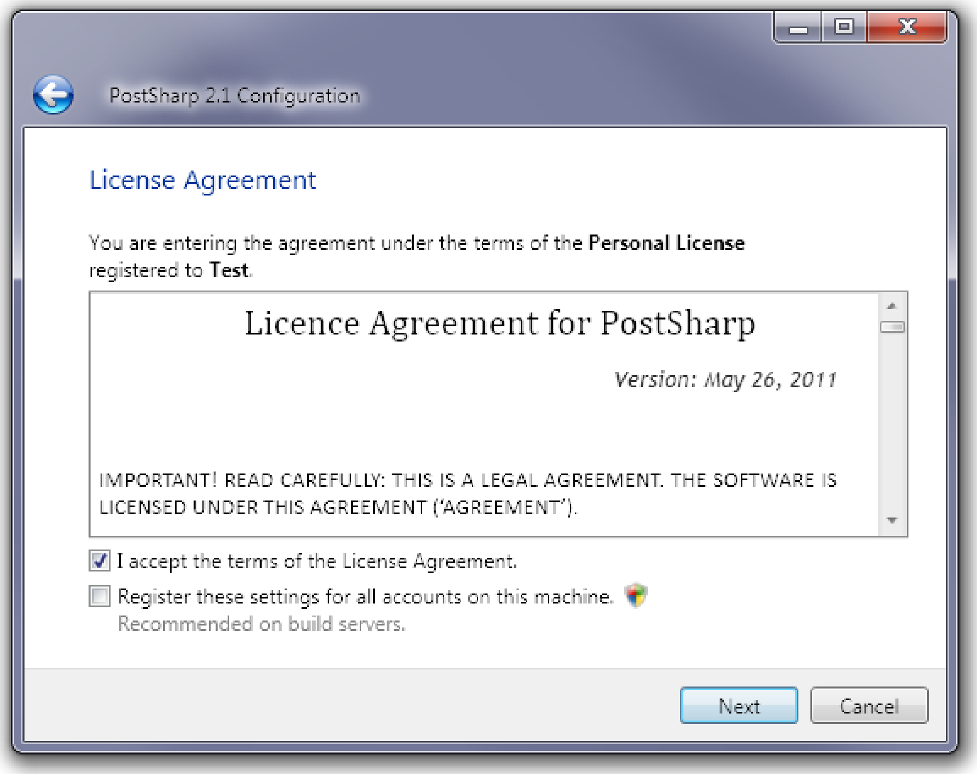
Enter the URL of the license server.

http://server/SharpCraftersLicenseServer

Open the license server in a web browser.

After you click **Next** , this wizard will acquire a lease to test the connection with the license server.

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



PostSharp 2.1 Configuration

License Agreement

You are entering the agreement under the terms of the **Personal License** registered to **Test**.

Licence Agreement for PostSharp

Version: May 26, 2011

IMPORTANT! READ CAREFULLY: THIS IS A LEGAL AGREEMENT. THE SOFTWARE IS LICENSED UNDER THIS AGREEMENT ('AGREEMENT').

[✓] I accept the terms of the License Agreement.

[ ] Register these settings for all accounts on this machine.
Recommended on build servers.

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.**

Important

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.

Installing the license settings in source control

It is possible to install the license settings (license key or license server URL) in source control, so that these settings are automatically applied during the build.

To install license settings in source control:

1. Create a file named *PostSharp.Custom.targets* in the directory that contains the Visual Studio project file (**.csproj* or **.vbproj*), or in a parent of this directory (up to 8 levels from the project file). Typically, you would create this file in the root directory of the source control project.

2. Add the following content to the *PostSharp.Custom.targets* file:

```
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <PropertyGroup>
    <PostSharpLicense>xxx</PostSharpLicense>
  </PropertyGroup>
</Project>
```

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

License activation and audit

Althought most software packages are protected with a license activation mechanism, we judged 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 identificable 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.

2.4. Configuring PostSharp

PostSharp accepts several configuration settings such as the version and processor architecture of the target CLR, the search path of dependencies, and whether some features are enabled. Although the default value of settings are appropriate for most situations, you may have to fine-tune some of them to cope with particular cases.

Most PostSharp configuration settings are materialized as MSBuild properties. Therefore, adjusting PostSharp settings does not differ from modifying the settings of any other build component, such as the compiler.

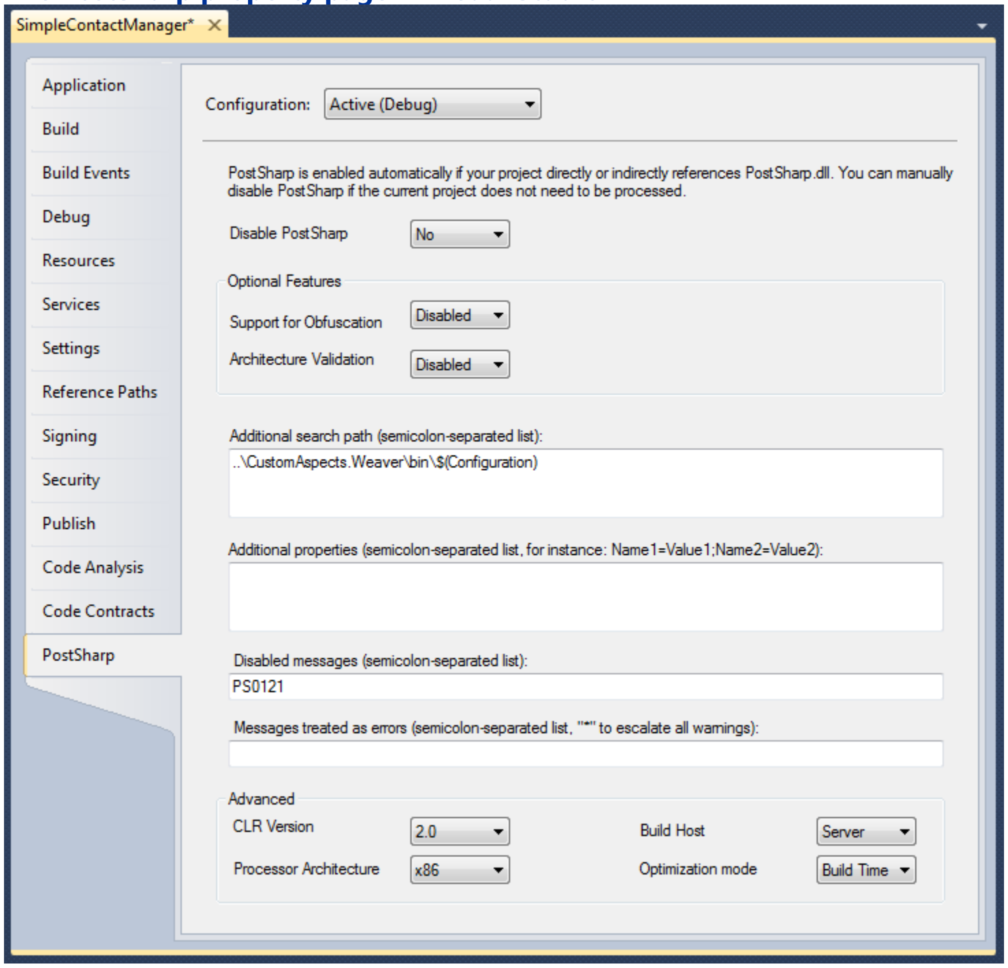
You can modify PostSharp settings by one of the following ways.

- Using the PostSharp project property page in Visual Studio at page 26
- Editing the project file with a text editor at page 26
- Configuring several projects at a time using PostSharp.Custom.targets at page 27
- Using a command-line parameter of MSBuild at page 28

Using the PostSharp project property page in Visual Studio

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

The PostSharp property page in Visual Studio



Configuration: Active (Debug)

PostSharp is enabled automatically if your project directly or indirectly references PostSharp.dll. You can manually disable PostSharp if the current project does not need to be processed.

Disable PostSharp: No

Optional Features

Support for Obfuscation: Disabled

Architecture Validation: Disabled

Additional search path (semicolon-separated list):
..\CustomAspects.Weaver\bin\$(Configuration)

Additional properties (semicolon-separated list, for instance: Name1=Value1;Name2=Value2):

Disabled messages (semicolon-separated list):
PS0121

Messages treated as errors (semicolon-separated list, "*" to escalate all warnings):

Advanced

CLR Version: 2.0

Build Host: Server

Processor Architecture: x86

Optimization mode: Build Time

Editing the project file 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:

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

   See Configurable MSBuild Properties at page 28 for the list of MSBuild properties used by PostSharp.

   > **Note**
   >
   > Since you are defining a standard MSBuild property, you are free to use all MSBuild features. For instance, you can use conditional elements to define properties specific to a given build configuration. See MSBuild configuration for details.

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**.

Configuring several projects at a time using PostSharp.Custom.targets

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

See Configurable MSBuild Properties at page 28 for the list of MSBuild properties used by PostSharp.

> **Note**
>
> Since you are defining a standard MSBuild property, you are free to use all MSBuild features. For instance, you can use conditional elements to define properties specific to a given build configuration. See MSBuild configuration for details.

Using a command-line parameter of MSBuild

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 /v:detailed /p:PostSharpAttachDebugger=true /p:PostSharpBuild=Diag /p:PostSharpTra
```

2.4.1. Configurable MSBuild Properties

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 defined properties and items that are not documented here. They are considered implementation details and can change without warning.

This topic contains the following sections.

General Properties

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

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

Property Name	Description
`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 Configurable PostSharp Properties at page 31.
`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. By default, the value of this property is chosen according to the target framework version of the project.
`PostSharpTargetProcessor`	Processor architecture of the PostSharp hosting process. Valid values are x86 and x64. By default, the value of this property is chosen according to the target platform of the project. If the target platform is AnyCPU, the architecture of the current machine will be chosen.

Property Name	Description
PostSharpHost	Kind of process hosting PostSharp. 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. However, it is the same algorithm as the one of PostSharp 1.5. The managed host is the only one that runs on Mono.
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 at page 232.
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 at page 232.
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 **PostSharp30** 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 **PostSharp30** to take action.

Property Name	Description
`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.

2.4.2. Configurable PostSharp Properties

The previous section listed properties that are understood by PostSharp but must be defined as MSBuild properties. Additionally, PostSharp has its own system of properties. A project can pass properties to PostSharp by setting the MSBuild property `PostSharpProperties`. Some system properties have already assigned a value and must not be overwritten.

How to define PostSharp properties?

Some properties are mapped to MSBuild properties and should be modified as described in Configuring PostSharp at page 25.

Other properties can be set in Visual Studio using the PostSharp project property page (text box **additional properties**). Alternatively, you can define the MSBuild property `PostSharpProperties` in your project files or PostSharp.Custom.targets at page 25. For instance, the following MSBuild project fragment defines the PostSharp property `IgnoredAssemblies`:

```
<PropertyGroup>
  <PostSharpProperties>
    $(PostSharpProperties);
    IgnoredAssemblies=ObfuscatedThirdPartyLib1,ObfuscatedThirdPartyLib2
  </PostSharpProperties>
</PropertyGroup>
```

List of properties

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

Property Name	MSBuild Property Name	Description
`Configuration`	`Configuration`	Build configuration (typically Debug or Release).
`Platform`	`Platform`	Target processor architecture (typically AnyCPU, x86 or x64).
`MSBuildProjectFullPath`	`MSBuildProjectFullPath`	Full path of the C# or VB project being built.
`IgnoredAssemblies`		Comma-separated list of assembly short names (without extension) that should be ignored by the dependency scanning algorithm. Add an assembly to this list if it is obfuscated, or contains native code, and causes PostSharp to fail.
`ReferenceDirectory`	`MSBuildProjectDirectory`	Directory with respect to which relative paths are resolved.
`SearchPath`	`PostSharpSearchPath`	Comma-separated list of directories containing reference assemblies and plug-ins.
`TargetFrameworkIdentifier`	`TargetFrameworkIdentifier`	Identifier of the target framework of the current project (i.e. the framework on which the application will run). For instance `.NETFramework` or `Silverlight`.
`TargetFrameworkVersion`	`TargetFrameworkVersion`	Version of the target framework of the current project (i.e. the framework on which the application will run). For instance `v4.0`.
`TargetFrameworkProfile`	`TargetFrameworkProfile`	Profile of the target framework of the current project (i.e. the framework on which the application will run). For instance `WindowsPhone`.

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

Accessing properties from source code

You can read the value of any PostSharp property thanks to the following piece of code:

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

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.

2.5. Using PostSharp on a Build Server

PostSharp 3 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 at page 33.

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 at page 17 for details.

See Also

2.6. Restoring packages at build time

NuGet Restore is a feature of NuGet Package Manager that restores packages from their repository during the build. This allows teams to avoid storing NuGet packages in source repository.

PostSharp is *not* fully compatible with NuGet Restore. 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.

To restore NuGet packages before build:

- Add a preliminary step before building the Visual Studio solutions or projects. This step should execute the command `NuGet.exe install *packages.config* -OutputDirectory *SolutionDirectory*\*packages*` for every *packages.config* file. where *SolutionDirectory\packages* is the directory where the NuGet packages should be installed.

Tip

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

2.7. Upgrading from PostSharp 2

This section explains how to upgrade from PostSharp 2 to PostSharp 3.

PostSharp 3 can be installed side-by-side with PostSharp 2.0 and PostSharp 2.1 on the same machine.

However, both versions of PostSharp cannot be used together in the same project. Every project can have only references to a single version of PostSharp. This applies both to direct and indirect references. The PostSharp 3 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.

Tip

Other sections of this chapter, specifically Installing PostSharp at page 14, Deploying License Keys at page 17 and Using PostSharp on a Build Server at page 32, are also useful if you need to upgrade from an earlier version of PostSharp.

Side-by-side installation

After you install PostSharp 3, you will still be able to open solutions that use PostSharp 2.

The Visual Studio extension of PostSharp 3 is backward compatible with PostSharp 2. However, both versions of the extension cannot coexist. Therefore, PostSharp 3 will attempt to uninstall the Visual Studio extension of PostSharp 2.

Upgrading solutions

Caution

Do not upgrade to PostSharp 3 in the following situations:

- The solution must build on Visual Studio 2008 or on Mono.
- The application targets .NET Compact Framework or Silverlight 3.
- The application has dependencies to third-party libraries that have references to earlier versions of PostSharp and have not been ported to PostSharp 3.
- Some aspects still use the old *PostSharp.Laos.dll* and have not been ported to the PostSharp 2 API.

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 to PostSharp 3:

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**.

Note

This procedure may be cumbersome if you have a large number of solutions. Please contact our technical support if you are in this situation.

Upgrading source code

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

- Every occurrence of the _Assembly interface has been replaced by the Assembly classes. You may have to change the signatures of some methods derived from AssemblyLevelAspect.

- Aspects that target Silverlight, Windows Phone or Windows Store must be annotated with the PSerializableAttribute custom attribute.
- PostSharp Toolkits 2.1 need to be uninstalled using NuGet. Instead, you can install PostSharp Pattern Libraries 3 from NuGet. Namespaces and some type names have been changed.

2.8. Installing PostSharp Unattended

PostSharp is composed of a user interface (a Visual Studio extension) and build components (NuGet packages). NuGet packages are usually checked into source control or retrieved from a package repository at build time (see Restoring packages at build time at page 33), so its deployment does not require additional automation. The user interface is typically installed by each user. It does not require administrative privileges.

> **Caution**
>
> The PostSharp user interface requires NuGet Package Manager 2.2, which is not installed by default with Visual Studio. Installing NuGet requires administrative privileges on the local machine.

You can install PostSharp automatically for a large number of users using a script.

To install PostSharp on a machine:

1. Ensure that NuGet 2.2 is installed. Your script will need to look for a file named *NuGet.Core.dll* under the following directories. The search should include up to 2 levels of subdirectories.

    - *C:\Program Files (x86)\Microsoft Visual Studio 10.0\Common7\IDE\Extensions* for Visual Studio 2010.
    - *C:\Program Files (x86)\Microsoft Visual Studio 11.0\Common7\IDE\Extensions* for Visual Studio 2012.

2. If NuGet 2.2 is not installed, install it with the command line `VsixInstaller.exe /q NuGet.Tools.vsix`. This command requires administrative privileges on the local machine. The current version of *NuGet.Tools.vsix* can be downloaded from Visual Studio Gallery[4]. Note that PostSharp is tested with versions that are current at the time of writing. If you need to be able to restore a working development environment several years from now, it is a good idea to archive a version of NuGet that is known to work with your version of PostSharp.

3. Execute command line `VsixInstaller.exe /q PostSharp-VERSION.vsix`. The file can be downloaded from Visual Studio Gallery[5]. Exit codes other than 0 or 1001 should be considered as errors.

4. http://visualstudiogallery.msdn.microsoft.com/27077b70-9dad-4c64-adcf-c7cf6bc9970c

4. Install the license key or the license server URL in the registry key `HKEY_CURRENT_USER\Software\SharpCrafters\PostSharp 3`, registry value `LicenseKey` (type `REG_SZ`).

This procedure can be automated by the following PowerShell 2.0 script:

```
# TODO: Set the right value for the following variables
$postsharpFile = "PostSharp-3.0.14-beta.vsix"   # Replace with the proper version number and a
$nugetFile = "NuGet.Tools.vsix"                 # Add the full path.
$license = "XXXX-XXXXXXXXXXXXXXXXXXXXXXXXX"     # Replace by your license key or license serve


$installNuget = $false

# Check NuGet in Visual Studio 2010.
if ( Test-Path "C:\Program Files (x86)\Microsoft Visual Studio 10.0\Common7\IDE\devenv.exe" )
{

    $vsixInstaller = "C:\Program Files (x86)\Microsoft Visual Studio 10.0\Common7\IDE\VsixInst

    $nugetVs10Path = dir "C:\Program Files (x86)\Microsoft Visual Studio 10.0\Common7\IDE\Exte

    if ( $nugetVs10Path -eq $null )
    {
        $installNuget = $true
    }
    else
    {
        $nugetVs10Version = [System.Diagnostics.FileVersionInfo]::GetVersionInfo($nugetVs10Pat
        Write-Host "Detected NuGet" $nugetVs10Version.FileVersion "for Visual Studio 2010."
        if ( $nugetVs10Version.FileMajorPart -lt 2 -or $nugetVs10Version.FileMinorPart -lt 1 )
        {
            $installNuget = $true;
        }
    }
}

# Check NuGet in Visual Studio 2012.
if ( Test-Path "C:\Program Files (x86)\Microsoft Visual Studio 11.0\Common7\IDE\devenv.exe" )
{

    $vsixInstaller = "C:\Program Files (x86)\Microsoft Visual Studio 11.0\Common7\IDE\VsixInst
    $nugetVs11Path = dir "C:\Program Files (x86)\Microsoft Visual Studio 11.0\Common7\IDE\Exte

    if ( $nugetVs11Path -eq $null )
    {
        $installNuget = $true
    }
    else
    {
        $nugetVs11Version = [System.Diagnostics.FileVersionInfo]::GetVersionInfo($nugetVs10Pat
        Write-Host "Detected NuGet"  $nugetVs11Version.FileVersion "for Visual Studio 2012."
        if ( $nugetVs11Version.FileMajorPart -lt 2 -or $nugetVs11Version.FileMinorPart -lt 1 )
        {
            $installNuget = $true;
```

5. http://visualstudiogallery.msdn.microsoft.com/a058d5d3-e654-43f8-a308-c3bdfdd0be4a

```
        }
    }
}


if ( -not ( Test-Path $vsixInstaller) )
{
    Write-Host "Cannot find " $vsixInstaller
    exit
}

# Install NuGet
if ( $installNuget )
{
    Write-Host "Installing NuGet"
    $process = Start-Process -FilePath  $vsixInstaller -ArgumentList @("/q", $nugetFile) -Wait
    if ( $process.ExitCode -ne 0 -and $process.ExitCode -ne 1001)
    {
        Write-Host "Error: VsixInstaller exited with code" $process.ExitCode -ForegroundColor
    }
}


# Install PostSharp
Write-Host "Installing PostSharp"
$process = Start-Process -FilePath $vsixInstaller -ArgumentList @("/q", $postsharpFile) -Wait
if ( $process.ExitCode -ne 0 -and $process.ExitCode -ne 1001)
{
    Write-Host "Error: VsixInstaller exited with code" $process.ExitCode -ForegroundColor Red
}


# Install the license key
Write-Host "Installing the license key"
$regPath = "HKCU:\Software\SharpCrafters\PostSharp 3"


if ( -not ( Test-Path $regPath ) )
{
    New-Item -Path $regPath | Out-Null
}

Set-ItemProperty -Path $regPath -Name "LicenseKey" -Value $license


Write-Host "Done"
```

2.9. Incompatibilities with Other Products

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 support for PCL	Use PostSharp 2.1 or Silverlight 4
.NET Compact Framework	No support for PCL	Use PostSharp 2.1 or Windows Phone 7
Mono	Not Supported	Compile on Windows using MSBuild
Visual Studio Express	Microsoft's licensing policy	Use Visual Studio Standard Edition or superior
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.

See Also

CHAPTER 3

Working with Ready-Made Aspects

PostSharp Pattern Libraries are sets of ready-made aspects. Some of these aspects, such as logging, are straightforward. Other aspects implement more complex design patterns such as the reader-writer-synchronized threading model.

PostSharp currently includes the following pattern libraries:

Pattern Library	Included aspects and pattern implementations
Diagnostics Pattern Library at page 41	Logging
Model Pattern Library at page 90	INotifyPropertyChanged, code contracts
Threading Pattern Library at page 55	Reader-writer-synchronized threading model, thread-unsafe threading model, actor threading model, thread dispatching.

3.1. Working with the Diagnostics Pattern Library

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.

This topic contains the following sections.

- List of available backends at page 41
- Changing the logging back-end at page 42

List of available backends

Name	NuGet Package	Description
Console	PostSharp.Patterns.Diagnostics	Logging using Console WriteLine(String)
Trace	PostSharp.Patterns.Diagnostics	Logging using Trace
Log4Net	PostSharp.Patterns.Diagnostics.Log4Net	

Name	NuGet Package	Description
NLog	PostSharp.Patterns.Diagnostics.NLog	
EnterpriseLibrary	PostSharp.Patterns.Diagnostics.EnterpriseLibrary	

Changing the logging back-end

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.
3. If the new NuGet package contains several implementations, set the `LoggingBackend` property in the PostSharp project file (*MyProject.psproj*) to the right value for the chosen logging backend.

See Also

Reference

LogAttribute

LogExceptionAttribute

3.1.1. 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.

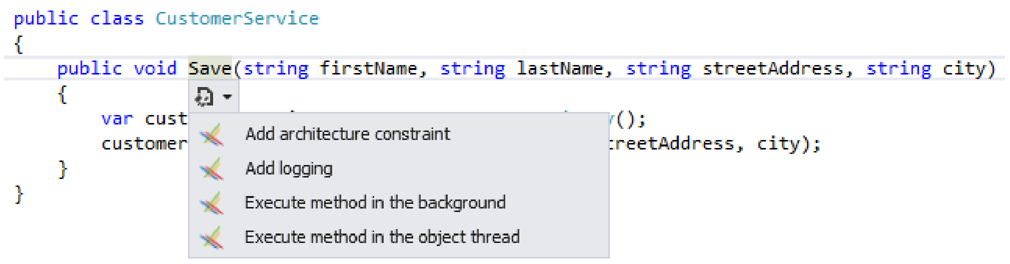
PostSharp offers a solution to all of these problems. The logging pattern library allows you to configure where logging should be performed and the pattern library takes over the task of keeping your log entries in sync as you add, remove and refactor your codebase. Let's take a look at how you can add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

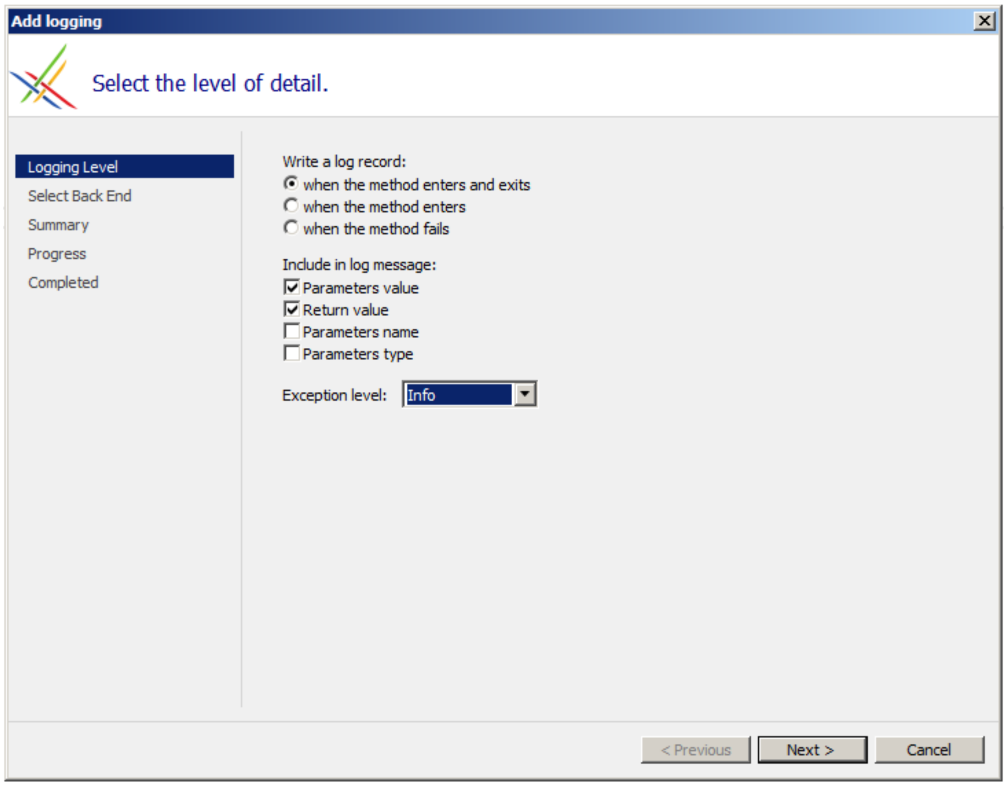
2. Put the caret on the `Save` method name and expand the Smart Tag. From the list select "Add loggging".



```
public class CustomerService
{
    public void Save(string firstName, string lastName, string streetAddress, string city)
    {
        var cust
        customer
    }
}
```

Add architecture constraint
Add logging
Execute method in the background
Execute method in the object thread

3. The first option that you need to select is the Logging Level. For this example we will take the default provided: it logs the method enters and exits, and include parameter values.



Add logging

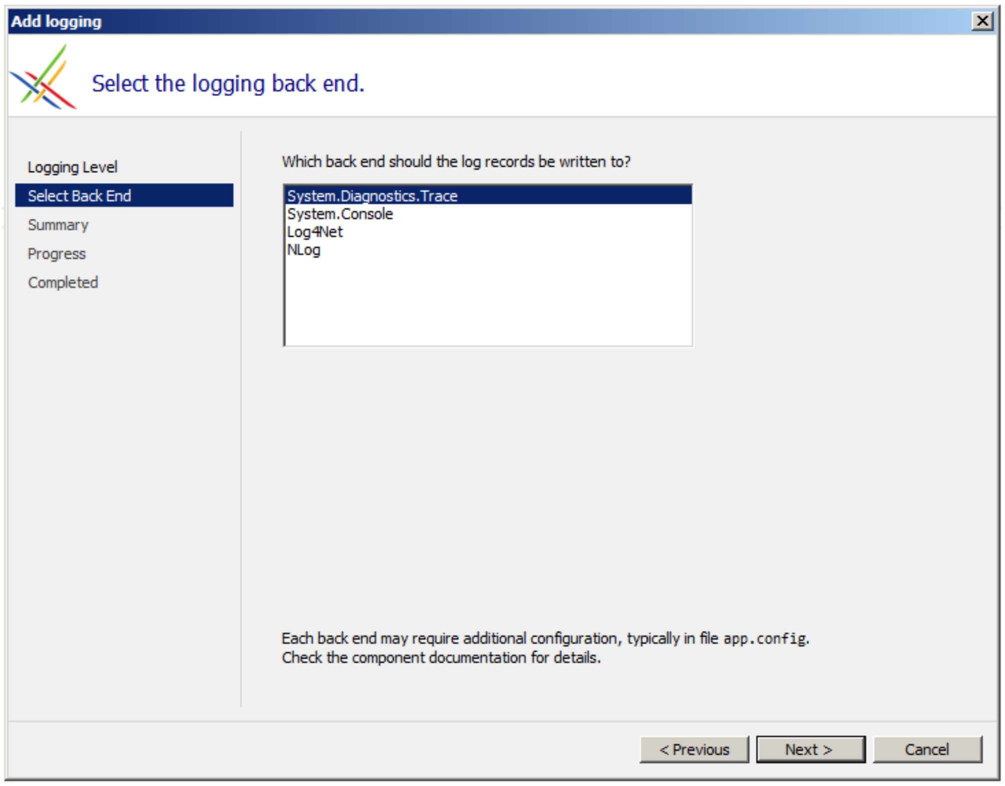
Select the level of detail.

Write a log record:
- (●) when the method enters and exits
- ( ) when the method enters
- ( ) when the method fails

Include in log message:
- [✓] Parameters value
- [✓] Return value
- [ ] Parameters name
- [ ] Parameters type

Exception level: Info

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

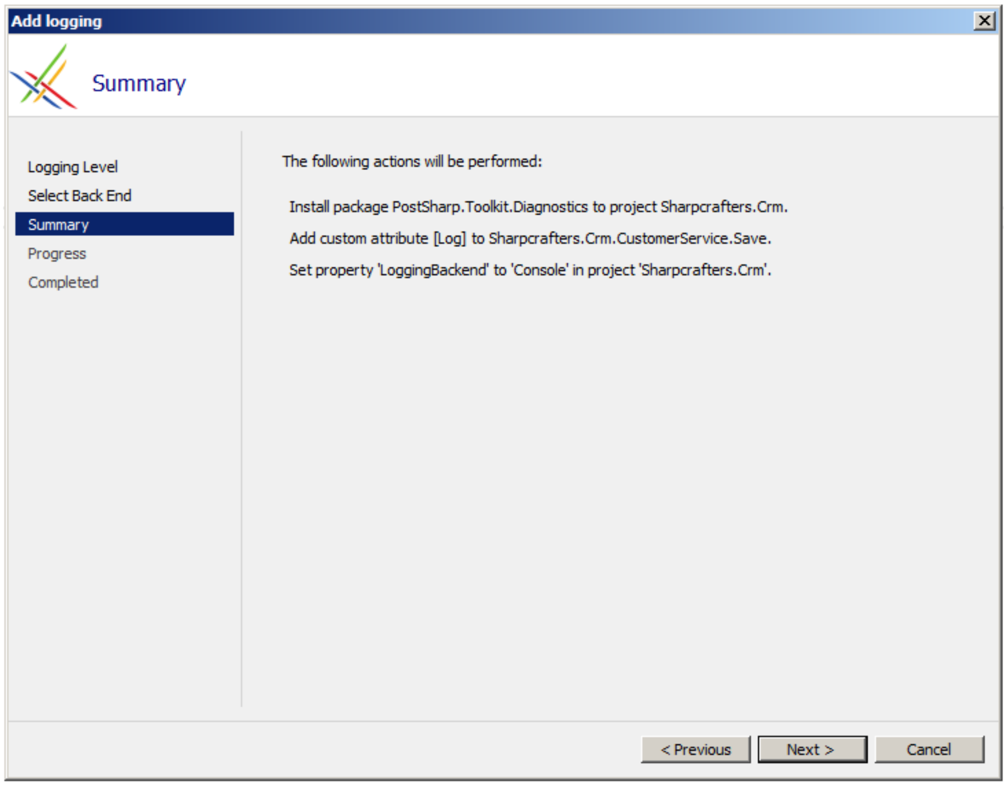


Which back end should the log records be written to?

System.Diagnostics.Trace
System.Console
Log4Net
NLog

Each back end may require additional configuration, typically in file `app.config`.
Check the component documentation for details.

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**.



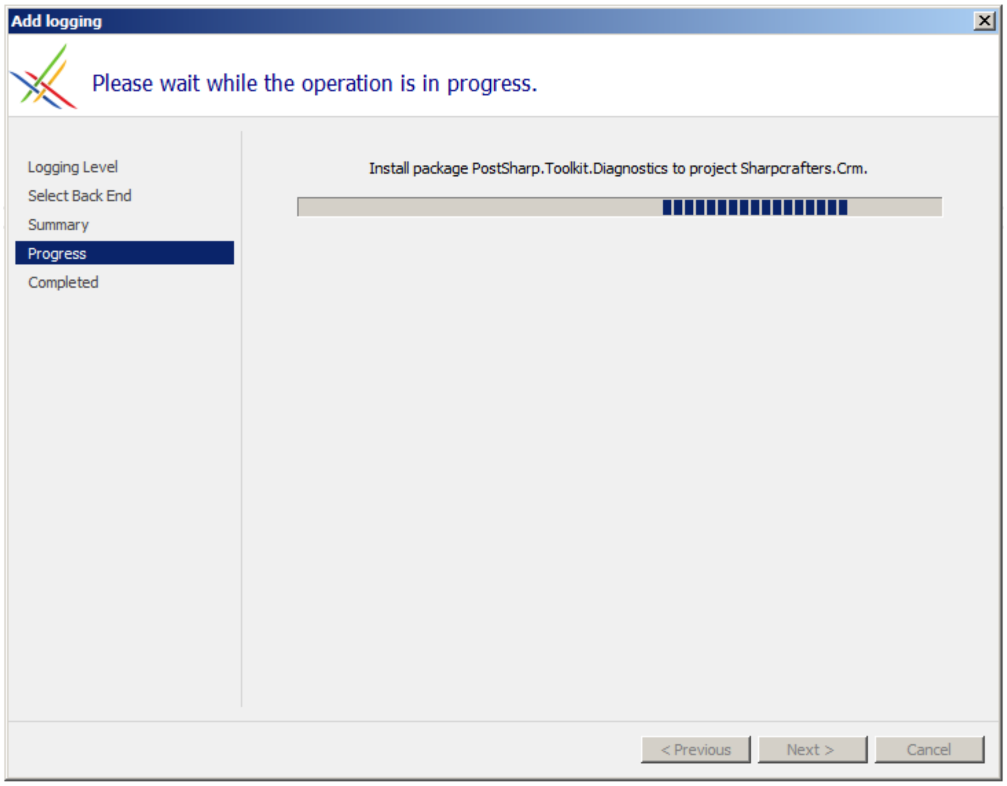
The following actions will be performed:

Install package PostSharp.Toolkit.Diagnostics to project Sharpcrafters.Crm.

Add custom attribute [Log] to Sharpcrafters.Crm.CustomerService.Save.

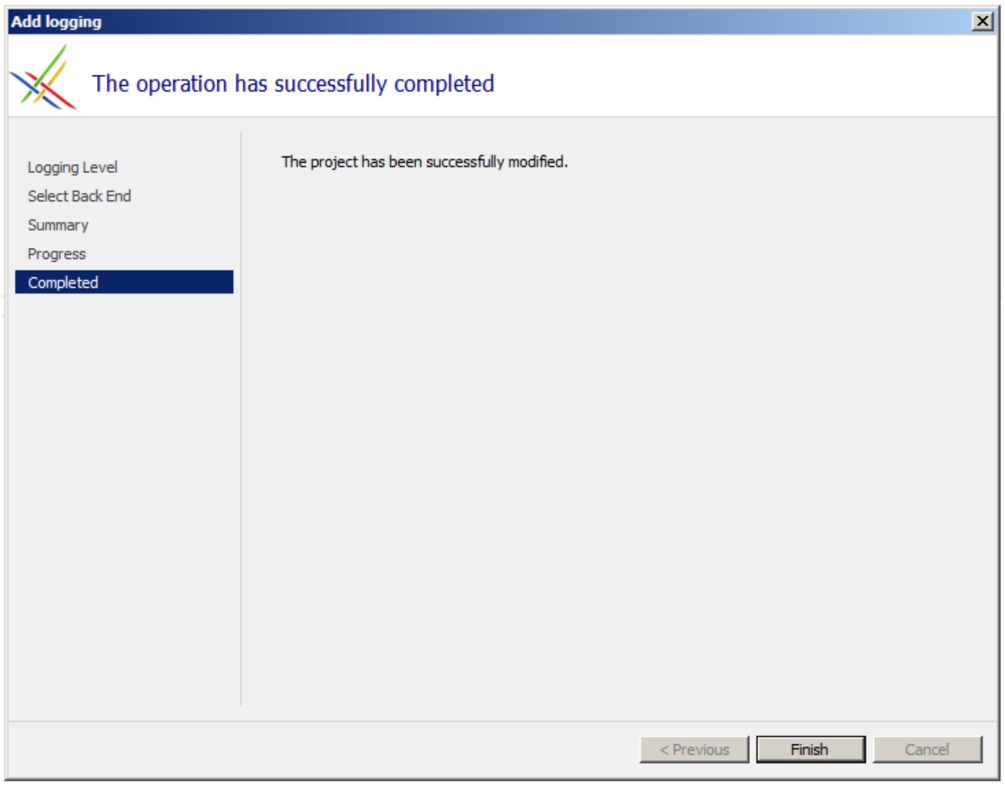
Set property 'LoggingBackend' to 'Console' in project 'Sharpcrafters.Crm'.

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.


Please wait while the operation is in progress.

Install package PostSharp.Toolkit.Diagnostics to project Sharpcrafters.Crm.

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.



The project has been successfully modified.

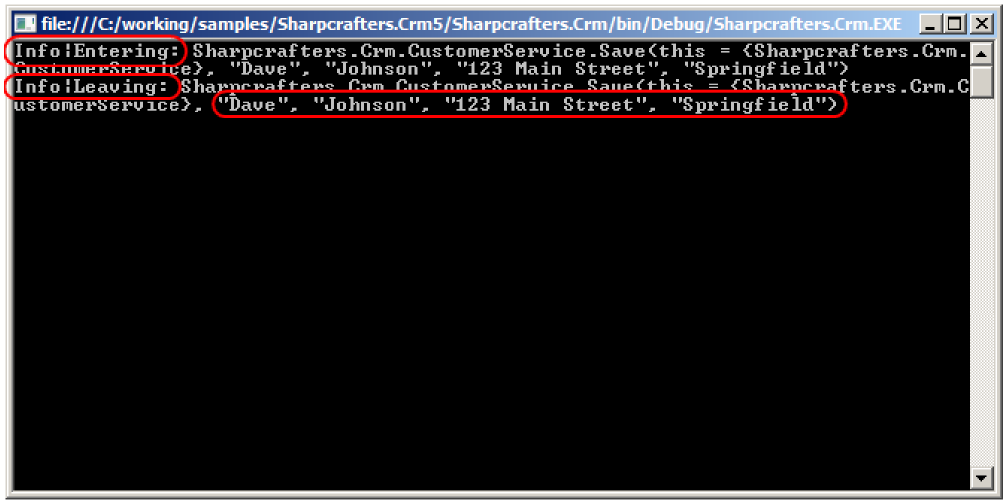
8. You'll notice that the code you added the logging to has changed slightly. PostSharp has added a LogAttribute attribute to the method. Since we chose the default logging profile, there is no argument to the LogAttribute attribute.

```
[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 at page 116.

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.



file:///C:/working/samples/Sharpcrafters.Crm5/Sharpcrafters.Crm/bin/Debug/Sharpcrafters.Crm.EXE

```
Info|Entering: Sharpcrafters.Crm.CustomerService.Save(this = {Sharpcrafters.Crm.
CustomerService}, "Dave", "Johnson", "123 Main Street", "Springfield")
Info|Leaving: Sharpcrafters.Crm.CustomerService.Save(this = {Sharpcrafters.Crm.C
ustomerService}, "Dave", "Johnson", "123 Main Street", "Springfield")
```

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 at page 116, adding logging to your codebase and maintaining it becomes a very easy task.

See Also

Reference

LogAttribute

3.1.2. 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.

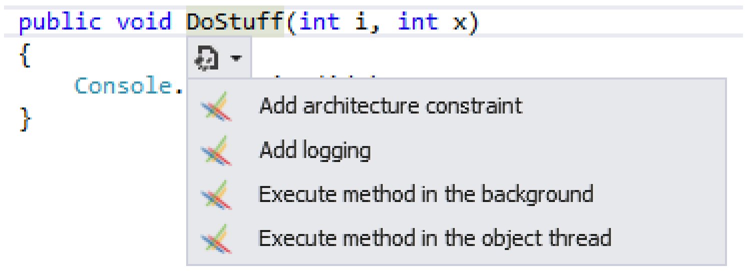
PostSharp offers a solution to all of these problems. The logging pattern library allows you to configure where logging should be performed and the pattern library takes over the task of keeping your log entries in sync as you add, remove and refactor your codebase. Let's take a look at how you can add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

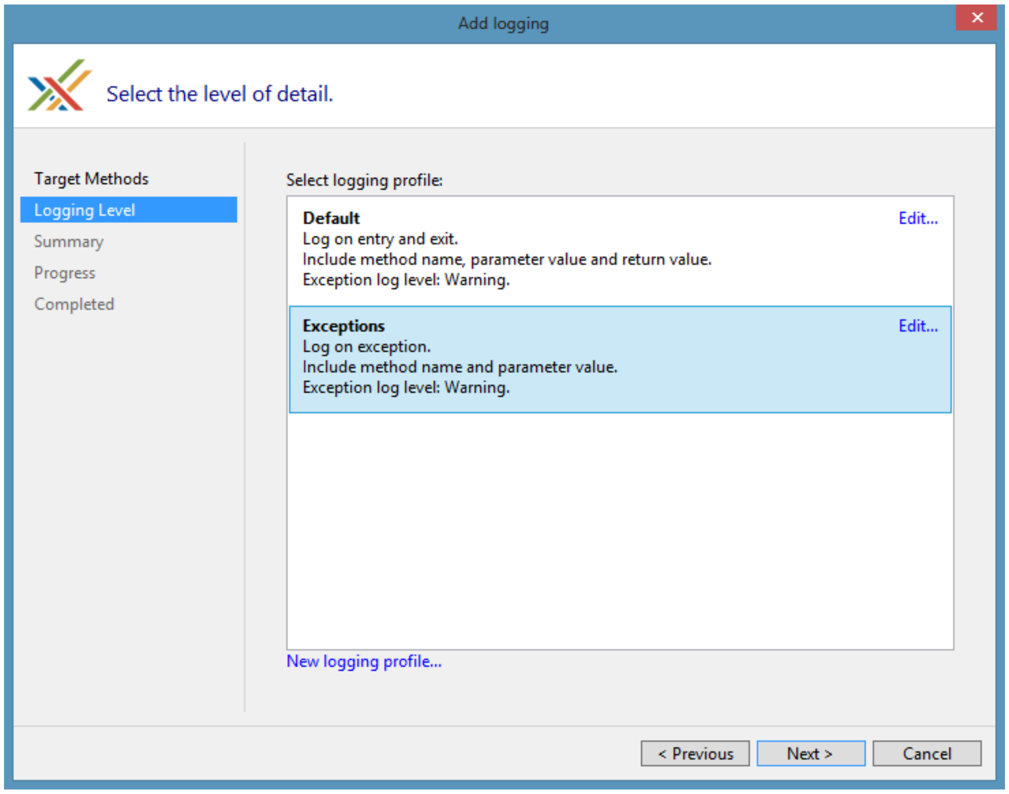
1. Let's add logging to our `DoStuff` method.

```
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 loggging".


```
public void DoStuff(int i, int x)
{
    Console.
}
```
Add architecture constraint
Add logging
Execute method in the background
Execute method in the object thread

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



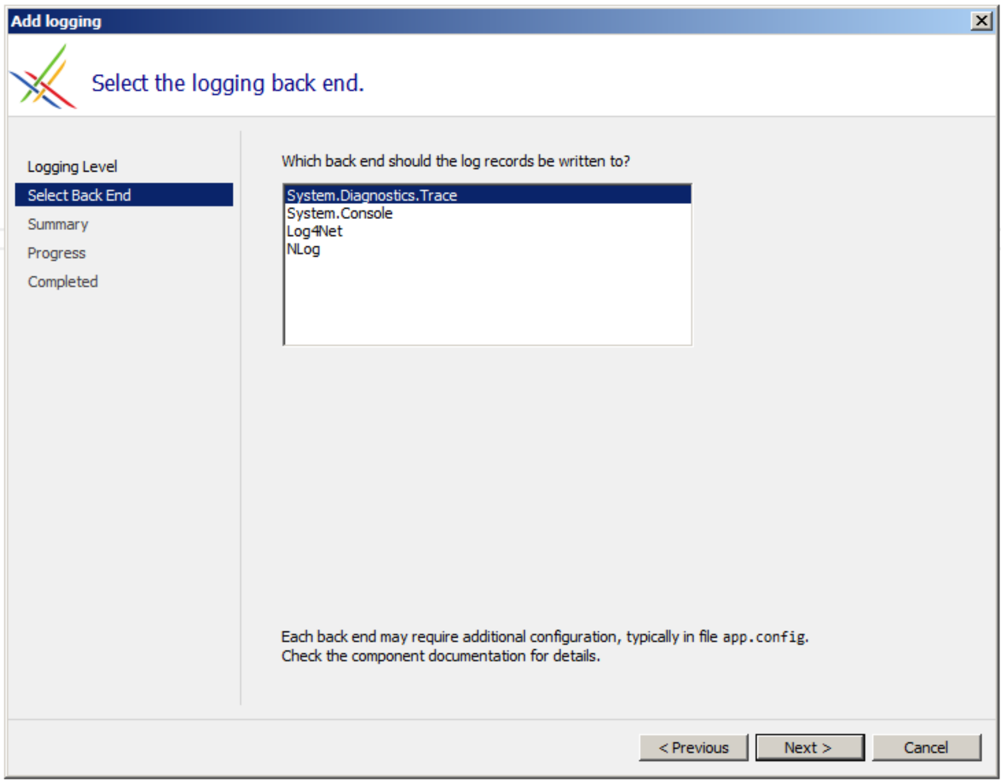
Select the level of detail.

Select logging profile:

Default
Log on entry and exit.
Include method name, parameter value and return value.
Exception log level: Warning.

Exceptions
Log on exception.
Include method name and parameter value.
Exception log level: Warning.

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

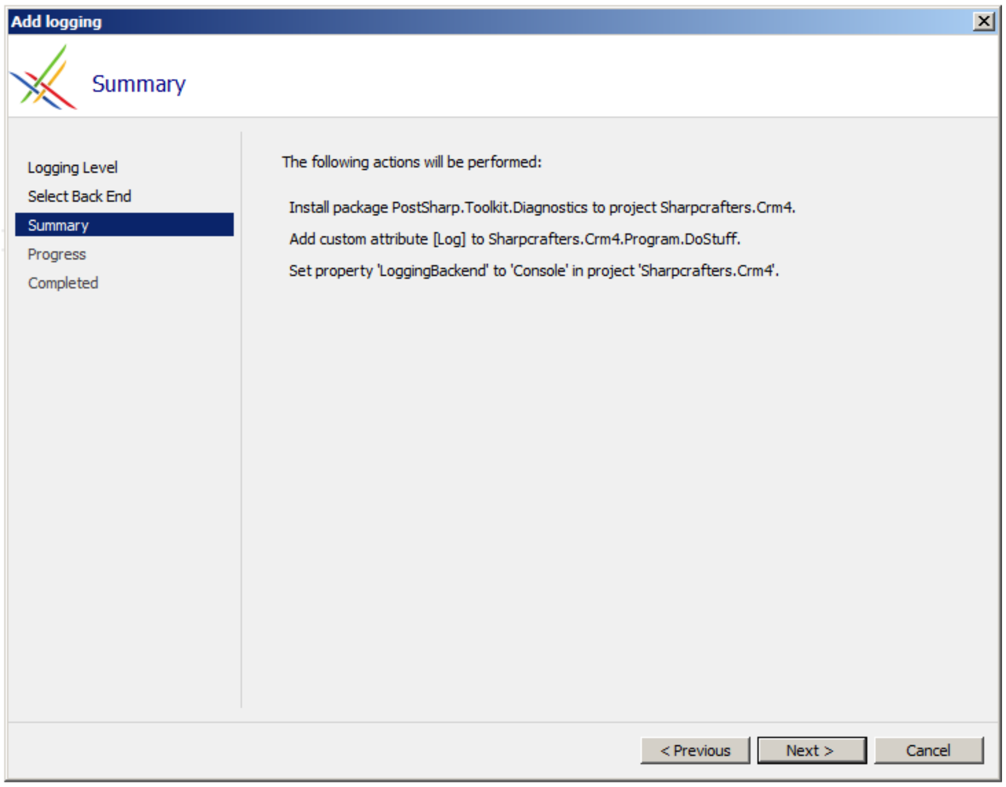


Which back end should the log records be written to?

System.Diagnostics.Trace
System.Console
Log4Net
NLog

Each back end may require additional configuration, typically in file `app.config`.
Check the component documentation for details.

5. The summary page gives you the opporunity 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**.



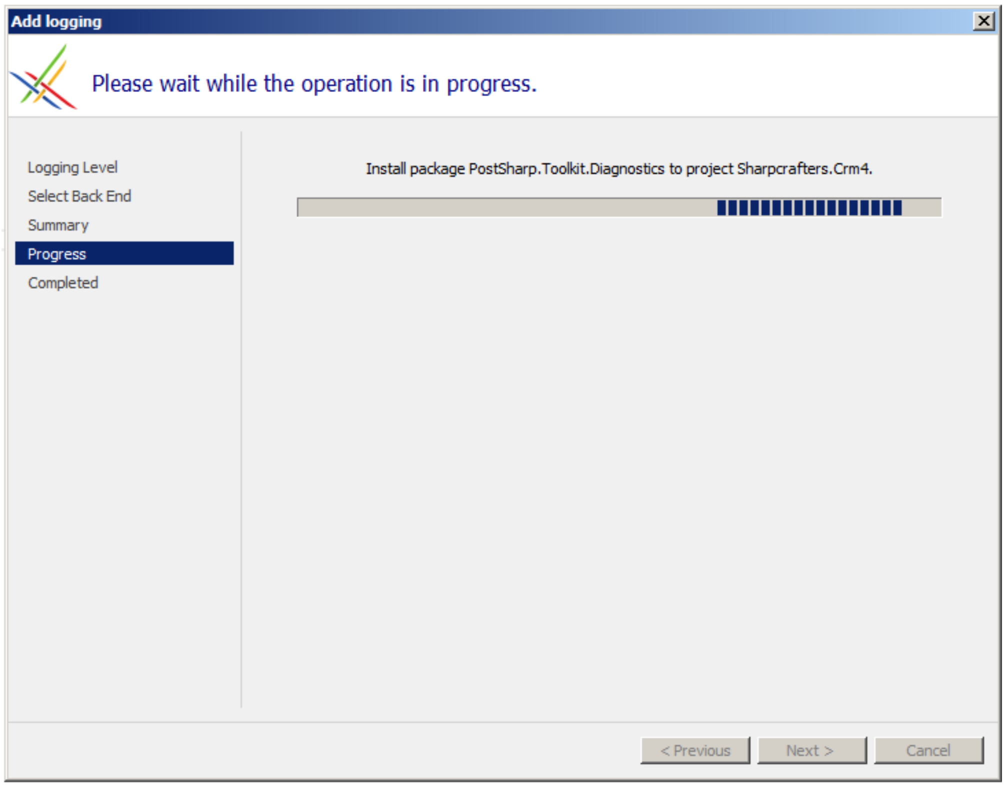
The following actions will be performed:

Install package PostSharp.Toolkit.Diagnostics to project Sharpcrafters.Crm4.

Add custom attribute [Log] to Sharpcrafters.Crm4.Program.DoStuff.

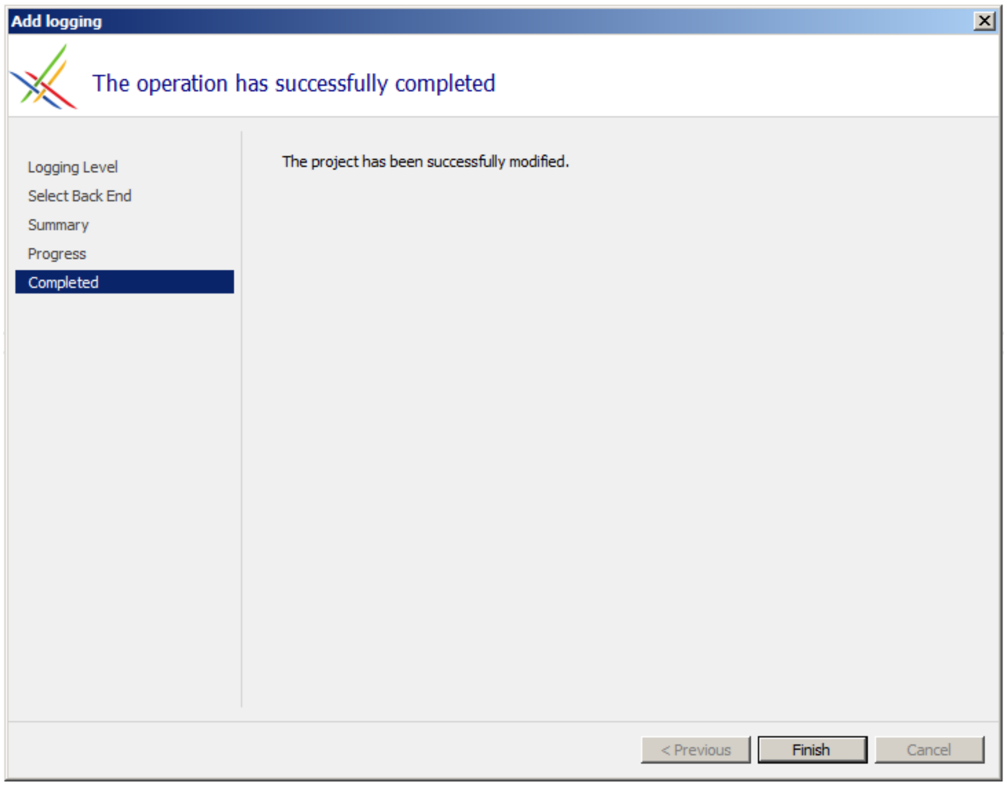
Set property 'LoggingBackend' to 'Console' in project 'Sharpcrafters.Crm4'.

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.



Install package PostSharp.Toolkit.Diagnostics to project Sharpcrafters.Crm4.

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.



The operation has successfully completed

The project has been successfully modified.

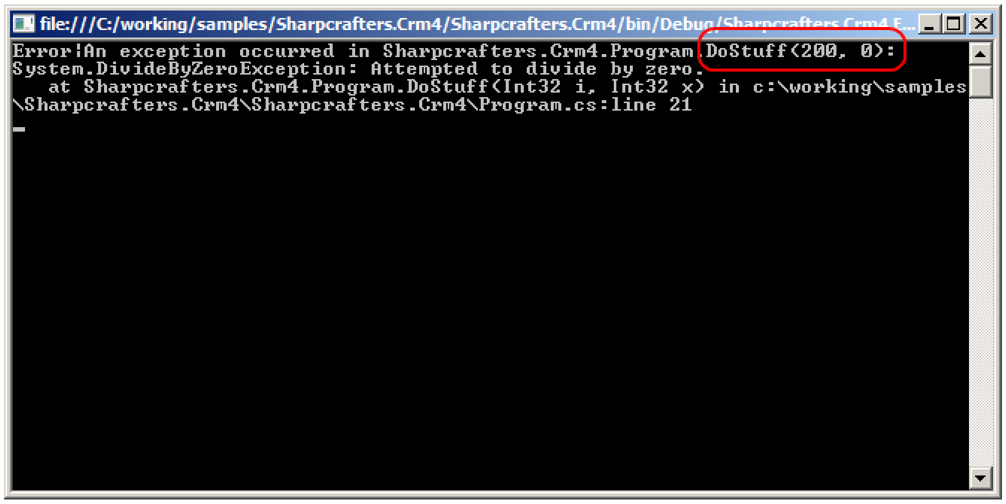
8. You'll notice that the code you added the logging to has changed slightly. PostSharp has added a LogAttribute attribute to the method and configured it based on the selections you made in the wizard. For this example you can see that the OnExceptionLevel was set to Error. This accounts for logging when the method fails. Both OnSuccessLevel and OnEntryLevel were set to None as they don't play any part in logging during failures.

```
[Log(OnExceptionLevel = LogLevel.Error, OnSuccessLevel = LogLevel.None, OnEntryLevel =
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 at page 114.

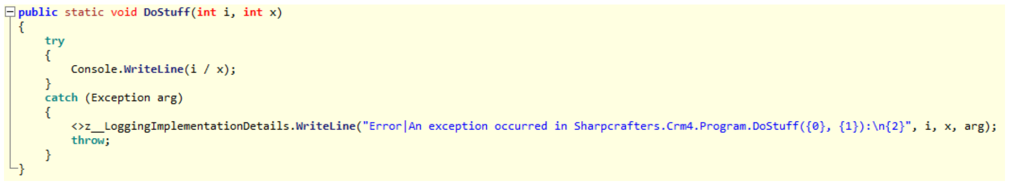
9. If you were to run this method from a console application and pass in a value of zero for the second parameter it would generate a `DivideByZeroException`. The trace logging that you added would output a log message plus the exception's stack trace to the console. Note that the parameter values are automatically included in the log message.



file:///C:/working/samples/Sharpcrafters.Crm4/Sharpcrafters.Crm4/bin/Debug/Sharpcrafters.Crm4.E...

```
Error|An exception occurred in Sharpcrafters.Crm4.Program.DoStuff(200, 0):
System.DivideByZeroException: Attempted to divide by zero.
   at Sharpcrafters.Crm4.Program.DoStuff(Int32 i, Int32 x) in c:\working\samples
\Sharpcrafters.Crm4\Sharpcrafters.Crm4\Program.cs:line 21
```

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 at page 114), adding logging to your codebase and maintaining it becomes a very easy task.

See Also

Reference

LogAttribute

OnExceptionLevel

OnSuccessLevel

OnEntryLevel

Error

3.2. Working with the Threading Pattern Library

The Threading Pattern Library helps building multithreaded applications with fewer lines of code and fewer defects. The library implements locking models (thread unsafe, reader-writer synchronized, actor), thread synchronization aspects (to background thread, to UI thread), and a deadlock detection facility.

It provides the following features:

- **Threading Models.** A threading model is a design pattern that gives guarantees that your code executes safely on a multithreaded computer. Three models are implemented: Thread Exclusive (thread unsafe), Reader-Writer Synchronized, and Actor. See Working with Threading Models at page 55 for details.
- **Thread Dispatching.** Custom attributes DispatchedAttribute and BackgroundAttribute cause the execution of a method to be dispatched to the UI thread or to a background thread, respectively. For details, read Dispatching a Method to the UI Thread at page 79 and Dispatching a Method to Background at page 86.
- **Deadlock Detection.** Detects deadlocks at runtime and throws an exception instead of allowing your application to freeze. For details, see DeadlockDetectionPolicy.

3.2.1. Working with Threading Models

A threading model is a design pattern that gives guarantees that your code executes safely on a multithreaded computer. Threading models both define coding rules (for instance: all fields must be private) and add new behaviors to existing code (for instance: acquiring a lock before method execution). Coding rules are typically enforced at build time or at run time; violations result in build-time errors or run-time exceptions. Threading models may also require the use of custom attributes in source code, for instance to indicate that a method requires read access to the object.

Tip

We recommend 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 multithreading is addressed. Compared to working directly with locks and other low-level threading primitives, using threading models has the following benefits:

- Threading models are **named solutions** to a recurring problem. Threading models are specific types of design patterns, and have the same benefits. When team members discuss the multithreaded behavior of a class, they just need to know which threading model this class uses. They don't need to know the very details of its implementation. Since the human short-term memory seems to be limited to 5-9 elements, it is important to think in terms of larger conceptual blocks whenever we can.
- Much of the code required to implement the threading model can be **automatically generated**, which decreases the number of lines of code, and therefore the number of defects. It also reduces development and maintenance costs.
- Your source code can be **automatically verified** against the selected threading model, both at build time and at run time. This makes the discovery of defect much more deterministic. Without verifications, threading defects usually show up randomly and provoke data structure corruption instead of immediate exceptions. Run-time verification would be too labour-intensive to implement without compiler support, so would be most likely ommitted.

PostSharp Threading Library provides an implementation for the following threading models:

Threading Model	Aspect Type	Description
Thread Unsafe	ThreadUnsafeAttribute	These objects may never be accessed concurrently by several threads. For details, see Ensuring Thread-Unsafe Objects are Not Shared at page 57.
Reader-Writer Synchronized	ReaderWriterSynchronizedAttribute	These objects that can be accessed concurrently by several threads. Every object is synchronized by a lock (a single lock can be shared by several objects). Public methods of this object must specify which kind of access they require (read or write, typically). Readers can run concurrently, but writers are exclusive. For details, see Using the Reader/Writer Synchronized Object Model at page 64.
Actor	Actor	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 `async` methods. For details, see Using the Actor Threading Model at page 73.

Optional section title

Add one or more sections with content

3.2.1.1. Ensuring Thread-Unsafe Objects are Not Shared

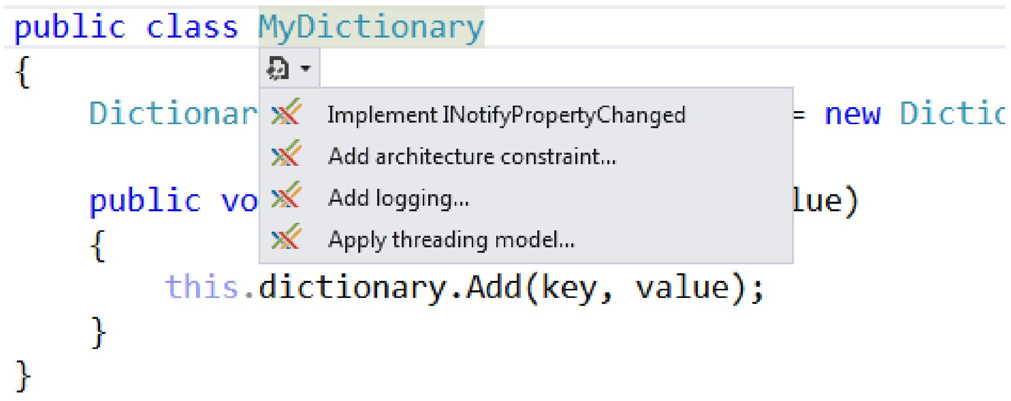
When you are dealing with multi-threaded code you will run into situations where some objects are not safe for concurrent use by several threads. Although these objects should theoretically not be accessed concurrently, it is very hard to proof that it never happens. And when it does happen, thread-unsafe 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

To mark an object as thread-unsafe:

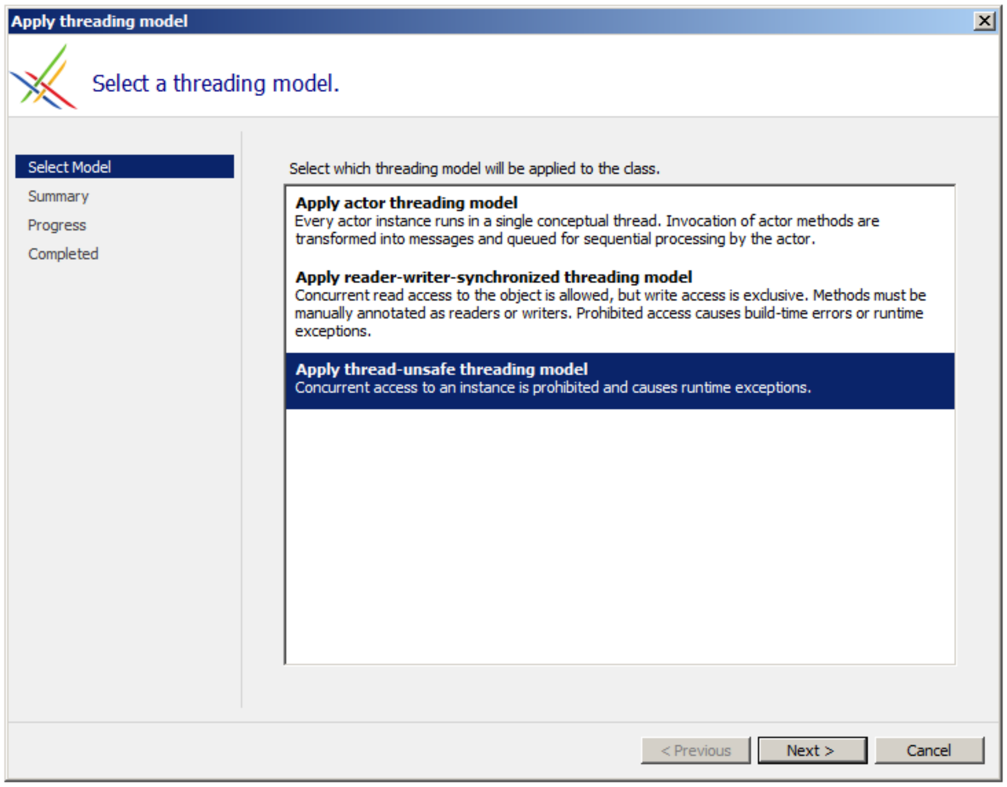
1. When you don't want multiple threads to access a single instance of a given class you will want to configure InstanceLevelAspect thread safety. To do this, select "Apply threading model" from the smart tag on the class.



```
public class MyDictionary
{
    Dictionar                                    = new Dictic

    public vo                                    lue)
    {
        this.dictionary.Add(key, value);
    }
}
```

Implement INotifyPropertyChanged
Add architecture constraint...
Add logging...
Apply threading model...

2. There are three different threading options provided by PostSharp. To restrict concurrent access you will choose the "Apply thread-unsafe threading model" option.



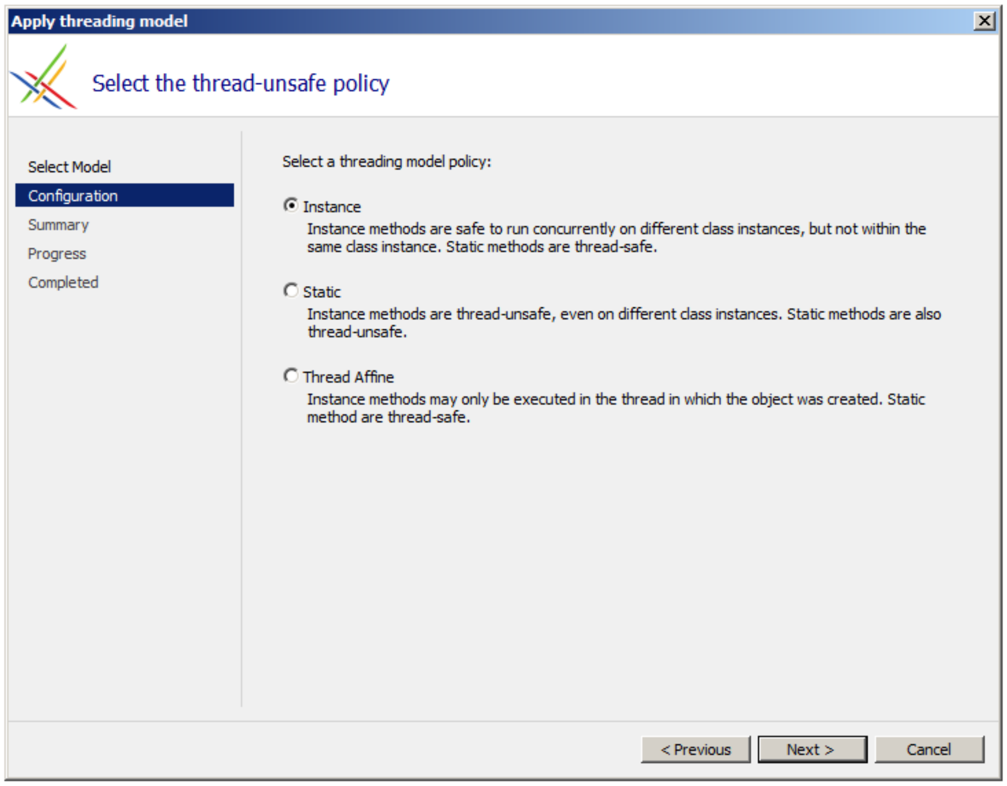
Select which threading model will be applied to the class.

Apply actor threading model
Every actor instance runs in a single conceptual thread. Invocation of actor methods are transformed into messages and queued for sequential processing by the actor.

Apply reader-writer-synchronized threading model
Concurrent read access to the object is allowed, but write access is exclusive. Methods must be manually annotated as readers or writers. Prohibited access causes build-time errors or runtime exceptions.

Apply thread-unsafe threading model
Concurrent access to an instance is prohibited and causes runtime exceptions.

3. At this point you are prompted to select the threading model policy that is needed by the target class. For this example you have determined that you do not want to have multiple threads concurrently accessing the same class instance. This is the "Instance" policy.



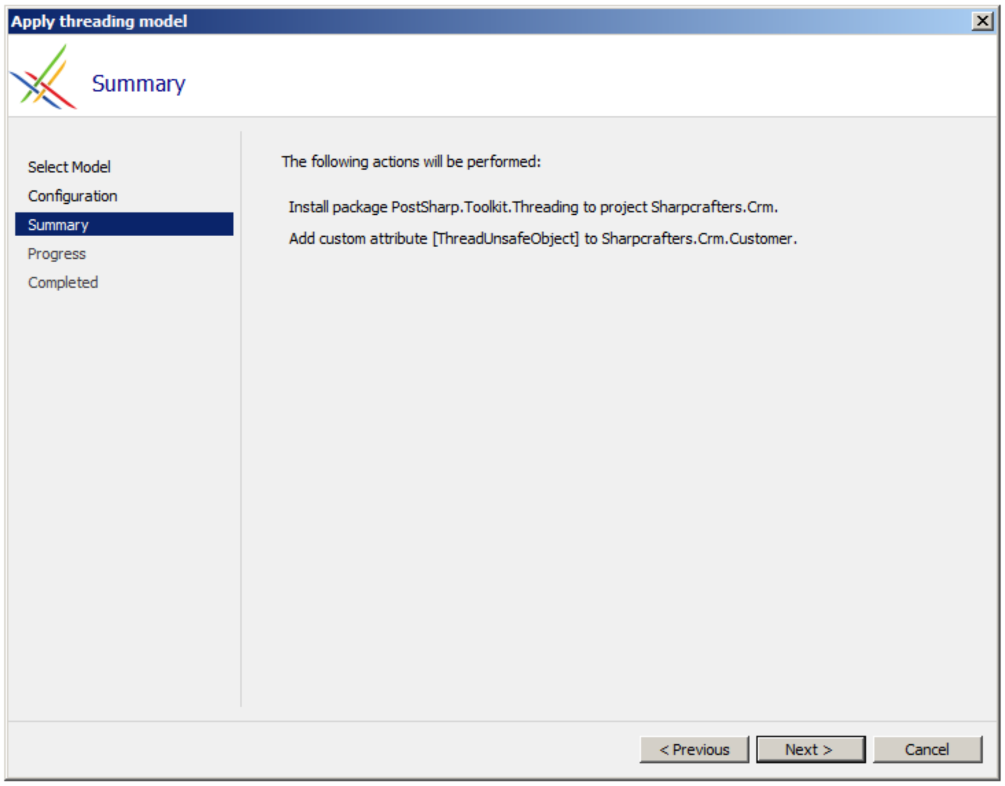
Select a threading model policy:

[✓ Instance]
Instance methods are safe to run concurrently on different class instances, but not within the same class instance. Static methods are thread-safe.

Static
Instance methods are thread-unsafe, even on different class instances. Static methods are also thread-unsafe.

Thread Affine
Instance methods may only be executed in the thread in which the object was created. Static method are thread-safe.

4. You will be prompted with a summary of the changes that will be made based on the configuration you selected in the wizard.

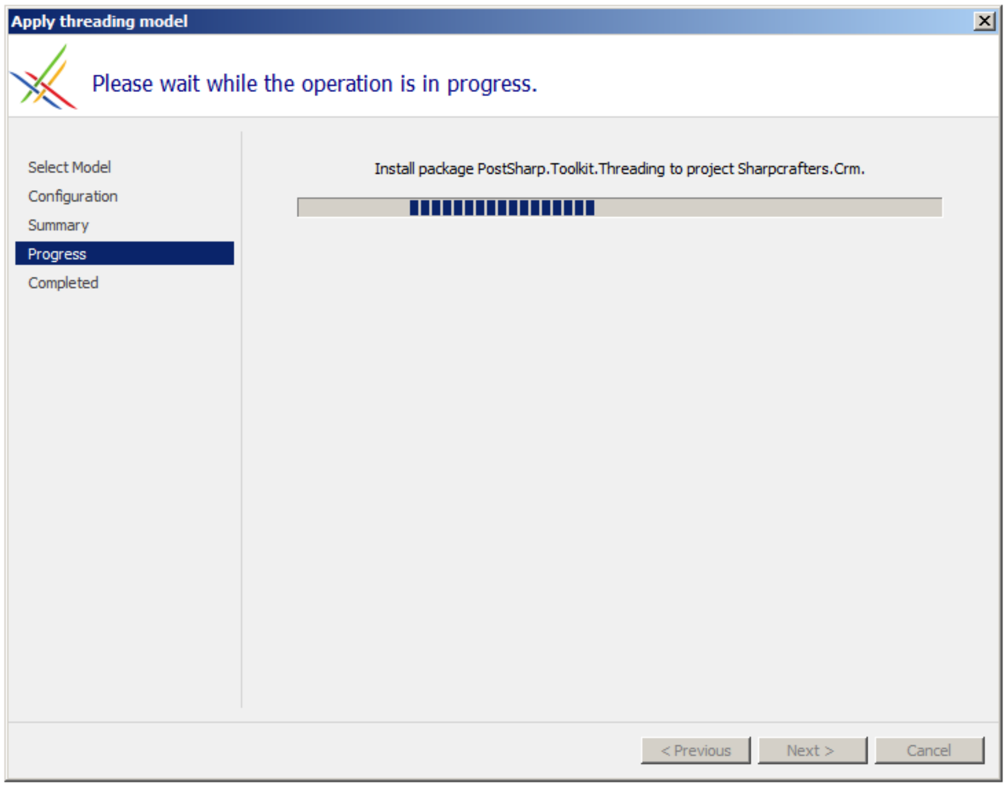


The following actions will be performed:

Install package PostSharp.Toolkit.Threading to project Sharpcrafters.Crm.

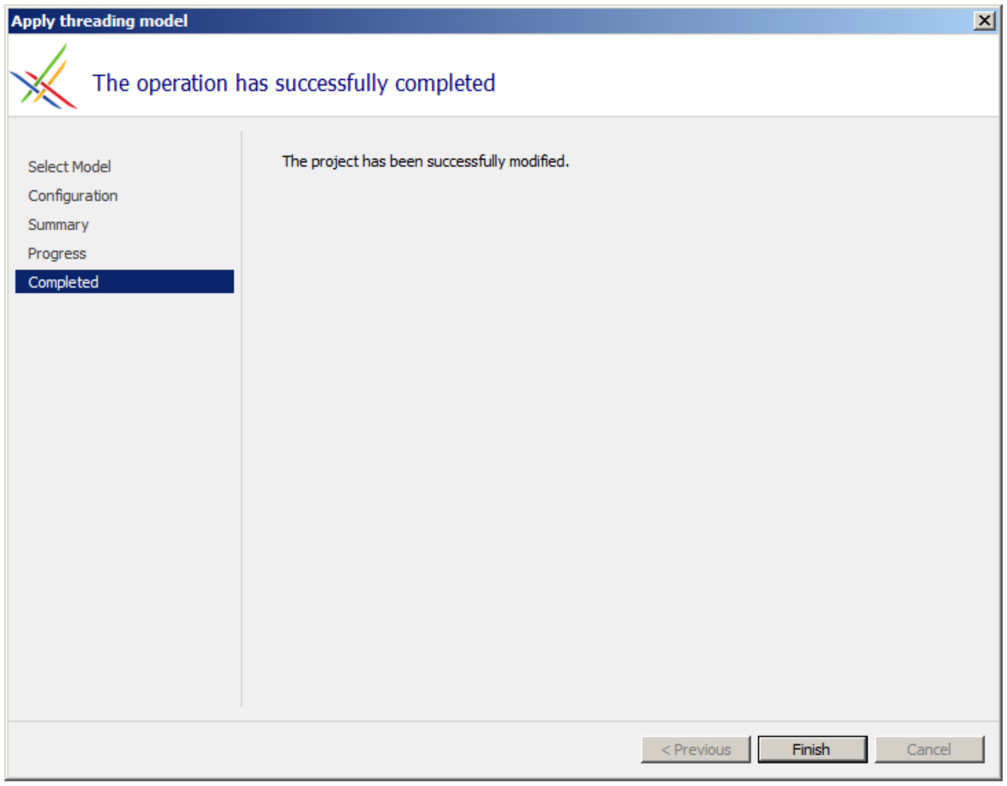
Add custom attribute [ThreadUnsafeObject] to Sharpcrafters.Crm.Customer.

5. PostSharp will download the Threading Pattern Library and add it to your project if that hasn't been done yet.



Install package PostSharp.Toolkit.Threading to project Sharpcrafters.Crm.

6. Once the process has completed successfully you'll be presented with the final page of the wizard.



The operation has successfully completed

The project has been successfully modified.

7. You'll notice that only one change was made to your codebase. The [ThreadUnsafeAttribute] attribute was added to the class you were targeting.

```
[ThreadUnsafe]
public class MyDictionary
{
    Dictionary<string,string> dictionary = new Dictionary<string,string>();

    public void Add(string key, string value)
    {
        this.dictionary.Add(key, value);
    }
}
```

Now when your application executes no two threads will be able to access a single instance of the `MyDictionary` 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 `Dictionary` would become corrupted, because this data structure is not thread safe.

Waiving verification

There may be situations where you don't need to verify thread safety on a specific method. PostSharp offers you the opportunity to waive verfication on specific methods by adding the [ExplicitlySynchronizedAttribute] attribute.

```
[ExplicitlySynchronized]
public void Add(string key, string value)
{
    lock ( this.dictionary )
    {
        this.dictionary.Add(key, value);
    }
}
```

The compiler won't generate code for methods annotated with [ExplicitlySynchronizedAttribute].

Resolving build-time errors

The thread-unsafe threading model puts a number of constraints on source code. These constraints allow PostSharp to generate high-performance runtime validation by adding verification code to public and internal methods only. These constraints are the following:

1. All fields must be private or protected.
2. Unless the threading policy is static:
   a. Static method cannot access fields.
   b. Static method cannot invoke private methods.

If your code does not respect these rules, you will see compiler errors. We recommend you first try to solve these errors by restructuring your code. If this is not possible, you can apply the [ExplicitlySynchronizedAttribute] attribute on methods or fields.

With [ExplicitlySynchronizedAttribute], the compiler won't generate errors when it detects code that it cannot guarantee to be correct. In other words, this attribute means that you, the developer, take responsibility from the compiler.

Adding verification to private and protected methods

By default, [ThreadUnsafeAttribute] only adds verification to public and internal methods. This is enough if your code respects the rule set by the thread-unsafe threading model. However, if you work around some rules using [ExplicitlySynchronizedAttribute], you may need to add verification to private and protected methods. This can be done by adding the [VerifyCallingThreadAttribute] attribute to methods that need to be verified.

See Also

Reference

InstanceLevelAspect

ThreadUnsafeAttribute

ExplicitlySynchronizedAttribute

VerifyCallingThreadAttribute

3.2.1.2. Using the Reader/Writer Synchronized Object 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
{
    public void Set(int amount, int discount)
    {
        if (amount < discount)
            throw new InvalidOperationException();

        this.Amount = amount;
        this.Discount = discount;
    }

    int Amount { get; private set; }
    int Discount { get; private set; }

    public int AmountAfterDiscount
    {
        get { return this.Amount - this.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.

- Synchronizing access to multiple instances
- Executing long-running write methods

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 reads than writes, and that concurrent reads are always safe. Reader-writer locks ensure that no other thread is accessing the object when it is being written. Reader-writer locks are normally implemented by the .NET classes ReaderWriterLock or ReaderWriterLockSlim. The following example shows how ReaderWriterLockSlim would be used to control reads and writes in the `Order` class:

```
class Order
{
    private ReaderWriterLockSlim cacheLock = new ReaderWriterLockSlim();

    public void Set(decimal amount, decimal discount)
    {

        if (amount < discount)
        {
            throw new InvalidOperationException();
        }

        cacheLock.EnterWriteLock();
        this.Amount = amount;
        this.Discount = discount;
        cacheLock.ExitWriteLock();
    }

    public decimal Amount { get; private set; }
    public decimal Discount { get; private set; }

    public decimal AmountAfterDiscount
    {
        get
        {
            cacheLock.EnterReadLock();
            decimal result = this.Amount - this.Discount;
            cacheLock.ExitReadLock();
            return result;
        }
    }
}
```

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 [ReaderLockAttribute] and [WriterLock-Attribute] 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.

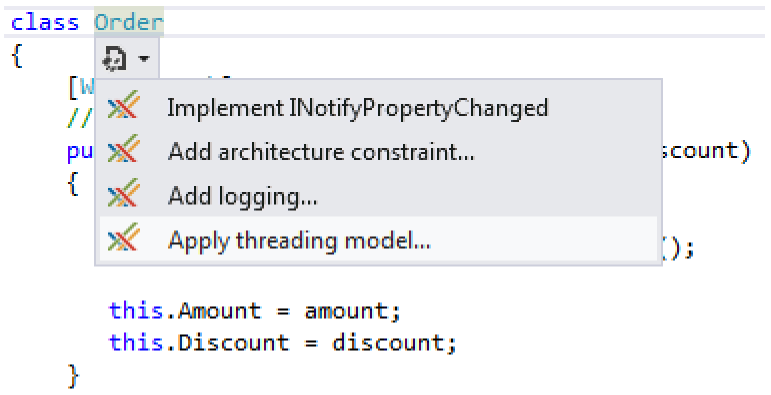
There are two ways to add the reader-writer-synchronized pattern to your class:

1. using the user interface
2. manually

To apply ReaderWriterSynchronizedAttribute using the user interface:

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

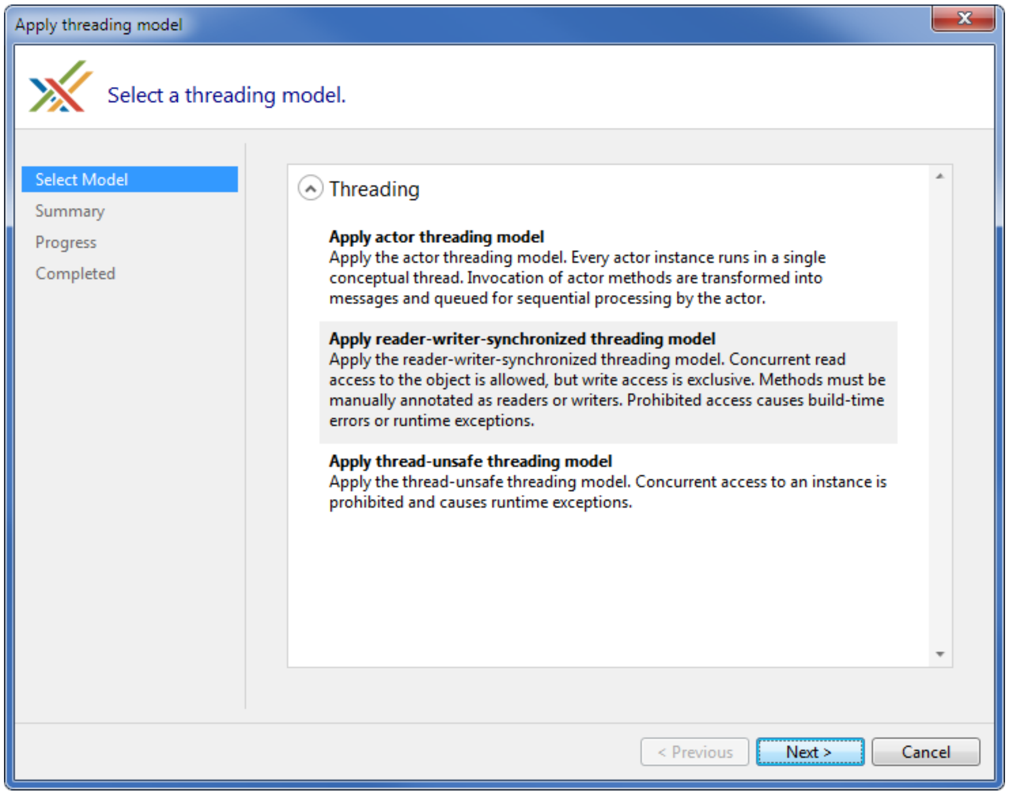
2. Click the smart tag dropdown and click "Apply threading model":



```
class Order
{
    [W
    //
    pu                                                       count)
    {
                                                             );

        this.Amount = amount;
        this.Discount = discount;
    }
```

Implement INotifyPropertyChanged
Add architecture constraint...
Add logging...
Apply threading model...

3. Select "Apply reader-writer-synchronized threading model" and click **Next**:



Apply actor threading model
Apply the actor threading model. Every actor instance runs in a single conceptual thread. Invocation of actor methods are transformed into messages and queued for sequential processing by the actor.

Apply reader-writer-synchronized threading model
Apply the reader-writer-synchronized threading model. Concurrent read access to the object is allowed, but write access is exclusive. Methods must be manually annotated as readers or writers. Prohibited access causes build-time errors or runtime exceptions.

Apply thread-unsafe threading model
Apply the thread-unsafe threading model. Concurrent access to an instance is prohibited and causes runtime exceptions.

4. Click **Next** on the summary screen and the click on **Finished** to complete the process. The ReaderWriterSynchronizedAttribute 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 [ReaderLockAttribute] or [WriterLockAttribute] to the class.

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

```
[ReaderWriterSynchronized]
class Order
{
    [WriterLock]
    public void Set(decimal amount, decimal discount)
    {
        if (amount < discount)
            throw new InvalidOperationException();

        this.Amount = amount;
        this.Discount = discount;
    }

    decimal Amount { get; private set; }
    decimal Discount { get; private set; }


    public decimal AmountAfterDiscount
    {
            [ReaderLock] get { return this.Amount - this.Discount; }
    }
}
```

ReaderLockAttribute 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 WriterLockAttribute will be blocked until all reads are complete.

Likewise, invoking properties or methods marked with WriterLockAttribute 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 ReaderLockAttribute or WriterLockAttribute, ReaderWriter-SynchronizedAttribute 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 or methods that read a single field are intrinsically thread-safe and don't need to be marked with a [ReaderLockAttribute] custom attribute.

Enabling and Disabling ReaderLock/WriterLock checks

Runtime checks for ReaderLockAttribute and WriterLockAttribute attributes can be expensive and therefore it's recommended that these checks be disabled in the final release build. To accommodate this, the ReaderWriterSynchronizedAttribute attribute contains a member called RuntimeVerification-Enabled which is set in the constructor to turn verifications on or off. Setting this flag is best accomplished by defining a pair of constant bool's in a separate class which are set to true/false based on the build.

This is demonstrated in the following example where a class called `ConditionalConstants` defines a variable called `IsDebug` and sets it to `true` or `false` based on the type of build. The constant is then assigned to RuntimeVerificationEnabled in ReaderWriterSynchronizedAttribute's constructor:

```
static class ConditionalConstants
{
#if DEBUG
    public static const bool IsDebug = true;
#else
    public static const bool IsDebug = false;
#endif
}


[ReaderWriterSynchronized(RuntimeVerificationEnabled=ConditionalConstants.IsDebug)]
class Order
{
    // Details skipped for brevity.
}
```

Raising synchronous events

In some situations, a method with write access needs to allow other threads to read the object before another write is performed on the object. The implementation of **[I:System.Collections.Specialized.INotifyCollectionChanged]** gives a typical example of this situation. The **CollectionChanged** event defined by this interface is typically raised from a write method but is consumed from the user interface thread. The object cannot have changed between the moment the event is raised and it is processed by the UI thread, because the event arguments contain data that relates to the current state of the object. Using only WriterLockAttribute and ReaderLockAttribute would either result in deadlocks or in inconsistencies, respectively.

The solution to this problem is to use the ObserverLockAttribute custom attribute, which allows read access from other threads but prevents any other thread from acquiring a writer lock.

In the following example, `OrderCollection` is a collection of `Order` objects. In this example, the `Add()` and `Remove()` methods are marked with the WriterLockAttribute attributes. Listeners can be notified about these changes by subscribing to the **CollectionChanged** event which is exposed through the implementation of **[I:System.Collections.Specialized.INotifyCollectionChanged]**

Since listeners can be on other threads (e.g. a UI thread), this event is invoked by the `Add()` and `Remove()` methods via a method called OnCollectionChanged() which has been marked with the ObserverLockAttribute attribute. This lock ensures that the listener (which may be in another thread space) can read the current state of the collection without the collection being modified by another invocation of the `Add()` or `Remove()` operations from another thread.

```
[ReaderWriterSynchronized]
class OrderCollection : ICollection, INotifyCollectionChanged
{
    ArrayList list = new ArrayList();

    // Details skipped.

    [ReaderLock]
    public int Count
    {
        get
        {
            return list.Count;
        }
    }

    [WriterLock]
    public void Add(Order o)
    {
        list.Add(o);
        NotifyCollectionChangedEventArgs changedArgs = new NotifyCollectionChangedEventArgs(Nc
        OnCollectionChanged(changedArgs);
    }

    [WriterLock]
    public void Remove(int index)
    {
        NotifyCollectionChangedEventArgs changedArgs = new NotifyCollectionChangedEventArgs(Nc
        list.RemoveAt(index);
        OnCollectionChanged(changedArgs);
    }

    [ObserverLock]
    private void OnCollectionChanged(NotifyCollectionChangedEventArgs changedArgs)
    {
        CollectionChanged(this, changedArgs);
    }

    [ReaderLock]
    public Order Get(int index)
    {
        return (Order)list[index];
    }

    public event NotifyCollectionChangedEventHandler CollectionChanged;
}
```

Synchronizing access to multiple instances

The previous example showed how synchronization can be handled for a single class instance. In cases where synchronization is required for multiple instances (e.g. when an object is part of a collection), the class requiring synchronization must also implement PostSharp's IReaderWriter-

Synchronized interface. IReaderWriterSynchronized returns an instance of PortableReaderWriterLock which is used to manage access to resources.

To demonstrate this, we'll begin by adding a collection of `Line` classes to the `Order` class, each of which contains an amount which makes up the order. Since access to instances of the `Line` class are to be synchronized, the `Line` class will be marked with ReaderWriterSynchronizedAttribute and will also implement IReaderWriterSynchronized. We've also added a collection of `Line` objects and an `AddLine()` method to the containing `Order` class which adds new `Line` objects to this collection. This method is marked with the WriterLockAttribute attribute since it creates, and adds a `Line` object to the collection.

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

    //collection of objects to syncronize
    List<Line> lines = new List<Line>();

    [WriterLock]
    public Order AddLine(decimal amount)
    {
        Line line = new Line(this, amount);
        this.lines.Add(line);
        this.Amount += amount;
        return line;
    }

    [ReaderLock]
    public Line GetLine(int index)
    {
        return lines[index];
    }

    public decimal Amount { [ReaderLock] get; private set; }

    [ReaderWriterSynchronized]
    public class Line : IReaderWriterSynchronized
    {
        decimal amount;
        Order   parent;
        internal Line(Order parent, decimal amount)
        {
            this.Lock =PostSharp.Post.Cast<Order,IReaderWriterSynchronized>(parent).Lock;
            this.amount = amount;
            this.parent = parent;
        }

        public decimal Amount
        {
            [ReaderLock]
            get { return this.amount; }

            [WriterLock]
            set
            {
                decimal difference = value - this.amount;
                this.amount += value;
                this.parent.Amount += value;
            }
        }

        public PortableReaderWriterLock Lock
        {
            get; private set;
        }
    }
}
```

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 UpgradeableReaderLockAttribute and WriterLockAttribute attributes. The UpgradeableReader-LockAttribute 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 WriterLockAttribute 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 UpgradeableReaderLockAttribute 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 WriterLock-Attribute:

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

    public decimal Amount
    {
       [ReaderLock] get;

       [WriterLock]
       private set;
    }
```

```
        [UpgradeableReaderLock]
        public void Recalculate()
        {
            decimal total = (decimal)0.0;
            for (int i = 0; i < lines.Count; ++i)
            {
                total += lines[i].Amount;
            }

            this.Amount = total;
        }
    }
```

See Also

Reference

IReaderWriterSynchronized

PortableReaderWriterLock

ReaderWriterSynchronizedAttribute

WriterLockAttribute

UpgradeableReaderLockAttribute

3.2.1.3. Using the 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 a `Player` class for a simple ping pong like game in which Players draw random numbers. If the random number is within the predefined skill range for a given player then the player signals that it's the other player's turn to draw a random number. This logic continues until a player draws a random number that's above their skill level, in which case the player signals that the other player is the winner. Each player records the number of balls it successfully received.

Here is the single-threaded version of this game:

```
class Player
{
    readonly string name;
    readonly Random random = new Random();
    int totalBallsReceived;
    readonly double skills;

    public Player(string name, double skills )
    {
        this.name = name;
        this.skills = skills;
    }


    public Player Ping(Player peer)
    {
        if (random.NextDouble() <= this.skills)
        {
       this.totalBallsReceived++;
            return peer.Ping(this);
        }
        else
        {
            return peer;
        }
    }
}
```

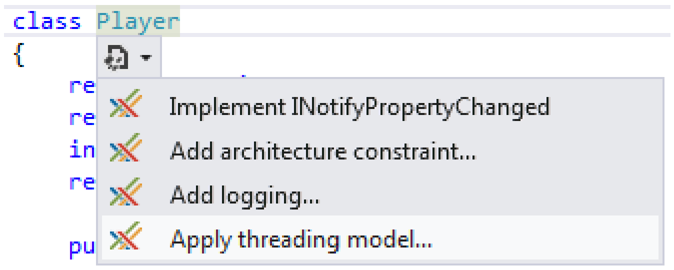
Note that accesses to the "`random`" and "`totalBallsReceived`" fields make the `Ping` method thread-unsafe. If we want an actor to be simultaneously involved in several games, we would need to make the object safe for concurrent accesses using locks. However, working directly with locks results in source code that is larger, less readable, and less reliable.

A better 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:

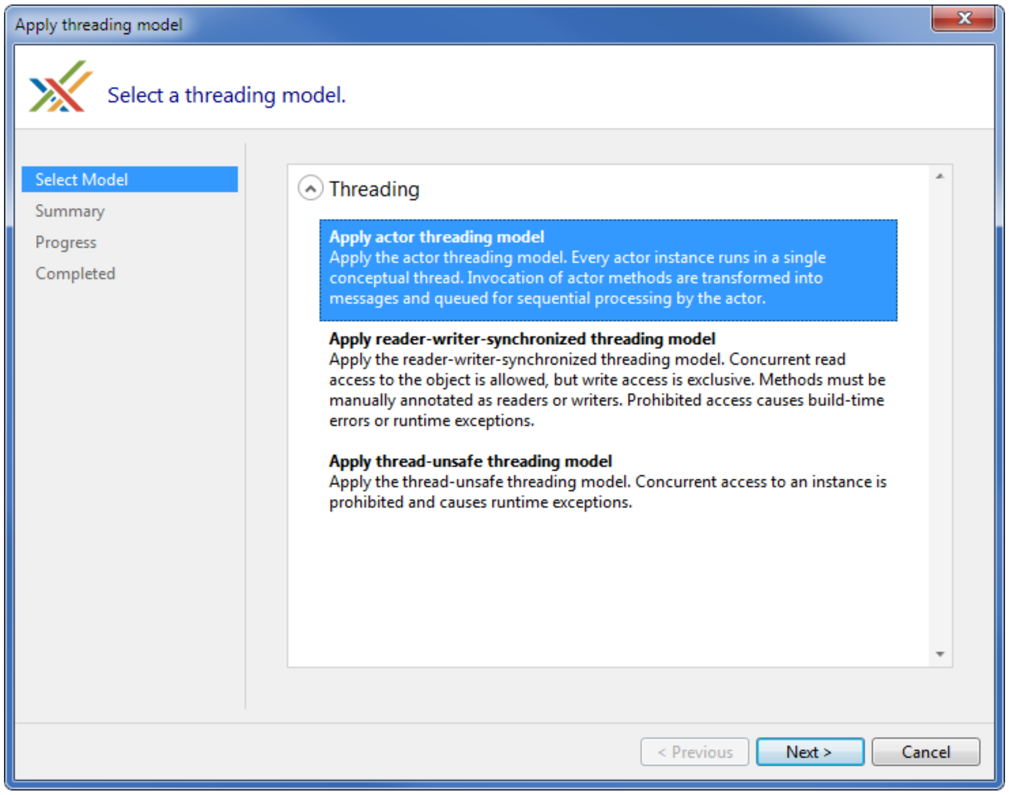
1. Place the mouse cursor over your class name and select “Apply threading model...” from the dropdown.



```
class Player
{
    re
    re
    in
    re

    pu
```

Implement INotifyPropertyChanged
Add architecture constraint...
Add logging...
Apply threading model...

2. Select “Apply actor threading model” and click **Next**.

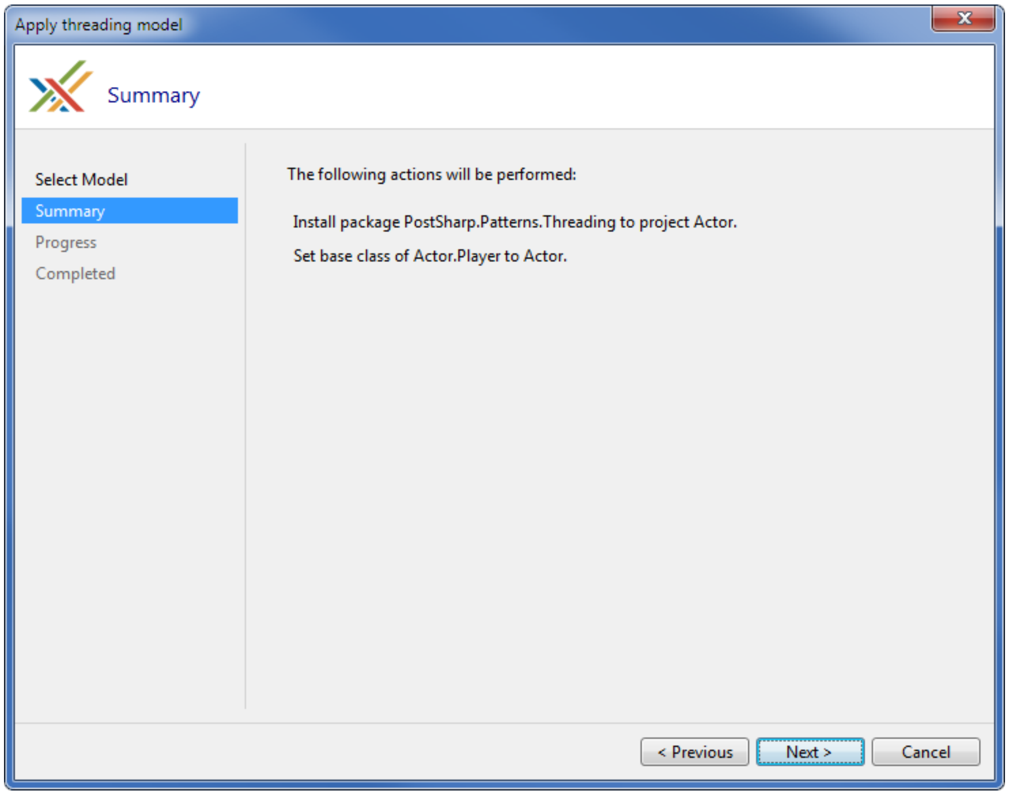


Apply actor threading model
Apply the actor threading model. Every actor instance runs in a single conceptual thread. Invocation of actor methods are transformed into messages and queued for sequential processing by the actor.

Apply reader-writer-synchronized threading model
Apply the reader-writer-synchronized threading model. Concurrent read access to the object is allowed, but write access is exclusive. Methods must be manually annotated as readers or writers. Prohibited access causes build-time errors or runtime exceptions.

Apply thread-unsafe threading model
Apply the thread-unsafe threading model. Concurrent access to an instance is prohibited and causes runtime exceptions.

3. Verify the actions on the Summary screen and click **Next**.

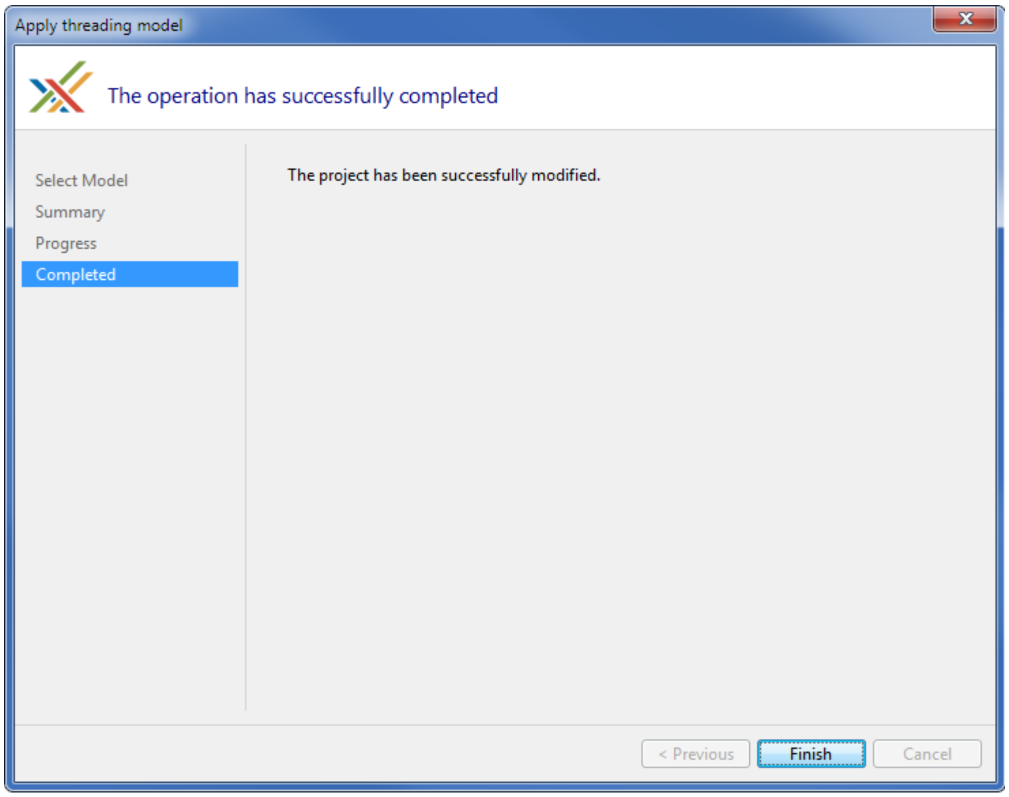


The following actions will be performed:

Install package PostSharp.Patterns.Threading to project Actor.

Set base class of Actor.Player to Actor.

4. Click **Finish** after the installation completes:



The operation has successfully completed

The project has been successfully modified.

Your class will now derive from Actor and all dependencies will have been added to the project.

Applying the Actor manually

To apply the Actor threading model manually:

1. Add the PostSharp.Patterns.Threading NuGet package to your project.
2. Derive your class from PostSharp.Patterns.Threading.Actor.

In the reworked example below, the `Player` class has been derived from the Actor class and the `Ping` method has been changed into an asynchronous method:

```
class Player : Actor
{
    readonly string name;
    readonly Random random = new Random();
    int totalBallsReceived;
    readonly double skills;

    public Player(string name, double skills )
    {
        this.name = name;
        this.skills = skills;
    }


    public async Task<Player> Ping(Player peer)
    {
        if (random.NextDouble() <= this.skills)
        {
       this.totalBallsReceived++;
            return await peer.Ping(this);
        }
        else
        {
            return peer;
        }
    }
}
```

TODO: show of the code is used.

Behind the scenes, each invocation of `Ping()` is added to the message queue by the Actor class, 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 `Player` class is never being accessed concurrently by two threads, and eliminates the need to make the class thread-safe.

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.

Using the `Player` class example from above, we may want to add a `ToString()` method which returns the name of the player. In this situation we don't really need this method to be dispatched since it's guaranteed that concurrency is guaranteed to will never be an issue.

To avoid dispatching this method, apply the ExplicitlySynchronizedAttribute attribute:

```
class Player : Actor
{
    readonly string name;
    .
    .
    .

    [ExplicitlySynchronized]
    public string ToString()
```

```
        {
            return this.name;
        }
    }
```

Applying this attribute explicitly tells PostSharp that we take responsibility for the synchronization of this method.

3.2.2. Dispatching a Method to the UI Thread

When you are building rich client user interfaces you have to be vigilant not to lock up the user interface while performing intensive background tasks.

This topic contains the following sections.

Using BackgroundWorker

One of the techniques available is to make use of the built in .NET BackgroundWorker object. This allows you to remove background processing from the UI thread. As you can see in the example below there are a lot of pieces that you need to make this work.

```
public partial class CustomerEdit : Form
{
    private readonly BackgroundWorker _backgroundWorker;

    public CustomerEdit()
    {
        InitializeComponent();
        _backgroundWorker = new BackgroundWorker();
        _backgroundWorker.DoWork += DoSave;
        _backgroundWorker.RunWorkerCompleted += SaveCompleted;
    }

    private void SaveCompleted(object sender, RunWorkerCompletedEventArgs e)
    {
        lblStatus.Text = "Finished Saving";
    }

    private void btnSave_Click(object sender, EventArgs e)
    {
        _backgroundWorker.RunWorkerAsync();
    }

    private void DoSave(object sender, DoWorkEventArgs doWorkEventArgs)
    {
        var customerRepository = new CustomerRepository();
        customerRepository.Save(BuildCustomerFromScreen());
    }

    private Customer BuildCustomerFromScreen()
    {
        return new Customer();
    }
}
```

To properly manage and interact with a BackgroundWorker process you need to hook the `DoWork` and `RunWorkerCompleted` events. You need to couple the code that should be run to the `DoWorker` event handler. The code to update the UI needs to be coupled to the `RunWorkerCompleted` event handler and, finally, you need to execute the BackgroundWorker object's `RunWorkerAsync` method to kick off the whole process.

As you can see, there are a lot of moving pieces in a very simple BackgroundWorker example. Here's how PostSharp simplifies this common coding scenario.

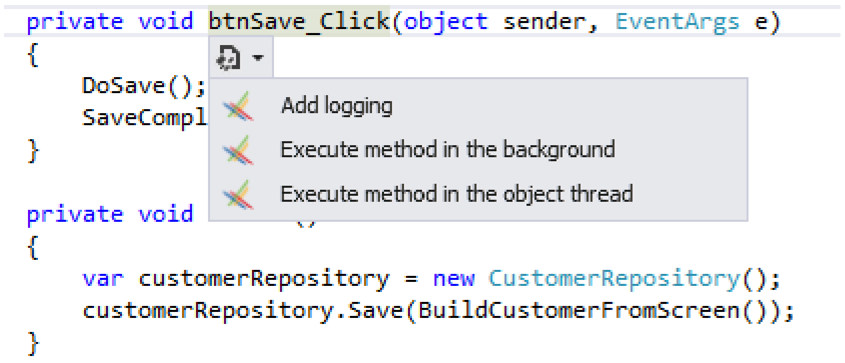
Adding background processing

To add background processing:

1. The first thing you need to understand is that you can write your code as if you had no thoughts of having any background processing occuring.

```
public partial class CustomerEdit : Form
{
    public CustomerEdit()
    {
        InitializeComponent();
    }

    private void SaveCompleted()
    {
        lblStatus.Text = "Finished Saving";
    }

    private void btnSave_Click(object sender, EventArgs e)
    {
        DoSave();
        SaveCompleted();
    }

    private void DoSave()
    {
        var customerRepository = new CustomerRepository();
        customerRepository.Save(BuildCustomerFromScreen());
    }

    private Customer BuildCustomerFromScreen()
    {
        return new Customer();
    }
}
```

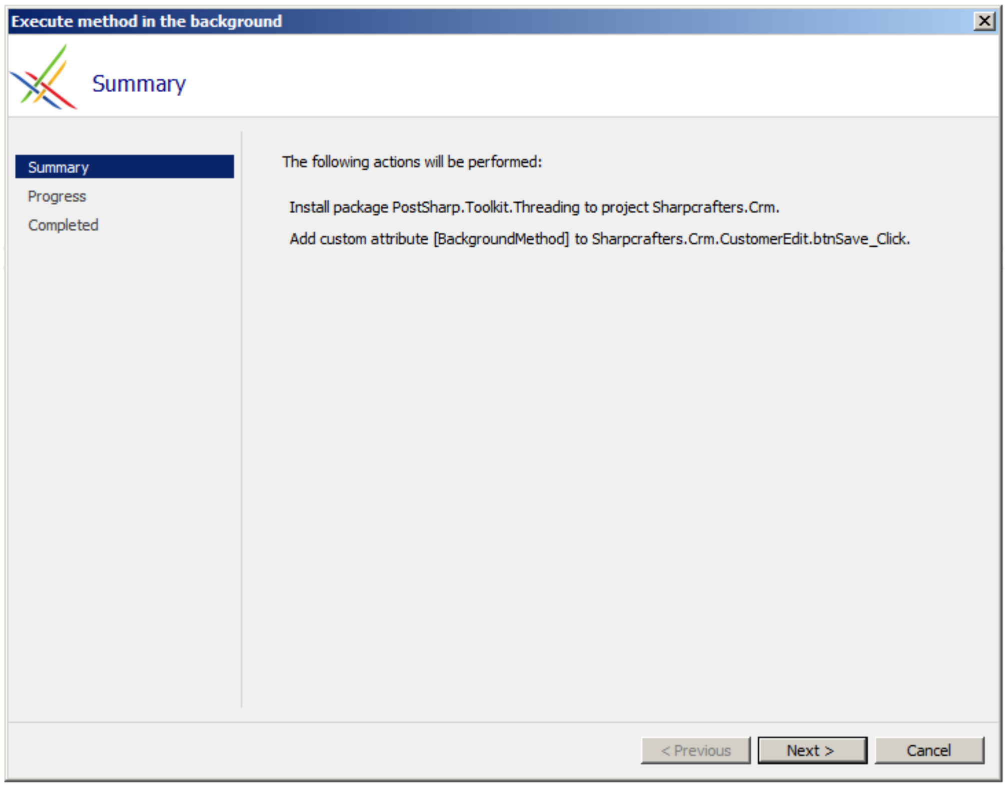
2. Once you have written your code you will need to determine which piece of it should run in the background and in the UI threads. In this example the `btnSave_Click` method should execute in the background and the `SaveCompleted` method should execute in the UI thread. First, you should add the background processing to the `btnSave_Click` method.



```
private void btnSave_Click(object sender, EventArgs e)
{
    DoSave();
    SaveCompl
}

private void
{
    var customerRepository = new CustomerRepository();
    customerRepository.Save(BuildCustomerFromScreen());
}
```

Add logging
Execute method in the background
Execute method in the object thread

3. You will be prompted with a wizard to complete the process. Accept the first page.

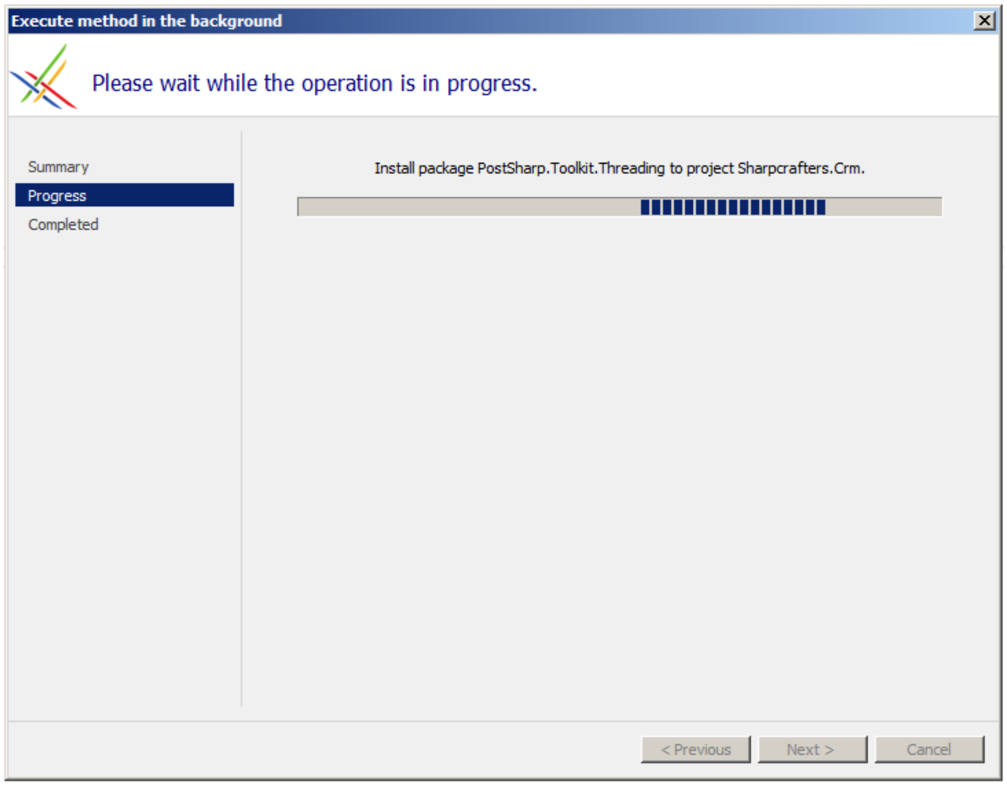


The following actions will be performed:

Install package PostSharp.Toolkit.Threading to project Sharpcrafters.Crm.

Add custom attribute [BackgroundMethod] to Sharpcrafters.Crm.CustomerEdit.btnSave_Click.

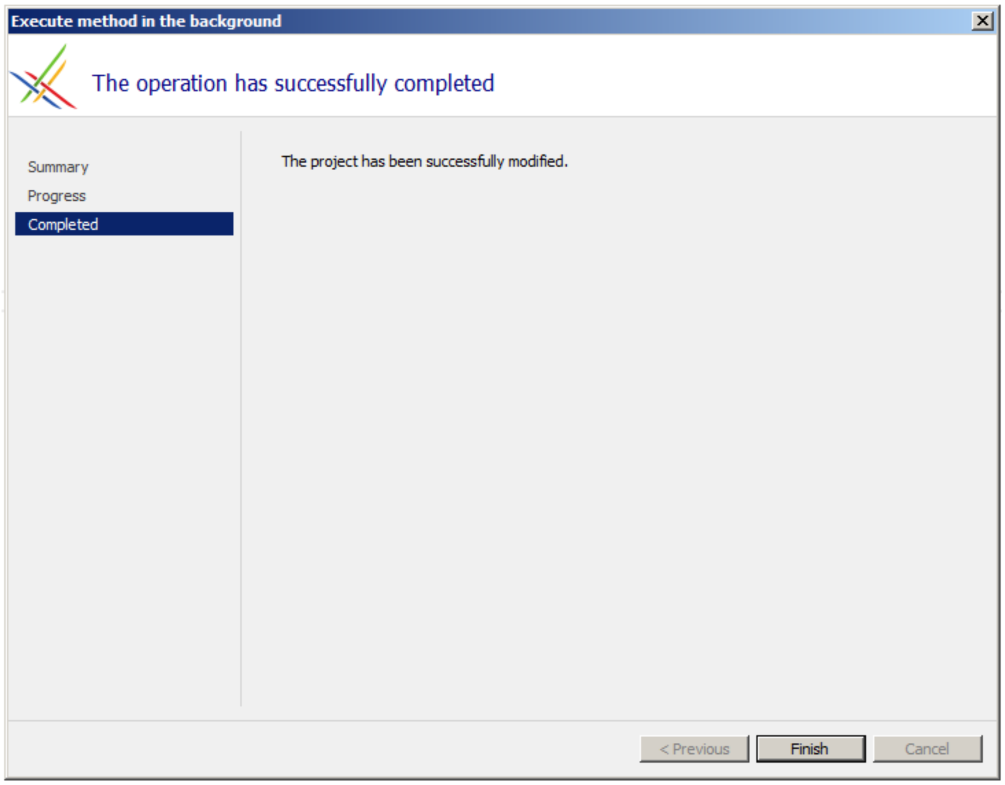
4. If you have not yet added background processing to your project, PostSharp will download the Threading Pattern Library and add it for you.



Please wait while the operation is in progress.

Install package PostSharp.Toolkit.Threading to project Sharpcrafters.Crm.

5. Click **Finish** to complete the process.



The operation has successfully completed

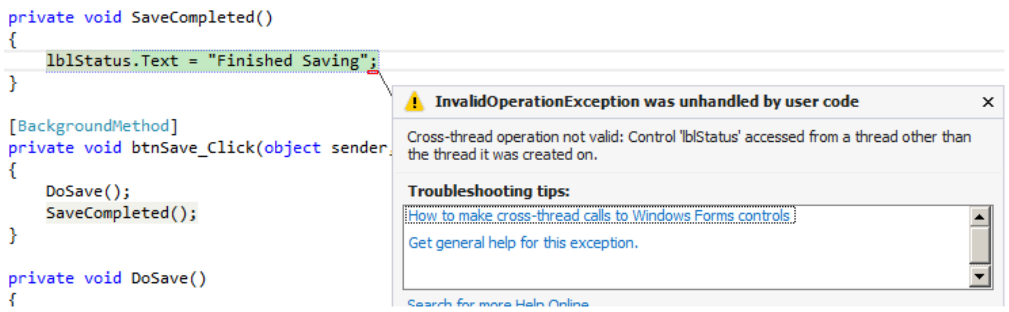
The project has been successfully modified.

6. When you look at your code you will see that only one thing has changed. The method you designated to run in the background is now decorated with the [BackgroundWorker] attribute.

```
[BackgroundMethod]
private void btnSave_Click(object sender, EventArgs e)
{
    DoSave();
    SaveCompleted();
}
```

Adding UI thread processing

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.



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

[BackgroundMethod]
private void btnSave_Click(object sender,
{
    DoSave();
    SaveCompleted();
}

private void DoSave()
{
```

InvalidOperationException was unhandled by user code

Cross-thread operation not valid: Control 'lblStatus' accessed from a thread other than the thread it was created on.

Troubleshooting tips:

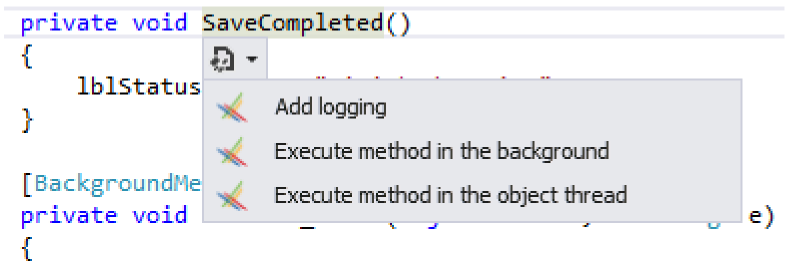
How to make cross-thread calls to Windows Forms controls

Get general help for this exception.

Search for more Help Online

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.


Add logging
Execute method in the background
Execute method in the object thread

```
private void SaveCompleted()
{
    lblStatus
}

[BackgroundMe
private void                                        e)
{
```

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 [DispatchedMethod] attribute to the SaveCompleted method.

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

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.

See Also

Reference

BackgroundWorker

3.2.3. 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.

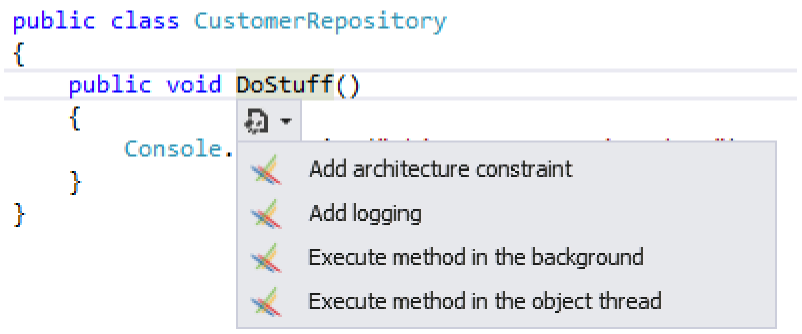
PostSharp provides you with the ability to push execution of a method to a background thread without having to worry about all of the boiler plate code.

To add [BackgroundMethod] attribute:

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

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

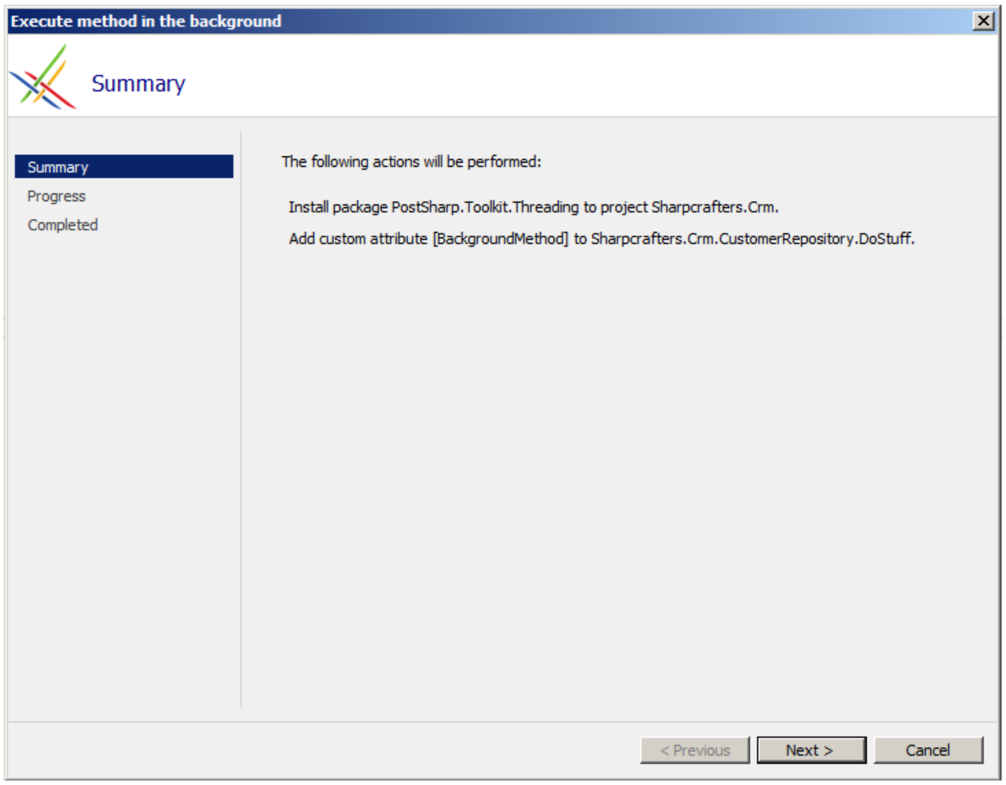
2. Select "Execute method in the background" from the Smart Tag available under the method name.



```
public class CustomerRepository
{
    public void DoStuff()
    {
        Console.
    }
}
```

Add architecture constraint
Add logging
Execute method in the background
Execute method in the object thread

3. A summary of the changes that will be made is presented to you.

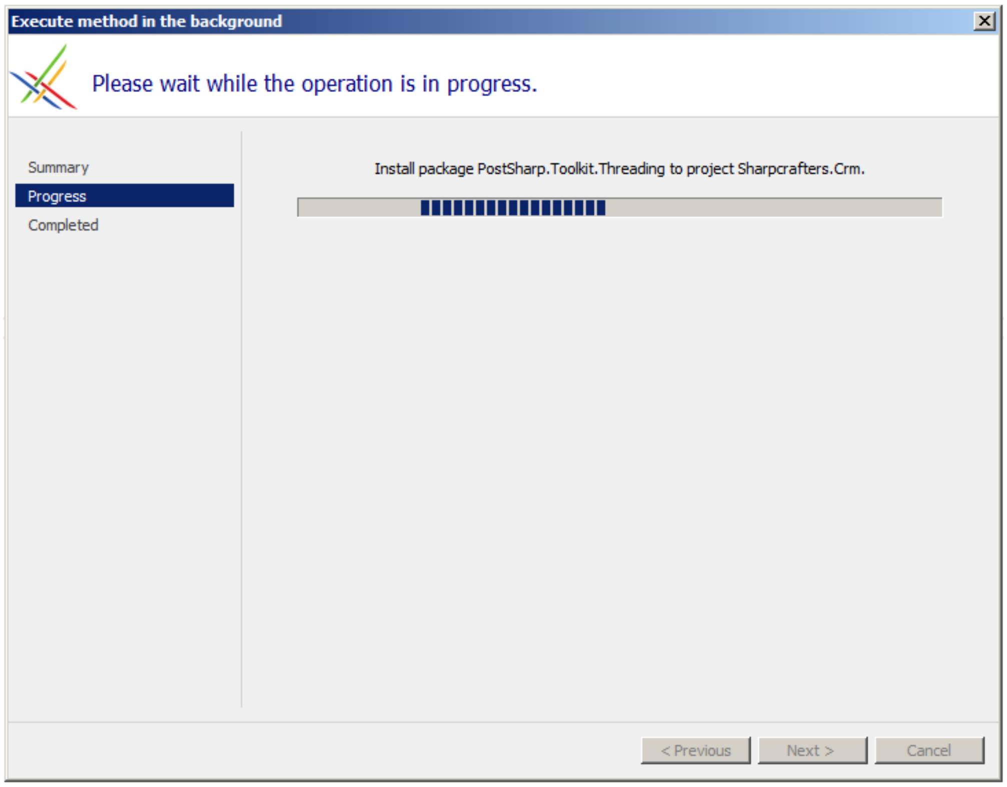


The following actions will be performed:

Install package PostSharp.Toolkit.Threading to project Sharpcrafters.Crm.

Add custom attribute [BackgroundMethod] to Sharpcrafters.Crm.CustomerRepository.DoStuff.

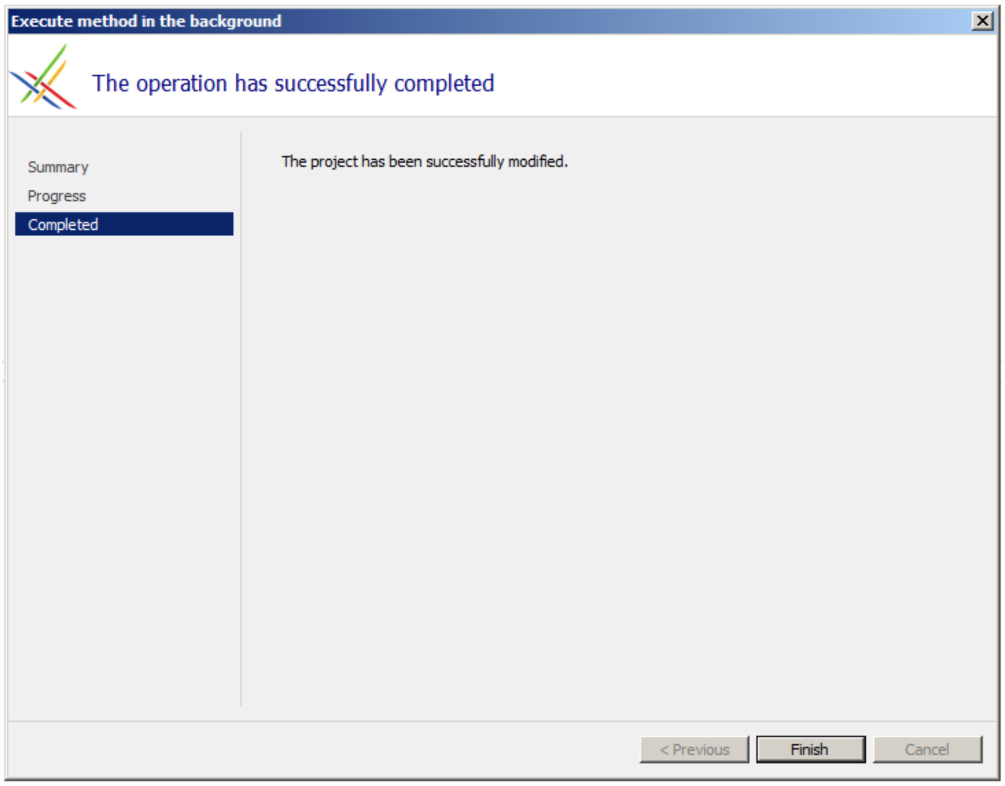
4. At this time, and if it is necessary, PostSharp will download the Threading Pattern Library and add it to your project.



Please wait while the operation is in progress.

Install package PostSharp.Toolkit.Threading to project Sharpcrafters.Crm.

5. Once the process has completed successfully you'll be presented with the final page of the wizard.



The operation has successfully completed

The project has been successfully modified.

6. You'll notice that only one change was made to your codebase. The `[BackgroundMethod]` 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.

```
public class CustomerRepository
{
    [BackgroundMethod]
    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.

See Also

Reference

BackgroundWorker

3.2.4. Detecting Deadlocks at Runtime

The Threading Pattern Library contains a deadlock detection aspect. Instead of letting your application freeze because of a deadlock, it will throw an exception with detailed information about which threads and which objects are involved in the deadlock. This makes it possible to diagnose deadlock that happens in production, which is otherwise usually very difficult.

To enable the deadlock detection algorithm, you need to add the DeadlockDetectionPolicy custom attribute to each assembly of the solution.

See DeadlockDetectionPolicy for details.

3.3. Working with the Model Pattern Library

The Model Pattern Library provides the following features:

- **INotifyPropertyChanged.** The NotifyPropertyChangedAttribute aspect implements the INotifyPropertyChanged interface and automatically raises the PropertyChanged event for the relevant properties whenever an object is changed. For details, see Automatically implementing INotifyPropertyChanged at page 90.
- **Code Contracts**. The namespace PostSharp.Patterns.Contracts contains custom attributes such as RequiredAttribute, which can be applied to fields, properties or parameters, and validates their value at runtime. See Validating Parameters, Fields and Properties at page 103 for details.

See Also

Reference

NotifyPropertyChangedAttribute

PostSharp.Patterns.Contracts

3.3.1. Automatically implementing 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 elminate 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.

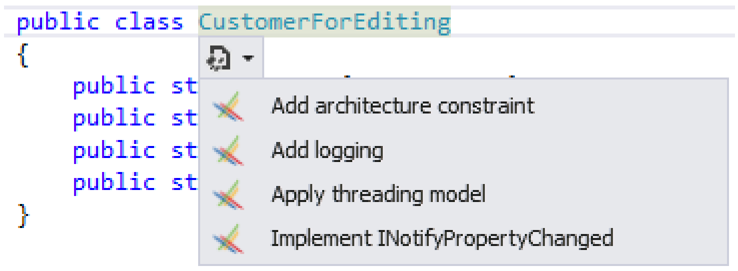
PostSharp can completely eliminate all of that repetition for you. All you have to do is make use of the Model Pattern Library's NotifyPropertyChangedAttribute aspect.

To add INotifyPropertyChanged aspect:

1. Let's add NotifyPropertyChangedAttribute to the `CustomerForEditing` class:

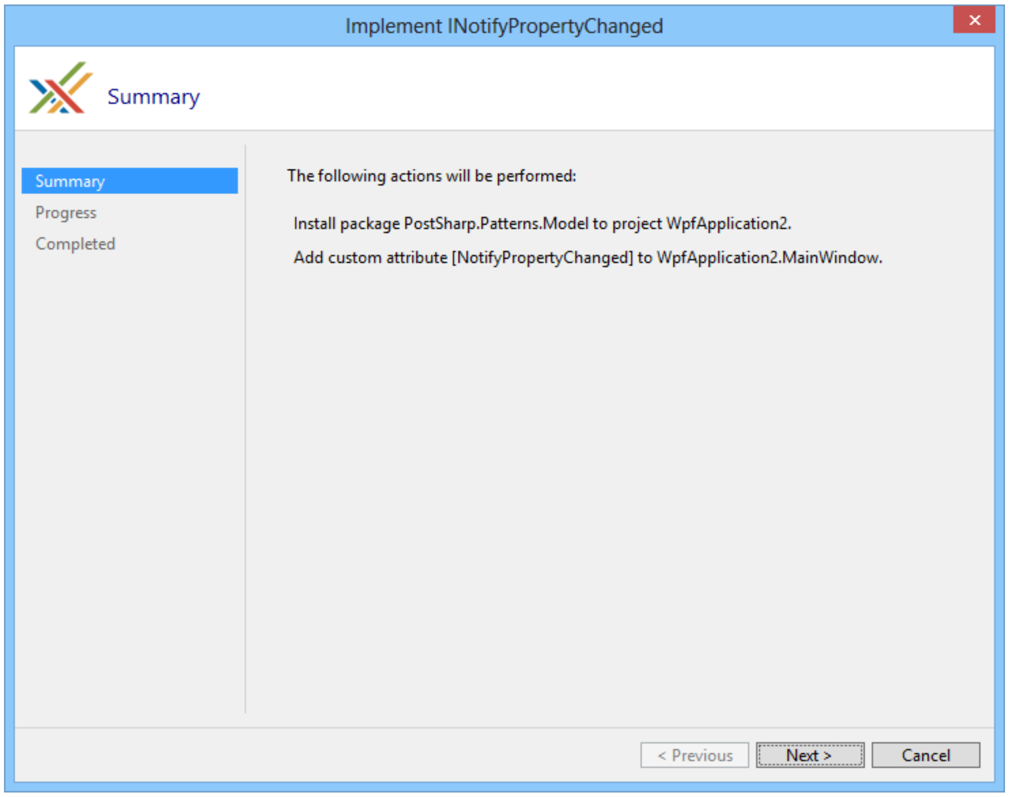
```
public class CustomerForEditing
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string FullName
    {
        get { return string.Format("{0} {1}", this.FirstName, this.LastName);}
    }
    public string Phone { get; set; }
    public string Mobile { get; set; }
    public string Email { get; set; }
}
```

2. Put the caret on the class name and expand the Smart Tag. From the list select "Implement INotifyPropertyChanged".



```
public class CustomerForEditing
{
    public st
    public st
    public st
    public st
}
```

3. 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.

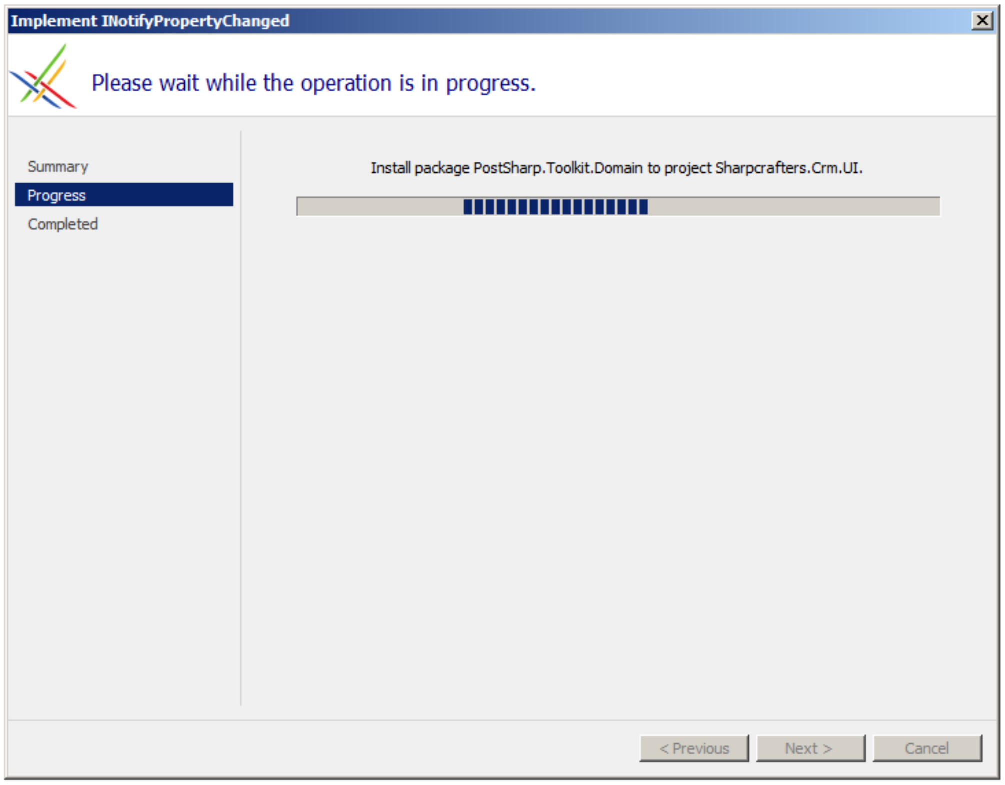


The following actions will be performed:

Install package PostSharp.Patterns.Model to project WpfApplication2.

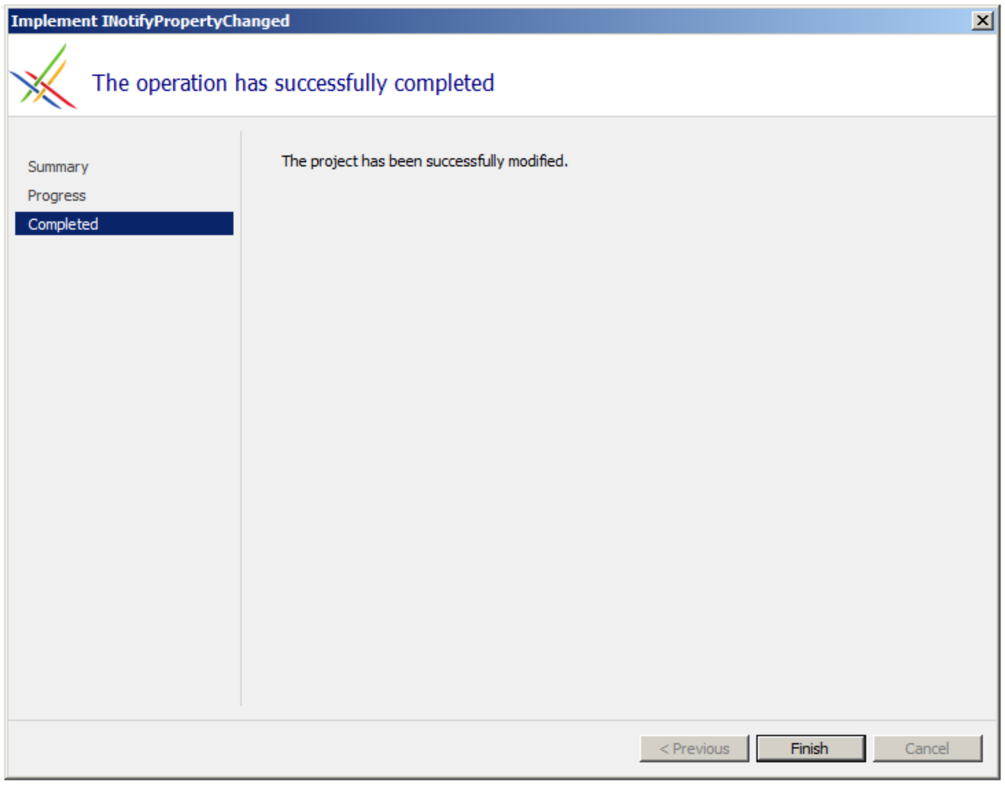
Add custom attribute [NotifyPropertyChanged] to WpfApplication2.MainWindow.

4. PostSharp will download the Model Pattern Library and add the attribute.


Implement INotifyPropertyChanged
Please wait while the operation is in progress.
Summary
Progress
Completed
< Previous
Next >
Cancel

Install package PostSharp.Toolkit.Domain to project Sharpcrafters.Crm.UI.

5. 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.



The operation has successfully completed

The project has been successfully modified.

6. 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 NotifyPropertyChanged-Attribute 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); }
    }
    public string Phone { get; set; }
    public string Mobile { get; set; }
    public string Email { get; set; }
}
```

Note

This example has added NotifyPropertyChangedAttribute to one class. If you need to implement NotifyPropertyChangedAttribute to many different classes in your codebase you will want to read about using aspect multicasting. See the section Adding Aspects to Multiple Declarations at page 116.

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.

See Also

Reference

INotifyPropertyChanged

NotifyPropertyChangedAttribute

3.3.1.1. 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 at page 96
- Handling Virtual Calls, References, and Delegates in a Get Accessor at page 96
- Handling Local Variables at page 98
- Handling Dependencies on Pure Methods at page 99

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 IgnoreAutoChangeNotificationAttribute 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 [SafeFor-DependencyAnalysisAttribute] 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 Depends On 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 dependant 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.SecondaryContac
      }
    }
  }
```

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.

See Also

Reference

PureAttribute

IgnoreAutoChangeNotificationAttribute

SafeForDependencyAnalysisAttribute

Depends On

Depends

3.3.1.2. 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 Notify-PropertyChangedAttribute 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 `LastName` properties.

1. Add the `CustomerModel` class to your project ensuring that the NotifyPropertyChanged-Attribute 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.Fi

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

}
```

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.Fi
    public string LastName { get { return this.model.LastName; } set { this.model.Last

    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.Fi
    public string LastName { get { return this.model.LastName; } set { this.model.Last

    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 at page 95 to learn about referencing properties without using the `this.field.Property` form.

See Also

Reference

NotifyPropertyChangedAttribute

3.3.2. Validating Parameters, Fields and Properties

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

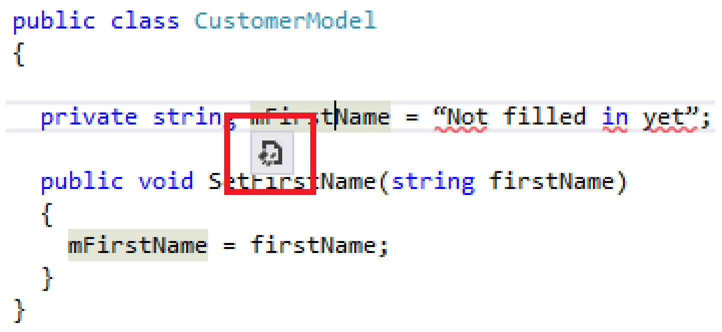
This section contains the following subsections.

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.

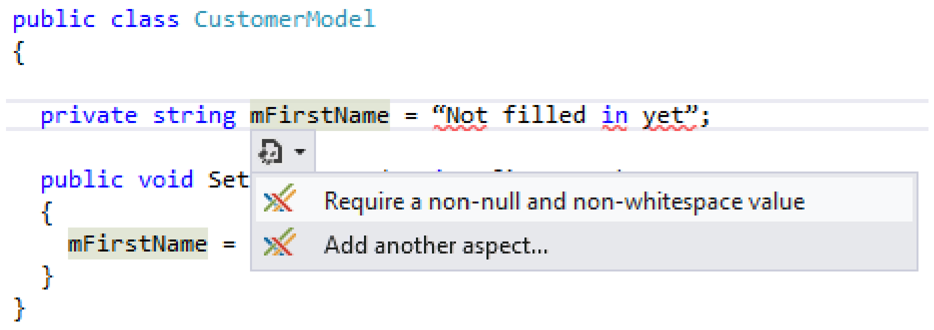
To add contract using the UI:

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 dropdown will appear:



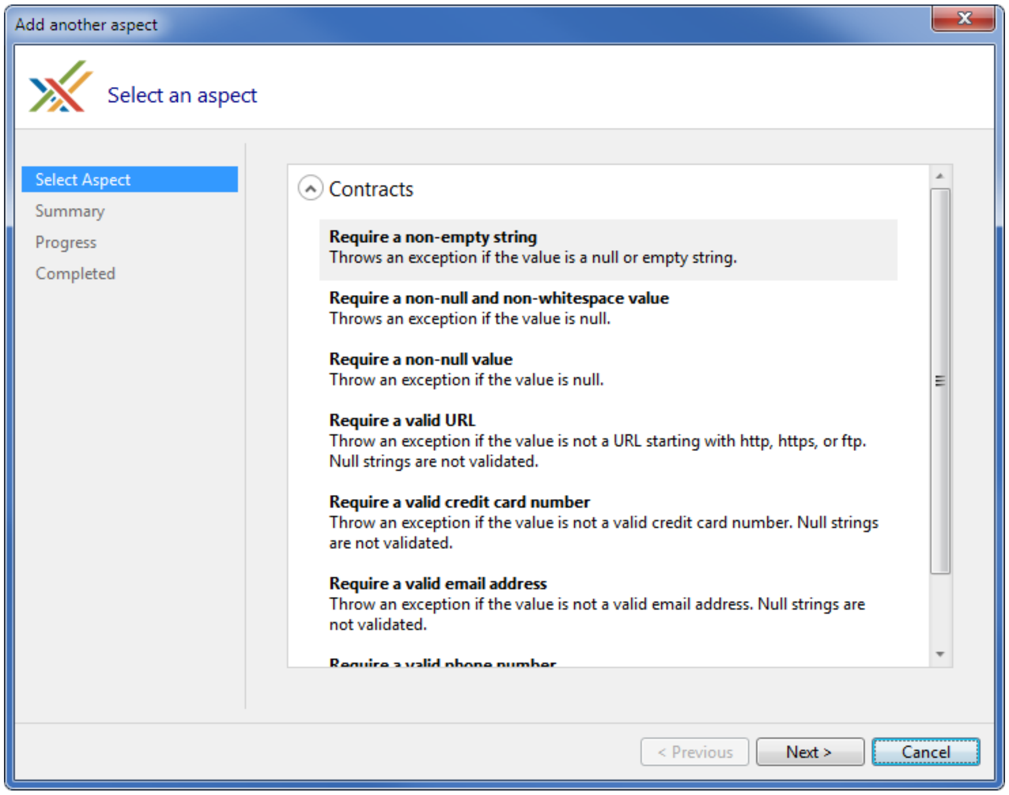
```
public class CustomerModel
{
  private string mFirstName = "Not filled in yet";

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

2. Click on the smart tag dropdown to reveal the contracts available:



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

  public void Set
  {
    mFirstName =
  }
}
```

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



Require a non-empty string
Throws an exception if the value is a null or empty string.

Require a non-null and non-whitespace value
Throws an exception if the value is null.

Require a non-null value
Throw an exception if the value is null.

Require a valid URL
Throw an exception if the value is not a URL starting with http, https, or ftp. Null strings are not validated.

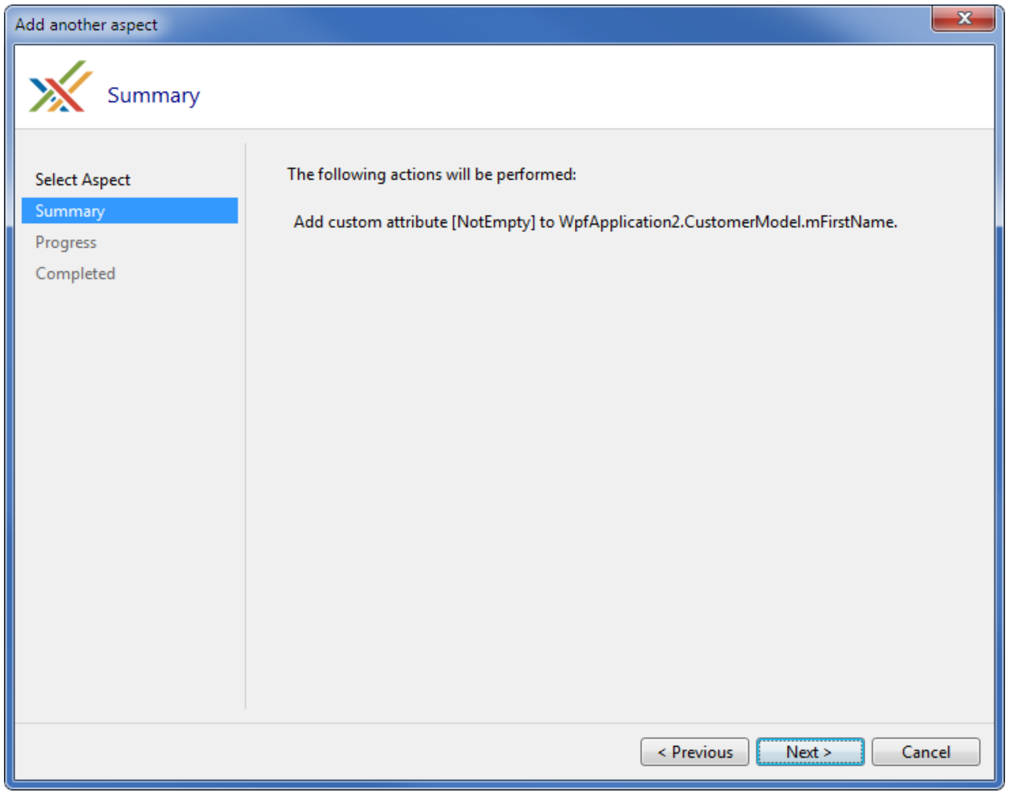
Require a valid credit card number
Throw an exception if the value is not a valid credit card number. Null strings are not validated.

Require a valid email address
Throw an exception if the value is not a valid email address. Null strings are not validated.

Require a valid phone number

4. Select a contract and click **Next**.

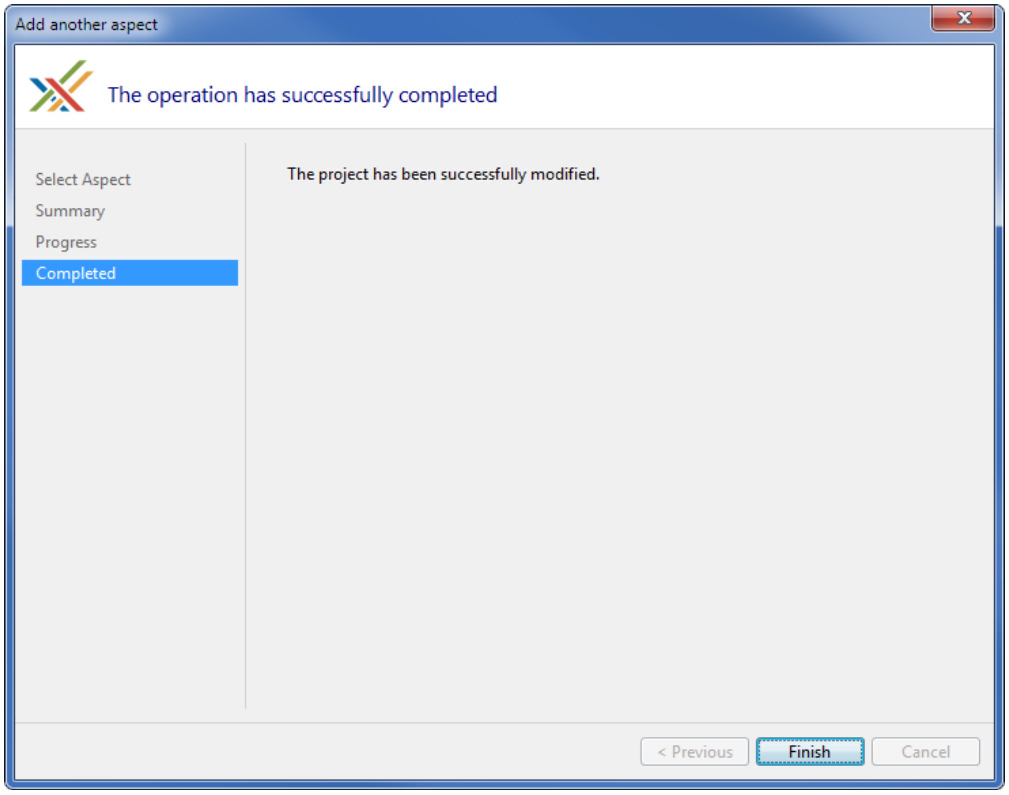
5. Confirm the addition of the contract and click **Next**:



The following actions will be performed:

Add custom attribute [NotEmpty] to WpfApplication2.CustomerModel.mFirstName.

6. Click **Finish** when the dialog indicates that the operation completed:



The project has been successfully modified.

The aspect has now been added in code:

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 ValidateValue(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:

```
public class NonZeroAttribute : LocationContractAttribute, ILocationValidationAspect<i
{
    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:

```
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:

```
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 PostSharp and PostSharp Model pattern assembly.

See Also

Reference

RequiredAttribute

PostSharp.Patterns.Model

PostSharp.Patterns.Contracts

PostSharp.Aspects

PostSharp.Reflection

LocationContractAttribute

ErrorMessage

ILocationValidationAspect

ValidateValue(T, String, LocationKind)

CHAPTER 4

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 at page 114 and Understanding Aspect Inheritance at page 133.

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 CompileTimeValidate(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.

```
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 at page 180 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`.

4.1. 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 wildcard 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 at page 115
- Adding Aspects to Multiple Declarations at page 116
- Adding Aspects to Derived Classes and Methods at page 119
- Overriding and Removing Aspect Instances at page 125
- Reflecting Aspect Instances at Runtime at page 129

For a conceptual overview of this feature, see:.

- Understanding Attribute Multicasting at page 130
- Understanding Aspect Inheritance at page 133

See Also

Reference

MulticastAttribute

MulticastAttributeUsageAttribute

IAspectProvider

4.1.1. 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.
        }
    }
```

4.1.2. 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 at page 136) and dynamic aspect providers (see more in the section Adding Aspects Programmatically using IAspect-Provider at page 137).

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 PostSharp's MulticastAttribute.

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

1. Open the `AssemblyInfo.cs`, or create a new file `GlobalAspects.cs` if you prefer to keep things separate (the name of this file does not matter).
2. Add an `[assembly:]` attribute that references the aspect you want 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.Controller
```

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 Attribute-Exclude to restrict where the aspect is attached.

```
[assembly: OurLoggingAspect(AttributeTargetTypes="OurCompany.OurApplication.Controllers.*", At
[assembly: OurLoggingAsepct(AttributeTargetMembers="Dispose", AttributeExclude = true, Attribu
```

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 at page 125 for more details about exclusing 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.Controller
                            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.Controller
```

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 CompileTime-Validate(Object) method return `false` without emitting any error, and the candidate target will be ignored. See the section **[usage-validation]** for details.

The second approach is opt-in. See the section Adding Aspects Programmatically using IAspect-Provider at page 137 for details.

See Also

Reference

MulticastAttribute

AttributeTargetTypes

AttributeExclude

AttributePriority

AttributeTargetTypeAttributes

CompileTimeValidate(Object)

PersistMetaData

4.1.3. 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 at page 247

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 hardcoding it into the custom attribute. In the following snippet, we've removed the Multicast-AttributeUsageAttribute attribute from `AutoDataContractAttribute`:

```
[

          // 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, a
    }

    public override void OnExit(MethodExecutionArgs args)
    {
        Console.WriteLine(string.Format("Leaving {0}.{1}.", args.Method.DeclaringType.Name, ar
    }
}
```

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.
    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 `MultiPageArcticle.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 at page 116 for more details), adding a method-level attribute to a class implicitly adds it to all method of this class.

```
[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.

See Also

Reference

MulticastAttributeUsageAttribute

TypeLevelAspect

InstanceLevelAspect

DataContractAttribute

DataMemberAttribute

IAspectProvider

MulticastAttribute

Inheritance

4.1.4. 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

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 at page 201 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 MulticastAttributeUsageAttribute AllowMultiple determines whether multiple instances of the same aspect are allowed on an element of code. By default, this property is set to `true` for all aspects.

In the following example, the methods in type `MyClass` are enhanced by one, two and three instances of the `Trace` aspect (see code comments).

```
using System;
using System.Diagnostics;
using PostSharp.Aspects;
using PostSharp.Extensibility;
using 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.
        }
    }
    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`.

```
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.
        }
    }
    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, Categ

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.
        }
    }
    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 MulticastAttribute AttributeExclude 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
            {

            }
          }
```

4.1.5. 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 MulticastAttributeUsageAttribute PersistMetaData property to `true`.

For instance:

```
[MulticastAttributeUsage( MulticastTargets.Class, PersistMetaData = true )]
        public class TagAttribute : MulticastAttribute
        {
          public string Tag;
        }
```

Note

Multicasting of attributes is not limited only to PostSharp aspects. You can multicast any custom attribute in your codebase in the same way as shown here. If a custom attribute is multicast with the PersistMetaData property set to `true`, when relfected on the compiled code will look as if you had manually added the custom attribute in all of the locations.

4.1.6. Understanding Attribute Multicasting

This topic contains the following sections.

Overview of the Multicasting Algorithm

Every multicast attribute class must be assigned a set of legitimate targets using the Multicast-AttributeUsageAttribute custom attribute, which is the equivalent and complement of Attribute-UsageAttribute for multicast attributes. Multicast attributes can be applied to types, methods, fields, properties, events, or/or 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.

Directly applied to	Implicitly applied to
Method	This method and the parameters of this method.
Field	This field.
Parameter	This parameter.

Filtering Target Elements of Code

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

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

Developer-Specified Filtering

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

For instance, the following piece of code specifies that the attribute `GuiThreadAttribute` can be applied 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
[AttributeUsage(AttributeTargets.Assembly|AttributeTargets.Class|AttributeTargets.Method, Allo
[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.*", AttributeTargetMember
```

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.
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.

MulticastAttribute Property	MulticastAttributeUsage-Attribute Property	Description
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 at page 133).

⚠ Caution

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

Overriding Filtering Attributes

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

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

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

4.1.7. 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 MulticastAttribute-UsageAttribute; see Adding Aspects Declaratively Using Attributes at page 114 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.
  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.Method2`.
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**
>
> 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 at page 125 for details.

See Also

Reference

MulticastAttribute

MulticastAttributeUsageAttribute

4.2. 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 DataContract and DataMember Attributes at page 247.

```
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 svproj), but with the .psproj extension:

```
<?xml version="1.0" encoding="utf-8"?>
<Project xmlns="http://schemas.postsharp.org/1.0/configuration">
    <Multicast xmlns:my="clr-namespace:MyCustomAttributes;assembly:MyAssembly">
        <my:AutoDataContractAttribute  AttributeTargetTypes=" 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:

```
<Multicast xmlns: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:AutoDataContractAttribute  AttributeTargetTypes="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 at page 116 for details about these properties.

See Also

4.3. 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 at page 95.

Note

PostSharp does not automatically initialize the aspects provided by IAspectProvider, even if the method `CompileTimeInitialize` is defined. Any initialization, if necessary, should be done in the `ProvideAspects` method or in the constructor of provided aspects.

However, these aspects are initialized at 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 run-time. 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.

See Also

Reference

IAspectProvider

NotifyPropertyChangedAttribute

ProvideAspects(Object)

AspectInstance

CHAPTER 5

Developing Custom Aspects

5.1. 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(MethodExecution-Args), OnSuccess(MethodExecutionArgs), OnException(MethodExecutionArgs) and OnExit(Method-ExecutionArgs)

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 at page 14 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 at page 178 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 at page 144.
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 at page 151.
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 at page 156.
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 at page 157.

Aspect Type	Targets	Description
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 at page 164.
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 at page 167.
CustomAttributeIntroductionAspect	Any	This aspect introduces a custom attribute on any element of code. A custom attribute can be represented as a CustomAt-tributeData or a ObjectConstruction. For details, see Introducing Custom Attributes at page 170.
ManagedResourceIntroductionAspect	Assemblies	This aspect introduces a managed resource into the current assembly. For details, see Introducing Managed Resources at page 175.
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 Validating Parameters, Fields and Properties at page 103.

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 System Attribute with multicasting capability. See Adding Aspects to Multiple Declarations at page 116 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.

5.1.1. 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 at page 156.

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 [SerializableAttribute] 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(Method-ExecutionArgs) 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 OnEntry(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(MethodExecutionArgs) 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.

```
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 reentrance into account. See the article Working with the Diagnostics Pattern Library at page 41 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 at page 156.

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(MethodExecutionArgs) 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(MethodExecution-Args), 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(Method-ExecutionArgs) and OnExit(MethodExecutionArgs) is `Continue`. For OnException(Method-ExecutionArgs) 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);
    }
}
```

Note

The value stored in MethodExecutionTag will not be shared between different instances of the aspect. If the aspect is attached to two different pieces of target code, each attachment will have its own unshared MethodExecutionTag for state storage.

See Also

Reference

OnMethodBoundaryAspect

Arguments

MethodExecutionTag

OnMethodBoundaryAspect

SerializableAttribute

PSerializableAttribute

OnExit(MethodExecutionArgs)

OnSuccess(MethodExecutionArgs)

OnException(MethodExecutionArgs)

OnEntry(MethodExecutionArgs)

Method

Arguments

Other Resources

PostSharp Aspect Framework - Product Page[6]

5.1.2. 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
{
```

6. http://www.postsharp.net/aspects

```
    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 at page 116 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 `GetExceptionAspect` 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 `BusinessExceptionAttribute` 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
   }
}

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

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` field of OnException(MethodExecutionArgs)'s args parameter. In the following snippet, OnException(MethodExecutionArgs) 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 `NullReferenceException` 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.Excep
    }
}
```

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.

See Also

Reference

OnException(MethodExecutionArgs)

MethodExecutionArgs

OnExceptionAspect

SerializableAttribute

GetExceptionType(MethodBase)

5.1.3. Intercepting Methods

Required introduction

Optional section title

Add one or more sections with content

5.1.4. Intercepting Properties and Fields

In .NET, both fields and properties are "things" that can be set and get. You can intercept get and set operations using the LocationInterceptionAspect. It makes it possible to develop useful aspects, such as validation, filtering, change tracking, change notification, or property virtualization (where the property is backed by a registry value, for instance).

This topic contains the following sections.

Intercepting Get operations

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 [SerializableAttribute].

   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.

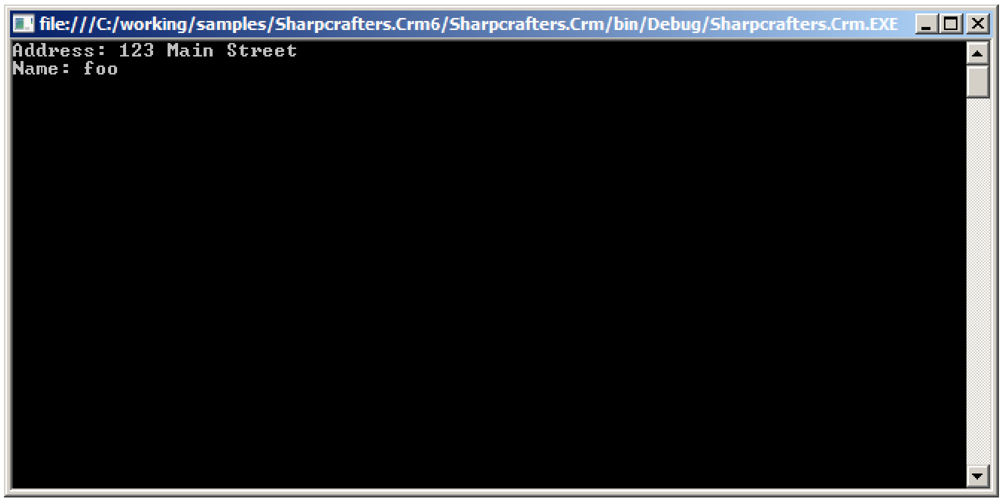
```
public class Customer
{
    [StringChecker]
    private readonly string _address;

    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 at page 116 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();
    }
}
```


file:///C:/working/samples/Sharpcrafters.Crm6/Sharpcrafters.Crm/bin/Debug/Sharpcrafters.Crm.EXE

```
Address: 123 Main Street
Name: foo
```

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 LocationInterceptionArgs’s GetCurrentValue method. This can be useful to check the current property value when a setter is called and then take some appropriate action.

For example, the following snippet shows a modified OnSetValue(LocationInterceptionArgs) method which gets the current underlying property value and compares the (new) value passed into the setter against the current value. If current and new value don't match then some message is written:

```
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(LocationInterceptionArgs). If several aspects are applied to the property (and/or to the property setter), GetCurrentValue will go through the next aspect in the chain of invocation.

PostSharp also provides a mechanism to set the underlying property in a getter via the SetNewValue(Object) method of LocationInterceptionArgs. This could be used for example, to ensure that a default value is assigned to the underlying property if there is currently no value. The following snippet shows a modified OnGetValue(LocationInterceptionArgs) method which gets the current underlying value, and sets a default value if the current value is null:

```
public override void OnGetValue(LocationInterceptionArgs args)
{
    object o = args.GetCurrentValue();
    if (o == null)
    {
        args.SetNewValue("value not set");
    }

    base.OnGetValue(args);
}
```

Intercepting fields

One benefit to implementing a LocationInterceptionAspect is that it can be applied directly to fields, allowing for reads and writes to those fields to be intercepted, just like with properties.

Applying a LocationInterceptionAspect implementation to a field is simply a matter of setting it as an attribute on a field, just as it was done with a property:

```
public class Customer
{
    .
    .
    .
[StringChecker]
public string name;
}
```

With the attribute applied to the name field, all attempts to get and set that field will be intercepted by `StringChecker` in its OnGetValue(LocationInterceptionArgs) and OnSetValue(LocationInterceptionArgs) methods.

Note that when a LocationInterceptionAspect is added to a field, the field is replaced by a property of the same field and visibility. The field itself is renamed and made private.

Getting the property or property being accessed

Information about the property or field being intercepted can be obtained through the LocationInterceptionArgs via its Location property. The type of this property, LocationInfo, can represent a FieldInfo, a PropertyInfo, or a ParameterInfo (although LocationInterceptionAspect cannot be added to parameters).

One use for this is to reflect the property name whenever a property is changed. In the following example, we have an `Entity` class that implements INotifyPropertyChanged and a public `OnPropertyChanged` method which allows notifications to be made whenever a property is changed. The `Customer` class has been modified to derive from `Entity`.

```
class Entity : INotifyPropertyChanged
{

  public event PropertyChangedEventHandler PropertyChanged;

  public void OnPropertyChanged(string propertyName)
  {
    if (PropertyChanged != null)
     PropertyChanged(this, new PropertyChangedEventArgs(propertyName));
  }
}

 class Customer : Entity
{
  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 Automatically implementing INotifyProperty-Changed at page 90.

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.

See Also

Reference

LocationInterceptionAspect

OnGetValue(LocationInterceptionArgs)

OnSetValue(LocationInterceptionArgs)

LocationInterceptionArgs

Location

LocationInfo

FieldInfo

PropertyInfo

ParameterInfo

INotifyPropertyChanged

NotifyPropertyChangedAttribute

5.1.5. Intercepting Events

You interact with events in three primary ways; subscribing, unsubscribing and raising them. Like methods and properties, you may find yourself needing to intercept these three interactions. How do you execute code 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(EventInterception-Args) 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 RemoveHandler(Delegate)

```
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 ex
                              args.Handler.Method,
                              string.Join(", ", args.Arguments.Select(
                                                                        a => a == null ?
                              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.

See Also

Reference

EventInterceptionAspect

Handler

Event

PSerializableAttribute

SerializableAttribute

OnAddHandler(EventInterceptionArgs)

OnAddHandler(EventInterceptionArgs)

OnRemoveHandler(EventInterceptionArgs)

OnInvokeHandler(EventInterceptionArgs)

EventInfo

RemoveHandler(Delegate)

5.1.6. 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.

```
[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.

```
[Serializable]
public class GeneralCompose : CompositionAspect
{
    private readonly Type _interfaceType;
    private readonly Type _implementationType;

    public GeneralCompose(Type interfaceType, Type implementationType)
    {
        _interfaceType = interfaceType;
        _implementationType = implementationType;
    }
```

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 interace type.

   To access the dynamically applied interface you must make use of a special PostSharp feature. The Cast SourceType, 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 instace 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");
    }
}
```

5.1.7. 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 DataContract-Attribute to a class and DataMemberAttribute to members of a class at build time.

1. Start by creating a class called `AutoDataContractAttribute` which derives from TypeLevel-Aspect. TypeLevelAspect transforms the class into an attribute which can be applied to other classes. Also implement IAspectProvider which exposes the ProvideAspects(Object) method for iterating on class members. ProvideAspects(Object) will be called for each member in the target class and will contain the code for applying the attributes:

   ```
   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 Custom-AttributeIntroductionAspect objects:

```
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(T
        CustomAttributeIntroductionAspect introduceDataMemberAspect =
            new CustomAttributeIntroductionAspect(
                new ObjectConstruction(typeof (DataMemberAttribute).GetConstructor(Typ


        // 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 |
        {
            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:

```
[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 at page 247 to learn how to have the DataContractAttribute 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 **DataAnnotations** namespace – `Editable`, `Display`, and `Required` respectively. Below `Product` is another class called `ProductViewModel` containing the same properties to which we want to copy the attributes to:

```
class Product
{
    [EditableAttribute(false)]
    [Required]
    public int Id { get; set; }

    [Display(Name = "The product's name")]
    [Required]
    public string Name { get; set; }
    public int RevisionNumber { get; set; }
}

class ProductViewModel
{
    public int Id { get; set; }
    public string Name { get; set; }
    public int RevisionNumber { get; set; }
}
```

To copy the attributes from the properties of `Product` to the corresponding properties of `ProductViewModel`, create an attribute class which can be applied to `ProductViewModel` to perform this copy process:

1. Create a TypeLevelAspect which implements IAspectProvider. In the snippet below our class is called `CopyCustomAttributesFrom`:

```
class CopyCustomAttributesFrom : TypeLevelAspect, IAspectProvider
{
}
```

2. Create a constructor to take in the class type from which the property attributes are to be copied from. This class type will be used in the next step to enumerate its properties:

```
class CopyCustomAttributesFrom : TypeLevelAspect, IAspectProvider
{
private Type sourceType;

public CopyCustomAttributesFrom(Type srcType)
{
    sourceType = srcType;
}
    }
```

3. Implement ProvideAspects(Object):

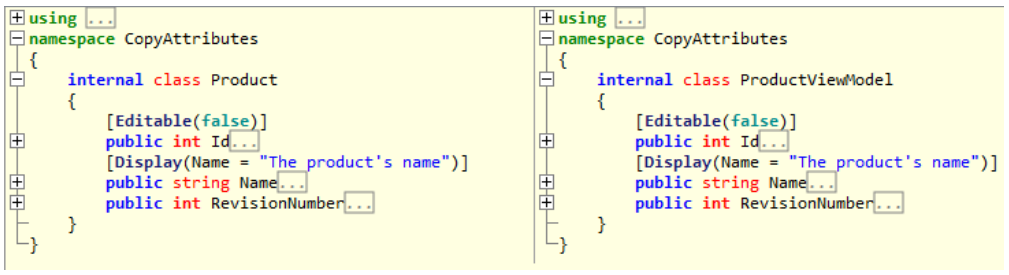
```
class CopyCustomAttributesFrom : TypeLevelAspect, IAspectProvider
{
    // Details skipped.

    public IEnumerable<AspectInstance> ProvideAspects(object targetElement)
    {
        Type targetClassType = (Type)targetElement;

        //loop thru each property in target
        foreach (PropertyInfo targetPropertyInfo in targetClassType.GetProperties())
        {
            PropertyInfo sourcePropertyInfo = sourceType.GetProperty(targetPropertyInf

            //loop thru all custom attributes for the source property and copy to the
            foreach (CustomAttributeData customAttributeData in sourcePropertyInfo.Get
            {
                //filter out attributes that aren’t DataAnnotations
                if (customAttributeData.AttributeType.Namespace.Equals("System.Compone
                {
                    CustomAttributeIntroductionAspect customAttributeIntroductionAspec
                        new CustomAttributeIntroductionAspect(new ObjectConstruction(c

                    yield return new AspectInstance(targetPropertyInfo, customAttribut
                }
            }

        }
    }
}
```

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 CopyCustomAttributesAttribute at build time:



```
using ...
namespace CopyAttributes
{
    internal class Product
    {
        [Editable(false)]
        public int Id...
        [Display(Name = "The product's name")]
        public string Name...
        public int RevisionNumber...
    }
}
```

```
using ...
namespace CopyAttributes
{
    internal class ProductViewModel
    {
        [Editable(false)]
        public int Id...
        [Display(Name = "The product's name")]
        public string Name...
        public int RevisionNumber...
    }
}
```

Note

It is not possible to delete or replace an existing custom attribute.

See Also

Reference

ProvideAspects(Object)

CopyCustomAttributesAttribute

DataContractAttribute

DataMemberAttribute

TypeLevelAspect

IAspectProvider

AspectInstance

CustomAttributeIntroductionAspect

DataAnnotations

5.1.8. 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 AssemblyLevel-Aspect. 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:

```
public sealed class AddBuildInfoAspect : AssemblyLevelAspect, IAspectProvider
{
    public IEnumerable<AspectInstance> ProvideAspects(object targetElement)
    {
    }
}
```

2. Implement the ProvideAspects(Object) method:

```
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 ManagedResourceIntroductionAspec

        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 ManagedResourceIntroduction-Aspect object, and returned via an AspectInstance. PostSharp then embeds the resource into the current assembly.

3. Open your project's AssemblyInfo.cs file and add a line to include the `AddBuildInfoAspect` class:

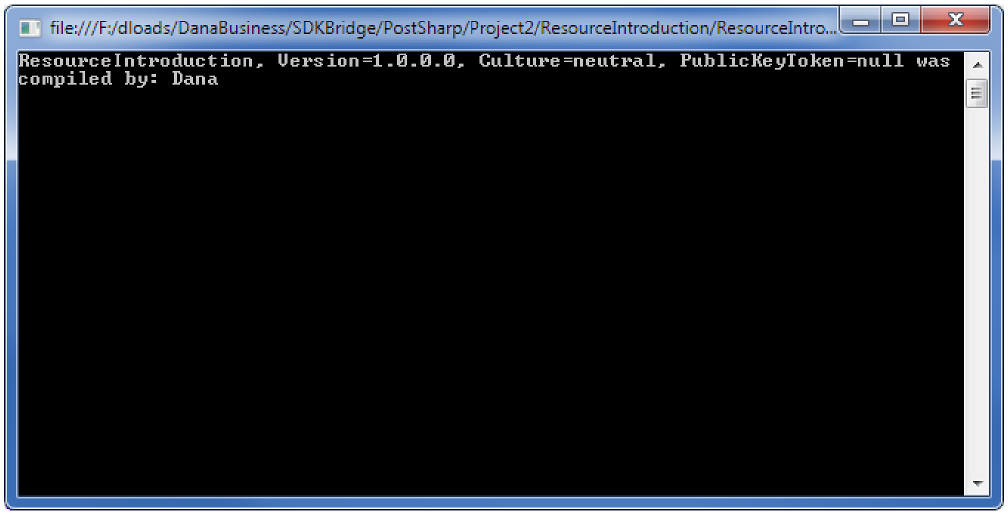
```
[assembly:AddBuildInfoAspect]
```

With this code in place the assembly will now embed the date and time as a resource into itself during compilation.

The following code demonstrates how to retrieve the data at 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:


file:///F:/dloads/DanaBusiness/SDKBridge/PostSharp/Project2/ResourceIntroduction/ResourceIntro...

```
ResourceIntroduction, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null was
compiled by: Dana
```

See Also

Reference

AssemblyLevelAspect

IAspectProvider

ManagedResourceIntroductionAspect

AspectInstance

5.2. 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 SerializableAttribute 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:

```
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 `GenericClass<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 IInstance-ScopedAspect:

```
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 at page 180 for details.

Phase	Step	Description
	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 at page 249

See Also

Other Resources

[5fafef23-0313-4eb9-9e4f-2ef80c616908]

5.3. 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 [MulticastAttributeUsage-Attribute], as described in the article Adding Aspects Declaratively Using Attributes at page 114. 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. Perfom a check to see if the target class implements the interface in question.

```
Type targetType = target.DeclaringType;
if (!typeof(ISecurable).IsAssignableFrom(targetType))
{

}
```

4. If the target does not implement the interface you must signal the compilation process that this target should not have the aspect applied to it. There are two ways to do this. The first option is to throw an InvalidAnnotationException.

```
if (!typeof(ISecurable).IsAssignableFrom(targetType))
{
  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 Attribute-TargetTypeAttributes or AttributeTargetMemberAttributes, its real power is to go beyond those filtering techniques. By using CompileTimeValidate(Object) you are able to filter aspect application in any manner that you can interogate your codebase using reflection.

Using Message Sources

If you plan to raise many messages, you may prefer to define your own MessageSource. A Message-Source 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** 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:

```
internal class ArchitectureMessageSource : MessageSource
{
    public static readonly ArchitectureMessageSource Instance = new ArchitectureMessag

    private ArchitectureMessageSource() : base( "PostSharp.Architecture", new Dispense
    {
    }

    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

                case 2:
                    return "{0} {1} cannot be referenced from {2} {3} because of the [

                case 3:
                    return "Cannot use [ComponentInternal] on {0} {1} because the {0}

                case 4:
                    return "Cannot use [Internal] on {0} {1} because the {0} is not pu

                default:
                    return null;
            }
        }
    }
}
```

3. Then you can use a convenient set of methods on your `MessageSource` object:

```
MyMessageSource.Instance.Write( classType, SeverityType.Error, "CUS001", new object[]
```

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 IValidable-Annotation.

Examples

Example: Dispatching a Method Execution to the GUI Thread at page 241

See Also

Reference

MethodLevelAspectCompileTimeValidate(MethodBase)

TypeLevelAspectCompileTimeValidate(Type)

FieldLevelAspectCompileTimeValidate(FieldInfo)

LocationLevelAspectCompileTimeValidate(LocationInfo)

EventLevelAspectCompileTimeValidate(EventInfo)

AssemblyLevelAspectCompileTimeValidate(_Assembly)

IValidableAnnotation

MessageSource

Message

MessageWrite

5.4. Initializing Aspects

As explained in the section Understanding Aspect Lifetime and Scope at page 178, 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 IInstanceScopedAspect CreateInstance(AdviceArgs) and IInstanceScoped-Aspect RuntimeInitializeInstance 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.

See Also

5.5. Developing Composite Aspects

PostSharp offers two approaches to aspect-oriented development. The first, as explained in section Developing Simple Aspects at page 141, 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 at page 201). 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 behaviour 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, LocationLevel-Aspect 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 at page 186	Advices with equivalent functionality as OnMethodBoundaryAspect, MethodInterceptionAspect, LocationInterceptionAspect, and EventInterceptionAspect.
[interface-introduction]	Make the aspect introduce an interface into the target class. The interface is implemented by the aspect itself.
Accessing Members of the Target Class at page 198	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.

5.5.1. Adding Behaviors to Existing Members

In order to add new behaviours 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
- Advice Kinds at page 187
- Pointcuts Kinds at page 188
- Grouping Advices at page 189

How to Add a Behavior to an Existing Member

1. Start with an empty aspect class deriving AssemblyLevelAspect, TypeLevelAspect, InstanceLevelAspect, MethodLevelAspect, LocationLevelAspect or EventLevelAspect. Mark it as serializable.
2. Choose an advice type in the list below. For instance: OnMethodEntryAdvice.
3. Create a method. The signature of this method should match exactly the signature matched by this advice type.
4. Annotate this method with a custom attribute of the advice type you chose. For instance: `[OnMethodEntryAdvice]`.
5. Choose a pointcut type in the list below. For instance: SelfPointcut. Annotate the advice method with that custom attribute. For instance: `[SelfPointcut]`.

Example

The following code shows a simple tracing aspect implemented with an advice and a pointcut. This aspect is exactly equivalent to a class derived from OnMethodBoundaryAspect where only the method OnEntry(MethodExecutionArgs) has been overwritten. The example is a method-level aspect and Self-Pointcut 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.
        }
    }
}
```

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 OnMethodBoundaryAspect OnEntry(MethodExecutionArgs). 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 ILocationValidationAspect T aspect interface. It validates values assigned to their targets and throws an exception in case of error.

Advice Type	Targets	Description
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 at page 114) 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.NonAbstrac
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 T**) of objects that are type-compatible with the kind of elements of code to which the *advice* applies.

For instance, the following code applies the `OnPropertySet` advice to all writable properties that are not annotated with the `IgnorePropertyChanged` custom attribute.

```
private IEnumerable<PropertyInfo> SelectProperties( Type type )
{
    const BindingFlags bindingFlags = BindingFlags.Instance |
        BindingFlags.DeclaredOnly | BindingFlags.Public;

    return from property
             in type.GetProperties( bindingFlags )
           where property.CanWrite && !property.IsDefined(typeof(IgnorePropertyChanged))
           select property;
}

[OnLocationSetValueAdvice, MethodPointcut( "SelectProperties" )]
public void OnPropertySet( LocationInterceptionArgs args )
{
    // Details skipped.
}
```

As you can see in this example, pointcut methods can use the power of LINQ to query System.Reflection.

Self Pointcut

The pointcut type SelfPointcut simply selects the target of the aspect.

Grouping Advices

The table of above shows advice types grouped in families. Advices of different type but of the same family can be grouped into a single logical *filter*, so they are considered as single transformation.

Why Grouping Advices

Consider for instance three advices of the family OnMethodBoundaryAspect: OnMethodEntryAdvice, OnMethodSuccessAdvice and OnMethodExceptionAdvice. The way how they are ordered is important, as it results in 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.

```
void Method()
{
  try
  {
    OnEntry();

    try
    {
      // Original method body.
    }
    finally
    {
      OnExit();
    }
  }
  catch
  {
     OnException();
     throw;
  }
}
```

```
void Method()
{
  OnEntry();

  try
  {
      // Original method body.
  }
  catch
  {
     OnException();
     throw;
  }
  finally
  {
      OnExit();
  }
}
```

The code in the left column may make sense in some situations, but it is not consistent with the code generated by OnMethodBoundaryAspect. Note that the advices may have been ordered differently: the order `OnEntry`, `OnException`, `OnExit` would have generated the same code as in the right column. However, you would have had to use custom attributes to specify order relationships between advices (see Ordering Advices at page 205). 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 at page 206).

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 at page 187 and one pointcut custom attribute (see Available Pointcuts at page 188), 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)
{
}
```

See Also

Reference

PostSharp.Aspects.Advices

5.5.2. 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 at page 192
- Introducing methods at page 194
- Introducing properties at page 195
- Controlling the visibility of introduced members at page 196
- Overriding members or interfaces at page 197

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 [SerializableAttribute].

> **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 **[Microsoft.TeamFoundation.TestManagement.Client.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; }
}
```

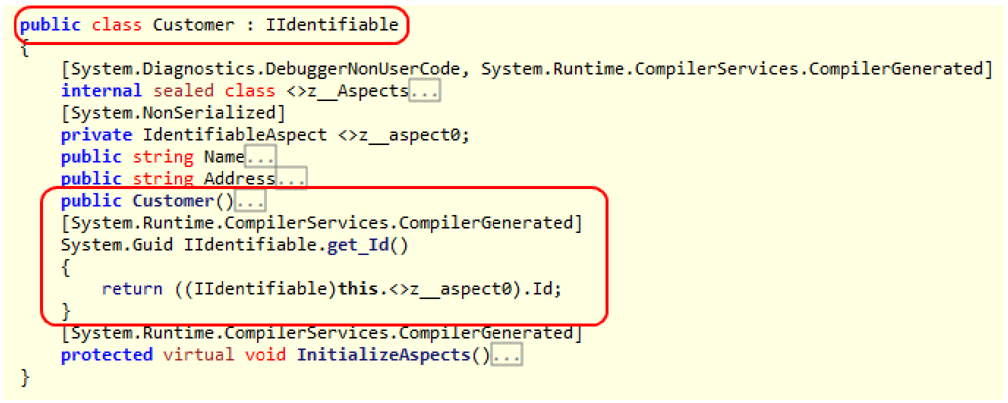
4. Add the IntroduceInterfaceAttribute attribute to the aspect and include the interface type that you want to add to the target code.

```
[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 Cast SourceType, TargetType (SourceType) pseudo-operator. The Cast SourceType, 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 instace 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 constucts 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 at page 184.

Introducing methods

The introduction of methods to your target code is very similar to introducing interfaces. The biggest difference is that you will be introducing code at a much more granular level.

1. Create an aspect that inherits from InstanceLevelAspect and add the custom attribute [SerializableAttribute].

2. Add to the aspect the method you want to introduce to the target code.

```
[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.

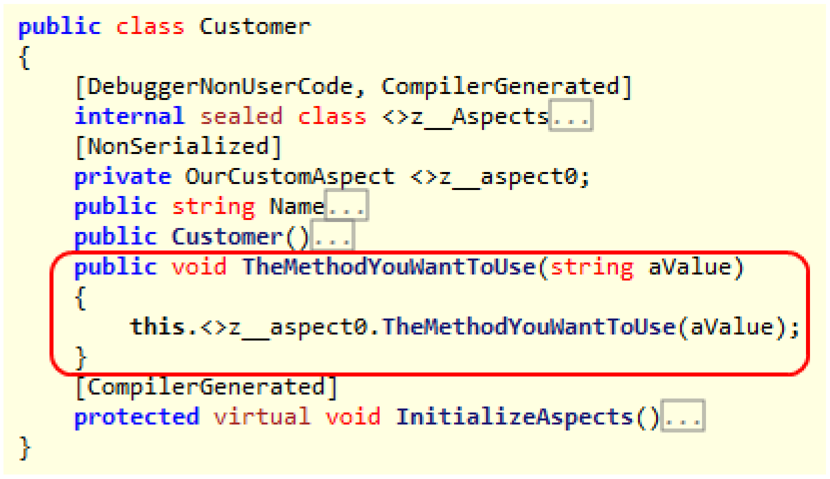
3. Decorate the method with the IntroduceMemberAttribute attribute.

```
[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.



```
public class Customer
{
    [DebuggerNonUserCode, CompilerGenerated]
    internal sealed class <>z__Aspects...
    [NonSerialized]
    private OurCustomAspect <>z__aspect0;
    public string Name...
    public Customer()...
    public void TheMethodYouWantToUse(string aValue)
    {
        this.<>z__aspect0.TheMethodYouWantToUse(aValue);
    }
    [CompilerGenerated]
    protected virtual void InitializeAspects()...
}
```

Introducing properties

The introduction of properties is almost exactly the same as the introduction of methods. Like introducing a method you will use the IntroduceMemberAttribute attribute. Let's take a look at the details.

1. Create an aspect that inherits from InstanceLevelAspect and add the custom attribute [SerializableAttribute].

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.

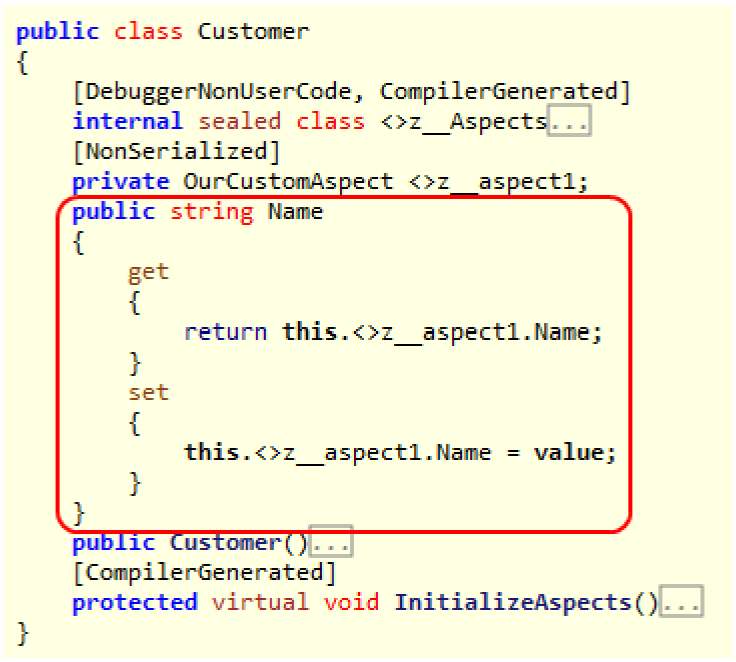
3. Decorate the property with the IntroduceMemberAttribute attribute.

```
[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. PostSharp allows you to control the visibility of the introduced member through the use of the Visibility property on the aspect. To declare that a member should be introduced with private visibility, all you have to do is declare it as such.

```
[IntroduceMember(Visibility = Visibility.Private)]
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 IsVirtual 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 Override-Action 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 OverrideOr-Ignore. 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; }
```

See Also

Reference

CreateImplementationObject(AdviceArgs)

CompositionAspect

Cast SourceType, TargetType (SourceType)

INotifyPropertyChanged

IDisposable

InstanceLevelAspect

PSerializableAttribute

SerializableAttribute

IntroduceInterfaceAttribute

IntroduceMemberAttribute

Visibility
IsVirtual
OverrideAction
Fail
Ignore
OverrideOrFail
OverrideOrIgnore
GetPublicInterfaces(Type)
CreateImplementationObject(AdviceArgs)
CreateInstance(Type, ActivatorSecurityToken)

5.5.3. 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

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 Func TResult . The delegate signature should exactly match the signature of the imported method.
Property	Property TValue , where the generic argument is the type of the property.
Collection Indexer	Property TValue, 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	Event TDelegate , 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 (Property TValue Get, Property TValue Set; Event TDelegate Add, Event TDelegate Remove). These delegates can be invoked by the aspect as any delegate.

The property ImportMemberAttribute IsRequired 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 targetting the same class are sorted (see Coping with Several Aspects on the Same Target at page 201) and executed.

3. Advices of the same aspect are sorted and executed in the following order:
   a. Member imports which have the property ImportMemberAttribute Order set to `BeforeIntroductions`.
   b. Member introductions.
   c. Members imports which have the property ImportMemberAttribute Order 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 ImportMemberAttribute IsRequired is `true`, ignored otherwise (by default).
	Non-virtual member defined.	Member imported.
	Virtual member defined.	If ImportMemberAttribute Order 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 IntroduceMemberAttribute OverrideAction is `Ignore` or `OverrideOrIgnore`, otherwise fail (by default).
	Virtual member defined in a parent class	Introduce a new `override` method if the property IntroduceMemberAttribute OverrideAction 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 IntroduceMemberAttribute OverrideAction 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

Example: Raising an Event When the Object is Finalized at page 249

See Also

Reference

IntroduceMemberAttribute

ImportMemberAttribute

5.5.4. Adding Aspects Dynamically

Additionally to providing advices, an aspect can provide other aspects dynamically using IAspect-Provider. 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 at page 137.

See Also

Reference

IAspectProvider

5.6. 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 ProvideAspectRoleAttribute. 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.

Action	Description
`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, StandardR
[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`AsyncAttribute` 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 AspectConfiguration AspectPriority, see Configuring Aspects at page 234).

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

Do not confuse the property AspectPriority with AttributePriority. The latter determines in which several custom attributes of the same time 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.Exce
```

See Also

Reference

PostSharp.Aspects.Dependencies

5.6.1. Ordering Advices

The section Coping with Several Aspects on the Same Target at page 201 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 at page 185), 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 AspectDependency-Attribute.

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.

See Also

Reference

AdviceDependencyAttribute

5.7. Targeting Windows Phone, Windows Store or Silverlight

Starting from version 3, PostSharp supports the following platforms without any limitation:

- Silverlight 4, 5

- Windows Phone 7, 8
- Windows Store 8

These platforms are supported through the Portable Class Library (PCL). The PCL version of *PostSharp.dll* has version numbers ending in 3 (for instance, 3.0.20.3), whereas the .NET versions end in 9 (for instance, 3.0.20.9). NuGet should automatically add the right version of the library to your project.

When using the portable version of PostSharp, you should use the portable serializer instead of the .NET BinaryFormatter:

- Use PSerializableAttribute instead of SerializableAttribute.
- Use PNonSerializedAttribute instead of NonSerializedAttribute.

For more information about aspect serialization, see section Understanding Aspect Serialization at page 208

5.8. 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 LocationInterception-Aspect OnSetValue(LocationInterceptionArgs) invokes the method LocationInterceptionArgs Get-CurrentValue , 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	MethodInterceptionArgs Binding
LocationInterceptionAspect	ILocationBinding	LocationInterceptionArgs Binding
EventInterceptionAspect	IEventBinding	EventInterceptionArgs Binding

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.

See Also

Reference

LocationInterceptionAspect

MethodInterceptionAspect

EventInterceptionAspect

5.9. Understanding Aspect Serialization

As explained in section Understanding Aspect Lifetime and Scope at page 178, 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 at page 209
- Aspects without serialization at page 209

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 Aspect-Serializer 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 AspectConfiguration SerializerType of the configuration of this aspect class or instance.

See section Configuring Aspects at page 234 for details.

The following code shows how to choose the serializer type for an OnMethodBoundaryAspect:

```
[OnMethodBoundaryAspectConfiguration(SerializerType=typeof(MsilAspectSerializer))]
public sealed MyAspect : OnMethodBoundaryAspect
```

See Also

Reference

AspectSerializer

BinaryAspectSerializer

PortableAspectSerializer

MsilAspectSerializer

AspectConfiguration SerializerType

5.10. 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 at page 210 explains how to test the behavior of an aspect.
- Testing that an Aspect has been Applied at page 212 shows how to test that an aspect has been applied to the expected set of code artifacts.
- Consuming Dependencies from the Aspect at page 213 describes several ways for aspects to consume dependencies from dependency-injection containers and service locators.
- Attaching a Debugger at Build Time at page 232 explains how to debug build-time logic.

5.10.1. 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 of 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 at page 239. High code coverage is achieved by varying the target code and testing with null and non-null parameters.

```
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++;
        }
    }
}
```

5.10.2. 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 at page 210.

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 at page 129.

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

The PostSharp symbol file format is undocumented and unsupported. It means that PostSharp support team cannot answer questions related to this file format.

5.10.3. 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:

5.10.3.1. 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 ThreadStaticAttribute to make the container local to the current thread is not a reliable approach.

Important

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

```
using System;
using System.ComponentModel.Composition;
using System.ComponentModel.Composition.Hosting;
using System.ComponentModel.Composition.Primitives;
using System.Reflection;
using PostSharp.Aspects;
using PostSharp.Extensibility;

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
        // containing target methods. This is why we cannot use the aspect in the Program
        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);
        }
    }
}
```

```
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();
        }
    }
}
```

5.10.3.2. 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 PostSharpEnvironment IsPostSharpRunning 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 **ExportProvider-GetExport** .

Important

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 source code demonstrates how to call a global service locator from the aspect static constructor.

```
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);
        }
    }
}
```

```
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()
        {
            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();
        }
    }
}
```

5.10.3.3. 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. Althought 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

```
using System;
using System.ComponentModel.Composition;
using System.ComponentModel.Composition.Hosting;
using System.ComponentModel.Composition.Primitives;
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);
        }
    }
}
```

```
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();
        }
    }
}
```

5.10.3.4. 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 catalogue.

Example: testable logging aspect with contextual dependency resolution

```
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
        {
            return new Lazy<T>(() => GetServiceImpl<T>(aspectType, targetElement));
        }

        private static T GetServiceImpl<T>(Type aspectType, MemberInfo targetElement) where T
        {
            // 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;
        }

        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);
        }
    }
}
```

```
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 TestLogg
                )
            {
                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();
        }
    }
}
```

5.10.3.5. 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 IInstanceScopedAspect interface. See Understanding Aspect Lifetime and Scope at page 178 for more information about aspect scopes.

Instance-scoped aspects can consume dependencies from the objects to which they are applied. They can also add dependencies to the target objects.

For instance, an aspect can consume a service `ILogger` using the following procedure:

To consume a service from an instance-scoped aspect:

1. Add a public property of name `Logger` and type `ILogger` to the aspect and add the IntroduceMemberAttribute custom attribute. This will cause the aspect to add a property to the target class. Use the parameter `MemberOverrideAction.Ignore` to ignore the property if it already exists in the target type 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

```
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
    [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);
        }
    }
}
```

```
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
    {
        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();
        }
    }
}
```

5.10.4. 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 at page 232). 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[7]). 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.

7. 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.

5.10.5. 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.

5.11. Advanced

5.11.1. 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 **FormatterServicesGetUnitializedObject(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 **FormatterServicesGetUnitializedObject(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 Binary-Formatter 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 AspectUtilities InitializeCurrentAspects . A call to this method will be translated into a call to `InitializeAspects`. It has to be invoked from a non-static method of an enhanced class.

If the class from which InitializeCurrentAspects is invoked has not been enhanced by an aspect requiring initialization, the call to this method is simply ignored.

Note

Using this approach may be brittle in some situations: calls to InitializeCurrentAspects will have no effect if aspects are applied to derived classes, but not to the calling class. In this scenario, it is preferable to define the method `InitializeAspects`.

5.11.2. 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 OnExceptionAspectConfiguration-Attribute.

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 IAspectBuildSemantics GetAspectConfiguration(Object). This method should return a configuration object, derived from the class AspectConfiguration. Every aspect class has its own aspect configuration class. For instance, the configuration attribute of the aspect class OnExceptionAspect is OnExceptionAspectConfiguration. The aspect type OnExceptionAspect implements IAspectBuildSemantics GetAspectConfiguration(Object) by creating an instance of OnExceptionAspectConfiguration, then it invokes the method OnExceptionAspect GetExceptionType(MethodBase) and copies the return value of this method to the property OnExceptionAspectConfiguration ExceptionType. 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 IAspectBuildSemantics GetAspectConfiguration(Object). If your aspect does not derive from the aspect class OnExceptionAspect, but directly implements the aspect interface IOnExceptionAspect, you can use only the later method.

5.12. Examples

5.12.1. 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 Runtime-Initialize(MethodBase) to improve runtime performance.

Example

```
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 runtim
        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 targ
        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.

5.12.2. 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

```
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)
        {
            // Display the error dialog with a parent window.
            MessageBox.Show(window, message, this.Caption, MessageBoxButton.OK, MessageBox
        }
        else
        {
            // Display the error dialog without a parent window.
            MessageBox.Show(message, this.Caption, MessageBoxButton.OK, MessageBoxImage.Er
        }

        // Do not rethrow the exception.
        args.FlowBehavior = FlowBehavior.Return;
    }
    }
}
```

5.12.3. 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 Flow-Behavior to skip the execution of a method when its value is found in cache. The property Method-ExecutionTag is used to store the cache key between the **OnEntry** and **OnSuccess** advices.

Example

```
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 alloc
            if (instance == null && arguments.Count == 0)
                return this.methodName;

            // Add all arguments to the cache key. Note that generic arguments are not part of
```

```
                // key, so method calls that differ only by generic arguments will have conflictin
                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, Aspect-Info). 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.

See Also

Reference

OnMethodBoundaryAspect

MethodExecutionArgs FlowBehavior

MethodLevelAspect CompileTimeInitialize(MethodBase, AspectInfo)

5.12.4. 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

```
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 = MulticastAttrib
    [AttributeUsage(AttributeTargets.Assembly | AttributeTargets.Class | AttributeTargets.Meth
    public class GuiThreadAttribute : MethodInterceptionAspect
    {
        public bool Asynchronous { get; set; }

        // Method invoked at build time. It validates that the aspect has been applied to an a
        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 memb
                              "derived from DispatcherObject.", method);
                return false;
            }

            // If the call is asynchronous, there should not be any return value or ByRef para
            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} b
                                  "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 Act
                }
                else
                {
                    dispatcherObject.Dispatcher.Invoke(DispatcherPriority.Normal, new Action(a
                }
            }
        }
    }
}
```

See Also

Reference

MethodInterceptionAspect

CompileTimeValidate(MethodBase)

5.12.5. 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

```
using System;
using Microsoft.Win32;
using PostSharp.Aspects;

namespace Samples
{
    public sealed class RegistryValueAttribute : LocationInterceptionAspect, IInstanceScopedAs
    {
        // True if the the value has already been fetched from registry and stored in the fiel
        // 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.DefaultVal


                // 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
    }
}
```

See Also

Reference

LocationInterceptionAspect

IInstanceScopedAspect

5.12.6. 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();
                }
            }
        }
    }
```

See Also

Reference

EventInterceptionAspect

5.12.7. 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");
        }
    }
}
```

See Also

Reference

CompositionAspect

5.12.8. 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 `NotDataMemberAttribute`.

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.
            CustomAttributeIntroductionAspect introduceDataMemberAspect =
                new CustomAttributeIntroductionAspect(
                    new ObjectConstruction(typeof (DataMemberAttribute).GetConstructor(Type.Em


            // 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 | Bin
            {
                if (property.CanWrite && !property.IsDefined(typeof (NotDataMemberAttribute),
                    yield return new AspectInstance(property, introduceDataMemberAspect);
            }
        }
    }

    [AttributeUsage(AttributeTargets.Property)]
    public sealed class NotDataMemberAttribute : Attribute
    {
    }
}
```

See Also

Reference

CustomAttributeIntroductionAspect

IAspectProvider

5.12.9. 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
    public class ObservableLifetimeAttribute : InstanceLevelAspect, IObservableLifetime
    {
        // True if the aspect instance is a 'real' instance, false if it is the prototype inst
        // We do not want to raise the Finalize event on prototype instances.
        [NonSerialized] private bool notPrototype;

        // 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 o
        [ImportMember("Dispose", IsRequired = true, Order = ImportMemberOrder.BeforeIntroducti
            BaseDisposeMethod;

        // Overrides the method Dispose(bool) of the target type.
        [IntroduceMember(IsVirtual = true, OverrideAction = MemberOverrideAction.OverrideOrFai
            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;

        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 `BaseDisposeMethod`, 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.

See Also

Reference

InstanceLevelAspect

ImportMemberAttribute

IntroduceMemberAttribute

IntroduceInterfaceAttribute

Other Resources

[e4599448-160c-4d56-8a75-f255195fcd17]

5.12.10. More Examples

You can find more examples and tutorials on our web site[8].

8. http://www.postsharp.net/support

CHAPTER 6

Enforcing Design Rules

Besides aspect-oriented programming, you can use PostSharp to validate your source code against architecture and design rules named *constraints*. Constraints are piece of codes that validate the code against specific rules at build time.

PostSharp provides ready-made constraints for the following scenarios:

- Restricting Interface Implementation at page 253
- Controlling Component Visibility Beyond Private and Internal at page 258

Additionally, you can develop custom constraints to enforce your own design rules. For details, see Developing Custom Architectural Constraints at page 270.

6.1. 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 artefact is required, the programming language does not give you any option to enforce the desired restriction. Enter the InternalImplementAttribute from PostSharp.

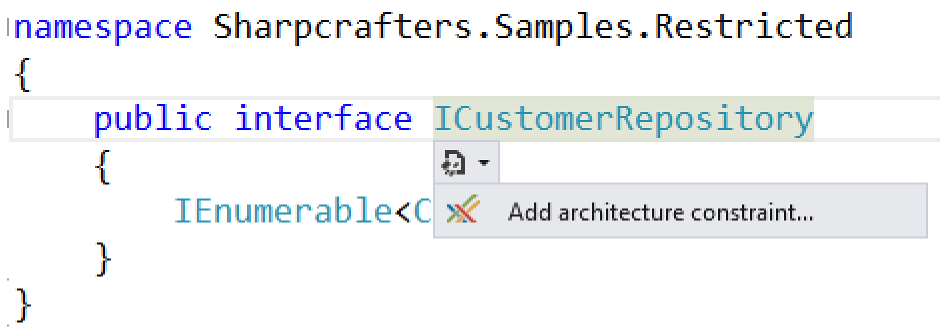
This topic contains the following sections.

- Adding the constraint to the interface at page 254
- Emitting an error instead of a warning at page 257
- Ignoring warnings at page 257

Adding the constraint to the interface

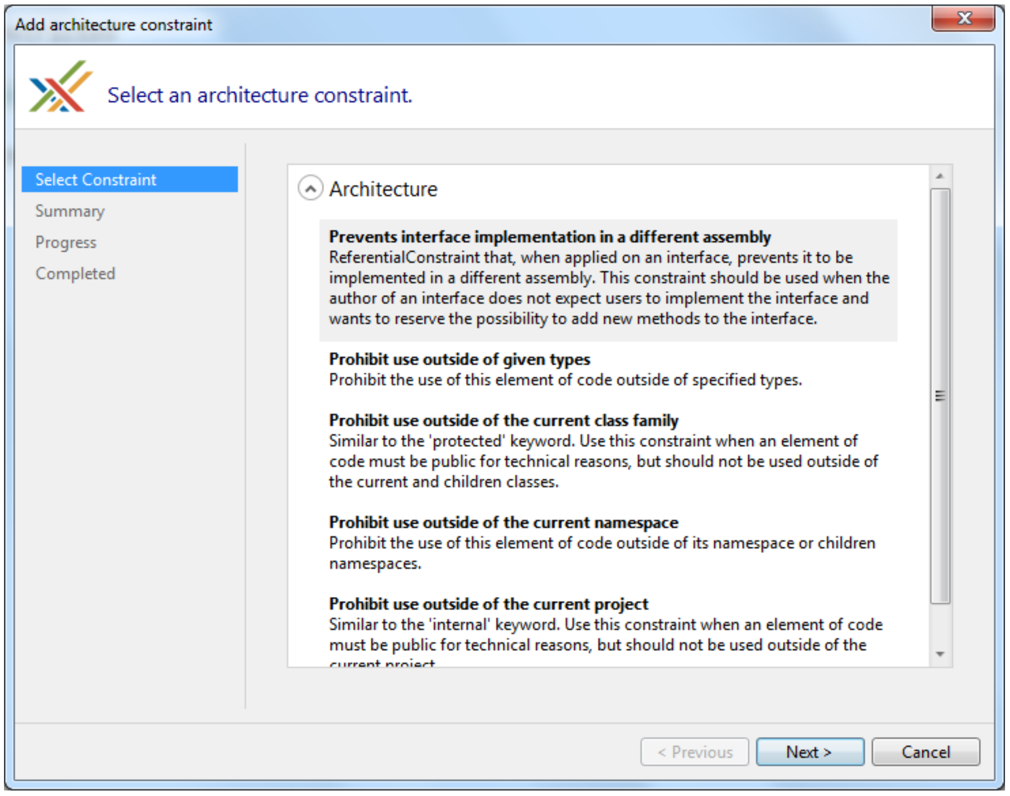
To restrict implemention of publically declared interfaces you simply need to add [InternalImplement-Attribute] to that interface.

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
    }
}
```

2. Select "Prevent interface implementation in a different assembly" and select **Next**.



Add architecture constraint

Select an architecture constraint.

Architecture

Prevents interface implementation in a different assembly
ReferentialConstraint that, when applied on an interface, prevents it to be implemented in a different assembly. This constraint should be used when the author of an interface does not expect users to implement the interface and wants to reserve the possibility to add new methods to the interface.

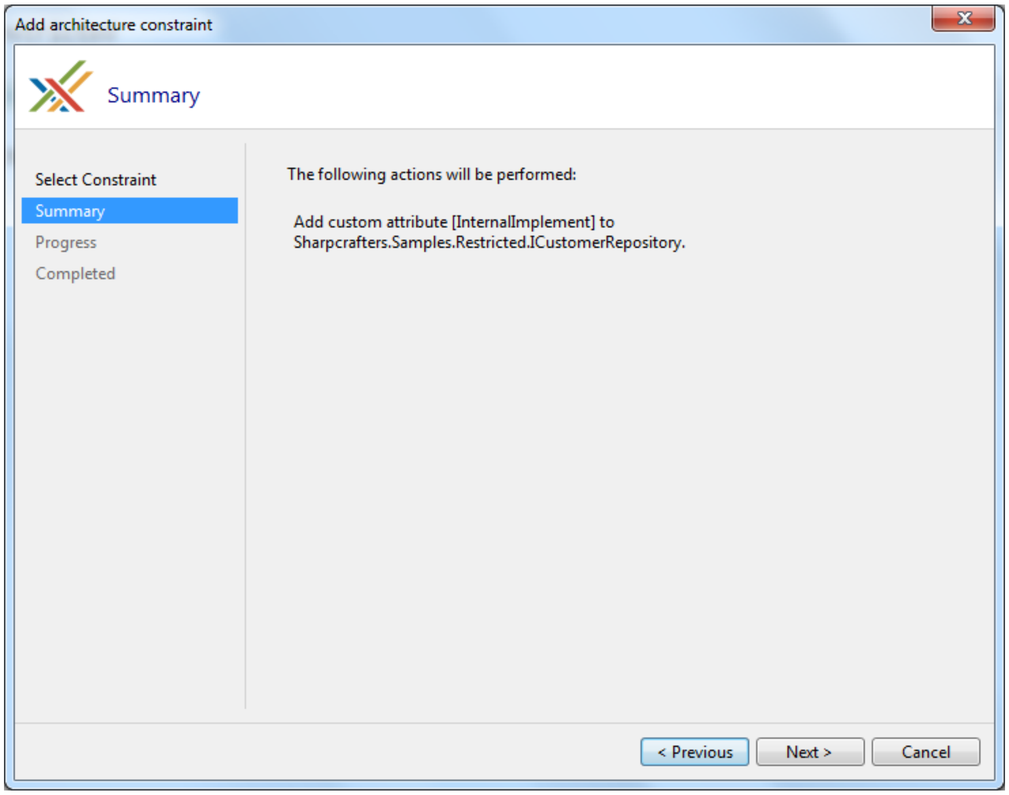
Prohibit use outside of given types
Prohibit the use of this element of code outside of specified types.

Prohibit use outside of the current class family
Similar to the 'protected' keyword. Use this constraint when an element of code must be public for technical reasons, but should not be used outside of the current and children classes.

Prohibit use outside of the current namespace
Prohibit the use of this element of code outside of its namespace or children namespaces.

Prohibit use outside of the current project
Similar to the 'internal' keyword. Use this constraint when an element of code must be public for technical reasons, but should not be used outside of the

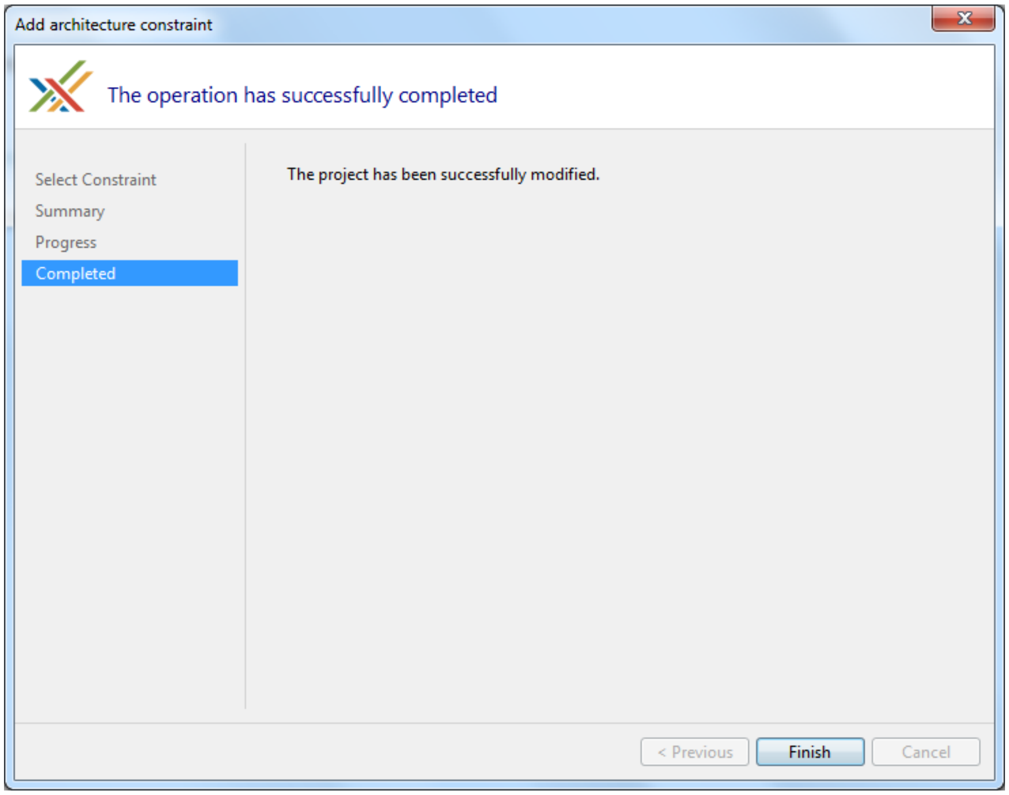
3. Verify that you will be adding the InternalImplementAttribute attribute to the correct piece of code.



The following actions will be performed:

Add custom attribute [InternalImplement] to Sharpcrafters.Samples.Restricted.ICustomerRepository.

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.



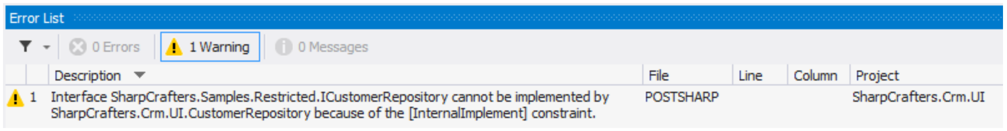
The operation has successfully completed

The project has been successfully modified.

5. You'll notice that the only thing that has changed in the code is the addition of the [Internal-ImplementAttribute] 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.



	Description	File	Line	Column	Project
1	Interface SharpCrafters.Samples.Restricted.ICustomerRepository cannot be implemented by SharpCrafters.Crm.UI.CustomerRepository because of the [InternalImplement] constraint.	POSTSHARP			SharpCrafters.Crm.UI

Note

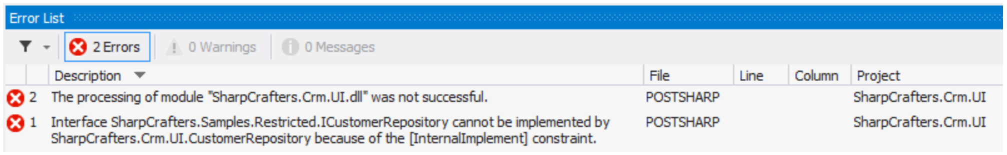
To perform this architectural validation the project that is trying to implement the interface will need to be processed by PostSharp.

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.



	Description	File	Line	Column	Project
2	The processing of module "SharpCrafters.Crm.UI.dll" was not successful.	POSTSHARP			SharpCrafters.Crm.UI
1	Interface SharpCrafters.Samples.Restricted.ICustomerRepository cannot be implemented by SharpCrafters.Crm.UI.CustomerRepository because of the [InternalImplement] constraint.	POSTSHARP			SharpCrafters.Crm.UI

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 IgnoreWarning-Attribute 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 supress 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;
    }
}
```

See Also

Reference

InternalImplementAttribute

IgnoreWarningAttribute

6.2. 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.

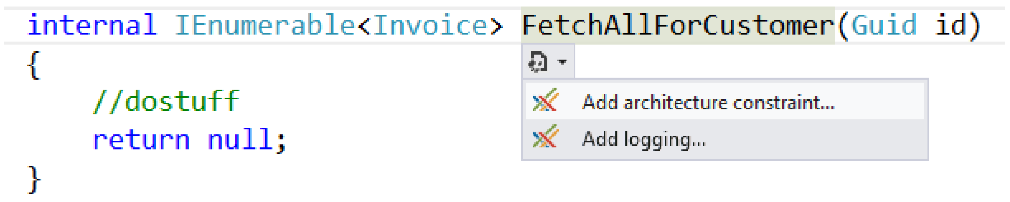
As an example let's look at a data access related class. As a precaution against developer's circumventing our data access structure we want to limit access to this repository class.

This topic contains the following sections.

Restricting access to specific namespaces

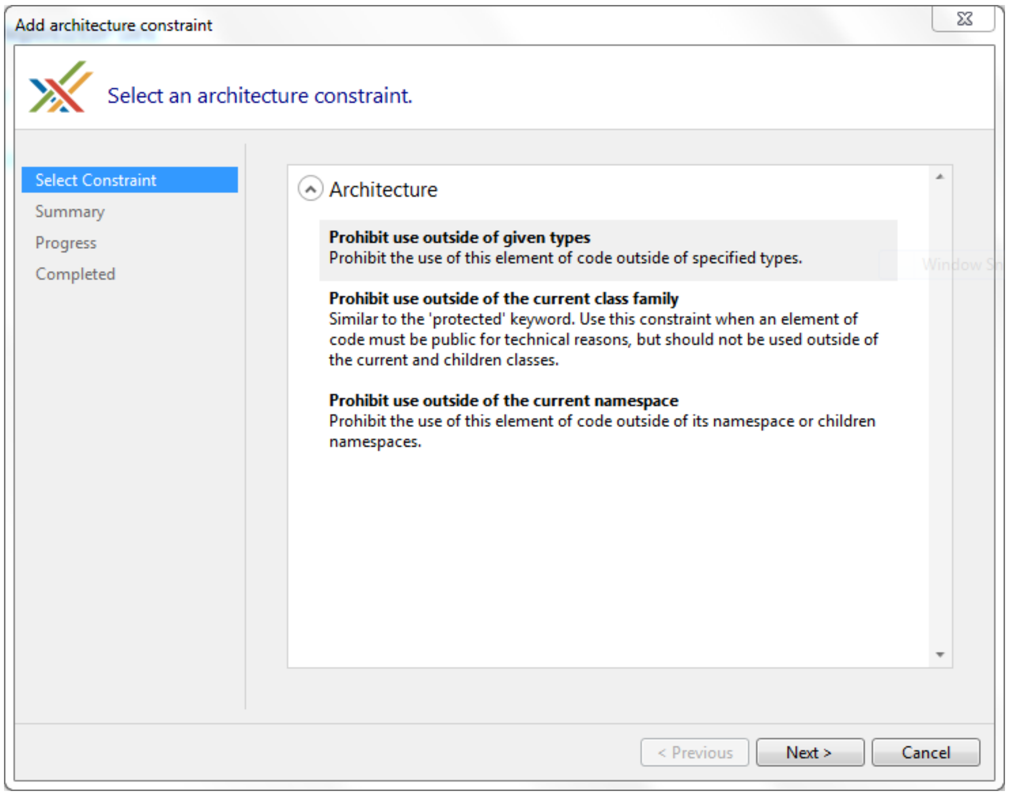
Our first step is to limit access to this class only to other classes within the validation namespace.

1. Put the caret on the `internal` class that should have restricted access. Select "Add architectural constraint..." from the smart tag options.



```
internal IEnumerable<Invoice> FetchAllForCustomer(Guid id)
{
    //dostuff
    return null;
}
```

2. Select "Prohibit use outside of given types" from the list of options.

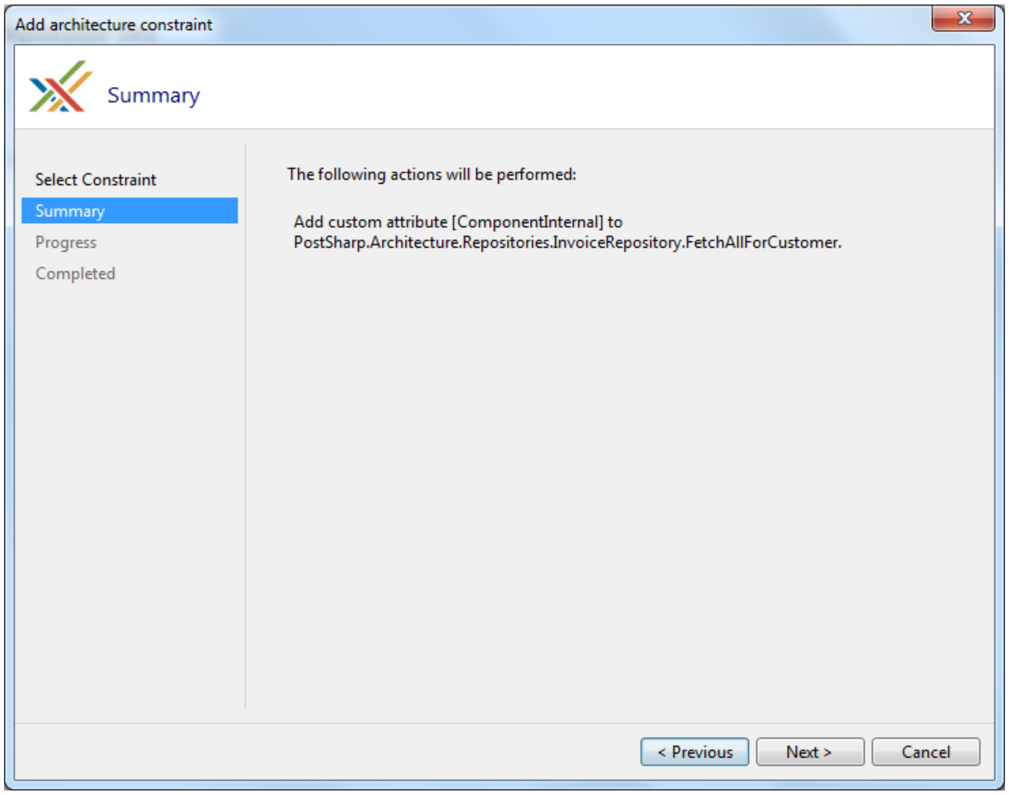


Prohibit use outside of given types
Prohibit the use of this element of code outside of specified types.

Prohibit use outside of the current class family
Similar to the 'protected' keyword. Use this constraint when an element of code must be public for technical reasons, but should not be used outside of the current and children classes.

Prohibit use outside of the current namespace
Prohibit the use of this element of code outside of its namespace or children namespaces.

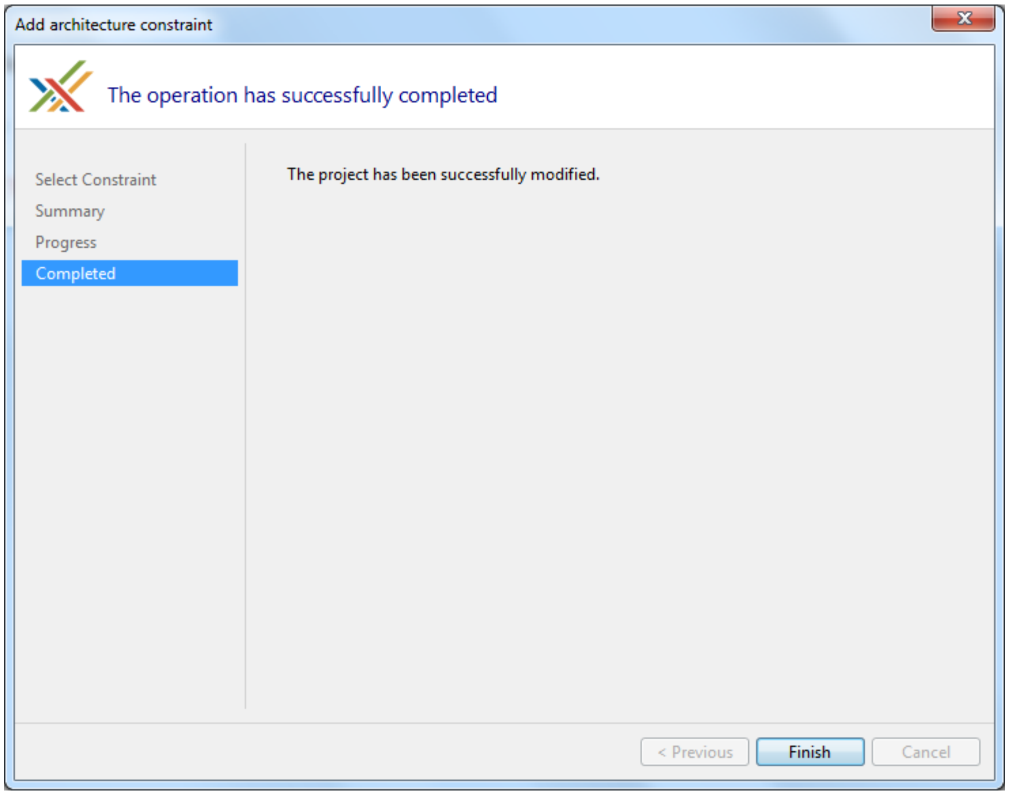
3. Verify that you will be adding the ComponentInternalAttribute attribute to the correct piece of code.



The following actions will be performed:

Add custom attribute [ComponentInternal] to PostSharp.Architecture.Repositories.InvoiceRepository.FetchAllForCustomer.

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.


Add architecture constraint
The operation has successfully completed
Select Constraint
Summary
Progress
Completed
< Previous
Finish
Cancel

The project has been successfully modified.

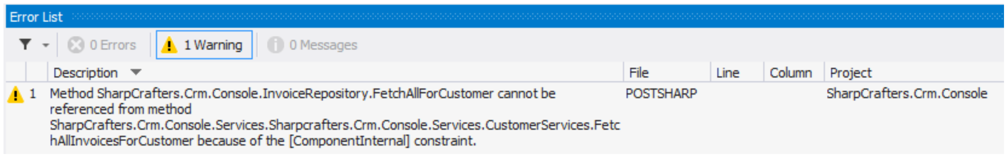
5. You'll notice that the only thing that has changed in the code is the addition of the [ComponentInternalAttribute] attribute.

```
namespace Sharpcrafters.Crm.Console.Repositories
{
    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.

```
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
                           });
          }
      }
}
```



Error List

0 Errors | 1 Warning | 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

Note

If you are trying to access the component from a namespace that is in a different project you will need PostSharp to process that project for the validation to occur.

Restricting access to specific types

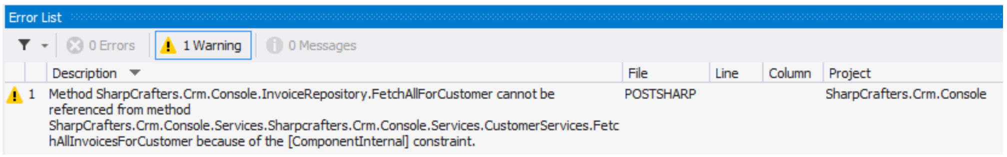
Under some circumstances namespace level restrictions may not be tight enough for your needs. In that situation you have the ability to apply this constraint at a type level.

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.

```
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());
    }
}
```



	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

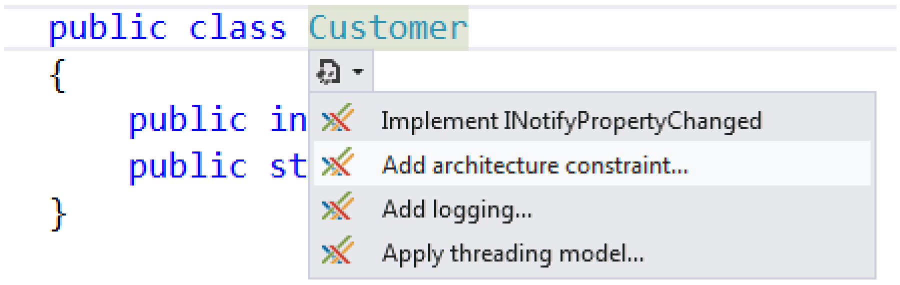
Because of framework limitations or automated testing requirements you sometimes need to declare components as public so that you can perform the desired tasks or testing. For some of those components you probably don't want external applications accessing them. For instance, WPF controls need a default 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; }
    }
}
```

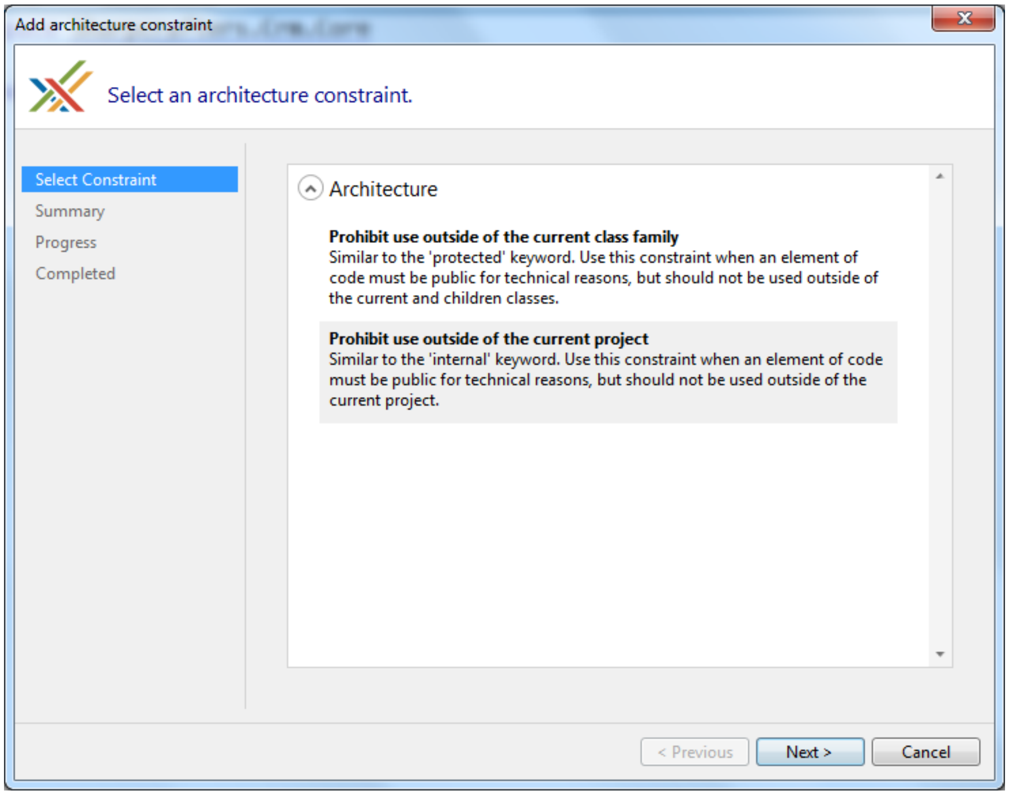
2. Place the caret on the publically declared component that you want to restrict external access to and expand the smart tag. Select "Add architectural constraint".



```
public class Customer
{
    public in
    public st
}
```

Implement INotifyPropertyChanged
Add architecture constraint...
Add logging...
Apply threading model...

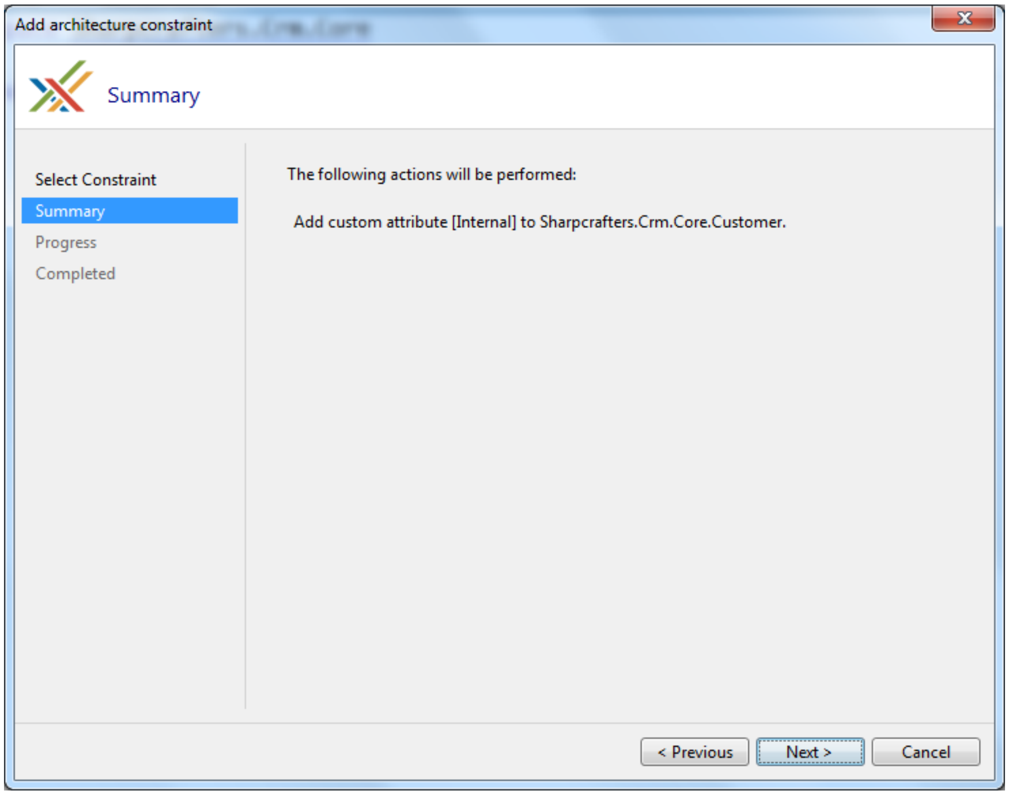
3. When prompted to select a constraint, choose to "Prohibit use outside of the project".



Prohibit use outside of the current class family
Similar to the 'protected' keyword. Use this constraint when an element of code must be public for technical reasons, but should not be used outside of the current and children classes.

Prohibit use outside of the current project
Similar to the 'internal' keyword. Use this constraint when an element of code must be public for technical reasons, but should not be used outside of the current project.

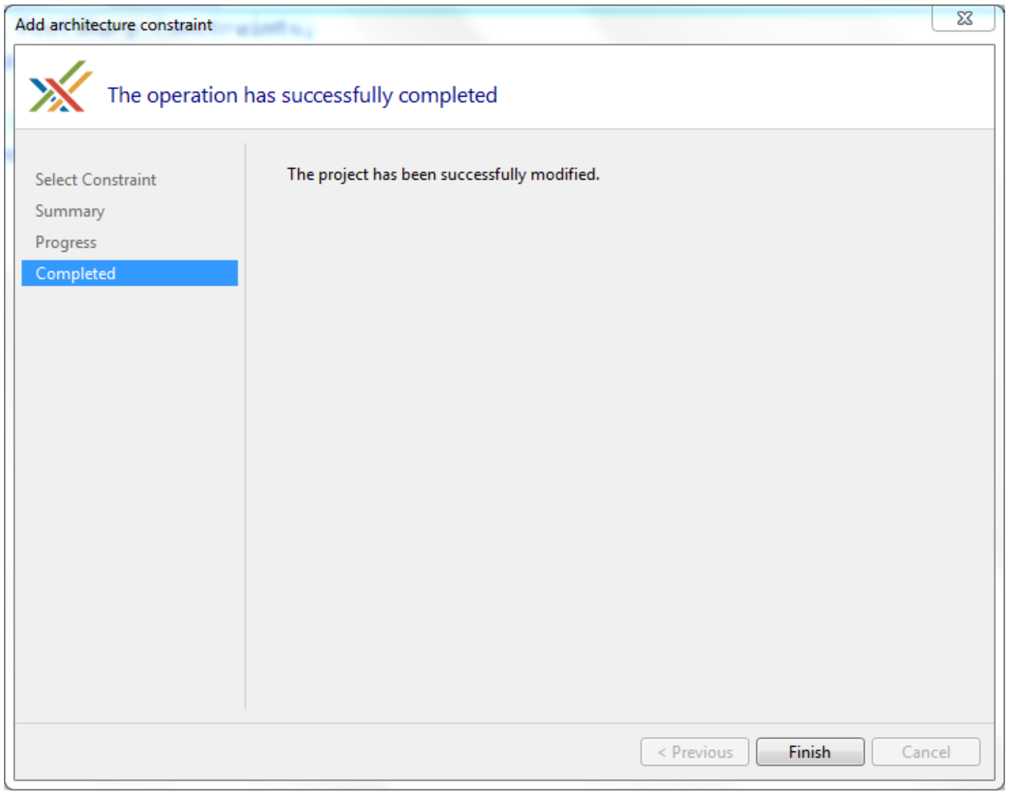
4. The summary page gives you the opportunity to review the selections that you have made. If you notice that the configuration is not what you wanted you can click the **Previous** button and adjust your selections. If the configuration meets your needs click **Next**. In this demo you will see that the [InternalAttribute] attribute is being added to the Customer class.



The following actions will be performed:

Add custom attribute [Internal] to Sharpcrafters.Crm.Core.Customer.

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.



The operation has successfully completed

The project has been successfully modified.

6. You'll notice that the only thing that has changed in the code is the addition of the [Internal-Attribute] 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.

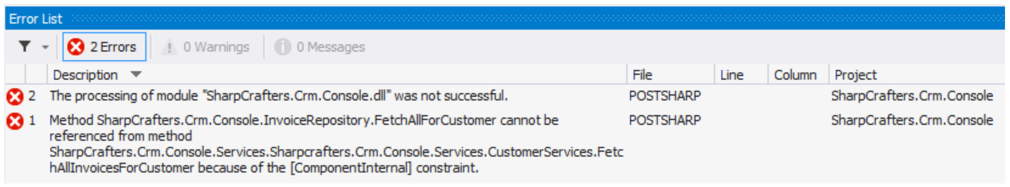
Emitting errors instead of warnings

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.



```
   Description                                                                File        Line  Column  Project
2  The processing of module "SharpCrafters.Crm.Console.dll" was not successful.  POSTSHARP                 SharpCrafters.Crm.Console
1  Method SharpCrafters.Crm.Console.InvoiceRepository.FetchAllForCustomer cannot be  POSTSHARP             SharpCrafters.Crm.Console
   referenced from method
   SharpCrafters.Crm.Console.Services.Sharpcrafters.Crm.Console.Services.CustomerServices.Fetc
   hAllInvoicesForCustomer because of the [ComponentInternal] constraint.
```

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 IgnoreWarningAttribute 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());
    }
}
```

See Also

Reference

IgnoreWarningAttribute

ComponentInternalAttribute

InternalAttribute

6.3. Developing Custom Architectural Constraints

When you are creating your applications it is common to adopt custom design patterns that must be respected accross 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 MulticastAttributeUsageAttribute attribute. In this case we want the validation to occur on types only, and we want to enable inheritance.

```
[MulticastAttributeUsage(MulticastTargets.Class, Inheritance = MulticastInheritance.St
public class BusinessRulePatternValidation : ScalarConstraint
{
}
```

3. Override the ValidateCode(Object) method.

```
[MulticastAttributeUsage(MulticastTargets.Class, Inheritance = MulticastInheritance.St
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.St
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`, `MethodInfo`, `ConstructorInfo`, `PropertyInfo`, `EventInfo`, `FieldInfo`, and `ParameterInfo`.

5. Write a warning that the rule being broken to the Output window in Visual Studio.

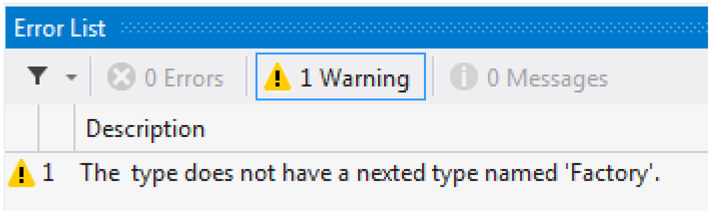
```
[MulticastAttributeUsage(MulticastTargets.Class, Inheritance = MulticastInheritance.St
public class BusinessRulePatternValidation : ScalarConstraint
{
    public override void ValidateCode(object target)
    {
        var targetType = (Type)target;

        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 at page 116

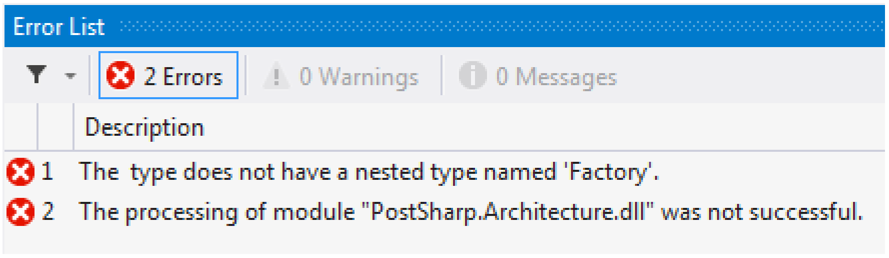
7. Now if you compile the project you will see an error in the Output window of Visual Studio when you run a build.



```
1  The type does not have a nexted type named 'Factory'.
```

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.St
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);
        }
    }
}
```



Error List

2 Errors | 0 Warnings | 0 Messages

	Description
1	The type does not have a nested type named 'Factory'.
2	The processing of module "PostSharp.Architecture.dll" was not successful.

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 MulticastAttributeUsage-Attribute attribute to the class.

```
[MulticastAttributeUsage(MulticastTargets.Class, Inheritance = MulticastInheritance.St
public class BusinessRuleUseValidation : ReferentialConstraint
{
}
```

3. Override the ValidateCode(Object, Assembly) method.

```
[MulticastAttributeUsage(MulticastTargets.Class, Inheritance = MulticastInheritance.St
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.St
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.St
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);
        }
    }
}
```

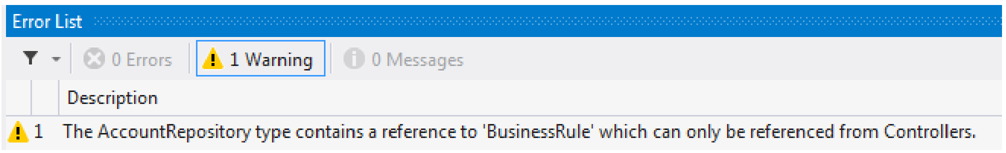
6. Attach the referential constraint that you created to any code that needs to be checked. In this example, add an attribute to the `AccountRepository` class.

```
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 at page 116.

7. Now when you compile the project you will see a warning in the Output window in Visual Studio.



```
1  The AccountRepository type contains a reference to 'BusinessRule' which can only be referenced from Controllers.
```

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

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 `BusinessRulePatternValidation` may be a valid constraint only on classes that exist in the `Models` namespace.

Let's look at how we can ensure that this constraint is only enforced on classes that exist in the `Models` namespace.

1. Open the `BusinessRulePatternValidation` class that you created earlier.
2. Override the ValidateConstraint(Object) method.
3. Write the validation logic to ensure that this constraint is only applied to classes in the `Models` namespace.

   Note

   When the ValidateConstraint(Object) method returns `true`, it tells PostSharp that the constraint should be applied to that target code element. When the Validate-Constraint(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.

See Also

Reference

MulticastAttributeUsageAttribute

ScalarConstraint

IgnoreWarningAttribute

MessageId

Reason

MulticastAttributeUsageAttribute

CHAPTER 7

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 at page 281
- Emitting Errors, Warnings, and Messages at page 282.

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.

7.1. 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 Configuring Post-Sharp at page 25 for details.

- By defining the `PostSharpDisabledMessages` or `PostSharpEscalatedMessages` MSBuild properties. See Configuring PostSharp at page 25 and Configurable MSBuild Properties at page 28 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 **IgnoreConstraintWarning-Attribute** 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.

7.2. 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 ome 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-neglictible performance penalty because of the cost of instantiating the `ResourceManager`.

CHAPTER 8

Combining with Other Technologies

8.1. 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 project file (a site is actually a directory), and these projects must not be built. Although they don't use MSBuild, you can theoretically still use PostSharp with ASP.NET site projects thanks to the project PostSharp4AspNet[9]. As explained on the home page of this project, using PostSharp with ASP.NET Site projects is not recommended and not officially supported.

See Also

Other Resources

PostSharp4AspNet[10]

8.2. ILMerge

⚠ **Caution**
The current version of ILMerge has bugs that prevent it from being used with assemblies processed by PostSharp.

9. http://postsharp4aspnet.codeplex.com/
10. http://postsharp4aspnet.codeplex.com/

You can use **ILMerge** to combine *PostSharp.dll*, or assemblies processed by PostSharp, into a large assembly. However, all assemblies enhanced by aspects must be made aware of this.

As explained in Understanding Aspect Lifetime and Scope at page 178, aspects are serialized at build time and deserialized at run time. Serialization data include the type name of aspects. If aspect classes are moved to a different assembly, the deserializer will complain that the aspect type does not exist any more.

Therefore, you have to make the serializer aware of the new name of assemblies. This can be achieved by tweaking the BinaryAspectSerializationBinder object.

```
((BinaryAspectSerializationBinder) BinaryAspectSerializer.Binder).Retarget("OldAssemblyName",
```

The aspect serializer must be configured before aspects are deserialized, i.e. before the first class affected by an aspect is accessed by the program.

Note

You can also provide your own implementation of SerializationBinder by setting the property BinaryAspectSerializer Binder.

8.3. Obfuscation Tools

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

Caution

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.

8.4. Microsoft Code Analysis (FxCop)

Code Analysis for managed code analyzes managed assemblies and reports information about the assemblies, such as violations of the programming and design rules set forth in the Microsoft .NET Framework Design Guidelines.

The analysis tool represents the checks it performs during an analysis as warning messages. Warning messages identify any relevant programming and design issues and, when it is possible, supply information about how to fix the problem.

Code Analysis in integrated into Visual Studio Premium or Visual Studio Ultimate. The same product is available for free and is known as **FxCop**; FxCop requires manual integration with your build scripts.

Code Analysis must process assemblies before they have been processed by PostSharp. Indeed, PostSharp may add internal and private members, or generate instructions, that do not comply to Code Analysis rules. However, the warnings you would get from analyzing post-processed assemblies are not likely to be relevant.

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.

If you are executing FxCop manually or by using custom MSBuild integration, you should process assemblies from directories *obj\CodeAnalysis\Before-PostSharp* instead of *bin\CodeAnalysis*.