SharpMedia Aspect Orientated Design

© 2007 Žiga Osolin – SharpMedia project, document version: DRAFT

Contents

[About 2](#_Toc174347783)

[Goals 2](#_Toc174347784)

[Deployment Structure 2](#_Toc174347785)

[Unresolved Issues 2](#_Toc174347786)

[Namespace Structure 3](#_Toc174347787)

[Parameter Validation 3](#_Toc174347788)

# About

SharpMedia Aspect Orientated is a collection of attributes that allow you to constumize your code using aspects. Aspect is applied as an attribute to certain element (method, class, property ...) and constumizes its behaviour by injecting code at post compilation step. We use PostSharp and PostSharp.Laos for AOP (Aspect Orientated Programming), with a custom plugin. Plugin is implemented as a part of »installation service«.

Read through PostSharp before continuing: http://www.postsharp.org/about

# Goals

The goal of AOP is to allow custom behaviour and veryfication that is injected in MSIL. The main aim is better productivity. Writing and managing attributes (especially on global level, using multi casting) instead of inline code is much simpler and faster (not in performance, though). What is more, attributes can change behaviour in future and this will effect all code that uses attributes.

One potencial drawback of AOP is performance. We use PostSharp because it is capable of doing analysis at compile time and at runtime. In future[[1]](#footnote-2), we expect to rewrite more and more attributes to use compile time facilities and gain runtime performance. We could also perform static analysis and omit some frequently used attributes when this is not needed.

# Deployment Structure

Code is deployed in SharpMedia core library because we think it is essential that all other assemblies have access to it (and make full use of it).

# Unresolved Issues

# Namespace Structure

The code is organized into **SharpMedia.AspectOrientated** namespace and its subnamespaces. The main namespace contains all of the generally usable aspect attributes. Subnamespaces contain code for aspect setup (usually with internal classes) and in some rare cases also not so generally usable aspects.

## Parameter Validation

A set of attributes are provided for static and dynamic parameter validation. For now, most of the code is injected and no static analysis is made but you can expect this to change in the future.

To enforce parameter validation, you add a specific attribute to the parameter, for example:

public static void Method

(

[NotNull] object parameter1,

[NotEmpty] object[] parameter2,

[MinInt(0), MaxInt (5)] int parameter3,

[InRange(1, 4)] List<object> parameter4

)

{

// Code ...

}

Attributes are specific for their types. You cannot use **NotEmpty** on integer – this will result in compile time (actually post-compile time) error. The order of argument validation is not specified.

**NOTE: You can add more than one attribute to the parameter.**

***GUIDELINES: You should always prefer using Aspect orientated parameter validation. If you need special configuration, try configuring the attribute first (attributes have many optional arguments) or if this does not work, create a new attribute.***

Some attributes are generic by their nature and can be applied to many different types. Because casting all those types to object (many times invloves boxing) and doing validation on this level is a loss of performance and knowledge of the type, we prefer specialized attributes. Because C# does not allow generic attributes, we specialize them by appending the type name after their name:

**MinAttribute + Int -> MinIntAttribute**

All specialized attributes extend the base attribute: **MinIntAttribute : MinAttribute**.

A list of available parameter validation attributes is defined here.

|  |  |
| --- | --- |
| Attribute Signature | Applies To |
| NotNull | All reference types |
| NotEmpty | To IList interface |
| NotEmptyString | To string |
| Min\*[[2]](#footnote-3) |  |
| Max\* |  |
| InRange(int min, int max) | To IList interface |
| NotEqual\* |  |
| Greater\* |  |
| GreaterEqual\* |  |
| Less\* |  |
| LessEqual\* |  |
| Positive |  |
| Negative |  |
| OneOf\*(\*[] all) |  |
| Regex | To string, performs regex validation |
| TypeConstantReturn | The return must not depend on any non-constant variable |
| ... |  |

### Properties

Applying attribute to properties is the same as for method. You can use the

**[param: Attribute]**   
set {...}

syntax to apply attributes to property variable.

### Abstract, interface methods

All attributes are inherited if certain method overrides other method. If you want to ignore base parameter attributes, apply

**[IgnoreBaseAspects]**

attribute to parameter or method. If you apply it to parameter, only that parameter will not inherit attributes, otherwise all parameters of method do not inherit attributes. Note that some parameters cannot ignore base aspects (if they are forced). Each AOP parameter can be forced by setting **ForceAttribute** named property to true (default is false).

All attributes can be multicasted with wildcards (this is PostSharp capability).

## Logging and Tracing

Logging facilities allow logging special strings on method joints. The **TraceAttribute** is available which will inject logging code. The logger uses **Common.Log** facilities to log messages. You can supress messages by either disabling tracing (in plugin compile) or by not forwarding log events on certain class (this is not performance wise).

## Exception handling

You can custumize method's error handling by cloacking the original exception with new type (and use the original exception as internal exception) or by providing custom cleanups (such as disposing). We provide the following handlers:

* **DisposeOnExceptionAttribute** – disposes self on exception and rethrows exception
* **CallOnExceptionAttribute** – calls other method on exception and rethrows (or not, if flag is set to no throw).
* **SuppressOnExceptionAttribute** – ignores the exception.
* **RetryOnExceptionAttribute** – retries the method on exception. Finnaly is executed before retrying is done
* **CloakOnExceptionAttribute** – cloaks the exception with new exception and no information about the exception itself (this is good for security resons).
* **LogOnExceptionAttribute** – performs logging on exception.

All exception handlers can be provided to methods only.

1. When PostSharp allows static code analysis and gives eve better compile time support. [↑](#footnote-ref-2)
2. The star indicates the type that is appended. For example, MinInt or MinString. [↑](#footnote-ref-3)