

PostSharp. Overview

Прочитать статью на сайте

Прежде чем перейти непосредственно к теме повествования, необходимо сказать несколько слов о системных вещах.

Аспектно-Ориентированное программирование

Аспектно-ориентированное программирование (АОП) — парадигма программирования, основанная на идее разделения функциональности для улучшения разбиения программы на модули.

Существует достаточно много классов задач, решение которых невозможно в контексте ООП или же приведет к большой захламленности кода и связности модулей, что плохо. Например, логирование. Для того чтобы вести лог работы программы, необходимо в каждый метод поместить несколько служебных строчек вызова службы логирования, так же возможно придется ее передавать через входные параметры методов. Весь этот код не будет относиться к выполнению реальной задачи возложенной на метод, а только мозолить глаза. К тому же, сколько лишних строчек кода придется написать вручную!

Еще примером может служить авторизация и проверка прав доступа. По-хорошему, перед началом выполнения важных методов, надо каждый раз проверять, имеет ли текущий пользователь права на запуск указанного метода. Тут тоже может быть очень много мороки и кода.

В целом, любой сквозной код выпадает из возможностей ООП. АОП программирование, наоборот, предоставляет все средства для выделения «сквозного кода» в отдельные сущности, что существенно упрощает код как для тестирования, так и для использования.

Основные понятия АОП:

- Аспект (англ. aspect) — модуль или класс, реализующий сквозную функциональность. Аспект изменяет поведение остального кода, применяя совет в точках соединения, определённых некоторым срезом.
- Совет (англ. advice) — средство оформления кода, который должен быть вызван из точки соединения. Совет может быть выполнен до, после или вместо точки соединения.
- Точка соединения (англ. join point) — точка в выполняемой программе, где следует применить совет. Многие реализации АОП позволяют использовать вызовы методов и обращения к полям объекта в качестве точек соединения.
- Срез (англ. pointcut) — набор точек соединения. Срез определяет, подходит ли данная точка соединения к данному совету. Самые удобные реализации АОП используют для определения срезов синтаксис основного языка и позволяют их повторное использование с помощью переименования и комбинирования.
- Внедрение (англ. introduction, введение) — изменение структуры класса и/или изменение иерархии наследования для добавления функциональности аспекта в инородный код. Обычно реализуется с помощью некоторого метаобъектного протокола (англ. metaobject protocol, MOP).

На этом можно перейти непосредственно к главной теме повествования.

PostSharp

PostSharp – это аспектно-ориентированный фреймворк, он привносит аспектно-ориентированное программирование на платформу .Net. С помощью PostSharp и АОП вы сможете выделить технические требования в отдельные сущности (аспекты).

- Технические требования реализовываются как простые C# классы.
- Аспекты применяются к бизнес-объектам без изменения их исходного кода.
- Во время сборки, аспекты внедряются в бизнес-объекты.

Рассматриваемый фреймворк позволяет указывать к каким типам данных (классам, сборкам) применять аспекты, как осуществлять наследование аспектов и как они могут взаимодействовать друг с другом, очередность и множество других нюансов использования.

На данный момент официальный релиз есть у версии 1.5, версия 2.0 находится в стадии СТР или уже сильно RC. В основном речь пойдет о версии 1.5, но о вкусностях 2.0 я тоже не буду забывать и сообщать, где это требуется.

Основные аспекты

OnMethodBoundary

Аспект можно создать наследованием от класса OnMethodBoundaryAspect или от интерфейса IOnMethodBoundaryAspect.

Данный аспект применяется, когда у вас есть доступ к исходным кодам метода и вы хотите отслеживать события начала вызова метода, конец вызова, исключение и успешность завершения. Все это делается переопределением методов OnEntry, OnExit, OnException и OnSuccess соответственно.

Схематически код после применения аспекта будет выглядеть так (естественно, это будет в закрамах IL сборки)

```
int MyMethod(object arg0, int arg1)
{
    OnEntry();
    try
    {
        // Original method body.

        OnSuccess();
        return returnValue;
    }
    catch ( Exception e )
    {
        OnException();
    }
    finally
    {
        OnExit();
    }
}
```

На самом деле все выглядит более сложно, но для понимания это отличная иллюстрация. Объект типа MethodExecutionEventArgs передается во все 4 метода, и с помощью указанного объекта вы можете читать и писать значения в передаваемые параметры, получать текущее сообщение об ошибке, переопределять возвращаемое значение и изменять течение процедуры.

Доступ к параметрам осуществляется с помощью методов `GetReadOnlyArgumentArray()` и `GetWritableArgumentArray()` класса `MethodExecutionEventArgs`. Прошу заметить, что изменяться могут только те значения, которые передаются по ссылке. По большому счету оба метода возвращают один и тот же набор объектов, но в будущих версиях это поведение может измениться.

OnFieldAccess

Аспект можно создать наследованием от класса `OnFieldAccessAspect` или от интерфейса `IOnFieldAccessAspect`.

Применение данного аспекта позволяет перехватывать все операции чтения/записи для поля класса. Этот аспект имеет 2 метода для переопределения: `OnGetValue` и `OnSetValue`. Оба этих метода получают в качестве параметра класс `FieldAccessEventArgs`, который имеет 2 интересующих нас свойства `StoredValue` и `ExposedValue`. Первое позволяет получить доступ к реальному значению в поле, второе же играет роль буфера, при чтении там будет объект который будет возвращен, при записи, объект который пришел для записи.

Другими словами, реализация по умолчанию для `OnGetValue`:

```
eventArgs.ExposedValue = eventArgs.StoredValue
```

И `OnSetValue` будет реализованно как:

```
eventArgs.StoredValue = eventArgs.ExposedValue
```

Более подробно о механизме действия аспекта этого будет в других статьях.

OnMethodInvocation

Аспект можно создать наследованием от класса `OnMethodInvocationAspect` или от интерфейса `IOnMethodInvocationAspect`.

Так же как и аспект `OnMethodBoundary`, рассматриваемый аспект позволяет добавить дополнительное поведение к методу. Разница в том, что этот аспект не модифицирует тело исходного метода, но перехватывает обращение и подменяет его собой, точнее тем, что будет написано в методе `OnInvocation`. Данный метод принимает в качестве параметра класс `MethodInvocationEventArgs` и самое интересное заключается в методе `Proceed`, который собственно и вызывает оригинальный метод. Реализация метода `OnInvocation` по умолчанию вызывает уже `Proceed`, таким образом не стоит еще раз вызывать этот метод если вы используете `base.OnInvocation`. И соответственно наоборот.

При работе с данным аспектом возможно получить доступ к исходному методу посредством свойства `Delegate`. Так же есть способ получить передаваемые параметры, вызвав метод `GetArgumentArray` у класса `MethodInvocationEventArgs`. Результатом будет массив объектов, которые можно менять на свой вкус. Итоговое значение надо записать в свойство `ReturnValue`.

```
eventArgs.ReturnValue = eventArgs.Delegate.DynamicInvoke( eventArgs.GetArgumentArray() );
```

OnException

Аспект можно создать наследованием от класса `OnExceptionAspect` или от интерфейса `IOnExceptionAspect`.

Данный аспект добавляет обработчик исключений в метод, к которому применен аспект, что позволяет заключать код обработки исключений в отдельный атрибут, а не размазывать это все по классам. В методе `GetExceptionType` можно определить какие типы исключений будут обрабатываться, если ничего не написать, то основной обработчик будет перехватывать все исключения. Сам же обработчик исключений будет в методе `OnException`. Пришедшее исключение будет доступно через свойство `Exception` (только для чтения) класса `MethodExecutionEventArgs`. Если нужно будет заменить исключение на какое-либо другое, надо создать новое исключение. Игнорирование исключений управляется через `ControlFlow`.

ImplementMethod

Аспект можно создать наследованием от класса `ImplementMethod` или от интерфейса `IImplementMethod`.

Этот аспект позволяет разделить объявление и реализацию абстрактных и внешних методов. После применения этого аспекта на метод, он перестает быть абстрактным и получает реализацию определенную в аспекте в переопределенном методе `OnExecution` с параметром класса `MethodExecutionEventArgs`. С помощью этого класса можно получить доступ к самому методу (свойство `Method`) и к его аргументам (метод `GetWritableArgumentArray()`). Задать выходное значение можно с помощью свойства `ReturnValue`.

Composition

С помощью данного аспекта можно внедрять новые интерфейсы в существующий тип и отделять их реализацию в другие объекты. Композиция типов как правило считается более хорошим способом для реализации множественного наследования.

Что еще более важно, данный аспект может быть использован для автоматической реализации интерфейсов, чей код в целом повторяющийся и ничем не примечателен. Например как для интерфейса `INotifyPropertyChanged`.

Более подробно об этом аспекте пойдет речь в других статьях, тут много чего можно порассказать.

Пример использования

Допустим у вас есть большой и долгоиграющий проект с использованием многопоточности. Вдруг стала возникать трудноуловимая ошибка, которая не воспроизводится на вашей машине с дебаггером и вы принимаете решение залогировать все обращения к пространству имен `System.Threading`.

Для того, чтобы начать использовать `PostSharp` в вашем проекте, нужно добавить в список ссылок проекта две библиотеки `PostSharp.Public.dll` и `PostSharp.Laos.dll`.

Т.к. у нас будут логироваться методы не из наших сборок, то будем применять аспект `OnMethodInvocationAspect`.

```
using System;
using PostSharp.Laos;

namespace PostSharp.Laos.Test
{
    [Serializable]
```

```
public class MyOnMethodInvocationAspect : OnMethodInvocationAspect
{
    public override void OnInvocation(MethodInvocationEventArgs context)
    {
        Console.WriteLine("Calling {0}", context.Delegate.Method);
        context.Proceed();
    }
}
}
```

Аспекты **всегда** должны быть помечены как сериализуемые, это нужно в процессе выполнения кода и вставки во все нужные места программы, к тем методам и классам к которым вы применили аспект. Далее мы перегружаем метод **OnInvocation** в котором и делаем вывод имени метода на консоль.

Применить аспект можно следующим образом:

```
[assembly: MyOnMethodInvocationAspect( AttributeTargetAssemblies = "mscorlib",
AttributeTargetTypes = "System.Threading.*")]
```

Эта строчка пишется в файле `AssemblyInfo.cs` для интересующей вас сборки. С помощью такого задания вам не придется приписывать аспект в миллионе мест вызова функций связанных с многопоточностью, аргумент **AttributeTargetTypes** позволяет указать, к чему применять аспект с помощью регулярных выражений.

В результате PostSharp найдет все вызовы из пространства имен `System.Threading` и заменит их на вызов метода `MyOnMethodInvocationAspect.Invoke`. Естественно это будет на этапе компиляции приложения.

Конец первой части

В этой статье я рассказал о том, какие аспекты бывают и как они работают в целом. В следующих статьях я планирую более подробно остановиться на каждом из них. Рассказать, как аспекты могут взаимодействовать друг с другом, как ими можно управлять через атрибуты, как их применение можно валидировать во время компиляции. Какие новые штуки появились во второй версии этого фреймворка.

Hard'n'heavy!