Introduktion til AOP i C#

Denne artikel giver en introduktion AOP (Aspect Oriented Programming) i C#.

Den forudsætter et pænt kendskab til programmering, OOP og C#.

Skrevet den **03. Feb 2009** af **arne_v** I kategorien **Programmering / C#** | ★★★☆☆

Historie:
V1.0 - 03/08/2008 - original

Introduktion

Denne artikel vil give en kort beskrivelse af AOP (Aspect Oriented Programming), hvor nogle af nøgle begreberne beskrives, og komme med nogle simple eksempler.

AOP er et svært emne. Jeg vil anlægge en praktisk orienteret synsvinkel fordi:
- ellers bliver det for tungt
- jeg er ikke selv nogen ørn til teorien

Hvad er AOP

Som så mange andre koncepter er AOP opfunder på Xerox PARC.

AOP erstatter ikke OOP men supplerer OOP. Man kan også sige at AOP fylder nogle huller som OOP har efterladt.

For 20 år siden var det almindeligt at lave OOP i C++ med massiv brug af arv - man lod ofte klasser arve løs fra alle mulige klasser.

Det gør man ikke længere idag. Java og C# tillader kun implementation arv fra en enkelt klasse. Og ofte anbefaler man composition over extension (arv).

Kort sagt skal en klasse kun arve fra en anden hvis der er en ægte "is a" sammenhæng i den virkelighed man modellerer. Det er noget skidt at arve fra en klasse bare for at genbruge noget kode fra den.

Det er overordnet set en sund ændring.

Men erfaringen viser også at der er en noget funktionalitet af mere teknisk art som man gerne vil have ud i alle sine klasser.

Man kan naturligvis putte koden i alle klasserne, men

det ville jo være smart hvis man kunne nøjes med at have
koden engang.

AOP løser denne opgave !

Nøgle begreberne i AOP er:

concern = funktionalitet

cross-cutting concern = funktionalitet på tværs af klasser

point-cut = sted i koden hvor man gerne vil have udført noget kode (advice)

advice = den kode som man gerne vil have udført forskellige steder (point-cuts)

aspect = point-cut + advice

AOP drejer sig altså om at identificere noget funktionalitet som skal være
i mange forskellige klasser (cross-cutting concerns) og definere nogle steder
i koden (point-cuts) hvor man gerne vil have udført noget kode (advices) og
AOP kode består af aspects i stedet for classes.

Ved at flytte de cross-cutting concerns ud af alle classes og over i specielle
aspects får man en pænere og mere vel strukturet kode.

Traditionelle cross-cutting concerns er:
* logging
* exception handling
* transactions
* security
* caching

AOP i .NET

Der er flere forskellige AOP frameworks til .NET.

Men intet fra Microsoft (jeg betragter ikke ContextBoundObject som
værende en brugbar løsning) og ingen som er specielt meget udbredte.

AOP har aldrig rigtigt slået an i .NET verdenen og AOP er noget mere
udbredt i Java verdenen.

Det er der ikke nogen teknisk begrundelse for.

Jeg vil vise AOP med 2 forskellige AOP frameworks for .NET:
- AspectDNG
- PostSharp

Jeg kan personligt godt lide AspectDNG fordi det minder meget om
AspectJ som jeg kender fra Java.

Men der er andre som foretrækker PostSharp.

AspectDNG kan hentes herfra:
http://aspectdng.tigris.org/

PostSharp kan hentes herfra:
http://www.postsharp.org/

Eksempel

Jeg vil bruge følgende meget trivielle kode til at illustrere AOP.

IntMath.cs:

```
public class IntMath
{
    public static int Add(int a, int b)
    {
        return (a + b);
    }
    public static int Subtract(int a, int b)
    {
        return (a - b);
    }
    public static int Multiply(int a, int b)
    {
        return (a * b);
    }
    public static int Divide(int a, int b)
    {
        return (a / b);
    }
}
```

DoubleMath.cs:

```
public class DoubleMath
{
    public static double Add(double a, double b)
    {
        return (a + b);
    }
    public static double Subtract(double a, double b)
    {
        return (a - b);
    }
    public static double Multiply(double a, double b)
    {
        return (a * b);
    }
    public static double Divide(double a, double b)
    {
        return (a / b);
    }
}
```

Test.cs:

```
using System;

public class Test
{
    public static void Main(string[] args)
    {
        Console.WriteLine("2+3=" + IntMath.Add(2, 3));
        Console.WriteLine("2.0*3.0=" + DoubleMath.Multiply(2.0, 3.0));
        Console.WriteLine("10/0=" + IntMath.Divide(10, 0));
    }
}
```

Og kørsel uden brug af AOP ser ud som:

```
C:\e\aop_cs\aspectdng>csc Test.cs IntMath.cs DoubleMath.cs
Microsoft (R) Visual C# 2005 Compiler version 8.00.50727.1433
for Microsoft (R) Windows (R) 2005 Framework version 2.0.50727
Copyright (C) Microsoft Corporation 2001-2005. All rights reserved.

C:\e\aop_cs\aspectdng>Test
2+3=5
2.0*3.0=6

Unhandled Exception: System.DivideByZeroException: Attempted to divide by
zero.
   at Test.Main(String[] args)
```

Ikke noget overraskende i dette.

Jeg starter med AspectDNG.

Tracing

Nu vil vi prøve at trace kaldene i koden.

Trace.cs:

```
using System;

using DotNetGuru.AspectDNG.Joinpoints;

public class Trace
{
    [AroundCall("* IntMath::*(*)")]
    [AroundCall("* DoubleMath::*(*)")]
    public static object MathTrace(MethodJoinPoint mjp)
    {
```

```
        Console.WriteLine("Before " + mjp);
        object result = mjp.Proceed();
        Console.WriteLine("After " + mjp);
        return result;
    }
}
```

Kørsel:

```
C:\e\aop_cs\aspectdng>csc /r:aspectdng.exe /out:Test.exe Test.cs IntMath.cs
DoubleMath.cs Trace.cs
Microsoft (R) Visual C# 2005 Compiler version 8.00.50727.1433
for Microsoft (R) Windows (R) 2005 Framework version 2.0.50727
Copyright (C) Microsoft Corporation 2001-2005. All rights reserved.

C:\e\aop_cs\aspectdng>aspectdng.exe Test.exe
C:\e\aop_cs\aspectdng>Test
Before ?static method IntMath::Add(2,3)
After ?static method IntMath::Add(2,3)
2+3=?5
Before ?static method DoubleMath::Multiply(2,3)
After ?static method DoubleMath::Multiply(2,3)
2.0*3.0=?6
Before ?static method IntMath::Divide(10,0)

Unhandled Exception: System.DivideByZeroException: Attempted to divide by
zero.
   at DotNetGuru.AspectDNG.Joinpoints.JoinPoint.Proceed()
   at Test.Divide_AspectDNG_Call_2(Int32 , Int32 )
   at Test.Main(String[] args)
```

Lad os hoppe tilbage til AspectDNG syntaxen og forklare hvad der sker:

AroundCall = poincut around method call

* IntMath::*(*) =
  any return type
  class IntMath
  any method
  any number of parameters

* DoubleMath::*(*) =
  any return type
  class DoubleMath
  any method
  any number of parameters

Exception handling

Nu vil vi prøve at fange exceptions.

Error.cs:

```
using System;

using DotNetGuru.AspectDNG.Joinpoints;

public class Error
{
    [AroundCall("* IntMath::*(*)")]
    [AroundCall("* DoubleMath::*(*)")]
    public static object MathError(MethodJoinPoint mjp)
    {
        try
        {
            return mjp.Proceed();
        }
        catch(Exception e)
        {
            Console.WriteLine("Exception in math calculation ---- " +
e.Message);
            throw e;
        }
    }
}
```

Kørsel:

```
C:\e\aop_cs\aspectdng>csc /r:aspectdng.exe /out:Test.exe Test.cs IntMath.cs
DoubleMath.cs Error.cs
Microsoft (R) Visual C# 2005 Compiler version 8.00.50727.1433
for Microsoft (R) Windows (R) 2005 Framework version 2.0.50727
Copyright (C) Microsoft Corporation 2001-2005. All rights reserved.

C:\e\aop_cs\aspectdng>aspectdng.exe Test.exe
C:\e\aop_cs\aspectdng>Test
2+3=?5
2.0*3.0=?6
Exception in math calculation ---- ?Attempted to divide by zero.

Unhandled Exception: System.DivideByZeroException: Attempted to divide by
zero.
   at DotNetGuru.AspectDNG.Joinpoints.JoinPoint.Proceed()
   at Test.Divide_AspectDNG_Call_2(Int32 , Int32 )
   at Test.Main(String[] args)
```

Special checking

Nu vil vi prøve at se den kommende exception inden den sker. Det kræver at vi checker argumenter i kaldene.

Check.cs:

```
using System;

using DotNetGuru.AspectDNG.Joinpoints;

public class Check
{
    [AroundCall("* *::Divide(*)")]
    public static object MathCheck(MethodJoinPoint mjp)
    {
        object o = mjp[1];
        if(o is int)
        {
            if((int)o == 0)
            {
                Console.WriteLine("About to divide by zero");
            }
        }
        if(o is double)
        {
            if((double)o == 0)
            {
                Console.WriteLine("About to divide by zero");
            }
        }
        return mjp.Proceed();
    }
}
```

Kørsel:

```
C:\e\aop_cs\aspectdng>csc /r:aspectdng.exe /out:Test.exe Test.cs IntMath.cs
DoubleMath.cs Check.cs
Microsoft (R) Visual C# 2005 Compiler version 8.00.50727.1433
for Microsoft (R) Windows (R) 2005 Framework version 2.0.50727
Copyright (C) Microsoft Corporation 2001-2005. All rights reserved.

C:\e\aop_cs\aspectdng>aspectdng.exe Test.exe
C:\e\aop_cs\aspectdng>Test
2+3=?5
2.0*3.0=?6
About to divide by zero?

Unhandled Exception: System.DivideByZeroException: Attempted to divide by
zero.
   at DotNetGuru.AspectDNG.Joinpoints.JoinPoint.Proceed()
   at Test.Divide_AspectDNG_Call_1(Int32 , Int32 )
   at Test.Main(String[] args)
```

Det samme med PostSharp

Test.cs:

```
using System;

public class Test
{
    public static void Main(string[] args)
    {
        Console.WriteLine("2+3=" + IntMath.Add(2, 3));
        Console.WriteLine("2.0*3.0=" + DoubleMath.Multiply(2.0, 3.0));
        Console.WriteLine("10/0=" + IntMath.Divide(10, 0));
    }
}
```

IntMath.cs:

```
[Trace]
[Error]
[Check]
public class IntMath
{
    public static int Add(int a, int b)
    {
        return (a + b);
    }
    public static int Subtract(int a, int b)
    {
        return (a - b);
    }
    public static int Multiply(int a, int b)
    {
        return (a * b);
    }
    public static int Divide(int a, int b)
    {
        return (a / b);
    }
}
```

DoubleMath.cs:

```
[Trace]
[Error]
[Check]
public class DoubleMath
{
    public static double Add(double a, double b)
```

```
    {
        return (a + b);
    }
    public static double Subtract(double a, double b)
    {
        return (a - b);
    }
    public static double Multiply(double a, double b)
    {
        return (a * b);
    }
    public static double Divide(double a, double b)
    {
        return (a / b);
    }
}
```

Trace.cs:

```
using System;

using PostSharp.Laos;

[Serializable]
public class Trace : OnMethodBoundaryAspect
{
    public override void OnEntry(MethodExecutionEventArgs ctx)
    {
        Console.WriteLine("Before " + ctx.Method);
    }
    public override void OnExit(MethodExecutionEventArgs ctx)
    {
        Console.WriteLine("After " + ctx.Method);
    }
}
```

Error.cs:

```
using System;

using PostSharp.Laos;

[Serializable]
public class Error : OnMethodBoundaryAspect
{
    public override void OnException(MethodExecutionEventArgs ctx)
    {
        Console.WriteLine("Exception in math calculation ---- " +
ctx.Exception);
    }
```

```
}
```

Check.cs:

```
using System;

using PostSharp.Laos;

[Serializable]
public class Check : OnMethodBoundaryAspect
{
    public override void OnEntry(MethodExecutionEventArgs ctx)
    {
        object o = ctx.GetReadOnlyArgumentArray()[1];
        if(o is int)
        {
            if((int)o == 0)
            {
                Console.WriteLine("About to divide by zero");
            }
        }
        if(o is double)
        {
            if((double)o == 0)
            {
                Console.WriteLine("About to divide by zero");
            }
        }
    }
}
```

Kørsel:

```
C:\e\aop_cs\postsharp>csc /r:PostSharp.Laos.dll /r:PostSharp.Public.dll
Test.cs
IntMath.cs DoubleMath.cs Trace.cs Error.cs Check.cs
Microsoft (R) Visual C# 2005 Compiler version 8.00.50727.1433
for Microsoft (R) Windows (R) 2005 Framework version 2.0.50727
Copyright (C) Microsoft Corporation 2001-2005. All rights reserved.


C:\e\aop_cs\postsharp>PostSharp.exe Default.psproj /p:SearchPath=.
/p:Output=TestPS.exe /p:IntermediateDirectory=. /p:CleanIntermediate=True
/p:SignAssembly=False /p:PrivateKeyLocation=. Test.exe

PostSharp 1.0 [1.0.9.373] - Copyright (c) Gael Fraiteur, 2005-2008.

info PS0035: c:\WINDOWS\Microsoft.NET\Framework\v2.0.50727\ilasm.exe
"C:\e\aop_cs\postsharp\Test.il" /QUIET /EXE
"/RESOURCE=C:\e\aop_cs\postsharp\Test.res"
```

```
"/OUTPUT=C:\e\aop_cs\postsharp\TestPS.exe" /SUBSYSTEM=3 /FLAGS=1 /BASE=4194304
/STACK=1048576 /ALIGNMENT=512 /MDV=v2.0.50727

C:\e\aop_cs\postsharp>TestPS
Before Int32 Add(Int32, Int32)
After Int32 Add(Int32, Int32)
2+3=5
Before Double Multiply(Double, Double)
After Double Multiply(Double, Double)
2.0*3.0=6
Before Int32 Divide(Int32, Int32)
About to divide by zero
Exception in math calculation ---- System.DivideByZeroException: Attempted to
divide by zero.
   at IntMath.Divide(Int32 a, Int32 b)

Unhandled Exception: System.DivideByZeroException: Attempted to divide by
zero.
   at IntMath.Divide(Int32 a, Int32 b)
   at Test.Main(String[] args)
After Int32 Divide(Int32, Int32)
```

Vi ser at PostSharp er lidt anderledes.

Advice fungerer stort set på samme måde som i AspectDNG.

Men point-cuts defineres ikke sammen med advice men derimod i koden som det gælder for.

I eksemplet har jeg brugt alle 3 aspect's samtidigt, da jeg ellers skulle vise 3 versioner af IntMath og DoubleMath.

Jeg foretrækker klart AspectDNG's måde over PostSharps måde fordi:
1) jeg foretrækker at have hele aspect'et samlet
2) jeg foretrækker at koden der skal modificeres ikke skal ændres (hvis aspect er permanente, så er er det lige meget, men hvis apsect kun skal bruges til debug eller performance måling, så betyder det noget)

Endvidere kunne jeg ikke få PostSharp OnMethodInvocationAspect til at virke som dokumenteret. Og jeg er ikke den eneste - se:

http://blogs.microsoft.co.il/blogs/dorony/archive/2007/02/02/Using-AOP-and-PostSharp-to-Enhance-Your-Code_3A00_-Part-B.aspx
lige før "Conclusion".

Jeg vil derfor klart anbefale AspectDNG over PostSharp.

Duer det ?

Efter min bedste overbevisning er AOP et yderst kraftfuldt værktøj til moderne software udvikling.

Min træerne vokser naturligvis ikke ind i himlen.

AOP er et meget vanskeligt koncept at mestre. Generelt vil jeg kun anbefale det til erfarne udviklere som kan gennemskue hvad de laver.

AOP giver også kun benefits ved store projekter.

Jeg konkluderer derfor at:
- frameworks kan og bør bruge AOP
- business applications bør i de fleste tilfælde undlade at bruge AOP og i stedet for bruge nogle frameworks som bruger AOP

Kommentar af thesurfer d. 08. Oct 2008 | 1

Enten har jeg fuldstændigt overset pointen med denne artikel, eller også er der bare ikke noget i den. Et lige så godt (og simpler) alternativ er Try/Catch.

Kommentar af krukken d. 10. Oct 2008 | 2

Rigtig god introduktion til Aspekter.

Kommentar af goblinhero (nedlagt brugerprofil) d. 12. Oct 2008 | 3

Ok intro-artikel - måske ville det være en god ide, at forklare formålet med AOP lidt mere i dybden. I det korte eksempel får man ikke rigtig set styrken ved AOP - og jeg er ikke enig i konklusionen - AOP giver mulighed for at 'slippe' for at have en masse boiler-plate kode i sine klassser, som minimerer muligheden for at lave slåfejl i logging/exception-kodeblokke, som kan være svære at spore. Og bare moderat store business applications har man masser af cross-cutting concerns i form af security, errorhandling og logging, hvor AOP giver rigtig meget mening.

Kommentar af frosty-dk d. 27. Aug 2008 | 4

Dog er jeg ikke særlig erfaren koder men har stiftet bekendskab til OOP, selvom jeg nok ikke har gennemskuet hele konceptet synes jeg det var en rigtig god forklaret artikel, og jeg har da fået noget ud af den. Så er man interesseret i OOP ville jeg mene den er værd at læse!