JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2007**

Vol. 6, No. 3, March - April 2007

ASPECT ORIENTED PROGRAMMING IN .NET. BASED ON ATTRIBUTES

Miguel Katrib Mora and **Yamil Hernández Saá**
Computer Science Department.University of Havana

Abstract

Aspect Oriented Programming proposes an approach to increase code functionality with aspects that are not part of the main code functionality. The current work shows how can be done AOP in .NET thanks to one of the most interesting .NET innovations: its capacity to put custom metadata inside a software component by means of .NET attributes. This paper defines different kinds of such attributes to represent aspects and shows how the functionality embedded in the attributes can be integrated with the functionality of the code decorated by them.

Such aspect-attributes are inserted into an existing .NET component without forcing reprogramming the client code of the component. Then the code of the aspect functionality is woven into the code of the component.

1 INTRODUCTION

The dream of software developers is a world in which components could be easy assembled using high level languages and tools. But this would be a very simplistic point of view to assume a world in which each problem domain could be factored into discrete components only interacting by method invocation. Such a premise ignores the fact that some aspects of an application program tend to permeate all its parts.

The application code tends to become contaminated with code snippets trying to deal with those "aspects" that are not the central part of the problem domain. A classical example of such aspect is security (most applications are worried about security despite security is not the central bussinees of them). These aspects usually obstruct the original code crossing the problem domain and affecting the goal to obtain reusable solutions. To provide reuse mechanisms for these problems is the focus of the so called Aspect Oriented Programming (AOP) [1].

Based on the separation of concerns principle, AOP tries to provide mechanisms for factoring out the parts of an application that are not pertinent to the central problem domain. Then, the AOP approach could offer two main benefits: Application code will be

Cite this column as follows: Miguel Katrib Mora and Yamil Hernández Saá: “Aspect Oriented Programming in .NET. Based on Attributes”, in *Journal of Object Technology*, vol. 6, no. 3, March-April 2007, pp. 53-70 http://www.jot.fm/issues/issue_2007_03/article1

no longer tangled with code that is no related to the problem at hand and factoring such aspects beyond the problem domain helps to reuse them in other applications.

.NET has two great features that supports the AOP basis. One is the reflection and code emission mechanism. Reflection allows dynamically introspect a .NET assembly, code emission allows generate code at runtime. The other feature is a novelty, the capacity to embed meta-information into an assembly through attributes (see [2]). However, .NET lacks a direct and integrated mechanism to relate the semantics that could be expressed by such attributes with the main functionality of the application code "decorated" with them.

Covering reflection and code emission goes beyond the scope of this paper, thus we will mention them only when it will be needed for implementation details (for more about reflection refer to MSDN and see [3] and [4]). The following section 2 proposes how to use the attributes to express different types of aspects. Section 3 explains an implementation pattern for "weaving" the aspects functionality with the code functionality.

2 .NET ATTRIBUTES AND ASPECTS

An attribute is meta-information that can be attached to different parts of a .NET code. Attributes do not apply any functionality by themselves, so there must be other code, or the same code in which an attribute is embedded, the code that must interpret the attribute to offer some behaviour.

Different .NET source languages will express attributes with their own syntax. After compiling a source code with attributes, each attribute will remain embedded in the .NET assembly as a special serialization of an "object instantiation" of a class derived of the .NET type `System.Attribute`. So, the real instantiation must be done by a tool or application using reflection.

The C# code snippet in Listing 1 shows a custom `Currency` attribute definition and also shows how it is used to indicate the currency type of an employee salary.

Listing 1

```
enum CurrencyType = {USD, Yen, Euro};

[AttributeUsage(AttributeTarget.Property | AttributeTarget.Field,
AllowMultiple = false)]
class Currency: System.Attribute
{
 public Currency(CurrencyType currency){
  this.currency = currency;
}
 public readonly currency;
  ...
}
...
class Employee
{
```

```
        [Currency(CurrencyTypes.Euro)]
        public float Salary
        { ...
        }
        ...
      }
```

For information about the predefined attributes included in the .NET library see [2]. Among these attributes is interesting to remark the "meta" attribute `AttributeUsage`, an attribute to attach to an attribute definition. For example, the statement

```
[AttributeUsage(AttributeTarget.Property | AttributeTarget.Field,
      AllowMultiple = false)]
```

in Listing 1 indicates that the attribute `Currency` can only be attached to properties and fields. The parameter `AllowMultiple = false` indicates only one attribute can be attached.

Using attributes to define aspects

Using attributes to represent an aspect should be associated to some semantics about the aspect, i.e. it is necessary to define some aspect's functionality and to specify "how" and "when" those functionality is integrated to the essential functionality of the application (this is known as *weaving* in the AOP terminology).

Normally .NET attributes are written in source code (for example C#) mixed with the source code they decorate. In parallel with this work we are developing a tool to put attributes into an existing assembly (independently of the source code producing the assembly).This capability to put aspects without mixing then with the main source code will favours maintainability and modularity.

The type to define *aspect-attributes* will be the base class `AspectAttribute` (Listing 2). `AttributeUsage` indicates aspect-attributes may be attached to a class, interface, method or property. An aspect-attribute decorating a class (interface) will be interpreted as if it would be associated to all the methods and properties of the class (interface) and their derived types.

Listing 2

```
public abstract class AspectAttribute: Attribute
{
  public abstract void Advice(object target,
                              object result,
                              MethodBase method,
                              object[] parameters
                              );
}

[AttributeUsage(AttributeTargets.Method |
                AttributeTargets.Property |
```

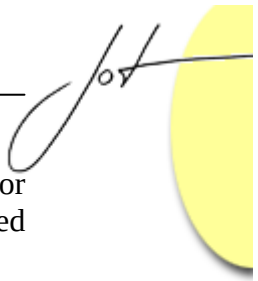
```
                AttributeTargets.Class |
                AttributeTargets.Interface,
                Inherited = true)

public abstract class BeforeAttribute: AspectAttribute
{
}

[AttributeUsage(AttributeTargets.Method |
                AttributeTargets.Property,
                Inherited = true,
                AllowMultiple = false)

public abstract class InsteadAttribute: AspectAttribute
{
}


[AttributeUsage(AttributeTargets.Method |
                AttributeTargets.Property |
                AttributeTargets.Class |
                AttributeTargets.Interface,
                Inherited = true)

public abstract class AfterAttribute: AspectAttribute
{
      }
```

The abstract method `Advice` represents the functionality of the aspect that should be defined by each actual subclass of the aspect-attribute class. Such `Advice` will be applied related to the execution of the method to which the aspect is attached (to all methods of a type if the attribute is attached to a class or interface). The code of the `Advice` will be interwoven with the code decorated with the aspect. According to the patterns applied by the interwoven mechanism there are different types of aspects attributes.

Semantics of the aspect-attribute

The execution of the `Advice` is related to the execution of the method decorated by the aspect-attribute when the method is called from a qualified call. In this context the `Advice` signature (Listing 2) means the following:

The `target` parameter receives the target object of the call. For example, if a method `F` has the aspect `A`, then when the `Advice` of `A` will be executed associated to a call `x.F(...)`, the value passed to the parameter `target` will be `x`.



The parameter, `result`, is used when an `AfterAttribute` is attached to a property or to a non void method. In such case the `result` parameter will receive the value returned by the method or property, so this value could be used by the `Advice` method.

The parameter `method` receives the `MethodBase` reflection object describing the called method. Example, for the call `x.F(...)` the value passed to this parameter is the `MethodBase` object corresponding to `F`.

The value passed to `parameters`, is an `object[]` array with the parameters used in the call to the method `x.F(...)`.If there are parameters of value type they will be received transparently as `object` type (without mediation of the client code) thanks to the boxing mechanism. A `null` value will be passed when the target method doesn't have parameters.

Aspects types according to the interwoven pattern

Different types of aspects attributes are defined. The interwoven mechanism of the `Advice` depends on the type of the aspect and the call to the method that has been decorated with this aspect. As shown in Listing 2 the following aspects inherit from `AspectAttribute`

`BeforeAspect`: The `Advice` is executed before executing the called method or property (get or set method).

`AfterAspect`: The `Advice` is executed after executing the called method or property (get or set method).

`InsteadAspect`: The `Advice` is executed instead of the called method or property (get or set method). Note the attribute usage of `Instead` has `AllowMultiple = false` because only one *instead* aspect has sense.

Example of BeforeAspect

Lets a class

```
class A{
  ...
  public void F1(){...}
  public void F2(){...}
}
```

It is possible to count the calls to the method `F1` decorating the method `F1` with the aspect `CountingCalls` (Listing 3)

```
class A{
  ...
  [CountingCalls]
  public void F1(){...}
  public void F2(){...}
}
```

To count the calls to all the methods of a type `B` we can decorate the class definition.

```
[CountingCalls]
class B{
```

```
  ...
  public void H1(){...}
  public void H2(){...}
}
```

To count the calls to different methods, it is necessary to have a call counter for each method (Listing 3). The proposed aspect mechanism has a global dictionary `AspectContext`. In this case the `Advice` method will use this dictionary to put an entry for each method having the `CountingCall` attribute.

Listing 3

```
class CountingCalls: BeforeAttribute
{
  public override void Advice(
                                object target,
                                object result
                                MethodBase method,
                               object[] parameters
                             )
  {
    string methodName = method.Name;
    if (!AspectAttribute.AspectContext.ContainsKey(methodName))
      AspectAttribute.AspectContext.Add(methodName, 1);
    else
      AspectAttribute.AspectContext[methodName]=
        ((int)AspectContext[methodName])+1;
  }
}
```

A `BeforeAspect` could be used too as a precondition to a method execution. The `Advice` of the aspect is executed as a precondition to execute the method called. So, if the `Advice` execution is not successful, then an exception will be thrown and the called method, decorated by the aspect, is not executed.

In the code excerpt below a method `Push` is decorated with the aspect `NotFull` (Listing 5) to guarantee that before making an insertion the stack must be not full.

```
class Stack
{
    ...
  [NotFull ]
  public void Push(T x){...}
  public void Pop(){...}

      public object Top {...}
      public bool Full {...}
      public bool Empty {...}
       ...

   }
```

Listing 5

```
class NotFull: BeforeAspect
{
  public override void Advice(
                                object target,
                                object result
                                MethodBase method,
                               object[] parameters  {
   Stack stack= (Stack) target;
   if (stack.Full)
      throw new Exception("Stack cannot be full.");
   }
}
```

The sentence `Stack stack = (Stack) target;` supposes the parameter `target` receives an object of `Stack` type. This is the case in the code excerpt above where the aspect `NotFull` decorates the method `Push`.

Example of InsteadAspect

This example defines an aspect-attribute to indicate a method is fault tolerant adding the necessary functionality to retry the method execution when an exception occurs.

Using this aspect-attribute the following code defines the method `F1` will be executed up to 3 times if the execution fails, and the method `F2` will try to execute up to 5 times

```
class A{
  ...
  [FaultTolerant(3)]
  public void F1(){...}
  [FaultTolerant(5)]
```

```
  public void F2(){...}
}
```

`FaulTolerant` is a subclass of the `InsteadAspect` type (Listing 4). For an `InsteadAspect` the weaver mechanism will guarantee that when a call `a.F1(...)` is done, the method `F1` will not be executed but rather will be executed the `Advice` method of the `FaultTolerant` aspect. Note how the `Advice` code wraps the call to `F1` in a `try catch` loop. The call to `F1` is then done by reflection through the sentence `method.Invoke(target, parameters)`.

Isolating this fault tolerant pattern inside the aspect avoids the replication of a similar code for each method you want will be fault tolerant.

Listing 4

```
class FaultTolerant: InsteadAspect
{
  int times;
  public FaulTolerant(int times){this.times=times;}
  public override void Advice(
                                object target,
                                object result
                                MethodBase method,
                               object[] parameters)
  {
   while(true)
    try{
      return method.Invoke(target,parameters);
    }
    catch (Exception e)
    {
     if (times == 0) throw e.InnerException;
     else times --;
    }
  }
}
```

Example of AfterAspect

An `AfterAspect` applies when a method finishes and just before returning to the caller. So an `AfterAspect` could be used for example to transform the value returned by a method (or property). The `ConvertUSDToEuros` aspect of Listing 6 changes US dollars to euros. The following code excerpt applies this aspect to the methods of the `Account` class.

```
class Account
{
  ...
  [ConvertUSDToEuros]
  public int Balance{...}
  public void Add(int amount) {...}
  public void Withdraw(int amount) {...}
  ...
}
```

To convert to different currency type it is necessary to have the current conversion value.

Listing 5

```
class ConvertUSDToEuros: AfterAspect
{
  public override void Advice(
                                object target,
                                object result
                                MethodBase method,
                                object[] parameters  {
    //modifying the value returned from the decorated method
   result = result * tax_exchange
   }
}
```

An `AfterAspect` could be used as a post condition, i.e an assertion to be fullfilled after a method execution. So, if the `Advice` execution it is not successful, then an exception will be thrown.

In the code excerpt below the method `Push` of the class `Stack` is decorated with the aspect `LIFO` to guarantee that after the `Push` method execution the parameter `x` is on the stack top.

```
    class Stack
{
  ...
  [LIFO]
  [NotFull ]
  public void Push(T x){...}
  public void Pop(){...}

      public object Top {...}
      public bool Full {...}
      public bool Empty {...}
       ...

}
```

Listing 6

```
class LIFO : AfterAspect
{
  public override void Advice(
                                object target,
                                object result
                                MethodBase method,
                               object[] parameters  {
   Stack stack = (Stack) target;
   object pushed = parameters[0];
   if (stack.Top != pushed)
      throw new Exception("The Stack must apply a LIFO policy.");
   }
}
```

As in the precondition sample, the sentence `Stack stack = (Stack) target;` supposes the parameter `target` receives an object of `Stack` type. This is the case for the aspect `LIFO` decorating the `Push` method. The sentence `object pushed = parameters[0];` supposes the array `parameters` has the parameters passed to the `method Push` (i.e. the object to be pushed. Unfortunately, we have no way to guarantee this `LIFO` attribute would not be attached to a method different from `Push.`

Reader could note this use of before and after attributes resambles the Design by Contract Metaphor. A more declarative proposal will be referred in the section 4.

3 IMPLEMENTING THE ASPECT'S FUNCTIONALITY INTERWOVEN

The aspects functionality must be interwoven with the code they decorate. The aspects proposed in this paper decorate methods and properties. So, the method (property) execution will be the interwoven point (the join point using the AOP terminology).

Aspect attributes either must be placed by the programmer in the original source code or inserted directly in the assembly using some tool[1]. Finally, considering an assembly with aspects attributes embedded on it, the task of an interwoven mechanism to apply aspect functionality could be implemented under two approaches.

One approach is to generate a proxy who intercepts method calls and applies the functionality of the aspect. This approach requires the original assembly plus an extra assembly that includes code for the proxy and aspects.

[1] .NET security rules prevent to modify an assembly. Our ILLego tool produces the effect to include attributes in an existing assembly generating a new equivalent assembly but with the attributes embedded.

The second approach is to produce a target assembly with the same functionality as the original plus the aspect functionality included but without dependencies to the original.

To implement the later approach it is necessary to extract code from the original assembly to replicate it in the target. But such "copy and paste" capabilities are not offered by the .NET namespace `System.Reflection`. To solve it we used the library `Reflection.Editor` [5, 6]. This library emulates the capabilities of the .NET `System.Reflection` but also supports IL code extraction and manipulation.

The first step to generate the final assembly using `Reflection.Editor` is to create an "edition assembly". This edition assembly clones the original, ensuring the same functionality, but allows its modification.

```
Assembly sourceAssembly = Assembly.LoadFrom("...originalAssembly.dll");
EditionAssembly targetAssembly = new EditionAssembly(sourceAssembly);
```

The next step is to change the name of each method `M` decorated with aspects by the name `Hidden_M`. This is done by the following functionality of the `Reflection.Editor`.

```
OriginalMethod.SetName("Hidden_" + OriginalMethod.Name);
```

Then a new public method, with the same signature as the original, is generated.

```
EditionMethod newMethod = eType.DefineMethod(originalMethod.Name,
          originalMethod.Attributes, originalMethod.ReturnType, paramTypes);
```

This new method includes the code to manage the aspects and to call the renamed method (the original functionality) according to the aspect pattern (before, after, etc). The process is illustrated in Figure 1.

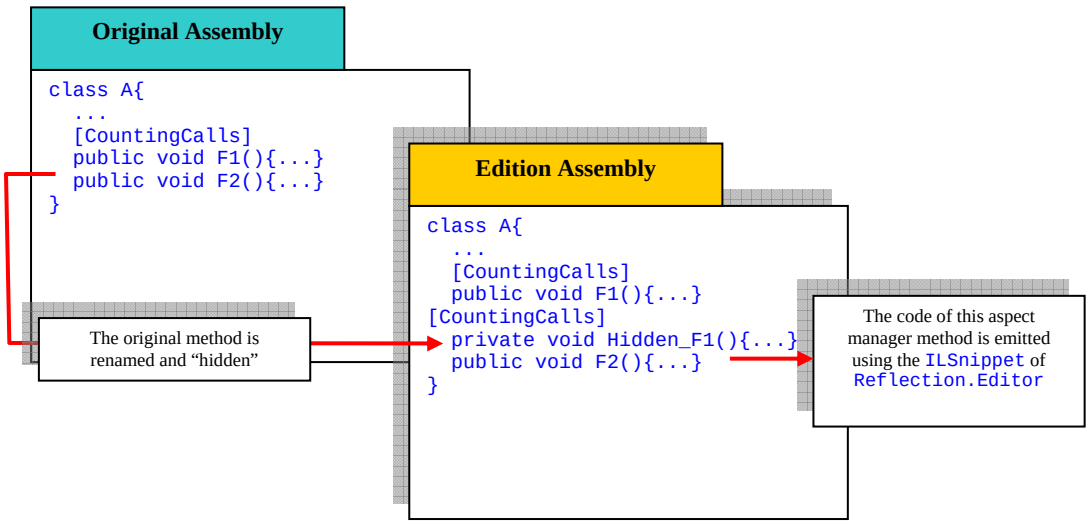


Figure 1

To produce the code of this method is used a feature of the `Reflection.Editor` named `ILSnippet`. An `ILSnippet` supports IL code emission and IL code "edition".

In short an `ILSnippet` is a "sequence" of IL instructions, plus local variable definitions and exception handlers. `Reflection.Editor` allows to extract the body of a method as an `ILSnippet`, Then by copying, inserting and changing this snippet it is possible to obtain the weaving of the main code with the aspect code.

IL code can be emitted adding IL instructions to an `ILSnippet` using the `ILSnippet` method `AppendInstruction(OpCode opCode , object operand).`

An IL instruction can be accessed by its index. The `myILSnippet[k]` returns the IL instruction at position *k* in the `ILSnippet`

Resume the example of the method `F1` decorated with the `CountingCalls` aspect

```
class A{
  ...
  [CountingCalls]
  public void F1(){...}
  public void F2(){...}
}
```

The generated code for the aspect manager will look like the code excerpt of Listing 7.

Listing 7

```
private AspectAttribute[][] typeAspects;
private AspectAttribute[][] Aspect_Of_Method_F1();
public class A
{
  public A();
  public void F1(){...};
  private void Hidden_F1(){...};
}
```

Here `typeAspects[0]` is an array of `BeforeAttribute` attributes and `typeAspects [1]` is an array of `AfterAttribute` attributes attached to a type definition and then applied to all methods of the type

`Aspect_Of_Method_F1` is an array containing three arrays as shown in Listing 8

Listing 8

```
this.Aspect_Of_Method_F1=new AspectAttribute[][3];


this.Aspect_Of_Method_F1[0]=((BeforeAttribute[])MethodBase.
        GetCurrentMethod().GetCustomAttributes(typeof(BeforeAttribute),true));

this.Aspect_Of_Method F1[1]=((InsteadAttribute[])MethodBase.
       GetCurrentMethod().GetCustomAttributes(typeof(InsteadAttribute),true));

this.Aspect_Of_Method_F1[2]=((AfterAttribute[])MethodBase.
         GetCurrentMethod().GetCustomAttributes(typeof(AfterAttribute),true));
```

`this.Aspect_Of_Method_F1[0]` contains the `BeforeAttribute` that decorates method `F1`

`this.Aspect_Of_Method_F1[1]` contains the `InsteadAttribute` that decorates method `F1`

`this.Aspect_Of_Method_F1[2]` contains the `AfterAttribute` that decorates method `F1`

Then the method `F1` will manage its aspects and will call the original `F1` that was renamed to `Hidden_F1.`

The code excerpt of Listing 9 shows the `F1` emitted code to retrieve the `BeforeAspect` attributes.

Listing 9

```
...

myILS.AppendInstruction(OpCodes.Ldarg_0);

myILS.AppendInstruction(OpCodes.Ldfld, fbMethodAspects);

myILS.AppendInstruction(OpCodes.Ldc_I4_0);

myILS.AppendInstruction(OpCodes.Call,
                typeof(MethodBase).GetMethod("GetCurrentMethod"));

myILS.AppendInstruction(OpCodes.Ldtoken, typeof(BeforeAttribute));

myILS.AppendInstruction(OpCodes.Call,
                        typeof(Type).GetMethod("GetTypeFromHandle",
                        new Type[] { typeof(RuntimeTypeHandle)} ));

myILS.AppendInstruction(OpCodes.Ldc_I4_1);

myILS.AppendInstruction(OpCodes.Callvirt,
                        typeof(MemberInfo).GetMethod("GetCustomAttributes",
                        new Type[] { typeof(Type), typeof(bool) }));

myILS.AppendInstruction(OpCodes.Castclass, typeof(BeforeAttribute[]));

//The BeforeAspects are stored in fbMethodAspects[0]
myILS.AppendInstruction(OpCodes.Stelem_Ref);

...
```

Listing 10 generates the code to invoke the `Advice` method of each `Before` aspect

Listing 10

```
...

myILS.AppendInstruction(OpCodes.Ldarg_0);
myILS.AppendInstruction(OpCodes.Ldfld, fbMethodAspects);

//Load the BeforeAspects stored in fbMethodAspects[0]
myILS.AppendInstruction(OpCodes.Ldc_I4_0);
myILS.AppendInstruction(OpCodes.Ldelem_Ref);

//Load the BeforeAspect stored in fbMethodAspects[0][count]
myILS.AppendInstruction(OpCodes.Ldloc_S, count);
```

```
myILS.AppendInstruction(OpCodes.Ldelem_Ref);

//Load the Advice args (target,result,method,parameters)
myILS.AppendInstruction(OpCodes.Ldarg_0);
myILS.AppendInstruction(OpCodes.Call,
                        typeof(MethodBase).GetMethod("GetCurrentMethod"));
myILS.AppendInstruction(OpCodes.Ldloca_S, returnValue);

myILS.AppendInstruction(OpCodes.Ldc_I4_S, paramTypes.Length);
myILS.AppendInstruction(OpCodes.Newarr, typeof(Object));
myILS.AppendInstruction(OpCodes.Stloc_S, parameters);
myILS.AppendInstruction(OpCodes.Ldloc_S, parameters);

for (int i=0; i<paramTypes.Length; i++)
{
       myILS.AppendInstruction(OpCodes.Ldc_I4_S,i);
       myILS.AppendInstruction(OpCodes.Ldarg_S,i+1);
       myILS.AppendInstruction(OpCodes.Stelem_Ref);
       myILS.AppendInstruction(OpCodes.Ldloc_S, parameters);
}

//Invoke the Advice
myILS.AppendInstruction(OpCodes.Callvirt,
                        typeof(AspectAttribute).GetMethod("Advice") );

...
```

If there is an `Instead` attribute then Listing 11 generate the code to invoke its `Advice` method

Listing 11

```
...

myILS.AppendInstruction(OpCodes.Ldarg_0);
myILS.AppendInstruction(OpCodes.Ldfld, fbMethodAspects);

//Load the InsteadAspect stored in fbMethodAspects[1][0]
myILS.AppendInstruction(OpCodes.Ldc_I4_1);
myILS.AppendInstruction(OpCodes.Ldelem_Ref);
myILS.AppendInstruction(OpCodes.Ldc_I4_0);
myILS.AppendInstruction(OpCodes.Ldelem_Ref);

//Load the Advice args (target,result,method,parameters)
myILS.AppendInstruction(OpCodes.Ldarg_0);
myILS.AppendInstruction(OpCodes.Call,
                        typeof(MethodBase).GetMethod("GetCurrentMethod"));
myILS.AppendInstruction(OpCodes.Ldloca_S, returnValue);

myILS.AppendInstruction(OpCodes.Ldc_I4_S, paramTypes.Length);
myILS.AppendInstruction(OpCodes.Newarr, typeof(Object));
myILS.AppendInstruction(OpCodes.Stloc_S, parameters);
myILS.AppendInstruction(OpCodes.Ldloc_S, parameters);

for (int i=0; i<paramTypes.Length; i++)
{
       myILS.AppendInstruction(OpCodes.Ldc_I4_S,i);
```

```
myILS.AppendInstruction(OpCodes.Ldarg_S,i+1);
       myILS.AppendInstruction(OpCodes.Stelem_Ref);
       myILS.AppendInstruction(OpCodes.Ldloc_S, parameters);
}

//Invoke the Advice
myILS.AppendInstruction(OpCodes.Callvirt,
                        typeof(AspectAttribute).GetMethod("Advice"));

...
```

Otherwise, Listing 12 generates the code to invoke the original method (renamed as `Hidden_F1`). Note method `Hidden_F1` was generated using the `Reflection.Editor` copy, paste and rename capabilities.

Listing 12

```
...

myILS.AppendInstruction(OpCodes.Ldarg_0);

//Load the method args
for (int i=0; i<paramTypes.Length; i++)
       myILS.AppendInstruction(OpCodes.Ldarg_S, i+1);

// Invoke the original method renamed as "Hidden_ " + method name
myILS.AppendInstruction(OpCodes.Call, hidden_mb);

//Store the method return in returnValue
if (hidden_mb.ReturnType != typeof(void))
{
if (hidden_mb.ReturnType.IsValueType)
              myILS.AppendInstruction(OpCodes.Box, hidden_mb.ReturnType);

       myILS.AppendInstruction(OpCodes.Stloc_S, returnValue);

}

...
```

The code to invoke the `Advice` method of each `AfterAspect` is generated like those for each `BeforeAspect`.

Presented approach, using `Reflection.Editor` capabilities, generates a new assembly that keeps the original functionality (without reference dependencies to the original assembly) but including also the code executing the aspects. Therefore, this new assembly can substitute the original in any project that wants to apply aspects functionality.

Using `Reflection.Editor` library it is also possible to implement the reverse process, i.e., remove aspects evaluation from the code to obtain an assembly equivalent to the original.

4 RELATED WORKS

An alternative to incorporate aspects in .NET is based on the use of contexts and messages exchange [7]. Nevertheless, this approach forces the client code to handle the infrastructure of messages exchanges. Another drawback of this approach is that each attribute defined for aspect purposes must repeat the same pattern in its implementation, so lacking reusability. In [8] we proposed an approach of aspects "interwoven" based on intercepting the calls to methods decorated by the aspect. This is done using the .NET context infrastructure and the concepts of real and transparent proxies. But such infrastructure overloads performance. Furthermore, this solution forces to do modifications in the client code.

AOP offers an interesting alternative for specification of nonfunctional component properties (such as fault-tolerance properties or timing behavior), an aspect-specific tool that adds fault tolerance to .NET components using aspect oriented techniques is described in [9].

There are approaches based on the use of pre-processors [10], however, such approaches request the acceptation of new source languages or language modifications. Our current work has focused in the use of .NET attributes to express non-functional component properties without disturbing the existing language syntax and semantics.

There are a variety of language extensions with AspectJ [11] as most prominent example. The CAMEO project [12] extends the C# compiler to add AOP extensions similar to those in AspectJ. It uses XML aspect definition files and outputs standard C# code which is compiled by the standard compiler. CAMEO is a static weaver.

CLAW is a .NET dynamic weaver implemented in C++ using the Common Object Model (COM) to extend the CLR. by linking in to the profiling mechanism, the CLAW architecture is defined in [13]. With this mechanism, it is possible to add a new method and to inject IL code at runtime into an existing method body, relocate methods from one type to another, and recompile existing methods. But this is a low level and tightly coupled approach depending on the profiler feature.

CONCLUSIONS

To do AOP based on modifications of existing languages, changing compilers and building a lot of acompanning tools is not a pragmatic solution. The approach proposed in the current paper tries to follow the mainstream of .NET not requesting changes in the source languages, but offering libraries, a single tool and proposing an AOP programming methodology. As was explained, this can be achieved under .NET thanks the .NET support to attributes and the enhanced reflection capabilities. The .NET reflection constrains to manipulate the IL code of assembly and to weave it with code of other assembly was overcome with the new `Reflection.Editor` library. A detailed

description of this library goes beyond the scope of this paper and could be explained in a future paper.

Other patterns of aspects could be considered in further works:

- aspects to express functionality to execute when the called method throws an exception
- aspects to indicate a timeout for the execution of a method (useful in remote applications)
- aspects to express undo-redo capability of a method execution

REFERENCES

[1] Gregor Kiczales and others, Aspect Oriented Programming, Springer Verlag, Proceedings of ECOOP 1997

[2] Barnaby Tom, Bock Jason, Applied .NET Attributes, APress 2003

[3] Box Don, Sells Chris Essential .NET: The Common Language Runtime, Addison-Wesley 2003

[4] del Valle Mario, Katrib Miguel, El poder de la reflexión en .NET, dotNetManía No 3, Abril 2004

[5] Bacallao Erick, Katrib Miguel, Parodi Yoelvis, Reflection.Editor una biblioteca para programar la edición de ensamblados .NET DotNetmanía No 15, Mayo 2005

[6] Bacallao Erick, Katrib Miguel, Parodi Yoelvis, Entretejido de Código IL en Ensamblados .NET usando Reflection.Editor, DotNetmanía No 23, Febrero 2006

[7] Shukla Drama, Fell Simon, Sells Chriss Aspect-Oriented Programming Enables Better Code Encapsulation and Reuse¸MSDN Mag, March 2003.

[8] Hernández Yamil, Katrib Miguel, Aspectos e Intercepción de Métodos en .NET, dotNetManía No 10, Diciembre 2004

[9] Schult W., Polze A., Aspect-Oriented Programming with C# and .Net. In Proc. of International Symposium on Object-oriented Real-time distributed Computing (ISORC) 2002, pp. 241-248, Crystal City, VA, USA, Mayo 2000.

[10] Safonov Vladimir, Aspect.NET: Concepts and Architecture, .NET Developer's Journal, October 2004.

[11] Kiczales G, Hilsdale E, Huguin J, Kersten M, Palm J, Griswold V, An Overview of AspectJ, Springer-Verlag proceedings of the ECOOP 2001.

[12] M. Devi Prasad and B.D. Chaudhary. AOP support in C# , AOSD 2003

[13] Lam J., Cross Language Aspect Weaving. Demonstration, AOSD 2002, Enschede, 2002.

About the authors

Miguel Katrib (mkm@matcom.uh.cu) is a PhD professor at the Computer Science Department of the University of Havana where he leads the WEBOO group dedicated to web and object oriented programming. Miguel authored several papers in programming languages and object technologies. He is an enthusiast of .NET technology working as redactor for the spanish journal dotNetManía. He is also a .NET programming advisor of the software company CARE Technologies, Denia, Spain.

Yamil Hernández (yhsaa@matcom.uh.cu) is an instructor at the Computer Science Department of the University of Havana. He is developer of WEBOO group dedicated to web and object oriented programming. Yamil is an enthusiast of .NET technology. His main interested areas are Aspect and Object Oriented Programming, Reflection and Compiling Techniques.