Journal of Basic and Applied Engineering Research
Print ISSN: 2350-0077; Online ISSN: 2350-0255; Volume 1, Number 10; October, 2014 pp. 69-86
© Krishi Sanskriti Publications
http://www.krishisanskriti.org/jbaer.html

Language Independent Implementation of Aspects in Aspect-Oriented Programming

Anita Bhatia

Dronacharya Institute of Management and Technology, Kurukshetra

Abstract: **The term aspect-oriented programming (AOP) has come to describe the set of programming mechanisms developed specifically to express crosscutting concerns. Since crosscutting concerns cannot be properly modularized within object-oriented programming, they are expressed as aspects and are composed, or woven, with traditionally encapsulated functionality referred to as components. Many AOP models exist, but their implementations are typically coupled with a single language. To allow weaving of existing components with aspects written in the language of choice, AOP requires a language-independent tool. This paper presents Weave.NET, a load-time weaver that allows aspects and components to be written in a variety of languages and freely intermixed. Weave.NET relies on XML to specify aspect bindings and standardized Common Language Infrastructure to avoid coupling aspects or components with a particular language. By demonstrating language-independence, Weave.NET provides a migration path to the AOP paradigm by preserving existing developer knowledge, tools, and software components. The tool's capabilities are demonstrated with logging aspects written in and applied to Visual Basic components.**

Keywords: **Aspect-oriented programming, Weave.NET, Common Language Infrastructure, language-independence.**

1. INTRODUCTION

Crosscutting concerns are “properties or areas of interest” [1] that normally defy object-oriented (OO) modelling, because the deployment of functionality to support them does not align with the composition operations available in an object model [2]. Even conceptually simple crosscutting concerns, such as tracing while debugging and compiling, may lead to tangling, in which the set of code statements addressing the crosscutting concern become interlaced with those addressing other concerns within the application.

“To eliminate this problem, AOP offers aspects: mechanisms beyond subroutines and inheritance for localizing the expression of a crosscutting concern.” [1] An *aspect* [3] provides a unit of encapsulation that couples the behaviour of a crosscutting concern with a join point specification that details where in component code the behaviour is to be applied. In the context of AOP, *components* [3] correspond to units of well-encapsulated behaviour be it source code or binaries. The aspects and components of an application are composed, or *woven*, to produce a single program.

Unfortunately, none of these AOP technologies support *language independence*, in this way they do little to present the composition model as decoupled from source code, or demonstrate by their implementation strategies and the ability to intermix aspects and components written in a different languages. AspectJ [4] views aspect and their implementation as a Java coding exercise. As Aspects are only present in source code, and after compilation they are no longer discernable. By the researchers, extending this aspect model to other languages is left as an exercise outside the AspectJ team, and no alternate is made to allow reuse of aspects across different languages. Demeter's aspect model is based around object graph traversal, which exists in most, if not all, object models. Weave.NET exploits the multi-language support of Microsoft's Common Language Infrastructure (CLI) [5], developed for the .NET Framework, to provide a solution for these problems. Weave.NET is a language-independent aspect weaver that avoids coupling aspects or components with a particular language. Weave.NET performs binary-level composition according to an XML-based composition script, meaning that the composition specification is not written in terms of, or using extensions to, a particular programming language. The script is applied at load-time, well after component and aspect behaviour is compiled to binary form. As such, the weaver is oblivious as to the implementation language of these behaviours.

2. PROGRAMMING MODEL

The Weave.NET programming model addresses two issues: how to specify aspects, and what architecture is used to compose those aspects with components. We provide an introduction to both issues and then contrast the Weave.NET approach to aspect specification in AspectJ.

2.1 Specifying Aspects

AspectJ syntax allows aspects to contain the same members as Java classes in addition to a set of exclusively Aspect Oriented (AO) constructs, such as point-cuts and advice; however, Weave.NET keeps AO constructs separate. In Weave.NET the cross-cutting details of an aspect are written in an XML deployment script. Non-AO type members, and indeed the behaviour of aspect advice, are obtained from an existing type implementation.

Weave.NET allows aspect behaviour and components to be implemented in any language that targets the CLI. Weave.NET places the declarative elements of an aspect in an XML file separate from source code. The declarative elements reference binaries that implement aspect behaviour, while the target components are specified when the Weave.NET API is called. Thus, aspect behaviour, as well as that of components, is compiled separately from the weaving process. The aspect programmer can then choose a suitable implementation language for aspect behaviour without affecting the ability to apply that behaviour in a crosscutting manner.

2.2 Weaving Aspects

At the centre of the composition architecture is the Weave.NET tool as shown in Figure 1. The input to Weave.NET is an existing CLI binary component, packaged as a .NET *assembly*, and an XML file containing the crosscutting specifications of an aspect. The behaviour of an aspect is provided separately in another assembly. Weave.NET recreates the input assembly, but in this new version join points are bound to behaviour in the aspect assembly as per the advice statements in the XML. Unlike .NET approaches that bind components and aspects via proxies [6, 7], Weave.NET modifies the CIL of the components to access aspect behaviour via method calls. As a result, clients of components are unaffected by weaving and weaving on call join points is fully supported.

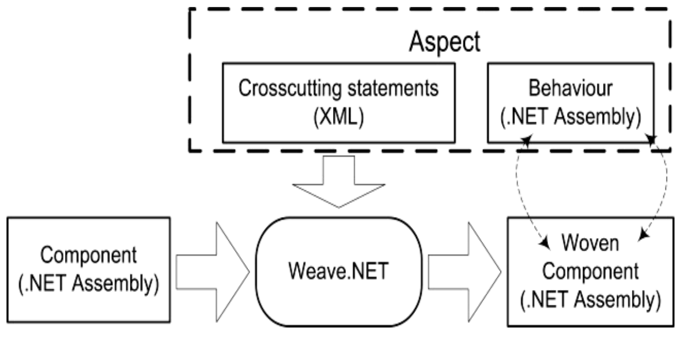
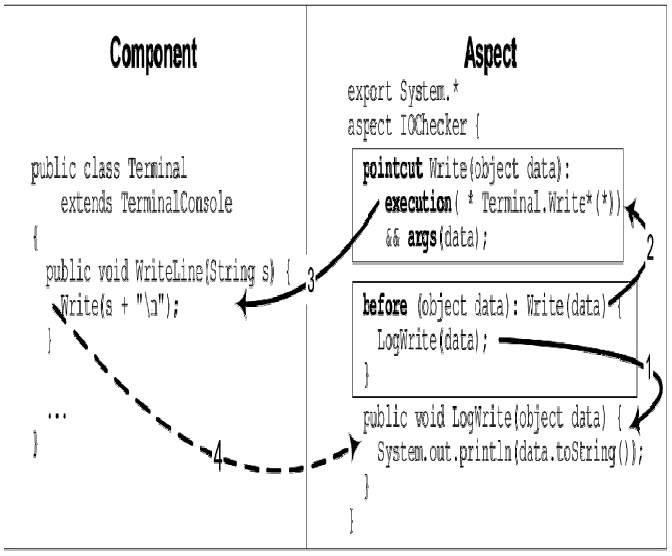

Aspect
Crosscutting statements (XML)
Behaviour (.NET Assembly)
Component (.NET Assembly)
Weave.NET
Woven Component (.NET Assembly)

Fig. 1. User-level view of weaving

2.3 Contrasting Weave.NET and AspectJ

Figures 2 and 3 contrast the approach to implementing a logging aspect in AspectJ and Weave.NET respectively. We start by explaining the aspect's function using the AspectJ example, and then review the Weave.NET implementation looking for contrasts with the AspectJ approach.

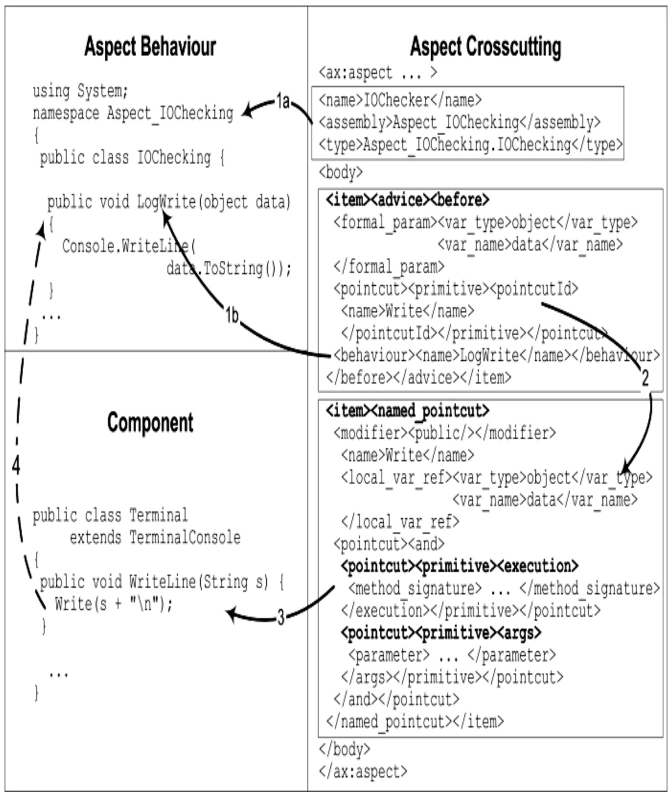


```
public class Terminal
   extends TerminalConsole
{
  public void WriteLine(String s) {
    Write(s + "\n");
  }

  ...
}
```

```
export System.*
aspect IOChecker {
  pointcut Write(object data):
    execution( * Terminal.Write*(*))
    && args(data);
  before (object data): Write(data) {
    LogWrite(data);
  }
  public void LogWrite(object data) {
    System.out.println(data.toString());
  }
}
```

Fig. 2. Interpretation of an AspectJ aspect.



```
using System;
namespace Aspect_IOChecking
{
 public class IOChecking {

  public void LogWrite(object data)
  {
    Console.WriteLine(
                data.ToString());
  }
  ...
}
```

```
<ax:aspect ... >
<name>IOChecker</name>
<assembly>Aspect_IOChecking</assembly>
<type>Aspect_IOChecking.IOChecking</type>
<body>
 <item><advice><before>
  <formal_param><var_type>object</var_type>
               <var_name>data</var_name>
  </formal_param>
  <pointcut><primitive><pointcutId>
   <name>Write</name>
   </pointcutId></primitive></pointcut>
  <behaviour><name>LogWrite</name></behaviour>
 </before></advice></item>
 <item><named_pointcut>
  <modifier><public/></modifier>
   <name>Write</name>
   <local_var_ref><var_type>object</var_type>
                 <var_name>data</var_name>
   </local_var_ref>
  <pointcut><and>
   <pointcut><primitive><execution>
    <method_signature> ... </method_signature>
   </execution></primitive></pointcut>
   <pointcut><primitive><args>
    <parameter> ... </parameter>
   </args></primitive></pointcut>
  </and></pointcut>
 </named_pointcut></item>
</body>
</ax:aspect>
```

```
public class Terminal
     extends TerminalConsole
{
 public void WriteLine(String s) {
  Write(s + "\n");
 }

  ...
}
```

Fig. 3. Weave.NET equivalent of Figure 2

Broadly speaking, the logging aspect is meant to report the data being written to I/O by a terminal emulator package called tcdIO. This I/O library was developed for introductory OO and VB instruction [8], and is referred to in examples throughout this paper. In the AspectJ implementation of Figure 2, the body of an advice statement implements the aspect's behaviour. Arrow 1 highlights how *before advice* references another member of the aspect type, LogWrite, to print data to the logging output. The before advice is applied to join points identified by the Write named pointcut, as indicated by arrow 2. Write specifies an intersection of execution join points specified with the execution and args primitive pointcut designators The execution designator identifies the output methods of a Terminal type, while the args designator selects from among these methods those that take a single argument. args also exposes this parameter for manipulation by aspect advice. Among the join points selected is the execution of the WriteLine method as indicated by arrow 3. At compile time, AspectJ composes the aspect with component behaviour such that the execution of WriteLine initially transfers control to the before advice, as visualised by arrow 4.

3. MAPPING THE ASPECT MODEL TO CIL

The aspect model in Weave.NET is derived from that of AspectJ. In this section we summarize this model's elements, and where possible, relate the elements to CLI architecture.

3.1 Join Point Model

The Weave.NET aspect model contains only dynamic join points. Dynamic join points are "well-defined points in the execution flow of the program" [9]. In contrast, static join points correspond to types to which new members can be added. The focus on dynamic join points stems from their identification as core to the AspectJ aspect model [9].

Dynamic join points are best understood by organising them into three categories: execution join points, call join points and field access join points as shown in Table 1. This organisation is show in AspectJ documentation [10] provides a better characterisation of specific join point types.

Table 1 Categorization of dynamic join points

Join point category	Join point types
Execution	Method execution Initializer execution Constructor execution Static initializer execution Handler execution Object initialization
Call	Method call Constructor call Object pre-initialization
Field access	Field reference Field assignment

Weave.NET execution join points correspond to blocks of CIL. In a .NET assembly, CIL code is located on a method by method basis. The assembly's metadata identifies which block of IL code corresponds to which method signature. This is true for constructors as well, since constructor bodies are modelled as methods with special names, such as .ctor in case of an instance constructor, and with certain metadata flags used to distinguish them from other methods.

Fine grained join points are resolved by closer inspection of the implementation of the method body. In the case of exception handlers, extra metadata tables associated with the method's code identify blocks of exception handling code. For execution join points related to object instantiation, it is necessary to examine the IL at the start of the constructor to distinguish constructor execution from object initialization. This is because data member initialization and flow of control between different constructors in a class' inheritance hierarchy is written explicitly into each constructor method.

```
public void WriteFloat(float value){
  WriteLine(value.ToString());
}
```

```
.method public hidebysig instance void
        WriteFloat(float32 A_1) cil managed
{
 // Code size       14 (0xe)
 // Execution start
 .maxstack  2
 IL_0000:  ldarg.0
 IL_0001:  ldarga.s   A_1
 IL_0003:  call instance string
         [mscorlib]System.Single::ToString()
 IL_0008:  call instance void
         [tcdIO]tcdIO.Terminal::WriteLine(string)
 // Execution end
 IL_000d:  ret
} // end of method Terminal::WriteFloat
```

Fig. 4 An execution join point

To clarify the concept of execution join points, the example in Figure 4 shows C# source code and corresponding IL of an execution join point in the tcdIO library. The start and end of the execution join point are identified relative to the CIL with embedded comments in bold font.

Call join points are present on the calling side of a method invocation or when the new operator is called for object construction. These points are observed as IL opcodes of type InlineMethod. These opcodes indicate the target method with a metadata token. Using this token, it is possible to lookup the signature of the method being called. The signature also indicates where on the stack the call context is located.

Constructors present a special case. They may be accessed as part of a call join point, for instance as part of a new operation, or they can be accessed as part of an execution join point, for instance via this() and super() calls in Java. Fortunately, these two cases are distinguished by the opcode used to access the constructor, which is NewObj in the case of a constructor call join point.

```
public void WriteFloat(float value){
  WriteLine(value.ToString());
}
```

```
.method public hidebysig instance void
        WriteFloat(float32 A_1) cil managed
{
 // Code size       14 (0xe)
 // Execution start
 .maxstack  2
 IL_0000:  ldarg.0
 IL_0001:  ldarga.s   A_1
 IL_0003:  call instance string
          [mscorlib]System.Single::ToString()
 IL_0008:  call instance void
          [tcdIO]tcdIO.Terminal::WriteLine(string)
 // Execution end
 IL_000d:  ret
} // end of method Terminal::WriteFloat
```

Fig. 5 A call join point

Revisiting the example in Figure 4, we can identify two call join points. In Figure 5, we highlight the call join point for the invocation of the WriteLine method in bold font.

The final category of join point is that of field access, which corresponds to a read or write access to a data member, or field in CLI terminology. These join points do not include final fields, i.e. constant fields emitted as literals in IL. These join points are observed as special IL opcodes used to access static and non-static fields. These opcodes are associated with a metadata token identifying the signature of the field being accessed.

3.2 Identifying Join Points

To a large extent, the point of our aspect model is to allow succinct identification of join points and expose portions of their execution context. To do so, we adopt AspectJ's pointcut mechanism and its join point selection operators, called *primitive pointcut designators*, used to specify pointcuts. A pointcut selects from among all the join points in a component those that are relevant to a particular crosscut. To do so it relies on primitive pointcut designators that select from certain join point types, as defined by that designator, those whose metadata description matches the designator's argument. Thus, this argument is usually a signature or type pattern, depending on the designator. Finally, several designators can be used together with logical operators that take the union or intersection of their join point sets.

Designators can be broken into three categories according to the argument that they take. Table 2 identifies designators that identify join points in control flow directly from signatures or type patterns associated with the source of these join points. Table 3 identifies designators that identify join points relative to those of another pointcut. Finally, Table 4 identifies designators that select join points according to objects and arguments used in the execution context of the join point. These designators can also be used to expose the join point's execution context to the aspect.

Table 2 Designators specified with a signature or type pattern.

Designator	Joint points selected
call(*Signature*)	Method and constructor calls.
execution(*Signature*)	Method and constructor execution.
initialization (*Signature*)	Object initializer execution.
get(*Signature*)	Field reference.
set(*Signature*)	Field assignment.
handler(*TypePattern*)	Exception handler execution.
staticinitialization (*TypePattern*)	Static initializer execution.
within(*TypePattern*)	All join points defined by the selected type.
withincode (*Signature*)	All join points defined within method or constructor matching declarations

Table 3 Designators specified with a pointcut.

Designator	Joint points selected
cflow(*pointcut*)	All join points encountered during the execution of join points identified by the pointcut.
cflowbelow(*pointcut*)	Identical to *cflow*, but does not include the join points identified by the pointcut argument.

Table 4 Designators that can expose execution context.

Designator	Joint points selected
this(*TypePattern or Id*)	Join points in which the object bound to this is an instance of a particular type.
target(*TypePattern or Id*)	Join points in which the object on which a call or field operation is applied to is an instance of a particular type.
args(*TypePattern or Id,* ...)	Join points where there are arguments whose types match those listed by the designator.

In the case of signatures and type patterns, Weave.NET supports both *name-based crosscutting* and *property-based crosscutting* [9]. Name-based crosscutting corresponds to the literal expression of signatures and type patterns. Thus, with name-based crosscutting the signatures and type patterns used in a pointcut must match those of the targeted join points exactly. The CLI provides the System.Reflection API to access this data. Property-based crosscutting exploits wildcards to partially specify designator arguments. In property-based crosscutting, the signatures and type patterns used in a pointcut correspond to regular expressions. Fortunately, the CLI supplies a library to support regular expression use that greatly simplifies resolving these wildcards.

Pointcuts imply a traversal of all join points in the targeted source code. The CLI provides limited tools for directly accessing metadata, but none for accessing IL directly. Fortunately, there is a performance-conscious library called CLIFile Reader [11] that allows direct access to IL streams.

4. WEAVER IMPLEMENTATION

Weave.NET is an aspect weaver implemented as a .NET component. Its weaving interface accepts as input a reference to a component assembly and to an XML document that contains the specification for an aspect. The result of calling this interface is a new version of the component assembly that is bound to aspect behaviour at the IL level.

The weaver implementation has two subsystems: code generation and aspect modelling. The aspect modelling system is responsible for interpreting the XML aspect specification, modelling aspects in terms of their pointcuts and advice, and detecting whether join points match any aspect advice. The code generation system is responsible for converting an existing assembly to a dynamic assembly and instantiating objects to represent join points. The bridge between these two systems is the JoinPoint class hierarchy.

4.1 Code Generation Architecture

The code generation system creates a dynamic assembly, i.e. a System.Reflection.Emit object hierarchy, corresponding to the assembly targeted for weaving. Were it not for the modifications specified by the aspect, this hierarchy would be emitted as a new, but functionally identical assembly. However, as per the aspect, there will be some differences. The principle classes used by the Emit library to model a dynamic assembly are shown in Figure 6. Here, a module corresponds to a physical file. Thus, an assembly can span files. Types and their constituent members are contained in one module or another.The System.Reflection API has been suggested as a tool for introspecting on existing assemblies [7], but, as noted previously, this API lacks the ability to directly access the IL stream. Without access to IL it is impossible to expose call join points, so the code generation system bypasses the convenience of the Reflection library and examines the assembly metadata directly with the CLIFile Reader API [11. The CLIFile Reader library provides abstractions to access intra-method details such as the IL stream and exception handling table. Directly accessing the file was considered, but CLIFile Reader provides decompression, metadata table modelling and greatly simplifies resolving cross-references within table entries.

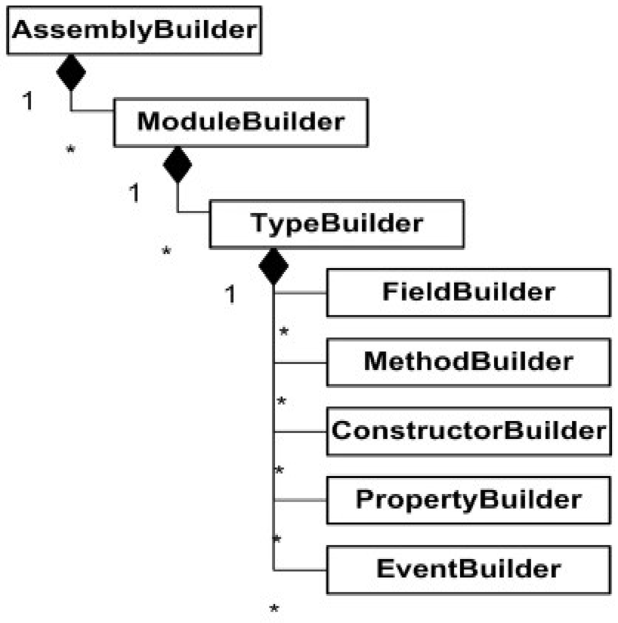


Fig. 6 Dynamic assembly as modelled by Emit library.

The major drawback with using CLIFile Reader is that the metadata in a .NET assembly is organised on a module basis. That is, type members are keyed with module-wide identifiers that do not immediately identify their containing type. In contrast, the Emit library expects a type to directly reference its constituents. To bridge these two views, we introduce wrappers for each object class in the Emit library hierarchy to provide both views, as shown in Figure 7.

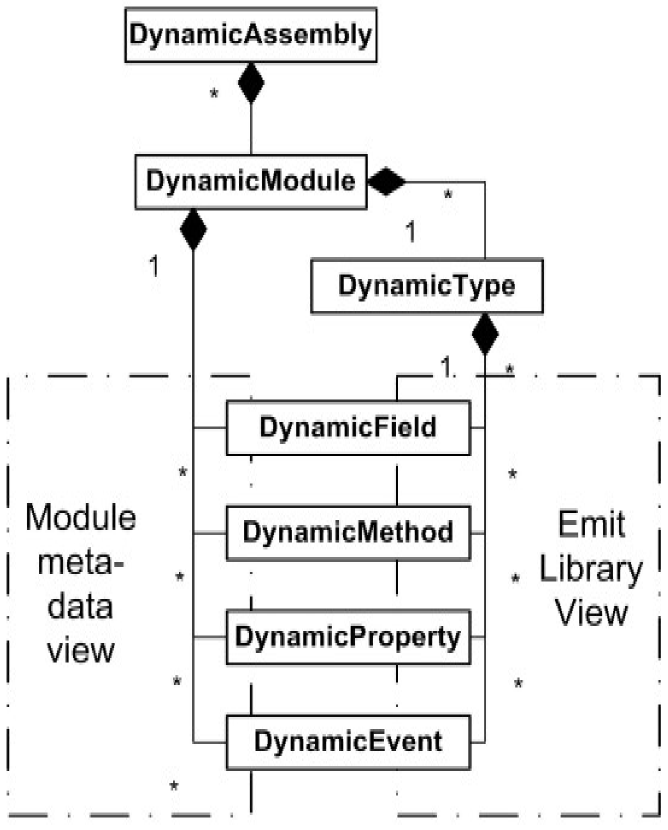


Fig. 7 Resolving Emit object hierarchy and CLI metadata indexing.

In this system, conversion to a dynamic assembly requires a complete traversal of the CIL of every method. This traversal gives the code generation system an opportunity to expose supported join points. The join points are modelled by the class hierarchy defined in Figure 8, where JoinPoint and JoinPointMethodSig are abstract classes. Currently, Weave.NET only exposes call and execution join points. As far as code generation is concerned, JoinPoint classes embed aspect advice by marshalling parameters and then calling the method that implements aspect advice. Embedding is requested by the code generation system before and after it emits the code corresponding to the join point. Separate classes are required to model each join point type as the opcodes required or marshalling parameters vary according to join point type.

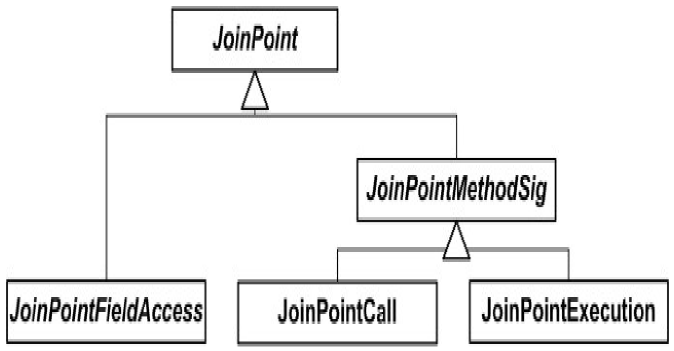


Fig. 8 JoinPoint class hierarchy.

Aspect instances are associated with class objects through a field added during code generation. Proper instantiation of aspect instances requires advance knowledge of which component types are associated with which aspect instances. Our single-pass weaver cannot determine this information in advance, which leads to the addition of potentially unused fields corresponding to aspect instances. Thus, our work on aspect instantiation is incomplete.

5. CONCLUSION AND FUTURE WORK

In this paper we describe the operation of Weave.NET from a programmer's point of view, and provides details on the underlying aspect model. The aspect model is drawn from AspectJ, while language interoperability is based on the Common Language Infrastructure (CLI) designed for the .NET Framework. The crosscutting statements of the aspect are written with an XML script based on the syntax of AspectJ, and they apply behaviour from the aspect's binary component. The weaver is implemented with two subsystems, one responsible for code generation and the other for aspect modelling. Language-independence was verified in service-side and client-side engineering scenarios. Specifically, logging, written in Visual Basic code, was added to the execution of methods in an I/O package written in Visual Basic. Weave.NET's CLI focus is shared by other technologies, but these do not match its language-independence capabilities. Neither do the implementations of other popular aspect models.

Future work in Weave.NET will involve broadening its crosscutting capabilities and reflection support to allow for more interesting aspect behaviour. While the aspect XML schema is complete, the full set of primitive pointcut designators and advice statements are not supported, which limits the effectiveness of our aspects. For example, our initial assessment noted proper logging requires signature specification be broadened to include accessibility modifiers. Also, testing indicates the need to make available a metadata object to provide aspects with reflective access to the join point's execution context.

6. REFERENCES

[1] Elrad, T., Filman, R.E. and Bader, A. "Aspect-oriented Programming". *Communications of the ACM*, *44* (10), pp.29-32, October 2001.

[2] Booch, G. *Object-oriented Analysis and Design with Applications*. Benjamin/Cummings, Redwood City, California, 1994.

[3] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C.V., Loingtier, J.-M. and Irwin, J., Aspect-Oriented Programming. In *The 1997 European Conference on Object-Oriented Programming (ECOOP'97)*, (Jyväskylä, Finland, 1997), Springer-Verlag, pp.220-242, *1997*.

[4] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J. and Griswold, W.G., An Overview of AspectJ. In *ECOOP 2001*, (Budapest, Hungary, 2001), Springer-Verlag, pp.327-355, 2001.

[5] ECMA International. *Standard ECMA-335 Common Language Infrastructure (CLI)*, ECMA Standard, http://www.ecma-international.org/publications/standards/ecma-335.htm, 2003.

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

[7] Schult, W. and Polze, A., Aspect-Oriented Programming with C# and .NET. In *5th IEEE International Symposium on Object-oriented Real-time Distributed Computing*, (Washington, DC, 2002), IEEE Computer Society Press, pp.241-248, 2002.

[8] Cahill, V. and Lafferty, D. *Learning to Program the Object-Oriented Way with C#*. Springer-Verlag UK, London, 2002.

[9] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J. and Griswold, W.G. "Getting Started with AspectJ". *Communications of the ACM*, *44* (10), , pp.59-65, October 2001.

[10] The AspectJ Team. *The AspectJ Programming Guide (V1.0.6)*, http://download.eclipse.org/technology/ajdt/aspectj-docs-1.0.6.tgz, 2002.

[11] Cisternino, A. *CLIFileReader Library*, C# Source Code, http://dotnet.di.unipi.it/MultipleContentView.aspx?code=103, 2002.