# An Aspect-to-Class Advising Architecture Based on XML in Aspect Oriented Programming

T. Hussain, M. M. Awais, S. Shamail, M. A. Adnan
*Department of Computer Science*
*Lahore University of Management Sciences, DHA, Lahore, Pakistan*

## Abstract

*In aspect oriented programming, cross-cutting concerns are represented as aspects. These aspects can advise classes as well as other aspects. This paper defines an XML schema to express aspect information for the aspects advising classes. Defining such schema provides a platform that is implementation independent and can be used for further research like aspect weaver development. In aspect weaving, conflicts amongst aspects may arise while advising classes. This paper proposes a solution that resolves aspect conflicts related to pre-compilation weaving. Two weaving algorithms based upon the proposed solution are also developed and explained.*

## 1. Introduction

Any problem domain consists of different concerns. These concerns may be physical, logical and functional entities. Physical and logical entities are known as concepts in object-oriented paradigm and lead to the specification of classes. Functionality-related concerns are distributed amongst different classes, with particular functions assigned to a distinct class. However, there are some functionality-related concerns that cannot be associated with a distinct class; and thus have a system-wide range. Such concerns are known as cross-cutting concerns and are discussed in [1, 2, 3]. Aspect-oriented programming techniques involve separation of these cross-cutting concerns from classes into groupings called aspects. There are defined certain points in program's execution where aspects can specify the desired actions to be taken. This is achieved through an advice that is the code that runs at a predetermined point when certain conditions are met. The program related to cross-cutting concerns (aspect code) is woven into the class code (core code) before execution of the main program. This process is known as weaving and is defined by a set of weaving rules. Weaving aspects are discussed in [4]. Aspect oriented extensions to object oriented compilers have been suggested in the literature as in [5, 6]. Figure 1 gives an overview of aspect-oriented software development process differentiating pre-compile weaving and post-compile weaving techniques. Aspects are typically identified when functionality is being assigned to the classes related to a problem domain under consideration. Aspects can be defined to advise classes or other aspects. Generally speaking there are two categories of advices: Aspect-To-Aspect (ATA) advising and Aspect-To-Class (ATC) advising. This paper primarily discusses ATC.

Schonger et al observes "Although AOP seems to become useful in solving real-world problems, the situation is not perfect from a research point of view as well as for some practical applications: there is no strong theoretical basis yet, the existing prototypes are always bound to a particular base language and the user usually has no support to extend the language" [7]. Therefore, in this paper, we first present an XML schema that can appropriately represent necessary information related to aspects. It provides a very compact and an implementation independent framework that can be effectively utilized for further research. Once an XML schema is developed, all aspects can be expressed in the form of XML documents and a validating XML parser can be used to check whether the XML description of an aspect follows the rules specified in the schema.

This paper also studies the resolution of aspect conflicts that may arise when two or more aspects advise a class. The order in which these aspects advise a class can affect the final results. A solution suggesting priorities is presented and the original XML schema is modified accordingly. Two different algorithms for this implementation are also proposed.

## 2. Basic XML Schema for Aspects

XML is used for description of aspects because of the ease with which it allows any information to be expressed according to any given structuring rules. Conformance to these structuring rules can be easily checked with a validating XML parser. XML also simplifies the task of

implementing a weaver because all aspect information can be picked from the XML document by an XML parser.

The XML schema given in Figure 2 defines information about aspects. Important issues regarding this definition are discussed as follows.

## 2.1 Identifying Aspects

The XML document may contain more than one aspect. So each aspect should be uniquely identified. This can be achieved using a unique numbering scheme or a unique name allocation scheme.

## 2.2 Class Advice and Other Class Information

Code to be woven into the core code is contained in the "advice" part of an aspect. The advice part is divided into segments on the basis of the locations at which the advice code is to be placed inside the core code. The XML schema defines the location of the core code along with class names to be advised.

## 2.3 Join Points

Join points are well defined locations in the core code where aspect code can be weaved. Only two join points are used for methods - before the start of the core code and after the end of the core code. Advice code added before the start of the core code is referred to as 'before code', while advice code added after the core code is known as 'after code'. The XML schema defines segments for 'before code' and 'after code'.

## 2.4 Weaving Granularity and Interface Width

Weaving granularity with the above specification of join points is quite coarse. A finer weaving granularity is required, for example, specifying join points in terms of statements. This means that advice code can be woven before or after a statement of the core code. A finer weaving granularity may not be recommended because it can impose more restrictions on data variables and on statements in the core and aspect code. Increasing these restrictions means that the core-aspect interface width is increased and freedom in designing classes and aspects independent of each other is decreased. In other words orthogonality and modularity are decreased [8].

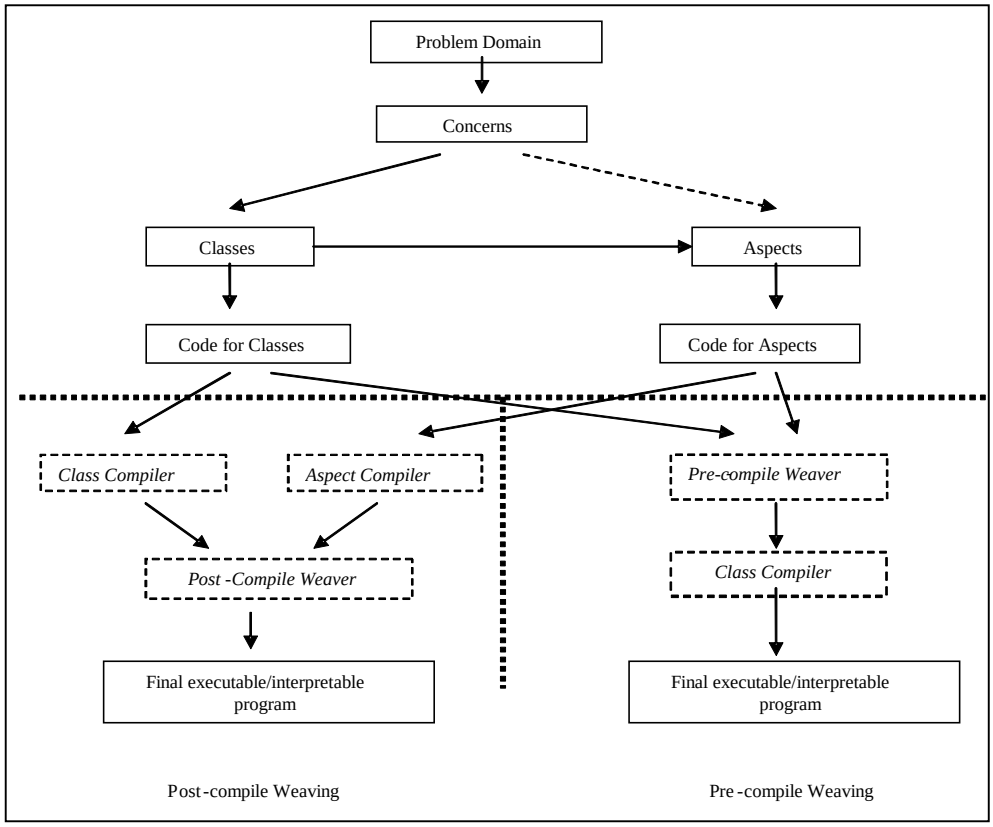


**Figure 1: An Overview of Aspect-Oriented Software Development and Programming**

```
<!-- XML Schema for Aspects -->
<Schema xmlns = "urn:schemas-microsoft-com:xml-data">

<ElementType name = "AspectList"  model = "closed"
 content = "eltOnly" order = "many">
<description>only Aspect elements contained</description>
<element type = "Aspect"/>
</ElementType>

<ElementType name = "Aspect" model = "closed"
 content = "eltOnly" order = "seq">
<description>Details of an Aspect</description>
<element type = "Extends" minOccurs = "0" maxOccurs = "1"/>
<element type = "Implements" minOccurs = "0" maxOccurs = "*"/>
<element type = "AdvisedClass" minOccurs = "0" maxOccurs = "*"/>
<element type = "Introduced" minOccurs = "0" maxOccurs = "1"/>
<element type = "Before" minOccurs = "0" maxOccurs = "1"/>
<element type = "After" minOccurs = "0" maxOccurs = "1"/>
</ElementType>

<ElementType name = "Extends"  model = "closed"
 content = "textOnly" >
<description>name of class extended</description>
</ElementType>

<ElementType name = "Implements"  model = "closed"
 content = "textOnly" >
<description>name of interface implemented</description>
</ElementType>

<ElementType name = "AdvisedClass"  model = "closed"
 content = "eltOnly" order = "seq">
<element type = "ClassId" minOccurs = "1" maxOccurs = "1"/>
<element type = "MethodId" minOccurs = "0" maxOccurs = "*"/>
<element type = "Location" minOccurs = "1" maxOccurs = "1"/>
</ElementType>

<ElementType name = "ClassId"  model = "closed"
 content = "textOnly" >
<description>name of class advised</description>
</ElementType>

<ElementType name = "MethodId"  model = "closed"
 content = "eltOnly" order = "seq">
<element type = "MethodName" minOccurs = "1" maxOccurs = "1"/>
<element type = "MethodSignature" minOccurs = "1" maxOccurs = "1"/>
</ElementType>

<ElementType name = "MethodName"  model = "closed"
 content = "textOnly" >
<description>name of method advised</description>
</ElementType>

<ElementType name = "MethodSignature"  model = "closed"
 content = "textOnly" >
<description>signature of method advised </description>
</ElementType>

<ElementType name = "Location"  model = "closed"
 content = "textOnly" >
<description>location of core code source</description>
</ElementType>

<ElementType name = "Implements"  model = "closed"
 content = "textOnly" >
<description>name of interface implemented</description>
</ElementType>

<ElementType name = "Introduced"  model = "closed"
 content = "textOnly" >
<description>code introduced into class</description>
</ElementType>

<ElementType name = "Before"  model = "closed"
 content = "textOnly" >
<description>before part of advice to method</description>
</ElementType>

<ElementType name = "After"  model = "closed"
 content = "textOnly" >
<description>after  part of advice to method</description>
</ElementType>

</Schema>
```

**Figure 2: XML Schema to Represent Aspects in ATC Advising**

## 2.5 Method Identification

In the XML schema, aspect code for methods of a class is specified by the method name and arguments list for the method. The arguments list allows method overloading to work.

## 3. Aspect Conflicts

A conflict between two or more aspects advising a class method can result only if the aspects codes share a variable or object and at least one of the aspects codes writes to the shared variable or object. A method of ensuring correct weaving using assertions is described in [9]. Another method based on priorities is presented below.

## 3.1 Aspect Priorities

In the case of conflicting aspects, the weaver needs to be informed about the correct order of weaving respective aspects in the code for a class method. This can be done by assigning priorities to conflicting aspects.

## 3.2 XML Schema with Priorities

XML schema given in Figure 2 is modified as follows. Priorities for 'before code' and 'after code' are added to the *MethodId* element of Figure 2 as *BeforePriority* and *AfterPriority* elements respectively. The modified code segment, **printed in bold,** is given in Figure 3.

## 3.3 Weaving Algorithms

Two algorithms defined as class-by-class weaving and aspect-by-aspect weaving are presented here.

### 3.3.1. Class-by-Class Weaving Algorithm

In this algorithm the process of weaving takes place in a class-wise sequence. All aspects advising a class are examined before moving on to another class. An array is maintained to record the introduce advice relevant to the class being considered from every aspect examined. Before code and after code of an aspect can be applied to multiple methods of the class. Also multiple aspects can advise each method. Two arrays (for the before and after advice) are maintained for each method of the class. Introduce advice does not have priorities and can be woven in the order it was collected (or any other order, as long as code from different aspects is not mixed). However before and after advice has to be sorted in order of priority before it is woven. For describing this algorithm, called ClasswiseWeaver, some additional notation is shown in Table 1.

**Algorithm ClasswiseWeaver**

```
for (every Class in ClassesCollection)
{
 create and initialize IntroducedArray  to empty;
 for  (every Method in Class)
 {
    create and initialize Method.BeforeArray  to empty;
    create and initialize Method.AfterArray  to empty;
 }
 while (unexamined aspects exist in AspectsCollection)
 {
    find next aspect Relevant  that advises Class;
    create object Introduced from information in
    Relevant;
    add Introduced to IntroducedArray;

    for (every Method of Class advised by Relevant)
    {
      create object Before from information in
      Relevant;
      add Before to Method.BeforeArray;
      create object After from information in Relevant;
      add After to Method.AfterArray;
    }
 }
 apply code from IntroducedArray to class;
 for (every Method of Class )
 {
    sort Method.BeforeArray according to priorities;
    apply code from  Method.BeforeArray to Method;
    sort Method.AfterArray according to priorities;
    apply code from  Method.AfterArray to Method;
 }
}
```

```
<ElementType name = "MethodId"  model = "closed"
 content = "eltOnly" order = "seq">
<element type = "MethodName" minOccurs = "1" maxOccurs
= "1"/>
<element type = "MethodSignature" minOccurs = "1"
maxOccurs = "1"/>
<element type = "BeforePriority" minOccurs = "0"
maxOccurs = "1"/>
<element type = "AfterPriority" minOccurs = "0"
maxOccurs = "1"/>
</ElementType>

<ElementType name = "BeforePriority"  model = "closed"
 content = "textOnly" dt:type = "int">
</ElementType>
<ElementType name = "AfterPriority"  model = "closed"
 content = "textOnly" dt:type = "int">
</ElementType>
```

**Figure 3: XML Schema with Priorities**

### 3.3.2. Aspect-by-Aspect Weaving Algorithm

In this algorithm, called AspectwiseWeaver, each aspect is examined and all its advice applied to the relevant classes after searching for them. An intermediate data structure is created for collecting before and after advice relevant to all methods of all classes. This data structure can be in the form of a multidimensional array where first dimension is the classes, the second dimension is the methods of a particular class, and the third dimension contains the Method.BeforeArray and Method.AfterArray objects (described in the previous section) for a particular method. Other notations used for describing this algorithm also have the same definitions as given in the previous section. Searching for classes is done in this data structure. Search operations can be speeded up if a tree like data structure is used. To keep the weaving algorithm independent of the data structure implementation, the data structure will be wrapped up in an abstraction called ClassStore. ClassStore has the operations defined as shown in Table 2.

**Algorithm AspectwiseWeaver**

```
for (every Aspect in AspectsCollection)
{
 for (every Class advised by Aspect)
 {
   // not all methods are advised
   for (every advised Method  of Class)
   {
      create Before object;
      // carry out operation on ClassStore
      InsertBeforeObject(Class, Method, Before);
      create After object;
      // carry out operation on ClassStore
      InsertAfterObject(Class, Method, After);
   }
 }
}
   // carry out operation on ClassStore
SortArrays(  );
   // Apply advice to core code
for (every Class in ClassesCollection)
{
 for (every Method of Class)
 {
      // carry out operation on ClassStore
   AdviceCollection = GiveBeforeAdvice(Class, Method);
   Apply advice in AdviceCollection to Method;
    // carry out operation on ClassStore
   AdviceCollection = GiveAfterAdvice(Class, Method);
   Apply advice in AdviceCollection to Method;
 }
}
```

**Table 1: Terms Used in ClasswiseWeaver Algorithm**

| | |
|---|---|
| `AspectsCollection` | An XML document containing all the aspects. |
| `ClassesCollection` | A collection of code for all the classes |
| `Class` | A class in ClassesCollection. |
| `Relevant` | An aspect in AspectsCollection that advises a particular class in Classes. Method is a uniquely identified method of Class. Uniquely identified means that its identity consists of method name and arguments signature. |
| `Introduced` | An object containing advice introduced by an aspect into a class. |
| `IntroducedArray` | An array of Introduced objects. |
| `Before` | An object containing before advice from a particular aspect and priority for a particular Method. |
| `After` | An object containing after advice from a particular aspect and priority for a particular Method. |
| `Method.BeforeArray and Method.AfterArray` | Arrays of Before and After objects containing advice for a particular Class.Method. |

**Table 2: Operations Defined by ClassStore**

| | |
|---|---|
| `InsertBeforeObject(Class, Method, Before)` | Inserts the specified Before object into the Method.BeforeArray of the specified Method of the specified Class. |
| `InsertAfterObject(Class, Method, After)` | Inserts the specified After object into the Method.AfterArray of the specified Method of the specified Class. |
| `SortArrays( )` | Sorts, according to priorities, every Method.BeforeArray and Method.BeforeArray in ClassStore. |
| `GiveBeforeAdvice(Class, Method)` | Returns sorted before advice for the specified Method of the specified Class. |
| `GiveAfterAdvice(Class, Method)` | Returns sorted after advice for the specified Method of the specified Class. |
| `AdviceCollection` | the final ordered collection of all before or after advice from all aspects that is to be woven into the core code. |

## 4. Conclusions and Future Work

The descriptive power of XML has been illustrated in the domain of aspect description. The flexibility and extensibility of basic XML schema has been demonstrated to easily incorporate new concepts like the solution presented for resolving aspect conflicts.

The XML-based architecture developed focuses on multi-aspect advising. A schema can be developed on similar lines for aspect-to-aspect advising and combined multi-aspect and aspect-to-aspect advising. These capabilities will allow for the capture of more issues from a problem domain for more flexible aspect-oriented design.

The most obvious direction of any future work on the topics covered in this paper is the development and implementation of a complete weaver. However if aspect-oriented programming is considered in general there are many promising areas (especially since aspect-oriented programming is still in its infancy):

Aspect-oriented programming can be studied in the context of the structured programming paradigm. This area has been largely neglected because most of the people currently working on aspect-oriented programming have backgrounds in object-oriented software development and they have looked at aspects from the point of view of classes. The idea of aspects may have originated from classes but this does not imply that aspect-oriented programming cannot be used to enhance the structured programming paradigm.

Work can be carried out on application of formal methods to aspect-oriented programming. For example, the correct advising issues discussed in this thesis can be studied using a formal methods approach.

## 5. References


[1] G. Kiczales, J. Lamping, A. Medhakar, C. Maeda, C. Lopes,J. Loingtier, J. Irwin, "Aspect-Oriented Programming", Proceedings of the European Conference on Object Oriented Programming, number 1241 in Lecture Notes in Computer Science, Springer Verlag, June 1997.

[2] T. Highley, M. Lack, P. Myers, "Aspect-Oriented Programming: A Critical Analysis of a New Programming Paradigm", University of Virginia, Department of Computer Science Technical Report CS-99-29, May 1999.

[3] L. Carver, W. Griswold, " Sorting out Concerns", First Workshop on Multi-Dimensional Separation of Concerns in Object-Oriented Systems, Denver, Colorado, November 1999.

[4] K. Bollert, "On Weaving Aspects" , ECOOP '99 International Workshop on Aspect-Oriented Programming, 1999.

[5] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, W. Griswold, "An Overview of AspectJ", Proceedings of the European Conference on Object-Oriented Programming, 2001.

[6] O. Spinczyk, A. Gal, W. Schroder-Preikschat, "AspectC++: An Aspect-Oriented Extension to C++", International Conference on Technology of Object-Oriented Languages and Systems, Sydney, Australia, 2002

[7] S.Schonger, E.Pulvermuller and S.Sarstedt, "Aspect-Oriented Programming and Component Weaving:Using XML Representations of Abstract Syntax Trees", Second German AOSD Workshop, Bonn, Germany, 2002.

[8] R.J. Walker, E.L.A. Baniassad, G.C. Murphy, "An Initial Assessment of Aspect-Oriented Programming, Proceedings of the 21st International Conference on Software Engineering, ACM Press, 1999, pp 120-130.

[9] H. Klaeren, E. Pulvermuller, A. Rashid, A. Speck, "Aspect Composition applying the Design by Contract Principle", Second International Symposium on Generative and Component-Based Software Engineering, GCSE 2000, Erfurt, Germany, October 2000.