feature

Aspect.NET

New approach to aspect-oriented programming supports component reuse and modification

BY VLADIMIR SAFONOV

Aspect-oriented programming (AOP) is a new programming paradigm that supports software component reuse and modification. Each application can be regarded as a collection of implementations of ideas or concerns.

Some concerns can be implemented as a programming module or a collection of modules. We'll refer to these modular concerns. Other concerns are referred to as crosscutting concerns. They cannot be implemented by a set of modules only but also require the addition of program fragments, usually executable statements, to the modules implementing other concerns. Two examples:

- Implementing a new construct of a language (e.g., adding generics to C#)
- Making a library MT-safe

The need for crosscutting concerns was realized in 1970s. A. Fouxman formulated the concept of a vertical cut as a set of spread actions that implements some extension function.

AOP originated in the 1990s during the age of commercial use of object-oriented programming (OOP). A group at Xerox Palo Alto Research Center supervised by Gregor Kiczales introduced the notions of crosscutting concerns and aspect. Kiczales' group developed world's first AOP tool, AspectJ, based on Java. IBM's HyperJ is another Java-based AOP project. The AOP Web site, http://aosd.net, contains a variety of other references.

The approach to AOP presented here is based not only upon the ideas of classic AOP projects but also on my 25 years of experience in developing compilers, software engineering tools, knowledge engineering, and Java technology.

A concern is a collection of knowledge on solving some task in an application domain. An aspect is an implementation of a crosscutting concern in some programming language (e.g,. C#).

An aspect consists of several modules, data, and actions. Each of the actions is to be woven into the underlying program according to some preliminary formulated rule.

In terms of knowledge engineering, an aspect can be regarded as a hybrid collection of knowledge. Procedural knowledge is used to define the implementation of the aspect's modules and actions, conceptual and heuristic knowledge – to specify the application domain of the aspect and the rules of weaving the aspect's actions into a program.

Actually all programmers use aspects in their everyday work but usually in some implicit way. As a result not enough attention is paid to making the task of locating and updating aspects easier. This makes program debugging and maintenance costly. One reason is lack of AOP functionality in programming languages and tools. Another problem is reusability of aspects. Not only modules but also aspects may be useful in many programs or be applied several times to parts of the same program.

A simple example is a semaphore-based synchronization aspect. It can be implemented by a module (class) defining operations P and V, and by two actions – calls for operations P and V. The rules of weaving these actions are as follows: Operation P is to be called before an update of the common resource; Operation V is to be called after an update of the resource.

Aspect definitions and weavings are the basis of aspect-oriented programming, which consists of aspect-oriented analysis, design, and implementation.

Unlike a traditional module whose call is located at some definite point of the program, aspect weaving is spread around the program and crosscuts a lot of modules. The semantics of aspect weaving can be regarded as inserting the aspect's modules, data, and actions into the appropriate points of the program, a kind of "crosscutting macro expansion."

From a more general viewpoint, aspects are "orthogonal" to any programming languages or paradigms. The notion of aspect is as deep, general, and fundamental as the con-

cepts of module or abstract data type. The notion of aspect relates to programming methodology and software architecture in general rather than to concrete paradigms or languages. So there can be aspects in OOP, procedural, functional, or logic programming.

In terms of knowledge engineering, aspect weaving rules can be regarded as metaknowledge - "higher order" knowledge on how to use the knowledge implemented by programming modules.

So an ideal solution would be to implement aspects in a generally reusable form that goes beyond concrete programming languages. There should be a unified AOP meta-language for aspect definition and use in various programming languages, and a unified aspect supporting APIs for implementing aspects in different languages. This approach will help to make aspects what they should be – cross-language reusable modules.

Currently, Microsoft .NET the only platform that has the general mechanisms to make this happen. In .NET, regardless of how the aspects are syntactically expressed in programming languages, they can be represented in the general form of MSIL (implementation of aspect's modules) and metadata (aspect structure and attributes).

The approach taken in AspectJ (aspect extension of Java) is limited by restrictions of the Java technology. So the opportunity to define and use aspects such as metaknowledge is completely lost and AOP is limited to "Java aspects" only.

Requirements for an AOP meta-language are as follows:

- Simplicity
- Conceptual economy
- Independence of concrete programming languages and paradigms.

One of the most important tasks of AOP, which hasn't yet been paid enough attention is aspectizing – transforming a non-aspect–oriented program into an aspect-oriented one. Aspectizing is decomposition of a program into a collection of aspect definitions and weavings. The purpose is to make analysis, maintenance, and development of programs easier.

An aspectizer can be based on a set of "starting" source file names of the aspect, and a set of wildcards describing its identifiers. Then, the aspectizer must take into account all modules called from underlying source code and suspected of belonging to the aspect, and so on. Finally, the aspectizer should suggest to the user its version of decomposing the program into aspects. But the users should be able to make the final decision and, if necessary, to manually aspectize parts of the program using GUI.

Requirements for an AOP Tool:

- Ability to define and weave aspects
- Reusability of aspects
- Aspect visualization, modification, adding and deletion through GUI
- The options to check the results of aspect weaving and to alter it manually
- Implementation of AOP meta-language
- Aspectizing, automated or manual

Some features of AspectJ conflict with these goals, such as the ability to introduce new local fields and methods to any class, and arbitrary self-modification of programs around join points. To make AOP modular I suggest the following requirements:

- Aspect definition shouldn't contain definitions of data to be inserted into other modules during weaving.
- Aspect definition without AOP meta-language annotations should be in a correct and complete compilation unit, e.g., a C# namespace. An aspect's data should be localized within it and defined as variables, fields, or properties.
- An aspect should not be an analog of a class, or a procedure, etc. AOP functionality should not be an extension or a substitute of constructs of any implementation language.
- Using aspects should make understanding and modifying programs easier, rather than trickier.
- AOP shouldn't:
  –Define an aspect's methods apart from its class
  –Create aspect instances; instead, aspect application should be used
  –Define aspect constructors; initialization of an aspect's data should be performed in its modules
  –Mix aspects and classes together in a hierarchy
  –Apply an aspect to an object, rather than to the code of the whole class;
  –Modify the program "around" join points, except for inserting aspect's actions.

An appropriate technique to make aspects more general is generics, first introduced into CLU language in the 1970s. Generic aspects should be part of AOP meta-language. For example, the synchronization aspect could be parameterized by synchronization conditions. By instantiation we get a more specific aspect of synchronization by a concrete resource or condition.

AOP Meta-Language

Syntax and semantics of AOP meta-language are made as simple as possible. Each of its constructs (an AOP annotation) is written as a separate line of the source code and starts with a special keyword. All AOP keywords start with "%" to easily separate them from the rest of the source. If AOP annotations are deleted or commented out, the source remains a correct compilation unit. AOP meta-language is equally applicable to C#.NET or VB.NET. The example shown in Listing 1 is a generic Synchronization aspect definition. AOP annotations are in bold.

An example of weaving the aspect:

```
%to MyNameSpace %apply
Synchronization <%assign
Res.buffer>
```

Weaving with another synchro-

nization condition, an assignment to the field Res.r1 or Res.r2, looks as follows:

```
%to MyNameSpace %apply
Synchronization <%assign Res.r1,
Res.r2>
```

Defining aspects using AOP meta-language should not be the only way to introduce a new aspect. With an AOP tool all features should be also available via GUI.

Representing Aspects for Microsoft .NET

The best way to represent aspect metaknowledge is to use metadata with custom attributes. Custom attributes provide a way to keep any annotations to programs, for example, AOP annotations.

Due to .NET architecture, extra information on aspects represented as custom attributes will not interfere with the work of the CLR, compilers, or other .NET tools. Aspect metadata is used only by AOP tools, and other tools will just ignore it. However, unlike metadata, it is not possible to attach custom attributes to CIL instructions. No doubt such feature could be useful – and not only for AOP purposes – but it could decrease runtime performance of .NET.

So, while it's possible to keep AOP information for definitions, there is no straightforward way to indicate that a piece of CIL belongs to an aspect. Neither is it possible to attach mnemonic labels or comments to the binary format of CIL.

The solution is as follows. CIL code contains metadata tokens – references to named entities used in the code. A tool like AbsIL or PEReaderWriter should be used to analyze CIL instructions and metadata, extract metadata tokens from CIL instructions, and recognize from their custom attributes to which aspect the CIL code belongs.

In general any of the actions of an aspect may contain not only references to the aspect modules and data, but also references to other aspects or to standard APIs such as System.Console::WriteLine. I suggest requiring each of the statements of an aspect's actions to refer to its modules or data, or to be represented as a separate method or function within the aspect compilation unit. Otherwise aspect actions after weaving, for example, an explicit WriteLine call, will be "dissolved" within the program's CIL code, leaving no information on the applied aspect.

So, if we are to implement, say, a Logging aspect with two actions that log the start and the end of a call, we should define two methods, LogStart and LogFinish, within the aspect compilation unit. Only in this case we are guaranteed that the CIL code of the aspect woven into any program will contain the metadata token for this local aspect's method, so the program will recognize that this CIL code belongs to the Logging aspect.

Another solution is using ILAsm source code as an intermediate representation of a program in an AOP tool. Unlike CIL binaries, ILAsm code contains mnemonic labels for each instruction and may contain comments. The usual format of ILAsm label is IL_n, where n is the instruction number. Replacing this label with a mnemonic one containing the name of the aspect and a unique id of this concrete weaving of the aspect will not crush ILAsm utility. But as a result ILAsm will generate CIL binaries without any trace of the above labels or comments, with all references to CIL instructions replaced by their offsets, signed integers.

Besides, using ILAsm/ILDASM and another intermediate form of the program can decrease performance of AOP tools. In compiler terms, ILAsm code would require several extra passes of the whole program, whose size may be very large.

In addition, using ILAsm code is not convenient for users who prefer to work in terms of the source code in C#, VB, or another high-level language.

However, "assembler-level aspects" with AOP annotations in meta-language and the implementation of aspects in ILAsm code may be useful themselves.

Theoretically a possible solution would be to keep for each module a custom attribute AppliedAspectActions, a list of all weavings of aspect actions within a module. But it doesn't correspond to either .NET architecture or to general principles of databases. If the program is modified by a non-AOP tool, the values of the AppliedAspectActions attribute may become out of date.

Keeping in mind all of the foregoing, the following architecture for an AOP tool for the .NET platform appears to be the best.

- Any named and typed entity in a program (class, field, method, etc.) shall have an AspectRef custom attribute in the Name field stating the name of the aspect to which the given entity belongs. Values of the AspectRef attribute can be defined in the source code either by the "syntactic sugar" of AOP meta-language or by the common syntax of .NET attributes, by an expression within square brackets for C#, or in angular bracketss for VB.NET.
- An aspect definition in the form of a compilation unit plus AOP annotations, as discussed earlier, is compiled into a PE??? assembly file containing aspect CIL code and metadata. Compilation of AOP annotations can be done as part of the overall compilation process by the common C# or VB compiler, or as a separate preprocessing phase. For each of the items of the aspect definition, an AspectDef custom attribute is defined, with three main fields: the Name of the aspect; the Role of the item within the aspect (module, data, action), and its Value (role-specific information contained in a string). For an action, its Value is the weaving rule. For the main module the Value contains the aspect's formal parameter list, if it has one. On processing, the aspect, its header items, and the weaving rules are transformed into the values of the AspectDef attribute. The actions are transformed into methods in CIL code, with custom attributes indicating the role of the method

(as action) within the aspect and the weaving rule of the action.
- Applying an aspect is implemented by copying the CIL code and metadata of its actions into appropriate join points of the assembly file of the program using the aspect. Which aspect a CIL instruction belongs to should be determined by analyzing its metadata tokens.
- An important user interface issue is how to map the representation of aspects back to the program source. Users should not have to learn or handle aspects at the CIL code level and should have an opportunity to work with aspects in terms of the source code of the language being used. Surely decompilation should be avoided as the method of such mapping. So, two solutions are possible: to rely on debugging information and the appropriate options of the compiler to generate the mapping, or to have a special custom attribute (say, Source) for each module to hold the reference to its source or a copy of its source. The former method seems appropriate to .NET architecture; the latter is unacceptable for the same reason as mentioned regarding the AppliedAspectActions attribute.
- The task of aspectizing a program is equivalent to evaluating the AspectRef attribute for each of its named and typed entities, and also identifying the aspect(s) for each of its CIL instructions (or source code constructs).

Now consider the earlier example in C# in "AOP-preprocessed" form, with the AOP annotations replaced by C# attribute usage constructs. In Listing 2, AOP is the AOP API provided by Aspect.NET.

Conclusion

The above techniques are the basis of the new AOP tool, Aspect.NET, being developed for .NET platform due to a grant provided by Microsoft Research. The plan is to issue the first working version by summer 2003. In perspective, I will be interested to see not only further development of the Aspect.NET tool but more general research work on further integration of AOP and .NET, and on a closer relationship between AOP and knowledge engineering. ◉

References

- *Aspect-oriented programming:* http://aosd.net.
- Laddad, R. (2002). "I Want My AOP." Parts 1, 2, and 3. – *Java World*, issues. 1, 3, 4.
- Fouxman, A. (1979) "Technological Aspects of Program Systems Development." Finances and Statistics Publishing Co.
- *The aspectj project:* http://aspectj.org
- *Hyper/JTM: Multi-Dimensional Separation of Concerns for Java:* www.research.ibm.com/hyperspace/HyperJ/HyperJ.htm
- Safonov, V.O. (1988). *Programming Languages and Techniques for the Elbrus System*. Science Publishing Co.
- Safonov, V.O. (1992). *Expert Systems – Intelligent Advisors of Human Experts*. Knowledge Publishing Co.
- Safonov, V.O. (2002). *Introduction to Java Technology*. Science Publishing Co.
- *Microsoft .NET:* www.microsoft.com/net
- Highley, T.J.; Lack, M.; Myers, P. (1999) "Aspect Oriented Programming. A Critical Analysis of a New Programming Paradigm."

Vladimir Safonov received his master's degree in computer science at St. Petersburg University in 1977. He has been with the university for 25 years and is now a professor of computer science, and the author of 5 books. Vladimir's research interests include the Microsoft .NET platform, programming technologies, compiler development, and knowledge engineering.

v_o_safonov@mail.ru

Listing 1

```
 %aspect Synchronization
<ResourceUpdate>

 public class Synchronization
 {

 %modules

  public class Semaphore {

          private bool Open =
true;

          public void P()
          {
             lock (this) {
                Open = false;
                while (!Open) {

System.Threading.Monitor.Wait(this
);
                }
             }
          } // P

          public void V()
          {
             lock (Open) {
                Open = true;

System.Threading.Monitor.Pulse(thi
s);
             }
          } // V

       } // Semaphore

 %data

  public static Semaphore s = new
Semaphore();

 %rules

       %before ResourceUpdate
       %action
    public static void
lockAction() { s.P(); }

       %after ResourceUpdate
       %action

    public static void
unlockAction() { s. V(); }

 } // Synchronization
```

Listing 2:

```
using AOP;

[AspectDef
  (Name = "Synchronization",
  Role = "MainModule",
  Value ="Param:<resourceUpdate>")
]

public class Synchronization
{

 [AspectDef
   (Name = "Synchronization",
   Role = "Module"
   )
  ]

 public class Semaphore { ... }

 [AspectDef (Name =
"Synchronization", Role = "Data")]

 public static Semaphore s = new
Semaphore();

 [
 AspectDef
  (Name = "Synchronization",
  Role = "Action",
  Value = "%before resourceUpdate"
  )
 ]

 public static void lockAction() {
... }

 [
 AspectDef
  (Name = "Synchronization",
  Role = "Action",
  Value = "%after resourceUpdate")
 ]

 public static void unlockAction()
{ ... }

    } // Synchronization
```