Aspect.NET:
Concepts and Architecture

Aspect-oriented programming for .NET

BY VLADIMIR SAFONOV

HOME

CLIENT

SERVER

EVERYWHERE

his article presents a new approach to aspect-oriented programming (AOP) – an advanced programming paradigm and technology that supports the separation, design, implementation, location and updating of crosscutting concerns implemented as aspects. The most common AOP tools – AspectJ, HyperJ, etc. contain a lot of AOP foundation ideas and useful features, but they are focused on Java technology and are currently available as AOP extensions of Java. My approach, on the other hand, takes full advantage of the .NET architecture. It's based on the following principles: aspects for Microsoft.NET, as well as applications, should be cross-language; they should be defined by simple language neutral AOP annotations (or AOP meta-language); representing aspects for .NET should be based on custom attributes; and there should be an aspectizer tool to locate aspects in existing non aspect-oriented applications and turning them into aspect-oriented (similar to "aspect mining"). Finally, I describe design principles and the architecture of Aspect.NET – an AOP framework for .NET – and present a simple example of an aspect and its weaving.

▶ Origin and Related Work

AOP, as a new prospective programming paradigm, has attracted attention since the mid 1990s when Gregor Kiczales and his Xerox PARC team became the pioneers and the classicists of AOP. They developed its basic concepts and the first AOP tool, AspectJ. AOP and its related approaches – generative programming, adaptive programming, etc. –

have become so popular for several reasons. First, there is a growing need to make the process of developing, maintaining, and extending complicated software easier and more systematic. Second, object-oriented programming (OOP) can't solve all of the problems of complicated software. In particular, it can't adequately support crosscutting concerns – features that can't be implemented by a generalized procedure (function, macros, class, etc.). Informally speaking, a crosscutting concern is any new functionality that can't be implemented by simply adding a new class of function, but the implementation is "spread" (or "tangled") among the application's code.

The code to implement a crosscutting concern – typically, a collection of components and tangled pieces of code, such as method calls or definitions – is referred to as an aspect. There are a plenty of examples of crosscutting concerns both in research and in commercial software. For instance, any code developed by the SSCLI team to port Rotor to MacOS, or any code developed by the Gyro team to implement generics in Rotor can be regarded as aspects.

In the mid-1990s, when Java was the most advanced software development platform, the first AOP tools were developed as AOP extensions of Java. In particular, AspectJ provides a variety of Java extensions: aspects (as explained previously); pointcuts – collections of join points; advices – actions belonging to an aspect to be executed when an appropriate join point is reached; and dynamic updates of control flow before, after, or instead of the join points code, etc. The basic idea

of AspectJ is as follows: when a join point is reached, an appropriate advice code belonging to one or several aspects is executed. The concept of dynamic join points by Gregor Kiczales can be regarded as an extension of the concept of breakpoints, as in many debuggers. However, Gregor Kiczales invented a lot of things to extend this approach; his main new concept is a special kind of module to define an aspect. Very important is his summarizing paper in which he offers a systematic view on known AOP tools – AspectJ, HyperJ, Demeter, DemeterJ, and AOP models – as well as PA, TRAV, COMPOSITOR and OC.

Another set of problems related to AOP is aspect mining, or, as I prefer to call it, aspectizing. – locating and "coloring" aspects in existing non-aspect–oriented applications. Aspectizing could help to make the code of the existing applications more readable and easily maintainable. There are several AOP aspect mining tools based on Java: Aspect Mining Tool (AMT), Aspect Browser, and FEAT.

When .NET was developed, it became desirable to use aspects to facilitate language interoperability for .NET. There are now a number of research projects related to AOP support for .NET. They can be subdivided into three groups, in accordance with how aspects are represented:
- using XML schemas to define AOP specifications, such as Weave.NET;
- using interceptors (in COM+ style) to dynamically insert and activate AOP functionality; the configuration of the system is described by XML files
- using custom attributes to represent aspects

44

Another prospective research project related to AOP is being done now, on aspects with artifacts, and how AOP relates to component engineering and application engineering.

▶ Principles of the Aspect.NET Approach

I have a somewhat different approach to AOP, taking full advantage of the .NET architecture.

Based on this approach, both aspects with dynamic join points and aspects with "static" join points (whose weaving works as "tangled" macro expansions) can be implemented.

Alongside with the dynamic join points approach, "static" join points are also important for developing, maintaining, updating, and analyzing applications in AOP style.

A lot of existing applications, as well as those being developed now, suffer greatly from "tangled" aspect fragments, which should at least be "colored" to make further progress of maintaining and developing such applications more systematic. This can be done both by defining new aspects and by aspectizing the existing software components and models. Another possible alternative, which I have been trying to follow, is to combine both approaches in one AOP model and tool.

My approach to AOP is based on the following main ideas:
- *"tangled" macro expansions and dynamic join points*
- *language neutral AOP annotations:* AOP annotations and AOP meta-language that are independent of the programming language being used
- *using custom attributes to tightly attach AOP annotations to the code they relate*
- *cross-language aspects interoperability:* multi-language aspects and CIL level aspects being used together
- *aspect-oriented knowledge man-agement:* treating and processing aspects as meta-knowledge that consists of *procedural knowledge* – implementation of components that are parts of aspect definition, *conceptual knowledge* – domain-specific knowledge on the application domains (formal specifications), and *heuristic knowledge* – the weaving rules for aspect actions.

Tangled Macro Expansions

While many of the useful AOP features (e.g., error handling) are comfortable with dynamic join points, there's a growing need to "color" aspects in existing and upcoming applications that adequately corresponds to a "static" (tangled macro expansion) AOP model. With this model, aspects can be clearly seen and updated by an AOP GUI-based tool. On the other hand, my approach to AOP is open to the "dynamic" treatment of join points.

Language Neutral AOP Annotations

AUTHOR BIO
Vladimir Safonov is a professor of computer science, doctor of technical sciences at St. Petersburg University with 25 years of active work leading major commercial and research projects on compiler development, software engineering methodologies and tools (including abstract data types and modular programming), knowledge engineering, OOA/OOD/OOP, Java techhnology, Web technologies, Microsoft.NET.

▶ *v_o_safonov@mail.ru*

Coming Next Month .NET JOURNAL

Microsoft's Smart Personal Objects Technology
.NET wristwatches really hit the SPOT
By Donald Thompson

Microsoft CRM Mobile
Data exchange in a service-oriented architecture
By Arif Kureshy

Developing Orientation-Aware Smart Device Applications with Visual Studio 2005
Would you like that application portrait or landscape?
By Jim Wilson

Location-Based Services
.NET knows where you are
By Paul Harris

Most of the currently known AOP tools are based on their own specific languages, both for specifying aspects and defining their implementation. One of the goals of my Aspect.NET project, in the spirit of .NET, is to make aspects really cross-language. For this purpose, there's a simple AOP "meta-language" that can be used for specifying aspects regardless of their implementation language (C#.NET, VB.NET, C++.NET, JScript.NET, J#.NET, etc.). AOP specifications defined in our AOP meta-language can be regarded as AOP annotations that don't really depend on the aspects' implementation language. I only use self-evident and language-neutral notions as "compilation unit," "programming module,", "data definition," and "generics."

Using Custom Attributes

Compared to XML-based approaches, implementing aspects for .NET using custom attributes is more reliable and justifiable. Custom attributes are intended for any program annotations related to specific parts of applications (assemblies, classes, methods, etc.), in particular, to AOP annotations. Their advantages are their clarity and self-evidence as well as their close relation and inherence to the program parts to which they tied. Custom attributes are always tied to the program units and fragments they relate, whereas XML schemas can be very easily lost or "forgotten" during any program transformations. More clearly and concretely, custom attributes always accompany any kind of program transformations and are used only when appropriate (by AOP tools only, or by any kind of tools "understanding" these attributes) and can be ignored by any other common use .NET tools (debuggers, garbage collectors, etc.).

Cross-Language Aspects Interoperability

As mentioned previously, cross-language aspects interoperability is one of the primary reasons for my approach to AOP. Currently, the only appropriate common language to express aspects for .NET is CIL (plus metadata). It's quite suitable for system programmers who may define "CIL level" aspects, due to the OOP nature of CIL, but it's not quite appropriate for a number of end users who are tied to their favorite languages (C#, VB, etc.). The decision is to allow for "CIL/metadata level" aspects, but to implement conversion or transformation of "language neutral" AOP "syntactic sugar" (AOP meta-language) to language-specific custom attributes definitions (e.g., the appropriate C# constructs in square brackets, or VB.NET constructs in angular brackets). The idea behind this approach is that AOP custom attributes can be regarded as language neutral annotations or even comments to the source of an application. If necessary, we can forget about aspects, comment out the AOP annotations (which is easy to implement, because of their simplicity and their keywords starting with "%") and use them for information only. On the other hand, software developers who will use our approach to AOP can just ignore our "syntactic sugar" (suppose they just don't like any kind of sugar) and implement aspects explicitly in their applications using AOP attribute definitions in a language-dependent form.

Aspect-Oriented Knowledge Management

Let's forget that almost all existing AOP tools are based in Java. In more general viewpoint, aspects can be regarded as hybrid knowledge, or even meta-knowledge because they contain information how to apply them, or, in other words, the procedural knowledge implemented in their definitions, to the existing applications – in our terminology, weaving rules, in classical terms – pointcuts and join points. Taking this viewpoint, aspects can be considered as a special kind of modules, in our approach – compilation units of any implementation language used, accompanied by AOP annotations, no matter expressed in "syntactic sugar" or "language dependent" form.

In perspective, we think such treatment of aspects can provide the basis for further research in what we call aspect-oriented knowledge management - research on using formal specifications as part of aspect definitions.

▸ **Summary of Features of Aspect.NET**

Based on the above, we are implementing the following features in Aspect.NET:
- defining and weaving aspects, including generic (parametrized) aspects
- supporting reusability of aspects – in particular, the ability to apply aspects to a whole application, or a selected class or namespace, etc.
- GUI for aspects visualization, modification, adding, weaving, and deletion
- checking the results of "automatic" aspect weaving, to undo part of these modifications and to perform them by hand
- implementation of AOP meta-language; in the first release – as a pre-processor for C# that converts "syntactic sugar" AOP annotations to "C# dependent" ones – constructs of defining and using appropriate custom attributes
- aspectizing (automatic or partially by hand) with the appropriate GUI.

Aspect GUI enables the users, first, to check and "manually" control the results of aspects weaving by hand (to prevent unclear results of weaving when using wildcards), and, second, to initiate, check and "manually" control the results of aspectizing, to enable an appropriate start of aspectizing with a reasonable set of types or regular expressions wildcards, and to prevent unclear and inappropriate results of "automated" aspectizing.

In Aspect.NET, an aspect definition is a compilation unit of the implementation language being used (C#, VB.NET, etc.) – namespace, class, etc., plus AOP annotations (in AOP meta-language). Aspect definition consists of:
- header (with optional formal parameters);
- data (optional);
- modules (classes, methods, functions, etc.);

- actions (method calls or just statements);
- weaving rules for each action in AOP meta-language (suggestions where and how to join the action to the application or its module subject to the weaving operation). An aspect weaving can be expressed by either a construct of an AOP meta-language, or by aspect GUI menu/window, etc. GUI item (gadget). Semantically, an aspect weaving consists of:
- the name of the underlying application or its module to which the aspect is applied
- the name of the aspect (definition) to weave
- (optional) the aspect actual parameters corresponding to its formal parameters in the aspect definition, if any.

In Aspect.NET, aspect definitions are treated as definitions of modules, actions and the weaving rules for each of the actions that can be applied to any application or to its class, etc. as a kind of "tangled" macro expansion, with the actual parameters (if any) substituted instead of aspect's formal parameters.

As a pointcut-like construct, we introduce rule sets. A rule set in Aspect.NET is a special kind of module (with AOP annotations only) for defining a set of join points that can be used for weaving any aspect.

Now we can briefly summarize the architecture of the first release of Aspect.NET:

- **C# preprocessor:** Converts AOP annotations into AOP attribute definitions. It works with C# source code. The current version of the preprocessor is implemented on the basis of C# regular expressions handling features.
- **Aspect weaver:** Performs the command "Weave aspect A [parametrized by <FP>] [with actual parameters <AP>] into a program, class, method, … P". It works with PE files, including AOP attributes.
- **Aspectizer:** Converts a non-aspect–oriented program to aspect-oriented. It takes an initial list of source code files and a list of wildcards to recognize identifiers. It works with C# source

codes and with PE files.
- **Aspect editor:** Performs the operations: "locate an aspect"; "update an aspect"; "delete an aspect". It works with PE files, including AOP attributes.
- **Aspect visualization GUI:** Provides interactive user interface for the above components. Visualizes different aspects in different colors.

To illustrate our approach, consider an example of C# definition of a useful aspect that performs logging of some action in an application – a call of a given method, an assignment to a given field, etc.

Please note that Aspect.NET features allow to define that aspect in the most general form – as parametrized by the logging condition and the appropriate message to be issued. So there is no need to define a lot of specific logging aspects for more concrete logging conditions instead.

The definition of the aspect looks as follows (fragments of the AOP meta-language are in bold):

```
%aspect Logging <LoggingCondition,
LoggingMessage>

public class Logging
{

%modules

    public static void LogStart()
    (

System.Console.WriteLine("Starting "
+ LoggingMessage);
    }

    public static void
LogFinish()
    (

System.Console.WriteLine("Finishing
" + LoggingMessage);
    }


%rules

    %before LoggingCondition
    %action
    public static void
logStartAction()
{Logging.LogStart();}
    %after LoggingCondition
```

```
    %action
        public static void
logFinishAction()
{Logging.LogFinish();}

} // Logging
```

Example of weaving the Logging aspect:

```
%to MyClass %apply Logging <%call
MyMethod, " MyMethod call">
```

The effect of the above AOP meta-language construct is as follows: before and after calling the given method, a logging message of starting and finishing the call will be issued. Weaving the same Logging aspect with another logging condition - an assignment to the given public static field named MyField, and another related message is activated in AOP meta-language the following way:

```
%to MyClass %apply Logging
<%assign MyField, " MyField assign:
value =" + MyClass.MyField>
```

Before and after each assignment to the given field, a logging message containing the current value of the field will be issued.

Conclusion

Based on all of the above, we do think AOP is one of most prospective software development paradigms and technologies, especially for .NET, due to its advanced generic architecture. In our opinion, .NET is currently the only platform that enables full-fledged AOP support.

The most prospective directions to continue the Aspect.NET research project, in our opinion, are as follows:

- implementing AOP support for other .NET languages, in particular, for VB.NET, JScript.NET, C#.NET, etc.;
- aspect-oriented knowledge management, or experiments with various forms of application domain knowledge specifications – primarily, with algebraic specifications – and experiments of expressing weaving rules in modern production languages like REFAL or Visual Prolog. ◉