White Paper Draft
The AOP Alliance: Why Did We Get In?

Renaud Pawlak

July 11, 2003

WARNING: This paper is a DRAFT! It is an attempt to make a first specification/guideline/roadmap(?) for the AOP Alliance. Many ideas here come from the discussion between the members of the ML, including Cedric Beust, Rod Johnson, Gregor Kiczales, Bob Lee, Rickard Oberg, Andrei Popovitchi, Jon Tirsen, myself, and others. This draft can be discussed, and I hope it will help to reach a common and quite precise point of view of what AOP Alliance is.

Contents

Introduction

The aim of this document is to present the AOP Alliance project. Its goals, its philosophy, what answers it should provide, and what it should not. It is a draft proposal which has to be further discussed with other members of the AOP Alliance in order to reach a common view on what we are doing here. It should also be completed as soon as an interesting point comes out from the discussions on the list.

This document is a white paper and can be used on internal purpose by the AOP Alliance members, but also to provide an insight and an understanding about what is AOP Alliance for external people.

In section 1, I will try to explain in general terms the goals of the AOP Alliance. Our motivations comes from the fact that AOP can improve solutions such as J2EE-based ones. If we manage to define a normalize set of APIs, it would be possible to integrate AOP in existing solutions or to build AOP environments using existing AOP tools. Section 2 gives an overview of a proposed architecture and APIs for aspect-oriented environments. I try to guess what APIs should be specified by the AOP Alliance. Finally, in section 3, I enter into the details of the identified components and their roles within AOP environments.

1 AOP Alliance goals

1.1 AOP Advantages: The J2EE Case

Aspect-Oriented Programming (AOP) is a great way for designing and programing the applications. It offers a better solution to many problems than do existing technologies such as EJB.

J2EE is a typical target environment (but not the only one) that could benefit from AOP Alliance. Indeed, J2EE environments partially solve some issues by providing means to handle technical issues such as persistence or transactions. However, the J2EE architecture is not flexible enough to easilly add new technical concerns related to particular needs. Moreover, it would be interesting to be able to remove one solution when not needed or when a ligther solution is preferable.

AOP provides a generic means to build new technical concerns (crosscutting concerns) and to plug them within an application in a flexible and modular way. Applying some AOP concepts in J2EE can also really simplify its use. For instance, regular Java objects (POJOs) can be used in place of EJBs. So, being able to easily apply full AOP to J2EE will greatly increase the usability of J2EE. It would also bring much more power to J2EE-compliant application servers.

1.2 The Current Brakes To AOP

AOP is gaining in popularity. However, most of the AOP tools were not designed in the purpose to be applied in any environment (mainly because most of the tools where designed on experimentation purpose). Thus, when trying to use AOP in a specific environment, we can face some issues because, for instance, the environment already supports some builtin aspects that may not be compliant with the AOP tool implementation.

This problem arises because AOP needs to modify the objects/classes of the application in order to work fine. This objet-modification logic is implemented by a specific part of the AOP tool: the weaver. A weaver can be well-fitted to a given environment but could break some important system properties in another one. For instance, an interesting discussion on the AOP Alliance's list beetween Gregor and Rickard seemed to show that AspectJ's weaver implementation for introductions (specific weaving operations) was not well fitted in some environments having some kinds of distribution and persistence cacabilities.

1.3 The AOP Alliance Claim

Most of the people here do not believe in the perfect system. We think that a system is always suited to a given problem and environment (it does not necesseraly fit the other one). This is exactly the case for the AOP tools that we may use within complex environments such as J2EE. Depending on the faced problem, it would be useful to have a specific implementation of AOP.

There are already a lot of specific implementations of AOP or AOP-related techniques such as generic proxies, interceptors, or bytecode translators. For instance, among others:

- AspectJ: an AO source-level (and bytecode-level) weaver. New Language.
- AspectWerkz: an AO framework (bytecode-level dynamic weaver+configuration in XML).
- BCEL: a bytecode translator.
- JAC: an AO middleware (bytecode-level dynamic weaver+configuration+aspects). Framework.
- Javassist: a bytecode translator with a high-level API.
- JBoss-AOP: interception and metadata-based AO framework (running on JBoss application sever + a standalone version).
- JMangler: a bytecode translator with a composition framework for translations.
- Nanning: an AO weaver (framework).
- Prose: an AO bytecode-level dynamic weaver (framework).

To us, these implementations reflect that there is no good or bad implementations, but implementations suited to some problems/environments.

So, the AOP Alliance goal is neither to come with a new AOP model, nor to provide a great AOP implementation that will work for all the cases or on a given J2EE application server. The AOP Alliance goal is rather to enable all the exisiting implementations to speak the same core language in order to:

- avoid re-building of existing AOP components by reusing them,
- simplify the adaptation of existing AOP components for a given target environment (typically a J2EE environment),
- simplify the aspects reusing by having a common root AOP API.
- simplify the implementation of development tools that whish to integrate AOP features.

2 Aspect-Oriented Architectures

2.1 A Common Architectural Vision

In sections 1.2 and 1.3, we explain that it is difficult to agree on a common AOP model and implementation because it is too tightly linked to the context of use and the environment (implementation may differ in a pure Java approach and in a J2EE-compliant approach). However, we think that it is possible to agree on a common architectural vision for Aspect Oriented Environnments (AOE).

Indeed, when building an Aspect-Oriented Environment (AOE)[1], designers need to define an architecture. In most of the existing AOEs, the architecture defines and combines some elementary modules/components/APIs that implement the basic functions of the system. By looking at the existing tools, we can identify common components (i.e. components that provides close functionalities in the considered architecture, but not necesseraly using the same implementation techniques). For instance, JBoss' weaver uses Javassist to implement an interception mechanism (which is instrumented at the client's side) whilst JAC's weaver uses BCEL to implement an interception mechanism (which is instrumented at the server's side). Other techniques like intercessing the JIT compiler can be employed to perform the same effect. All of them heavily rely on the environment.

In the next section, we will try to extract components that can be useful to AOEs. These components may be used to build contextual-dependent AOEs.

[1]With AOE, we mean any environment that supports AOP in one way or another. E.g. JBoss AOP is an AOE but any J2EE application server can be also regarded as a reduced AOE with buit-in aspects.

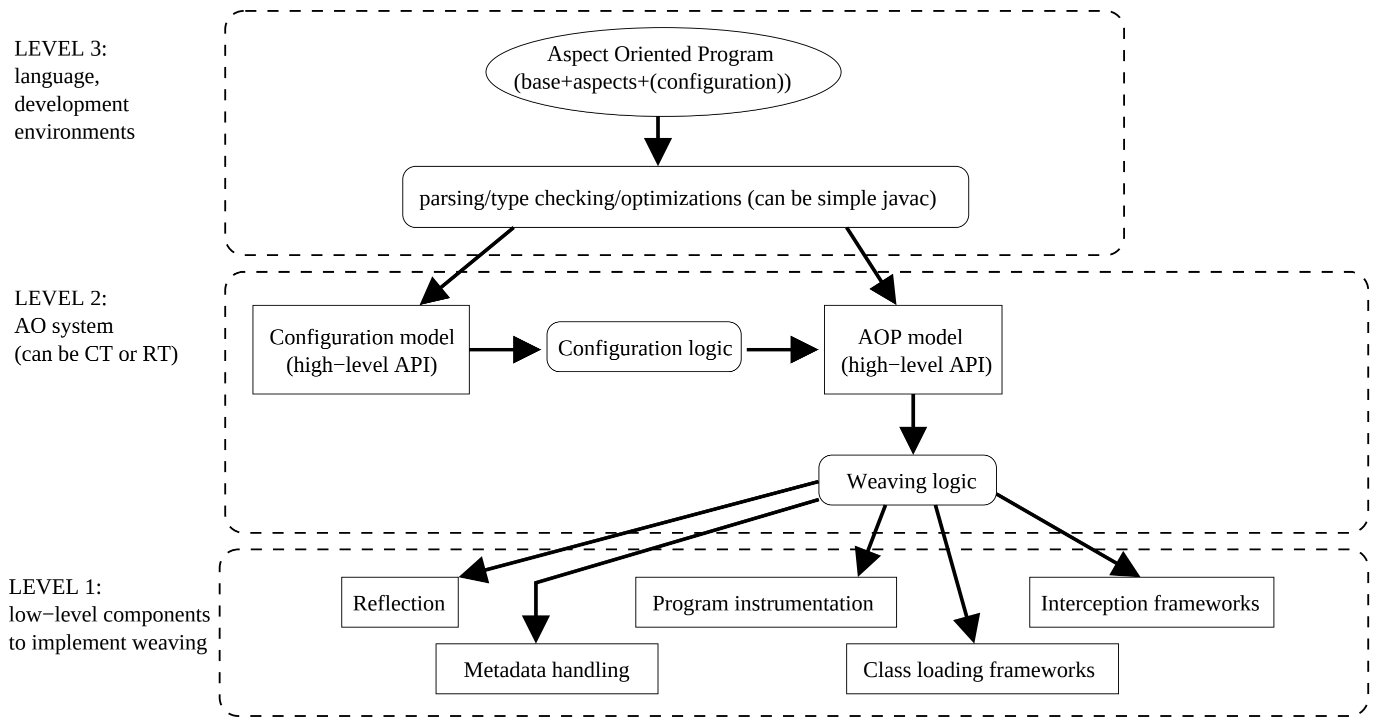
LEVEL 3: language, development environments
Aspect Oriented Program (base+aspects+(configuration))
parsing/type checking/optimizations (can be simple javac)
LEVEL 2: AO system (can be CT or RT)
Configuration model (high−level API)
Configuration logic
AOP model (high−level API)
Weaving logic
LEVEL 1: low−level components to implement weaving
Reflection
Program instrumentation
Interception frameworks
Metadata handling
Class loading frameworks

Figure 1: A three-layer architecture for aspect-oriented environments.

2.2 A 3-Layer Typical Architecture

A typical architecture could be drawn as shown in figure 1. This simplified diagram contains some components (boxes) and some core logics (rounded boxes) that can use (bold arrows) the components' APIs. It is meant to run an initial AO program on the top of the figure. Note that this architecture does not intend to be a reference architecture but only one possible architecture. Indeed, several possibilities exist for composing the different core components of an AO architecture.

One can split this architecture in three layers:

- a low-level layer (1) that provides basic components to implement the weaving (the main process of AOP) on the target platform,
- a high-level layer (2) that provides basic components for AOP, in its original meaning, plus the logic that implements the AO semantics (would depend on the target platform),
- a development-level layer (3) that includes the UI in its largest sense (can be supported by a language, can be a modelling tool) and other tools that are needed to help the developer trusting the AO programs (such as type-checking, visualization tools, debuggers, and so on).

2.3 What Shall AOP Alliance Specify Here?

As said before, the AOP Alliance goals are not to provide new models or better implementations of existing tools. In fact, the AOP Alliance goals are to specify normalized API for the components that are identified in common Aspect Oriented Environments (AOEs) implementations. If we manage to do this, it will be possible to build better AOEs than existing ones by integrating the components that best fit the context in which we want to use AOP. In particular, it should be possible to use the very best of AOP, even in complex environments such as J2EE application servers.

So, if we refer to figure 1, the AOP Alliance's role should be to define the API of the identified components. The most important components are the low-level ones because their implementations will influence the environment in which the AOE can be used. Some technical caracteristics may also have deep inpact on the resulting system properties (e.g. are the aspects can be dynamically woven/unwoven? is the system scalable regarding distribution? can the system cohabit with built-in aspects such as persistence or transactions?). However, the high-level components are also quite interesting for tools such as IDE, debuggers, modelling tools, and so on. Having a common AOP concepts manipulation API will help the tools to better support several AOP implementations in different environments.

The AOP Alliance could provide some reference implementations for some components (by using existing tools). However, it would be better if existing tools (most of the tools creators are in the Alliance) provide their own implementations of the defined APIs. These implementations will validate the API correctness.

The AOP Alliance will not tackle the weaving logic and the configuration logic since it really depends on the

AOE implementation. However, we should also provide some reference implementation in order to show how our API should be used to build AOEs.

Finally, the AOP Alliance will not address the third layer (development-level). We should let the development tools implementors use our API when the AOP tools they integrate implement it.

3 AOP Alliance Components

Let us now dive into the global picture of the core AOP Alliance components. **WARNING: These components are a first proposal draft on the APIs the AOP Alliance sould specify. Some of them may be removed, and some may be added. Note that some of them have already begun to be specified with Java interfaces.**

3.1 Low-Level Components

Low-level components are quite important ones because the entire AOE relies on them for its implementation. The way these components are implemented will be crutial and may drastically affect the system's properties such as performance, scalability, integration capabilities, or security.

3.1.1 Reflection

The reflection API is very important for any AOE. In fact, the weaver needs to introspects the classes of the base program in order to apply the advices or the introductions. For instance, if a pointcut tells that all the methods of a class should be adviced (using some kind of regular expression or an ALL keyword), then the weaver will need to use the reflection API to explicitely know the list of the methods that actually need to be adviced.

When the weaving process is done at runtime, the SUN's java.lang.reflect implementation can be sufficient to build the AOE. However, in most of the existing systems, the weaving process occurs at the compile-time or at class load-time. In these cases, a specific implementation of a reflection API is needed. According to AOP Alliance, it is quite important to normalize this API in order to be able to switch the underlying implementation depending on the running context of the AOE.

3.1.2 Program Instrumentation

From the weaver's point of view, if the reflection is the read access to the woven program, the instrumentation is the write access [2]. However, in AOP, the allowed program modifications are a reduced set of modifications. The allowed modications are incremental regarding the existing structure of the initial program so that the aspects can be correctly composed together. These kinds of incremental modifications are called instrumentations because of previous discussions on the list.

There is no standard API for instrumentation. However, like reflection, instrumentation can happen at run time, compile time, or load time. Moreover, for each category, different implementations can be performed depending on the context and the AOE's environment (for instance, the instrumentation can be done directly on the source code or on the bytecode). It is thus important to us that the instrumentation API is normalized in order to change the underlying implementation depending on the AOE requirements.

3.1.3 Interception Frameworks

Another type of base components that can be extremely useful to build AOEs are the interception frameworks. With the dynamic proxies, Java provides a standard API/framework for interception. However, several enhancements on transparency, performance, etc, can be achived by other implementations (most of them use an instrumentation API). It is thus also interesting to define a standard interception API/framework with a clear semantics.

Interception frameworks have many advantages since they allow to implement very easily the around advices of the AOP model. Moreover, they can be standalone and most of the time provide quite clear AOP-like code despite written in pure Java. For these reasons, several interception frameworks have been implemented in many projects and environments (including J2EE application servers, see JBoss). Hence, the AOP Alliance should provide an abstract interception framework in order to standardize this AOP important toolbox.

3.1.4 Metadata Handling

Metadata handling is useful when implementing AOEs, especially when coupled with an interception framework. It allows the weaver to extend the classes semantics in a non-invasive manner. Since most implementations on metadata allows dynamicity, it can also be used for dynamic configuration/reconfiguration of the aspects.

Even if the JDK1.5 will provide a standard implementation for metadata, it should be useful to provide a standard API that allows multiple implementations. These may take into account some environmental specificity such as distribution, serialization, that may not be correctly handled by the default implementation.

3.1.5 Class Loading Frameworks

In many AOEs, byte-code level manipulation is required. It can be used to implement an interception framework, or to directly implement the weaver's intrumentation of the programs. In some cases, this byte-code level manipulation can be done at class's load-time because the AOP instrumentations are quite simple. Thus, most of

[2] In reflection's foundations, this operation is called intercession.

the AOEs use the flexible class-loading architecture of Java.

However, several environments also use the class-loaders to implement their own functionalities. For instance, distributed environnments may generate the stubs using specific class loaders. Within these environments, the AOE's class-loading mechanism could lead to system crashes because of class-loaders incompatibilities.

Consequently, we think that it could be important to normalize a class-loading framework that would be flexible enough to easily enable different class loaders comming from different environment to cooperate in a safe way.

3.2 High-Level Components

The high-level components are important to normalize if we want the tools defined in the third layer (development layer) to provide better support for AOP in general.

3.2.1 AOP API

Explaining the goal of our AOP API can be explained great by quoting Gregor: "Clearly we want to do whatever we can to avoid needless inconsistency among AOP tools. It is much too soon to actually standardize, we still need room for meaningful variance. But needless variance is clearly worth eliminating."

So, our AOP model will no be a new model. It will just try to bring together what all the current models have in common. The AspectJ model is doubtless the most achieved one and there are already some tools that support it. So we will probably take a subset of AspectJ here.

3.2.2 Configuration API

Many aspects can be implemented in a generic fashion. This means that they implement a logic that is potentially reusable for any program in which you would want to weave the aspect functions. Most of the time, this aspect-reusing process implies a parameterization of the generic aspect (for instance, tell a generic persistence aspect which class should be persistent and how). In AspectJ, this can be done by subclassing abstract aspects. But it can also be done by using external tools (e.g. preprocessors). In J2EE environment, the configuration process of the built-in aspects (technical concerns of the EJB container) is parameterized by XML deployment files. In JBoss/AOP and other frameworks, the configuration can also be done using XML files. In JAC, the configuration can be done in Java programs with the aspect configuration API or by using a specific scripting-like language, and so on.

It would be great if we could normalize a configuration API. It would make AOEs integration easier for development tools. It would also facilitate the reusing of specific configurations from an AOE to another (e.g. AspectJ and JBoss/AOP).

Note that it is maybe unrealistic to make the aspects portable from an AOE to another because of the potentially important differences. But it seems less unrealistic to make the aspect configurations portable, which is already a first step towards AOEs interoperablity.

Conclusion

This paper tries to explain the reasons of the AOP Alliance project, and also to specify some goals. I tried to speak in the name of many people on the list, regarding what I have understood. It is quite unprecise for the moment and only draws a global picture. Maybe some of the people will disagree or will be disapointed. This would be great if we can have some really good discussion that we help us to reach a good feeling on what we really want here.