Aspect-Oriented Programming

Vladimir O. Safonov
St. Petersburg State University, Russia

INTRODUCTION

Aspect-oriented programming (AOP) is a prospective programming technology approach aimed at modularizing cross-cutting concerns – functionalities in software products that cannot be implemented by generalized procedures (like functions or classes) only but, on the contrary, whose implementations should consist of a number of code fragments (declarations or statements) scattered over the whole target program code. Examples of crosscutting concerns are: security, logging, and error handling. Adding such functionalities is a complicated and error-prone task since their implementation code penetrates the whole code of the target application, so such task should be modularized and automated, and those are the main goals of aspect-oriented programming. Currently aspect-oriented programming is at the stage of transition from research projects to wide industrial use. The objective of the article is to formulate main concepts of aspect-oriented programming, to analyze its current status, tools, advantages, issues, and perspectives.

BACKGROUND

A *concern* in software development is an idea, consideration, or design of some functionality to be implemented in the target software application. Software concerns can be subdivided into *common concerns* and *cross-cutting concerns*. So the task of software development can be regarded as separating and implementing its concerns. A *common concern* is a concern whose implementation can be made using *generalized procedures* (procedures, macros, classes, or other traditional modular programming features) only. However, there also exist *cross-cutting concerns* whose implementation, due to their nature, should consist of a set of scattered code fragments penetrating the code of the target application. Typical example of a cross-cutting concern is a set of security checks (for some permission) that should be done in each of or in most of the target application's software modules. If security checks are lacking in the initial version of the application they should be inserted, which is an error-prone task, uncomfortable and unsafe to do "by hand," especially if the target application is large. With traditional programming style, fragments of implementations of cross-cutting concerns are tangled within the target application code, so it may be difficult to separate them from each other.

Aspect-oriented programming is intended to solve the task of implementing and handling cross-cutting concerns in modular way. Each cross-cutting concern is implemented by a special, novel kind of module, referred to as *aspect* – an implementation of a cross-cutting concern. The concept of aspect extends the original concept of software module introduced by Myers in the 1970s.The definition of an aspect, in our terms (Safonov, 2008), contains *aspect actions* – the code fragments to be activated in some selected points of the target application. The target application is updated using an aspect or set of aspects by their *weaving* – injecting aspect actions into the desirable *join points* of the target application, or enabling to activate aspect actions in those join points some other way at runtime. The join points of the target application where to activate the appropriate aspect actions are filtered on the basis of *weaving rules* – the rules to locate join points in target applications. Weaving rules are parts of aspect definition. Each weaving rule is associated with some aspect action. Typical example of a weaving rule and its associated aspect action is as follows: Before call of each method of the target application, insert a code to issue a message of the kind: "Hello *M*" where *M* is the method name. In more traditional terms (Kiczales et al., 1997), a *pointcut* is a set of conditions to filter out the join points. A pointcut may be not related to any concrete aspect, and different aspects can refer to the same pointcut. *Advice* is an aspect action to be executed at a selected join point. Any part of aspect

DOI: 10.4018/978-1-4666-5888-2.ch693

definition specifying how to weave some advice can be regarded as a construct of the kind:

if *Condition* **then** *Advice*

where *Condition* is a condition to hold during execution of the target program for the *Advice* to be activated.

Join point model is an approach to implementation of the concepts of weaving and join points. Join point models can be different. Weaving can be implemented at the source code level, at intermediate code level, at binary object code level, at just-in-time compilation level, or at runtime (the latter approach is similar to the concept of a breakpoint in traditional debuggers).

ASPECT-ORIENTED PROGRAMMING: HISTORY, TOOLS, AND PERSPECTIVES

History

Ideas of separation of cross-cutting concerns in software originated long ago, since the 1970s, due to realizing the importance and fundamental nature of a cross-cutting concern. For example, the book (Fouxman, 1979) proposes a programming technology of scattered actions, and the concept of vertical cut, as a set of code fragment to implement some cross-cutting concern. The date of official origin of aspect-oriented programming is considered to be 1995 when the first and still the most widely spread aspect-oriented programming toolkit *AspectJ* (Kiczales et al., 1997) was developed for the Java platform by a team from Xerox PARC supervised by Kiczales, the father of aspect-oriented programming. AspectJ is an extension of the Java language by aspect-oriented programming features (aspect-oriented Java), and its implementation. AspectJ provides a number of kinds of join points (e.g., method call, method execution, and exception handler); constructs for pointcuts and advice; general construct of aspect definition; inter-type declarations to be inserted into the code of some other class for the purpose of using by some aspect. AspectJ has its own compiler of extended Java – *ajc*, and its own interactive development environments integrated to NetBeans and Eclipse. The influence of AspectJ to researchers and developers in the area of aspect-oriented programming is still deep and profound, and many other aspect-oriented tools have been developed based on similar ideas and architectures as AspectJ.

Approaches to Aspect-Oriented Programming and Aspect-Oriented Programming Tools

For the historical reasons stated above, most aspect-oriented programming tools are developed for the Java platform, starting from AspectJ. Typical approaches of implementing aspect-oriented programming for Java are as follows: extending the Java language by aspect-oriented constructs; using XML to configure aspects; using *interceptors* to execute aspect-oriented advice during method calls. Most of the Java AOP tools follow AspectJ concepts, syntax, and/or semantics.

The *AspectJ language* includes a variety of useful AOP features and is still used as a criterion of completeness for many other AOP framework and tools that appeared later. The main of them are *aspect definitions, named pointcut definitions* and *inter-type declarations*. Aspect definitions contain *advice declarations* which, in turn, can include *thisJoinPoint* – type constructs to handle reflective information about the join points in the target application.

Aspect definition in AspectJ is a new kind of modular units that encapsulates and can expand (weave into target application) some kind of cross-cutting behavior, like logging or security checks. An example of aspect definition in AspectJ:

```
aspect Logging { // Aspect that
                 Introduces log-
                 ging behavior
   pointcut AnyCall (): // Named
                          pointcut
       call (void MyClass.*(..));
   before(): AnyCall () {
      System.out.println("Hello"
         + thisJoinPoint);
   }
   after(): AnyCall () {
      System.out.println("Bye"
          + thisJoinPoint);
   }
} // Logging
```