Aspect-oriented programming with AspectJ

A. Colyer
A. Clement

Aspect-oriented programming (AOP) is an exciting new development in the field of software engineering. The open-source AspectJ® project has taken a leading role in defining what an aspect-oriented programming language should look like and in building tools that enable aspect-oriented techniques to be employed in the development of large-scale commercial software. IBM both contributes to the development of AspectJ and uses it internally with its accompanying IDE (integrated development environment) support, AspectJ Development Tools (AJDT). This paper provides an introduction to aspect-oriented programming using AspectJ and AJDT. We also discuss the role that open source (and being an open-source project) has played in the ongoing development of AspectJ, and how this has facilitated a level of collaboration and exploitation that would not have been possible otherwise.

The first paper to use the term *aspect-oriented programming* (AOP) was published in 1997 by a research group at the Palo Alto Research Center (PARC**).[1] Since that time, interest in aspect-oriented programming has steadily grown to the point that it now attracts large audiences at developer conferences, and a growing number of companies are using AOP to build production applications. In this paper we first introduce AOP and the benefits it brings and then look at the AOP language AspectJ**.[2] AspectJ is an open-source project initiated by PARC and now led by IBM. The AspectJ Development Tools (AJDT) project[3] is a related open-source project, also led by IBM, that provides IDE (integrated development environment) support for programming with AspectJ within the Eclipse IDE.[4] After introducing the language and tools, we discuss the adoption of AspectJ within IBM. The paper concludes by considering the role that open source has played both in the development of AspectJ and AJDT and in IBM's involvement in that process.

WHAT IS AOP?

AOP is a term used to describe a programming technique and a way of thinking about the construction of software applications that complements the forms of expression found in object-oriented programming. The goal of AOP is to improve the modularity of software applications, making them easier to develop, test, and maintain. Aspect-oriented programs comprise of a mixture of *objects* and *aspects*. Both encapsulate state and behavior,

but whereas the behavior in an object is executed only when explicit calls are made to the object's methods, the behavior in an aspect can also execute at points in the runtime of the program determined by the aspect's specification.

This simple idea turns out to be enormously powerful at modularizing the implementation of certain types of application features and functions. For example, given the requirement to issue a change notification to any registered listeners when the state of any one of a set of model objects changes, an object-based implementation requires the addition of a call to notifyListeners( ) (or some similar method) after each state-changing operation. An aspect-based implementation can simply code (in one module known as an aspect) the following, "After the state of any model object changes, call notifyListeners( )." As another example, consider the use of a persistence service that requires all updates to the persistent state of an application to occur within the scope of a session or transaction. An object-based implementation requires the insertion of logic to start a session before every interaction with the persistence service and to close it again afterwards—managing any exceptions that may be generated by the service along the way. An illustration of this example using the Hibernate[5] persistence framework for session and transaction management is shown in the following listing:

```
try {
  Session session = sessionFactory.openSession( );
  Transaction tx = null;
  try {
    tx = session.beginTransaction( );

    // perform updates to persistent state
    // here ...

    tx.commit( );
  } catch (HibernateException hEx) {
    if (tx != null) tx.rollback( );
    throw hEx;
  } finally {
    session.close( );
  }
} catch (HibernateException hEx) {
  // handle exception
}
```

An aspect-based implementation of the same requirement can be coded once in an encapsulated module, "Before updating the persistent state of the application start a session, and after the update has been completed, close it again." Although the application may involve many interactions with the persistence service, the aspect-based implementation will need to be written only once.

Both the change notification and session-management requirements are examples of what the AOP community has termed a *crosscutting concern*. A crosscutting concern is a single concern in the design or implementation of a system that impacts multiple places in the static structure of the system or in its runtime control flow. Aspects modularize crosscutting concerns, allowing the implementation of a single (crosscutting) concern in a single module, and therefore eliminating violations of the DRY (don't repeat yourself) principle.[6] Instead of the same fragment of code being repeated in many places throughout the application (e.g., the calls to notifyListeners( ), or the session-management logic), the code can be written once in an aspect. This makes the implementation easier to add, remove, and maintain. In the aspect-oriented community the term *scattering* is often used to describe the situation where multiple fragments of code that all do the same thing (or that do closely related things) are spread throughout a code base in a non-modular fashion.

A concept closely related to scattering is *tangling*. Tangling occurs when a module (typically a class in an object-oriented system) contains logic pertaining to more than one feature or function. The implementations of the different features are said to be tangled together in the module. As an example, consider the implementation of a BankAccount class that contains logic to place a message in a queue whenever a withdrawal over a certain threshold is made. The implementations of the BankAccount feature and a portion of a SupervisorAlerts feature have become tangled in a single class. By using an aspect to modularize the implementation of the SupervisorAlerts feature, this tangling can be eliminated.

Tangling is an indication of a less than ideal system modularity. In particular, tangling makes it harder to test, maintain, and reuse the tangled feature implementations. Karl Lieberherr recently formulated an update to the Law of Demeter designed to

eliminate tangling in a design: "Speak only to your friends that share your current concerns."[7]

By eliminating the effects of scattering and tangling that are prevalent in object-oriented systems, the goal of AOP can be stated simply: "Every module in the system should do one thing and one thing only."

The usual software engineering criteria for what makes a good module (coupling, cohesion, etc.) apply equally to AOP and to object-oriented programming.

Underpinning all AOP systems is something called a *join point* model. Join points are events that occur during the runtime execution of a program (for example, the initialization of a class, the execution of a method, the handling of an exception, or the updating of a field). The join point model determines which of these events are exposed to the aspect-oriented programmer. *Pointcuts* are predicates that match join points. For example, all AOP systems known to the authors will provide for a pointcut that matches the execution of a given method (or set of methods). Blocks of code known as *advice* are written to execute at any join point matched by a pointcut expression associated with the advice.

To implement the change notification requirement that we introduced at the start of this section, an aspect-oriented programmer would write a pointcut that matched join points representing the setting of a field value within any of the model objects. The programmer would then write a simple piece of advice associated with that pointcut to call the notifyListeners( ) method.

ASPECTJ AND AJDT

In the following section we provide a brief introduction to AspectJ and the AJDT and show how the ideas of aspect orientation are implemented in the AspectJ language and supported by AJDT.

AspectJ

The AspectJ language is an extension of the Java** language that supports AOP. The programs generated by the AspectJ compiler can run on any Java Virtual Machine (JVM**) and have no special runtime requirements, other than that the small AspectJ runtime library, aspectjrt.jar, be available somewhere on the classpath (a listing of locations where Java can find class files).

The declaration of an aspect in AspectJ looks very much like a class declaration, except that the keyword class is replaced by the keyword aspect:

```
public aspect SupervisorAlerts {
...
}
```

Aspects can have state (fields) and behavior encapsulated in methods in just the same way as a class:

```
public aspect SupervisorAlerts {
  private static final Money
    WITHDRAWAL_THRESHOLD = new Money(1000, 0);
  private QueueConnectionFactory
    connectionFactory;
  ...
  private void sendMessage(String messageText) {
    ...
  }
}
```

AspectJ's join point model includes join points for:

- A method or constructor call
- The execution of a method or constructor
- The accessing or updating of a field
- The handling of an exception
- The initialization of a class or object
- The execution of advice

Pointcuts in AspectJ are declared using the pointcut keyword. To issue a supervisor alert whenever a withdrawal is made over some threshold, a pointcut called withdrawal( ) that will match a join point representing the execution of a withdrawal method can be defined:

```
pointcut withdrawal( ) :
  execution(* withdrawal(Money));
```

AspectJ supports three basic kinds of advice: *before advice, after advice,* and *around advice. Before advice* runs before the execution of a matched join point, *after advice* runs after the execution of a matched join point, and *around advice* gives control over the actual execution of a matched join point. To implement the withdrawal alert, we choose to send a message to the supervisor after the successful completion of a large withdrawal:

```
after( ) returning : withdrawal( ) {
  // if the withdrawn amount was > threshold
  // then send supervisor alert
}
```

After-returning advice runs after a successful return from the execution of a matched join point. (Later we will see *after-throwing advice*, which runs when a join point is left via an exception.) To implement the body of the advice, we need more information—in particular, we need to know the BankAccount object in question and the amount of the withdrawal. Pointcuts can be used to provide contextual information at matched join points, and we extend the definition of the withdrawal( ) pointcut to do this:

```
pointcut withdrawal(BankAccount acc,
                    Money amount) :
  execution(* withdrawal(Money)) &&
  this(acc) &&
  args(amount);
```

The new definition of the withdrawal( ) pointcut matches any join point that represents the execution of a withdrawal method, taking one argument (the amount to withdraw). The this(acc) component of the pointcut specifies that the object executing the method is bound to the pointcut parameter acc and therefore must be an instance of type BankAccount. The args(amount) component of the pointcut specifies that the single argument to the method is bound to the pointcut parameter amount, and therefore must be of type Money.

Now that the needed contextual values are provided by the withdrawal( ) pointcut, they can be used in the advice declaration:

```
after(BankAccount account,
      Money amountOfWithdrawal) returning :
  withdrawal(account,amountOfWithdrawal) {
  if (amountOfWithdrawal.greaterThan(
                          WITHDRAWAL_THRESHOLD)) {
    sendMessage(
      "Large withdrawal from account: " +
      account + " : " + amountOfWithdrawal);
  }
}
```

Notice how an advice declaration can have parameters much like a method declaration. Instead of the parameter values being passed to the advice when it is called by a program statement (as happens for a method), the parameter values for advice are provided by the associated pointcut expression at each matched join point. The basic implementation of the SupervisorAlerts aspect is now complete:

```
public aspect SupervisorAlerts {
  private static final Money
    WITHDRAWAL_THRESHOLD=new Money(1000,0);
  private QueueConnectionFactory
    connectionFactory;
  ...
  pointcut withdrawal(BankAccount acc,
                      Money amount) :
    execution(* withdrawal(Money)) &&
    this(acc) &&
    args(amount);
  after(BankAccount account,
        Money amountOfWithdrawal) returning :
    withdrawal(account,amountOfWithdrawal) {
    if (amountOfWithdrawal.greaterThan(
                          WITHDRAWAL_THRESHOLD)) {
    sendMessage(
      "Large withdrawal from account: " +
      account + " : " + amountOfWithdrawal);
    }
  }
  private void sendMessage (String messageText) {
    ...
  }
}
```

This aspect could now be extended to encompass the other supervisor alerts that are required, keeping the whole alerting feature modularized and encapsulated. For example, given the requirement to alert a supervisor whenever an account operation fails with an InsufficientFundsException, or when an AuthorizationException is generated by any method in the banking package, then the aspect can be extended as follows:

```
pointcut accountOperation(BankAccount acct) :
  execution(* *(. .)) && this(acct);
after(BankAccount account)
  throwing(InsufficientFundsException ex) :
    accountOperation(account) {
    sendMessage(
      "Insufficient funds for transaction " +
      "on account"+ account +
      " : " + ex.getMessage( ));
}
```

```
pointcut bankingOperation( ) :
   execution(* *(. .)) && within(banking.*);
after( ) throwing(AuthorizationException ex):
   bankingOperation( ) {
      sendMessage ("Authorization failure: " +
                        ex.getMessage( ) );
}
```

If the logic to send supervisor alerts had not been encapsulated in the aspect in this way, there would be many places throughout the banking application where fragments of code concerned with implementing this feature would be found.

The AspectJ language includes many more features that can be used to improve the modularity of software applications, but a full treatment is beyond the scope of this paper. Interested readers are referred to the online AspectJ tutorial[8] or one of the many books on AspectJ, for example, References 9 and 10.

AJDT

AJDT provides IDE support for programming in the AspectJ language and is freely available from the Eclipse website.[3] Along with the usual syntax highlighting, building, and error-reporting elements, AJDT also provides a wealth of features that help users understand the effects of the aspects in their program. This part of the AJDT tool set provides *aspect-browsing* capabilities, similar to the class-browsing capabilities that are available for object-oriented programs.

Figure 1 shows a screenshot of AJDT in use. Both the BankAccount class and the SupervisorAlerts aspect are being edited, and the syntax highlighting that AJDT provides can be seen. Notice the markers in the gutter to the left of the BankAccount editor that indicate the presence of advice on a join point that the marked code will give rise to at runtime. In this case, when the withdrawal method is executed, it will give rise to an execution join point that is advised by the SupervisorAlerts aspect. The Outline View to the right of the Eclipse window shows an outline for the SupervisorAlerts aspect. In addition to indicating the members of the aspect (the fields, methods, advice, and pointcuts in this case), the Outline View shows the places that a piece of advice is in effect (the *advises* relationship). The links can be used for navigation to the advised locations. The Outline View for an advised type also provides *advised by* relationships that allow the developer to see and to navigate to any advice affecting the type.

AJDT also includes comprehensive help on using AJDT and AspectJ, a visualizer that provides an overview of an entire AspectJ project at a glance, integrated debugging support, wizards for creating aspects and AspectJ projects, full access to the AspectJ compiler options, and more.

ADOPTION WITHIN IBM

Based on our experiences within IBM, we recommend a staged approach when adopting AOP and AspectJ. At the first stage of adoption, aspects can be written that enforce design constraints and contracts. The central mechanisms used to do this are the AspectJ constructs declare warning and declare error. Like the advice forms in the previous section, these constructs are also associated with a pointcut. Instead of taking action during the runtime execution of the program, they signal the compiler to detect code that will give rise to matching join points during the compilation process and raise a warning or error at each match. The following statement can be read as "raise a compile-time warning if a call is made to JDBC**(Java Database Connector) outside of the persistence layer":

```
declare warning :
   call(* java.sql. .*(. .)) &&
   !within(org.xyz.persistence. .*)
      : "Only the persistence layer should " +
        "be calling JDBC.";
```

In addition to enforcing such design constraints, aspects can be written at this stage of adoption that help in testing and debugging, for example, and that check pre- and post-conditions on methods. Within IBM we wrote a simple aspect that issued a warning on violations of API (application programming interface) contracts within the WebSphere* platform (similar in style to the example just shown). The aspect and the simple scripts that enable a product team not skilled in AspectJ to pick up and easily use the language have been widely used by product groups inside IBM—over 20 product teams have incorporated this approach into their testing to date, and many thousands of problems have been detected and eliminated.

At the second stage of adoption, we recommend the use of aspects for implementing non-core function.

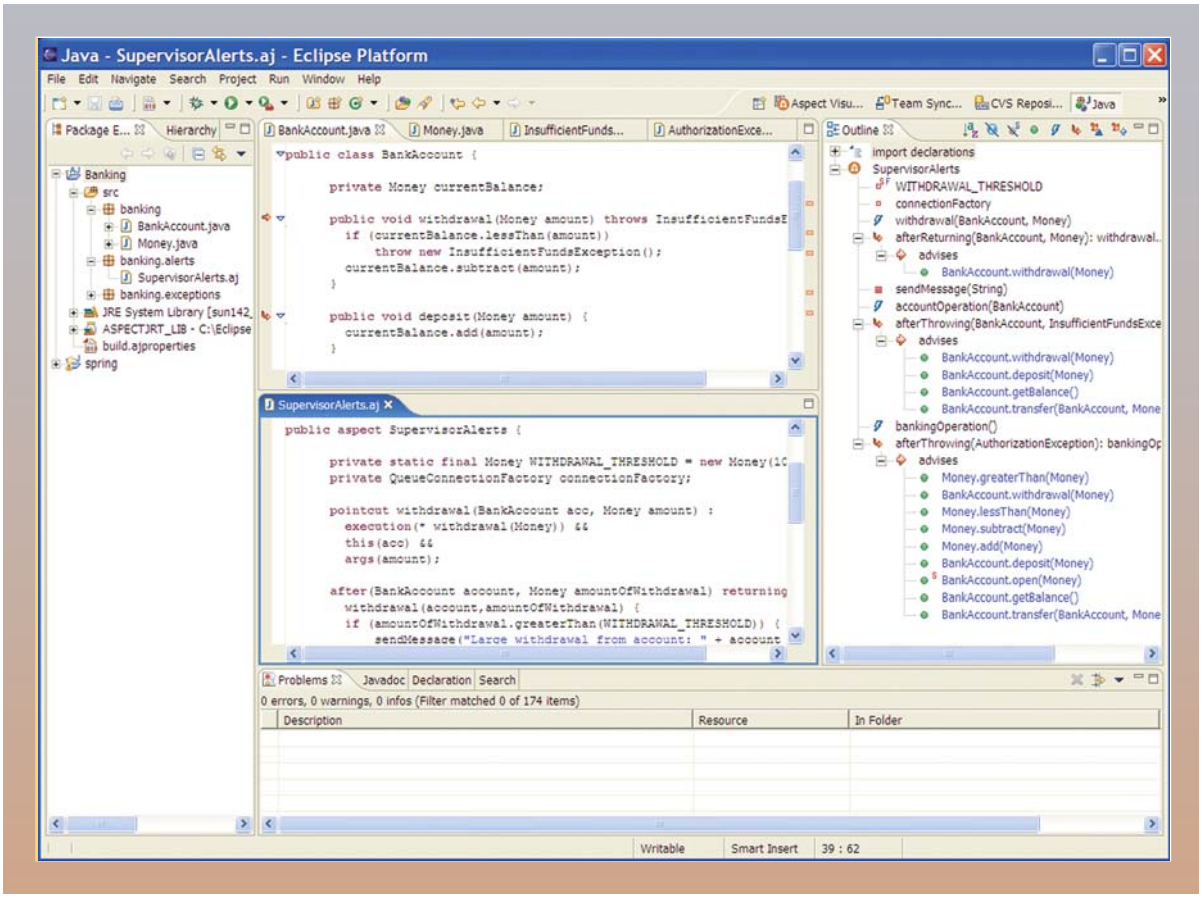

Figure 1
AJDT screenshot

Sometimes we call these *auxiliary* aspects. Within IBM we have modeled several of the WebSphere platform policies (for example, the WebSphere policy for first-failure data capture when an error or exception is detected) as aspects.[11,12] These approaches are being adopted by product groups instead of hand-coding the implementation of the policies throughout large code bases.

At the third stage of adoption, aspects are also used to implement core pieces of a software program's functionality. A number of projects within IBM have reached this stage of adoption. One emerging theme is the use of aspects to aid in the incremental addition of function around a small microkernel. We also undertook a large-scale study to assess the use of aspects in refactoring existing software in order to improve modularity and to create new reusable components. The results of this successful study are documented in Reference 13.

THE IMPORTANCE OF BEING OPEN SOURCE

The open-source nature of the AspectJ and AJDT projects played a critical part in their adoption and success, and in IBM's involvement.

The early adopters who initially acquired AspectJ and started experimenting with it were able to do so without any inhibitors being placed in their way. The three key features of the AspectJ project (which are shared with many open-source projects) that enabled this early adoption are:

1. *The AspectJ binaries are freely available for download.* If there had been a license fee to pay, it is unlikely that many of the early adopters would have tried out AspectJ, especially because these individuals typically worked at companies and research establishments and were often acting out of their own curiosity.

2. *The AspectJ source code is freely available*. This became of central importance to those early adopters who had tried out AspectJ, found it to be beneficial, and wanted to use it in the development of their applications. We remember a conversation with one early adopter at a commercial organization who was asked "Are you concerned about the lack of commercial support for AspectJ?" He replied that his company had as much confidence, if not more, in using AspectJ as in many commercial tools they were already using because of the direct access to developers on the AspectJ mailing lists and the ability to download the source code and fix problems themselves if necessary.
3. *There is ready access to AspectJ's developers and users.* Like most good open-source projects, AspectJ maintains a user mailing list through which support from the AspectJ community and from the AspectJ developers themselves is readily available. (At the time of writing, there are almost 1,000 subscribers to the AspectJ mailing list.) By being responsive to questions posed on the mailing list and to bug reports submitted, the AspectJ team has won the confidence of their user community to continue using AspectJ, even while AspectJ itself has undergone significant development.

The AspectJ project continually exploited the close feedback that an open-source project can provide in order to develop both the core AspectJ language and the supporting tools. This feedback helped to make the technology easier to use and better able to address user requirements. Planned extensions and changes to the language were discussed openly on the mailing list, and user feedback helped to drive many enhancements, such as improved error messages, faster compilation times, the inclusion of Ant[14] tasks for building AspectJ programs, support for incremental compilation, program browsing tools, and more.

Recall that AspectJ began life as an open-source project led by a team from PARC. IBM interest in the project began simply when an IBM employee downloaded and experimented with a release of AspectJ in much the same way that any user might proceed. Exchanges on the AspectJ mailing lists led to the establishment of an initial relationship. When IBM first announced another open-source project, Eclipse, in late 2001, an IBM group at Hursley proposed to the AspectJ development team the idea of a joint open-source project under the eclipse.org[15] umbrella to provide IDE support for AspectJ inside Eclipse. This project became known as AJDT and was first made public in May of 2002. The joint development that occurred during work on the project quickly built up a strong relationship between the IBM and PARC teams, as well as a growing understanding of the AspectJ code base within IBM.

At the time that the AJDT project was beginning, several members of the AspectJ development team from PARC visited IBM Hursley to undertake a proof-of-concept demonstration applying AspectJ in the context of IBM product development. This visit led to several suggestions for enhancements to the AspectJ compiler, which in turn culminated in a substantial change in the implementation of AspectJ in the Version 1.1 release. The results of that proof-of-concept demonstration and a description of the changes in AspectJ's implementation that it helped to trigger have been published in References 11 and 12.

By late 2002, it had become clear to the management at PARC that AspectJ was maturing beyond the point of being a research project, and the decision was made to move the project from its home at PARC. An agreement was made to transfer the AspectJ project to eclipse.org under the Common Public License (CPL).[16] The existing Eclipse-based collaboration between IBM and PARC was a contributing factor in the choice of eclipse.org as a home for the AspectJ project. When AspectJ became an Eclipse Technology Project in December of 2002, the first external programmer to join the existing developers from PARC on the AspectJ project was an IBM employee. Once again the collaborative development made possible through the open-source nature of the project enabled the further development of AspectJ expertise within IBM. When the AspectJ project leader stepped down from the role in 2003, the newly elected leader was an IBM employee. The open-source nature of three projects, Eclipse, AJDT, and AspectJ, had played a crucial role in building trust and expertise across organizational boundaries, ultimately allowing this transfer of leadership to occur. Today AspectJ remains an open-source project on eclipse.org, with contributors from several different organizations.

CONCLUSION

AOP is an exciting new development in the field of software engineering with the goal of improving the modularity of software applications and making them easier to develop, test, and maintain. The open-source AspectJ project has taken a leading role in defining what an AOP language should look like and in building tools that enable aspect-oriented techniques to be employed in the development of large-scale commercial software. IBM not only contributes to the development of AspectJ and its accompanying IDE support, AJDT, but also uses them internally. The open-source nature of AspectJ and AJDT projects has played and continues to play a critical role in the evolution, adoption, and ongoing development of these projects.

*Trademark or registered trademark of International Business Machines Corporation.

**Trademark or registered trademark of the Palo Alto Research Center, Inc. or Sun Microsystems, Inc.

CITED REFERENCES

1. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Videira Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-Oriented Programming," *Proceedings of the 11th European Conference on Object-Oriented Computing (ECOOP'97)*, Jyväskylä, Finland, June 9–13, 1997, *Lecture Notes on Computer Science* **1241**, Springer-Verlag, New York (1997), pp. 220–242.
2. AspectJ Project, The Eclipse Foundation, http://www.eclipse.org/aspectj/.
3. AspectJ Development Tools Subproject, The Eclipse Foundation, http://www.eclipse.org/ajdt/.
4. *Eclipse Platform Technical Overview*, The Eclipse Foundation, http://www.eclipse.org/whitepapers/eclipse-overview.pdf.
5. Hibernate—Relational Persistence for Idiomatic Java, http://www.hibernate.org/.
6. A. Hunt and D. Thomas, *The Pragmatic Programmer: From Journeyman to Master*, Addison-Wesley Professional, Boston, MA (1999).
7. K. J. Lieberherr, "Controlling the Complexity of Software Designs," *Proceedings of the 26th International Conference on Software Engineering (ICSE 2004), Edinburgh*, Scotland, UK, May 23–28, 2004, ACM, New York (2004), pp. 2–11.
8. The AspectJ Team, *The AspectJ Programming Guide*, The Eclipse Foundation, http://www.eclipse.org/aspectj/.
9. R. Laddad, *AspectJ in Action*, Manning Publications Co., Greenwich, CT (2003).
10. A. Colyer, A. Clement, G. Harley, and M. Webster, *Eclipse AspectJ: Aspect-Oriented Programming with AspectJ and the Eclipse AspectJ Development Tools*, Addison-Wesley Professional, Boston, MA (2004).
11. R. Bodkin, A. Colyer, and J. Hugunin, "Applying AOP for Middleware Platform Independence," *Practitioner Report, 2nd International Conference on Aspect-Oriented Software Development (AOSD 2003)*, Northeastern University, Boston, MA, March 17–23, 2003, http://aosd.net/archive/2003/program/bodkin.pdf.
12. A. Colyer, A. Clement, R. Bodkin, and J. Hugunin, "Using AspectJ for Component Integration in Middleware," *Proceedings of the 18th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2003)*, Anaheim, CA, October 26–30, 2003, ACM, New York (2003), pp. 339–344.
13. A. Colyer and A. Clement, "Large-Scale AOSD for Middleware," *Proceedings of the 3rd International Conference on Aspect-Oriented Software Development (AOSD 2004)*, Lancaster, UK, March 22–26, 2004, ACM, New York (2004), pp. 56–65.
14. The Apache™ Ant Project, The Apache Software Foundation, http://ant.apache.org/.
15. Eclipse.org is the Web site of the Eclipse Foundation, http://www.eclipse.org/.
16. *Common Public License—v1.0*, The Eclipse Foundation, http://www.eclipse.org/legal/cpl-v10.html.

Accepted for publication September 29, 2004.
Published online April 7, 2005.

Adrian Colyer
MP 146, IBM Hursley Park, Winchester, England SO21 2JN (adrian_colyer@uk.ibm.com). Adrian Colyer is an IBM Senior Technical Staff Member based in Hursley, England. He leads the open-source AspectJ and AJDT projects on eclipse.org and is a frequent writer and speaker on AspectJ and aspect-oriented programming (AOP). He is a co-author of the book *Eclipse AspectJ: Aspect-Oriented Programming with AspectJ and the Eclipse AspectJ Development Tools.* In 2004 he was named as one of the top 100 young innovators by the *MIT Technology Review* magazine. Before working on aspect-oriented technologies, Adrian worked on distributed systems middleware for nearly a decade.

Andy Clement
MP 146, IBM Hursley Park, Winchester, England SO21 2JN (clemas@uk.ibm.com). Andy Clement is a senior software engineer at IBM Hursley Park. With a background in transaction processing and enterprise systems development, he is currently involved in the use of aspects in J2EE™ middleware and has given tutorials on best practices for using aspect-oriented programming (AOP) techniques. He is one of the founders of the AspectJ Development Tools (AJDT) for the Eclipse project and is an active participant in the AspectJ project. He is a co-author of the book *Eclipse AspectJ: Aspect-Oriented Programming with AspectJ and the Eclipse AspectJ Development Tools.* ■