Chapter 13
Aspect Oriented Architectures

Yan Liu

13.1 Aspects for ICDE Development

The ICDE 2.0 environment needs to meet certain performance requirements for API data retrievals. To try and guarantee this performance level, the actual behavior of an ICDE implementation needs to be monitored. Performance monitoring allows remedial actions to be taken by the development team if the required performance level is not met.

However, ICDE v2.0 is a large, multithreaded and distributed system, comprising both off-the-shelf and custom written components. Such systems are notoriously difficult to monitor and isolate the root cause of performance problems, especially when running in production environments.

The time-honored strategy for monitoring application performance and pinpointing components causing performance bottlenecks is to instrument the application code with calls to log timing and resource usage. However this approach leads to duplicate code being inserted in various places in the source. As always, duplicate code creates code bloat, is error prone and makes it more difficult to maintain the application as the ICDE application evolves.

The ICDE team was aware of the engineering problems of inserting performance monitoring code throughout the ICDE code base. Therefore they sought a solution that could separate the performance monitoring code from the application implementation in a modular, more maintainable way. Even better would be if it were possible to inject the performance monitoring code into the application without the need to recompile the source code.

So, the ICDE team started to look at aspect-based approaches and technologies to address their performance monitoring problem. Aspect-oriented programming (AOP) structures code in modules known as aspects. Aspects are then merged at either compile time or run time to form a complete application.

The remainder of this chapter provides an overview of AOP, its essential elements and tool support. It also discusses the influence of aspect-based approaches on architecture and design. Finally, the chapter describes how the ICDE system could leverage aspect-based techniques to monitor application performance in a highly flexible, modular and maintainable way.

I. Gorton, *Essential Software Architecture*,
DOI 10.1007/978-3-642-19176-3_13,

13.2 Introduction to Aspect-Oriented Programming

Aspect-oriented programming (AOP) is an approach to software design invented at Xerox PARC in the 1990s.[1] The goal of AOP is to let designers and developers better separate the "crosscutting concerns" that a software system must address. Crosscutting concerns are elements of a system's behavior that cannot be easily localized to specific components in an application's architecture. Common crosscutting concerns are error handling, security checks, event logging and transaction handling. Each component in the application must typically include specific code for each crosscutting concern, making the component code more complex and harder to change.

To address crosscutting concerns, AOP provides mechanisms for systematic identification, separation, representation and composition. Crosscutting concerns are encapsulated in separate modules, called "aspects", so that localization can be achieved.

AOP has a number of potential benefits. First, being able to identify and explicitly represent crosscutting concerns helps architects consider crosscutting behavior in terms of aspects at an early stage of the project lifecycle. Second it allows developers to easily reuse the code for an aspect in many components, and thus reduces the effort of utilizing (often this means copying) the code. Third, AOP promotes better modularity and encapsulation as component code is succinct and uncluttered.

Structuring applications with aspects and directly implementing the design using aspect-oriented programming languages has the potential for improving the quality of software systems. Aspects can make it possible for large and complex software systems to be factored and recomposed into simpler and higher quality offerings. To see how this works, let's look at this approach in more details.

13.2.1 Crosscutting Concerns

Separation of concerns is a fundamental principle of software engineering. This principle helps manage the complexity of software development by identifying, encapsulating and manipulating those parts of the software relevant to a particular concern. A "concern" is a specific requirement or consideration that must be addressed in order to satisfy the overall system goal.

[1] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Videira Lopes, C., Loing tier, J.-M., and Irwin, J, *Aspect-Oriented Programming*, Proceedings European Conference on Object-Oriented Programming, Vol. 1241. Springer-Verlag, (1997) 220–242.

Any application is composed of multiple functional and nonfunctional concerns. Functional concerns are relevant to the actual use of the application, whereas nonfunctional concerns pertain to the overall quality attributes of the system, such as the performance, transactions and security. Even applications that are designed in a highly modular fashion suffer from tangling of functional and nonfunctional aspects. For example, cache logic to improve database performance might be embedded in the business logic of many different components, thus mixing or tangling functional and performance concerns. Other examples of crosscutting concerns include performance monitoring, transaction control, service authorization, error handling, logging and debugging. The handling of these concerns spans across multiple application modules, replicating code and making the application more complex.

13.2.2 Managing Concerns with Aspects

Using conventional design techniques, a crosscutting concern can be modularized using an interface to encapsulate the implementation of the concern from its invoking client components. Although the interface reduces the coupling between the clients and the implementation of the concern, the clients still need to embed code to call the interface methods from within its business logic. This pollutes the business logic.

With aspect-oriented design and programming, each crosscutting concern is implemented separately in a component known as an aspect. In Fig. 13.1, the difference between implementing a logging concern using conventional programming and AOP is demonstrated. The aspect defines execution points in client components that require the implementation of the crosscutting concern. For each

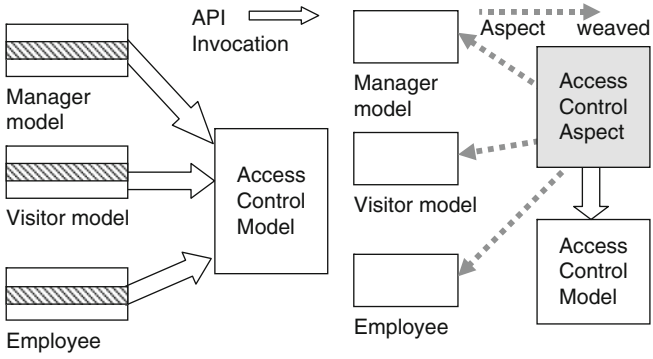

Fig. 13.1 Implementation of a logging concern

execution point, the aspect then defines the behavior necessary to implement the aspect behavior, such as calling a logging API.

Importantly, the client modules no longer contain any code to invoke the aspect implementation. This leads to client components that are unpolluted by calls to implement one or more concerns.

Once defined, the use of an aspect is specified in composition rules. These composition rules are input to a programming utility known as a "weaver." A weaver transforms the application code, composing the aspect with its invoking clients. Aspect-oriented programming languages such as AspectJ provide weaving tools, and hence AOP languages and tools are necessary to effectively implement aspect-oriented designs.

13.2.3 AOP Syntax and Programming Model

"Crosscutting" is an AOP technique to enable identification of concerns and structuring them into modules in a way that they can be invoked at different points throughout an application. There are two varieties of crosscutting, namely static and dynamic. Dynamic crosscutting modifies the execution behavior of an object by weaving in new behavior at specific points of interest. Static crosscutting alters the static structure of a component by injecting additional methods and/or attributes at compile time. The basic language constructs and syntax used to define crosscutting in AOP are:

- A "join point" is an identifiable point of execution in an application, such as a call to a method or an assignment to a variable. Join points are important, as they are where aspect behaviors are woven into the application.
- A "pointcut" identifies a join point in the program at which a crosscutting concern needs to be applied. For example, the following defines a pointcut when the *setValue* method of the *Stock* class is called:

```
pointcut log(String msg):args(msg)
execution(void Stock.setValue(float))
```

- An "advice" is a piece of code implementing the logic of a crosscutting concern. It is executed when a specified pointcut is reached.
- An "introduction" is a crosscutting instruction that can make static changes to the application components. An introduction may, for example, add a method to a class in the application.
- An aspect in AOP is equivalent to a class in object-oriented programming. It encapsulates pointcuts and associated advice and introductions.

In Fig. 13.2 the relationship between these AOP terms is illustrated.

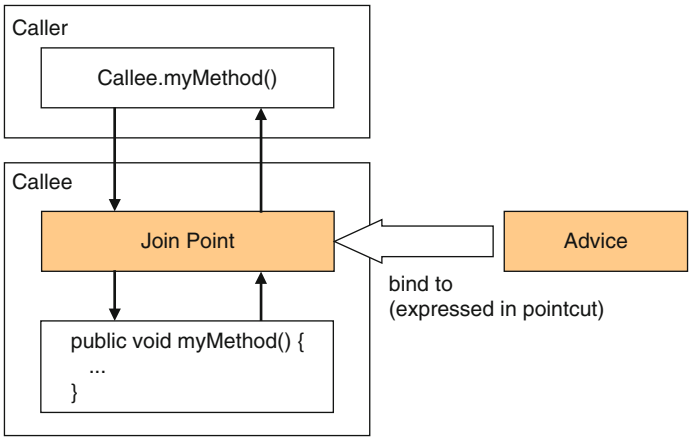

Fig. 13.2 The anatomy of AOP

13.2.4 Weaving

Realizing an aspect-oriented design requires programming language support to implement individual aspects. The language also defines the rules for weaving an aspect's implementation with the rest of the application code. Weaving can follow a number of strategies, namely:

1. A special source code preprocessor executed during compilation
2. A postprocessor that patches binary files
3. An AOP-aware compiler that generates woven binary files
4. Load-time weaving (LTW); for example, in the case of Java, weaving the relevant advice by loading each advice class into the JVM
5. Run-time weaving (RTW); intercepting each join point at runtime and executing all relevant advices. This is also referred to as "hotswapping" after the class is loaded

Most AOP languages support compile-time weaving (CTW) using one of the first three options. In the case of Java, the way it typically works is that the compiler generates standard Java binary class files, which any standard JVM can execute. Then the *.class* files are modified based on the aspects that have been defined. CTW isn't always the best choice though, and sometimes it's simply not feasible (e.g., with Java Server Pages).

LTW offers a better solution with greater flexibility. In the case of Java, LTW requires the JVM classloader to be able to transform or instrument classes at runtime. The JDK[2] v5.0 supports this feature through a simple standard mechanism. LTW must process Java bytecode at runtime and create data structures (this can

[2]Java Development Kit.

be slow) that represent the bytecode of a particular class. Once all the classes are loaded, LTW has no effect on the speed of the application execution. AspectJ,[3] JBoss AOP[4] and AspectWerkz[5] now support LWT.

RTW is a good choice if aspects must be enabled at runtime. However, like LTW, RTW can have drawbacks in terms of performance at runtime while the aspects are being weaved in.

13.3 Example of a Cache Aspect

In this section we'll use a simple example to illustrate the AOP programming model.[6] This simple application calculates the square of a given integer. In order to improve performance, if a particular input value has been encountered before, its square value is retrieved from a cache. The cache is a crosscutting concern, not an essential part of computing the square of an integer.

The example is implemented using AspectJ and shown in Fig. 13.3. The cache is implemented as an aspect in *Cache.aj* and separated from the core application implementation, *Application.java*. The method *calculateSquare* is a join point and it is identified by the pointcut *calculate* in the *Cache* aspect, as in the following:

```
pointcut calculate(int i):args(i)
  &&(execution(int Application.calculateSqure(int)));
```

The implementation of the cache function, retrieving a value from a *java.util.Hashtable*, is provided inside the *around* advice. Note that this advice is only applied to the class *Application*. The cache aspect is weaved into the application code at compile time using an AspectJ compiler.

The following output from executing the program demonstrates the advice is invoked at the join point.

```
Cache aspect is invoked for parameter 45
The square of 45 is 2025
Cache aspect is invoked for parameter 64
The square of 64 is 4096
Cache aspect is invoked for parameter 45
The square of 45 is 2025
Cache aspect is invoked for parameter 64
The square of 64 is 4096
```

[3] http://www.eclipse.org/aspectj/

[4] http://www.jboss.org/products/aop

[5] http://www.aspectwerkz.codehaus.org/

[6] Chapman, M., Hawkins, H. *Aspect-oriented Java applications with Eclipse and AJDT*, IBM developerWorks, http://www-128.ibm.com/developerworks/library/j-ajdt/

```
//Source code of Application.java
package Caching;

public class Application {
  public static void main(String[] args) {
    System.out.println("The square of 45 is " + calculateSquare(45));
    System.out.println("The square of 64 is " + calculateSquare(64));
    System.out.println("The square of 45 is " + calculateSquare(45));
    System.out.println("The square of 64 is " + calculateSquare(64));
  }
  private static int calculateSquare(int number) {
    try {
      Thread.sleep(6000);
    }
    catch (InterruptedException ie) {}
    return number * number;
  }
}


//Source code of Cache.aj
package Caching;
import java.util.Hashtable;

public aspect Cache {
   private Hashtable valueCache;
   pointcut calculate(int i) : args(i)
        && (execution(int Application.calculateSquare(int)));
   int around(int i) : calculate(i) {
      System.out.println("Cache aspect is invoked for parameter "+i);
      if (valueCache.containsKey(new Integer(i))) {
        return ((Integer) valueCache.get(new Integer(i))).intValue();
      }
      int square = proceed(i);
      valueCache.put(new Integer(i), new Integer(square));
      return square;
   }
   public Cache() {
      valueCache = new Hashtable();
   }
}
```

Fig. 13.3 A cache aspect implemented using AspectJ

13.4 Aspect-Oriented Architectures

An aspect relating to a system's quality attributes heavily influences the application architecture, and many such aspects are basically impossible to localize. For example, to guarantee the performance of a loosely coupled application, consideration must be paid to the behavior of individual components and their interactions with one another. Therefore, concerns such as performance tend to crosscut the system's architecture at the design level, and they cannot be simply captured in a single module.

AOP provides a solution for developing systems by separating crosscutting concerns into modules and loosely coupling these concerns to functional requirements. In addition, design disciplines like aspect-oriented design (AOD) and aspect-oriented software development (AOSD) have been proposed to extend the concepts of AOP to earlier stages in the software lifecycle. With AOD and AOSD, the separation of concerns is addressed at two different levels.

First at the design level, there must be a clear identification and definition of the structure of components, aspects, joint points and their relationship. Aspect design and modeling are the primary design activities at this level. Individual concerns tend to be related to multiple architectural artifacts.

For example, a concern for performance may be associated with a set of use cases in the architecture requirements, a number of components in the design and some algorithms for efficiently implementing specific logical components. The requirements for each aspect need to be extracted from the original problem statement, and the architecture needs to incorporate those aspects and identify their relationship with other components. It is also important to identify potential conflicts that arise when aspects and components are combined at this level. To be effective, this approach requires both design methodologies and tool support for modeling aspects.

Second, at the implementation level, these architectural aspects need to be mapped to an aspect implementation and weaved into the implementation of other components. This requires not only the expressiveness of an AOP language that can provide semantics to implement join points, but also a weaving tool that can interpret the weaving rules and combine the implementations of aspects.

13.5 Architectural Aspects and Middleware

As explained in Chap. 4, component-based middleware technologies such as JEE provide services that support, for example, distributed transaction processing, security, directory services, integration services, database connection pooling, and so on. The various issues handled by these services are also the primary nonfunctional concerns targeted by AOSD. In this case, both component technology and AOP address the same issue of separation of concerns.

Not surprisingly then, middleware is one of the most important domains for applying AOP. Research on aspect mining[7] shows that 50% of the classes in three CORBA ORB implementations are responsible for coordination with a particular aspect. AOP has been used in such cases to effectively refactor a CORBA ORB and modularize its functionality.

Following on from such endeavors, attempts have been made to introduce AOP to encapsulate middleware services in order to build highly configurable middleware architectures. Distribution, persistence and transaction aspects for software components using AspectJ have been successfully implemented, and AspectJEE extends AspectJ to implement the EJB model and several JEE services. In the open source product world, JBoss AOP provides a comprehensive aspect library for developing Java-based application using AOP techniques.

[7]Zhang, C., Jacobsen, H. *Refactoring middleware with aspects*. In IEEE Transactions on Parallel and Distributed Systems, IEEE Computer Society, (2003), 14(11):1058 – 1073.

The major problem in applying AOP to build middleware frameworks is that middleware services are not generally orthogonal. Attaching a service (aspect) to a component without understanding its interaction with other services is not sensible, as the effects of the services can interact with each other.

For example, aspects are commonly used for weaving transactional behavior with application code. Database transactions can be committed using either one phase or two phase (for distributed transactions) commit protocols. For any individual transaction, only one protocol is executed, and hence only one aspect, and definitely not both, should be weaved for any join point. In general, handling interacting aspects is a difficult problem. Either a compile-time error or a runtime exception should be raised if the two interacting aspects share a join point.

13.6 State-of-the-Art

Recent research and development efforts have been dedicated to various aspect-oriented technologies and practices. These include AOP language specification, tool support for aspect modeling and code generation, and integration with emerging technologies such as metadata based programming. Let's discuss each of these.

13.6.1 Aspect Oriented Modeling in UML

Several approaches exist to support aspect modeling for AOD and AOSD. Most of these approaches extend UML by defining a new UML profile for AOSD. This enables UML extensions with aspect concepts to be integrated into existing CASE tools that support standard UML.

An advantage of aspect oriented modeling is the potential to generate code for aspects from design models. In aspect oriented modeling and code generation, aspect code and nonaspect code is generated separately. Using Model Driven Architecture (MDA) approaches, tools use a transformation definition to transform a platform independent model (PIM) into one or more platform specific models (PSMs), from which the automated generation of aspect code and weaving can take place. MDA technologies are explained in detail in the next chapter.

13.6.2 AOP Tools

The fundamental model of AOP is the join point model. All AOP tools employ this model to provide a means of identifying where crosscutting concerns are applied.

However, different tools implement the specifics of the aspect model in their own way, and introduce new semantics and mechanisms for weaving aspects.

For example, in JBoss AOP, advices are implemented through "interceptors" using Java reflection, and pointcuts are defined in an XML file that describes the place to weave in an advice dynamically at run time. In AspectJ, both advices and pointcuts are defined in an aspect class and weaved statically.

This diversity in AOP tools is problematic for software development using aspects, because of the semantic differences of different AOP models and the different ways an aspect is weaved with other classes. It is not possible to simply redevelop an existing aspect in order for it to be weaved with other aspects developed with another AOP model.

In order to address this problem, AspectWerkz[8] utilizes bytecode modification to weave Java classes at project build-time, class load time or runtime. It hooks in using standardized JVM level APIs, and has a powerful join point model. Aspects, advices and introductions are written in plain Java and target classes can be regular POJOs. Aspects can be defined using either Java 5 annotations, Java 1.3/1.4 custom doclets or a simple XML definition file. (In true aspect-oriented style, AspectWerkz was weaved into the AspectJ v5.0 release in 2006).

13.6.3 Annotations and AOP

The join point model can utilize the properties of program elements such as method signatures to capture join points. However it cannot capture join points needed to implement certain crosscutting concerns, such as transaction and role-based security, as there is no information in an element's name or signature to suggest the need for transactional or authorization related behaviors. Adding metadata to AOP systems is therefore necessary to provide a solution for such cases.

In the programming language context, metadata known as "annotations," capture additional information attached to program elements such as methods, fields, classes, and packages. The JSE v5.0 and the C#/VB .NET languages provide language standards to attach annotations to program elements. A good example of applying annotations is declaring transactions in the JEE and .NET frameworks. For example, the following annotation declares the transaction attribute of the method *update()* in EJB 3.0:

```
@TransactionAttribute
      (TransactionAttributeType.REQUIRED)
  public void update (double newvalue)
      throws Exception
```

[8] http://www.aspectwerkz.codehaus.org/

13.7 Performance Monitoring of ICDE with AspectWerkz

When running in production, it is desirable to be able to inject performance monitoring code into ICDE components without recompiling the complete application. Using aspects, this can be achieved using LTW. Hence the ICDE team starts to design an aspect-based architecture using AspectWerkz as shown in Fig. 13.4.

In this architecture, the performance instrumentation for different ICDE components is encapsulated in a dedicated aspect that can be injected into the ICDE application. This is necessary because the metrics that must be recorded are different in nature. For example, the performance monitoring of a JMS server measures both the message processing rate and the message size, while the instrumentation of SQL statements measures response time.

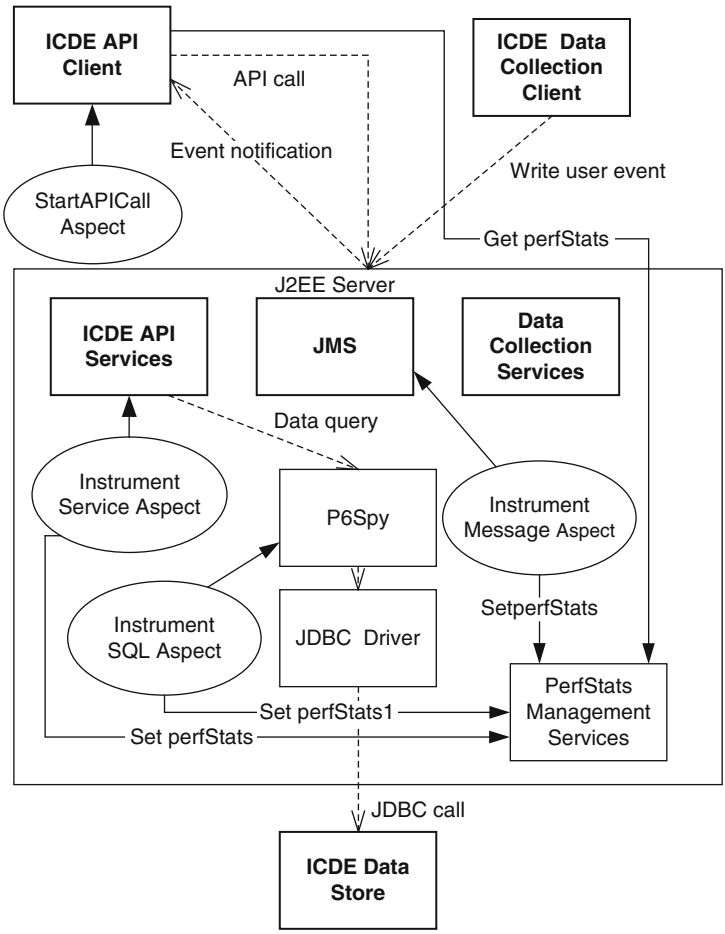

Fig. 13.4 ICDE 2.0 aspect-based architecture for ICDE performance monitoring

In order to instrument the database query response time, an open source component, P6Spy,[9] is used. This acts as a layer between the JEE connection pool and the JDBC drivers, capturing the SQL statements issued by JEE application. An aspect must also be applied to this component to retrieve the SQL statement information.

Once all the performance data is captured, there are a variety of options to make it available for subsequent processing. It can be simply written to a log file periodically or loaded into a database. A more flexible and efficient solution to provide direct access to live system performance data is to use a standard protocol such as Java Management eXtension (JMX)[10] that existing JEE management tools can display and track.

```
public class InstrumentSQLAspect
{
   public Object logJdbcQueries(final JoinPoint joinPoint)
            throws Throwable
   {
      //access Runtime Type Information
      MethodRtti rtti = (MethodRtti)joinPoint.getRtti();
      String query = (String) rtti.getParameterValues()[0];
      Long startTime = System.currentTimeMillis();
      //execute the method
      final Object result = joinPoint.proceed();
      Long endTime = System.currentTimeMillis();
      // log the timing information for this SQL statement execution
      perfStatsManager.log(query,"Statement",endTime-startTime);
      return result;
   }

   public Object logValuesInPreparedStatement(final JoinPoint
   joinPoint) throws Throwable
   {
      MethodRtti rtti = (MethodRtti)joinPoint.getRtti();
      Integer index = (Integer)rtti.getParameterValues()[0];
      Object value = rtti.getParamterValues()[1];
      String query = "index="+ index.intValue()+ "value="
                        + value.toString();
      Long startTime = System.currentTimeMillis();
      //execute the method
      Final Object result = joinPoint.proceed();
      Long endTime = System.currentTimeMillis();
      //log the timing information for this PreparedStatement
      //execution
      perfStatsManager.log(query, "PreparedStatement", endTime-
      startTime);
      return result;
   }
};
```

Fig. 13.5 SQL statement instrumentation aspect implementation

[9] http://www.p6spy.com/

[10] http://www.java.sun.com/products/JavaManagement/

```
<aspectwerkz>
   <system id="ICDE">
     <package name="com.icde.perf.aop">
      <aspect class="InstrumentSQLAspect"
            deployment-model="perThread">
         <pointcut name="Statement" expression=
            "execution(* java.sql.Connection+.prepare*(..))" />
         <pointcut name="PreparedStatement" expression=
            "execution(void java.sql.PreparedStatement+.set*(..))" />
         <advice name="logJdbcQueries(final JoinPoint joinPoint)"
            type="around" bind-to="Statement" />
         <advice name="logValuesInPreparedStatement(final JoinPoint
            joinPoint)" type="around" bind-to="PreparedStatement" />
      </aspect>
     </package>
   </system>
</aspectwerkz>
```

Fig. 13.6 `InstrumentSQLAspect` XML definition file

To illustrate the design, implementation and deployment of AspectWerkz aspects, we'll describe in detail the `InstrumentSQLAspect`. To measure SQL statement response times, we need to locate all method calls where a `java.sql.Statement` is created and inject timing code immediately before and after the SQL query is executed. We also have to trace all method calls where a value is set in a `java.sql.PreparedStatement` instance. The resulting code snippet for the `InstrumentSQLAspect` is illustrated in Fig. 13.5.

The next step is to compile the aspects as a normal Java class with the AspectWerkz libraries. The weaving rules for binding the advice to the pointcut is specified in the *aop_xml* file as shown in Fig. 13.6.[11]

LTW for AspectWerkz is achieved by loading the AspectWerkz library for the JDK v5. The ICDE application can then be booted normally and the aspect code will be weaved in at load-time

In summary, using AOP techniques, instrumentation code can be separated and isolated into aspects. The execution of the aspects can be weaved into the system at runtime without the need to recompile the whole system.

13.8 Conclusions

AOP was originally introduced as a programming mechanism to encapsulate crosscutting concerns. Its success has seen aspect-oriented techniques become used in various application domains, such as middleware frameworks. It has also

[11]Note that as JEE containers are multi-threaded, and individual requests are handled by threads held in a thread pool, the aspect is deployed in *perThread* mode.

spawned modeling and design techniques which influence the architecture of a software system built using aspect-oriented techniques.

AOP brings both opportunities and challenges for the software architect. In limited domains, AOP has demonstrated a great deal of promise in reducing software complexity through providing a clear separation and modularization of concerns. Fruitful areas include further integrating AOP and middleware to increase the flexibility of configuring middleware platforms. Even in this example though, challenging problems remain, namely coordinating multiple aspects to deal with conflicts, as crosscutting concerns are not completely orthogonal.

Aspect oriented design and implementation requires the support of efficient AOP tools. With such tools, on-going research and development is still attempting to provide better solutions in several areas, namely:

- *Maintenance*: Designing quality aspect-oriented systems means paying attention to defining robust pointcuts and sensibly using aspect inheritance. Pointcuts that capture more join points than expected or miss some desired join points can lead to brittle implementations as the system evolves. Consequently an efficient debugging tool is needed to detect the faulty join point and the pointcut implementation.
- *Performance*: Using AOP introduces extra performance overheads in applications, both during the weaving process and potentially at runtime. The overhead of AOP needs to be minimized to provide good build and runtime performance.
- *Integration*: The reusability of aspects hasn't been explored sufficiently, so that designers could utilize libraries of aspects instead of developing each aspect from scratch. As each AOP tool only provides aspect implementations specific to its own AOP model, an aspect implemented by one AOP model cannot be easily weaved into a system with aspects using a different AOP model. This is potentially a serious hurdle to the adoption of aspect-orientation in a wide range of software applications.

In summary, aspect-oriented techniques are developing and maturing, and proving themselves useful in various application and tool domains. These include security, logging, monitoring, transactions and caching. Whether aspect-orientation will become a major design and development paradigm is very much open to debate. However it seems inevitable based on current adoption that aspect-oriented techniques will continue to be gradually infused into the software engineering mainstream.

13.9 Further Reading

A good comparison of four Java AOP tools, namely AspectJ, AspectWerkz, JBoss AOP and Spring AOP, in terms of their language mechanisms and development environments is:

M. Kersten, *AOP Tools Comparison*. IBM developerWorks, http://www-128.ibm.com/developerworks/library/j-aopwork1/

A source of wide-ranging information on aspects is maintained at the AOSD wiki at:

http://www.aosd.net/wiki/index.php?title=Main_Page

The (deprecated) home page for *Aspectwerkz* is

http://www.aspectwerkz.codehaus.org/

AspectJ documentation can be found at:

http://www.eclipse.org/aspectj/docs.php

Good practical guides to AspectJ and aspects in database applications are:

Ramnivas Laddad, *Aspectj in Action: Enterprise AOP with Spring Applications*, Manning Publications, 2009.
Awais Rashid, *Aspect-Oriented Database Systems*, Springer-Verlag, 2009.