In this first half of a two-part article, author Ramnivas Laddad provides a conceptual overview of the new Java™ metadata facility and shows where AOP could most benefit from the addition of metadata annotations. He then walks you through a five-part design refactoring, starting with a metadata-free AOP implementation and concluding with one that combines the Participant design pattern with annotator-supplier aspects.

The new Java metadata facility, a part of J2SE 5.0, is perhaps the most significant addition to the Java language to date. By providing a standard way to attach additional data to program elements, the metadata facility has the potential to simplify and improve many areas of application development, including configuration management, framework implementation, and code generation. The facility will also have a particularly significant impact on aspect-oriented programming, or AOP.

The combination of metadata and AOP raises important questions, including:

- What is the impact of the metadata facility on AOP?
- Is metadata-fortified AOP optional, or is it necessary?
- What are the guidelines for using metadata effectively with AOP?
- How will this combination affect the adoption of AOP?

I'll begin to answer these questions in this two-part article, the second in the new AOP@Work series. In this first half of the article I'll start with a conceptual overview of metadata and the new Java metadata facility. I'll also explain the difference between supplying metadata and consuming it, and provide some common programming scenarios that invite the use of metadata annotation. Next, I'll quickly review the basics of AOP's join point model and explain where it would benefit from metadata fortification. I'll conclude with a practical example, evolving a design through five stages using metadata-fortified AOP. In Part 2, I'll demonstrate a novel way to view metadata as a signature in a multidimensional concern space, talk about the impact of metadata on AOP adoption, and conclude with some guidelines for effectively combining AOP and metadata.
Throughout the article I'll put the concepts presented into practice using examples from three of the leading AOP implementations: AspectJ, AspectWerkz, and JBoss AOP. See Resources for a listing of introductory articles about the Java metadata facility and aspect-oriented programming.

### Metadata concepts

*Metadata* is data about data. In the programming language context, metadata is additional information attached to program elements such as methods, fields, classes, and packages. Metadata is expressed using program elements called *annotations*. The semantics associated with metadata range from mere documentation to execution-behavior modification. For example, you could use metadata to describe the author and the copyright holder of a class, in which case it would have no effect on program execution; or you could use it to describe method properties such as transactionality characteristics, which would likely modify the behavior of the method, as I'll explain later in the article.

#### About this series

The AOP@Work series is intended for developers who have some background in aspect-oriented programming and want to expand or deepen their knowledge. As with most developerWorks articles, the series is highly practical: you can expect to come away from every article with new knowledge that you can put immediately to use.

Each of the authors contributing to the series has been selected for his leadership or expertise in aspect-oriented programming. Many of the authors are contributors to the projects or tools covered in the series. Each article is subjected to a peer review to ensure the fairness and accuracy of the views expressed.

Please contact the authors individually with comments or questions about their articles. To comment on the series as a whole, you may contact series lead Nicholas Lesiecki. See Resources for more background on AOP.

While there are numerous metadata tools for the Java language (XDoclet being the most well known), metadata annotations have only been incorporated into the core Java language with Java 5.0. The Java metadata facility (JSR 175; see ) includes a mechanism for adding custom annotations to your Java code, as well as providing a programmatic access to metadata annotation through reflection.

Key to understanding and using metadata are the concepts of supply and consume. A metadata *supplier* is a facility that associates annotation instances with program elements, whereas a *consumer* is a facility that reads, interprets, and acts on the annotation instances. In the next sections I'll discuss these concepts in more detail.

### Supplying metadata

A metadata facility defines a mechanism to supply annotations to program elements. Optionally, a metadata facility can specify a way to define annotation types. Annotation types specify a template to create annotation instances, much the same way classes specify a template to create objects. The metadata facility can then check annotation instances against the annotation types. A metadata facility may also specify a general way to consume annotations. One thing a metadata facility does not do is define the interpretation and semantics associated with annotations. That is left up to the consumer, as I'll discuss in the next section.
The capabilities of metadata facilities vary. For example, the Javadoc specification allows you to specify annotations in the comments associated with a program element. Alternatively, some metadata facilities use a separate document (often an XML document) to express metadata. For example, EJB deployment descriptors specify additional characteristics of enterprise beans. Unlike the Javadoc facility, this approach loosely couples program elements and metadata; on the downside, it requires developers to modify multiple sources describing the same element.

The Java metadata facility adds new language support for metadata to declare annotation types and annotate program elements. It also makes it possible to retain metadata at source code level in class files, and at runtime controlled by a retention policy.

The supplier of metadata may include a facility to attach certain annotations in a crosscutting manner rather than individually supplying them to multiple elements. Due to the crosscutting nature of such annotations, AOP is a good way to supply them. I'll examine the details of such a facility later in this article.

The choice of metadata facility affects the experience of expressing metadata, but the fundamental idea of associating additional data to program elements is common to all metadata facilities.

### Consuming metadata

Creating value from metadata annotations requires consuming them. Metadata can be consumed in a variety of ways, and understanding these uses will help put the combination of AOP and metadata into perspective. The following use cases should also help you understand how annotation supplied for non-AOP purposes can be consumed in an AOP implementation.

---

**In the eye of the beholder**

Metadata is additional information associated with program elements. What constitutes "additional," however, is subjective. Further, the target programming language influences what programmers consider to be inherent information about elements rather than metadata. For example, consider checked exceptions independent of the Java language. They could be thought of as metadata that instructs the compiler. In fact, in languages without checked exceptions (such as C#), metadata is a good choice to convey the same information.

Similarly, in weakly typed languages even the argument and return types of a method could be expressed using metadata. Or consider the Java language without generics. Frameworks such as Hibernate and EJB use metadata to specify the same information that can be defined using Java generics. Even modifiers such as access specification could be considered metadata in these cases. In fact, if someone had seen the need sooner, chances are that the Java metadata facility's built-in `@Override` annotation would have been implemented with an extra modifier -- say the `override` keyword. The bottom line is that the difference between a program element and metadata depends on your perspective.

---

**Code generation**

Code generation is perhaps the most familiar way to use metadata. With XDoclet-like tools, you can consume annotations specified as Javadoc tags to produce artifacts such as XML documents or Java code. The generated code in turn affects the runtime behavior of the annotated elements. A new release of XDoclet that supports the Java metadata facility is being developed. The command-line utility `apt` (Annotation Processing Tool), a part of the Java 2 SDK 5.0 distribution, also provides a way to process annotations by writing plugins. For example, Contract4J, a recently
released contract enforcement tool, uses apt to generate aspects to enforce design-by-contract (DBC) contracts.

**Programmatic behavior modification**
The standard metadata facility offers a way to keep annotation available at runtime. It also allows you to programmatically access annotation instances using reflection. The annotation instances can then be used much like other objects to modify the behavior of the program. Such programmatic consumption may also let programmers skip the code generation route for applications where the generated artifacts only allow information encoded in the annotations to be read.

**Framework consumption**
Metadata is commonly used to facilitate communication between program elements and frameworks or tools such as EJB, EMF, and TestNG. The framework itself may choose to use code generation, reflective access, or AOP to apply certain logic to execution. The proposed use of annotations in EJB 3.0, such as `@Remove` and `@Session`, communicates with the framework the role of the program element. The Eclipse Modeling Framework uses annotations (currently expressed as Javadoc tags) to create UML models and XML persistence support. And on the tool side, TestNG (for example) uses metadata to communicate between test cases and the test execution tool.

**Language extension**
This use of metadata extends the underlying programming language and the compiler. Associating semantic properties with metadata means that compiled classes may have a different structure and behavior from those without it (see Resources for a further discussion on this topic). The recently announced Pluggable Annotation Processing API (JSR 269) may lead to a standard way to process annotations for this purpose. The use of metadata to extend the Java language can be both powerful and dangerous: On the one hand, annotations allow us to add new features to the Java language without modifying the core language, thus making it an open language; in the best-case scenario, principled extensions could overcome some of the limitations of the host language. On the other hand, a nonstandard, ad hoc, or incoherent set of annotations could result in incomprehensible code.

Incidentally, enabling AOP within plain object-oriented languages is one example of using metadata for language extension. AspectWerkz and JBoss AOP use metadata to change the semantics of a class as an aspect, a data field as a pointcut, a method as an advice, and so on. AspectJ 5 will similarly support the `@AspectJ` syntax, which is a result of the merging of the AspectJ and AspectWerkz projects. See Resources to learn more about the different AOP implementations and Java language extension.

In the next section I'll quickly review the basics of AOP's join point model, then explain where it could be beneficially fortified with metadata.
Metadata and the join point model

A join point is an identifiable point in the execution of a system. The join point model, the most fundamental and distinguishing concept in AOP, defines which join points in a system are exposed and how they are captured. To implement crosscutting functionality using aspects, you need to capture the required join points using a programming construct called a pointcut.

Pointcuts select join points and collect the context at selected join points. All AOP systems provide a language to define pointcuts. The sophistication of the pointcut language is a differentiating factor among the various AOP systems. The more mature the pointcut language, the easier it is to write robust pointcuts. See the first article in the AOP@Work series for a detailed discussion on the importance of pointcut language (reference in Resources).

Capturing join points

A pointcut specifies the properties of a given element of a program. A major portion of the art and science of writing good aspects lies in writing robust pointcuts; the other major portion being well-designed aspect inheritance. Pointcuts that capture more join points than expected or miss the expected join points can lead to a brittle implementation as the system evolves. Writing good pointcuts is key to mastering AOP, although it is often a struggle for newcomers.

Currently, the most common way to capture join points utilizes the implicit properties of program elements, including static properties such as signature (which consists of type and method name, argument types, return type and exception type, etc.) and lexical placement, as well as dynamic properties such as control flow. Judiciously using wildcards in join point signatures often leads to good, succinct pointcut definitions. You can also compose individual pointcuts to form more complex ones. The join point model based on implicit properties of program elements is both powerful and useful, as evidenced by the current success of AOP in several production systems.

The implicit information available in the signature of program elements is often enough to capture the required join points. In this model, sometimes called dynamic crosscutting, the combination of implicit data, wildcards, and dynamic properties such as control flow lets you capture join points without modifying the captured program elements. For example, you can capture all RMI calls by specifying operations throwing RemoteException in a class implementing the Remote interface. A pointcut such as execution(* Remote+.*(..) throws RemoteException) (defined in AspectJ) neatly captures all RMI operations without modifying the program elements, and also ensures a robust pointcut. The beauty here is being able to capture join points without additional collaboration beyond that required by the RMI infrastructure.

Capturing join points with metadata

Signature-based pointcuts cannot capture the join points needed to implement certain crosscutting concerns. For example, how would you capture join points requiring transaction management or authorization? Unlike the RMI example, nothing inherent in an element’s name or signature suggests transactionality or authorization characteristics. The pointcut required in these situations can get unwieldy, as you can see for yourself in the following example. (The example is in AspectJ but pointcuts in other systems are conceptually identical.)
pointcut transactedOps()
   : execution(public void Account.credit(..))
   || execution(public void Account.debit(..))
   || execution(public void Customer.setAddress(..))
   || execution(public void Customer.addAccount(..))
   || execution(public void Customer.removeAccount(..));

Situations like this one invite the use of metadata to capture the required join points. For example, you could write a pointcut as shown below to capture the execution of all the methods carrying the `@Transactional` annotation.

pointcut execution(@Transactional * *.*(..));

Metadata and modularity

While the above example may make using metadata to capture join points seem like a no-brainer, it’s important to consider the implications of such usage, particularly when it comes to modularity. Once you begin using metadata in your pointcuts, methods must carry the appropriate annotation to collaborate in the crosscutting implementation of aspects that use them, as shown here:

```java
public class Account {
    ...
    @Transactional(kind=Required)
    public void credit(float amount) {
        ...
    }
    @Transactional(kind=Required)
    public void debit(float amount) {
        ...
    }
    public float getBalance() {
        ...
    }
    ...
}
```

Similarly, the `addAccount()`, `removeItem()`, and `setAddress()` methods in the `Customer` class now have to carry the `@Transactional` annotation.

Most AOP practitioners are currently implementing transactional and authorization concerns with existing AOP support, typically through the use of design patterns utilizing aspect inheritance. As you'll see in this article, however, adding metadata to AOP systems can improve them considerably. I'll talk more about how adding metadata effects the modularity of AOP systems, as well as the scenarios where metadata is most beneficial in Part 2 of this article. In the next section I'll begin to explain more concretely how AOP implementations can be extended to incorporate metadata.
Metadata-fortified AOP

AOP systems and their join point models can be augmented by consuming metadata annotations. JBoss AOP, Spring AOP, AspectWerkz, and AspectJ all provide or plan to provide mechanisms to utilize metadata. JBoss AOP and AspectWerkz support metadata in their current versions. Spring AOP supports metadata by letting you write pointcuts programmatically, by implementing the `org.springframework.aop.Pointcut` interface. The upcoming version of AspectJ will support metadata by modifying the AspectJ language.

In the previous section I showed you the basics of how an AOP system can consume metadata, using the example of picking out methods with the `@Transactional` annotation. In this section and throughout the remainder of the article I'll focus on the specifics of combining AOP and metadata.

While the examples in this article focus on AOP implementations that support metadata, it's possible to consume metadata even when the core AOP system doesn't directly support it, by piggybacking on code generation support. For example, Barter is an open source tool that uses annotations and a code generation pre-step to enforce DBC contracts with an older version of AspectJ that doesn't support capturing join points based on Javadoc tags. Today, Contract4J performs a similar task using Java metadata facility-style annotations. See Resources to learn more about these tools.

Metadata support in AOP systems

To support metadata-based crosscutting, an AOP system needs to provide a way to consume and supply annotations. I'll explain the basics of both types of support here. In the sections that follow I'll offer more detail on the specifics of each.

Support for consuming annotations

An AOP system that supports consuming annotations will let you select join points based on annotations associated with program elements. The current AOP systems that offer such support do so by extending the definition for various signature patterns to allow annotation types and properties to be specified. For example, a pointcut could select all the methods carrying an annotation of type `Timing`. Further, it could subselect only methods with the value property exceeding, say, 25. To implement advice dependent on both annotation type and properties, the system could include pointcut syntax capturing the annotation instances associated with the join points. Last, the system could also allow advice to access annotation instances through reflective APIs.

Support for supplying annotations

With the standard Java metadata facility, one annotation instance is declared for each program element being annotated. If the same annotation declaration appears for multiple program elements, however, it can lead to unnecessary clutter. It is possible to harness AOP's crosscutting mechanism to declare an annotation once for all affected elements. An AOP system that supports supplying annotations will let you attach annotations to program elements in a crosscutting manner. For example, you could attach `@Secure` annotation to all methods of the `Account` class with a simple declaration, and without having to scatter annotations across all those methods.
Not all AOP systems support all of the possibilities mentioned here, as you'll learn from the more detailed discussion that follows. I'll start with a look at how several AOP systems provide support for consuming annotations.

**Consuming annotation in AOP**

Pointcut syntax differs in the various metadata-fortified AOP systems. You can see the difference for yourself by studying how each system handles a pointcut that captures all methods carrying the `@Transactional` annotation instance. In these examples I'll focus on selecting join points by annotation type; further down I'll explain some of the other factors that can weigh in on join point selection.

**AspectJ5**
The AspectJ 5 syntax (in milestone releases at the time of this writing) extends the definition of signature for type, method, and fields to include metadata annotation as a part of the signature, as shown here:

```java
pointcut transactedOps(): execution(@Transactional * *.*(..));
```

If you were to use the @AspectJ-styled definition in AspectJ 5, the same pointcut would be defined as shown here:

```java
@Pointcut("execution(@Transactional * *.*(..))")
void transactedOps();
```

**AspectWerkz**
Like most other AOP tools, AspectWerkz's pointcut syntax closely resembles that of AspectJ. The pointcut declaration in the following snippet uses metadata-style annotation, although the XML style would have identical pointcut expression:

```java
@Expression("execution(@Transactional * *.*(..))")
Pointcut transactedOps;
```

Note that AspectWerkz uses metadata to extend the Java programming language to support AOP, as shown in the example. Hence, AspectWerkz uses metadata for two purposes: to extend the semantics of programming elements and to implement crosscutting based on metadata. In the above example I've focused on the latter usage.

**JBoss AOP**
JBoss AOP isn't conceptually much different from the other AOP systems, although it uses a different syntax. The pointcut shown here is equivalent to the other examples but expressed in JBoss's XML syntax:

```xml
<pointcut name="transactedOps" expr="* *->@Transactional(..)"/>
```

As you can see, there isn't much conceptual difference between how different AOP systems capture join points based on the metadata annotation attached to them.
Selecting by annotation property

Type isn't the only consideration when selecting join points: properties may also be considered. For example, the following pointcut captures all methods with \@Transactional annotation with the \texttt{value} property set to \texttt{RequiredNew}:

\begin{verbatim}
execution(@Transactional(value==RequiredNew) \ast \ast(...))
\end{verbatim}

At the time of this writing, no AOP system supports annotation properties-based pointcuts. Instead, each system relies on dynamic decision inside advice to inspect properties and invoke appropriate logic (or at least the use of dynamic checks with an \texttt{if()} pointcut). There are certain advantages to annotation properties-based pointcuts, especially for compile-time checks and efficiency of static selection. Future versions of AOP systems will support this sort of pointcut.

Exposing annotation instances

Since advice logic can depend on both the metadata annotation \texttt{type} and \texttt{instance properties}, each annotation instance must be exposed as context in the same manner as other context at the join point (for example, object, method arguments, etc.). AspectJ 5 extends its existing pointcuts and adds a few new ones to expose annotations. For example, the following pointcut collects the annotation instance of the \texttt{Transactional} type associated with the captured join point:

\begin{verbatim}
pointcut transactedOps(Transactional tx)
  : execution(@Transactional \ast \ast(...)) \&\& \@annotation(tx);
\end{verbatim}

Once captured, annotation instances may be used just the same way as any other context. For example, in the following advice, the captured annotation is queried for its properties:

\begin{verbatim}
Object around(Transactional tx) : transactedOps(tx) {
  if(tx.value() == Required) {
    \ldots implement the required transaction behavior
  } else if(tx.value() == RequiredNew) {
    \ldots implement the required-new transaction behavior
  }
  \ldots
}
\end{verbatim}

Most AOP systems use only the reflective API to expose annotation instances at the captured join point, and do not allow you to bind annotations as join point context. In these cases you can query the object representing the advised join point for its associated annotations. AspectJ offers both reflective access and the traditional join point context. See \href{Resources}{Resources} to learn more about AspectJ's support for exposing annotation instances.

Supplying annotation in AOP

The basic idea behind supplying annotation using an AOP construct is to avoid cluttering the program element’s definition with annotations. Conceptually, such a construct allows you to attach annotations to program elements in a crosscutting manner. At first glance, supplying annotations using an AOP construct, and then using those annotations to capture join points seems unnecessary and convoluted. After all, if you can identify the join points needing annotations, you can write a pointcut and advise those join points directly. However, supplying annotations
in a crosscutting manner can be very helpful. First, such a declaration can act as a conduit to communicate with non-AOP clients. Further, supplying annotation in a crosscutting mechanism makes it possible to design a more loosely coupled system while still avoiding annotation clutter.

I'll explain some of the design opportunities presented by using an AOP construct to supply annotation toward the end of this article. For now, I'll show you the basic syntax to supply annotation in a crosscutting manner.

**Supplying annotation syntax**

The proposed syntax for AspectJ extends the current static crosscutting constructs to create a new `declare annotation` syntax. The following code snippet will attach an annotation of type `Authenticated` with the `permission` property set to `banking`:

```
declare annotation : * Account.*(..)
    : @Authenticated(permission="banking");
```

The `@AspectJ` pointcuts also support the same functionality through use of the `@DeclareAnnotation`, where the same declaration can be written as follows:

```
@DeclareAnnotation("* Account.*(..)")
@Authenticated(permission="banking")
void bankAccountMethods();
```

In JBoss AOP, when using XML-style aspects, you attach annotations using an `annotation-introduction` element. The `invisible` attribute indicates if the annotation should not be retained at runtime (equivalent to `RetentionPolicy.SOURCE` in the standard Java metadata facility).

```
<annotation-introduction expr="method(* Account-*{..})"
    invisible="false">
    @Authenticated(permission="banking")
</annotation-introduction>
```

As you can see, the principle for supplying annotation is the same across the various AOP systems, although the syntax differs.

**AOP design with metadata**

Combining metadata with AOP is fairly simple, as you've seen in the previous sections. The important thing is knowing where to apply metadata-based crosscutting and where to leave it out. In this section, I'll begin to answer this question by considering how a system might evolve from an AOP implementation that uses implicit properties of join points to one that incorporates metadata-based pointcuts. In Part 2 I'll delve deeper into the conceptual considerations of choosing a metadata-driven approach.

The discussion in this section should be helpful in two ways: First, it's important to understand that using metadata isn't always your first or only choice as an AOP practitioner. Second, the example implementation here will serve as a guide for how to evolve a design when you do decide to employ metadata-based crosscutting.
The example for this exercise is a transaction management program. While I've used AspectJ to develop all the code for the example, implementations in other AOP systems are conceptually identical. Consider each of the steps in this exercise a refactoring of the original design. The goal is to gradually decouple the system and improve its modularity.

**Version 1: Naive aspect**

My first attempt at modularizing a crosscutting concern employs a system-specific aspect that contains pointcut definitions as well as advice to that pointcut. This is a very simple scheme and very often the first design you'll encounter in learning about AOP. Figure 1 shows the schematics of a design that employs one aspect:

**Figure 1. First attempt at implementing transaction management using AOP**

Listing 1 implements the above design:

**Listing 1: Transaction management aspect for a banking system**

```java
public aspect BankingTxMgmt {
  pointcut transactedOps() : execution(void Customer.setAddress(..))
    || execution(void Customer.addAccount(..))
    || execution(void Customer.removeAccount(..))
    || execution(void Account.credit(..))
    || execution(void Account.debit(..));

  Object around() : transactedOps() {
    try {
      beginTransaction();
      Object result = proceed();
      endTransaction();
      return result;
    } catch (Exception ex) {
      rollbackTransaction();
      return null;
    }
  }

  // ... implementation of beginTransaction() etc.
}
```
This scheme works well for aspects that require very little knowledge of the underlying system. For example, if you wanted to enable pooling, you could write a generic aspect advising calls to the creation and destruction of a resource being pooled. For crosscutting functionality that could not capture the needed join points in a generic manner, however, this approach would be limiting. First, the aspect isn't reusable because the pointcut definition is system specific. Second, changes to the system could result in the the aspect also needing to be changed. In other words, this first version of the system leaves us with N-to-one coupling between program elements and the pointcut. Because this isn't the best option, I'll try again.

**Version 2: Reusable-base aspect**

My second attempt improves the example aspect by making it reusable. I've extracted the reusable portion of the aspect and added a subaspect defining pointcut(s) in a system-specific way. Figure 2 shows the structure after the base aspect has been extracted:

**Figure 2. Extracting a reusable transaction management aspect**

Listing 2 shows the base aspect, which is now reusable. You'll note two changes compared to Listing 1: the aspect is marked `abstract` and the `transactedOps()` pointcut is marked `abstract` with its definition removed:
Listing 2. Reusable transaction management base aspect

```java
public abstract aspect TxMgmt {
    public abstract pointcut transactedOps();

    Object around() : transactedOps() {
        try {
            beginTransaction();
            Object result = proceed();
            commitTransaction();
            return result;
        } catch (Exception ex) {
            rollbackTransaction();
            return null;
        }
    }
}
... implementation of beginTransaction() etc.
```

Next, I need to write a subaspect for my base aspect. The subaspect below defines a pointcut that captures the join points needing transaction management support. Listing 3 shows a banking-specific subaspect that extends the `TxMgmt` aspect in Listing 2. The subaspect defines the `transactedOps()` pointcut with an identical definition, as in Listing 1.

Listing 3. System-specific subaspect

```java
public aspect BankingTxMgmt extends TxMgmt {
    public pointcut transactedOps():
        execution(void Customer.setAddress(..))
        || execution(void Customer.addAccount(..))
        || execution(void Customer.removeAccount(..))
        || execution(void Account.credit(..))
        || execution(void Account.debit(..));
}
```

While it's an improvement, this design still leaves us with an \(N\)-to-1 dependency between the subaspect and classes. Any changes in the transactionality requirements for the banking system would require the pointcut definition of `BankingTxMgmt` to be modified. Since this is far from ideal, I'll try again.

Version 3: The Participant pattern

I've addressed the problem of reusability above, but I still need to get rid of that \(N\)-to-1 dependency. I can do so by employing the Participant pattern (see Resources). Instead of using one subaspect for my whole system, I can use many subsaspects -- one per subsystem -- making it possible to write a relatively stable pointcut. A subsystem in this context could be a package, a set of packages, or even a class. Figure 3 shows the structural relationship between different elements:
Figure 3. Employing the participant design pattern

Listing 4 shows the `Customer` class with a participant subaspect that is responsible for defining the pointcut for the embedded class.

**Listing 4. Customer class with an embedded participant aspect**

```java
public class Customer {
    public void setAddress(Address addr) {
        ...
    }
    public void addAccount(Account acc) {
        ...
    }
    public void removeAccount(Account acc) {
        ...
    }
    ...
    private static aspect TxMgmtParticipant extends TxMgmt {
        public pointcut transactedOps() {
            : execution(void Customer.setAddress(..))
            || execution(void Customer.addAccount(..))
            || execution(void Customer.removeAccount(..));
        }
    }
}
```

The subaspect in the example `customer` class simply enumerates all the methods with wildcards for arguments. In practice, however, you would likely use wildcards to simplify the pointcut definition. For example, you could declare all public methods of the class to be captured by the `transactedOps()` pointcut using the following definition:

```java
public pointcut transactedOps() {
    : execution(public * Customer.*(..));
}
```

In Listing 5 you can see how the `Account` class embeds a subaspect to participate in the system’s transaction-management functionality.
Listing 5. Account class with an embedded participant aspect

```java
public class Account {
    public void credit(float amount) {
        ...
    }
    public void debit(float amount) {
        ...
    }
    public float getBalance() {
        ...
    }
    ...

    private static aspect TxMgmtParticipant extends TxMgmt {
        public pointcut transactedOps()
            : execution(void Account.credit(..))
                || execution(void Account.debit(..));
    }
}
```

As with the `Customer` class, adding a step would simplify the pointcut. For example, what if I realized that all public methods, except the `getBalance()` method, needed to be executed in transaction management? I could define the pointcut to capture this realization as follows:

```java
public pointcut transactedOps()
    : execution(public void Account.*(..))
        && !execution(float Account.getBalance());
```

Now if a class changes, I need to modify only the nested subaspect in that class. Instead of a large \( N \), I've reduced the system coupling to a smaller number of program elements captured by each subaspect, say \( n \)-to-1. Further, if a class changes with respect to its transaction management needs, I need only change the pointcut in the embedded participant aspect, thus preserving locality.

This example illustrates an important point often missed in AOP-related discussions: If you try to find a signature pattern for the whole system, you will meet with unpleasant surprises -- unstable, complex, and incorrect pointcuts. When you consider subsets of the system, however, you can often find a signature pattern that works quite well for the subsystem. The use of the Participant pattern with one aspect per class considers a class as a subsystem, though any logical division into a subsystem will do just fine.

This solution is a reasonable one for most situations. Its downside is that the classes have a direct dependency on the base aspect, so the base aspect must always be present in the system. Another issue with this solution is that its crosscutting functionality won't apply unless the classes explicitly "participate" in the collaboration, by embedding a nested subaspect. This issue has to do more with the nature of the crosscutting functionality than the solution itself. And, as you'll shortly see, I can improve it a little.
Version 4: Metadata-based pointcut

In this iteration I'm going to modify each method to have an annotation and go back to one subaspect for the system (as in Version 2). This time, however, my subaspect will use a metadata-based pointcut to capture the required join points -- the methods that carry the annotation I just supplied. The subaspect itself will be reusable across systems. Figure 4 shows the schematic of this version.

Figure 4. Metadata driven transaction management

With a metadata-based subaspect, when a join point in a class changes its characteristics only the annotation for the join point needs to be changed. Listing 6 shows the subaspect that extends theTxMgmt aspect from Version 2 and defines the transactedOps() pointcut by simply capturing all the methods that carry an annotation of the Transactional type.

Listing 6. Metadata-driven transaction management subaspect

```java
public aspect MetadataDrivenTxMgmt extends TxMgmt {
    public pointcut transactedOps(): execution(@Transactional * *.*(..));
}
```

The class must collaborate with the subaspect by attaching an annotation of the Transactional type to every method that needs to be executed under transaction management. Listing 7 shows the implementation of the Customer class with methods carrying annotations:
Listing 7. Customer class with annotation

```java
public class Customer {
    @Transactional
    public void setAddress(Address addr) {
        ...
    }

    @Transactional
    public void addAccount(Account acc) {
        ...
    }

    @Transactional
    public void removeAccount(Account acc) {
        ...
    }
}
```

Listing 8 shows a similar implementation of the `Account` class:

Listing 8. Account class with annotations

```java
public class Account {
    @Transactional
    public void credit(float amount) {
        ...
    }

    @Transactional
    public void debit(float amount) {
        ...
    }

    public float getBalance() {
        ...
    }
}
```

At this point I've established a one-to-one dependency between methods and the collaborating aspect. I've also removed the direct dependency between the aspect and the classes. As a result, if I want to change the base aspect I can now do so without any changes to the system anywhere.

The use of the base aspect is optional (that is you could collapse the hierarchy). However, separating the base aspect from the metadata-driven subaspect has a few advantages. First, the derived aspect has a choice of annotation type. In one system, it may use `Transactional` as the annotation type to capture join points, whereas in another system the annotation type might be `Tx`. Second, it pushes down the choice between Participant pattern and metadata-driven approach to the derived aspect. Third, this approach makes it possible to derive a transactional pointcut from business annotations such as `@Purchase` or `@OrderProcessing`. And last, it enables the combination of the metadata-driven approach and the Participant pattern-based approach.
By collaborating through annotations, the participation responsibility is shifted to individual methods (instead of a participant subaspect). The dependency between MetadataDrivenTxMgmt and classes is limited to annotation types and their associated semantics.

For most cases this iteration is as good as it gets. There is one particular scenario, however, where I can refactor the design one step further for optimum results.

**Version 5: Aspect as metadata supplier**

In certain situations most of the methods in a class need to carry an annotation (such as the one in Version 4). Further, many crosscutting characteristics will require one or more annotations per method. These circumstances can lead to many annotations declared for every method, a phenomenon often described as *annotation hell*. Annotation clutter can be reduced by combining the Participant pattern with annotator-supplier aspects. This is a useful option when there is a clear way to express participating join points. In such cases, the annotator-supplier design avoids the risk of missed annotations for join points.

**Figure 5. Aspect as metadata supplier**

Annotator aspects simply use one or more *declare* annotations: for example, *declare annotation : <Method pattern>:<Annotation definition>*. In this example, I've used the Participant pattern-like collaboration with one annotator aspect per class. However, doing so is not an inherent requirement of this design. You could, for example, implement one annotator aspect per package. The core idea is to find a suitable subsystem with a clear signature pattern or dynamic context information (control-flow, etc.) that captures the required join points and avoid annotation clutter in those cases. Listing 9 shows the *Customer* class with an annotator aspect.
**Listing 9. Customer class with embedded annotator aspect**

```java
public class Customer {
    public void setAddress(Address addr) {
        ...
    }
    public void addAccount(Account acc) {
        ...
    }
    public void removeAccount(Account acc) {
        ...
    }
    ...
    private static aspect Annotator {
        declare annotation: public Customer.*(..): @Transactional;
    }
}
```

Similarly, the `Account` class in Listing 10 includes an annotator aspect.

**Listing 10. Account class with embedded annotator aspect**

```java
public class Account {
    public void credit(float amount) {
        ...
    }
    public void debit(float amount) {
        ...
    }
    ...
    private static aspect Annotator {
        declare annotation: public Account.*(..): @Transactional;
    }
}
```

Now compare this implementation with the one using the Participant pattern in Version 3. That version suffered from one major drawback: it tied a class to particular aspects. In a sense, it was a very active participation -- the base aspect must always exist (since it is the base aspect for all participating aspects). With the annotator aspect approach, the participation occurs only at the level of a shared understanding of annotation type.

**Bridging annotation types**

A variation of this technique uses the annotator aspect as a bridge between annotations used for business purpose and annotations used for aspect implementation. For example, if you knew that all methods with `@Purchase` and `@OrderProcessing` annotation had to be transaction managed, you could write an aspect as shown in Listing 11.

**Listing 11. Translating business annotations into crosscutting annotations**

```java
public aspect BusinessTransactionBridge {
    declare annotation: @Purchase *.*(..): @Transactional;
    declare annotation: @OrderProcessing *.*(..): @Transactional;
}
```
This aspect attaches @Transactional annotation to all the methods with @Purchase or
@OrderProcessing annotation. This approach, in combination with the aspect in Listing 2 and
Listing 6, results in surrounding the methods' execution with transaction management logic.

Conclusion

Metadata is a way to express additional information about program elements. The new metadata
facility in the Java programming language enables the use of typed annotations. Using metadata
is quite simple, although consuming it brings up many choices. Aspect-oriented programming
presents itself as a principled consumer of metadata. The join point model augmented with
metadata makes AOP accessible by facilitating simpler pointcuts for crosscutting concerns that
would not be so easily specified by stable signature-based pointcuts.

In this first part of a two-part article, I've given you a high-level overview of metadata concepts
and explained how AOP can specifically leverage information contained in the metadata for
program elements. I've also briefly surveyed the mechanisms involved in supporting metadata-
based pointcuts for various AOP systems, and walked you through a five-part design refactoring to
demonstrate the application of metadata to an AOP system.

In the second half of the article I will delve further into the design considerations of defining and
using metadata with AOP as a consumer and supplier. I'll talk about how incorporating metadata
affects the obliviousness principle in AOP systems, as well as how it could impact on the adoption
of AOP systems. I'll also introduce you to a novel way to approach AOP as a signature in a
multidimensional concern space, a concept that is useful in everyday AOP practice as well as in
designing annotation types for non-AOP purposes.

Acknowledgments
I wish to thank Ron Bodkin, Wes Isberg, Mik Kersten, Nicholas Lesiecki, and Rick Warren for
reviewing this article.

© Copyright IBM Corporation 2005
Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)