



AOP@Work: AOP and metadata: A perfect match, Part 2

Multidimensional interfaces with metadata

Level: Intermediate

Ramnivas Laddad (ramnivas@aspectivity.com), Principal, Aspectivity

12 Apr 2005

In this second half of his two-part article on combining metadata and AOP, author and AOP practitioner Ramnivas Laddad suggests a novel way to conceptualize metadata as a signature in a multidimensional concern space. He also introduces a series of guidelines for effectively combining metadata and AOP and discusses the impact of metadata annotations on the adoption of aspect-oriented programming.

In the [first half of this article](#), I introduced the basics of the new Java™ metadata facility and explained how and where AOP's join point model could be most beneficially fortified by metadata annotation. I also gave an overview of existing metadata support in three leading AOP systems: AspectWerkz, AspectJ, and JBoss AOP. I touched on the impact of metadata on an AOP system's modularity, and concluded with a refactoring example demonstrating the gradual introduction of metadata annotation into an AOP system.

In this second half of the article, I'll suggest a novel way to look at metadata as a pathway into multidimensional signature space. This approach has a practical application in untangling element signatures and is also conceptually useful for designing annotation types, even for developers who do not practice AOP. I will also focus on the most beneficial use of metadata annotation in aspect-oriented programming. Among other things, I will demonstrate an effective use of AOP to reduce the loss of modularity that is sometimes associated with metadata annotation. I'll conclude the article with a set of guidelines for determining when and how best to use metadata. I'll also consider the effects of adding a metadata facility on the adoption of AOP.

The tyranny of the dominant signature

The signature of a program element such as a type, method, or field consists of a few components, including the element's name, access specification, base types, method arguments, exception specification, and so on. Not all components are applicable to every kind of program element, but in any case the name given to a program element is the most precious commodity, as it communicates the role of that element to its users.

In programming without metadata, you can use only one name in each program element. It is common in this scenario to name program elements after their dominant or primary role, which is usually the core logic implemented by the program element. For example, examine the signature for the credit method below:

```
public void credit(float amount);
```

The method name reflects only one property of the method: the business logic to perform the credit operation. All of the element's other characteristics are lost. While this signature is sufficient to inform clients using the method of its dominant role, it leaves other clients -- those that implement crosscutting concerns such as security and transaction management -- uninformed. This is the tyranny of the dominant signature, caused by the limitation of expressing the element with only one name.

An analogous situation is often discussed in AOP research literature: the tyranny of dominant decomposition, in which the prominent concerns (usually the business logic) dominate the class design, especially the inheritance hierarchy.

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.

Signature tangling

One way to ensure that an element's signature matches its crosscutting roles is to change its name to reflect all of its roles. For example, a credit operation that needed to also run inside a transaction and be authenticated might have the following signature:

```
public void transactional_authorized_credit(float amount);
```

While seeing a name like the one above does ensure that you will think about the method's crosscutting transaction and authentication requirements, the resulting code is ugly enough that few developers will utilize it. In fact, it is seen only under special circumstances such as using a special naming convention to mark methods for internal consumption.

The above approach can also pose serious problems as a system evolves. If a program element's crosscutting characteristics change, its name and all its referers will need to be changed accordingly. For example, if in the above example the system evolved so that credit operations no longer needed to be authorized, the method name would need to be changed to `transactional_credit()` and all of its method calls revised accordingly.

Another problem with this approach is that using names that combine elements from multiple concerns increases the cognitive burden on the caller. For example, the business caller of the above method is unconcerned with the method's other characteristics, but is still subjected to the other concerns.

If you are familiar with AOP, you already know that code-tangling -- mixing code from multiple concerns in a module -- can lead to obscure and difficult-to-maintain code. *Signature tangling* is similar. When a program element's name reflects its roles in multiple concerns, as it does in the above example, we call it a tangled signature.

Given a choice between signature tangling and obscure signatures (simply ignoring the other characteristics of the program elements and simplifying the above method name, for example, to `credit()`) most developers will go for simplicity. What you really need, however, is a third option. Adding metadata annotation to your element signatures lets you express non-core characteristics in a systematic way.

Untangle me, metadata!

With the metadata facility in Java 5.0, you can use typed annotations to convey non-core characteristics programmatically. For example, the `credit` method that needs to be executed in transaction management and authenticated for security purposes could be annotated as follows:

```
@Transactional
@Authorized
public void credit(float amount);
```

On the surface, the difference between using annotations versus the signature-tangled method name above seems superficial; after all, I've just moved non-core role designations from the method name to annotations. The real benefit to this approach is on the caller side, where the method's business clients do not need to understand other concerns. Business clients also need not modify their code when annotations attached to the method change.

In addition to untangling element signatures, annotations can be used to express any data associated with your code's crosscutting concerns. In this case, the annotation could incorporate parameters related to the concern. For example, the following method declares that its transaction must have the **Required** semantics, whereas the permission checked must be `"accountModification"`.

```
@Transactional(Required)
@Authorized("accountModification")
public void credit(float amount);
```

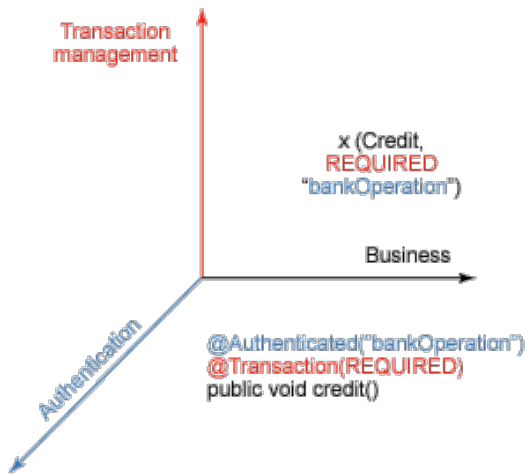
As you can see, annotations are very handy for untangling the names of crosscutting elements into multiple pieces. Essentially, annotations let you express a program element's non-core characteristics without tangling their names or placing undue burden on the caller side.

Metadata as a multidimensional signature

Conceptually, metadata can be seen as values in additional dimensions, which enable us to express the characteristics of a program element in a multidimensional space. Such a view facilitates a systematic approach toward dealing with metadata and aligning it with the goals of AOP. Even for developers who do not take an aspect-oriented approach, the multidimensional view of metadata is a valuable conceptual aid in designing annotation types.

You can approach the view of metadata as an enabler of multidimensional signature space in a systematic way: Each dimension of the space represents a *concern*, and the value expressed in each annotation represents a *projection into the multidimensional signature space*. Consider the signature for the `credit()` method just discussed. Figure 1 shows the `credit()` method in a three-dimensional signature space.

Figure 1. Multidimensional signature space using metadata



The name of the operation in Figure 1 essentially yields a value in a one-dimensional signature space. The only dimension of such a signature space is the business dimension and the value of the operation in that dimension (a.k.a. projection into that dimension) is "credit." Incorporating the `Transactional` annotation with the `value` property set to `Required`, I am able to evolve the example into a two-dimensional signature space. The value `Required` is the signature of the same method in the newly added `transaction management` dimension.

Similarly, consider the `Authorized` annotation with the `value` property set to `accountModification`. In this instance, the addition of annotation adds another dimension, resulting in a three-dimensional signature space. Most signatures will yield significant value into only a few dimensions. This is similar to projecting a two-dimensional geometrical point into a three-dimensional space: The value of such projection will be zero in one dimension.

Without annotation, element signatures typically ignore non-core dimensions completely. The main cause of this inattention is the unavailability of a proper means to express them. The new Java metadata facility provides a concise way to express program characteristics in every dimension.

Extreme metadata

You can stretch the multidimensional-signature concept to an extreme by considering each part of the signature as a dimension. For example, you would consider a part of the signature such as the exception specification a separate dimension. Looked at this way, there is no distinction between a signature-based pointcut and a metadata-based pointcut: There is only a pointcut that uses an extended signature comprised of what is known as the *inherent signature* and metadata. This concept and usage of metadata is sometimes called *explicit programming* (see [Resources](#)).

This style of multidimensional signature decomposition is useful without bringing AOP into picture; however, AOP practitioners will recognize that AOP addresses the same kind of decomposition of crosscutting concerns. This is one example of the fundamental synergy between metadata and AOP.

Mapping signature space to concern space

AOP systems such as Hyper/J (now evolved into Concern Manipulation Environment; see [Resources](#)) emphasize the multidimensional view of the concern space; with metadata, we also have a way to provide multidimensional signature space.

In the concept of metadata as a multidimensional signature, each consumer of the signature uses only the projection that is relevant to its concern. So, in the previous example, a projection of the point represented by the signature into the

business dimension would yield the name "credit". For the business concern's implementation all that matters is that the operation performed is the "credit operation." It does not know (or need to know about) values in any other dimension. For example, the business concern does not need to know the transactionality property.

Similarly, when the same operation is projected onto the transaction management dimension, we obtain the value *Required*. Now, from transaction management implementation's perspective, the value of the same point in the business dimension does not matter -- it could be *credit*, *debit*, or anything else. As you can guess, the same is true for the authentication concern's implementation: The point's projection into the business and transaction management dimensions are unimportant.

Enabling multidimensional interfaces

Extending the notion of the multidimensional signature, annotations enable you to express multidimensional interfaces. Instead of having conventional one-dimensional interfaces expressing only the core dimension, you get multiple interfaces -- one for each concern in the application. The implementation of a concern then considers only the interface projected into the relevant dimension. Just as traditional interfaces serve well the object-oriented view of the world, the metadata-enabled ones -- *aspectual interfaces* -- serve the aspect-oriented view of the world.

AOP practitioners will recognize that the first introduction to AOP often involves explaining a similar multidimensional decomposition to the one enabled by the use of metadata. It is for this reason that metadata concepts map so well to AOP concepts. In a metadata-fortified AOP implementation, you map core concerns to classes just as you have all along. The difference is that you now map the system's crosscutting concerns to aspects that use a multidimensional interface, which is projected into the associated dimensions.

An example interface

Consider the example of the `Account` class in Listing 1. Notice how metadata annotation paves the way for the class's projection into multidimensional interfaces.

Listing 1. The `Account` class with annotated methods

```
public class Account {
    @Transactional(kind=Required)
    @WriteOperation
    @Authorization(kind="bankModification")
    public void credit(float amount) {
        ... credit operation business logic
    }

    @Transactional(kind=Required)
    @WriteOperation
    @Authorization(kind="bankModification")
    public void debit(float amount) {
        ... debit operation business logic
    }

    @Transactional(kind=None)
    @ReadOperation
    @Authorization(kind="bankQuery")
    public float getBalance() {
        ... balance query operation business logic
    }

    ...
}
```

From the business perspective a projection of this interface would map to the following:

```
public class Account {
    public void credit(float amount) {
```

```

    ... credit operation business logic
}

public void debit(float amount) {
    ... debit operation business logic
}

public float getBalance() {
    ... balance query operation business logic
}

...
}

```

The above interface simply includes all the members without any annotation. Projecting the same interface into the transaction management dimension would yield the following interface:

```

public class Account {
    @Transactional(kind=Required)
    * *.*(..) {
    }

    @Transactional(kind=Required)
    * *.*(..) {
    }

    @Transactional(kind=None)
    * *.*(..) {
    }

    ...
}

```

In this example, I've used AspectJ wildcard notation to represent methods without consideration for name, type, argument, etc. Projections into other dimensions would work similarly.

Other dimensions

As previously explained, when a business client wants to use a method it needs to understand only the projection into the business dimension(s). For example, the `transfer()` method in the `AccountServices` class could be implemented as shown in Listing 2.

Listing 2. A business client of the Account class

```

public class AccountServices {
    @Transactional(Required)
    public void transfer(Account to, Account from, float amount) {
        to.credit(amount);
        from.debit(amount);
    }
}

```

Apart from the annotation it carries (which you can ignore for the purpose of this discussion), there is nothing special about the above method's implementation. In particular, the business client -- the `transfer()` method -- does not know or care about the transactionality or authentication dimensions of the `credit()` or `debit()` methods.

Like the business client, the transaction management aspect is concerned only with its own dimension. (Note that Listing 3 is a refactoring of the aspect shown in Listing 2 of [Part 1](#) of this article.)

Listing 3. The transaction management aspect

```

public aspect TxMgmt {

```

```
public pointcut transactedOps(Transactional tx)
: execution(@Transactional * *.*(..)
  && @annotation(tx);

Object around(Transactional tx) : transactedOps(tx) {
  if(tx.value() == Required) {
    ...
  }
}
}
```

The transaction management aspect uses only the interface projected into the transaction management aspect. In other words, the aspect couldn't care less if the advised method is called `credit()` or `foo()`.

Utilizing metadata to describe aspectual interfaces is a powerful concept. Considering metadata as an aspectual interface from classes to crosscutting concerns in the system translates some of the best practices from the object-oriented world to AOP. You wouldn't, for example, name a method `creditUsingJDBC()`, because you want to describe method characteristics in the business concern space alone, and "JDBC" isn't part of that space. The same applies to aspectual interfaces: You wouldn't want an annotation type called `ReadLock` whose purpose was to describe the use of the type. A much better type would be `ReadOperation`, which simply describes the interface to the outside world.

Using metadata in AOP programming limits the coupling between classes and aspects only to the metadata attached to the program elements. When you extend this concept to metadata-enabled multidimensional interfaces, you see that each aspect is coupled to only the relevant interface. Therefore, the coupling between aspects and classes isn't different from that between classes in object-oriented programming.

Guidelines for using metadata correctly

While metadata is extremely useful when used in conjunction with AOP, there is a danger of overusing it. Overusing annotations under-utilizes the information inherently contained in the program element's signature and dynamic context. Adding metadata to your code also increases its complexity. First, program elements must carry metadata annotations. Second, incorrectly using metadata with AOP can lead to *macros on steroids*, which are very useful at first sight and save a lot of duplicated code, but make programs incomprehensible over time. It is, therefore, important to go beyond understanding the mechanics involved in combining AOP and metadata: It is also important to grasp the concepts and best practices entailed in using the two technologies together. In this section, I'll provide a series of guidelines for correctly combining metadata with AOP.

1. Don't use it if you don't need it

If the program elements of interest contain enough information in their inherent signature and dynamic context to capture the required join points, there is no reason to attach annotations. Once you attach annotations and write your pointcuts to consume them, any element that you *do not* annotate will not be able to participate in the crosscutting concerns. So, always first consider these alternatives to using metadata:

Treat global concerns globally: Program elements should not have much say in the participation of concerns such as tracing and profiling in a crosscutting implementation. For example, it makes little sense to attach `@Profiled` annotation to elements that need to be profiled. Profiled elements should be selected at the global level, and their selection depends on the current profiling goal. Using annotations in this situation would mean having to modify program elements as the system's profiling goal changed. A global aspect utilizing a signature pattern based on packages and the inheritance hierarchy would serve you better in this situation.

Utilize naming conventions: If your projects are using good naming conventions, it is best to utilize them for crosscutting purposes as well. Following better naming conventions is also valuable in itself. For example, you could use an `execution(* *.get*()) || execution(boolean *.is*())`

pointcut to capture execution of all getter methods and `execution(void *.set*(*))` for all setters. Notice how pointcuts specify types and wildcards -- getters take zero arguments and getters starting with "is" return a boolean, while setters take a single argument and return void. In some cases such a naming pattern could end up capturing wrong elements, such as a `settle()` method if it returns a void and takes a single argument (see [Resources](#) for Sam Pullara's blog on this case).

It is generally better to exclude unintended join points conforming to the expression in pointcuts, especially when you consider what would happen if you forgot to add `@Getter` and `@Setter` annotations in the above case.

Utilize component and framework API patterns: Component and framework boundaries (such as calls to or extension points of Web services components, the JDBC API, or the Hibernate API) tend to be well defined and easily picked up by a pointcut expression without using explicit metadata. Consider, for example, a situation where you want to capture calls to all listeners of any Swing components. A pointcut that uses method signature such as `call(void JComponent+.add*Listener(EventListener+)` will suffice without the need for any annotations. Such a pointcut will match unintended methods only under pathological situations. If you use metadata annotation, the chance of forgetting to mark a method with an annotation (say `@ListenerManagement`) is much higher than that of unintended methods being matched. Also consider an aspect implementation to monitor remote calls. You could simply deduce a remote method invocation by noting that the class implements `Remote` and augmenting it with exception part of method signature; something like `* Remote+.*(..) throws RemoteException`. In this case, not only would the use of metadata make you work harder (attaching an annotation to all the implementation's remote methods), but it would also leave open the possibility of a missed method.

The core message here is to avoid the temptation of using metadata as the first choice, and instead strive to distill the signature pattern of program elements and write your pointcuts without metadata. Using wildcards, inheritance hierarchy, package structure, good naming conventions, and good pointcut composition can yield acceptable pointcuts in many situations. To correctly apply these techniques you must have a good understanding of the pointcut language in the AOP system you are using, but the effort of such learning will be worthwhile.

2. Employ aspect inheritance

Ongoing discussions in AOP blogs and discussion groups show that many developers get stuck on logging and tracing examples. For these operations you can often find pointcuts that are stable across the whole system, which means you can write a single aspect for the whole system. When you try to apply the same approach to other crosscutting functionality, however, it simply does not scale. The pointcut expression, if you can write one, will be complex and often unstable with respect to the system evolution. This often leaves newcomers thinking that AOP is difficult or kludgy to implement.

The trick is to understand that if you cannot define a signature for the whole system, you need to break it up into smaller subsystems and define a pointcut for each one. There are many opportunities to decompose the system into a few parts and find a good signature pattern for each part. The earlier guidelines should help at each subsystem level.

Employing aspect inheritance is a matter of first creating an abstract-base aspect that contains most of the implementation (advice, inter-type declarations, etc.) and a few abstract pointcuts (see [Listing 4](#)). You then break the system into several parts such that you can find a good pointcut expression for each subsystem. Each subsystem can be as large as the whole system or as small as a class. This technique is at the core of the Participant pattern, which I discussed in detail in [Part 1](#). The Participant pattern lets you make the transition from simple aspects to more complex ones such as those for transaction management, thread safety, security, etc.

3. Use annotation that exists for other purposes

Aspects can utilize metadata annotation applied for various purposes. For example, the Enterprise JavaBeans (EJB) 3.0 annotations let you write pointcuts based on metadata related to transaction, security, and the role of a method (that is, `remove`, `init` etc.). Hibernate annotations let you write pointcuts based on table, column name, and relationship

information. And the Eclipse Metadata Framework (EMF) lets you write pointcuts based on the relationship between classes such as composition, association, cardinality, and so on.

4. Use abstract annotation types

If after carefully studying the situation you decide that using annotations is the best choice, how should you use them? Ideally, annotation types should represent abstract characteristics of a program element and not their expected consumption. This kind of thinking and implementation avoids the loss of obliviousness discussed below, because elements merely provide additional information about themselves and are unaware of the aspects. For example, if you consider `@ReadOperation` and `@WriteOperation` as merely extended signatures, then the class can remain oblivious to the implementation of the read-write lock pattern.

Consider the base aspect implemented in Listing 4, which follows the guideline of employing aspect inheritance discussed earlier. (See [Resources](#) if you need information on the `perthis()` aspect association: The focus here is on defining pointcuts.)

Listing 4. The `ReadWriteLockSynchronizationAspect` base aspect

```
public abstract aspect ReadWriteLockSynchronizationAspect
    perthis(readOperations() || writeOperations()) {

    public abstract pointcut readOperations();

    public abstract pointcut writeOperations();

    private ReadWriteLock lock = new ReentrantReadWriteLock(true);

    before() : readOperations() {
        lock.readLock().lock();
    }

    after() : readOperations() {
        lock.readLock().unlock();
    }

    before() : writeOperations() {
        lock.writeLock().lock();
    }

    after() : writeOperations() {
        lock.writeLock().unlock();
    }
}
```

Listing 5 shows a metadata-driven subaspect of the `ReadWriteLockSynchronizationAspect` aspect for a banking system.

Listing 5. The concurrency management subaspect

```
public aspect BankingReadWriteLockSynchronizationAspect
    extends ReadWriteLockSynchronizationAspect {

    pointcut lockManagedExecution()
        : execution(* banking.Customer.*(..))
        || execution(* banking.Account.*(..));

    public pointcut readOperations()
        : execution(@ReadOperation * *.*(..))
        && lockManagedExecution();

    public pointcut writeOperations()
        : execution(@WriteOperation * *.*(..))
        && lockManagedExecution();
}
```

Note that you wouldn't simply take locks around all `read` and `write` operations, as not all classes need such a concurrency control mechanism. It is also incorrect for classes to express the desire to have their operations managed with a lock, as the classes selected will depend on the application's usage of those classes. (Such a realization may have been responsible for making the new collection classes in Java standard libraries thread unsafe.) In the above example, the `lockManagedExecution()` pointcut selects a subset of join points that need lock management.

The power of abstract annotation types

Once you have abstract annotations describing characteristics in place, you can use the same annotations to implement other concerns. For example, you could use the `@ReadOperation` and `@WriteOperation` for dirty tracking and caching. You could even write enforcement aspects to ensure that a write operation on the same object would not be invoked while in the control-flow of a read operation!

If you use `@ReadLock` and `@WriteLock` annotation types to "tag" methods needing a lock management, the classes will be strongly coupled with lock management functionality. Further, leveraging the same annotations for other purposes could prove error-prone, because the annotations will be attached with a very specific concern in mind instead of representing the state modification characteristics of the methods.

Further examples of abstract annotation types

Let's discuss a few more examples of annotation types. Consider using `WaitCursor` as the annotation type for modularizing the wait cursor management around all methods that carry an annotation instance of that type. The code saved will be substantial (especially when you consider a try/finally block needed in each method). However, you would have lost the opportunities that a more characteristics-describing annotation type would have made possible. Consider instead, annotation type `Timing` with properties such as *mean*, *variance*, and *distribution*, as follows:

```
@Timing(mean=5, variance=0.34, distribution=Gaussian)
```

This annotation could be consumed in a couple of ways:

- Performing Rate Monotonic Analysis (RMA; see [Resources](#))
- Providing feedback to user for a long running operation (such as a wait cursor)

Of course, such a highly specific characteristic expression works only when you have quantitative timing data. Otherwise, you can use qualitative information. For example, an annotation instance might be `@Timing(value=Long)`. This scheme holds to the principle of describing the characteristics of the join point and allows for interesting possibilities such as:

- Performing selection for profiling only those methods with certain timing characteristics.
- Checking the self-consistency of timing information, such as finding methods marked with an `ExtraLong` value occurring in computational path of a method marked with a `Large` value.

Consider another example where methods are marked with the `RetryOnFailure` annotation so that an aspect can retry a failed service. If you instead use `Idempotent` as the annotation type (signifying that multiple invocations of the same operation would result in the same result), you can not only retry a failed operation, but also execute simultaneously with multiple services, if such an optimization is required.

Finally, consider the authorization and authentication example from above. The permission property may be a direct expression of the permission required. A better expression of the same concept would be a property signifying the classification of operations. Then the aspect could map classification to actual permission and it could change between systems. For example, one system might not distinguish between read and write operations from the access control perspective, while others might. An even better approach is to use business-related annotations types such as `AccountModification`, `CustomerInformationAccess`, and `Purchase`. Program elements carrying annotation of these types can be captured by not only authorization and authentication aspects, but also other implementations such as privacy policy enforcement aspects. However, creating such types right from the beginning can be a complex task. A pragmatic approach is to start with a level of abstraction that gets the job done, and then refactor to move towards a better level.

Wisdom comes with experience

If you are faced with implementing crosscutting concerns and you aren't able to follow the above guidelines, use an aspect-oriented solution anyway, even if it means "tagging" each method that needs an advice. Using AOP even with tagging at least achieves a level of modularization and gives you some freedom to modify the implementation.

The tagging approach calls for writing an aspect with metadata-driven pointcuts and then annotating program elements with the annotation type used in the pointcut. This usage is akin to using a glorified macro facility. While, the code will have a strong dependency between core elements and the aspect through annotation types, it will save substantial code (by moving the common code to an aspect or advice instead of placing it in every single method). The resulting modularization also helps to limit modifications to the concern implemented in the aspect itself.

Wisdom often comes with experience. Even if you start with the tagging approach, you will soon come to understand subtler issues and modify implementations (often with refactoring) accordingly. Remember your first few designs with OOP? Knowing what you know now, if you reevaluated your original OOP designs, you would likely find that they weren't as abstract as they could be, although the design was still better than a pure procedural approach. Similarly, your initial use of metadata with AOP may resemble a transitory period from OOP to AOP.

Metadata and obliviousness

No discussion of metadata and AOP would be complete without considering metadata's impact on obliviousness, although in practice it is modularity of concerns that really matters and not obliviousness. In the context of AOP, the *obliviousness property* requires that the core system be unaware of crosscutting functionality. For example, consider a banking system that needs to have certain operations invoked inside a transaction. The obliviousness property requires that the core banking system should be unaware that a transaction management aspect advises those operations.

Annotation can come into existence in two ways. First, you could tag elements with annotations so that a pointcut could pick the annotated elements. This is essentially client-driven augmentation, where annotations exist only to support pointcuts. Second, you could tag elements such that they always declare their non-core characteristics, regardless of the need by pointcuts. The difference between the two approaches may be just a difference in perspective, however: In the latter example elements with annotations aren't aware of the aspects (or other metadata consumers) and thus retain the obliviousness property.

Further, when you view metadata as an "aspectual interface" and follow the conventional wisdom associated with good interface design, you will automatically retain the obliviousness property. Essentially, retaining obliviousness is a matter of using the right type of annotation in the right situations, as discussed in the earlier guidelines.

Metadata and AOP adoption

Consuming metadata appropriately will be one of the interesting challenges facing Java developers in the coming years. Metadata in Java programs is a relatively new concept (even considering attempts such as XDoclet) and the new Java metadata facility will be the first systematic approach to retain metadata until runtime. Consequently, it will take practice for Java developers to fully understand and optimally use the new facility, with or without AOP.

That said, when used correctly, metadata can make AOP more accessible for newcomers and seasoned programmers alike and make widespread adoption of AOP more possible. The benefits of combining metadata with AOP are twofold: First, the inability to capture certain join points with non-core crosscutting characteristics using traditional pointcut syntaxes has long flustered early adopters of AOP. While design patterns such the Participant pattern can yield a similar effect without actually requiring metadata support, they are somewhat abstruse to use. As I've shown in this article, metadata

can go a long way toward capturing these join points to supplement crosscutting functionality.

The second adoption-related advantage of metadata with AOP is that it smoothes the adoption curve. Even in this nascent stage, metadata could serve as a kind of "training wheels" (as Bruce Tate has put it; see [Resources](#)) for learning about aspect-oriented programming. For developers writing their first AOP-style programs, the ride may yet be somewhat bumpy. Limited experience could make the guidelines discussed in this article awkward to follow, and the results could even (as previously noted) resemble glorified macros.

Sticking with the process is beneficial, however. As the new paradigm becomes more comfortable it will become more natural to explore inherent characteristics in join point signatures (along with Participant-like patterns) and use more abstract metadata. Such usage would represent a balanced use of metadata and AOP, and could also be a step toward tapping the ultimate power of AOP. Along the way, we will all develop a richer set of best practices to follow in combining these two technologies.

Conclusion

The standard metadata facility provides an easy way to express additional information about program elements. When used correctly, the facility simplifies the creation of software systems. While there are many ways to utilize metadata, it works particularly well when combined with AOP. The best practices discussed in this article are a first step toward fully leveraging metadata with AOP. In addition to the guidelines, I've explained the scenarios where AOP most benefits from being combined with metadata, discussed the support for metadata in three of the leading AOP systems, and demonstrated a systematic refactoring of an AOP system to incorporate metadata. I've also shown you a novel way to conceptualize metadata as a signature in a multidimensional concern space, a usage that has as much relevance for developers creating metadata types outside the AOP paradigm as those working in it.

AOP and metadata seem like a match made in heaven. The addition of a standard metadata facility to the Java platform and support for it in various AOP systems could very well remove the last obstacle to mainstream adoption of AOP. However, like other matches made in heaven, the two parties need to understand each other's strengths and weaknesses as well as respect each other's boundaries. I wish this couple a happy life together!

Acknowledgments

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

Resources

- *AOP@Work* is a year-long series dedicated to helping you incorporate AOP into your day-to-day Java programming. Don't miss a single article in the series. See the [complete series listing](#).
- Read the first half of this article for an [overview of combining metadata and AOP](#) (developerWorks, March 2005).
- For an introduction to the new Java metadata facility, see Brett McLaughlin's two-part article [Annotations in Tiger](#) (developerWorks, September 2004).
- Mik Kersten's two-part [AOP tools comparison](#) (developerWorks, February 2005) is a good starting point for learning about the leading AOP systems.
- Examples and discussion in this article are based on the following AOP implementations:
 - [AspectJ](#)
 - [AspectWerkz](#)

- If you're a user of the Spring framework, you might also want to investigate [Spring AOP](#).
- Learn more about the [proposed language modifications](#) to the AspectJ language, with support for Java 5.0 features including metadata.
- Learn more about JBoss AOP by reading Bill Burke and Adrian Brock's [Aspect-Oriented Programming and JBoss](#) (*OnJava.com*, May 2003).
- [Hyper/J](#) (now CME) is an AOP tool that supports the multidimensional separation of concerns.
- Also see the IBM Research listing of papers about the [Concern Manipulation Environment](#) and its forebears.
- Learn more about [Rate Monotonic Analysis \(RMA\)](#), which could be one of the primary consumers of the annotation type `Timing`.
- Sam Pullara wrote an interesting blog discussing the [unintentional capturing of join points when using wildcards](#).
- Bruce Tate discussed his concept of AOP with training wheels in his blog on [Time, wisdom and AOP](#).
- [Browse for books](#) on these and other technical topics.
- Also see the [Java technology zone tutorials page](#) for a complete listing of free Java-focused tutorials from [developerWorks](#).

About the author

Ramnivas Laddad is an author, speaker, consultant, and trainer specializing in aspect-oriented programming and J2EE. His most recent book, *AspectJ in Action: Practical aspect-oriented programming* (Manning, 2003), has been called the most useful guide to AOP/AspectJ. He has been developing complex software systems using technologies such as the Java platform, J2EE, AspectJ, UML, networking, and XML for over a decade. Ramnivas is an active member of the AspectJ user community and has been involved with aspect-oriented programming from its early form. He speaks regularly at conferences such as JavaOne, No Fluff Just Stuff, Software Development, EclipseCon, and O'Reilly OSCON. Ramnivas lives in Sunnyvale, California. You can find more about Ramnivas from his Website: <http://ramnivas.com>.