What's keeping you from trying out AOP? Whether you think it's only good for low-level functions like tracing and logging, worry that it'll get in the way of unit testing, or would simply rather stick with the object-oriented alternatives, Ramnivas Laddad gives you good reason to reconsider. Follow along as this popular author and speaker digs beneath the surface of 15 myths that hinder the adoption of AOP.

View more content in this series

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 what they know. 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 Related topics for more background on AOP.

Like any new and exciting technology, AOP has generated its fair share of buzz, and also its fair share of myths and misunderstandings. After following AOP coverage on the Web and listening to the questions asked at conferences, I started to see some common themes (or myths) that deserve to be cleared up.

The myths I address in this article are for the most part not malicious: many of them often stem from AOP advocates who are themselves confused about the technology. Nonetheless, these myths make it harder for developers to assess accurately whether or not to adopt AOP. Left alone, they will continue to cause incorrect perceptions and hinder the beneficial use of AOP.

Of the 15 myths I hope to resolve, two are common enough that even developers with no interest in AOP have probably heard them; a handful deal with various technologies or practices believed
to eliminate the need for AOP altogether; and the remainder are systemic, having to do with the incorporation of AOP into the bigger picture of application design and development. Once I've laid these myths gracefully to rest, I'll discuss ways that you can adopt AOP in a pragmatic, risk-managed fashion, so that you can safely try it out in your own system and learn from experience rather than hearsay.

I've used AspectJ as the primary AOP implementation for examples in this article, though the arguments are equally applicable to other AOP systems such as Spring and JBoss. I'll start my discussion with the most common myth.

**Myth 1: AOP is good only for tracing and logging**

**Reality: AOP is suitable for many other crosscutting concerns**

Open any AOP book or tutorial, attend any AOP talk, or read any AOP blog and you'll surely find an example of AOP used to implement logging and tracing. Because not many of these sources show more complex examples of AOP, it's commonly assumed that AOP is good just for tracing and logging.

### Why tracing?

Tracing is a poster child of AOP for a variety of reasons. First, it's a crosscutting concern whose problem is readily apparent. Second, it's particularly suitable for AOP because it is quite easy to select the required join points -- the points that need to be logged. For example, if you need to select execution of public methods, you can write a simple pointcut:

```
execution(public * *(..))
```

If you want to select all exception handlers (the try-catch blocks) for SQLException, the pointcut is no more complex:

```
handler(SQLException)
```

If you want to do more selection such as selecting calls to all remote method invocations, the pointcut is more complex, but only slightly so:

```
call(public * Remote+.*(...) throws RemoteException+)
```

These simple pointcuts make writing tracing aspects relatively effortless, and therefore are central to much introductory AOP material.

In fact, tracing and logging should be considered the "hello world" examples of AOP, which is used to implement many other crosscutting concerns. The trouble is that most crosscutting concerns aren't as easy to present at the introductory level, because of time and space restrictions. Because much of the material about AOP remains introductory, more advanced examples have yet to be widely proliferated -- but they're there!

At the system level, security, transaction management, and thread-safety concerns are commonly implemented using AOP. Many business logic problems (also known as domain-specific concerns) also exhibit crosscutting concerns upon close examination and lend themselves to modularization using aspects. On the other side, units such as classes and packages show code tangling and code scattering at a smaller scale (what I call local or micro crosscutting concerns). Aspect-oriented refactoring can help resolve these kinds of concerns by using aspects to improve the code structure. (See **Related topics** to learn more about aspect-oriented refactoring techniques.)

### Learn as you go

AOP usage follows much the same path as object-oriented programming. When you started with objects, you probably started modeling classes after domain objects -- account, customer, window,
and so on. It was only after acquiring experience (and after lots of reading) that you implemented other kind of objects such as commands, actions, and observers. The same is true for AOP. Start using AOP for commonly known concerns and, as your understanding grows, you will find aspects solving problems you weren't able to even imagine before. When working on projects using AOP, you will often find that many functionalities automatically present themselves as crosscutting functionalities and lend themselves well to implementation using aspects. Many of these aspects don't fall into any definitive classification such as security or transaction management. This is not unlike object-oriented programming, where once you start programming, classes automatically show up.

Once developers understand the spectrum of problems AOP can address, some realize that AOP actually solves the same problems they've faced for years. Funny that this should lead us directly to the next most common myth about AOP.

**Myth 2: AOP doesn't solve any new problems**

*Reality: You're right -- it doesn't!*

This is one AOP "myth" that is actually correct. AOP isn't a new computation theory that solves yet-unsolved problems. It's merely a programming technique that targets a specific problem -- modularization of crosscutting concerns.

What AOP really offers is a way to reduce the apparent complexity of a problem by reducing implementation overhead for crosscutting concerns. Developers wrestle with implementation overhead again and again in application programming; for example, when dealing with exceptions. An effective exception-handling strategy requires you to account for failure handling, chaining of exceptions, exception conversion, and so on. Further, when it comes to actually implementing your chosen strategy, you have to include the code that implements the strategy in every catch block in your system, which makes implementing the functionality much harder than it needs to be. While exception handling is an old problem, AOP handles it in a new way by reducing its complexity. Once you've chosen a strategy, you simply write an aspect implementing that strategy and you're done. All the familiar overhead of scattered implementation in the catch blocks disappears.

**Two kinds of complexity**

Resolving these first two very common myths is often enough to get developers thinking seriously about AOP. But for the more skeptical developer, it's necessary to go a step further. Because AOP doesn't, in fact, solve new problems (but solves old problems in a new way), it's natural to compare AOP to traditional techniques for implementing crosscutting concerns. The next three myths address techniques that are sometimes said to render AOP unnecessary. The question is, do any of these techniques provide the same advantages AOP does?

**Myth 3: Well-designed interfaces obviate AOP**

*Reality: Good interfaces are nice, but don't replace aspects*

AOP promises to separate classes from aspects making them largely independent of each other. This separation makes it possible to swap implementations of a module without affecting other modules in the system. You can achieve a similar separation in object-oriented programming
by creating abstract interfaces, which allow you to create various implementations of the same interface to address different functionality. Because both aspects and abstract interfaces make it possible to swap one implementation for another, some object-oriented developers have suggested that a well-designed interface eliminates the need for AOP. But a closer look at how each approach impacts on the application as a system tells a different story.

In AOP, the invocations to interface methods are modularized into an aspect, so it's easy to change the implementation by simply changing invocations in the aspect. For example, if you implemented logging using aspects and wanted to switch between log4j and the standard logging of the Java™ platform, you could do so by simply modifying the aspect to use the appropriate logging API. This would obviate the need for a common interface with two implementations delegating to the appropriate API.

But what about ...

The most common reaction to the above scenario is "But what about Commons Logging?" It's true that Commons Logging would allow you to swap the logging implementation without much change to other parts of the system. But consider that log4j was first released in 2000, whereas Commons Logging was released in 2003. How many of us were writing wrappers for log4j before Commons Logging was developed? Not me, for sure. Also, when you see a new API, how do you respond? By saying, "Let's create an abstract wrapper around it," or simply "Let's use it!"?

When the primary goal is to solve business problems, it simply isn't that productive, in most cases, to write a wrapper around all APIs (especially those that aren't central to the business problem at hand). Further, even if you could somehow justify the investment, it's not that easy to write a good interface. Any such interface must be abstract enough to allow the use of a wide variety of underlying implementations and concrete enough to allow uniform behavior of all implementations, from the client's perspective. If abstractness dominates the solution, you end up with the Least Common Denominator (LCD) problem, where the power of the underlying implementation isn't made available to API users. Conversely, if concreteness dominates the API, the result is a rigid API that is unable to accommodate many underlying implementations. All of this is further complicated by the fact that you don't know about implementations that you will encounter in the future. (If you did write a wrapper around log4j back in 2000, how easy was it for you to swap the underlying implementation with the standard logging API?)

If you're not yet convinced, consider this: There is no commonly available wrapper for Doug Lea's Concurrency library. This means that you either have to write a wrapper for it yourself, or (if you needed to switch over to the Java 5 java.util.concurrent classes) you would have to swap out every usage of that API with the new one. If you used aspects, on the other hand, you could simply change the aspects and be done with it. See Related topics for more on this.

**Myth 4: Design patterns obviate AOP**

**Reality: Design patterns can introduce complexity not found in AOP**

A second commonly proposed alternative to AOP is design patterns, which most developers are quite familiar with. Design patterns represent solutions to recurring problems where object-oriented programming doesn't offer a direct and simple solution. Design patterns definitely help improve
modularizing crosscutting concerns to some extent, but upon closer examination, it's clear that some patterns exhibit a crosscutting nature themselves. In fact, design patterns can introduce an undesirable amount of complexity as compared to alternative approaches using AOP.

Consider the chain of responsibility (COR) pattern, which is commonly posed as an object-oriented way of modularizing crosscutting concerns. The basic idea behind the COR pattern is to add a chain of handlers around the target logic. Then you can modularize crosscutting logic into a handler. This all sounds quite useful, and it is, as shown by use of the patterns in servlets filters and Axis handlers. With COR, you can easily implement performance monitoring, session management for persistence layers, encryption/compression of streams, and certain kinds of security. But such modularity is possible only because of existing infrastructure. Without such support, you would have to implement the support yourself or face lack of modularization. For example, servlet filters have only become available since Servlet Specification 2.3. Until then, everyone implemented ad-hoc solutions.

Let's look at this sizable downside a little more closely, using the COR pattern as an example.

**Did you say complexity overhead?**

Optimal use of the COR pattern requires you to have one (or a very small number of) service methods, which is why it works well with servlets and Web service processors. COR wouldn't be such a good choice if you wanted to add a chain in front of an `Account` or `Customer` class without corrupting the class design itself. In that case, you would be left without any good support for your business classes.

So what would it take to implement a COR servlets filter? A lot, it turns out. First, you would need to define the filter API, shown in Listing 1:

**Listing 1. Servlet Filter interface**

```java
public interface Filter {
    public void init(FilterConfig filterConfig);
    public void destroy();
    public void doFilter(ServletRequest request,
                         ServletResponse response,
                         FilterChain chain);
}
```

Then you would need the implementation of filter chain that executes filters before executing the target method, as shown in Listing 2:
Listing 2. Core part of filter chain implementation

```java
public class FilterChainImpl implements FilterChain {
    private int currentPos, totalFilters;
    private Filter[] filters;
    private Servlet servlet;

    ...

    public void doFilter(ServletRequest request,
                         ServletResponse response)
        throws IOException, ServletException {
        if(currentPos < totalFilters) {
            Filter currentFilter = filters[currentPos++];
            currentFilter.doFilter(request, response, this);
        }
        servlet.doService(request, response);
    }
}
```

Next, you would need some way of configuring filters. The XML snippet to add a servlets filter, shown in Listing 3, applies performance monitoring using a filter. (Listing 4 shows the filter implementation itself.)

Listing 3. Configuring a performance monitor filter

```xml
<web-app>
    ...
    <filter>
        <filter-name>monitor</filter-name>
        <filter-class>
            com.aspectivity.servlet.filter.PerformanceMonitorFilter
        </filter-class>
    </filter>
    <filter-mapping>
        <filter-name>monitor</filter-name>
        <url-pattern>/*.do</url-pattern>
    </filter-mapping>
    ...
</web-app>
```

Of course, you would also need code to parse the XML, create specified filters using reflection, and put them in the filter chain. But for brevity's sake, I won't show that code.

All of the preceding code would be required to just set up infrastructure for the COR pattern in the Servlet environment. Finally, consider the filter itself. Listing 4 is a filter that simply monitors the performance of a servlet by taking timestamps before and after executing the rest of the chain:
Listing 4. Performance monitoring filter's implementation

```java
public class PerformanceMonitorFilter implements Filter {
    public void doFilter(ServletRequest req,
                         ServletResponse res,
                         FilterChain chain)
        throws IOException, ServletException {
        long startTime = System.nanoTime();
        chain.doFilter(req, res);
        long endTime = System.nanoTime();
        monitorAgent.record(((HttpServletRequest)req).getRequestURI(),
                             endTime - startTime);
    }

    // empty implementations for init() and destroy()
}
```

Now consider the following aspect that implements the same functionality:

Listing 5. Aspect implementation for monitoring servlet performance

```java
public aspect ServletPerformanceMonitor {
    Object around(HttpRequest request)
        : execution(* HttpServlet.do*(..)) && args(request,..){
        long startTime = System.nanoTime();
        Object retValue = proceed(request);
        long endTime = System.nanoTime();
        monitorAgent.record(request.getRequestURI(),
                             endTime - startTime);
        return retValue;
    }
}
```

That's it. There is no other file in the aspect solution code (specifically, no infrastructural code is needed). While you could argue that in the COR implementation, the infrastructure code is written only once, consider that implementing patterns requires a degree of predictability. For example, while in hindsight a servlet filter is a great idea, it wasn't until Servlet Specification 2.3 that filters were supported. It's even harder to predict the future needs of your business classes.

Finally, infrastructure is usually provided for only specific purpose(s). For example, you can't use the performance filter idea to monitor RMI calls. With the aspect-oriented solution, however, it's almost trivial to write an aspect such as the one shown in Listing 6:

Listing 6. Aspect implementation for monitoring RMI calls performance

```java
public aspect RMIPerformanceMonitor {
    Object around()
        : execution(public * Remote+.*(..) throws RemoteException+)
            {long startTime = System.nanoTime();
             Object retValue = proceed(request);
             long endTime = System.nanoTime();
             monitorAgent.record(thisJoinPointStaticPart,
                                  endTime - startTime);
             return retValue;
            }
}
```
More pattern examples

While I've used the COR pattern as my example here, other design patterns commonly pitted against AOP suffer from similar problems. For example, the Observer design pattern decouples observer implementation from the subject by communicating changes through notifications, thus modularizing the observation logic. However, the pattern suffers in many ways. First, the subject infrastructure -- adding, removing, and notifying the observer -- is duplicated in all subjects. Second, notifications from the subject are fixed regardless of the observers' need, forcing it to perform event filtering itself. Aspect implementation avoids these problems by creating a reusable aspect to capture the essence of the observer protocol. You can then write a subsaspect to select the problem-specific observation logic. You can customize this aspect to your heart's content to meet your precise requirements. With COR, the requirements had to exist within the bounds set by the infrastructure; with aspects, implementation can morph easily to meet requirements. (In practice you will not copy, paste, and modify the aspect as shown, but extract a base aspect, creating an even more elegant solution and saving even more code. For a more complete implementation of this approach, see Related topics.)

With all that said, design patterns can be effective when you aren't using AOP or a problem's crosscutting nature isn't apparent. Often, problems with an existing solution surface only when something new obviates the need for that solution. It is also critical to note that not all design patterns benefit from AOP: only those with crosscutting nature in them do. See Related topics for more on design patterns and AOP.

Myth 5: Dynamic proxies obviate AOP

**Reality: AOP provides a simpler solution than dynamic proxies**

Note that there are proxy-based AOP frameworks, most notably Spring AOP. The discussion here applies mostly to direct usage of dynamic proxies and not AOP frameworks based on it. AOP frameworks using dynamic proxies exhibit characteristics closer to AOP than proxies, since they hide proxies under a framework using dynamic proxies merely as an implementation technique.

Dynamic proxies (an implementation of the Proxy design pattern) seem to be a good alternative to AOP. After all, you only need to write the `invoke()` method, which appears to mimic around advice. The strength of this approach is that it modularizes crosscutting behavior in the invocation objects. The weaknesses of the dynamic proxies solution include the following:

- Forces a complex reflection-based programming model.
- Weakens static type-checking because within the invocation handler, the objects' type is only `Object`. You can see this weakness in proxy-based AOP implementations.
- Requires creating objects through a factory (unless you replace each `new` with code to create dynamic proxies (a crosscutting concern in itself). You can also see this weakness in proxy-based AOP implementations, although it may be hidden by a framework.
- Creates the object-identity problem: The proxy and the underlying object are two different entities logically representing a single object. So should these two object be treated as the same (such as for the purpose of implementing the `equals()` method)? The answer really depends on the purpose of checking the equality and may not be right in all situations.
An aspect-oriented solution implementing identical functionality is simpler than dynamic proxies. All you need is an advice that adds the required dynamic behavior. Because no extra object is created, the factory isn't required and the object-identity problem disappears.

**Myth 6: Application frameworks obviate AOP**

*Reality: Frameworks introduce limitations not found in AOP*

Application frameworks such as EJB are commonly used to modularize crosscutting concerns. Such frameworks take a simple approach to resolving problems in application development: (1) understand commonly encountered concerns in a particular domain, (2) impose restrictions on user code to collaborate with the framework, and (3) implement the chosen concerns. For example, the EJB framework uses beans to handle persistence management, transaction management, authentication and authorization, concurrency, and so on. On the surface this seems like a fine approach, but upon closer examination it has several problems:

- Application frameworks often have a steep learning curve. You need to not only understand the basics, but also understand the framework's best practices and gotchas. Granted, AOP has a learning curve, too. However, you need to get over AOP's learning curve only once and then reuse the knowledge across a diverse set of problems. If you're reliant on frameworks, then you need to learn every framework (and every major version) anew.
- Frameworks typically offer a take-it-or-leave-it approach. If you don't like the solution offered by a framework, you have two alternatives: put up with what the framework offers or implement that particular functionality yourself, ensuring that your solution plays well with the rest of the framework. Particularly, you must satisfy any assumptions made by the framework. Pluggable choices of implementation can soften this issue, but as previously discussed, creating a useful, pluggable interface can be a complex task. In the context of frameworks, the LCD problem shows up quite often.
- You're still on your own for any concern not implemented by your framework of choice. For example, most frameworks fall short when it comes to logging, fine-grained security, and business rules. Implementing these concerns ad-hoc means you will end up with a scattered implementation.
- The frameworks approach is limited to application domains where reasonably good frameworks are available. What if there isn't a framework for your domain? For example, how do you deal with crosscutting concerns in a UI application?

The AOP-based approach takes a general view of a domain problem and provides a generic solution. In AOP, each crosscutting implementation is modularized in a separate module. You then compose the system using aspects to meet your needs. Modern lightweight frameworks (such as Spring) take a different approach. The core framework provides services such as object wiring (through dependency injection) and leaves crosscutting functionality to aspects. Of course, the framework may provide some prewritten aspects that work well in most systems. However, the use of those aspects is non-prescriptive -- you can always roll your own to meet specific requirements.
Myth 7: Annotations obviate AOP

Reality: Annotations and AOP are highly complementary

The mechanism of supplying metadata through annotations is a relatively new kid on the block. Because annotations fit with the framework paradigm, some people believe that once you have annotations, you don’t need AOP. In fact, as the AspectJ development environment has shown, AOP and annotations work especially well together. Further, annotations may not work so well separate from AOP.

Annotations are merely a way to attach additional data called metadata to programming elements. By themselves, they do not carry any semantics about the program flow around annotated elements. Tools such as annotation processing tools (APT), specialized compilers, and frameworks like Hibernate and JSF associate a meaning to the annotations, as does AOP. Unlike specialized tools, AOP offers a much simpler programming model for dealing with annotations. Unlike frameworks, AOP offers complete flexibility in associating meaning with annotations. When used correctly, annotations simplify the task of writing aspects. In certain cases, aspects also offer a great way to avoid annotation clutter (also known as annotation hell). Overall, annotations and AOP make a good pair. They do not replace each other, and in fact are complementary. See Related topics for an in-depth discussion of the relationship between AOP and annotations.

While there are many ways of dealing with crosscutting concerns, the fact is that in most instances, AOP is the more sophisticated approach. Once this has been established, the final major obstacle to AOP adoption is the notion that aspects are not easily incorporated into the overall application system or development process. These myths often cause developers to stay with existing technologies even after understanding their limitations, so I'll see what I can do to resolve them here.

Myth 8: Aspects obscure program flow

Reality: AOP raises the level of abstraction

AOP stretches the level of program abstraction more than it's ever been stretched before. AOP abstracts and modularizes crosscutting concerns into aspects -- separate from classes. Like other abstraction mechanisms, it creates an automatic problem of removing the details of a crosscutting concern's implementation from the main code. Like other mechanisms, you need to make an effort to understand the aspect's interaction with classes. Normally, however, you can understand the aspect in isolation and worry about interaction only when necessary.

AOP isn't the first programming technique that requires you give up understanding of precise program flow. When you look at a piece of object-oriented code, it's relatively easy to follow the program flow. I emphasize relatively, because you never know exactly what's going to happen. For example, if you're invoking a method on an object, you don't know exactly which method will be called because the method called will depend on the dynamic type of the object. In many cases, you will want to resort to object-oriented tooling, such as viewing type hierarchies, to determine what might happen.
Here’s an irony: The higher the abstraction of your classes, the less clear the program flow is. For example, using interfaces makes your program flow harder to understand. If you see a call being made to an object whose static type is an interface, looking at that part of the code won’t give you any idea which implementation will be invoked. The same basic problem of abstraction obscuring directness is seen in many circumstances: Cascading Stylesheets (CSS) separate the formatting but make understanding HTML formatting harder; dependency injection or inversion of control (IOC) in frameworks such as Spring abstract the configuration of services but make understanding the exact nature of services harder.

But developers still love abstraction. If anything, we strive to raise -- not lower -- the level of abstraction. This means we’re ready to give up precise knowledge about control flow to create a higher level of abstraction. Initially, all new abstraction mechanisms meet certain resistance. We’ve seen it with structured programming, object oriented programming, Web frameworks and persistence frameworks, etc. In each of these cases, what was at first considered obfuscation was eventually determined to be a feature!

**A worthwhile trade-off**

Rather than worry that it obscures program flow, you can look at the effect of AOP on program control flow another way: AOP gives you a direct mapping between requirements, design, and code elements. In other words, crosscutting requirements aren't lost in the forest of code. Separating crosscutting requirements into aspects allows you to preserve the “global picture” of your system in code. On the other hand, the “local picture” created by interaction between aspects and classes becomes implicit, and so less obvious.

You can mediate this effect by using an AOP-aware IDE such as Eclipse with the AJDT plug-in. With AJDT’s crosscutting views, you can see both the global and local picture. It’s not so easy to recover the same information in a strictly object-oriented program. If you’ve implemented your crosscutting concerns directly within classes, you’ll have to examine each class participating in the concern for the same perspective. If you’re used to not having the big picture, it’s easy to not realize what's missing, and therefore what AOP is offering. But once you get over the initial learning curve, the compelling advantages of AOP become clear. See [Related topics](#) for more information on how AJDT can help in understanding class-aspect interaction.

**Myth 9: Debugging with aspects is hard**

**Reality: AOP simplifies debugging crosscutting functionality**

The fact is debugging requires the right tools -- with or without aspects. Procedural programming feels sweet because you can understand the program flow, which maps directly to program elements. Object-oriented programs, especially those written well, require extra work to map out the flow. For example, you might see a data access object (DAO) object, but you may need to examine how that object was initialized to understand the exact implementation used (for example, JDBC-based or Hibernate-based). This may involve some detective work to see through how the reference is initialized, and then navigate upwards, including perhaps the configuration file (for example, if you use an IOC framework such as Spring).
Despite the frustration, most developers have come to understand that this abstraction is ultimately for the good -- as discussed in the previous section. Creating a layer of abstraction provides clarity and improves your understanding of what is important in a given context. And this is where a debugger comes into picture. The debugger, which understands the exact type of an object and the exact control flow, gives linear flow to the navigation between different entities.

While aspects make control flow less explicit, they also raise the level of abstraction. With the right debugger, you can then examine the precise interaction between classes and aspects. Object-oriented purists also tend to overlook the clarity that aspects can bring to debugging. Debugging tools tend to fall short when certain crosscutting functionality doesn't behave as expected. If your code is purely object-oriented, it requires a tremendous amount of effort to spot all the failure points for a scattered implementation and fix them. But with recent improvements to the Eclipse AJDT plug-in, debugging aspect-oriented programs is almost as easy as debugging object-oriented ones. Thinking this way, it isn't an overstatement to say that AOP actually simplifies debugging crosscutting functionality.

You can easily test out this myth for yourself: download the latest AJDT, set your break points wherever you want, including advice, and try debugging the code. You'll quickly see for yourself whether it's harder or easier to debug code with aspects. See Related topics for more on testing and debugging aspect-oriented programs.

**Myth 10: Aspects can break as classes evolve**

**Reality: Poorly written aspects can break as classes evolve**

This myth is correct -- to an extent. Depending on how you write your pointcuts, aspects may not work as expected (that is, they may apply where you didn't expect them to and not apply where where you did). Imagine you wrote a pointcut that selected a few methods of a class using some naming pattern and advised those methods, as shown here:

```java
pointcut accountActivity() : execution(public void Account.credit(float))
    || execution(public void Account.debit(float))
    || execution(public float Account.getBalance());
```

Now imagine if one of the methods went through a "rename method" refactoring. For example, let's say you renamed the `debit()` methods in the `Account` class to `withdraw()`. As a result, the advice to the `accountActivity()` pointcut would no longer apply to the `withdraw()` method. You could manage this issue in several different ways. First, you could try writing a more stable pointcut that used the inherent characteristics of the join points. For example, you might define the `accountActivity()` pointcut as follows:

```java
pointcut accountActivity() : execution(public void Account.*(..))
    && !execution(public String Account.toString(..));
```

Another option would be to use a tool such as AspectJ's crosscutting comparison view, which tells you how an aspect's interaction with classes has changed.
The third solution is simply to write good unit tests, which are useful with or without AOP. A good unit test would flag any mismatch immediately, simply by failing. But can aspects be unit tested? Not according to the next myth!

**Myth 11: Aspects can't be unit tested**

*Reality: Aspects can be unit tested as easily as classes*

Most developers today have come to agree that unit testing is good, so it's natural to want to apply unit testing to aspects. Because aspects crosscut many parts of the system, it isn't immediately clear how they can be unit tested, which has led some developers to believe they cannot be.

In fact, aspects can be unit tested just as easily as classes. The key to unit testing both object-oriented and aspect-oriented code is to isolate the unit being tested from the rest of the system using techniques such as mock objects. See Related topics for an article that discusses numerous techniques for unit testing aspects.

**Myth 12: AOP implementations don't require a new language**

*Reality: Every AOP implementation uses a new language*

So, okay, AOP is good and even useful. But do we a really need a new language to create AOP implementations? For developers taking a cursory look at XML- and annotation-based AOP implementations such as Spring AOP, JBoss AOP, or AspectWerkz, the answer is typically "no." At a high level, it seems that these implementations provide AOP support without requiring a new language.

In fact, every AOP implementation uses a new language and the only difference is the flavor of the language used. While traditional AspectJ syntax uses direct extensions to the base language, other implementations use an XML- or annotation-based language. Consider Table 1, where you can see pointcut definition as written by various AOP implementations.

**Table 1. Pointcut definition in AOP implementations**

<table>
<thead>
<tr>
<th>Style</th>
<th>AOP implementation</th>
<th>Pointcut</th>
</tr>
</thead>
<tbody>
<tr>
<td>Language extension</td>
<td>AspectJ</td>
<td><code>pointcut logOp() : execution(* Account.*(..));</code></td>
</tr>
</tbody>
</table>
| Annotation-based             | AspectWerkz (using Javadoc)             | `/** *

  * @Expression execution(* Account.*(..)) */

  public Pointcut logOp;` |
| Annotation-based             | AspectJ (using Java 5 annotations)      | `@Pointcut("execution(* Account.*(..))")

  public void logOp() {}` |
| XML-based                    | Spring 2.0 (using AspectJ pointcut language) | `<aop:pointcut id="logOp" expression="execution(* Account.*(..));"/>` |
| XML-based                    | JBoss AOP                               | `<pointcut name="logOp" expr="execution(* Account.*(..))"/>` |
A quick glance reveals that the difference in pointcut language between multiple AOP implementations is rather minor. The same is true for other constructs, such as pointcut composition and advice. Essentially speaking, there is no escaping the fact that AOP constructs need to be expressed somehow. You do get to choose the flavor you use to express those construct, however.

**Myth 13: AOP is just too complex**

**Reality: Yes, but still simpler than the alternatives!**

When you look at a pointcut like the one below, it's natural to be daunted by the complexity of the syntax (especially without much background in pointcut syntax, or without any guidance of someone more experienced):

```
cflow(execution(* HttpServlet.do*(..)))
&& call
(public * Remote+.*(..) throws RemoteException+)
&& target(remoteService)
```

**Pointcut composition**

The pointcut discussed in this myth is actually a good example of how not to write a pointcut. A better approach to pointcut composition is to write simpler and semantically meaningful pointcuts and then compose them to form higher level ones. You can see this technique at work as I rewrite the pointcut. First, I write a pointcut that selects all operations occurring in the control flow of any HttpServlet method starting in do, such as doService() and doGet():

```
pointcut duringServletRequest() :
  cflow(execution(* HttpServlet.do*(..));
```

Next, I write a pointcut that selects all RMI calls and collects the object on which the call was invoked (also called the target object):

```
pointcut remoteCall(Remote remoteService) :
  call(public * Remote+.*(..) throws RemoteException+)
  && target(remoteService);
```

Finally, I advise the composed pointcut:

```
before(Remote remoteService) :
  remoteCall(remoteService)
  && duringServletRequest() {
    ...
  }
```

Writing pointcuts this way makes them much simpler and easier to understand. It also leads to smaller pointcuts that you can reuse for other advice (in this case the duringServletRequest() and remoteCall() pointcuts).
If you take just a few hours to study the pointcut's syntax, however, you will be comfortable with pointcut syntax generally. When I give an introductory talk on AOP, I introduce the basic syntax for a pointcut, and then I ask attendees to unravel the syntax of more complex pointcuts. I almost always get the right answers, suggesting that pointcut syntax is logical and predictable -- two virtues of a good language. While I have given an example in AspectJ, the same holds true for other AOP implementations such as Spring AOP and JBoss AOP.

AOP isn't trivial, but it isn't rocket science either. One reason AOP feels complex is simply because it addresses complex problems. Because these problems are complex, so are the solutions. What's important to recognize is that AOP does offer a different way of thinking about problem solving, as discussed in the next two myths.

Myth 14: AOP promotes sloppy design

*Reality: It takes experience to use any technology optimally*

It is true that sometimes the AOP hammer makes every problem look like a nail, and certain usages of AOP could be best resolved by better design. For some developers (especially those newer to AOP), AOP might seem like a magic wand that can easily resolve sloppy code or bad design. It could be tempting to "patch" a system with aspects rather than fixing design flaws.

The truth is that it takes experience to use any technology optimally. It's possible to write sloppy programs with AOP just as it is possible in any programming methodology. No one learned object-oriented programming in a day! (Or months, years, or decades.) Making mistakes is part of the game. Consider the many object-oriented developers who only learned to not overuse inheritance by doing just that and then had to deal with all the problems that came with it.

Should you wait until all the usage patterns have been established before using any new technology? Or would you rather use history as a guide and proceed in a pragmatic manner? The latter approach lets you learn and grow with the technology. You benefit from it with very little risk. What do you have to lose?

Myth 15: AOP adoption is all or nothing

*Reality: AOP can be adopted incrementally*

Contrary to popular belief, it's not necessary to take an all-or-nothing approach to AOP. Many aspects pave the way for incremental adoption. For example, you could (and I recommend that you do) start with tracing aspects. These aspects can help you with finding problems during development. They can be invaluable in complex scenarios such as figuring out multithreaded issues that have slim chances of discovery using debugging and similar techniques. You can improve the likelihood of reproducing bugs and fixing them by using a trace-enabled version of a product during QA testing. You then could choose to omit these aspects in production by simply not including them in your build system.

A second kind of starter aspect involves policy enforcement using aspects. Here you use a combination of compile-time declarations (in AspectJ) and runtime advice to detect violation of
policies in your system. Policy detection can be left in the system during development and QA phases. A good example of this kind of aspect is detecting violation of Swing thread safety rule, where only the event dispatch thread is allowed to access or alter the state of a UI component. Violations of this policy can result in symptoms ranging from simple UI anomalies to outright crashes. Without aspects, such violations can be very difficult to detect and fix. With a short aspect you can detect violations, capture enough information to allow an easy fix, and produce a better quality system. (See Related topics for more on this.)

Another kind of starter aspect is the testing aspect. Here you can inject faults to test "sad paths," where a system goes through a problem area such as network and database failures. Testing for these conditions is difficult, but no less important. With aspects, you can inject faults into the system. This can also help in improving the code coverage and boosting confidence that you have a solid product.

All of these aspects offer a pluggability feature that does not force you to include aspects in production (and not even require using a special compiler or weaver). These aspects also help you play with aspects and understand the the AOP paradigm, which paves the way for fully leveraging the power if AOP.

**In conclusion**

In this article, I've attempted to clear up the most common myths and misunderstandings that hinder the adoption of AOP. While I've discussed the reality behind each myth from an aspect-oriented-programming perspective, the only way to find out whether AOP is a good addition to your development practice is to try it for yourself. I suggest that you set aside one full day to experiment with AOP -- not by reading articles, books, or blogs, but by coding aspects for yourself. Doing that will let you form your own opinion of AOP, which ultimately is what really matters.

**Acknowledgments**

I will like to thank Ron Bodkin, Adrian Colyer, Mik Kersten, Gregor Kiczales, Nick Lesiecki, and Mira Mezini for reviewing this article and providing valuable feedback.
Related topics

- "Aspect-oriented refactoring" (Ramnivas Laddad, ServerSide.com, December 2003): A two-part introduction to aspect-oriented refactoring techniques. Also demonstrates implementation swapping using aspects, based on the concurrency-control example used here.
- AspectJ in Action (Ramnivas Laddad, Manning 2003): Learn more about policy enforcement using aspects and a wide range of problems that can be addressed using AOP.
- The AspectJ home page: Download the latest AspectJ compiler and tools.

© Copyright IBM Corporation 2006
Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)