Applying AspectJ to J2EE Application Development

Nicholas Lesiecki
VMS
5151 E. Broadway, Ste. 155
Tucson, AZ 85711
520 202-3119
ndlesiecki /a/t/ apache /d/o/t/ org

ABSTRACT

This report focuses on the application of AspectJ to the development of a J2EE web application for Video Monitoring Services of America (VMS). Aspects were used to cleanly modularize concerns ranging in scope from auxiliary (error-handling) to application-specific (shopping basket price calculation) to framework-level (object relationship management). VMS saw benefits resulting from the aspect-oriented implementation of these concerns in the areas of code size, understandability, and reduced defects. This report will detail specific areas to which AspectJ was applied, the development team's reaction to the new technology, strategies employed to ease adoption, and some of the pitfalls encountered when using the development tools.

Keywords

Aspect-orientation, AspectJ, XP, Extreme Programming, -ilities

1. INTRODUCTION

In January 2004, Video Monitoring Services of America (VMS) began to investigate the adoption of AspectJ into the development of their J2EE-based application, Adbase. Adbase supplies a web-based search and ecommerce interface on VMS's library of advertising data. In many ways, it might be regarded as typical of J2EE application development. It uses a wide range of J2EE technologies, both commercial (e.g. Weblogic's EJB and Servlet implementations) and open source (e.g. Hibernate and Tapestry). The Adbase team varied from four to seven full-time developers during the time frame covered by this report. We follow a modified version of Extreme Programming (XP) with a focus on emergent design and adherence to the code-level practices of XP (such as programmer testing and pair programming). As Principal Engineer of the project and coauthor of *Mastering AspectJ* [1], I championed the investigation and subsequent adoption of AOP.

This report will first discuss examples of concerns successfully modularized with AspectJ, and discuss some of the benefits observed as a result. The first example covers a common auxiliary concern handled with aspects: improved error logging. In the second example, the report describes an application-specific aspect typical of Adbase: dynamic repricing of a shopping basket after a change to its contents. The final example covers VMS's most sophisticated aspects: those that manage bidirectional relationships between persistent objects.

The rest of the report will focus on the issues surrounding the adoption of AOP into the team's development process.

Because AOP allows modules to affect the behavior of other modules transparently, two perceived risks of AOP adoption are loss of comprehensibility and unexpected side effects. We managed these risks by drawing on the strengths of our established development processes: testing and pair programming.

A key challenge to our adoption has been tool support. We spend most of our development time incrementally changing code, compiling, testing, and changing again. The long compile times typical of AspectJ development disrupt this cycle and lower hour-to-hour productivity. Furthermore, AspectJ IDE support for concerns such as refactoring lags behind pure Java support.

Although not a major concern, the Adbase team did encounter some incompatibilities between other tools and AspectJ. This report documents those problems.

Despite these caveats, the Adbase team in general regards AspectJ as successfully integrated into both the application and our development process.

2. APPLICATIONS

2.1 Error Logging

Our investigation of AspectJ began with a typical "first use". Our application had been plagued by poor exception handling in a third party library. Our problem was code of the following form:

```
public void evaluate(String arg) throws
JspException{
  try{
    useReflectionToCallIntoApplication(arg);
  }catch(Exception e){
    throw new JspException( e.getMessage() );
  }
}
```

(Note that this is pseudocode.) This type of handler block discards the stack trace from the underlying exception, leaving application developers mystified about the original cause.

The Adbase team decided to use the bytecode-weaving capabilities of AspectJ to advise handler blocks of this type. Because the offending library was open source we could have patched the library to correct the offending handlers. However, we would then face a choice between submitting the patch and ensuring our code was general enough to serve all clients of the library and maintaining our modifications as new versions of the library were released.

To improve the error behavior we crafted a pointcut to identify handler blocks in an area of the code that we knew from experience was troublesome. Then we added before advice to log the exception "argument" to the handler block. Writing an ant script to automate the weaving of the library took less than an hour.

Although this implementation fulfilled our needs, it did bring us into contact with one of the implementation limitations in the ajc compiler. The end of a handler block is indeterminate in bytecode. Because of this, ajc does not allow after or around advice to be applied to handler join points. [2] This limitation stymied several attempts to make the aspect more sophisticated (techniques were possible with AspectJ 1.0). For instance, if handler join points supported after advice, it would be easy to hold a reference to the original exception and subsequently attach it (using Java 1.4 exception-chaining) to a newly thrown exception. Despite this limitation, the team has since revisited the error-logging aspect and added similar behavior by using percflow aspects and heuristics about the likely source of an exception.

2.1.1 *Analysis of Impact*

The net effect of our first aspect was positive. Errors that formerly required a trip to the debugger or careful analysis of the source code were solved in a matter of moments once we saw stack trace information in the application log. The exposure to AspectJ in this limited context encouraged the team to explore further.

2.2 Application-specific aspects

After the successful investigation above, the Adbase team experimented with other auxiliary aspects (e.g. for gathering quick-and-dirty performance statistics). Finally, after formal presentations to the team about AOP and AspectJ, we agreed to add AspectJ to our development process with a pilot aspect. The aspect was deliberately limited in its scope, because we wanted the ability to reincorporate a traditional implementation of the feature should the experiment prove unsuccessful. This aspect managed the qualification of new data by an administrator, and also the corresponding suspension of automated processing of that data. Although we encountered some challenges, the pilot was a success and the aspect became the first of many limited-scope, application-specific aspects that we wrote in the following months. While aspects are frequently touted for their utility at modularizing widely crosscutting concerns (such as security or transaction management) we have derived significant benefit from using them in areas where only a few classes or join points are affected.

2.2.1 *Basket Repricing*

A good example of such an aspect is the one we wrote to handle the dynamic repricing of an online shopping basket in response to changes on it's content or to its delivery options. Basket pricing in Adbase is complex. Different charges apply to items, groups of items, and the order as a whole. Some charges depend on the presence of other charges. A change to a setting in one part of the basket can require the recalculation of the price of items in another part. Since so many operations and objects can impact the chargeable state of the basket, we decided to that repricing was a clear crosscutting concern.

The aspect we chose to implement operates by tracking state changes with a dirty flag. Any operation that can affect the price of the basket sets the flag. Then, before a client reads pricing information from the basket, the aspect reprices the basket and clears the flag.

The aspect selects dirtying operations with pointcuts like the following:

```
pointcut basketChange(ShoppingBasket basket):
  (execution(public  void
             addItem(ShoppingBasketItem, ..))
   || execution(public void remove*(..))
  ) && this(basket);


pointcut groupChange(ShoppingBasketGroup group):
  (execution(public void setDeliveryMedium(..))
   || execution(public void setCopies(..))
  ) && this(group);
```

Then it uses an inter-type field and advice to mark the basket dirty after each state change:

```
private boolean ShoppingBasket.isDirty = false;
after(ShoppingBasket basket) returning :
  basketChange(basket)
{
  basket.isDirty = true;
}
```

Next, it selects operations that represent reads of the basket's pricing state:

```
pointcut chargeableReads() :
  (execution(
     public * Chargeable+.getCharges(..))
   || execution(
     public * Chargeable+.getTotalCharge (..))
   ||execution(
     public * Chargeable+.hasCharges(..))
  ) && !cflow(execution(public * Pricer.*(..)));
```

Each of the components of the cart that can receive charges implements the Chargeable interface. (Chargeable is actually what we term an "AspectJ Interface," an interface with concrete behavior supplied by inter-type declarations.) This interface makes it easy to pick out reads across three different classes. Note that the pointcut excludes reads that happen during the flow of execution of methods in the Pricer object. This prevents a recursive call to the advice during repricing.

Finally, the aspect uses this pointcut with before advice to reprice the whole basket before each read operation:

```
before(ShoppingBasket basket) :
  chargeableReads() && this(basket)
{
  priceAndClean(basket);
}


private void priceAndClean(ShoppingBasket
basket) {
  if(basket.isDirty){
    pricer.price(basket);
    basket.isDirty = false;
  }
}
```

We validated the aspect's effects with an integration test that modified the ShoppingBasket and its contents, and then checked that prices arrived at expected amounts after each change. This test gave us the confidence to refine and refactor the aspect to arrive at the optimal design.

2.2.2 *Analysis of Impact*

I would list this aspect as one of the clearest successes of application-specific aspects in Adbase. Because of it, and the aspect that implements Chargeable, ShoppingBasket and its associated items contain almost no code related to pricing. This allows them to better represent their primary role (a structured collection of items for purchase). It also allows developers to reason about and evolve the pricing/repricing behavior in isolation, without having to inspect and/or modify the various price-triggering events.

By separating pricing logic, the aspect solution added *extensibility* and *composability* to the basket package. (Alternative pricing strategies could be swapped in with fewer changes to the code.) Further, we could tell that the codebase's *agility* had increased. Because price refresh strategies that touched a dozen or more operations could be implemented easily in a single location, we felt free to experiment with alternative designs.

Another benefit of using aspects for repricing logic became apparent on inspection of the code in preparation for publication. The sophistication of the cflow pointcut allows the expression of complex conditions for the execution of code. Excluding reads that happen during pricing would have been an awkward task without the use of !cflow(execution(public * Pricer.*(..))).

2.2.3 *Thoughts on Obliviousness and Evolution*

Reduction in tangling represents a key value proposition for AOP. By removing non-core behavior from a class to an aspect, one can free developers using the class from having to think about the removed, non-core behavior. However some problems qualify this promise of obliviousness. We encountered a few while developing the ShoppingBasket.

First, as developers added new operations that required repricing, they had to modify the aspect to cover these new join points. In an ideal world, as a module evolved, all aspects that affected it would display robustness: their pointcuts would match new operations automatically and track refactorings to existing operations. However, AspectJ is not currently ideal, and it may never be completely ideal in this sense. (See section 5.4 for thoughts on refactoring support in existing tools.)

We could have gained some ground in this area without changing the tools or the language. Some simple refactorings, such as using execution(void ShoppingBasket+.remove*(..)) instead of enumerating specific methods, could have increased the robustness of the aspect if they had been applied sooner. Best practices for writing robust pointcuts are evolving [3] and it's clear that robust pointcut authorship will represent a core competency for aspect-oriented developers.

Also, it became clear that, although not everyone on the team needed to know how, say, the pricing aspect worked, they did need enough tool support (and awareness) to quickly investigate whether and how an aspect might be affect a given piece of code.

Both of these concerns promise to be ameliorated by the addition of metadata support in AspectJ 5.0. The ability of pointcuts to match on annotations would have allowed pointcuts like:

```
pointcut pricingChange():
  (execution(@AffectsPricing * *(..)));
```

Annotations such as `@AffectsPricing` could be useful when there is no inherent common property of the affected operations for a pointcut to use. Furthermore, the presence of the annotation on some methods in a class could cue programmers to add the same annotation to new methods that changed pricing information. However, annotations do not represent a silver bullet. Ramnivas Laddad's article on AOP and metadata [7] explores this space more thoroughly than is possible here.

2.3 Reusable aspect libraries (Relationship management)

The most advanced set of aspects developed by the Adbase team came in response to a widespread crosscutting concern. At the time we wrote the aspects, we were transitioning to a new persistence solution. We had abandoned EJB's Container Managed Persistence in favor of the lighter-weight Hibernate. Unfortunately, during the changeover, we discovered that Hibernate lacked the container-managed relationships that we had begun to take for granted with CMP. The thought of having to refactor our code to manually implement this concern made us willing to invest some time in an aspect-oriented solution.

The feature that was missing from Hibernate is what we termed "bidirectional relationship propagation." Figure 1 shows a model of two persistent objects in a typical relationship.

If the model in Figure 1 were implemented in pure Java code, calling 'child.setParent(someParent)' on the child would not have any particular effect on someParent's children collection. In an EJB container, if the container managed the relationship, calling 'child.setParent(someParent)' would result in the equivalent of a call to 'someParent.getChildren().add(child)'. The container would propagate the link to the other side of the relationship. This would make the relationship navigable in both directions. While Hibernate does this as it reads the objects from or writes them to the database, it does not propagate the relationship during the object's general use by an application.

The core of our solution was simple, but a complete implementation that adequately addressed all of the dimensions of the problem required several iterations of development. During the implementation of the solution, we relied on our suite of integration tests (developed while we

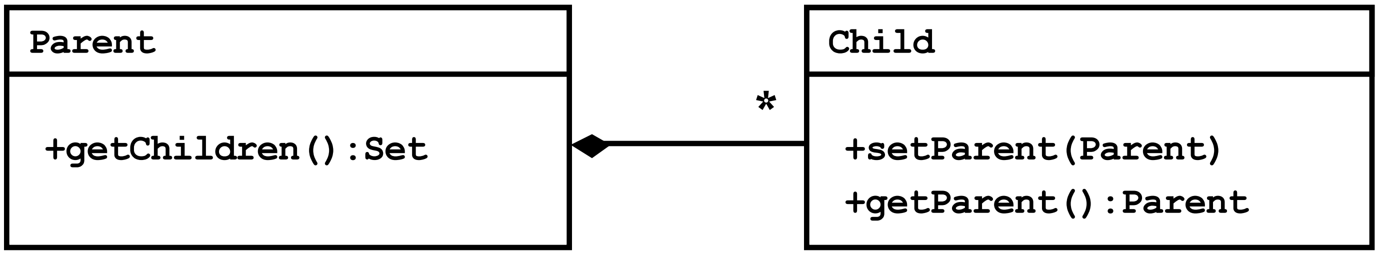

Figure 1. Model of persistent objects in a relationship

were relying on Weblogic's relationship management) to assure us that we had replicated the parts of the behavior that our application depended on.

2.3.1 Alternatives

Using aspects to address a weakness in or to extend an existing application or framework raises questions: Is there another way of using the framework that provides the same result? Would it be a better idea to modify the framework directly? In our case, we discerned from Hibernate's documentation that the preferred solution to our problem was to write code in each relationship setter to traverse and propagate the relationship. This was exactly what we hoped to avoid. Second, extending or patching Hibernate presented problems. It was our persistent objects whose behavior needed modification (Hibernate already "did the right thing" when loading the objects). A future version of Hibernate or an alternate persistence framework might have held out hope, but had already invested in Hibernate. We decided that if the aspect solution could be tested and deployed in under two weeks, we would stick with it.

2.3.2 Updating Parent's Children

We began our implementation with one half of the desired behavior. In order to automatically add a child object to a parent's collection after a call to 'setParent()', we first had to identify the "setParent" join points. To do so, we wrote the pointcut in Figure 2. The pointcut explicitly assumes that a method call signifies the start of a relationship if it meets these three criteria: 1) it begins with the letters "set" 2) the target of the method call implements our application's Persistent interface 3) the sole argument is also a Persistent. The pointcut exposes both the target and the argument object for use within the advice. Because the pointcut relies on an identifier pattern, we accepted the risk that the pointcut might match join points incorrectly (e.g. Account.settle(Account)). We judged this risk to be acceptably low because of the limiting effect of the other join point criteria.

The advice in Figure 2 extracts the name of the setter method using the implicit thisJoinPointStaticPart object. The advice then delegates to a static helper class that navigates the metadata provided by Hibernate. The helper class retrieves the "children" collection and adds the child object (referenced in Figure 2 by *self*) to that collection. The logic was localized in a helper class because it seemed beneficial to split the definition of the crosscutting structure into one module (the aspect) while allowing the details of Hibernate metadata traversal to reside in another. This allowed for easier testing of the corner cases of metadata traversal since the helper class could easily be fed the appropriate arguments. This decision reduced compilation time. Because a change to an aspect requires a full build (see section 5.3), splitting the implementation allowed us to change the traversal logic without giving up incremental compilation.

2.3.3 Updating Child's Parent

Of course, this aspect only solved half of the core issue. We also needed to detect and propagate relationships that occurred through modification of a parent's children collection. In order to detect these join points, we needed some way of identifying a children collection returned by a parent.

```
public aspect RelationshipManagement {
      //...
      pointcut setToOne(Persistent self, Persistent other) :
            call(void set*(Persistent+)) && target(self) && args(other) &&
            managedRelationshipAccess(); //restricts scope of pointcut

      before(Persistent self, Persistent other) : setToOne(self, other) {
            String methodName =
                  thisJoinPointStaticPart.getSignature().getName();

            BidirectionalRelationHelper.reciprocateToOne(
                  StringUtil.methodNameToPropertyName(methodName), self, other);
      }
}
```

Figure 2. A pointcut and advice that implement part of the bidirectional relationship behavior

We chose to do this by wrapping the returned collection in a decorator that delegated its methods to the original collection. This gave us a convenient way of writing pointcuts that detected modifying operations on the returned set. Again, we select relationship getters with a pointcut based on a combination of name and type information. The aspect then uses around advice to modify the return value of the affected join points (the original collection is obtained through a call to proceed()). The after returning advice shown in Figure 3 typifies the decoration logic. The advice inspects the return value of add(). If the addition was successful, it uses the helper class to propagate the relationship.

2.3.4 Severing Relationships After Deletion

After we implemented the aspects described in the previous two paragraphs, we encountered a set of test failures. If the application deleted a child object, Hibernate expected the application to remove it from the parent's collection. So we added another dimension to our solution. To detect deletion events, we wrote a pointcut that made use of a marker interface supplied by Hibernate. By writing the pointcut in terms of the Lifecycle.onDelete() method, we leveraged Hibernate's definition of a deletion event. After each such event, our aspect delegated to the helper class to iterate over all of the doomed object's persistent relationships and remove the dying object from them.

This simple solution hit upon a snag, however. Hibernate, like other ORM frameworks, manages cascade deletions. If our severing aspect affected a relationship in the middle of a cascade deletion, Hibernate would detect the condition and throw an exception. In order to exclude cascade deletions, we used a percflow aspect to retain state for the entire control flow of an object deletion. The aspect stored a set of already-deleted objects and passed this set to the helper class. The helper would then decline to sever a relationship to an object that had already been deleted. The code for this aspect appears in Figure 4.

2.3.5 Analysis of Impact

To gauge the impact of this aspect on the Adbase project, I gathered some metrics with the help of the crosscutting structure viewer. Hibernate persists sixty-three classes in the project. Those classes contain 42 methods of the form child.setParent() and 39 of the form parent.getChildren(). In addition, there are 11 relationships managed through primary key components (another dimension to our solution not covered here). These metrics give a sense of the impact of the aspects. Implementing the solution by manually inserting a call to the relationship helper would have involved modifications to 92 disparate sites. It's not hard to imagine the team missing 5 or 10 sites and stumbling across a bug later on. Thus the aspects could thus be said to improve the *maintainability* of the system.

Without the tangled calls to the bidirectional relationship helper, the persistent objects are easier to read and understand. (Thus the aspects add *understandability*.) It's possible for a developer to program Hibernate-enabled objects that participate in bidirectional, automatically severed relationships with no special effort. Of course, this benefit is most useful when programmers are sufficiently aware of the aspect to be unsurprised by the additional behavior. Because these (and other Hibernate aspects) are so pervasive, they are frequent causes of the "aspects as a red herring" situation described in section 3.2.

2.3.6 Reusable Aspect Libraries

Because the Hibernate relationship management aspects apply to objects that implement the Persistent interface, it would be easy for VMS to turn the aspects into a reusable library. This step has been planned for internal use, and VMS has also considered donating the aspects to open source.

2.4 Overall Analysis of Impact

VMS does not use a formal code review process. Instead we rely on as-needed flagging and review as pairs encounter code

```
pointcut persistentSetAccess(Persistent self) :
      call(java.util.Set Persistent+.get*())
      && target(self) && managedRelationshipAccess();


java.util.Set around(Persistent self) : persistentSetAccess(self) {
      String methodName = thisJoinPointStaticPart.getSignature().getName();
      String propertyName = StringUtil.methodNameToPropertyName(methodName);
      return new RelationshipAwareSet(propertyName, self, proceed(self));
}

//pointcut relationshipAdd() not shown
after(RelationshipCollectionDecorator self, Object o)
      returning(boolean added) : relationshipAdd(self, o) {

      //'source' is a reference to the originating object (stored by the decorator)
      if(added){
            BidirectionalRelationshipHelper
            .reciprocateToMany(self.propertyName,
                              self.source, (Persistent)o, true);
```

```
public aspect RelationshipSevering percflow(call(* Session+.delete(..))) {
      private Set otherObjectsBeingDeleted = new HashSet();

      pointcut onDelete(Persistent objectBeingDeleted) :
          execution(public boolean Lifecycle+.onDelete(Session+))
          && this(objectBeingDeleted);

      after(Persistent objectBeingDeleted) : onDelete(objectBeingDeleted) {
            BidirectionalRelationHelper.
                  severRelationships(otherObjectsBeingDeleted, objectBeingDeleted);
            otherObjectsBeingDeleted.add(objectBeingDeleted);
      }
}
```

Figure 4. Severing relationships on object deletion

that needs improvement. A more formal process could help publicize the successes of AOP to other teams in the company. It could also invite challenges to given use scenarios. Any new technology risks becoming a "golden hammer" if overused. Although I do not feel that VMS has succumbed to this temptation, skeptical review could provide an important safeguard.

Despite the lack of a detailed review, section two has attempted to provide for each example the results of our informal evaluations. In general we have found that aspects have improved various ilities such as understandability, maintainability, reusability, agility, composability, and extensibility.

3. TEAM REACTIONS TO AOP

3.1 Positive

Just to be clear, as the author of this practitioner report, I am biased. I had been working with AspectJ and AOP for a year and a half before joining the Adbase team, and had co-authored the second book on the language [1]. I could be considered an extremely early adopter. With that in mind, I was pleasantly surprised by the reaction of other, more mainstream, members of the Adbase team.

As a whole, the team's reaction to AOP and AspectJ was positive. Even programmers inexperienced with OOP were able to grasp the basic concepts quickly. Like learning any new language, mastering the nuances of AspectJ took time. However, I found that other programmers moved quickly from being passive observers to participants in or initiators of pairing sessions that involved AspectJ. About 6 months elapsed between the VMS's initial investigation of AOP and the development of the Hibernate aspects described in this report. This was calendar time, not effort time, as we spent (and still spend) only about 5 to 10% of our development effort on AOP. Because of our incremental approach to adoption (described in section 4) we were able to both learn AOP and realize value from it without significant schedule impacts.

As familiarity with the technology has grown, the team has become more excited about the potential for further applications. It's not uncommon to hear "we could use an aspect to do that" floating through the development area. Some members of the team have become champions of the technology. One programmer is investigating adding AspectJ to another Java project, and our manager acquired a custom license plate reading "AspectJ."

For programmers less invested in AOP, its non-invasive nature has been a blessing. Developers can craft components that participate in aspect-oriented behavior without needing to make numerous design concessions to do so. It's possible to work on the Adbase project without significant AspectJ experience—indeed some programmers have yet to write a single pointcut. Compared to invasive frameworks such as EJB or JSP, AspectJ has weighed lightly on the design and development of the project.

3.2 Negative/Challenging

Despite the generally positive reaction, AOP has received some challenges at VMS. First, I would observe that developers (and perhaps humans in general) regard new technologies and ideas with suspicion. The more radically an idea departs from accepted norms, the more a population will resist it. Because AOP demands a new way of thinking about software construction and a new language to express that thinking in, it necessarily meets a fair amount of this resistance. My team has been no exception. During initial presentations, it was clear that the audience saw the potential of the technology, but was apprehensive about the learning required to leverage it. As time passed, the apprehension dissipated. However, I found it present again in new team members who were approaching the technology for the first time. This points to a difficulty with adopting any non-mainstream technology. The less widely known it is, the more a project can expect to invest in training as staff changes.

Though our familiarity with AspectJ and AOP has grown, we are still in the process of altering our mental models to accommodate aspect-orientation. The Adbase team manager, Scott Segal put it like this: "The most difficult part of adopting AOP was making the mental shift from thinking of AOP as a technique to solve a narrow set of problems to embracing AOP as part of our *standard* approach to solving all problems. When looking at a problem now we try to see how the problem naturally separates into concerns and implement the crosscutting concerns with aspects. " OO enjoys a wealth of material from practitioners, gurus, and evangelists that helps programmers to adopt an OO mindset. As the AOP community is still in its childhood, embracing AOP requires a more conscious effort on the part of adopters.

3.2.1 Effects on System Comprehension

Although we have found that it is possible for programmers to remain mostly oblivious to aspects (especially very orthogonal ones), to work effectively on an aspect-oriented project, they must be aware of the potential that an aspect is affecting code they are modifying. Without this awareness (and knowledge of how AspectJ works in general) it's possible for programmers to become confused or surprised by aspectual effects. The less orthogonal the concern, the more programmers must be aware of it.

A particular flavor of this problem might be termed "aspects as a red herring." This situation has arisen for us when programmers uninvolved with the development of an aspect encounter a problem in that aspect's domain. Suspicious that the aspect may be involved, but lacking knowledge of its details, they feel overwhelmed by having to understand the aspect before solving the problem. The aspect may not actually be involved, but its possible presence can seem like a confounding factor.

In order to reduce the possibility of aspect confusion, one must build sufficient familiarity with both AOP mechanisms in general and the application's aspects specifically. Developers working on an EJB project would be expected to at least be familiar with declarative transactions. Aspects, it would appear, are no different.

Awareness of aspectual effects can be enhanced by appropriate tool support. Section 5 discusses how specific tools can increase the visibility of aspects.

4. ADOPTION PROCESS

We knew at the outset that adopting a technology as conceptually challenging as AspectJ would require more effort and care than simply picking up a new library. This section details the specific steps we took to minimize risk, and which of our existing processes supported the adoption.

4.1 Background

At the time VMS's investigation into AOP began (February 2004), none of the Adbase team (other than myself) had any significant exposure to AOP. Although the industry buzz around AOP grew steadily throughout the year, at that time it had not reached critical mass. Other AOP implementations were available, but two of the main AOP technologies today (AspectWerkz and JBoss AOP) had not emerged from beta. AspectJ was both the most proven project. We also had an in-house expert. Accordingly, VMS decided to focus its efforts on AspectJ.

4.2 Incremental Adoption

In order to minimize the impact of adoption, we decided to approach in phases. We undertook two significant experiments: the error-logging aspect described in section 1.1 of this report, and then the application-specific aspect described in section 1.2. The second experiment necessitated the integration of AspectJ with the existing Adbase build, which ultimately required us to restructure our build process. The experiments exposed the team to the technology in a low risk way. This increased familiarity and confidence. We also retained the assurance that we could easily revert to a pure Java version of the system with minimal effort should we need to.

After the experiments, we began to look for other application-level concerns that we could implement with aspects. Because these new aspects tended to be limited in scope, we were able to learn concepts, best practices, places where aspects made sense, and places where they didn't. (We have replaced more than one aspect with a non-aspect solution upon review of the resulting design.) With these experiences under our belt, we were finally able to approach the relationship management aspects—the most challenging and general set of aspects developed yet. This incremental approach both diffused adoption cost over several months and also allowed concepts and ideas to soak into the team consciousness.

4.3 Unit and Integration Tests

One of the practices of Extreme Programming that eased our adoption of AspectJ was automated testing. At the time we began adoption, we had amassed a suite of several hundred unit and integration tests. Automated testing helped us in two ways: 1) it allowed us to verify an aspect's implementation during development of that aspect 2) it allowed us to quickly detect side-effects or to notice when a refactoring had broken an aspect.

Programmer testing has many benefits whose merits have been explored elsewhere. We found most of those benefits to apply straightforwardly to aspect-oriented code. For instance, testing advocates assert that writing tests can improve object-oriented design. One design benefit arises from creating classes that can be easily invoked outside of their original context. Writing such classes helps to ensure loose coupling. We found this benefit to have a parallel in aspect-oriented code. Writing pointcuts that can match join points outside of the ones originally intended (i.e. ones artificially reached by a test) ensures that pointcuts are loosely coupled to the code they query. For instance, defining a pointcut in terms of a marker interface allows a test class to implement that marker interface and be affected by the aspect. Outside of the test, such a pointcut also allows new domain classes to implement the marker interface and subscribe to the aspect's effects.

4.4 Pair Programming

Another practice we found valuable was pair programming. There is still debate in the larger programming community about the merits of pairing as a general practice. However, speaking from personal experience, I find that it's an invaluable way to transfer knowledge. Transferring AOP/AspectJ skills was no exception. A typical early pair session would place me with a less-experienced developer as we tried to write an aspect to address a specific concern. As we came to new language features or a new aspect-oriented concept, I would explain the concept and continue with the implementation.

As team experience grew, aspect-oriented pairing sessions began between other team members. We used our shared workspace to take advantage of the "Expert in Earshot" [4] effect. Team members struggling with the nuances of a pointcut could easily get my attention for a quick resolution. Also, if I overheard discussion that indicated a misunderstanding, I could intervene to correct.

The combination of testing and pairing gave us the confidence to use our application as a learning environment. We were able to try out different ideas, and easily see their realization within a familiar context. This helped us learn AOP more rapidly and confidently than we otherwise might have.

4.5 Refactoring

In our XP environment, we refactor often, and aspects are no exception. Just as object-oriented code can vary in its readability and expressiveness, aspect-oriented code can be clear or confusing. Accordingly, we revisit our aspects (e.g. when we modify them to add a new feature) and attempt to express their intent more clearly. Many of the habits, practices, and guidelines for refactoring object-oriented code translate well to aspect-oriented code. One clear carryover from other programming paradigms is the principle of meaningful names. The name of an aspect or a pointcut can be an invaluable clue to the intended effect of the code.

In addition to refactoring aspects, we also refactor our code to add aspects where warranted, and to remove aspects when the intent of the program is better expressed without them. As an example, one refactoring of the repricing behavior that we experimented with dispensed with the aspect entirely.

One corner case that seems to have no clear solution is the case of an aspect that affects the implementation of a single class. If, for instance, an aspect affects five methods on a class, is it better to manually insert calls to code that would otherwise be advice, or to remove the (small amount of) duplication and place the behavior in an aspect? Static inner aspects offer an elegant compromise, but are fraught with difficulty because of tool issues.

The AOP community is beginning to put together material on aspect-oriented refactoring. [8] We look forward to applying some of these techniques to our development practice in the future.

4.6 Best Practices

With aspect-oriented programming having only a few years of production use under its belt, best practices for aspect-oriented development are still evolving. The Adbase team has developed few best practices outside of those already enumerated elsewhere. Most of our design experience at this point has taken the form of implicit knowledge shared informally or through pairing.

4.7 Management Reaction/Business Impact

A team adopting a new technology always incurs some risk. The technology's cost (expressed in time, productivity, and/or capital) may outweigh its benefits, or the cost may be prohibitively high despite a correspondingly large reward. Although learning AOP and AspectJ took time and effort, the cost has been lower and/or more diffuse than other technology adoptions during the life of the project. Accordingly management reaction to AOP has been largely neutral. The Adbase team could do more to evangelize the technology choice and highlight its benefits, but it faces no significant pressure to justify continued investment at this time.

4.8 Risk of Over/Misuse

As I suggested earlier, any new technology faces the risk of overadoption. An overeager team can apply a technology outside its area of strength or apply it so widely that weaknesses begin to manifest. Our XP process encourages developers to constantly question complexity, to "do the simplest thing that could possibly work." It also suggests to put off to tomorrow any implementation effort that does not address a need of today: "You ain't gonna need it." This philosophy encouraged a healthy skepticism that balanced the team's enthusiasm for the new technology. Although we did not follow any formal guidelines to limit the spread of aspects, discussions during pairing sessions offered developers the chance to vet proposed applications and suggest alternatives.

5. TOOL SUPPORT

A dramatic, if little noticed change in the mainstream programming community over the last five years has been the emergence of automated tools as central to the practice of development. An increasing number of programmers (the VMS team included) use an IDE for day-to-day coding. As time passes, it becomes increasingly difficult to disentangle a programming language from its supporting toolset. This contention applies especially to AspectJ. This section examines our experiences with the current AspectJ development tools. Our experience has centered on AJDT—the AspectJ plugin for Eclipse, though we have also used the Ant and Maven integrations.

Our experience has been mixed. While the debugging support and crosscutting views are essential to effective AspectJ development, there are significant practical concerns still to be addressed. Before further consideration of the tools, it's worth noting that they are improving rapidly. IBM and BEA continue to invest significant resources in AspectJ and AJDT's development. Over the course of our adoption (and even the writing of this report) the AJDT environment evolved to address concerns that would otherwise appear in this section as significant barriers to adoption. I hope that as of publication, further advances will have arrived.

5.1 Crosscutting Views

As I've stated before, depending on the degree of orthogonality, a programmer can be more or less oblivious to a crosscutting concern. When working on a module, a programmer must be somewhat aware of crosscutting aspects (or at least be able to transition quickly from oblivious to aware). This is especially true when the concern is tied closely to the core functionality of the module (for example, repricing a basket). Similarly, in the "aspects as a red herring" situation, it's critical that a programmer be able to rule out (or find) potentially troublesome aspects with confidence.

To assist in making the crosscutting structure of a program understandable, AJDT provides both an outline of the crosscutting structure and also gutter annotations that flag the application of advice to a given element in the editor.

5.1.1 The Crosscutting Structure/Outline View

Our experience with the crosscutting outline view has been excellent. It allows developers to view all of the join points affected by a given piece of advice. This functionality permits programmers to easily analyze the potential effect of an aspect. When writing a new pointcut, for instance, it alerts you if an error prevents a pointcut from matching any join points. Similarly, a quick scan has tells you if the pointcut matches unintended points. Unfortunately, as of the current stable release of AJDT (1.1.12) the outline view is only present after a full rebuild. Aside from the time cost, this issue does not hamper development, because it's readily apparent that the project needs to be rebuilt to see the view. Milestone releases of AJDT (1.2M2 and beyond) dispense with this limitation.

5.1.2 Gutter Annotations

When a programmer views a class (or indeed an aspect) in Eclipse's editor, gutter annotations indicate the application of advice to a given program element. This feature does much to assuage the fears of AOP neophytes. By providing a clear indication of aspect-impact and an easy means to navigate back to the aspect, AJDT's gutter annotations have helped us diagnose aspect-related problems on more than one occasion. They also serve as unobtrusive reminders that aspects contribute to the total behavior of the code in the current editor.

As of AJDT 1.12, the gutter annotations exhibit some of the same problems as the outline view. They require a full rebuild of an affected file before becoming visible. In the case of the gutter annotations, this is a more severe problem since there is no way of knowing that a rebuild might be needed to see them. The latest builds in the AJDT 1.2 stream have addressed these issues.

5.2 Debugging support

Although the debugging support in AspectJ exhibits a few problems (occasional inability to set breakpoints without a rebuild, some confusing stack frames to step through on cflow advice) the current implementation operates as desired. In other words, you can set breakpoints in classes and step into advice that applies to them. You can also set breakpoints in advice. As we developed the relationship management aspects, we frequently used the debugger to help us understand the dynamic behavior of the code. Once the minor deficiencies in the debugger are resolved, working with aspect-oriented code in the debugger should be as painless as working with object-oriented code.

5.3 Compilation Speed

The biggest challenge to further adoption of AspectJ at VMS is the significant increase in compilation times when using AspectJ. From experience and the literature [5], it seems that AspectJ provides batch compilation performance that is comparable to, if slower than, that of pure java compilers. For instance, it takes approximately 35 seconds to compile 700 classes and 70 or so aspects. Importantly, however, most builds using an unaugmented Java compiler are *incremental.* The compiler only compiles classes that have changed since the last compilation, or which depend on classes that have changed. AspectJ offers a similar incremental mode, but with two important restrictions. The first is that incremental compilations using AJDT (in the Adbase project) invariably take a couple of seconds longer than the near-instant times enjoyed by pure Java builds. The second problem is that changes to an aspect require a full rebuild (performed automatically by AJDT even if you are using incremental mode). Batch compilations can also be necessary when certain modifications to source files leave the incremental compiler in an inconsistent state.

These two types of slowdown have a noticeable impact on development velocity. Our XP-based development process makes heavy use of unit tests and incremental development. In an ideal world, we would modify code, compile it, and execute the relevant unit tests several times per minute. Indeed, that's how it works on our pure Java projects. Adding even a few seconds of delay to the cycle slows development on two fronts. First, time is taken by the compilation itself. Second, during the wait interval, human attention can wander and it can take time to re-contextualize after the compilation. This problem is particularly pronounced for the full builds, which tempt the programmer to switch to another task entirely (e.g. email, Slashdot headlines). One of VMS's most talented programmers, despite having positive experiences on the Adbase project and having designed an aspect-oriented testing framework, resists using AspectJ in his other projects because of these delays. He puts it this way: "The benefits realized by AOP over pure OO approaches do not outweigh the drawbacks of productivity loss and impediments to a Test-Driven Development approach, at least in a fairly small application."

The good news for performance is that the AspectJ and AJDT teams have made compilation speed improvements a goal for the next version of the AspectJ tools. Batch build time was cut by 50%, and incremental build time by 25% between the first and final draft of this report. Further planned enhancements aim at reducing the need for full compilations when an aspect changes.

5.4 Refactoring and Java Editing: Eclipse sets the bar high

Although AJDT brings most of the functionality of a good Java editor to the AspectJ language (code completion, syntax highlighting) there are important missing areas. These feature-holes have tripped up developers used to Eclipse's rich functionality. Code completion works only in certain contexts, for instance. Another example: the shortcut for "Open Type" does not apply to aspects.

More importantly, refactorings initiated from Java do not affect aspects. So, for instance, changing 'Foo.bar()' to 'Foo.baz()' using 'rename method' would not alter a pointcut like execution(public void Foo.bar()). Furthermore, orphaned pointcuts such as this only trigger warnings in AspectJ 5 and only if they prevent the entire pointcut from matching. This lack of refactoring support made wide-ranging refactoring a more delicate and manual process. The AJDT team has announced future support of both Java based and aspect-specific refactorings. VMS looks forward to it.

5.4.1 Miscellaneous Problems

It's important to note that the feature-holes described only affect aspects. In general, the experience when using AJDT with Java code is unchanged. One significant exception was that the Eclipse Java parser does not understand inter-type declarations. If you add a method foo() to your Cart object using an ITD, Eclipse will flag a call like someCart.foo() as an error even though an AJDT build will bless it as correct. As of this writing, AJDT users are left with the choice of disabling eager parsing (and eliminating helpful auto-corrects that come with it) or to tolerate these ersatz problem indicators.

5.5 Incompatibilities

For the most part, the AspectJ compiler has played well with other tools used at VMS. However, we have found some incompatibilities. Most of these can be explained with the following quote from a practitioner report at AOSD 2004: "Whilst the AspectJ compiler (ajc) produces 100% legal Java bytecodes, some tools that work at the bytecode level (for example, disassemblers) can get confused by the bytecodes that ajc emits. In general this is because the tools rely on recognizing bytecode patterns emitted by javac." [6] This section describes two of the incompatibilities we encountered while using AspectJ.

The first, and more serious, of the two issues occurred with the JRockit 8.1sp2 JVM and AspectJ 1.2. A bytecode pattern in an AspectJ classfile (that also confused one decompiler we tried) caused JRockit to crash. Happily, both BEA and the AspectJ team released a fix for this issue shortly after it was reported (the current versions are AspectJ 1.2.1 and JRockit 8.1sp3). Note also that BEA's support of AspectJ 5.0 makes future problems with their JRockit JVM as unlikely as they are with IBM's JVMs.

The second problem occurred while attempting to add an inter-type method to an EJB deployed to Weblogic. Recall that EJB components require separately defined interface and implementation classes, as well as post-processing to generate interceptors. The post processor for Weblogic 8.1sp2 detected the ITD as an error and refused to continue with the deployment of the EJB. We worked around this issue by implementing a solution that did not require the inter-type declaration.

Though both of these issues were relatively serious, they did not stop us from continuing the adoption. Both the AspectJ team and BEA's support department were helpful in investigating the problems.

6. Conclusions

VMS considers its adoption of AOP successful. The key barriers to further adoption within the company remain in the area of tool support rather than in training or conceptual understanding. In particular, compilation times incur a productivity penalty that offsets the general benefits of aspect-oriented development. The capabilities of the AJDT environment also lag behind the seamless experience of pure Java development in Eclipse. Luckily, both these areas are currently receiving significant attention.

Despite these barriers, aspects have noticeably improved the ilities of the Adbase application. The application exhibits improved resilience to change and ease of understanding in areas where we have successfully applied aspects. AOP has also reduced the amount needless duplication and busywork necessary when adding functionality to key areas (for example, adding a new persistent relationship).

The Adbase team has learned that its agile development approach handles the challenges of aspect-oriented development well. Our methodology limited risk, encouraged beneficial skepticism, and maximized knowledge transfer. In retrospect, an increase in the formal evaluation of the impact of AOP could also have yielded benefits.

6.1 Future Plans

VMS plans to increase its investment in AOP and AspectJ by encouraging new aspect-oriented development on the Adbase project and considering adoption on other projects. VMS eagerly anticipates future work in the area of aspect mining and aspect refactoring, in the hopes of more easily improving the quality of existing code.

The imminent arrival of AspectJ 5.0 also presents an exciting opportunity. The @AspectJ syntax and load-time weaving capabilities offer a low cost point of entry for AOP. In addition the support for annotation-based pointcuts promises to open up significant new areas of functionality.

7. ACKNOWLEDGMENTS

My thanks to Scott Segal, Angus McIntyre, Chad Woolley, and the other members of the Adbase team for their reviews and advice.

8. REFERENCES

[1] Gradecki, Joseph and Lesiecki, Nicholas. Mastering AspectJ: Aspect-Oriented Programming in Java. John Wiley and Sons. 2003.

[2] The AspectJ Team. The AspectJ Programming Guide. http://dev.eclipse.org/viewcvs/indextech.cgi/~checkout~/aspectj-home/doc/progguide/index.html. 2004

[3] Colyer, Adrian. "I don't want to know that... (writing robust pointcut expressions)". [Blog Entry]. The Aspects Blog. http://www.aspectprogrammer.org/blogs/adrian/2004/08/i_dont_want_to.html. August, 2004.

[4] Cockburn, Alistair. "Expert In Earshot". [Wiki Entry]. Portland Pattern Repository Wiki. http://c2.com/cgi/wiki?ExpertInEarshot. June 2004.

[5] Hilsdale, Erik and Hugunin, Jim. Advice Weaving in AspectJ. AOSD 2004 Technical Paper. March 2004.

[6] Colyer, Adrian, et. al. Using AspectJ for Component Integration in Middleware. OOPSLA 2003 Practitioner Report. October 2003.

[7] Laddad, Ramnivas. Metadata and AOP: A perfect match. IBM developerWorks. [Website]. March 2005 (prepublication review copy).

[8] Laddad, Ramnivas, Aspect-Oriented Refactoring Series. TheServerSide.com. [Website] http://www.theserverside.com/articles/article.tss?l=AspectOrientedRefactoringPart1. December 2003.