Large-scale AOSD for Middleware

Adrian Colyer
IBM UK Limited
Hursley Park
Winchester, England
+44 1962 816329
adrian_colyer@uk.ibm.com

Andrew Clement
IBM UK Limited
Hursley Park
Winchester, England
+44 1962 816658
clemas@uk.ibm.com

ABSTRACT
For a variety of reasons, today's middleware systems are highly complex. This complexity surfaces internally in the middleware construction, and externally in the programming models supported and features offered. We believed that aspect-orientation could help with these problems, and undertook a case study based on members of an IBM® middleware product-line. We also wanted to know whether aspect-oriented techniques could scale to commercial project sizes with tens of thousands of classes, many millions of lines of code, hundreds of developers, and sophisticated build systems. This paper describes the motivation for our research, the challenges involved, and key lessons that we learnt in refactoring both homogeneous and heterogeneous crosscutting concerns in the middleware.

Categories and Subject Descriptors
D.1.0 **[Programming Techniques]**: General.

General Terms
Design, Languages

Keywords
Aspect-oriented, middleware, refactoring.

1. INTRODUCTION
The goal of middleware is to make it easier to build distributed systems that achieve desired qualities of service. Middleware hides the complexity that developers would otherwise need to deal with when writing such applications. yet the variety and complexity of middleware itself still surfaces. This makes the building of distributed systems (such as enterprise applications) more difficult than we would like. Aspect-orientation is an evolutionary step in software engineering that helps tame complexity by improving modularity and untangling feature implementations. Can aspect-oriented software development (AOSD) techniques be applied in the middleware domain to help tame some of the complexity inherent both in building and applying middleware? We set out to answer this question within the context of an IBM middleware product-line. In particular, we wanted to know if AOSD could help us achieve consistent implementation of crosscutting concerns in the product-line, and if it could help us to increase flexibility by facilitating separation of tangled feature implementations. These goals are discussed further in sections 1.2.1 and 1.2.2. For the approach to be viable it had to scale to commercial project sizes.

AOSD 04, March 2004, Lancaster UK.

1.1 The causes of middleware complexity
The causes of middleware complexity are many and varied. Middleware products tend to be feature-rich, a statement of their relative maturity and of the competition in the marketplace. Implementing all of the needed features causes internal complexity, and understanding and choosing between the various features on offer causes external complexity. As a simple example, consider a middleware product that offers both synchronous and asynchronous communication. There are some situations in which a synchronous style is clearly to be preferred, and other situations in which asynchronous communication is clearly to be preferred. There are also many shades of grey between the two. The shades of grey are a source of confusion for designers. The more features we add, the more points there are in the design space, and the more complex the task of choosing between them.

In addition to individual middleware products being feature-rich, there are also many middleware products and technologies. Examples are databases, application servers, message brokers, edge servers, web servers and so on. These products are used together to build complete solutions. The presence of so many products further confuses the design space. It also adds to the administration and operation overhead.

Another driver of middleware complexity is the push into new markets, such as pervasive computing, that demand support for ever more heterogeneous environments and place new constraints on resource usage. Trends in middleware research to make middleware more flexible and adaptable for situations such as these also tend to increase programming model complexity by exposing additional APIs for refection and adaptation [1, 2].

1.2 The role of AOSD
We identified three areas in which aspect-orientation may help to reduce complexity in middleware product-lines: ensuring consistency across product-line members, separating concerns for increased flexibility in creating products in the line, and in simplifying the programming model. Our investigations to date have focused on the first two of these, which we believe will lay a foundation of understanding in applied AOSD that will help us to address the third challenge in due course.

1.2.1 Consistency across Product-Line Members
In any product-line you expect to find a certain family resemblance amongst product-line members. In the middleware

product-line used as a basis for this study there are numerous standards (policies) with broad applicability across the line. Many of these fit readily into the classic examples given for AOSD – things like tracing and logging, error and exception handling, and gathering of monitoring and statistics data. Can AOSD help us to capture these policies, and apply them *consistently* across the product-line? AOSD should also help us to apply the policies *efficiently* – making them easier to introduce and to evolve. Section 2 presents the results of our study in this area.

1.2.2 Refactoring for Product-Line Flexibility

The limitations of object-orientation, and the complexity and number of features in middleware products, mean that some feature implementations end up entwined [3]. Features that are not cleanly modularized cannot be easily separated, and therefore cannot be extracted from the base in which they are tangled. This prevents us from easily producing product-line members in which the feature is not present, and from reusing the feature implementation in new product-line members. Can AOSD as a modularization technology help us to solve this problem? Section 3 presents the results of our study in this area.

1.3 Challenges

In addressing the questions outlined in the preceding sections, merely achieving a separation of concerns is not enough. The AOSD techniques employed must be able to integrate into the development and build processes used within the product-line, and need to scale to commercial project sizes; in our case this meant tens of thousands of source files, many millions of lines of code, and hundreds of developers. The build time performance needs to be acceptable (the overnight builds had better still complete overnight), and so does the runtime performance. The reliability, serviceability and robustness of the result are vital – the cost of downtime for some customers of the product-line runs into millions of dollars per hour.

2. HOMOGENEOUS CROSSCUTTING CONCERNS

In the middleware product-line used as the basis for this part of the study, there are multiple standards (policies) that are applied across product-line members. Typically these standards are centrally documented, and then form a part of every developer's ongoing education. Developers need to read and internalize them, and then consistently apply these policies whilst working on their source modules. We have previously reported on some of our work refactoring these policies in [4, 5], and so present only a summary and update here.

We attempted to capture three of the basic policies of the product-line using AspectJ [6] aspects. These policies were for tracing and logging, first-failure data capture, and monitoring and statistics gathering. The crosscutting concerns captured by these policies are homogeneous in nature – whilst there is broad scattering, the scattered logic is very similar in each location.

2.1 Tracing and Logging

The tracing and logging requirements for the product-line are captured in an extensive policy document. We were able to capture the policy in an abstract aspect that defined both when and how tracing was to be performed. Each component in the product-line then only needed to supply a concrete sub-aspect specifying where to trace by providing a concrete specification of a scoping pointcut. This division of responsibility allows the architects to both define *and implement* a global policy, whilst component owners decide how that policy should be applied within their domain.

The aspect-based solution gave a more accurate and more complete implementation of the tracing policy. When examining the source-code of the product-line as part of capturing the policy, we found several examples where tracing was not implemented completely, other cases where there was inconsistency and ambiguity in interpreting the policy, and some places where tracing calls were not correctly guarded with checks on whether tracing was enabled. These last policy violations can cause runtime performance overhead when running in production. All of these mistakes are the natural consequence of asking humans to perform mundane and repetitive work.

Ironically (given its prevalence in the aspect literature), the tracing and logging policy was the one concern that we tried to capture in an aspect where we could not remain 100% faithful to the original product-line design. The tracing policy required each individual class to register with the tracing facility, and use a returned object for all future calls to the tracing service. The registration is done during static initialization of the class. The current version of AspectJ (v1.1.1) does not support inter-type declarations of static members to multiple classes. The proposed pertype language extension [7] will resolve this issue. We worked around it by registering at the component rather than the class level for the purposes of the study.

We ran some performance tests with AspectJ 1.1, and measured the overall system overhead in the case where tracing is disabled (the performance sensitive case) *and advice does not use 'thisJoinPoint'*, at 1% or below compared to the ideal hand-coded equivalent. Using 'thisJoinPoint' within the body of the advice can however cause significant additional overhead. This is because the 'thisJoinPoint' object is created before the advice body is entered – even if access to it within the advice body is guarded by a test that may evaluate to false (such as testing a tracing flag). Lazy creation of 'thisJoinPoint' objects is a candidate item for the AspectJ 1.2 release and will resolve this issue fully.

Bear in mind that a true comparison of aspect-based and scattered implementations should take into account that not every developer will faithfully follow all of the performance guidelines when implementing tracing by hand. Since the aspect implementation can be carefully optimized and then applied consistently everywhere, the overall system performance, even if not as high as the perfect scattered implementation, can still be very competitive. With an optimized AspectJ compiler implementation, the aspect-based solution may well exhibit better performance than the scattered solution you would expect to find in a typical code base.

2.2 First Failure Data Capture

The product-line uses a sophisticated error analysis and reporting subsystem following the principle of "first-failure data capture" (FFDC). FFDC seeks to ensure that all pertinent information relating to a failure is captured as close to the source of the error as possible. This information is then passed into a diagnostics and analysis component for logging, and execution of any recovery actions.

We were able to fully capture the FFDC policy using AspectJ. We used the same approach as outlined for tracing and logging, whereby a single abstract aspect captures the policy, and component owners provide a concrete sub-aspect that specifies where FFDC should be applied within their component. As well as handling the invocation of the FFDC mechanism upon detecting an error, we also used aspects to create and register component diagnostic modules. The modules provide additional diagnostic information to the FFDC engine on request. We found that the aspect-based approach to FFDC enabled us to go beyond what was practical with a scattered implementation, and we are now investigating the extension of our FFDC capabilities using AspectJ.

2.3 Monitoring and Statistics

The product-line is extensively instrumented to capture monitoring and statistics data. We analyzed the existing implementation of this data collection in a significant component of the product line. We found that it was scattered across ten classes in the component, and by considering the data collection as a concern in its own right, were able to uncover subtle inconsistencies in where information was collected.

We then implemented the data collection using a single aspect that applied a consistent policy for capturing the required information. Each statistic to be gathered mapped neatly into a single pointcut definition, making the code look just like the design document. Moreover, the aspect solution was able to encapsulate the (previously scattered) associated state data. We were easily able to generalize the approach for a second component with comparable convenience and further reduced effort.

2.4 The WSIF Story

The Web Services Invocation Framework (WSIF) is an open-source Apache project initiated by IBM [8]. WSIF is also used as a 'binary component' within the product-line: the project's jar files are included in the product-line build, but it is not built from source. When WSIF is used within the product-line, we would like it to have the same qualities of service as the product-line source that we do exclusively own and build. We were able to write WSIF sub-aspects of the tracing and logging, first-failure data capture, and monitoring and statistics aspects. These were compiled into an aspect library. We augmented a copy of the product-line build system to apply these policies to the WSIF jar files as an additional stage in the build process.

2.5 Organizational Implications

David Parnas observed that modules are a unit of work assignment [9]. On large development teams, the ability to modularize crosscutting concerns such as those described in this section has huge organizational implications. The team that specifies policies can move from writing and disseminating policy documents, to implementing the policy itself. Individual developers (of which the product-line has hundreds) need not then read, internalize, and remember to faithfully apply every one of those policies. Instead their education need be for awareness only. The aspect implementations ensure a more complete and accurate implementation of policy, and the cost of evolving the policy or its implementation is greatly reduced.

3. HETEROGENEOUS CROSSCUTTING CONCERNS

This section discusses our experiences refactoring a large-scale crosscutting concern in an application server belonging to the middleware product-line. We describe the task attempted, the process used, and the results of the study. Whereas the concerns addressed in section 2 were homogeneous (consistent application of the same or very similar policy in multiple places), the concern addressed in this section was heterogeneous; it still impacted multiple places and exhibited scattering and tangling, yet the behavior in each of those places was different.

3.1 Challenge

The build of the application server that we used as a basis for the study comprised approximately 15,000 source files and many millions of lines of code. Internally the application server was structured in over 250 modules (components), which we reduced to approximately 80 top-level concerns in the system. Examples of these concerns are the ORB, web container, EJB container, web services support, and so on.

The task that we undertook was to separate support for Enterprise JavaBeans™ (EJBs) from the rest of the application server. By setting a single flag in the build process, we wanted to be able to build a variant of the application server that either included or excluded support for EJBs. The refactored system, when built with EJB support included, should pass all of the build verification tests defined for the application server. When built without EJB support included, the application server should pass all of the build verification tests not involving EJBs, and should fail the EJB based tests in a graceful manner.

Even though the application server was well modularized, this represented a considerable challenge – support for EJBs entails more than just the EJB container itself, it also encroaches on administration, management, debugging, persistence, and many other components besides.

3.2 Process

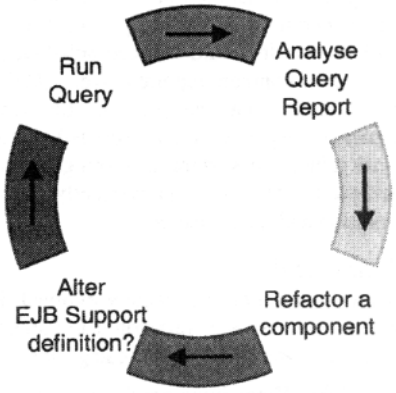

Figure 1. Iterative Refactoring Process

We used an iterative process involving simple concern modeling [10] to define and separate the EJB support concern. Each iteration refined the concern definition through a process of gradual discovery and incremental refactoring. The discovery was driven by searches that showed links from the application server to the evolving concern. Figure 1 shows the main steps in each iteration.

We used the Eclipse IDE [11] for the refactoring work, with the AspectJ Development Tools (AJDT) [12] installed. Our starting point was an initial definition of the EJB support concern that included solely the EJB container pieces that were already well modularized and easy to identify.

3.2.1 Searching for Links

On each iteration we searched for links (references) to the EJB support from the rest of the application server. To undertake these searches we started out by using declare warning statements in an enforcement aspect that looked something like the following:

```
public aspect EJBSupportSeparation {

  pointcut inEjbSupport() :
    within(T1) ||
    within(T2) ||
    ...;

  pointcut ejbSupportLink() :
    call(* T1.*(..)) ||
    call(* T2.*(..)) ||
    ...;

  declare warning :
    ejbSupportLink() && !inEjbSupport() :
    "Link to EJB Support found.";
}
```

We had a number of practical problems using this approach on our very large codebase, and as the definition of the EJB support concern grew. The foremost of these was that our iterative process meant we needed to run the query frequently. Using the AspectJ compiler to find the links via an aspect involved paying the cost of a full compilation for every query. This turned out to be simply too slow for effective working. We switched to using developing components of the Concern Manipulation Environment (CME) [13], the 'conman' concern manager and the 'puma' query engine, to execute the queries. We defined concerns to conman for each component in the application server, and also for the evolving EJB support concern. Our query produced a report of links to the EJB support concern ordered by component. Our colleagues in the IBM T.J. Watson Research Center enhanced the conman component to handle queries over a very large code-base: full index generation, followed by execution of the query and reporting of the results could be accomplished in under five minutes. At the peak, conman reported over 1,000 links to the EJB support concern that needed to be resolved. The EJBSupportSeparation aspect was not discarded; it stayed in place even in the final refactored system as a safeguard to ensure the separation achieved was not inadvertently broken during subsequent evolution and maintenance.

3.2.2 Analysing Links

By examining the links found we steadily evolved the definition of the EJB support concern and refactored the source code. We classified the links that we found in four categories:

1. **Rogue** links. These links are false positives. They arise because the build system declared a component dependency, which upon analysis turned out to be unneeded. We simply removed the dependency in the build scripts.
2. **Minor** links. These links are things like unused import statements. Again, they are easy to fix – we simply remove the redundant import.
3. **EJB support** links. These links are from code that was not included in the definition of the EJB support concern used for the query, but on analysis should be considered a part of the EJB support concern. Upon finding such links, we changed the definition of the EJB support concern to include them. At this point of course, the search needs to be started again as the definition of the EJB support has changed.
4. **External** links. These are genuine dependencies from outside of the EJB support concern, on types and type members defined within it. These links have to be separated by refactoring so that it is possible to build a variant of the application server in which they do not exist. It must also be possible to build a variant that is semantically equivalent to the pre-refactoring version, in the case when EJB support is to be included. We used a combination of object-oriented and aspect-oriented refactoring to achieve this.

For links reported by the search that required genuine refactoring in the code, we followed a policy of first attempting to refactor the code using object-oriented techniques, and only if that was not easily possible did we consider using aspect-oriented techniques (we used AspectJ in these cases). We also had a pragmatic goal of disturbing the original code-base as little as possible (to maximize the chance that the application server would execute correctly once refactored).

3.2.3 Refactoring

The simplest form of refactoring occurred when whole methods or fields declared within a type belonged to the EJB support concern. Here we simply moved them to an inter-type declaration in an aspect. In some cases the moved methods and fields were self-contained, but more often the method bodies referred to other methods or state in the class from which they came. If these methods or state were private then they had to be made visible to the aspect in some way. This could entail adding access functions for private state, promoting the visibility of private methods, or where the aspect was tightly coupled to the class, marking it privileged. In this latter case, what we really wanted to express was that the new aspect was a privileged aspect with respect to the donor class (rather than the aspect having general privileges to see any private state anywhere).

Where tangling of EJB support and non-EJB support concerns occurred within a single method, more sophisticated refactoring was required. Our most common approach here involved use of what we termed a 'mini-strategy' pattern. The tangled method was refactored to separate out the points of variance into separate methods, but then instead of defining a strategy interface, we simply left the methods in the class. In the majority of cases, the refactored source was clearer than the original, and exposed suitable joinpoints for aspects in the EJB support concern to extract the tangled logic into before or after advice. On those rarer occasions where a method was simply too tangled to sensibly split we removed it from the original class entirely, and used inter-type declarations in (different) aspects for both the EJB support and non-EJB support variants. Only one of these two aspects would be included in any given build (they represent alternate feature implementations).

The following simple refactoring example illustrates the approach, and is derived from a real portion of the application

server source code. First we present an extract from the original source:

```
public class DebugAppServerCI
  extends ComponentImpl {

// state declarations…
…
private EJBContainer ejbContainer;
…

public void initialize(Object config) throws …
{
  // some setup
  …
  if (debugMode) {
    // prepare debug filters
    …
    Tr.debug("registering EJB debug
              collaborator");
    ejbContainer =
      getService(EJBContainer.class);
    if (ejbContainer != null) {
      // register collaborator
      …
    }
    // register web app collaborator
    …
    // register other collaborators
    …
    // initialize debug agent
    …
  }
  …
}
…
}
```

The first step in the refactoring process here was to use ExtractMethod [14]to pull the section of the initialize method concerning collaborator registration into a separate method, "addCollaborators." This had the additional benefit of improving the clarity of the (rather long) initialize method. Then we created an aspect EJBCollaboratorRegistration, and moved the ejbContainer field into it. The EJB collaborator registration is removed from the addCollaborators() method (now that a suitable joinpoint is exposed), and the behavior is moved into advice.

```
public class DebugAppServerCI
  extends ComponentImpl {

// state declarations…
…
public void initialize(Object config) throws …
{
  // some setup
  …
  if (debugMode) {
    // prepare debug filters
    …
    addCollaborators();
    // initialize debug agent
    …
  }
  …
}

void addCollaborators() {
  // register web app collaborator
  …
  // register other collaborators
  …
}
…
}

public aspect EJBCollaboratorRegistration {
  private static final TraceComponent tc = …;

  EJBContainer DebugAppServerCI.ejbContainer;

  pointcut collaboratorRegistration(
             DebugAppServerCI ci) :
   execution(* addCollaborators(..)) &&
   this(ci);

  before(DebugAppServerCI ci) :
     collaboratorRegistration(ci) {
    // register ejb collaborator…
  }
}
```

With the EJB specific behavior cleanly encapsulated it became clear that the ejbContainer field was not referred to outside of the advice body, and the field was replaced with a local variable inside the advice body. At this stage, having proceeded with the refactoring in small self-contained steps, we have a new program in which the original behavior is restored if the aspect is included in a build, and no knowledge of the EJB support concern is present when the aspect is excluded. We defined a new TraceComponent for the aspect (the policy in the application server is to have one per type) rather than referring to the TraceComponent in the DebugAppServerCI class, because the aspect really is a new type in the system, and we want the trace records to reflect that.

3.3 Results

We were able to separate the EJB support concern from the rest of the application server, and we were able to do so in a relatively short space of time. AspectJ was comfortably able to build the very large project, and was easy to integrate into the build process. When building the full-function system using aspects to integrate the separated EJB support concern, the resulting executable passed all of the build verification tests. When building the variant without the EJB support concern included, all non-EJB tests passed, and the EJB based tests failed gracefully with a "NameNotFoundException" when trying to look up an EJB home in the namespace.

We took a number of measurements that compared the version of the application server without the EJB support to the original version including it, in order to assess the benefits of increasing flexibility in the product-line. Our refactored application server showed significant improvements in start-up time and memory footprint, and marginal performance improvements in running the build verification tests.

The remainder of this section provides more detail about the results of the refactoring, and discusses lessons learnt in the process.

3.3.1 Separation of Concerns

At the conclusion of the study, we had made changes in 35 of the 80 top-level components (and created a few new components too), giving an indication of the very broad crosscutting nature of the EJB support concern. The concern is heterogeneous, rather than homogeneous, since the logic that we needed to refactor in those 35 components was very different in each case. The final EJB support concern comprised over 700 types (classes and aspects), about 5% of which were aspects. 12 components ended up with a mix of EJB support and non-EJB support function still within them (but internally separated). This is because the dominant

decomposition chosen by the original designers was at least as good as any alternate structuring we considered. We handled this situation in the build process by adding the ability to build variants of components, based on the setting of global 'feature-inclusion' properties. When finally the EJB support concern was fully statically separated from the rest of the application server (the search reported no links remaining to the EJB support concern), we still found additional links at runtime caused by use of reflection, and the loading of classes by name from configuration files. Additional refactoring was required to break these links also.

Whilst we did not compute any metrics, it is our belief that the clarity and simplicity of the remaining application server portion (once the EJB support had been factored out) had improved. This portion could clearly (as we demonstrated) be 'reused' independently of the EJB support concern. The EJB support concern on the other hand retains many connections to the application server base from which it was taken. Significant additional work would be needed to turn the EJB support concern into something that could be reused in other contexts. This work was beyond the scope of our study.

3.3.2 Aspect-oriented vs. Object-oriented refactorings

We expected that the aspect-oriented refactorings would offer more expressive power than we could achieve with object-orientation alone. We were also pleased to discover that the aspect-oriented refactorings were often simpler and less-invasive than their object-oriented counterparts. Recall that we are trying to create a program family that allows for variance (the EJB support is present or not). Using object-orientation the primary tools available to us are subclassing (overriding method behavior in the base class(es) to handle variations), and use of patterns such as Strategy. If we choose the subclassing route, then we also need to introduce some kind of Factory pattern to ensure that the appropriate subclass is created for the features included in the current program family member. Then we need to change all creations of the class to go through the factory. Introducing a Strategy also creates disturbance in the classes adapted, and requires us to introduce some means of creating the appropriate strategy object for the situation. Whether using subclassing or strategy, the same problem arises of combinatorial complexity when considering the case of program families with multiple optional features. Suppose that we wanted to make the tracing, first-failure data capture, monitoring and statistics gathering, and EJB support all optional features in a program family. To cater for all the variants that might legitimately be created, we would need up to 2^4 subclasses of each type in the base program. This clearly becomes untenable very quickly. To address this we have to introduce further mechanism patterns such as ChainOfResponsibility. The whole process introduces significant complexity into the code and is very invasive. The introduction of variance in this manner constitutes a crosscutting concern in its own right.

Contrast this to the aspect-oriented solution, where we were frequently able to introduce the variance by capturing it directly in an aspect using inter-type declarations or advice (as described in 3.3.1). In this case, except for the removal of the variant code, the original structure was disturbed very little. The aspect-oriented refactorings are a smaller step to take (more 'agile') than their object-oriented equivalents. For n optional features implemented using an aspect-oriented solution, the number of types to be created is O(n).

As the refactoring work progressed, our rate of progress increased as we started to recognize common situations (or patterns) in the code that we needed to refactor, and to apply the same solution templates repeatedly to address them. This gives us hope that in time, and with a larger base of examples for validation, structured refactoring patterns can be found (along the lines of those prescribed in Martin Fowler's book, "Refactoring" [14]) for separating concerns using aspect-orientation.

3.3.3 The Power of Visualization

We used the SeeSoft style [15] visualizations produced by AJDT extensively in the early stages of refactoring. We worked component by component, and the visualizations gave us a very good indication of the extent of the tangling within each one, enabling us to perform an early assessment of the task ahead. We came to spot very quickly from the visualizations the different kinds of tangling that were likely to exist within the component, and this guided our investigation and refactoring efforts. The four extremes of visualization are depicted in Figure 2 through Figure 5. Figure 2 below shows the ideal situation – there are no links between the selected component and the EJB support concern.

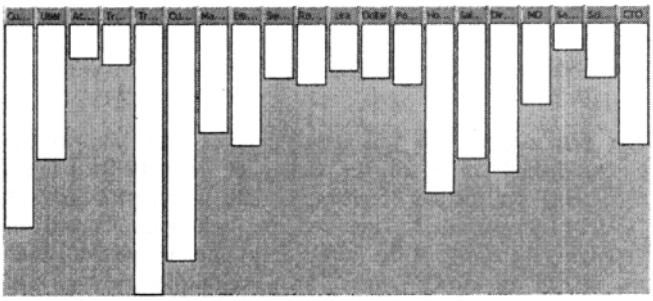

Figure 2. Separated Component

Figure 3 is an indication that a homogeneous crosscutting concern exists, and we should look to refactor it into an aspect. Each bar represents a source file in the component under investigation, and each red line a link from that component to the EJB support concern.

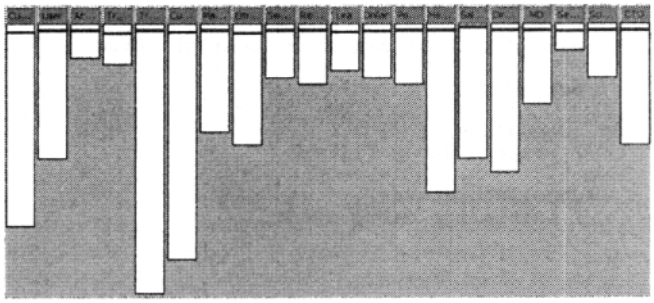

Figure 3. Homogeneous Crosscutting Concern

Figure 4 is an indication that within the component, there are a small number of classes largely dedicated to the EJB support concern. We may consider changing the concern definition to include them.

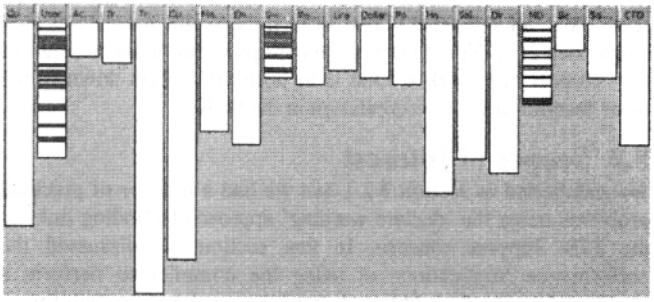

Figure 4. Dedicated Classes

Figure 5 either means that there's trouble ahead, or that perhaps this whole component should be considered part of the EJB support concern.

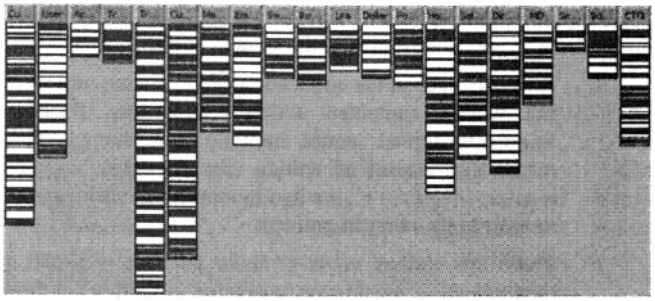

Figure 5. Here be Dragons

3.3.4 Nature of Language Constructs Used

Compared to the typical examples of AspectJ usage for modularizing crosscutting concerns, our use of AspectJ constructs was heavily skewed towards inter-type declarations. 74% of the declarations inside our aspects were inter-type declarations, and only 26% advice. Each piece of advice affected only a single joinpoint. Contrast this to the homogeneous concerns where we used advice much more frequently than inter-type declarations, and a single piece of advice may affect thousands of joinpoints. Of the inter-type declarations that we used in separating the EJB support concern, 92% affected only a single target type. The maximum number of types affected by a single introduction (through a `declare parents : implements` statement) was six. We used the `declare extends` and `declare implements` constructs (excluding use for targeting inter-type declarations at multiple types) a couple of times each. These numbers demonstrate the quite different nature of feature-based composition for heterogeneous concerns.

We compared the AspectJ implementation of some of these aspects to an equivalent solution using a variant of open class composition that we envision will be supported by tools in the CME family. This capability is a simplified derivative of Hyper/J's merge-by-name composition [16]. Using the example from section 3.3.1, the refactoring would start in the same way as with AspectJ, with the ExtractMethod move to extract the `addCollaborators` method. The next step would remove the EJB collaborator registration from the `DebugAppServerCI` as before, but in place of the aspect we would simply declare:

```
public class DebugAppServerCI
  extends ComponentImpl {

  private EJBContainer ejbContainer;

  void addCollaborator() {
    // register ejb collaborator…
  }
}
```

When multiple class definitions with the same name are encountered, the class fragments defined by each are simply merged. Each class fragment must be defined in a different concern (the class fragment above would be in the EJB support concern). Where methods with the same name and signature are defined in multiple class fragments, simple precedence determines the order in which these method fragments will be invoked. (In addition to an ordered invocation of method fragments, other supported options will include invoking the highest precedence fragment only, or disallowing method fragmentation altogether). From our work on refactoring the EJB support concern we believe these simple variants of open class composition can complement AspectJ in separating heterogeneous concerns. We return to this thought in the conclusion.

4. SCALABILITY AND INTEGRATION

4.1 Build Integration

AspectJ was easy to integrate into the product-line build, and both compilation and binary weaving times were acceptable. We found the AspectJ 1.1 compiler to be robust in use.

Integrating AspectJ into the ant-based builds used by members of the product-line was made a very easy task due to the plug-compatible nature of the AspectJ ant tasks with their Java™ equivalents. We performed complete compilations of product-line members using the AspectJ 1.1 compiler, and also used the aspect library capability of AspectJ 1.1 to compose aspects from a library with pre-built application source files as an additional stage in the build process. In total we have now compiled over 20,000 source files from members of the product-line. During this process, and whilst experimenting with numerous aspects and building options, we found two bugs in the AspectJ compiler. Both of these are fixed in AspectJ 1.1.1. In a benchmark compilation of one product-line member comprising about 3,000 source files, we found that with no aspects included in the system, the AspectJ compiler was 10.5% faster than javac – this is a credit to the Eclipse Java compiler on which AspectJ is based. Adding very pervasive aspects into the build (affecting nearly every source file) increased the batch compilation time by 30-50%. This overhead was acceptable for the product from which we took these benchmarks, amounting to less than a minute increase in overall build time. A complete build consists of multiple stages, including compilation but also packaging, testing, moving and copying resources etc.. The actual time spent in compilation is a relatively small proportion of the overall build to start with, which is why adding overhead to just this phase does not have a great impact.

4.2 Development Tools Integration

The AspectJ Development Tools (AJDT) support for Eclipse was easy to use with our existing Eclipse-based development tools and plugins. Some work on performance and memory usage is still needed for large projects. For AJDT to displace the Eclipse Java Development Tools as the primary working environment for general development, better integration is needed with the Java views and features in Eclipse.

Development of components for the application server used in the EJB support study is supported by Eclipse plugins that enable

components of the application server to be easily imported and worked on as individual Java projects in the Eclipse workspace. It was trivial to convert these projects to AspectJ projects (a single menu selection from the project context menu). For larger projects the compilation (weave) time during a project build was sometimes noticeably slower than the regular Java compilation of the same project without aspects. This was due to the additional time taken to build the structure model used by the AspectJ views. (The structure model is not generated during batch builds outside of Eclipse). Memory usage was also an issue with larger projects. Since this time, the AspectJ 1.1.1 release has improved both the time taken to generate structure model information, and also the memory usage. These improvements are exploited in the 1.1.4 release of AJDT. However, performance and scalability remains a focus item in both the AspectJ and AJDT project plans as users increasingly employ AspectJ on large-scale development projects.

We did not roll out the AJDT tools to large numbers of developers as part of this study. Before AJDT can be fully embraced as the everyday development environment of the hundreds of developers working on the product-line, more work is needed to ensure that the Java Development Tools (JDT) features that developers rely on in Eclipse continue to work. The key items are completing structure model support for incremental compilation, eager parsing to ensure that the outline updates as the user types rather than waiting for compilation, full debugging (needs JSR 45 support [17]), and support for code completion, code formatting and 'organizing imports' in source files declaring aspects. These are all items in the AJDT 2.0 development plan, and scheduled for early 2004. Note that for many modes of operation, it is not necessary for every developer to be using AJDT, and many can continue quite happily to work with the JDT. We characterize the current AJDT support as excellent for early adopters, but not yet up to the standard needed to make mass penetration into the early majority market.

It should be noted that the authors are also committers on the AspectJ and AJDT projects.

5. FUTURE REQUIREMENTS

The previous section touched on some detailed requirements for AJDT. In this section we discuss requirements emerging from the work that have a broader impact. These include extending existing support for binary weaving, an abstraction and composition mechanism for dealing with scopes of interest, and synthesis of concern modeling support with programming level techniques.

5.1 Multi-phase weave

We used AspectJ both for component compilation, and for the application of aspect libraries to pre-built components. It is easy to envisage situations in which we will want to do both of these things. For example, a developer may use an inner aspect to modularize a crosscutting concern within a single class during component development. We may then want to apply product-line quality of service aspects across the board at a later stage in the build using an aspect library. The current implementation of the AspectJ compiler precludes this usage: it is not possible to link (weave) aspects into class files that have already been linked (woven) with (other) aspects before. This requirement was also the subject of a discussion on the aspectj-users mailing list. If multi-phase linking is supported, the semantics should be the same as if all the aspects had been linked with the classes in a single step [18]. Note that this is a hard problem. In the absence of this support, the work-around is to push all aspects down to the most granular level of compilation in the build.

5.2 Scopes of Interest

We mentioned in section 3.2.1 that we had a number of practical problems using the 'declare warning' approach to finding links to the EJB Support concern. In that section we discussed the performance implications of using the compiler to perform a search-only task. An additional problem was the manageability of the aspect declaration itself. A mechanism to name and compose scopes, and then refer to them in pointcut expressions, would have resolved this latter problem.

Before we switched to using conman based queries, the EJBSupportSeparation aspect had grown to over 2,000 source lines. Three factors contributed to the growth in size:

1. Java packages in the application server source tree were not wholly contained within components. Package fragments spread across multiple components. This meant that instead of writing (for example) '`call(* a.b.c..*.*(..))`', we had to enumerate each type of interest in its own call pointcut.
2. There are various kinds of links that are possible: a method call, a constructor invocation, a field get, a field set, subclassing, and interface implementation. For each of the first four kinds of link listed, we had to duplicate the (long) list of types to look for. We didn't write pointcut expressions to detect subclassing or interface implementation (because it was getting too unwieldy), although this is possible.
3. It was not possible to use the simplifying strategy of using `declare parents` to add a marker interface to all EJB Support concern types and then looking for any call where `target(EJBSupport)`, since the target pointcut designator is not fully statically determinable and hence not supported in declare error / warning statements.

For this reason, we wanted to be able to define a scope representing the types in the EJB support concern, and then refer to that scope by name when writing pointcuts. We also believe that the added expressive power that comes from being able to name and compose scopes can capture intent more precisely, and improve the clarity of the source code. We'd like to be able to write something like this (the grammar used is solely to express the concept, not a concrete proposal):

```
public scope EJBSupport :
   packages(a.b.c..*) ||
   types( T1 || T2 || T3 || … );
```

With the scope defined, we could use it in pointcut expressions wherever a type pattern is supported. For example:

```
call(* EJBSupport.*(..))
execution(* EJBSupport.*(..))
```

would match any call to (or execution of) a method in the `EJBSupport` concern.

```
get(* EJBSupport.*)
set(* EJBSupport.*)
```

would match any get (or set) of a field defined in the EJBSupport concern, and so on.

Naming scopes in this way helps to raise the level of abstraction. We also propose to permit scope composition. For example:

```
public scope DebugEJB : EJBSupport && Debug;
```

defines the DebugEJB scope as the intersection of EJBSupport and Debug. The '!' and '||' operators would also be permitted, as used in pointcut composition also. In addition to the packages and types scope specifiers, method and field selection should be possible. For example,

```
public scope Debug :
  packages(com.xyz.debug..*) ||
  methods(* debug*(..));
```

declares the Debug scope to be all types in the com.xyz.debug package hierarchy, plus any method beginning with "debug" wherever it is defined.

With a mechanism similar to that outlined in this section, we would have been able to write:

```
declare warning :
  call(* EJBSupport.*(..)) &&
  !within(EJBSupport) :
  "Link to EJBSupport concern found.";
```

Stepping back a little, we see scopes (there's probably a better name) as providing naming (abstraction) and composition for static collections in the program. This complements the pointcut mechanism, which provides abstraction and composition for dynamic collections (of joinpoints). The proposal is in line with Cristina Lopes' suggestion to focus on improving the set of referencing forms available to programmers [19].

The extensive use of marker interfaces in AspectJ programs to emulate named scopes indicates that such a feature could have wide exploitation. AspectJ pointcut support for joinpoint matching based on metadata attributes (JSR 175 [20]) will also come close, but would still require an artificial 'declare attribute' for the idiom to work.

5.3 Integration of Concern Modeling

Observe the strong correlation of scopes as presented in the previous section, to concerns and the activity of concern modeling [10]. We illustrated the value of concern modeling during development in section 3.2. Tools such as FEAT [21] and the Concern Explorer from the CME support concern modeling and discovery. We believe integration between tools like these, and the programming language used to implement the concerns, will be a powerful facility. This could take the form of 'sending' a concern definition to a scope declaration, or even having scope declarations automatically created and updated as concerns are defined and updated. Likewise scope declarations could be promoted to concerns in the model.

6. RELATED WORK

Zhang and Jacobsen [22, 23] have undertaken aspect-oriented refactoring in a selection of ORBs, "we believe that middleware architecture is one of the ideal places where we can apply aspect-oriented programming to obtain a modularity level that is unattainable via traditional programming techniques." Their results indicate that they were able to significantly reduce the complexity of the ORB architecture whilst retaining the same performance levels. Their more recent work [24] suggests a design approach for tackling feature convolution in middleware through the identification of a minimal core, on top of which aspects are used to implement additional features. The work of Zhang and Jacobsen is closest to our own. Whereas they focused on smaller systems in greater depth, the product-line we used as a basis for our study, and the concerns that we separated, are an order of magnitude larger. We also concern ourselves with the integration of the AOSD techniques into the organization and its development processes.

Hunleth et al. have studied the use of AspectJ for feature and footprint management in a CORBA event channel, using aspects to introduce features incrementally and as independently as possible [25, 26].

Yvonne Coady and Gregor Kiczales undertook a retrospective refactoring on portions of the FreeBSD operating system using AspectC [27], and then replayed actual changes that had occurred in the system code to see how the aspect-oriented design fared under the same modifications. They found that with the aspect-oriented design changes were better encapsulated, redundancy was reduced, the creation of alternate system configurations became easier, and extensibility aligned with an aspect was more modular.

The heterogeneous nature of the EJB Support concern, and the approach we ended up using to separate it, bear a strong resemblance to Don Batory's work on Feature-Oriented Programming and the GenVoca design methodology [28]. GenVoca uses a process of refinement to add features to a program, and equations are used to define program family members. A feature of the work is the specification of a simple algebraic model for composition.

The concern modeling approach used to separate the EJB support concern builds on the work in Multi-Dimensional Separation of Concerns by Ossher & Tarr [16].

7. CONCLUSIONS

The results of this study, coupled with those from the related research cited in the previous section, draw us to the conclusion that AOSD can and will play an important role in tackling complexity not just in middleware but in large software systems of all kinds. We believe that AspectJ can scale to the large sizes of commercial middleware projects, and that it can be integrated into existing development processes. Both the build and run-time performance appear to be satisfactory. Enhancements to the AspectJ compiler as described in section 2.1 will resolve any remaining performance issues. The AJDT developer tools still need more work before they will be accepted for everyday usage by mainstream developers. The work items are well understood and are a focus of the AJDT plan for the next release.

We did not deliberately set out to pick concerns we knew would have homogeneous and heterogeneous characteristics. We simply picked some that looked to be typical of the kind of examples often quoted for aspect-orientation (especially AspectJ), and one that we knew would be difficult to separate (as a test for the approach). The different language feature usage patterns when separating the two kinds of concerns clearly illustrate that these represent two ends of a spectrum. We believe that the results demonstrate a need for both broadly crosscutting aspects, and a more feature-oriented style for feature composition. Both kinds of concerns occur simultaneously in the same systems, so we need an approach that makes both easy to address. We used AspectJ

across the board and it worked well. But we also used a lot of 'home-grown' solutions to integrate concern modeling into our process. We believe the style of feature composition illustrated in section 3.3.4 may present a gentler learning curve than the AspectJ equivalent in the common case of 1:1 inter-type declarations and advice used for feature integration. To exploit this, we need an environment in which both types of composition can be freely used together, ideally integrated with a concern modeling and query capability. One of the goals of the CME is to provide such an environment.

We have made some suggestions that we believe will aid in the future application of AspectJ and AOSD in large systems. Even as things stand today though, we are happy to recommend that adoption proceed cautiously in commercial projects.

8. ACKNOWLEDGEMENTS

Thanks to Harold Ossher, Peri Tarr, and Bill Harrison for their advice, insight and encouragement as we embarked on this project, and also for the excellent support they gave us whilst working with the conman concern modeling and query engine.

9. REFERENCES

[1] F. Kon, F. Costa, G. Blair, and R. H. Campbell, "The Case for Reflective Middleware," *Communications of the ACM*, vol. 45, pp. 33-38, 2002.

[2] M. Roman, F. Kon, and R. H. Campbell, "Reflective Middleware: From Your Desk to Your Hand," *IEEE Distributed Systems Online*, vol. 2, 2001.

[3] C. Zhang and H.-A. Jacobsen, "Refactoring Middleware With Aspects," *IEEE Transactions on Parallel and Distributed Systems*, vol. 14, 2003.

[4] R. Bodkin, A. Colyer, and J. Hugunin, "Applying AOP for Middleware Platform Independence," *Practitioner Reports, 2nd International Conference on AOSD*, 2003.

[5] A. Colyer, A. Clement, R. Bodkin, and J. Hugunin, "Using AspectJ for Component Integration in Middleware," *Practitioner Report, OOPSLA 2003*, 2003.

[6] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, "Getting Started with AspectJ," *Comm. ACM*, vol. 44, pp. 59--65, 2001.

[7] The AspectJ Team, "New pertype aspect specifier, AspectJ 1.1 Readme.," 2002.

[8] Apache Software Foundation, "Welcome to WSIF: Web Services Invocation Framework." http://ws.apache.org/wsif/.

[9] D. L. Parnas, "On the Criteria to be Used in Decomposing Systems into Modules," in *Communications of the ACM*, vol. 14, 1972, pp. 221-227.

[10] H. Ossher and P. Tarr, "Multi-dimensional Separation of Concerns and the Hyperspace Approach," in *Software Archictecture and Component Technology*, M. Ahksit, Ed.: Kluwer, 2002, pp. 293-323.

[11] "Eclipse.org - Main Page." http://www.eclipse.org.

[12] "Eclipse AspectJ Development Tools project." http://www.eclipse.org/ajdt.

[13] H. Ossher, P. Tarr, and W. Harrison, "Concern Manipulation Environment (CME): A Flexible, Extensible, Interoperable Environment for AOSD." http://www.research.ibm.com/cme/, 2003.

[14] M. Fowler, *Refactoring: Improving the Design of Existing Code*: Addison-Wesley, 1999.

[15] S. G. Eick, J. L. Steffen, and E. E. Sumner, "Seesoft - A Tool For Visualizing Line Oriented Software Statistics," *IEEE Transactions on Software Engineering*, vol. 18, 1992.

[16] H. Ossher and P. Tarr, "Using Multidimensional Separation of Concerns to (re)shape Evolving Software," *Communications of the ACM*, vol. 44, pp. 43-49, 2001.

[17] Sun Microsystems Inc., "JSR 45: Debugging Support for Other Languages." http://www.jcp.org/en/jsr/detail?id=45.

[18] G. Kiczales, "RE: [aspectj-users] SCM & AspectJ." post to aspectj-users@eclipse.org, 2003.

[19] C. V. Lopes, "Aspect-Oriented Programming: An Historical Perspective (What's in a Name?)," Institute for Software Research, University of California, Irvine UCI-ISR-02-5, December 2002 2002.

[20] S. M. Inc., "JSR 175: A Metadata Facilicy for the Java Programming Language." http://www.jcp.org/en/jsr/detail?id=175.

[21] M. Robillard, "FEAT: A tool for locating, describing, and analyzing concerns in source code." http://www.cs.ubc.ca/labs/spl/projects/feat/.

[22] C. Zhang and H.-A. Jacobsen, "Quantifying Aspects in Middleware Platforms," *Proceedings 2nd International Conference on Aspect-Oriented Software Development*, pp. 130-139, 2003.

[23] C. Zhang and H.-A. Jacobsen, "Re-factoring Middleware Systems: A Case Study," *Distributed Objects and Applications (DOA)*, 2003.

[24] C. Zhang and H.-A. Jacobsen, "Resolving Feature Convolution using Horizontal Decomposition in Middleware Systems," *University of Toronto, Computer Systems Research Group Technical Report Nr: 475*, 2003.

[25] F. Hunleth and R. Cytron, "Footprint and Feature Management Using Aspect Oriented Programming Techniques," presented at LCTES 02, Berlin, Germany, 2002.

[26] F. Hunleth, R. Cytron, and C. Gill, "Building Customizable Middleware using Aspect Oriented Programming," presented at OOPSLA 2001 Workshop on Advanced Separation of Concerns in Object-Oriented Systems, Tampa, Florida, 2001.

[27] Y. Coady and G. Kiczales, "Back to the Future: A Retroactive Study of Aspect Evolution in Operating System Code," *Proceedings 2nd International Conference on Aspect-Oriented Software Development*, pp. 50-59, 2003.

[28] D. Batory and B. J. Geraci, "Composition Validation and Subjectivity in GenVoca Generators," *IEEE Transactions on Software Engineering*, pp. 67-82, Feb. 1997.