Using AspectJ for Component Integration in Middleware

Adrian Colyer, Andy Clement
IBM UK Limited
Hursley Park
Winchester, England
+44 1962 810000
{colyer,clemas}@uk.ibm.com

Ron Bodkin
New Aspects of Security[i]
216 27th Street
San Francisco, CA 94131
+1 415-509-2895
rbodkin@newaspects.com

Jim Hugunin
Palo Alto Research Center
3333 Coyote Hill Road
Palo Alto, CA 94303
+1 650-812-5000
hugunin@parc.com

ABSTRACT
This report discusses experiences applying AspectJ [1] to modularize crosscutting concerns in a middleware product line at IBM®. Aspect oriented programming techniques were used to cleanly separate platform specific facilities for aspects such as error handling, performance monitoring and logging from base components, permitting those components to be reused in multiple environments. The initiative also guided the design of the AspectJ Development Tools (AJDT) for Eclipse, and influenced the technical direction of the AspectJ implementation

Keywords
Aspect-orientation, AspectJ, middleware.

1. INTRODUCTION
In March of 2002 a project team consisting of three consultants from PARC and six IBM employees undertook a study to investigate the potential for separating crosscutting concerns from middleware components. The concerns investigated were chosen both because they represented classic early use cases for AOP [2] (and hence were a good test to see if the claims for AOP could stand up in a real code base, as opposed to an educational example), and because project success would solve a real business need. This report describes the initial experiences from that project, and the subsequent development of the ideas and tools in the 18 months since then.

The business motivation for using AspectJ was to target multiple runtime environments with a single source code base. The IBM team was releasing certain components under an open source license, such that they could be used in open source environments. However, the same components were also used in an IBM middleware product line where it was important to continue to take advantage of improved platform-specific facilities. Putting IBM proprietary features into the open source code base was unacceptable both to IBM and to the open source community. Because of the pervasive nature of the concerns addressed, maintaining dual code bases would have been time consuming and error prone.

OOPSLA'03, October 26–30, 2003, Anaheim, California, USA.
ACM 1-58113-751-6/03/0010.

The most important concerns addressed were tracing and logging, event reporting, error handling, and performance monitoring. Figure 1 gives an indication of the pervasiveness of these concerns in the middleware product line. It presents an analysis of some of the components in the product line, showing how many other components directly depend on them. Existing IBM policy documents in each of these areas were interpreted and embodied in aspects with no concessions made to make the task easier. The team also assessed the organizational and architectural impacts of the aspect based solution, and the ability of the AspectJ tools to scale to an industrial setting.

The initial process involved two weeks of remote collaboration, including code reviews, preliminary design, and other preparation. This was followed by an intense week of hands on training and workshops that accomplished the following:

- A review of the design of the pilot components
- Analysis of the specific concerns under study
- Interactive design of new aspects to address those concerns
- Rapid prototyping of a solution, modifying production code using AspectJ 1.0.3
- Integration of the AspectJ tool set into production build processes
- Integration of prototype code into a deployable format
- Analysis of findings

At the time the study was initiated, four out of six of the IBM employees had no prior experience with AspectJ.

2. IMPLEMENTATION
This section describes how aspects were used to address the various concerns, together with an assessment of the benefits and drawbacks of the AspectJ based solution.

2.1 Tracing and Logging
All components in the product line have extensive logging requirements. The product architecture team defines a detailed policy (about a fifty page document), which has been revised with each major release of the application. There are two major applications of logging: for tracing method entries and exits,

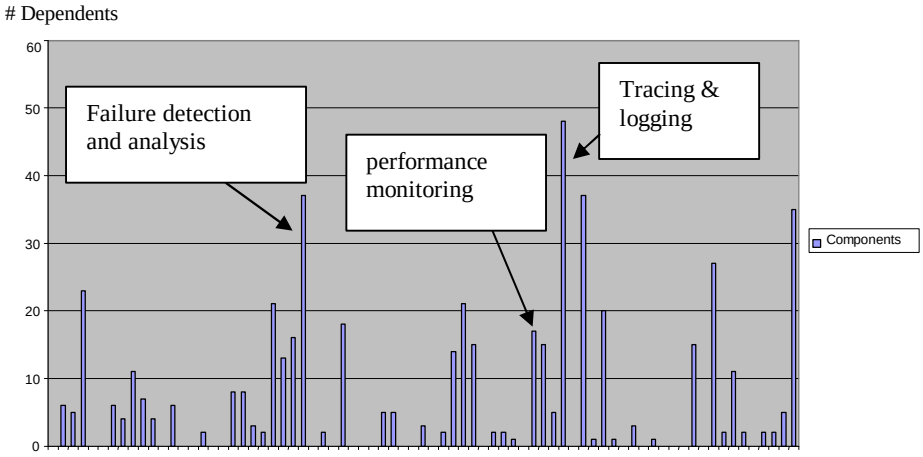

Figure 1. Most frequently used components

and for recording system events. Error handling also performs a type of logging, but that is handled through a separate infrastructure (and is further described in section 2.2).

The tracing policy was implemented using an abstract aspect that captured the policy, and concrete sub-aspects that defined the scope of policy application. The single abstract aspect defined both when and how tracing was to be performed. It also ensured that all calls to tracing were "guarded" with a check that determined whether tracing is enabled. One concrete sub-aspect was defined for each component to be traced in the product-line. This division of responsibility allows the architects to both set *and implement* a global policy, whilst component owners decide how that policy should be applied within their domain. A typical component aspect provides a concrete definition of a scope pointcut indicating where the trace policy should be applied within the component, and also a set of inter-type declarations for toTraceString() methods that override default logging output where necessary.

We were able to successfully implement the tracing policy using AspectJ. Furthermore, considering the accuracy and completeness of the implementation, the aspect-based solution compared favourably to the traditional scattered implementation of inserting trace calls into every class. In creating the prototype, the team 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 (by making calls that create strings needlessly). In subsequent work with aspect implementations of policy we have found this experience to be entirely typical – specifying and implementing the policy directly in an aspect gives a more complete and accurate implementation. The larger the development team (and hence the greater the degree of separation between the team specifying the policy and the developers who would have to implement it in a traditional manner), the greater the benefit of the aspect approach.

A further benefit of the aspect approach that has been subsequently exploited is the ease with which the tracing implementation can be updated or replaced. We have implemented tracing policies using the product-line internal interfaces, Log4j [3], and Jakarta Commons Logging [4]. Switching implementation is a simple build time decision of which implementation aspect to include. Clearly, the larger the code-base, the greater the cost of switching or updating a traditional scattered implementation - and therefore the greater the benefit of an aspect-based solution.

Our initial implementation of the tracing concern was based on AspectJ v1.03 and exposed the importance of optimized performance. We were concerned that the aspect based solution may impose an estimated performance overhead of up to 5%. The cause for this concern was the eager creation of 'thisJoinPoint' objects for advice that referenced 'thisJoinPoint' in the advice body, even if dynamically the 'thisJoinPoint' variable would never be accessed (tracing was turned off by a flag for example). We subsequently ran some performance tests with AspectJ 1.1, and measured the overhead in the case where tracing is disabled (the performance sensitive case) *and advice does not use 'thisJoinPoint'*, at 1% or below. When advice does use 'thisJoinPoint' the overhead caused by the additional object creations and also garbage collection can be significant, depending on the profile of the application. The lazy creation of 'thisJoinPoint' objects is a candidate item for the AspectJ 1.2 release and will resolve this issue fully. Also bear in mind that a truer 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.

The default tracing facility used by the middleware product line required classes to register once with the tracing facility, and to use a returned object for all future tracing. This registration is done during static initialization of the class. By convention, the name used for identifying this tracing is the name of the class. However, AspectJ did not support any means of identifying the class in which a static method is executing. This support would be important for statically initializing tracing for many classes from the same aspect. The current version of AspectJ (v1.1) does not support inter-type declarations of static members across multiple classes. The proposed 'pertype' language extension [5] will resolve this issue. These issues have been partially resolved on subsequent projects by tailoring the tracing policy to better match AspectJ's capabilities (by registering at the component, rather than class level). An alternative implementation could use HashMaps, but this solution has performance implications.

Systematic logging for capturing events was prototyped with good results. In this case, an aspect was created for each component that defined pointcuts for the events to be logged. In some cases, this required minor refactoring of the code to expose a joinpoint of interest. In general we have found that a refactoring required to facilitate an aspect-oriented approach to some concern improves the quality of the code independently of the support for the concern in question. A corollary to this is that the better the object-oriented design of the application, the easier it is to introduce aspect-orientation.

We anticipate that the aspect-oriented approach to tracing and event logging will make the serviceability reviews that IBM holds prior to product shipment easier: the tracing policy and set of logged events are captured explicitly for review. A concern with event logging though (as opposed to general tracing that tends to use robust property-based pointcuts) was the potential fragility of the event pointcuts in the face of program maintenance by development prior to shipment, or service afterwards. If code is refactored, this has the potential to cause the event pointcuts to no longer match as intended. One approach to mitigating this concern is using the AspectJ development tools such as AJDT that visually show where advice applies to given code. Another could be to extend AspectJ to allow declaring warnings or errors if the events are no longer present (i.e., the pointcuts are empty). A longer-term solution is to integrate pointcut definitions into refactoring tools, and rely on these tools to correctly refactor all elements of a program. The infrastructure needed to support this approach is now being developed as part of the Eclipse 3.0 release, and the AJDT project plans to exploit this to allow aspect-aware refactoring [6].

A final significant benefit of applying AspectJ to tracing and logging came from writing an aspect that policed improper usage: it generated compile-time errors when the user wrote results to System.out or System.err, or code that otherwise used the logging facility improperly. This policing aspect found several policy violations in one of the components. We have subsequently significantly extended our use of development-time policy enforcement aspects within the product-line.

2.2 Error Handling

The product line uses a sophisticated error analysis and reporting subsystem following the principle of "first-failure data capture" (FFDC). In essence 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.

When the FFDC capability was first introduced into large portions of the product-line a hand-built tool was used that rewrote source code. Another tool was created and maintained to test for violations of policy (including checking for comments to indicate that an exception should not be reported). New code needs to be manually instrumented. The tool has limited flexibility, and automates the process only for the initial introduction of error handling logic. However, the pain of handling the crosscutting error handling concern accurately made it better to introduce special purpose tools than try to enforce coding discipline without tools.

By contrast, it was easy and effective to implement the error handling policy in AspectJ. An abstract aspect was again developed to codify the error handling policy. This captured the points where errors were detected (in exception handlers and in method returns), and passed the exception details and context information into the FFDC analysis engine. In addition to the abstract aspect, the prototype included one concrete sub-aspect for each component following the same principle as that outlined for tracing and logging. Here the component aspects also needed to define any exceptions that should not be dealt with by error handling.

There was initially concern about pointcut fragility in determining where exceptions were being handled that shouldn't be passed to the error analysis and reporting subsystem. However, close analysis showed that there was always a principle behind which exceptions and in which context exceptions weren't analyzed and reported. So the pointcuts that excluded handling certain errors dealt mostly with classes of exception and domain classes, and did not need to enumerate lists of methods or combinations of methods and exceptions.

An example of a common case that needed to be excluded from the exception handling logic was all calls to 'Class.forName(xxx)'. This method throws a 'ClassNotFoundException' to indicate a missing class. Every time this method was used in the code base the exception was treated as a normal return value and handled at the call site. The reusable aspect was able to capture this pattern in a general way and remove the need to hand-label each call-site which the current hand-built tool requires.

The AspectJ solution was not only consistent in applying policy and making it explicit, but it also made it easier to change the policy in the future. Subsequent work has extended the set of FFDC capabilities handled by the aspect implementation. We have used aspects to implement and register component diagnostic modules that can provide component state information to the FFDC analysis engine on request. We have also prototyped an aspect approach to capturing important

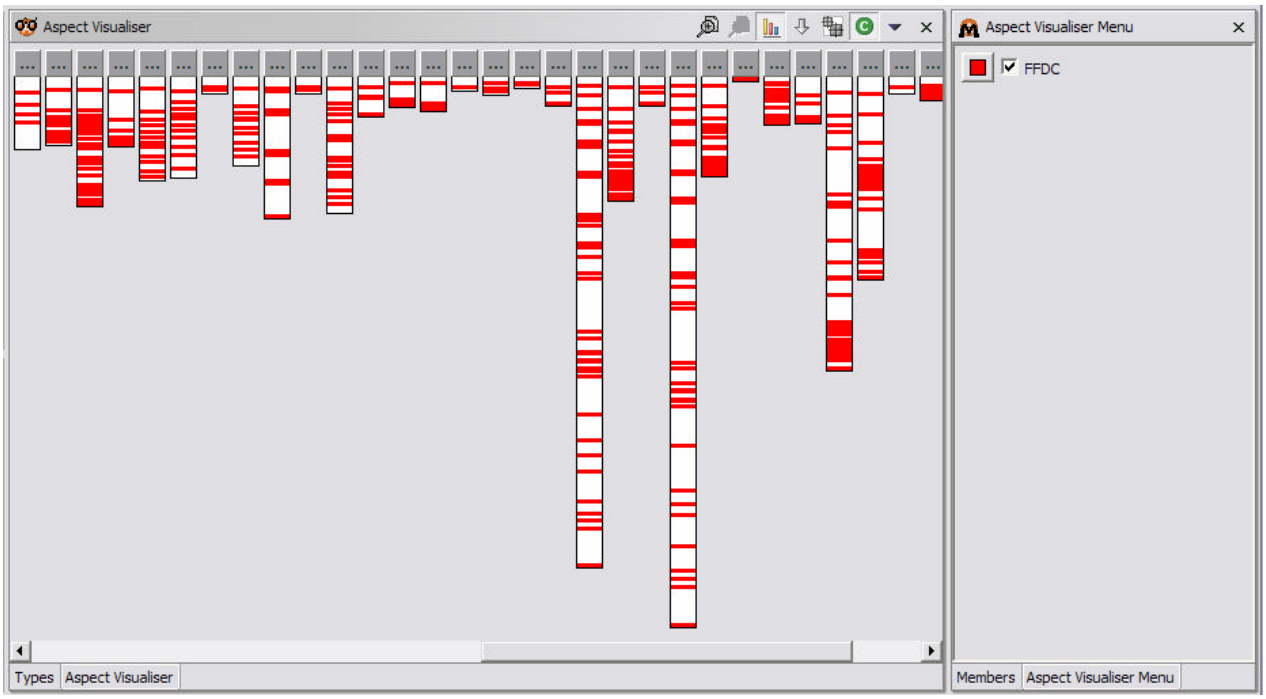

Figure 2. Visualization of the Impact of the FFDC Aspect

transient data not available on the call-stack and making that available to the FFDC engine in the case of failure too.

Figure 2 illustrates the impact of the FFDC policy as implemented in AspectJ for one of the components in the product-line. Each bar in the view represents a source file in the component. Every red line represents a place where the types defined in the source file are advised by the FFDC aspect.

This visualization capability is part of the AspectJ Development Tools (AJDT) for Eclipse developed subsequently to the original prototyping work. We have since found it to be very effective in persuading developers of the power of the aspect-based solution.

2.3 Performance Monitoring

This product-line is extensively instrumented to capture performance information (monitoring and statistics data). Components provide a class with stylized interfaces to access performance statistics for the component. We initially analyzed the existing implementation of performance monitoring data collection in a significant component of the product line. We found that data gathering 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 performance instrumentation concern for the component in a single aspect that applied a consistent policy for capturing the performance statistics. The team was especially pleased with the aspect implementation of this concern, since each statistic to be gathered mapped neatly into a single pointcut definition, making the code look just like the design document! Moreover, the original code had to manage state in multiple places just to count correctly. In contrast, the AspectJ version was able to centralize this logic and disentangle it from the core component logic.

We were easily able to generalize the approach for a second component with comparable convenience and further reduced effort. Figure 3 shows the impact of the performance monitoring aspect for this component. The view has been limited to show only affected classes – which are a small subset of the total for the component. Overall, statistics collection for performance monitoring was significantly improved by using AspectJ.

2.4 Impact on Program Comprehension

A common question that comes up when discussing aspect-oriented programs is that of program comprehension. Isn't it harder to understand what's going on in the system when multiple aspects are in place? Our experience was to the contrary; the aspect solution improved overall program understanding by making the cross-cutting policies explicit, and by removing tangling (noise) from other routines so that their intended function could be more readily seen. The following example illustrates this effect on a selected source extract.

```
01 try {
02  if (!removed)
03    entityBean.ejbPassivate( );
04  setState( POOLED );
05 } catch ( RemoteException ex ) {
06   FFDCFilter.processException(
07    ex,"EntityBeanO.passivate()",
08    "237",this);
09  destroy( );
10  throw ex;
```

```
11 } finally {
12  if ( !removed && pmiBean != null )
13    pmiBean.beanPassivated( );
14  removed = false;
15  beanPool.put( this);
16  if ( EJSDebug.EJSDEBUG)
17    Tr.exit( tc, “passivate” );
18 }
```

Lines 06-08, 12-13, and 16-18 are all arising from tangled concerns. The sample below shows the same extract, but this time with the crosscutting concerns refactored into aspects.

```
01 try {
02   if (!removed)
03     entityBean.ejbPassivate( );
04   setState( POOLED );
05 } catch ( RemoteException ex ){
06   destroy( );
07   throw ex;
08 } finally {
09   removed = false;
10   beanPool.put( this);
11 }
```

2.5 Additional Concerns

During the workshop the team also did preliminary prototyping and achieved good results in separating the definition of business events from source code. This was fairly analogous to defining events for logging purposes (as described in section 2.1). However, the pointcuts used were also able to support events in customer (3rd party)-written code by supporting a naming pattern (or customer defined pointcuts).

Subsequent work has used aspects to instrument components for management via JMX™. The aspects permit the addition of management operations to an existing class, and adaptation of fields and methods for management. An investigation into the use of aspects for profiling has reported on in [7].In addition to these very homogenous concerns, the IBM team has also investigated the use of AOSD to refactor large scale heterogeneous concerns in the product line, and this work will be reported on separately.

AspectJ was helpful as a debugging tool throughout the prototyping effort. In addition, one attendee of the training tutorial who was not part of the prototyping effort immediately applied AspectJ to debugging a distributed system. The aspect reduced the time required to solve the problem because it did not require invasive modification of code to identify what was wrong.

3. TOOLS INTEGRATION AND ASSESSMENT

This section discusses how adding AspectJ to the existing system affected integration with the project’s development tools and process, and how the AspectJ tools themselves stood up to the test.

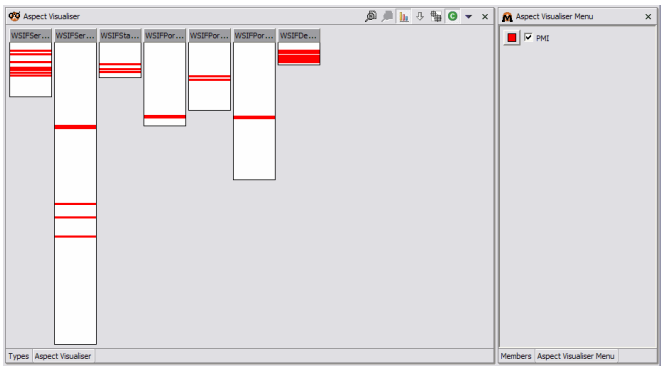

Figure 3. Implementation of Performance Monitoring Concern

The project team already had a very heterogeneous set of tools (including almost as many favored editing environments as there were people prototyping). Most developers on the team preferred to use command-line compilation. The combination of Eclipse integration, emacs integration, and the stand-alone browser tool supported everyone’s preferred development approach. (Subsequent to the initial workshop, there has been a significant migration to Eclipse-based tools). The team initially worked with an alpha version of the AJDT toolkit for Eclipse. This was helpful for understanding the effect of crosscutting declarations, but was hard to use because it was not yet a mature tool. The current release of AJDT is a vast improvement on that early tool, and the IBM team now uses it on a daily basis for their work.

The AspectJ 1.03 compiler worked quite well on the code base: it was easy to compile existing code, add aspects to it, and to test it. The product-line uses a sophisticated set of ant scripts, including a custom ant task for compilation, and maintains separate files that define the classes in each component. However, in about one person day of effort the team was able to integrate AspectJ compilation into the process completely.

The biggest drawback in the resulting build process resulted from how it handled reusable aspects in multiple components. AspectJ 1.0 did not provide a means for packaging a reusable library of aspects, so reusable aspects needed to be included as source in the definition of each component to which they applied. AspectJ 1.1 addresses this issue by allowing aspect libraries to be built and by supporting binary weaving. A secondary issue was the compilation time, both in batch builds and within the IDE. Part of this was attributed to the batch compiler implementation, and part to the lack of incremental compilation for AspectJ, which made compilation during development feel slow, though it still remained tolerable. Incremental compilation was addressed in AspectJ 1.1 and is now supported by AJDT. AspectJ 1.1 also switched the compiler implementation to be based on the Eclipse JDT compiler. Our latest measurements with the AspectJ 1.1 release indicate that compilation is now quicker than with the standard javac compiler (a benchmark compile of nearly 3,000 classes showed that ajc gave a 10.5% reduction in compile time over javac). We have also found the AspectJ 1.1 compiler implementation to be very robust, having test compiled almost 20,000 source files from the middleware product line and found

only two bugs (both now fixed). The ant support in AspectJ has also improved, and we have trivially integrated the AspectJ 1.1 compiler into the build process of other products in the family.

During the week of prototyping, the team had a good opportunity to assess the quality of the AspectJ 1.0.3 compiler's error messages. The consensus was that the error messages were good for compiling pure Java™ code, but needed improvement when AspectJ-specific problems occurred. In practice, even the most confusing error messages weren't a problem on this project because one of the AspectJ compiler writers was present to translate any odd messages. However, it was clear that improving these messages would be important for teams without this sort of on-site consulting. The clearest lesson learned from error handling was that having the compiler signal as many errors as possible was extremely helpful. All of the developers on the team used the 1.0 compiler's –Xlint options to get the most possible warnings and the only complaint with this was that it didn't indicate more problems. As a result of this experience, AspectJ 1.1 provided much more extensive support for catching simple spelling and type errors.

The project did not test ajdoc integration for generating Javadoc™ output, nor did it test the debugging support. It also did not investigate any issues in working with design tools that convert between Java code and UML diagrams, nor testing tools that parse Java code. There should not be integration issues with these if the project uses .aj file extensions for AspectJ source, rather than .java. However, these tools may introduce secondary problems (e.g., refactorings that break pointcuts or generated tests that don't take account of aspect behavior). 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.

4. EFFECTIVE ADOPTION

The results from the prototyping were quite promising technically, and the issues encountered were deemed to be addressable. Because of the scale and importance of the system under study, the dominant concerns to be considered in an adoption roadmap were risk management and change management (i.e., how to train people and how to change processes to use the technology).

The principles defined in the follow up plan were phased adoption, clear vision and sponsorship, and building on continued successes from using the technology. Indeed, these same factors worked together to produce good results in short iterations during the investigation process. Our experience has shown that face to face meetings accompanied by demonstrations of the technology in action are the most effective means of exciting others about the technology's potential. Indeed, whilst white papers, technical reports, and presentations can give an intellectual understanding of the benefits of aspect-oriented programming, demonstrations have proven to be the key that unlocks doors like nothing else. There's something about the claims of AO that seem just 'too good to be true' until you've seen it for yourself.

The phases of adoption identified were the use of:

1. development-time aspects to police architectural, design, and coding guidelines
2. auxiliary aspects for policies such as those discussed in this report
3. core aspects used to implement functional parts of the design
4. the creation of aspect libraries

Progress has been made in all of these areas. The phased adoption plan also envisioned increasing the scale and scope of usage to achieve increasing benefits over multiple releases. This, in turn, allowed for isolating how the technology would impact different roles and skill sets. In particular, an important goal would be to allow a small number of specialists to define and maintain project policies in AspectJ initially. This would limit the training required for most developers to a basic level of awareness, rather than learning how to design and develop with AOP. Some groups we have subsequently worked with have rejected this idea, preferring not to create a divide amongst the development team. Others are proceeding more as initially envisaged.

An additional consideration was the need to address integration with a broader set of tools, including how to interoperate with ones that parse Java code such as UML modeling and testing tools. To date this has not yet proved to be a major stumbling block.

5. CONCLUSIONS

The project had tremendous success in converting broad system-wide policies from large and potentially ambiguous paper documents into AspectJ source code that unambiguously captured the same policies. This made the policies easier to understand, implement, and modify. It provided a convincing demonstration that AspectJ could be used to modularize many important crosscutting problems. While the findings were mostly positive technically, the project also identified some specific areas, primarily tools maturity issues that needed improvement. Subsequently, many of these became the focus of improvement in developing AspectJ 1.1 and AJDT.

The project also achieved significant results culturally; a large organization learned about AspectJ and AOSD and many individuals started applying it to their own projects. Naturally, adopting a new technology like AOSD is not to be taken lightly on a massive engineering project, and there is a lot of additional effort required to mitigate risks and manage the change. Overall, the results of this effort were deemed to be very favorable and formed an important input to IBM's initial assessment of AOSD.

The IBM team has continued to work with AspectJ and to grow their involvement in the AspectJ project. Significant additional work has been done to further the team's understanding of how AspectJ can be applied within the product-line, and progress has been made in all the envisioned adoption phases.

6. ACKNOWLEDGEMENTS

Thanks to Tracy Gardner, Ian Robinson, Jeremy Hughes, Graham Wallis, and all the team at IBM Hursley for making this project happen. Thanks also to Gregor Kiczales who was instrumental in delivering the consulting, and to Erik Hilsdale, Mik Kersten and Wes Isberg for supporting the project efforts. George Harley and Matthew Webster helped carry the flag for subsequent extensions of the work inside IBM.

7. REFERENCES

[1] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W. *An Overview of AspectJ*. In Proc. of ECOOP '01, LNCS 2072, pp. 327-353, Springer, 2001

[2] 2. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J., Irwin, J. *Aspect Oriented Programming*. In Proc. of ECOOP '97, LNCS 1241, pp. 220-243, Springer-Verlag, 1997

[3] 3. Log4j: The Apache Jakarta Project, http://jakarta.apache.org/log4j/docs/

[4] 4. Jakarta Commons Logging: The Apache Jakarta Project, http://jakarta.apache.org/commons/logging.html

[5] 5. The AspectJ project team: *New pertype aspect specifier, AspectJ 1.1 Readme*.

[6] 6. Colyer, A. Clement, A. and Kersten, M. *Aspect Oriented Programming with AJDT*. In proceedings Analysis of Aspect Oriented Software workshop, ECOOP 2003.

[7] 7. Davies, J. et al. *Aspect Oriented Profiler*. Practitioner Report, AOSD 200

[i] Ron Bodkin was working for Palo Alto Research Center, Inc. at the time the initial study was undertaken.