Applying AOP for Middleware Platform Independence

Ron Bodkin[1], Adrian Colyer[2], Jim Hugunin[3]

[1] New Aspects of Security*, 216 27th Street, San Francisco, CA 94131
rbodkin@newaspects.com
[2] IBM UK Limited, Hursley Park, Winchester, Hanst. SO21 2JN
adrian_colyer@uk.ibm.com
[3] Palo Alto Research Center, 3333 Coyote Hill Road, Palo Alto, CA 94304
hugunin@parc.com

Abstract. This report discusses experiences applying AspectJ [1] in a consulting project at IBM. The purpose of this project was to evaluate the suitability of AspectJ for modularizing crosscutting concerns in a middleware product line. This report describes and assesses the design approaches, tools integration, and cultural impact.

1 Introduction

The investigation worked with several components of a middleware product line. The investigation project team consisted of three consultants from PARC working with six IBM employees. The overall development effort for this product line encompasses several hundred developers and thousands of Java classes.

The primary motivation for using AOP [2] was to target multiple runtime environments with a single source code base. This was intended to allow certain components to be released under an open source license, allowing them to run in open source environments. However, it was important to continue to take advantage of improved platform-specific facilities when executing inside a specific container. It was important to keep the source code for platform-specific features separate, yet to ensure the two versions would remain in synch.

The most important concerns that were addressed were tracing and logging, event reporting, error handling, and performance monitoring. The project addressed each of these, as well as several additional concerns. An important secondary motivation was assessing the overall impact on architectural quality of using AOP to separate out these concerns.

The 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:

- reviewed the design of the pilot components
- analyzed specific concerns for these components
- interactively designed new aspects
- rapid prototyping, modifying production code using AspectJ 1.0.3

* The author was working for Palo Alto Research Center, Inc. at the time of this work.

- integration of the AspectJ tool set into production build processes
- integration of prototype code into a deployable format
- analysis of findings

Following the workshop the team spent additional time investigating the issues involved in integrating the technology into the process and culture of the product development organization.

2 Technical Design

This section describes how aspects were used to address the various concerns and assessments of the benefits and drawbacks.

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, 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).

It was straightforward to implement an effective tracing policy with one global aspect plus one aspect per coarse-grained component of approximately one hundred classes. The global aspect defined a consistent policy of when and how tracing is performed. It also ensures that all calls to tracing are "guarded" with a check to a method that determines whether tracing is enabled. The per-component aspects defined the scope of application (i.e., they define a concrete pointcut so the tracing advice applies to the component in question) and made inter-type declarations of toTraceString methods to override default logging output where necessary. Moreover, it was easy to plug in any of multiple different logging implementations, just by using a different global logging aspect.

This infrastructure represented a significant improvement over the status quo. 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).

However, implementing the logging concern exposed the importance of optimized performance. The AspectJ compiler has been targeted at generated crosscutting code that has a performance within 1-5% of the performance for hand coded Java. However, even this small penalty isn't acceptable for the widespread tracing and logging code scattered across a highly optimized server. The major concern is that the AspectJ 1.0.3 compiler creates thisJoinPoint objects eagerly, which would degrade perform-

ance (even when not tracing). This issue should be completely resolvable by tuning the compiler output for this kind of situation.

Moreover, the tracing facility used by the middleware product requires classes to register once with the tracing facility, and to use a returned object for all future tracing. By convention, the name used for identifying this tracing is the name of the class. However, AspectJ 1.0.3 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 in the same aspect.

Systematic logging for capturing events was also prototyped with good results. In this case, an aspect was created for each component that defined pointcuts, typically using wildcards in names to identify when an event occurred. In some cases, determining that "events" happened required refactoring the code to extract a method. However, these refactorings generally improved the quality of the code independently of their supporting the logging concern.

While event logging was significantly improved with AspectJ, there was concern about pointcut fragility. If a developer subsequently refactors code, this can break the definition of events. One possible approach to mitigating this concern is using the AspectJ tools 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.

An additional significant benefit of applying AspectJ to 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 and code that otherwise used the logging facility improperly. This policing aspect found several policy violations in one of the components.

2.2 Error Handling

The product line uses a sophisticated error analysis and reporting subsystem which ensures that each error is logged once at the source. Classes catch errors and pass the error and associated context (e.g., the executing object, a unique identifier for the line, and the type of error) to the subsystem.

The Java code of the product line was converted to use the error handling subsystem with a complex hand-built tool 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 or third party code needs to be manually instrumented. This tool is inflexible, and automated 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 trying to enforce coding discipline without tools.

By contrast, it was easy and effective to implement the error handling policy in AspectJ. A single reusable aspect was developed to codify the error handling policy. This effectively represents the important points where errors were detected (in exception handlers and in method returns). Ideally, AspectJ would provide a throws join

point, to capture the first point where exceptions were generated. However, the prototyped version does invoke error handling immediately after an exception is thrown (when it is handled or the method exits). In addition to the abstract aspect, the prototype included one aspect for each component to define the scope of application and to define 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 java.lang.Class.forName. 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 and it automatically updates new code for the policy.

2.3 Performance Monitoring

This application is extensively instrumented to capture performance information. Components provide a class with Java beans interfaces to access performance statistics for the component. The first component that was prototyped had statistics gathering scattered across ten classes and subtle inconsistencies in where information was collected.

By contrast, a single aspect was prototyped that defined a consistent policy for how to capture all the performance statistics for a component. Indeed, looking at the places where performance monitoring advice in the Eclipse AJDT tool allowed the team to find bugs. For example, there was one case where the counter was not being updated but should have been. 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.

This approach was easily generalized to a second component with comparable convenience and further reduced effort. Overall, performance monitoring was significantly improved by using AspectJ.

2.4 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). How-

ever, the pointcuts used were also able to support events in customer (3rd party)-written code by supporting a naming pattern (or customer defined pointcuts).

Likewise, the project designed how to use of aspects to instrument code with JMX for systems management. An aspect would allow adding management operations to an existing class or defining an adapter for management. The aspect approach to systems management would be very similar to the solution used for performance monitoring.

AspectJ was also 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

This section discusses how adding AspectJ to the existing system affected integration with the project's development tools and process.

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.

The team worked with an alpha version of the AJDT toolkit for Eclipse. This was helpful for visualizing concerns, but was hard to use because it was not yet a mature tool.

The AspectJ compiler worked quite well on the code base: it was easy to compile existing code, add aspects to it, and to test it. The project uses a sophisticated set of ant scripts, including a custom ant task for compilation, and maintaining 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 drawbacks in the resulting build process resulted from how it handled reusable aspects in multiple components. AspectJ 1.0 does 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. A more troubling concern is the possibility that the build process will generate incompatible class files for the same reusable aspects because they were compiled separately. This issue was not encountered in prototyping, but it was a concern nonetheless. Incremental linking and intermediate forms for aspects, which will support jar libraries, are features that are expected in AspectJ 1.1 and that would resolve these issues. A secondary issue was the lack of incremental compilation for AspectJ, which made compiling components about twice as slow, though this still remained tolerable. This, too, is expected to be addressed in AspectJ 1.1.

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 will provide 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).

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.

The phased adoption plan envisioned increasing 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.

In addition, the plan needed 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.

5 Conclusions

The project had tremendous success in converting broad system-wide policies from large and ambiguous paper documents into AspectJ source code that unambiguously

captured the same policies. This made the policies easier to understand, implement, modify, and switch between different implementations. This 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.

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 assessment of AOSD.

6 Acknowledgements

Thanks to Andrew Clement, Tracy Gardner, Ian Robinson, Jeremy Hughes, Graham Wallace, 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.

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. 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