The Next Steps For Aspect-Oriented Programming Languages *(in Java)*

Jim Hugunin
Xerox Palo Alto Research Center
hugunin@parc.xerox.com

Based on work and discussions with:
Erik Hilsdale, Wes Isberg, Mik Kersten and Gregor Kiczales.

Aspect-oriented programming (AOP) languages have reached an important milestone – developers are beginning to use them to build real commercial systems [1]. Just like in the early days of Smalltalk, many developers can get great advantage from using these new technologies despite the rough-edges that are always present in a first-generation technology. But just as in the early days of object-oriented programming (OOP) there remain many interesting problems to be solved to further improve the usability and power of this new technology and to maximize the benefits it can provide to developers.

The AOP languages that are currently in most active use (Hyper/J [2] and AspectJ [3]) are both built on top of the Java programming language and platform. This is not a coincidence, but rather reflects the usefulness of this platform for developing new programming language technologies in a form that is accessible to real-world developers. The lessons learned from designing these languages on top of Java appear to generalize well to other languages. They've inspired many other projects which extend languages ranging from C to C++ to Smalltalk with support for AOP using the same or parallel language constructs as AspectJ or Hyper/J [4, 5, 6, 7, 8].

While there is an emerging interest in the very broad field of aspect-oriented software design [9], this paper focuses more narrowly on the programming languages and most basic tools that form the foundation of any such field. There are four key areas of research that can improve the power and usability of these young AOP languages -- improved separate compilation and static checking, increased expressiveness for pointcuts, simpler use of aspects in specialized domains, and enhanced usability and extensibility of AOP development tools.

In order to provide concreteness this paper explores these areas with a focus on AspectJ. These areas are also relevant for other AOP languages built on Java (such as Hyper/J) and for other AOP languages built with similar languages constructs to AspectJ such as those referenced above.

Separate compilation and static checking

Java's ability to support separate compilation and separate static type checking provide it with many practical benefits for improving the modularity of systems. When extending Java to support AOP, it is important to work with its existing powerful support for good modularity. In this spirit, both AspectJ and Hyper/J provide good support for the separate static checking of modules. Hyper/J uses the addition of dummy abstract methods to allow each individual unit to be checked separately. Advice in AspectJ is statically checked for typing and control-flow errors in the same way that standard methods are in Java. Even though separate static checking is possible, each of these language implementations currently require a static compile or link process that operates on a larger system.

Since these languages already support separate checking, the easiest way to achieve separate compilation is by moving the weaving process from a static pass over a large system to a dynamic process that can operate on individual classes at load-time. Java provides good support for this through its extensible ClassLoader class. A good first step towards separate compilation for Java-based AOP languages would be to design a custom ClassLoader that can implement weaving on a per-class basis at load time. Further down the road, the need for playing games with ClassLoaders could be eliminated by adding the right small extensions to the executing virtual machine.

Equally important as supporting separate static checking and separate compilation where it makes sense is recognizing those features that cannot be implemented with simple-minded per-class separate compilation. Some powerful constructs for capturing crosscutting concerns operate on the static structure of a program and must be taken into account at compile-time. An example of one of these features is AspectJ's ability to declare compile-time errors using its existing pointcut designators, i.e.

```
declare error: call(Point.new(..)) && !within(PointFactory):
               "Points can only be made in PointFactory";
```

This code enforces the standard Factory design pattern's rule – in this case that new instances of the Point class can only be created by the PointFactory. To generate the most useful feedback for a developer, this rule should be enforced at compile-time. This requires that the relevant aspect is present when compiling any classes that might try to create a new Point -- which violates the standard rules of separate compilation. There are many additional powerful ways to improve our handling of crosscutting concerns by weakening the restrictions that each class must be separately compilable and statically checkable. These including more sophisticated handling of checked exceptions, control-flow, and type systems.

Currently, the only effective approach to handling concerns that crosscut the static structure of a program is to perform a sort of quasi-whole program analysis. If we can more clearly define the structures that make up a compilation unit these concerns can be handled more cleanly and this should also encourage the development of more language features that take advantage of this.

The first step to defining an appropriate compilation unit is to recognize that programmers are already working with units that are typically much larger than a single class. Java's support for sealed packages and .jar files could provide an initial practical unit for separate compilation that is well defined and yet larger than the single class. Moving further into the future, languages are going to have to be extended to allow programmers to better specify these units of interest.

Expressiveness of pointcuts

Pointcuts allow a programmer to select a group of interesting points in the execution of their program and give those points a name. Being able to give a name to a collection of interesting points provides the well-known benefits of procedural abstraction. For many of these pointcuts, the best way to specify them with current language technology is through an enumeration of specific methods. Here's how I might specify all the join points at which a FigureElement moves in AspectJ:

```
pointcut move():
    call(void Point.setX(int))  || call(void Point.setY(int)) ||
    call(void Line.setP1(Point)) || call(void Line.setP2(Point));
```

While this sort of description is useful, it is both cumbersome to write and brittle in the face of many reasonable changes to the Point and Line class. For example, if a developer was to add a "void setXY(int, int)" method to the Point class, it wouldn't be captured by this pointcut even though it made the Point move. This problem can be addressed by using pattern matching to provide a more flexible description of the join points of interest:

```
pointcut move(): call(void FigureElement+.set*(..));
```

This pointcut will capture the new "void setXY(int, int)" method; however, it would also capture a new method called "void setup()" that was not involved in moving a Point in any way. In order to write pointcuts that are robust to common refactorings or additions to an existing system, we want the pointcut to directly express the property they are interested in. The closest that we can come to this with existing languages is to capture every time that a field in FigureElement is changed:

```
pointcut move(): sets(* FigureElement.*);
```

This pointcut still doesn't quite specify the exact points that we're interested in. Here are a few natural language versions of the sort of precise statement we'd like to be able to make. "Any time that a field whose value is referenced under the control-flow of the FigureElement.paint() method is changed." Or possibly this could be captured more directly as "Any time that data in my system is changed that will lead to a change in the computation performed by the FigureElement.paint() method." These sorts of descriptions talk about how data flows through the different parts of a running system. One important step towards increasing the expressiveness of pointcuts would be to provide a language for talking about dataflow – perhaps in a form that is similar to AspectJ's current support for talking about control-flow.

Aspects in specialized domains

Current AOP languages do a good job of capturing the semantic structure of an OO program. The join points described in AspectJ can capture well-defined points in the execution of a program such as the execution of a particular method or the initialization of a new instance of a class. These semantic points are created based on the well-defined behavior of the source or bytecode for a system.

Unfortunately, many OO systems run in an environment that changes the actual execution behavior from the language or bytecode specifications. Examples of this include distribution through RMI or the many forms of container management in an Enterprise Java Server. These systems operate on classes at a meta-level very similar to how an aspect-weaver works. As more

code runs in these managed environments it will be important for AOP systems to interact well with them.

The first step is to better understand these domains and to implement a few custom solutions to integrate aspects with an EJB server. The longer term goals would be to propose a general solution to this problem. It is possible that a single general purpose AOP language could be used to implement the various special-purpose meta-level transformations that each tool does today. If this doesn't prove possible then a shared meta-model of a Java system that all of these tools could manipulate might work.

Another way that AOP can be applied to specialized domains is through the creation of domain-specific aspect libraries. AspectJ has an extension mechanism that is modeled very closely on standard OO extension that supports both the equivalents of composition and inheritance for aspects. While these mechanisms have been used to build many example libraries they have not yet been tested in practice.

The best reusable libraries are built by taking existing concrete systems and finding shared structures that can be abstracted into reusable components. Right now the AOP community is just beginning the process of building serious concrete systems. In the near future this should provide a sufficient base of experience and code that can begin to be mind for the appropriate reusable pieces. I expect that when this happens that it will lead to pressure on AspectJ's extension mechanisms to be expanded to handle the creation of these reusable libraries.

While aspect libraries can provide some domain specific support in a general purpose AOP language such as AspectJ, it is likely that some domains will be important enough to warrant the creation of domain-specific AOP languages. Some engineering approaches to encourage that sort of work are discussed in the next section.

Usability and extensibility of tools

Current AOP languages are useable today on real systems of medium-size. The ability to use these tools on real software projects is valuable both for the developers who use them and for the research community who can see what does and doesn't work in practice. Much of the credit for the ability of these tools to move so quickly from research ideas into useful systems belongs to the well-defined platform that the Java language and virtual machine provides to build on top of.

There are still technical hurdles that need to be overcome to use these tools on very large programs or to make them as easy to use for smaller systems as Java is. These include support for fast incremental compilation, for the optimization of woven code, for effective debugging, and for integration with or duplication of the full feature set of standard OOP IDEs including features such as code insight.

Overcoming these technical hurdles is important for evaluating the success of the AOP paradigm. The productivity benefits of any new tool are always offset against the lack of polish on those new tools. The more competitive tool support for AOP languages is with existing Java tools the easier it will be to judge the effect these languages have on developer productivity. This challenge is made harder by the fact that the greatest problems with modularity, and thus the greatest potential for AOP to make a difference, are found in the largest software systems. Building tools that can operate effectively on these large systems is an important engineering challenge.

Given the technical work involved in building these systems they should be as extensible as possible. To the extent that existing tools can be used to explore new ideas, the pace of research in this area should be increased. The most primitive form of extensibility is available today in the Open Source release of AspectJ which lets developers modify the system by editing the source

code. The next obvious step for this extensibility is to provide clearly documented and stable public APIs for adding things like custom pointcut designators or new static program transformations through extensions of the declare language space. This would make it easy for other Java-based AOP languages such as DemeterJ [10] to be implemented as a compatible extension to AspectJ. Further into the future, work is needed to determine the right way to enable programmers to extend these languages without operating on the compiler.

Having an extensible compiler can clearly be valuable to research projects that are exploring variations in the space of aspect-oriented languages. If this extensibility could be made sufficiently attractive to the developers of non-AOP language extensions it could be an easy way to capture some of these ideas as well. There are many good ideas in the programming language research community ranging from parameterized types to multiple-dispatch to preconditions and postconditions. If these features can be provided cleanly as extensions to an AOP language, this will allow developers to use their favorite language technologies all together.

Conclusion

The current generation of AOP languages are now powerful enough that they can be used to build real commercial systems. While this is an exciting milestone, it is not the end of the journey. There remain many interesting research and engineering questions to explore that will further increase the power these languages offer to developers. This paper focuses on one language in the current generation, AspectJ, to identify four key areas of research that will lead to important improvements.

These areas are relevant to other AOP languages as well. Static typing and separate compilation are relevant for all AOP work that wants to interact effectively with a statically typed world. The expressiveness of pointcuts is a fundamental problem in this space. The integration with specific domains is a challenge that will occur for all AOP languages despite the fact that solutions to this problem are likely to be platform specific. Figuring out how best to build usable and extensible tools is a problem that any serious programming language project faces – it is impossible to judge the usefulness of a programming language in the absence of real developers using it, and it is impossible to convince real developers to use a language without high-quality tools.

References

1. Price, R., *Real-world AOP tool simplifies OO development*, JavaReport Sept. 2001

2. Hyper/J, http://www.research.ibm.com/hyperspace/HyperJ/HyperJ.htm

3. AspectJ, http://aspectj.org

4. AspectR, http://aspectr.sourceforge.net/

5. AspectS, http://www.prakinf.tu-ilmenau.de/~hirsch/Projects/Squeak/AspectS/

6. Apostle, http://www.cs.ubc.ca/labs/spl/projects/apostle/

7. AspectC++, http://www.aspectc.org/

8. AspectC, http://www.cs.ubc.ca/labs/spl/projects/aspectc.html

9. *Special section on Aspect-Oriented Programming*, Communications of the ACM Oct. 2001

10. DemeterJ, http://www.ccs.neu.edu/research/demeter/DemeterJava/