Aspect-Oriented Programming is Quantification and Obliviousness

Robert E. Filman
RIACS
NASA Ames Research Center
Moffett Field, CA 94035
rfilman@arc.nasa.gov

Daniel P. Friedman
Computer Science Department
Indiana University
Bloomington, IN 47405
dfried@cs.indiana.edu

Abstract

This paper proposes that the distinguishing characteristic of Aspect-Oriented Programming (AOP) systems is that they allow programming by making quantified programmatic assertions over programs written by programmers oblivious to such assertions. Thus, AOP systems can be analyzed with respect to three critical dimensions: the kinds of quantifications allowed, the nature of the actions that can be asserted, and the mechanism for combining base-level actions with asserted actions. Consequences of this perspective are the recognition that certain systems are not AOP and that some mechanisms are expressive enough to allow programming an AOP system within them. A corollary is that while AOP can be applied to Object-Oriented Programming, it is an independent concept applicable to other programming styles.

1. Introduction

This paper is about Aspect-Oriented Programming (AOP) qua programming language. We are interested in determining what makes a language AOP. This work was prompted by a question from Tzilla Elrad, who asked whether event-based, publish and subscribe (EBPS) (for example, [9]) is AOP. After all, in a publish-and-subscribe system, separate concerns can be realized by subscribing to the events of interest to those concerns. In thinking about that question, we have come to the belief that two properties, *quantification* and *obliviousness*, are necessary for AOP.[1] Understanding these relationships clarifies the variety of possible AOP languages. It demonstrates why some systems that might seem to be AOP are not, and why some systems are environments in which one might easily build an AOP system.

2. Local and unitary statements

Programming languages are about writing a structure of *statements* that a compilation or interpretation process will elaborate as a series of primitive directions. (The directions themselves will be a finite text, though their interpretation may be unbounded.) The earliest computer (machine language) programs had a strict correspondence between the program text and the execution pattern. Generally, each programming language statement was both *unitary* and *local*—unitary in that it ended up having effect in precisely *one* place in the elaborated program, and local in that it is almost always proximate to its neighboring statements.[2]

[1] We are addressing the structural essence of AOP here, not it's application—somewhat similar to the difference between defining Object-Oriented Programming (OOP) systems in terms of polymorphic methods and inheritance, versus waxing euphoric about objects as the appropriate way to model the world.

[2] A minor exception to locality and unitarity was the introduction of types. Type declarations can have nonlocal and quantified effects such as type-coercion and type consistency checks. Types in conventional languages are thus an example of a built-in separate concern.

The history (of this part) of programming languages has been about moving away from purely local and unitary languages—about mechanisms that let the programmer separate concepts pragmatically, instead of being tied to saying things just where they happen. The first exceptions to locality were subprograms (*i.e.*, procedures, subroutines, functions.) Subprograms were a great invention, enabling abstracting out some behavior to someplace else. They have all the virtues of separating concerns. For example, expertise in, say, Runge-Kutta methods could be centered in the writer of the Runge-Kutta library. The application programmers would be users of that library. They still had to know something about Runge-Kutta (when it was useful, how to invoke it), and had to locally and explicitly call it in their code. They had to be *cognizant* of what was going on. One could still identify which statements would execute in which order, and the program was still unitary: it exhibited a direct correspondence between one statement in the programming language written, one sequence of machine instructions executed.

Inheritance in object-oriented programming (OOP) was the second important introduction of non-locality. Executing inherited behavior is non-local. There are two different fashions of inheriting behavior, send super and mixins.

Send-super systems like Java and Smalltalk allow the programmer to explicitly invoke the behavior of its parent class or classes, without knowing exactly what behavior is being invoked. Adding behavior to classes higher in the class structure allows a limited form of *quantified* program statements—that is, statements that have effect on many *loci* in the underlying code. For example, suppose we wish to introduce a "display" aspect to a program about simulating traffic movement. We will want to make quantified statements like "Whenever something moves (executes its move method), the screen must be updated." Imagine that all things that move are descendants of the class of moveable-object. We can accomplish this with send-super inheritance, if we have a *cooperative* base-class programmer—that is, one who will consistently follow directions. We make the behavior of the move method in movable-object be the display update and request the programmers of derivative classes to invoke send-super at the end of their code. This requires the derived class programmers to know that they have to do something, but relieves them of having to know what exactly it is that they have to do. We're also restricted with respect to the locus of behavior—we can ask programmers to do the send-super at the start of their code, or at the end, but our directions probably need to be consistent throughout the system.

Requiring cooperation is not good enough. Programmers may fail to be systematically cooperative, the base program may itself be already written or it may be otherwise out of our control. For true AOP, we want our system to work with *oblivious* programmers—ones who don't have to expend any additional effort to make the AOP mechanism work. The earliest example of oblivious quantification is mixin inheritance, found in MacLisp and Symbolics Lisp [5,15]. With mixins, the derived-class functionality is determined by assembling the code of the derived class with the advice of its super classes. The aspect programmer can make quantified statements about the code by adding mixins, while the derived class programmer remains ignorant of these actions. The scope of quantification is controlled by which classes inherit the mixin. That is, we can quantify over the descendants of some superclass, for a given single method. In the screen update example, adding an "after" mixin to movable-object's move accomplishes the automatic update.

In using inheritance to achieve aspects, single superclass inheritance systems require all aspects to match the class structure of the original program, while multiple inheritance systems allow quantification independent of the program's dominant decomposition. Mixins with multiple inheritance are thus a full aspect-oriented programming technology.

In general,

> *AOP can be understood as the desire to make quantified statements about the behavior of programs, and to have these quantifications hold over programs written by oblivious programmers.*

We want to be able to say, "This code realizes this concern. Execute it whenever these circumstances hold." This breaks completely with local and unitary demands—we can organize our program in the form most appropriate for coding and maintenance. We do not even need the local markings of cooperation. The weaving mechanism of the AOP system can, by itself, take our quantified statements and the base program and produce the primitive directions to be performed.

3. Quantification

AOP is thus the desire to make programming statements of the form

In programs P, *whenever condition* C *arises, perform action* A. (1)

over "conventionally" coded programs *P*. This implies three major dimensions of concern for the designer and implementer of an AOP system:

- **Quantification:** What kinds of conditions *C* can we specify.
- **Interface:** What is the interface of the actions *A*. That is, how do they interact with base programs and each other.
- **Weaving:** How will the system arrange to intermix the execution of the base actions of program *P* with the actions *A*.

In an AOP system, we make quantified statements about which code is to execute in which circumstances. Over what can we quantify? Broadly, we can quantify over the static structure of the system and over its dynamic behavior.

3.1. Static quantification

The static structure is the program as text. Two common views of program text are in terms of the public interfaces of the program (typically methods, but occasionally also public variables) and as a parsed-program as abstract syntax tree.

Black-box AOP systems quantify over the public interface of components like functions and object methods. Examples of black-box systems include Composition-Filters [2], synchronization advice [11] and OIF [8]. A simple implementation mechanism for black-box AOP is to wrap components with the aspect behavior.[3]

Clear-box AOP systems allow quantification over the parsed structure of components. Examples of such system include AspectJ, which allows (among other things) quantifying over both the calling and accepting calls in subprograms [13], and Subject-Oriented Programming, whose composition rules allow quantifying over elements such as the interpretation of variables within modules [10]. A given AOP system will present a quantification language that may be as simple as just allowing aspect decoration of subprogram calls, or complex enough to represent pattern matching on the abstract syntax tree of the program. Understood this way, a clear-box AOP system could allow static quantifications such as "add a print statement to show the new value of any assignment to a variable within the body of a while loop, if the variable occurs in the test of the while loop."

Clear-box and black-box techniques each have advantages and disadvantages. Clear-box techniques require source code. They provide access to all the (static) nuances of the program. In environments without the trickery of CORBA proxies or the equivalent, they can straightforwardly implement "caller-side" aspects (aspects associated with the calling environment of a subprogram invocation). Black-box techniques are typically easier to implement (in environments like Lisp they can be downright trivial) and can be used on components where the source code is lacking.

[3] OIF applied AOP to distributed systems, where it is important to perform aspects actions both on the client and server machines. Fortunately, the common implementations of distributed object systems like CORBA employ a proxy compiler that is easily subverted to insert calls to aspects.

Because black-box techniques can't quantify over anything besides a program's interface, clear-box techniques are especially useful for debugging. For example, a clear-box system could implement a concern like a statement-execution counting profiler, or writing to a log file on every update of a variable whose name starts with "log." However, black-box techniques are more likely to produce reusable and maintainable aspects—an aspect tied to the code of a module can easily slip into dependence on the coding tricks of that module. Interfaces imply contracts.

Clear-box techniques are more difficult to implement, as they usually imply developing a major fraction of a compiler. A typical clear-box implementation of structural quantification needs to obtain a parsed version of the underlying program, run transformation rules realizing the quantified aspects over that abstract syntax tree, and output the resulting tree back in the source language for processing by the conventional language compiler. That can be a lot of work.[4]

3.2. Dynamic quantification

Dynamic quantification is tying the aspect behavior to something that happens at run-time. Examples of such things are

- The raising of an exception.[5]
- The call of a subprogram *X* within the temporal scope of a call of *Y*.[6]
- The size of the call stack.
- Patterns on the history of the program (*e.g.*, after the "try password" routine has failed five times, with no intervening successes.)

Keep in mind that the abstractions most programming languages present about the structure and execution of a program are only a subset of the possible available abstractions: Scheme allows a programmer to capture the "current context" and reinvoke the current behavior. C programmers glibly rummage around on the stack, content in the knowledge that the pattern of procedure calls is straightforwardly recognizable so long as the machine and compiler remain constant. 3-Lisp allows the programmer access to the interpreter's state [7]. Elephant allows reference to previous variable values [14]. The ability to program with respect to such properties is an aspect of programming language design. Even if such elements are absent in the underlying language, an aspect language may still allow quantification over them.

4. Implementation issues

Assertion (1) suggests several other dimensions of aspect-oriented language design bear mentioning, though we lack the space to discuss them more fully.

---

[4] The popularity of Java byte-coding allows the possibility of quantifying with respect to byte code. We can clearly quantify over byte-code features demanded by the JVM architecture (*e.g.*, method entry points). To the extent that recognizable features of the surface structure are realized in recognizable ways in the byte code, we can quantify over these features and work with class files. Good luck if someone used a different compiler.

[5] Exception-handling by catching remotely thrown exceptions provides dynamic quantification over the exception events (but too late to do most interesting things with them, as the exception context has been lost). In Interlisp [16], the DWIM (do-what-I-mean) mechanism allowed quantification over exception events within the context of the exception.

[6] This does not fall out of the structural relationships of subprograms, as *X* may have been invoked by some pointed-to function, as can easily arise in polymorphic OOP. The call of *X* within the context of *Y* problem is an instance of the "jumping aspect" problem [3]. A concrete example of this problem arises when move is being used as an interior step of a "grander move," such as moving a collection of objects simultaneously. In that case, we want to update the display only once, at the end of the grand operation.

- **Context**. What context of the underlying program can action A reference? Clear-box systems can go so far as to make the aspect action be a seamless part of the resulting code, though tangling the aspect expression with the specifics of an implementation raises questions of "real" separation of concerns.
- **Quantification scope**. Over what elements in the program can one quantify? Typical choices include all methods on all objects in this class, all methods on all objects in this package, all methods with a given name within the objects of all subclasses of a class, and instance level variations of the above.
- **Dynamic quantification**. Can quantified assertions be made in a running system, dynamically adding and removing actions?
- **Incomplete Obliviousness**. Though we want the application programmer to be mostly oblivious of aspect system, there are times that the application needs to communicate with the aspects. For example, consider an action that provides a higher quality of service to higher priority tasks. The application needs a mechanism for specifying the current task's priority.
- **Action interaction**. How do actions communicate? Both different actions on the same locus and the same action on different loci may need to exchange information. (For example, the authentication and access control aspects on method M of object X may need to interact. Similarly, the authentication aspect on all objects may need to share a common space.)
- **Relative aspect orderings**. How do we specify the order of multiple actions that apply to the same locus?
- **Inconsistent aspects**. Sometimes one action may violate the semantics of another. For example, an action that logs progress may violate the "all-or-nothing" semantics of a transactional action. An AOP system may have some linguistic mechanisms for warning or forestalling such inconsistencies.
- **Weaving**. In an implementation sense, how do we arrange for the behavior of the actions to be intermixed with the behavior of the base system code? The answer may include compile-time weaving and altering the run-time interpretation process.

5. Aspect-Oriented Languages

To return to Tzilla's question, what's an aspect-oriented language? Let us consider some possibilities:

- **Rule-based systems.** Rule-based systems like OPS-5 [4] or, to a lesser extent, Prolog are programming with purely dynamically quantified statements. Each rule says, "whenever the condition is true, do the corresponding action."[7] If we all programmed with rules, we wouldn't have AOP discussions. We would just talk about how rules that expressed concerns X, Y, and Z could be added to the original system, with some mention of the tricks involved in getting those rules to run in the right order and to communicate with each other. The base idea that other things could be going on besides the main flow of control wouldn't be the least bit strange.

  But by and large, people don't program with rule-based systems. This is because rule-based systems are notoriously difficult to program. They've destroyed the fundamental sequentially of almost everything. The sequential, local, unitary style is really

[7] Actually, it's not nearly that neat, because the rule-based systems people insist on doing only one of the matching conditions and then considering the whole problem again, but that's an implementation detail.

very good for expressing most things. The cleverness of classical AOP is augmenting conventional sequentially with quantification, rather than supplanting it wholesale.

- **Event-based, publish and subscribe**. In EBPS systems, the subscription mechanism is precisely a quantification mechanism. ("Let me know whenever you see something like ..."). The question is then, is EBPS oblivious? If the application's programming style is to use events as the interface among components, then EBPS is a black-box AOP mechanism. On the other hand, if we expect the programmer to scatter event generation for our purposes throughout otherwise conventional programs, it's not oblivious and therefore not AOP.
- **Intentional Programming and Meta-programming**. Intentional programming (IP) [1] and meta-programming (MP) [12] provide the ability to direct the execution order in arbitrarily defined computational patterns. They can be seen as environments for writing transformation compilers, a mechanism for implementing clear-box AOP, rather than as self-contained realizations of the AOP idea.
- **Generative Programming.** Similarly, generative programming [6] works by transforming higher-level representations of programs into lower-level ones (that is, by compiling high-level specifications.) By incorporating aspects into the transformation rules, one can achieve AOP in a generative programming environment. Correspondingly, the creator of a generative programming system may recognize some aspects as being important to the domain-specific system being defined, and precisely leave a place in the generative language for expressing those aspects.

6. Closing Remarks

In this paper we have identified AOP with the ability to assert quantified statements over programs written by oblivious programmers. This implies

- **AOP is not about OOP**. OOP is the current dominant programming language technology. Most implementations of new language ideas are done in the context of OOP. However, "oblivious quantification" is independent of OO concepts. Therefore, it would be perfectly reasonable to develop AOP for a functional or imperative language.[8]
- **AOP is not useful for singletons**. If you've got an orthogonal concern that is about exactly one place in the original code, and you're sure that that orthogonal concern will not propagate to other loci as the system evolves, it is probably a bad idea to use AOP for that concern. Just write a call to the aspect procedure into that one place in the code, or permute the source code in whatever way you thought necessary to achieve the aspect. The quantity of communication (among programmers) required to do aspects in general probably equals the quantity of communication required to modify just one program. The homogenized code leaves no ambiguity about what's happening, but may be less clear than what's happening as written as separate aspects.
- **Better AOP systems are more oblivious.** They minimize the degree to which programmers (particularly the programmers of the primary functionality) have to change their behavior to realize the benefits of AOP. It's a really nice bumper sticker to be able to say, "Just program like you always do, and we'll be able to add the aspects later." (And change your mind downstream about your policies, and we'll painlessly transform your code for that, too.)

[8] The class hierarchy of OO systems is a convenient structure over which to quantify. OOP is thus a "pleasant" environment for AOP, but not a necessary one.

Acknowledgements

Our thanks to Diana Lee and Tarang Patel for comments on the drafts of this paper.

References

1. Aitken, W., Dickens, B., Kwiatkowski, P., de Moor, O., Richter, D., and Simonyi, C., "Transformation in intentional programming," in Devanbu, P. and Poulin, J. (Eds.) Proc. 5th Intl Conf. on Software Reuse, Victoria, Canada, IEEE Computer Society Press, June 1998, pp 114–123. http://www.research.microsoft.com/ip/overview/TrafoInIP.pdf
2. Aksit M. and Bedir Tekinerdogan, B. Solving the modeling problems of object-oriented languages by composing multiple aspects using composition filters. AOP'98 workshop position paper, 1998. http://wwwtrese.cs.utwente.nl/Docs/Tresepapers/FilterAspects.html
3. Brichau, J., De Meuter, W., and De Volder, K. Jumping aspects. Workshop on Aspects and Dimensions of Concerns, ECOOP 2000, Cannes, France, June 2000.
4. Brownston, L., Farrell, R., Kant, E. and Martin, N. *Programming Expert Systems in OPS5.* Reading, Massachusetts: Addison-Wesley, 1985.
5. Cannon, H. Flavors: A non-hierarchical approach to object-oriented programming. Symbolics Inc. (1982).
6. Czarnecki, K. and Eisenecker, U. W. *Generative Programming: Methods, Tools, and Applications.* Boston: Addison-Wesley, 2000.
7. des Rivieres, J. and Smith, B. C. The implementation of procedurally reflective languages. Conference Record of the 1984 ACM Symposium on LISP and Functional Programming, pp. 331–347, Austin, Texas, August 1984.
8. Filman, R. E., Barrett, S., Lee, D. D., and Linden, T. Inserting ilities by controlling communications. *Communications of the ACM,* in press. http://ic-www.arc.nasa.gov/ic/darwin/oif/leo/filman/text/oif/oif-cacm-final.pdf
9. Filman, R. E. and Lee, D. D. Managing distributed systems with smart subscriptions. Proc. International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'2000), Las Vegas, June 2000, pp. 853–860.
10. Harrison, W., and Ossher, H., Subject-Oriented Programming – a critique of pure objects. Proceedings of 1993 Conference on Object-Oriented Programming Systems, Languages, and Applications, Washington, September 1993, pp. 411–428.
11. Holmes D., Noble J. and Potter J., Towards reusable synchronisation for object-oriented languages. Aspect-Oriented Programming Workshop, ECOOP'98, July 21, 1998. http://www.mri.mq.edu.au/~dholmes/research/aop-workshop-ecoop98.html
12. Kiczales, G., des Rivieres, J., and Bobrow, D. *The Art of the Metaobject Protocol.* Cambridge, MA: MIT Press, 1991.
13. Lopes, C. V. and Kiczales, G. Recent developments in AspectJ. *ECOOP'98 Workshop Reader.* Berlin: Springer-Verlag LNCS 1543, 1998. http://www.parc.xerox.com/csl/groups/sda/publications/papers/Lopes-AOPW-ECOOP98/
14. McCarthy, J. Elephant. http://www-formal.stanford.edu/jmc/elephant.html.
15. Moon, D. A. Object-oriented programming with flavors. ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '86) *ACM SIGPLAN Notices, vol. 21,* no. 11, 1986, pp. 1–8.
16. Teitelman, W. and Masinter, L. The Interlisp programming environment, *Computer vol. 14,* no. 4, 1981, pp. 25–34.