

Aspect Orientation as a Software Engineering Problem

John Daigle, Arash Akhlaghi

Abstract—Aspect oriented programming has been growing in popularity for several years. With many languages now available to programmers and software engineers, and extended UML with aspects, it is appropriate to place our attention on aspect orientation. In this paper we will attempt to define AOP, briefly describe several languages that implement aspects, and examine several claims of gain from the implementation of the Aspect Paradigm.

I. INTRODUCTION

As software systems become larger and more complex, the limitations of the object oriented paradigm become more obvious. Large scale systems that involve many different subsystems can be complicated to envision as objects. In addition, the object oriented paradigm can miss important relationships between areas.[6] Aspect-Oriented Programming, or AOP, is meant to solve this problem by decomposing the system into aspects which may cut across a number of subsystems.

There are several difficulties inherent in the Aspect Oriented Model. The first is that, unlike object oriented programming, which is robustly defined, AOP is poorly defined. Also, while most people have an intuitive grasp of the notion of an “object”, the concept of “crosscutting”, while evocative, is potentially misleading and leads to confusion.[12]

In this paper, we will define AOP, briefly describe its history, and describe on paradigm that has been adopted from OOP to AOP, the Demeter™ Method.

II. ASPECT ORIENTED PROGRAMMING

One way in which Aspect oriented programming can be defined is by the implementation. In much the same way that Smalltalk served as an early definition of OOP, other languages can serve to define AOP. Several AOP languages have so far been created, including AspectJ, AspectC, and Weave.net.[6] While defining a concept from this direction has the advantage of showing specific solutions to specific problems, it often is not clear about the underlying concept.

Similarly, defining object orientation by concept would not be as effective as defining it in terms of key concepts such as encapsulation, inheritance, and modularity. A specific and concrete definition of what is meant by an “object”, and how that differs from a C struct is invaluable both pedagogically and conceptually. The failure to communicate these concepts clearly and precisely in entry level classes creates great difficulty for students trying to understand object orientation later, precisely because the languages used impose poor style choices, or make OOP difficult.[5]

A. Implementation

One way in which Aspect oriented programming can be defined is by the language. In much the same way that Smalltalk served as an early definition of OOP, other languages can serve to define AOP. There are several popular languages that are considered Aspect Oriented. These include AspectJ, JAsCo, AspectCOOL, AspectualCAML, and AspectC++. We will briefly examine each of these languages to see what common elements make them Aspect Oriented.

1) *AspectJ*: AspectJ is widely considered the most popular of the AO languages.[2] In practical terms, it is similar in implementation to Objective C, a language which is essentially C, but with a small number of Object Oriented Extensions. Similarly, AspectJ is Java, but with extensions to allow for AOP. There are four constructs that are part of AspectJ but not part of Java.[3]

AspectDeclaration is much like an object declaration in Java. In addition to standard Java members, an Aspect must contain pointcut declarations and can contain advice.

Pointcut Pointcuts are descriptions of patterns of *join points*, run time events such as method calls that are relevant to a particular aspect.

Advice Additional behavior that is triggered when a join point matches.

Pattern Define when a particular point matches a pointcut and should trigger the advice defined in the Aspect Declaration.

These aspect oriented solutions allow for some fundamental changes in the way that Java behaves. For example, the programmer could use a pattern to define a relationship between two packages. Rather than code each member of both packages, a pattern could define the package name with a wildcard, various join points define what methods must be constrained, and the advice defines how the interaction between packages will be handled. Or, an Aspect can define a small subset of common behaviors, such as logging, caching, or using the screen, that must execute a common routine or two. In this case, the aspect *crosscuts* the various packages, creating a “metapackage” with its own data points and don’t behaviors.[4]

2) *AspectC++*: Aspect C++ was designed to allow AOP in systems that cannot afford the runtime costs of the JVM. AspectC++ follows the model of AspectJ by deploying Pointcuts, Advice, and Aspects. Like AspectJ, AspectC++ is implemented primarily as extensions to the C++ language. By design, AspectC++ is essentially a streamlined, C++ imple-

mentation of concepts already embodied in AspectJ.[11]

There are some differences between AspectJ and AspectC++, beyond the fully compiled code of the latter. In AspectJ, there are two ways to insert a method or attribute into an existing class, either by using a Generalized Type Name (GTN) or an Introduction.[13] A GTN is a pattern statement, so any class that matches the pattern will get the advice from the pattern statement. This is an implicit means of adding attributes or methods to a class. An Introduction is an explicit method of giving advice to a class. Spinczyk et al. conclude that this can be confusing to follow, and in AspectC++, there is a unified method of handling either GTN's or Introductions.[11]

3) *JAsCo*: JAsCo was developed specifically for Adaptive Programming. It is, like AspectJ, an extension to the Java Language, however, it introduces only two new concepts to the language.

aspect Beans extended Java bean that specifies behavior to be taken.

connector Defines a relationship between an Aspect bean and a match pattern.

The primary motivation for creation of JAsCo was to allow for the reusability of Aspects, in the same way that objects can be reused in OOP, and to follow the Law of Demeter for Aspect Oriented Programming.[14] The advantage of this is relatively obvious, the Aspect can be applied to any program simply by including the appropriate hooks into the base code, thereby allowing the crosscutting to take place.

JAsCo is also well suited to Adaptive Programming. Because of the highly consistent syntax of aspect beans, the addition of an additional connector, the *traversal connector*, allows aspect beans to be treated as an adaptive visitor.[15] This allows the associated aspects of particular methods to be determined at runtime.

4) *AspectCOOL*: AspectCOOL is built on top of the language COOL, the Classroom Object Oriented Language. AspectCOOL is designed to overcome a limitation of AspectJ. In AspectJ, separate compilation of class files is impossible. AspectCOOL lacks this limitation, in ACOOL, as in JAsCo, class files can be compiled separately.

There are two advantages to the dynamic loader in AspectCOOL. First, there is a development bonus. When aspects or objects can be compiled individually, it is easier to test and repair a piece of software. Second, there is an advantage in the design of the weaver. The AspectCOOL weaver is a run time function that dynamically intercepts method calls, introduces new advice to those methods, recompiles them, and continues the program.[1]

5) *AspectualCAML*: Predictably, AspectualCAML is a superset of ObjectiveCAML, a functional language known for its efficient compiler. AspectualCAML uses the same paradigm as AspectJ, creating Aspects, pointcuts, advice, and patterns to the base language. The base language uses type inference and allows polymorphism, so there are significant differences between the implementation of AspectualCAML and AspectJ. This difference is most telling in the design of the Aspect compiler, or weaver.

Specifically, AspectualCAML's weaver must begin by inferring types in all function definitions, then infers types in the aspect definitions. After this, it can begin simplifying advice definitions and eventually produce ObjectiveCAML, which can be compiled. [10]

6) *Concluding thoughts on languages*: The methodology for creating an AOPL is now clear. First, find an Object oriented language that does not have an Aspect Oriented Language written on top of it, or write a new library for Java. At the time of writing, AspectJavaScript, AspectCLOS, and AspectPython are all available. Then extend the core language, describing joinpoints, pointcuts, aspects, and patterns.

This is very similar to the addition of object oriented functions to existing languages that was seen in the 1980's and early 90's with C++, CLOS, Fortran90, Object Oriented Cobol, OCAML, and Object Pascal. While all of these languages have their adherents, the largest growth in object oriented programming probably belongs to Java, which was designed as an object oriented language, rather than as object oriented additions to an existing language. Aspect Oriented programming has yet to see a language designed to be Aspect Oriented.

B. Language Independent Description

Steimann defines an aspect as a 3-tuple $(\mathcal{P}, \mathcal{C}, \mathcal{A})$ such that in a program \mathcal{P} , whenever \mathcal{C} occurs, perform action \mathcal{A} . In addition, \mathcal{P} should be oblivious to the definition of \mathcal{C} or the action \mathcal{A} . [12] This brings up an immediate problem. Aspect Orientation is typically seen as an extension to Object Orientation. However, if the program must be oblivious to the context in which certain Aspects will be triggered, the Aspects themselves will need a great deal of access to the program, in order to determine whether the conditions hold. This seems to imply a certain lack of modularity.

Steinmann [12] argues at length that there is no good definition of Aspect Oriented programming beyond the 3-tuple, and that therefore, there is a fragility and failure to AOP as a solution to crosscutting concerns. Essentially, Steinmann's argument is that Aspect Oriented programming is at tension with the object system it rests on.

In contrast, Lopes & Bajracharya approach exactly that problem in an extensive study of a single application, analyzing a web application using Net Options Value. They found that the savings in aspects such as logging and authentication offset the costs of implementing those aspects in the first place. [9] However, though they discuss modularization in depth, they do so in the context of the object oriented understructure, never addressing the maintenance costs added by the aspect oriented structure, costs that invalidate in part the effort spend modularizing the program in the first place.

III. LOD UNDER ASPECT ORIENTATION

One unusual application for AOP is to check a system for violations of the LoD. The LoD defines which classes may call which method of another class as follows, For all classes \mathcal{C} , and for all methods \mathcal{M} attached to \mathcal{C} , all objects to which \mathcal{M} sends a message must be instances of classes associated with the following classes:

- 1) The argument classes of \mathcal{M} (including \mathcal{C})
- 2) The instance variable classes of \mathcal{C} [7]

Obviously, Demeter violations are a valid crosscutting concern, a violation in any part of the system affects all parts of the system. The solution suggested by Lieberherr et al is to construct an extension on AspectJ. The extension performs two tasks. First, it collects all methods that can be called from a particular object, and second, it verifies that those methods follow the LoD.[8] In order to do this, the system must show that the LoD can be expressed in terms of aspect oriented programming, specifically in terms of join points. In any aspect oriented program, there are lexical join points and dynamic join points. Lexical join points are defined within the program, dynamic join points are events within the execution of the program. The relationship between these concepts is referred to as *Shadow*.

Therefore, for any lexical join point \mathcal{L}_p , there is a collection of dynamic join points that shadow that join point. So, from a given execution of a program \mathcal{P} with input i , the dynamic join points can be derived from execution behavior, and the lexical join points can be derived from the shadow of the dynamic join points. This provides an opportunity to check the Law of Demeter for dynamic objects. However, future work remains to be done in order to properly check for lexical joins.

IV. SOME CONCLUSIONS

Strikingly, all of the languages discussed have been additions on existing Object Oriented Languages, two of which ObjectiveCAML and C++, are composed of object oriented extensions to existing procedural languages. In the case of Object Oriented Programming, the first languages, Simula and Smalltalk, were new creations meant to take advantage of the new paradigm.

In Aspect Oriented Programming, it seems like there is an imperfect solution to a fairly complex problem. That is, there can be any number of aspects to a particular program, but in any case, the crosscut represents an alternative decomposition of the program. To be blunt, at some point someone put together a system decomposition that made sense to them, and the purpose of Aspect Oriented programming is to break that system decomposition.

The use of Aspect Oriented Programming to check the Law of Demeter was likewise not an extension of the law into a new paradigm, but the addition of layer above a Java program in order to confirm correctness. But of course, it can be argued that the Law of Demeter is broken by any Aspect Oriented software anyway, as the entire point of crosscutting is to add methods and objects across multiple packages or objects, and to allow access to member variables to aspect level abstractions.

Aspect orientation seems to break the object oriented model. The point of encapsulation is to prevent tight coupling between objects. But in an aspect oriented program, after that tight coupling is carefully programmed out by the engineer, and the system is effectively designed to good practices, the aspect system is woven through every object, effectively recreating the coupling that was removed in the first place. It would seem

unclear what, if anything, is gained by including the Aspect Oriented paradigm in your programming toolkit.

REFERENCES

- [1] Enis Avdičaušević, Mitja Lenič, Marjan Mernik, and Viljem Zumer, *AspectCOOL: an experiment in design and implementation of aspect-oriented language*, SIGPLAN Not. **36** (2001), no. 12, 84–94.
- [2] Pavel Avgustinov, Elnar Hajiye, Neil Ongkingco, Oege de Moor, Damien Sereni, Julian Tibble, and Mathieu Verbaere, *Semantics of static pointcuts in aspectj*, POPL '07: Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages (New York, NY, USA), ACM Press, 2007, pp. 11–23.
- [3] Martin Bravenboer, Éric Tanter, and Eelco Visser, *Declarative, formal, and extensible syntax definition for aspectj*, OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications (New York, NY, USA), ACM Press, 2006, pp. 209–228.
- [4] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William Griswold, *Getting started with ASPECTJ*, Commun. ACM **44** (2001), no. 10, 59–65.
- [5] Michael Kölling, Bett Koch, and John Rosenberg, *Requirements for a first year object-oriented teaching language*, SIGCSE '95: Proceedings of the twenty-sixth SIGCSE technical symposium on Computer science education (New York, NY, USA), ACM Press, 1995, pp. 173–177.
- [6] Donal Lafferty and Vinny Cahill, *Language-independent aspect-oriented programming*, OOPSLA '03: Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (New York, NY, USA), ACM Press, 2003, pp. 1–12.
- [7] K. Lieberherr, I. Holland, and A. Riel, *Object-oriented programming: an objective sense of style*, OOPSLA '88: Conference proceedings on Object-oriented programming systems, languages and applications (New York, NY, USA), ACM Press, 1988, pp. 323–334.
- [8] Karl Lieberherr, David H. Lorenz, and Pengcheng Wu, *A case for statically executable advice: checking the law of demeter with AspectJ*, AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development (New York, NY, USA), ACM Press, 2003, pp. 40–49.
- [9] Cristina Videira Lopes and Sushil Krishna Bajracharya, *An analysis of modularity in aspect oriented design*, AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development (New York, NY, USA), ACM Press, 2005, pp. 15–26.
- [10] Hidehiko Masuhara, Hideaki Tatsuzawa, and Akinori Yonezawa, *Aspectual caml: an aspect-oriented functional language*, ICFP '05: Proceedings of the tenth ACM SIGPLAN international conference on Functional programming (New York, NY, USA), ACM Press, 2005, pp. 320–330.
- [11] Olaf Spinczyk, Andreas Gal, and Wolfgang Schröder-Preikschat, *Aspectc++: an aspect-oriented extension to the c++ programming language*, CRPIT '02: Proceedings of the Fortieth International Conference on Tools Pacific (Darlinghurst, Australia, Australia), Australian Computer Society, Inc., 2002, pp. 53–60.
- [12] Friedrich Steimann, *The paradoxical success of aspect-oriented programming*, OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications (New York, NY, USA), ACM Press, 2006, pp. 481–497.
- [13] Dominik Stein, Stefan Hanenberg, and Rainer Unland, *A UML-based aspect-oriented design notation for aspectj*, AOSD '02: Proceedings of the 1st international conference on Aspect-oriented software development (New York, NY, USA), ACM Press, 2002, pp. 106–112.
- [14] Davy Suvé, Wim Vanderperren, and Viviane Jonckers, *JAsCo: an aspect-oriented approach tailored for component based software development*, AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development (New York, NY, USA), ACM Press, 2003, pp. 21–29.
- [15] Wim Vanderperren, Davy Suvé, Bart Verheecke, María Agustina Cibrán, and Viviane Jonckers, *Adaptive programming in JAsCo*, AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development (New York, NY, USA), ACM Press, 2005, pp. 75–86.