An Examination of Aspect-Oriented Programming in Industry

A thesis submitted to the

Colorado State University Honors Program

By

Jeremy T. Bradley

Advisor: Dr. Roger T. Alexander

May 2003

Technical Report CS-03-108

Colorado State University

Department of Computer Science

Fort Collins, Colorado, USA

Contents

Abstract

This paper is an investigation into the impact that using aspect-oriented software techniques has on the qualities of software, such as complexity, correctness and testability. The methods used to conduct this investigation were based on interviews with developers who have used aspect-oriented technology in real world projects. This paper examines the problems that these developers encountered in their work, and provides possible explanations as to the cause of these problems. It concludes with an analysis of the use of aspect-oriented programming from the perspective of the developers interviewed, including factors limiting their use of this technology, and their perceived potential of this programming methodology.

1 Introduction

Software engineering is a relatively new field, and it has continued to evolve rapidly since its inception. Since that time, researchers and theorists have consistently sought to improve upon the techniques and procedures of software engineering, in order to improve our ability to create quality software. The basic desire is to produce engineering methods that allow for the efficient creation and maintenance of software. To that end, there are several key qualities that are we desire to improve:

- The *modularity* of the software
- The *reusability* of the software
- The *readability* and *understandability* of the implementation, and
- The *correctness* and *testability* of the software

While object-oriented programming has made some progress in improving these qualities, there is still much room for improvement. Even in a well-implemented object-oriented program, there is often functionality spread throughout most of the modules in a system. This functionality can include such things as security handling, logging and other more advanced functionality like state management. These and other properties are called *crosscutting concerns*. A crosscutting concern is some attribute of a software implementation that is

spread throughout the implementation, instead of being modularized. Devising methods to modularize these properties is the chief concern of *Aspect-Oriented Programming [1]*. The question, however, is whether Aspect-Oriented programming techniques allow for the modularization of crosscutting concerns without creating new problems for software developers. For instance, it is unclear whether the use of aspect-oriented techniques increases or decreases the complexity, reusability, correctness and other qualities of an implementation.

The purpose of this paper is to assess the utility of aspect-oriented programming by examining the primary literature on the subject and observing the usage of these new software engineering techniques in real world software projects. Specifically, this paper attempts to answer the following questions:

- *How does the usage of aspect-oriented programming affect the complexity, both structural and cognitive, of a given implementation?*
- *How can using aspect-oriented software design affect the correctness and testability of a program?*
- *How is aspect-oriented technology currently utilized in industry, and do the current users feel that aspect-oriented techniques still have the potential, given their current experience with this technology?*

In order to answer these questions, the aid of developers involved in industry projects using aspect-oriented technology was enlisted. These contributors are Avery Moon of Infotone Communications, Inc. and Renaud Pawlak of the Java Aspect Components (JAC) project. They have both provided insightful comments and pertinent evidence that has helped to compile this paper.

2 Motivation and Uses for Aspect-Oriented Programming

Before examining the details of aspect-oriented programming, it may be useful to discuss the potential uses of this technology, and specifically the problems that it tries to address. A very basic problem is the need to use *development code*, such as logging or contract reporting code, to help create and test software. Inserting this type of code into a software implementation can be a tedious and time-consuming process. To compound the problem, much of this code is only useful during the development phases of a project, and leaving this code in a production build is often undesirable because it can create performance problems, as well as inadvertently introduce software defects [2].

With the use of aspect-oriented programming techniques, it is possible to implement this development code in a modularized fashion, so that it can be inserted easily and efficiently into the production code. Additionally, because this

code is well modularized, it can be easily maintained or removed as well. [1] Overall, this usage of aspect-oriented programming can help to decrease development time, improve production system performance and minimize the chance of development code introducing defects into the implementation [3]. However, note that this is a trivial usage of aspect-oriented programming as software developers can accomplish this functionality with macros or other similar tools.

There are other more complicated problems with software development that aspect-oriented programming is trying to solve, such as state management, synchronization, and session management [3]. With existing software development tools, it is difficult to modularize this type of functionality. To explain, the difficulty with object-oriented design methods is that they rely on the modularizing the system into components based on decomposition into *functional units,* represented by classes. A clean object-oriented design may then have to be modified to add a feature, such as state management, which will involve several functional units. Therefore, the code that implements that feature will need to be placed in all of those components [3].

The intent of aspect-oriented programming is to create language mechanisms that allow all the functionality present in the system to be modularized, including the functionality that is scattered throughout multiple components of the system [2]. If this is done, it can greatly enhance the

maintainability, reusability and other qualities of the software. Although this is similar to the goals stated earlier dealing with development code, the specific goal here is to allow all of the main functionality of an implementation, whether or not it is a crosscutting concern, to be quickly and efficiently modularized.

It is easy to see how modularizing aspects can increase the maintainability of a program since it allows changes to particular functionality to happen in only one location, and this can lead to more reusable code. The idea is that, for example, an aspect that controls the screen updates for one drawing can be used again, without modification, to provide the same functionality to another similar program. This is where the most exciting potential of aspect-oriented technologies rests.

Another, ancillary, of aspect-oriented programming is to produce software that is efficient to run, without sacrificing other qualities, such as the readability and maintainability of the code. As an illustration of this, consider the example explored in the paper *Aspect-Oriented Programming,* by Kiczales, et. al [2]. This paper describes an experiment in which the authors create three separate implementations of an algorithm that is part of a graphics-filtering program.

The first implementation is a hand-coded algorithm that is well modularized, using procedural techniques, and easy to read. Unfortunately, this

implementation is highly inefficient in both execution time and storage requirements.

The second implementation is a hand-optimized version of the latter. This version, while much more efficient in both execution time and space requirements, is very difficult to read and understand by anyone, including the original author. The reason is that since many different concerns are tangled within a very complicated procedure.

The third implementation makes use of aspect-oriented techniques to construct a working unit of code that is both easy to read and maintain, and also roughly as efficient as the hand optimized solution.

So far, only functional uses of aspect-oriented programming have been discussed, but there are many other uses of aspect-oriented programming that are not specifically related to functionality. An example of a non-functional use includes enhancing the *readability* of the implementation. The readability of the code is important because it can affect many other qualities, such as the understandability, maintainability, correctness and reusability of the code. The techniques of aspect-oriented programming have the potential to create a more readable code base, because they can physically separate the different functional concerns in the code In effect, aspect-oriented techniques create layers of functionality in a software implementation. Each new aspect introduces a new functional characteristic of the program, and each layer can be read separately

from the others. If properly done, this can make understanding an implementation much easier.

3 Aspect-Oriented Programming

Before the influence of aspect-oriented techniques on software engineering can be discussed, it is important to understand the basis of these techniques. It is also important to understand the terminology and language used in this paper. In order to facilitate this, this paper will primarily use the AspectJ(TM) programming language and its terminology [4], but it also makes use of the Java Aspect Components (JAC) project, a Java(TM) based aspect-oriented framework [5]. However, there are many different languages that take advantage of aspect-oriented technology, all at various stages of development. AspectJ is used because it is one of the more mature projects.

AspectJ is “a simple and practical extension to the Java programming language that builds upon the object model of Java with enhancements that allow aspect-oriented programming techniques to be used” [1]. It is compiled into standard Java bytecode, and it is able to run on any Java platform.

Although this paper will discuss the basis of aspect-oriented programming and related tools such as AspectJ and JAC, a thorough explanation of these tools is beyond the scope of this paper. An in-depth explanation and reference to AspectJ can be found at the AspectJ Documentation Page [6], and

similar information for JAC can be found at the JAC Documentation Page [7]. General information on aspect-oriented programming can be found at the Aspect-Oriented Software Development web page [8].

3.1 What is Aspect-Oriented Programming?

Aspect-Oriented programming is a method of software engineering that is intended to build upon the earlier successes of procedural, functional and object-oriented programming by introducing aspect-oriented techniques to these programming paradigms. It does not intend to replace these programming techniques, but rather to augment and improve their abilities [4]. The aim of aspect-oriented programming is to allow the clean modularization of crosscutting concerns using aspects.

3.2 What is an Aspect?

Aspect-Oriented techniques provide mechanisms that allow crosscutting concerns to be expressed as separate units from the main implementation. These units are referred to as *aspects*, and they are the basic unit of modularization for crosscutting concerns in aspect-oriented programming. However, as aspect-oriented programming is only intended as an extension to existing programming methods, aspects work in conjunction with a base implementation represented with other constructs, such as classes or procedures

[4]. The base implementation that these aspects work with is referred to, for the purposes of this paper, as the *primary abstraction.*

Aspects are designed to allow crosscutting concerns to be easier to maintain, and more reusable. For instance, in the example of logging, the programming statements that generate log entries for the entire implementation can be maintained in one aspect, and changes to those statements need only be made in just one place, versus having the modify the entire code base. In languages such as AspectJ, aspects are represented in structures that are very similar to classes.

The following subsections define more terms related to aspects, and briefly describe the different types of aspects, and how languages such as AspectJ manage those aspects.

3.2.1 Dynamic and Static Crosscutting Aspects

There are two types of crosscutting that an aspect can facilitate. The first type is called *dynamic crosscutting*. Dynamic crosscutting makes it possible to "define additional implementation to run at certain well-defined points in the execution of the program" [4]. Dynamic crosscutting, contrary to the appearance of the name, does not mean that the code is modified at runtime. The concept of dynamically modifying code with aspects as runtime is discussed in section 3.2.3.

Instead, dynamic crosscutting refers to the selective modification of the primary abstraction at certain points of the program without affecting the static type signature of the program [1].

There are different methods used to define dynamic crosscutting in aspect-oriented programming. AspectJ, and languages similar to it, use the concept of a *join-point* to facilitate the introduction of aspect code into the primary abstraction. Join-points are the "well-defined points in the execution of a program" mentioned earlier. Put simply, join-points are places in the program code that are easily distinguishable from each other and the rest of the code. Examples of join-points include the beginning and end of a method or function, an object instantiation, and an exception handler execution.

When dynamic crosscutting is used in an aspect, it has two crucial parts. These parts are the new implementation code to add to the primary abstraction, and a specification of where to add it. In the AspectJ language, these parts are called the *advice* and the *pointcut*, respectively. To be more specific, a piece of advice is a method or procedure-like construct used to define additional behavior at a join-point, and pointcuts are a means of referring to collections of join-points. [4] As this definition suggests, a pointcut can refer to more than one join-point in the primary abstraction. The process of inserting an aspect's advice into the places designated by the point cut is commonly referred to as *aspect weaving*.

When a developer writes a piece of advice, they specify which pointcut or pointcuts that the advice should be inserted at, as well as the temporal ordering of the insertion of the advice. To that effect, there are three types of advice, called *before*, *after*, and *around* advice. The different types of advice correspond to the temporal placement of the advice at the join-points defined by the pointcuts. For example, if a before advice is inserted at a join-point which refers to the start of a method, then the advice is inserted before the rest of the method body. The temporal placement of before and after advice is clear, but around advice requires some explanation. Around advice is advice that can selectively preempt the normal computation at the join-point. [4] This means that the advice can be run instead of, or in addition to, the code at the join-point. [1]

In addition to dynamic crosscutting, aspects can modify the static structure of other elements in a program, a process called *static crosscutting*. This type of crosscutting, referred to as *introduction* in the AspectJ language, is similar to dynamic crosscutting in that it introduces additional implementation into the primary abstraction. However, instead of modifying the behavior of the primary abstraction at a join-point, it defines or modifies new members in the primary abstraction. For instance, in AspectJ introductions can add methods or fields to an existing class, modify an existing class to inherit from another, implement an interface in an existing class, and convert checked exceptions into

unchecked exceptions. [1] This is a powerful use of aspect-oriented programming, because it not only changes the behavior of components in an application, but also changes their relationship. [1]

3.2.2 Development and Production Aspects

Aspects can be used at many points in the system development life cycle, but generally there are two types of aspects. One type is called a *development aspect*. A development aspect is intended only for use during the development of software, and are expected to be removed from the final application [3]. This means that the functionality that the aspect provides will not be included in a production release. A good example of a development aspect is one that deals with execution logging or contract checking. Generally speaking, a developer will only need a contract checking aspect while he or she is trying to develop and test the software, and would not necessarily want that aspect to be included in the final product.

The other type of aspect is called a *production aspect*. Unlike a development aspect, these aspects deal with code that is intended to be used in the normal operation of the software. [3] The classic example of this type of aspect is an aspect that controls screen updates for a system, similar to the role of the observer in the observer pattern. In fact, many of the classic design patterns can be implemented with aspect-oriented techniques [9]. These are the

types of aspects that are of the most interest to researchers. Aspect-oriented programming has the potential to make it easy to modularize these types of operations to make them easier to create and maintain for future developers. Additional examples of development and production aspects were described in section 2.

3.2.3 Dynamic and Static Aspect Weaving

Finally, there are two ways in which aspects are currently woven into the primary abstraction. The first method is when the weaving process takes place at compile time, rather that at runtime [3]. This is sometimes called static aspect weaving. Static aspect weaving is the method that AspectJ uses to weave aspects into the primary abstraction [1]. The other method is a weaving process that occurs at the program run-time, sometimes referred to as dynamic aspect weaving. This type of aspect weaving has the advantage of allowing aspects to be removed from the primary abstraction, or “unwoven”, at runtime [3]. The JAC project uses this method [5].

4 Software Qualities and Aspect-Oriented Programming

Section 2 describes the ways in which aspect-oriented programming has the potential to increase the quality of a software implementation in regards to

its modularity, maintainability, and readability. However, it is unclear whether or not this potential is currently being realized, and, if not, whether it can be realized in the future. The creators of the AspectJ language have specifically stated that AspectJ is “the basis for an empirical assessment of aspect-oriented programming”, and that the method of that analysis will be based on its usage in real-world situations [4]. Similarly, this section assesses the usefulness of aspect-oriented programming based on its current usage in real world projects. The software qualities examined in this section include the structural and cognitive complexity of the implementation, the correctness of the code, and the testability of the system.

4.1 Software Complexity

We use software engineering principals to make creating software less complex. Most of the qualities we attribute to good software design revolve around how they affect the complexity of that software. For instance, modularized code is considered good because it can reduce the complexity of a large implementation [16].

Although reducing the complexity of a program is important, measuring this attribute is a difficult process that researchers have struggled to work with since the inception of software metrics [10]. In the case of aspect-oriented programming, this task is made even more complicated because there are very

few implementations to study. Further, those that do exist are, for the most part, using tools and processes that have not yet reached the maturity of other software development approaches. Hence, to access the complexity of aspect-oriented programs, this paper relies on the first-hand experience of developers who are actually using aspect-oriented programming.

There are several different ways that a software engineering project can be considered complex, but this paper examines only some of them. The first type of complexity that this paper deals with is the *structural complexity* of the source code. The structural complexity of an implementation can be observed through many attributes but this paper focuses on the effects of aspect-oriented software development on the implementation's modularity. Software developers seek to minimize structural complexity because it can affect the performance of software in many ways, including increased execution time, increases in storage needs and a higher probability of failure [10].

Another form of complexity that an implementation can have is *cognitive complexity*. Cognitive complexity can be defined and measured in terms of the readability and understandability of the implementation by a human. A high cognitive complexity can be very problematic for an organization because it often leads to increased development time, problems in maintaining a program and a heightened probability of defects [10].

It is important to note that, while these two types of complexity are often interrelated, it is not always the case. It is possible to have a program that is very simple, in terms of structural complexity, but is very difficult for a human to read and maintain. As a trivial example, a program written in binary or assembly code can be made very structurally simple, but still remain very complex for a human to read and understand. One of the purposes of high-level languages, such as Java, C and C++, is to reduce the inherent cognitive complexity of programs, while still making it possible to limit the structural complexity at the same time. Aspect-Oriented programming is an attempt to further this ability of other high-level language techniques [3].

4.1.1 How Aspect-Oriented Programming Affects Structural Complexity

The impact of aspect-oriented techniques on implementation complexity is not fully understood, but some real-world examples have shown that aspect-oriented programming can increase the structural complexity of a software implementation. The intended uses of aspect-oriented programming, described in section 2, indicate that aspect-oriented programming has the potential to decrease software complexity, yet real world usage has given us examples of complexity problems related to the size and modularity of aspect-oriented software. This section discusses those examples.

One way to decrease structural complexity is to improve the modularity of an implementation. Real world experience has shown that this is true, but it also demonstrates that it can create new modularity problems for software developers. Avery Moon of Infotone Communications explains this when he says that aspect-oriented programming has “proven to not work well for our large code bases (1 million or more lines)” [11]. Moon later clarifies this statement by pointing out that the root cause of the problem lies in the compilation method for AspectJ. In order to compile a part of a full implementation, compilers such as the one used in AspectJ need access to the entire source tree in order to function [11]. They cannot work if they only have access to a portion of the code, since the implementation details of the aspects are woven throughout all the code in the primary abstraction that is affected by the aspects.

This compilation problem may be an artifact of a more central problem with aspect-oriented programming, which hints at being a problem with modularity. The problem is that using aspect-oriented programming techniques can increase the interdependency between implementation components. This can be considered a violation of software modularity, as it creates a situation where elements of the implementation depend on other parts of the implementation for crucial functionality. While this is true in almost any programming language, what makes aspect-oriented programming unique is that this dependency takes the form of one module relying on another to specify

some of its internal and external behavior. Specifically, elements of the primary abstraction require some of their *internal* implementation details to be provided by their related aspects [3]. From the perspective of the aspects, this problem is indicated by the fact that an aspect may need to know the details of the object in the primary abstraction that it modifies to implement its algorithms [14].

The most prominent problem, seen by Moon, with this modularity issue is that it creates an efficiency problem when a developer is working on a large code base. Compilation times for large software project can sometimes be hours, or even days, and when aspect-oriented programming is used as the basis of such a project, a developer must recompile the entire implementation each time he or she wants to make a change. In contrast, with object-oriented and other contemporary languages, separate parts of the implementation can be compiled independently from each other. This demonstrates, on the surface level, good modularity. Of course, improvements in aspect-oriented technology, such dynamic aspects, have the potential to correct this problem, as the aspect weaving takes place at run time, not compile time. Unfortunately, the base languages of many aspect-oriented programming languages, such a Java and C++, do not readily support such behavior [12].

4.1.2 How Aspect-Oriented Programming Affects Cognitive Complexity

Another goal of software engineering is to increase the readability and understandability of an implementation. Aspect-Oriented techniques have the potential to improve these traits, but an analysis of aspect-oriented techniques and use in real projects has uncovered some problems that aspect-oriented techniques have created with the understandability of software. These problems are discussed in this section. Although many of these problems can be attributed to other factors, such as lack of developer knowledge and training, some of these problems appear to be caused by the language mechanisms themselves.

For instance, an analysis of aspect-oriented techniques shows that using aspect-oriented techniques can reduce developers' ability to work independently because it can increase the cognitive burden of the developers. To explain, the development methods devised with object-oriented and procedural languages divide the work of the implementation into nearly separate domains, or modules. With aspect-oriented techniques, the development team can take this modularization one step further by factoring out crosscutting concerns. While this can have a positive effect, it also has the potential to increase the cognitive burden on the developer.

To explain, this increase in complexity is created because the developer must understand a new type of interaction between their work and others'. Specifically, developers of modularized systems must understand how their

module interacts with other developers' modules, but with aspect-oriented programming, they also have to understand how the behavior of their module is affected by, or affects the behavior of others' modules. To explain, the developer of the primary abstraction must understand how their module's behavior is augmented by aspects that are woven into it, and the aspect developer must understand how their aspects interact with the code in the primary abstraction, and with other modules.

This is not just a problem that presents itself in an analysis of aspect-oriented techniques, as usage of aspect-oriented techniques at Infotone has shown that developers have found aspect-oriented techniques difficult to deal with. These developers are discovering that not being able to work with the "'end result' (i.e. post-weave) of [their] code is somewhere between annoying and unworkable" [11]. In other words, these developers feel that not being able to deal with one complete functional unit of code has hampered their ability to work effectively. The problem lies in the fact that, in using aspect-oriented programming, developers sometimes must work on a unit of code without fully understanding its functionality. Naturally, this means that they cannot fully understand how that code interacts with the rest of the system.

4.2 Software Correctness

Creating software that behaves correctly is one of the primary concerns of any developer. As such, software engineering paradigms have created techniques to help minimize the possibility for defects. Unfortunately, some of the techniques of aspect-oriented programming create new challenges in writing software that is functionally correct. Perhaps the most important challenge to writing functionally correct software introduced by aspect-oriented programming is how to handle the aspect weaving process.

4.2.1 Aspect Weaving and Emergent Properties

A large concern with aspect-oriented techniques develops when multiple aspects are woven into a single primary abstraction. This concern is compounded by the immaturity of current aspect-oriented compilers and frameworks, because with these compilers the weave order is not always well defined. To explain, the AspectJ compiler and other compilers like it perform the aspect weaving process in an arbitrary order. As discussed in section 3.1, the AspectJ language does provide some ways to specify this order, but the current implementation is severely limited in its capabilities. Specifically, it is not always possible to define the ordering of the aspect weave process when large numbers of aspects are in use [1].

This situation is very precarious. If several different related aspects are woven into a primary abstraction, it is possible that these aspects will interact with each other in undesirable ways. This happens because aspects need to cooperate note only with a primary abstraction, but also with other aspects [14]. These undesirable behaviors are called *emergent properties*. An emergent property is "an irreducible feature of a complex whole that cannot be inferred directly from the features of its simpler parts" [15]. In the language of aspect-oriented programming, an emergent property is a behavior of the end product, the woven code, which cannot be attributed to some property of the aspect(s) or the primary abstraction. Put another way, a software defect is an emergent property when it is only present in a fully assembled implementation, rather that the result of a defect in a module or other aggregate part of the whole.

Emergent properties can occur with any type of programming technique, but the aspect weaving process, especially when it is done in an arbitrary order, appears to increase the likelihood of emergent properties. Avery Moon, of Infotone Communications, states that defects created by weave order have always been a problem in their development process, and he goes on to say that these kinds of defects are currently an "ugly, mostly unsolvable problem" [11]. The fact that this problem has been unsolvable in some organizations is clearly a major hurdle that aspect-oriented programming must overcome before it can be used.

Of course, these problems cannot be attributed solely to the concepts of aspect-oriented programming. AspectJ and other aspect-oriented languages are still in their infancy, a fact that Moon readily attributes to being part of the problem. He is aware that most of their weave order problems are “just a legacy of a buggy compiler” [11]. He also points out that:

> “…first the compiler needs to be fixed; then us developers can ‘fix’ our mentality. The compilers have not been stable enough long enough for the mentalities to ‘solidify’” [11].

Clearly, Moon feels that part of the problem lies in the processes that developers use when programming with aspect-oriented languages.

4.2.2 Are Emergent Properties A Big Problem?

Problems with emergent properties can potentially affect both development and production aspects. Developers want their production builds to be defect free, and they also want their tools to aid, not hinder, them in this goal. Emergent properties can be even more insidious in development aspects because they can potentially prevent other defects from being discovered. Because of this, emergent properties created by aspect weaving are potentially a huge concern with aspect-oriented programming. The question is are these

problems prevalent, and are they hard to deal with? Experience at Infotone and the JAC project, as described in this section, has shown that this is the case.

Weave order defects have proven common enough at Infotone that the developers there have changed the way they use aspects, and aspect-oriented programming in general. To be specific, Moon states that their solution to the weave order dilemma is “just to ensure there is no more than one weave really going on in one place” [11]. Although they are still using the concept of aspects, problems with the language concepts and tools have kept from fully utilizing the ideals of aspect-oriented programming. It is clear from this action that weave order defects are a major problem at Infotone.

This problem of emergent properties created by weave order is evident not only with the work that Infotone does, but also with the developers working on Java Aspect Components (JAC). Although the developers of JAC appear to experience problems with less frequency that the developers at Infotone, the problem is still a major concern. Additionally, experience at JAC has shown that defects caused by emergent properties are sometimes very difficult to deal with as well. Renaud Pawlak states that, in reference to finding weave-order defects his team has:

> “...encountered serious and tricky problems (I think about 2 or 3 times). For instance, it may happen that [a] persistency aspect conflicts with

integrity or constraint-checking aspects... Sometime [sic] it takes days to figure out what is going wrong and we need very skilled programmer [sic] to do this debugging. However, it is quite rare (2 or 3 times within a year of aspect-oriented programming)“ [12].

Clearly, the weave order process of aspect-oriented programming introduces new problems in assuring the correctness of an implementation that are both common and sometimes difficult to deal with.

4.3 Software Testability

Aspect-Oriented programming creates concerns not only with the correctness of software, but also with the ability to test that software. As discussed earlier, it introduces a new set of problems to test for in the form of weave order defects. However, the current state of aspect-oriented programming also makes some established testing methods, such as unit testing, difficult or impossible to conduct. This has been demonstrated by the testing problems and procedures that both Infotone and the JAC project group encounter and use when testing their aspect-oriented implementations.

4.3.1 Unit Testing Aspects

The concept of a modularized crosscutting concern, or aspect, is not an easily tested unit of code, unlike classes or procedures. There appear to be no formal methods for testing aspects, and the concept of testing an aspect has proven to be a difficult task. As described by Pawlak, the developers of JAC have only found one way to test aspects, and that is to “try them on sample programs that are representative of the context in which they are going to be used” [12]. While this can test the functionality of the aspect in the context of the woven implementation, it does not test the aspect independently from the primary abstraction. Unfortunately, this can lead to problems in identifying and differentiating between defects in the aspect itself, and emergent properties created through the interaction of the aspect and the primary abstraction. This problem can be compounded if multiple aspects are woven into the primary abstraction, as this situation introduces a new possible source of emergent properties, namely the interaction of the two aspects [13].

Testing problems related to aspects have also been seen at Infotone. In this organization, there is little or no effort made to test aspects separately from the primary abstraction. Instead, the testers and developers first generate a weaved version of the implementation, and test only that [11]. This approach to testing aspects is different from the approached used by the JAC group, but it can still lead to problems with differentiating emergent properties from defects in the aspects and primary abstraction.

Although a partial answer to this problem may lie in new testing technologies specifically designed to handle aspects, the concepts of aspect-oriented design may preclude the ability of thoroughly unit testing an aspect. To explain, the root of this testing problem lies in the fact that an aspect is not a complete functional unit. It depends on being woven into the primary abstraction in order to become a fully developed unit. In many cases, an aspect has no meaning outside the context of the primary abstraction. Hence, a unit test may not be possible on this aspect because it does not represent a testable unit of functionality.

4.3.2 Testing with Development Aspects

Despite these testing problems created by aspect-oriented programming, there are still some benefits that real world usage has uncovered. One benefit is that the use of development aspects can aid in software testing. For example, the JAC team regularly uses "…well tested aspects…to enforce their testing (e.g. a logging or a constraint checking aspect can be used to check the business objects)" [12]. This benefit is one of the stated goals of development aspects; however, as mentioned earlier, this type of functionality is possible to obtain with other readily available tools.

Still, even using development aspects comes with risks. Specifically, development aspects can create emergent properties just a readily as production

aspects. At Infotone, this has been enough of a problem that they “tend to merge down and ‘remove’ from production builds. Where ‘remove’ means condense into the existing code set, rather than leave as aspects” [11]. In other words, the process of “merging down” means using the aspect compiler to create a woven source that is then used as a non aspect-oriented code base. In essence, this is using AspectJ as a preprocessor.

By permanently combining the development aspects and the primary abstraction, the developers at Infotone have lessened the possibility of future problems related to weaving in development aspects. However, they are also sacrificing a large benefit of using aspect-oriented techniques for development code, which is being able to remove development code from a production build.

5 Developers and Aspect-Oriented Programming

The previous section of this paper analyzes the utility of aspect-oriented programming by examining the impact of using aspect-oriented technology on specific software qualities. This section builds upon that analysis by revealing how these problems, and other, non software-quality related problems affect the way developers use aspect-oriented programming technology. This exposition is then followed by an explanation of the potential that these developers still see in aspect-oriented programming. The hope is that this will give the reader a better

understanding of the utility of aspect-oriented programming from the perspective of these developers.

5.1 Factors Limiting Usage

A major factor limiting the usage of aspect-oriented programming in real world projects is its relative immaturity when compared to other methods of software engineering The idea of aspect-oriented programming is relatively new and only recently has its concepts and tools begun to mature. This, of course, means that right now it has had very little opportunity to become widely used for real world purposes. As evidence of this, at the time of this writing, the Aspect-Oriented projects Software Development website lists only six projects that are known to use the techniques of aspect-oriented software development [8]. Further searching for other projects using aspect-oriented programming has not revealed any other such projects. This is an interesting fact since it begs the question of why aspect-oriented technology has not yet been embraced by the development community

Other possible factors limiting the usage of aspect-oriented programming have surfaced as a result of discussions with Avery Moon. Most of these problems appear to be related to either technical problems in the languages that provide aspect-oriented functionality or problems with the concepts of aspect-oriented software design. However, other factors seem to be

limiting the usage of aspect-oriented software. One such possible factor is revealed by Moon's opinion that:

> "the world has found precious few true cross-cutting concerns that admit an automated cut-and-paste solution...and they are the classical examples: logging, security, etc".

If this is true, then perhaps aspect-oriented programming is not being used in real world projects because it is a solution looking for a problem.

Of course, the technical and conceptual problems discussed in section 4 are also factors that are limiting the use of aspect-oriented programming, and in general, Moon has found that aspect-oriented programming has "failed to live up to [his] expectations primarily for operational reasons". What is interesting, however, is that these operational problems have not stopped the developers at Infotone from using aspect-oriented programming techniques, but they have led these programmers to modify their usage of aspect-oriented programming.

One of these usage changes, which was allowing only one aspect to avoid weave order defects, was mentioned in section 4.2.2. Another of these usage changes discussed in section 4.3.2 is that, instead of keeping their code separated into a primary abstraction and aspect code, they merge each new aspect into the code permanently, which is effectively treating aspect-oriented

programming like a preprocessor. Obviously, these technical and conceptual problems have not stopped them from using aspect-oriented programming, but they have limited the ways in which aspect-oriented concepts are applied.

In addition, these problems have also affected how the developers of the JAC project use aspect-oriented programming, though to a lesser degree. As discussed in section 4, they have encountered similar technical problems related to testing and development.

5.2 Do Developers Still See Potential?

As described in the previous section, current usage of aspect-oriented programming is limited by many different factors; however, this does not mean that developers do not still see potential for its use. In fact, quite the opposite is true. Renaud Pawlak of the JAC development team sees a lot of potential in the aspect-oriented programming approach to software development. In fact, his enthusiasm for aspect-oriented programming was one of the driving forces behind the creation of JAC. [12].

In addition, Avery Moon thinks that, regardless of the problems he has encountered to date, aspect-oriented programming can be useful. Specifically, he feels that aspects “best capture how I personally ‘visualize’ very complex (real-world, not toy contexts, like logging, etc) cross-cutting concerns”. He goes on to say that, “they hold the POTENTIAL [sic] to dramatically reduce maintenance

complexity and time, particularly when facing messy code evolution/compatibility issues" [11]. However, Avery feels that the best way to realize the potential of aspect-oriented programming is to use a dynamic aspect weaving, as described in section 3.2.3. Dynamic weaving is of interest to Infotone because it can alleviate some of the problems they have encountered in their use of aspect-oriented programming, such as the compiling problems mentioned in section 4.1.1.

Interestingly, the JAC project was created as an attempt to implement this dynamic weaving. Unfortunately, as the developers of JAC recognize, JAC's implementation of dynamic aspect weaving, which uses Java reflection, is inferior to direct support in the Java Virtual Machine (JVM) for this behavior. The primary problem with the dynamic aspect weaving model of JAC is that it:

> "...has a cost on performances which is due to the java.lang.reflect use. A joinpoint in JAC has an overhead that is similar to the reflection cost in Java (greatly optimized in java 1.4 but still slower than a regular call). Most of the time, this overhead is very neglectable [sic] compared to the aspects inherent overhead (e.g. a persistence aspect). However, it makes JAC not very suited to all the kind of AOSD" [5].

While efficient execution time is not a requirement for all software projects, it is still considered an important quality of many implementations.

It is clear that these developers see potential for aspect-oriented programming, and if the problems discusses in this paper can be resolved or otherwise mitigates, then perhaps aspect-oriented programming can live up to this potential.

6. Discussion

This section presents a discussion of the methods used to create this paper, what was learned in the process, and areas of further research that should be investigated further.

6.1 Research Methods

The concept of aspect-oriented programming is very new, and as such it has not attained a high degree of usage in industry, nor has there been time to thoroughly research its methods. Because of this, several methods of investigation were considered to create this paper including experimentation, analysis of existing projects, and interviews with aspect-oriented developers.

Experimentation with aspect-oriented programming was the first method considered, but it was ruled out for several reasons, the most important being a lack of experience with aspect-oriented software development.

Analyzing existing aspect-oriented projects was the second method considered. This method also proved difficult because, although there are dozens of aspect-oriented compilers and frameworks in development, only a few aspect-oriented software projects could be found. To illustrate, the Aspect-Oriented Software Development website lists only six projects that use aspect-oriented technology [8]. Unfortunately, none of these projects are mature enough to be used for research purposes.

Ultimately, the approach chosen was to interview several developers in industry that use aspect-oriented software development. However, as with analyzing code, this method also suffered from a lack of viable candidates. Eventually, five developers from many different types of projects, including commercial applications and open-source projects were solicited for interviews. Unfortunately, only two we able to respond. The specific developers interviewed were Avery Moon of Infotone Communications, Inc. and Renaud Pawlak of the Java Aspect Components project. Both of these developers have been working with aspect-oriented technology for several years now, and actively use aspect-oriented techniques. Information provided by moon and Pawlak has been used to try an understand how aspect-oriented software development is being used in industry. This paper would not have been possible without their help.

6.2 What was Learned

The literature on aspect-oriented programming examined for this paper extols the many potential benefits that aspect-oriented programming can bring to software design, such as improvements in modularity, decreases in structural and cognitive complexity, and enhancements to the reusability of software. However, evidence from real-world usage has shown that aspect-oriented programming techniques can also cause unique problems with the complexity, correctness and testability of a software implementation.

It is apparent that many of these problems are created by technical problems with the implementation of aspect-oriented languages. This is to be expected, as these tools have had little time to mature. Unfortunately, some of these problems, especially those related to the modularity and correctness of the implementation, appear to originate with the language mechanisms themselves, as discussed in section 4.

Regardless of these problems, the evidence still suggests that developers are optimistic that aspect-oriented programming can provide them with better ways to do their work. The comments and opinions of Avery Moon and Renaud Pawlak discussed in this paper have shown this to be true. Unfortunately, since there are still many problems to fix before it will be generally accepted, it remains to be seen whether aspect-oriented programming can live up to its potential.

6.3 Areas for Further Investigation

This paper has uncovered several areas of concern when dealing with aspect-oriented programming, but it has not attempted to provide an in depth analysis of these problems or possible solutions. Before the utility of aspect-oriented programming can be properly assessed, it is important the following questions be further researched:

- **How can the software complexity problems, both structural and cognitive, created by the use of aspect-oriented techniques be mitigated?** Can the methods of aspect-oriented programming be modified to help prevent these problems, or will human-level procedures to deal with them need to be created?
- **Can the problems in software testability created by aspect-oriented programming be corrected?** Can the risk of creating emergent properties be reduced in some way? Can methods be developed to reduce the burden of debugging weave order defects? Can unit testing be adapted to aspects?
- **Are there sufficient, non-trivial, crosscutting concerns to warrant the use of Aspect-Oriented Programming, or can current software engineering techniques deal sufficiently with these crosscutting concerns?**

- **Can dynamic aspect weaving help solve the technical problems associated with using aspect-oriented software development?**

  Can contemporary languages, such as Java and C++ be enhanced to allow the functionality necessary for dynamic aspect weaving?

7 Conclusion

This paper has examined the uses and problems associated with aspect-oriented software development. This examination has shown that aspect-oriented programming has met with numerous problems that affect the understandability, readability, testability and correctness of software in real-world software projects. However, the developers on these projects are confident that aspect-oriented programming has the potential to increase the quality of software and improve on the method of software development.

References

1. *AspectJ Programming Guide*. 2003, Xerox Corporation.
2. Kiczales, G., et al. *Aspect-Oriented Programming*. in *European Conference on Object-Oriented Programming (ECOOP)*. 1997. Finland: Springer-Verlag.
3. Harbulot, B., *Aspect-Oriented Programming*, in *Department of Computer Science*. 2002, University of Manchester.
4. Kiczales, G., et al. *An Overview of AspectJ*. in *15th European Conference on Object-Oriented Programming*. 2001. Budapest, Hungary: Springer-Verlag.
5. *JAC Programmer's Guide*. 2003, AOPSYS.
6. *AspectJ Documentation Page*. 2003, Xerox Corporation.
7. *JAC Documentation Page*. 2003, AOPSYS.
8. *Aspect-Oriented Software Development*. 2003.
9. Hannemann, J. and G. Kiczales. *Design Pattern Implementation in Java and AspectJ*. in *17th Annual ACM conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 2002.
10. Fenton, N.E. and S.L. Pfleeger, *Software Metrics: A Rigorous & Practical Approach*. 2 ed. 1997, Boston, MA: PWS Publishing Company.
11. Moon, A., *Personal Interview*. 2003.
12. Pawlak, R., *Personal Interview*. 2003.

13. Alexander, R.T. and J.M. Bieman, *Will Aspect-oriented Programming Improve Software Quality?* 2002, Colorado State University.

14. Huang, Jie, *Experience Using AspectJ to Implement Cord.* In Oregon Graduate Institute of Science and Technology. August 2000

15. Holland, J. H., *Emergence: From Chaos to Order.* 1999, Perseus Publishing

16. Meyer, B., *Object-Oriented Software Construction.* 1997, Prentince Hall PRT