

Challenges of Aspect-oriented Technology

Roger T. Alexander
Colorado State University
Department of Computer Science
Fort Collins, Colorado 80523
01-970-491-7026
rta@cs.colostate.edu

James M. Bieman
Colorado State University
Department of Computer Science
Fort Collins, Colorado 80523
01-970-491-7096
bieman@cs.colostate.edu

ABSTRACT

Aspect-oriented technology is a new programming paradigm that is receiving considerable attention from research and practitioner communities alike. It deals with those concerns that *crosscut* the modularity of traditional programming mechanisms, and its objectives include a reduction in the amount of code written and higher cohesion. As with any new technology, aspect-oriented technology has both benefits and costs. In this position paper, we explore these costs in terms of their impact on software engineering. We seek to understand both the strengths and limitations of this new technology, and to raise awareness of the potential negative side effects of its use.

1. INTRODUCTION

Many researchers and industrial practitioners are exploring the benefits and uses of aspect-oriented technology. There seems, however, to be little ongoing research into the costs and effects. At first glance, as with most new technologies, the benefits are promising. However, each new technology brings with it a set of costs. In this position paper, we explore these costs in terms of their impact on software engineering. We seek to understand both the strengths and limitations of this new technology, and to raise awareness of the potential negative side effects of its use. Hopefully, the issues and questions that we identify will help to mature this technology and make it a practical tool for the development of robust and high quality software.

The remainder of this paper is organized as follows. Section 2 presents background information on aspect-oriented technology. Sections 3 through 6 then present problem areas that pose significant challenges to the effective use of this technology. Finally, Section 7 presents conclusions and questions for further research.

2. BACKGROUND

Aspect-oriented programming is a new technology for dealing explicitly with *separation of concerns* in software

development. In particular, it deals with those concerns that *crosscut* the modularity of traditional programming mechanisms. For example, code that implements a particular security policy would have to be distributed across a set of classes and methods that are responsible for enforcing the policy. However, with aspect-oriented technology, the code implementing the security policy could be factored out from all the classes to an aspect. Thus, the aspect localizes in one cohesive place the code that affects the implementation of multiple classes and methods [1, 2].

Aspects make it possible to create cohesive modules that implement specific concerns that otherwise would have to be distributed across many *primary abstractions*. By placing these concerns separately in an aspect, the primary abstractions are made more cohesive since their implementations are relieved of the burden of managing concepts unrelated to their purpose. For example, all code implementing a particular synchronization policy could be placed in a single aspect. Later, this code would be integrated with the classes that must support this policy by a process known as *weaving*. Weaving injects the code of an aspect into well-defined locations, called *joinpoints*, in the syntactic structure of a primary abstraction.

The practical consequence of writing aspects is that less code is written. All the code that would otherwise be distributed throughout a collection of primary abstractions is now localized, thus reducing redundancies. A key observation here is that the originally distributed code actually has a modular structure of its own, and this is the key idea behind aspect-oriented programming [1].

3. UNDERSTANDABILITY

One of the fundamental principles of software engineering is that designs and implementations should exhibit low coupling. In general, software that exhibits these properties is much easier to understand. However, there are cases where this principle is sacrificed to some degree as a trade-off for other benefits afforded by new technology. A notable example of this is the use of inheritance in object-oriented technology where the implementation of descendants are often tightly coupled to their parents. Thus, to understand a child often requires understanding of its parents. Further, a change in the implementation of a parent often requires a change in the child. However, this cost is offset by the benefits of polymorphism and dynamic binding.

Aspect-oriented technology has similar issues. First, since an

aspect cannot stand on its own [3], understanding the aspect requires knowledge of the primary abstractions it is weaved into. The inverse of this is also true: to understand a primary abstraction also requires understanding the aspects that will be woven together. Thus, a many-to-many relationship can exist between aspects and the primary abstractions they integrate with.

To understand one aspect potentially requires the understanding of many others. To exacerbate this, it possible that multiple aspects woven into a primary abstraction class can interact in ways that are difficult to understand and result in emergent behaviors that are unexpected and beyond the composite specification of the woven artifacts. This leads not only to difficulties in understanding, but also has the potential to manifest faults that are extremely difficult to diagnose (see Section 4). The key question to be answered is are the benefits of this technology worth the costs?

4. EMERGENT PROPERTIES AND FAULT RESOLUTION

When a failure occurs, the first challenge is in diagnosing the failure and detecting the fault. In non-aspect-oriented programs, this means examining the code and possibility instrumenting with probes to isolate and localize the fault. With aspect-oriented programs, the model is similar. However, it is not sufficient to solely examine the code of the primary abstraction. Instead, the code of the woven aspects must also be examined. The consequence of the weaving process is that the fault may be located in one of several places. There are four alternatives that must be considered:

- **The fault resides in a portion of the primary abstraction that is not affected by a woven aspect.** The fault is unaffected by the data and control dependencies induced by the woven aspect. Thus, the fault is peculiar to the primary abstraction and could occur if there was no weaving.
- **The fault resides in code that is specific to the aspect, isolated from the woven context.** In this case, the fault would be present in any composition that included the aspect. However, the fault resides in aspect code that is independent of the data and control dependencies induced by the weaving process.
- **The fault is an emergent property that results from some interaction between the aspect and the primary abstraction.** This would occur when the result of the weaving process introduces additional data or control dependences not present in the primary abstraction or the aspect alone. Instead, these dependencies arise from the integration and interaction of code and data between the primary abstraction and the aspect.
- **The fault is an emergent property of a particular combination of aspects woven into the primary abstraction.** This is a more insidious version of the third alternative, but compounded by the integration and interaction of data and control dependences from multiple aspects combined with those occurring in the primary abstraction. The fault may or may not exist with a different combination of aspects with respect to the primary

abstraction.

With the exception of the first, each of the above alternatives likely results in a (possibly non-linear) increase in the testing effort required to achieve a given level of quality.

5. IMPLICIT CHANGES IN SYNTACTIC STRUCTURE AND SEMANTICS

Depending on how they are used, aspects have the potential to alter the syntactic structure and semantics of a primary abstraction. In one scenario, aspects are the result of refactoring code common to many primary abstractions and aggregating the code within an aspect [3]. The justification for doing this is that the code represents a cross-cutting concern that is integrated within many distinct abstractions. The refactoring results in smaller implementations of the respective abstractions, and, to a degree, allows the cross-cutting concern to be treated as a distinct entity of its own. The result of weaving the aspect back into the corresponding abstractions should result in behavior that is identical to that of the original non-factored implementations.

The second scenario is almost the inverse of the first. Instead of refactoring code from primary abstractions and aggregating to form the implementation of the aspect, the aspect is defined independently with respect to some cross-cutting concern not present in the primary abstractions (e.g. a synchronization or security policy) [4]. In this model, the cognitive burden shifts from understanding the commonalities of existing code to that of defining a new behavior that must be *pushed* into the primary abstractions. This shift in burden requires that the aspect author understand, at a detailed level, both the syntactic structure and semantics of each primary abstraction that will be affected by the aspect.

Regardless of which scenario is used, control and data dependencies of the composition resulting from the weaving process will be different from that of the primary abstraction. Also, in most cases, the control and data dependencies of the aspect are incomplete. This will be the case when the code and data dependencies of the aspect are dependent upon the context provided by the primary abstraction. Thus, it will not be until weave-time that the dependencies are resolved. Further, since an aspect has the potential to be woven into many primary abstractions, the set of concrete control and data dependencies that result are likely to be disparate.

6. EFFECTS ON COGNITIVE BURDEN

Weaving results in a change in the cognitive model of the author of a primary abstraction A , potentially leading to *cognitive non-determinism*. Each woven aspect that induces mutual data and control dependencies with A increases the cognitive distance between the woven implementation I_W and A 's implementation I_A . Thus, what the author knew to be true of I_A may no longer be true of I_W . The root of this difficulty is that weaving can alter base assumptions made by the author of a I_A , and has the potential to inject new assumptions in I_W that are inconsistent with those of I_A .

Another effect on cognitive burden is the specification of the woven artifact W . Weaving necessarily begins with the

specification of A that forms the base of W , but must also account for the behavioral modifications induced by the woven aspects. From the perspective of a client of A , the specification of W needs to be behaviorally compatible with A 's. Thus, a challenge for an aspect author is to ensure that the behavior of a woven artifact is no stronger than that of the primary abstraction it is based on.

How does an author know that his aspect will not cause undesirable emergent properties after weaving? This is particularly difficult if the aspect is to be woven with other aspects and with potentially many different primary abstractions.

A further complication arises when the collection of aspects to be woven are written by different authors (a likely scenario in a large system). For this to be effective, each author must have knowledge of the set of primary abstractions that their aspects can be woven with. Further, each must have knowledge of the other aspects that they make use of, either by direct composition or indirectly as the result of weaving.

7. CONCLUSION

This position paper has presented a number of areas that are problematic for the effective use of aspect-oriented technology. Many of these raise significant concerns that should be understood before this technology can mature. To that end, the following research questions need further investigation:

- **How do we measure the complexity that results from the weaving process?** Can this be predicted prior to weaving?
 - **Can we control or minimize the cognitive distance induced by the weaving process?** Are there ways to model the effects of a set of aspects on a primary abstraction, making apparent the effects of weaving?
 - **How do we maintain aspect-oriented programs?** Similar to the fragile-base class problem [5], changes to the primary abstraction that form the basis for a woven composition have the potential to require changes to the woven aspects. Also, changes to woven aspects may induce faults in other aspects. Thus, mechanisms are needed to understand the actual extent and impact of a potential change.
- **How do we effectively test aspect-oriented programs?** What new test adequacy criteria must be defined? Are the existing techniques sufficient?
 - **How do we analyze aspect-oriented programs?** What representations are needed? Representations that simply reflect the static pre-woven structure are necessary, but not sufficient. New representations and tools are needed that take into the account the effects of weaving and that can identify potential emergent properties that can induce faults.

Author Information

Roger Alexander is an Associate Professor of Computer Science at Colorado State University. He spent many years in industry as a software developer (and researcher) including experience at the Software Productivity Consortium, Michael Jackson Systems, and Cigital (formerly Reliable Software Technologies). His work is focused on the testing, reverse engineering, maintenance, design, and implementation of high-quality software.

Jim Bieman is the Editor-in-Chief of the *Software Quality Journal* and Associate Professor of Computer Science at Colorado State University. His work is focused on the evaluation and improvement of software design quality.

REFERENCES

1. Elrad, T., R.E. Filman, and A. Bader, *Aspect-oriented programming: Introduction*. Communications of the ACM, 2001. 44(10): p. 29-32.
2. Elrad, T., et al., *Discussing aspects of AOP*. Communications of the ACM, 2001. 44(10): p. 33-38.
3. Kiczales, G., et al. *An Overview of AspectJ*. in *15th European Conference on Object-Oriented Programming*. 2001. Budapest, Hungary.
4. The AspectJ Team, *The AspectJ(TM) Programming Guide*. 2002, Xerox Corporation.
5. Mikhajlov, L. and E. Sekerinski. *A Study of The Fragile Base Class Problem*. in *12th European Conference on Object-Oriented Programming (ECOOP '98)*. 1998. Brussels, Belgium: Springer-Verlag.