*Software Quality Journal* **Vol. 12 No. 2, 2004**

# Editorial: Aspect-oriented Technology and Software Quality

Aspect-oriented technology is a new programming paradigm that is receiving considerable attention from both the research and practitioner communities. Aspect-orientation involves software development concerns that *crosscut* the modularity of traditional programming mechanisms. Among the claimed benefits of this technology is a reduction in the amount of code written and higher cohesion. As with any new technology, aspect-oriented technology has both benefits and costs (Alexander, 2003). Here we examine these costs in terms of their impact on software engineering. We seek to understand both the strengths and limitations of this new technology. However, here we aim to raise awareness of the potential negative side effects of its use.

## Benefits often come with costs.

Although many researchers and industrial practitioners are exploring the benefits and uses of aspect-oriented technology, we find little ongoing research into the costs and effects. At a first glace, as with most new technologies, the benefits are promising. However, each new technology brings with it a set of costs, and aspect-oriented technology is no exception. If aspect-oriented methods are adopted, there will be an impact on software engineering. A better understanding of the limitations of this new technology will help to raise awareness of potential negative side effects. 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.

## What is aspect-orientation?

Aspect-oriented programming is a new technology for dealing explicitly with *separation of concerns* in software development. In particular, it supports modular programming to implement concerns that *crosscut* the modularity of traditional programming mechanisms. For example, code that implements a particular security policy is typically distributed across a set of classes and methods that must enforce the policy. However, with aspect-oriented technology, the code implementing the security policy is factored out from all the classes into one aspect. Thus, the aspect localizes in one cohesive place the code that affects the implementation of multiple classes and methods (Elrad et al., 2001a; Elrad et al., 2001b).

Aspects make it possible to create cohesive modules that implement specific concerns that otherwise would have to be distributed across many *primary concerns*. By placing these concerns separately in an aspect, the primary concerns are made more cohesive --- implementations of primary concerns will not need to manage concepts unrelated to their purpose. For example, all code implementing a particular sychronization 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*) into the syntactic structure of a primary concern.

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 concerns is now localized, thus reducing redundancies. A key observation here is that the code that was originally distributed actually has a modular structure of its own which is apparent when it is kept in one aspect. This notion of modularity is the key idea behind aspect-oriented programming (Elrad et al., 2001a ).

## Understandability effects.

One fundamental principle of software engineering is that designs and implementations should exhibit low coupling. In general, software with lower coupling is much easier to understand. However, sometimes this principle is sacrificed to some degree as a trade-off to gain other benefits afforded by new technology. A notable example is the use of inheritance in object-oriented technology where the implementation of descendants are often tightly coupled to their parents. To understand a child class often requires understanding of its parents and other ancestors. Further, a change in the implementation of an ancestor 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 (Kiczales et al., 2001); understanding an aspect requires knowledge of the primary concerns it is woven into. The inverse also holds: to understand a primary concern also requires understanding the aspects that will be woven together. Thus, a many-to-many relationship can exist between aspects and the primary concerns that they integrate with.

To understand one aspect potentially requires the understanding of many others. To make matters worse, multiple aspects that are woven into a primary concern 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. Not only will the software be difficult to understanding, but the weaving process may introduce faults that are extremely difficult to diagnose. The key question to be answered is are the benefits of this technology worth the costs?

## Emergent properties and fault resolution.

When a failure occurs, the first challenge is to diagnose the failure and detect the fault. In non-aspect-oriented programs, you must examine the code, and possibily instrument it with probes to isolate and localize the fault. With aspect-oriented programs, you might use a similar method. However, it is not sufficient to solely examine the code of the primary concern. Instead, you must also examine the code of the woven aspect. The consequence of the weaving process is that the fault may be located in one of several places. Consider the following four alternatives:

1. **The fault resides in a portion of the primary concern 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 concern and could occur if there was no weaving.

2. **The fault resides in code that is specific to the aspect and is isolated from the woven context**. In this case, the fault will 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.
3. **The fault is an emergent property that results from some interaction between the aspect and the primary concern**. This will occur when the result of the weaving process introduces additional data or control dependences not present in the primary concern or the aspect alone. Instead, these dependencies arise from the integration and interaction of code and data between the primary concern and the aspect.
4. **The fault is an emergent property of a particular combination of aspects woven into the primary concern**. 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 concern. The fault may or may not exist with a different combination of aspects with respect to the primary concern.

Alternatives 1—4 are likely to cause a (possibly non-linear) increase in the testing effort required to achieve a given level of quality.

## Implicit changes in syntactic structure and semantics.

Depending on how they are used, aspects may alter the syntactic structure and semantics of a primary concern. In one scenario, aspects are the result of refactoring code common to many primary concerns and aggregating the code within an aspect (Kiczales et al., 2001). The justification for doing this is that the code represents a cross-cutting concern that is integrated within many distinct concerns. The refactoring results in smaller implementations of the respective concerns, 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 concerns should result in behavior that is identical to that of the original non-factored implementations.

A second scenario is almost the inverse of the first. Instead of refactoring code from primary concerns 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 concerns (e.g. a synchronization or security policy) (The AspectJ Team, 2002). 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 concerns. This shift in burden requires that the aspect author understand, at a detailed level, both the syntactic structure and semantics of each primary concern that will be affected by the aspect.

Regardless of the scenario, control and data dependencies of the composition resulting from the weaving process will be different from that of the primary concern. Also, in most cases, the control and data dependencies of the aspect are incomplete. This occurs when the code and data dependencies of the aspect are dependent upon the context provided by the primary concern. 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 concerns, the set of concrete control and data dependencies that result are likely to be disparate.

## Effects on cognitive burden.

Weaving results in a change in the cognitive model of the author of a primary concern, say concern $A$, 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 know longer be true of $I_W$. The root of the problem is that weaving can alter base assumptions made by the author of a $I_A$, and may inject new assumptions into $I_W$ 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 concern it is based on.

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

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 concerns 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.

## Questions to answer.

The effective use of aspect-oriented technology will require the solution to the problems that we have pointed out. These problems will need to be solved before aspect technology can mature. Clearly, we would like to see research to find answers to the following questions:

- **How do we measure the "complexity" that results from the weaving process?** Can the complexity of a woven system 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 concern, making apparent the effects of weaving?
- **How do we maintain aspect-oriented programs?** Similar to the fragile-base class problem (Mikhajlov, 1998), changes to the primary concern 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.

Answering these questions is a challenge to the software engineering research community.

Roger Alexander and James Bieman
Fort Collins, Colorado
U.S.A

## References

Aldrich, J. 2000, Challenge Problems for Separation of Concerns, *Proc. OOPSLA 2000 workshop on Advanced Separation of Concerns*.

Alexander, R.T., The Real Costs of Aspect-Oriented Programming, *IEEE Software*, 20(6):91-93, November/December 2003..

Elrad, T., Filman, R. E. and Bader, A. 2001a, Aspect-oriented programming: Introduction, *Communications of the ACM*, 44(10): 29-32.

Elrad, T., Aksits, M, Kiczales, G. Lieberherr, K. and Ossher, H. 2001b, Discussing aspects of AOP, *Communications of the ACM*, 44(10): 33-38.

Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J. and Griswold, W.G.2001, An Overview of Aspect, *Proc.15th European Conference on Object-Oriented Programming*, Budapest, Hungary.

Mikhajlov, L and Sekerinski, E. 1998, A Study of The Fragile Base Class Problem, *Proc. 12th European Conference on Object-Oriented Programming (ECOOP '98}*, Springer-Verlag, 445: 355-382}.

Pace, J.A. D, and Campo, M.R. 2001, Analyzing the role of aspects in software design, *Communications of the ACM*, 44(10): 66-73.

The AspectJ Team 2002, *The AspectJ(TM) Programming Guide*, Xerox Corporation.