

Submitted as White Paper to the Workshop on  
New Visions for Software Design and Productivity:  
Research and Applications  
Vanderbilt University  
Nashville, Tennessee

## **Coupling Mechanisms in Aspect-Oriented Software**

Karl Lieberherr  
College of Computer Science  
Northeastern University  
Boston, MA 02115  
[lieber@ccs.neu.edu](mailto:lieber@ccs.neu.edu), 617 373 2077  
October 28, 2001

### **Abstract:**

The paper describes the need for the study and development of better crosscutting coupling mechanisms in software components to reduce brittleness. A software system consists of multiple crosscutting concerns and it is not only required to cleanly encapsulate and localize those concerns into components but in addition to control the coupling between the encapsulated concerns. Aspect-Oriented Programming has started to move in this direction; Adaptive Programming has since its inception studied crosscutting coupling mechanisms.

The innovative claim is new mechanisms to abstract from program concerns and to write the software in terms of those abstractions. This allows us to write programs that are less brittle, and easier to develop and maintain.

Technical barriers are how to properly formulate abstractions over program concerns and how to prove useful properties of concerns encapsulated as components that are parameterized by concern abstractions.

### ***Introduction***

It is well known that software is brittle and "small" changes can affect a program in many places. Recent advances in Aspect-Oriented Software Development (AOSD) (Aspect-Oriented Programming, Adaptive Programming, Composition Filters, Subject-oriented Programming and Multi-Dimensional Separation of Concerns [1]) have started to address

issues of scattering and tangling, but much more work is needed. It is important that aspects that encapsulate crosscutting concerns be loosely coupled to program concerns. Otherwise, we run again into issues of brittleness.

### ***Program Concern Abstractions***

We present three interesting program concern abstractions. A program concern is a concern related to the program under consideration rather than a concern that is tightly linked to a requirement. We call such concerns requirement concerns. Program concerns are only indirectly linked to the requirements. Examples of requirement concerns are "how to implement persistence" or "how to settle an account".

### **Object Structure Concern**

The object structure concern is about how objects are structured and it is typically expressed by a class graph (e.g., a UML class diagram). A suitable abstraction of a class graph for programming is a non-deterministic finite state automaton (NFA) that induces traversals in the objects belonging to the class graph. We write our programs with respect to a NFA. The NFA describes the important properties of the structural concern that are relevant to the current concern that is about implementing a specific behavior. How can we program against the NFA? A natural way is to annotate the NFA with code that is executed before entering a state and before leaving a state and before and after state transitions. The code defined in the NFA is executed as the objects of the class graph are traversed. It is useful to keep the code annotations defined in the NFA separate from the NFA itself because they address different concerns that might be separately reusable. Therefore, programming in this style uses a triple: (CLASS GRAPH, NFA, NFA ANNOTATION). Sometimes, this triple is called: (CLASS GRAPH, TRAVERSAL SPECIFICATION, VISITOR) because we have a variant of the Visitor Design Pattern.

For defining a behavior, it is useful to define the NFA in two steps, starting with the class graph of the application. In the first step, we eliminate from the class graph nodes and edges that are not needed for the behavior. In the second step, we define the NFA.

We call this programming style Pure AP, for Pure Adaptive Programming. In Adaptive Programming (AP) [3] several additional features are used. AP has also been adopted for model-integrated computing [2].

## Call graph concern

We use AspectJ (see articles in [1]) to illustrate the call graph concern. The call graph concern is, like the object structure concern, intimately linked to the current program: it is about how the methods in the current program call each other. A typical issue during programming is how to transport objects through a set of function calls. The following is adapted from [1].

Consider implementing functionality that would allow a client of a figure editor to set the color of any figure elements that are created. This requires passing a color, from the client down through the calls that lead to the figure element factory. All programmers are familiar with the inconvenience of adding a first argument to a number of methods just to pass this kind of context information.

Using AspectJ, this kind of context passing can be implemented in a modular way without duplicating the details of the program call structure in the aspect. We can abstract over the call graph and write the aspect generically so that it works with many different call graphs.

## Decision Point Concern

An alternative way to abstract over the call graph concern is to combine ideas from predicate dispatching [6] and AOP [1]. An important concern during programming is the decision point concern. The programmer is concerned about the precise conditions under which a piece of code should execute [5]. We call the conditions branch conditions and a branch condition together with the code to be executed when the condition is true: a branch. A branch condition may involve the run-time type of the arguments, the state of the arguments, the message itself, the enclosing branch from which the message was sent: ideally any element of the program state at the decision point. This goal can be approached by reifying decision points as program entities and specifying branch conditions as logical combinations of predicate expressions over these decision point entities. We call this approach to programming BOP for Branch-Oriented Programming or *Bedingung-Orientierte Programmierung*. BOP unifies both OOP and AOP and is motivated by predicate dispatching [6]. In BOP, a program consists of a set of decision point branches.

In BOP, we can easily abstract over the detailed structure of decision points because the decision points are reified and we can write branch conditions. For example, if a method with name M and argument a=1 was called at an

earlier decision point, execute the current branch body. Referring to earlier decision points combined with the capability to write around-branches, allows us to write reusable aspects.

The decision point concern and the call graph concern are similar but the underlying model is different. BOP is a symmetric AOP model where methods and advice are unified by branches.

### ***Relieving the Programmer from Details***

A concern that involves a group of connected objects is not easy to modularize because the connective structure between the objects often contains noise that we need to abstract out if the aspect is to be useful and reusable. We want to relieve the programmer from the details of the connective structure. We have indicated how Pure AP, AspectJ and BOP relieve the programmer from details.

Our work on Aspectual Collaborations (AC) also falls into the domain of relieving the programmer from the details of program concerns [4, 7]. Each AC is written with respect to its own class graph, called a participant graph, which only contains the structure that is needed for the behavior. An AC is typically embedded into an AC that has a larger class graph that contains many accidental details with respect to the first AC. An AC is similar to a UML collaboration, but a distinguishing feature is the aspectual methods. They enhance methods defined in other collaborations. It is beneficial to use AP to write the code for the AC, provided it encapsulates a traversal-related concern.

### ***Conclusions***

Pure AP, AspectJ, BOP, as well as AC offer interesting techniques for describing connective structure abstractly but a more systematic investigation is needed. Interesting questions are:

1. What kind of program structure concerns do we want to abstract over. We showed three examples: object structure and call graph and decision points.
2. What are appropriate abstractions? The AspectJ approach uses an abstraction that defines all nodes reachable from a given node. The same abstraction is also used in AP but relative to a different graph. In BOP we can write arbitrary programs to abstract over the decision point structure.

3. What are the constraints that must hold for an aspect to be applicable?  
Aspects written in terms of program concern abstractions are reusable and their range of applicability must be managed.

## Acknowledgements

This research has been supported by several DARPA and NSF grants as well as by industrial support.

## References

- [1] Tzilla Elrad and Robert Filman and Atef Bader, Special Issue on AOP, 12 papers, Communications of the ACM, Volume 44, Number 10, October 2001.
- [2] Gabor Karsai, Structured Specification of Model Interpreters, ECBS 1999. [www.isis.vanderbilt.edu](http://www.isis.vanderbilt.edu)
- [3] Karl Lieberherr, Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns, PWS Publishing Company, Boston 1996. [www.ccs.neu.edu/research/demeter](http://www.ccs.neu.edu/research/demeter)
- [4] Karl Lieberherr and David Lorenz and Mira Mezini, Programming with Aspectual Components, College of Computer Science, Northeastern University, 1999, March, NU-CCS-99-01. [www.ccs.neu.edu/research/demeter](http://www.ccs.neu.edu/research/demeter)
- [5] Doug Orleans, Incremental Programming with Extensible Decisions, College of Computer Science, Northeastern University, 2001, October, NU-CCS-01-11. [www.ccs.neu.edu/research/demeter](http://www.ccs.neu.edu/research/demeter)
- [6] Michael D. Ernst, Craig Kaplan, and Craig Chambers. Predicate dispatching: A unified theory of dispatch. In Proceedings of ECOOP '98, the 12th European Conference on Object-Oriented Programming, pages 186–211, Brussels, Belgium, July 20–24, 1998.
- [7] Johan Ovlinger, Aspectual Collaborations and Modular Programming, College of Computer Science, Northeastern University", 2000, November, NU-CCS-2000-04, Boston, MA, 1-27.