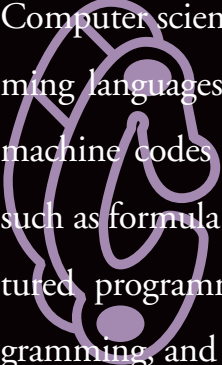




ASPECT-ORIENTED PROGRAMMING



Computer science has experienced an evolution in programming languages and systems from the crude assembly and machine codes of the earliest computers through concepts such as formula translation, procedural programming, structured programming, functional programming, logic programming, and programming with abstract data types. Each of these steps in programming technology has advanced our ability to achieve clear separation of concerns at the source code level.

**Tzilla Elrad, Robert E. Filman,
and Atef Bader, Guest Editors**

Currently, the dominant programming paradigm is object-oriented programming—the idea that one builds a software system by decomposing a problem into objects and then writing the code of those objects. Such objects abstract together behavior and data into a single conceptual (and physical) entity. Object-orientation is reflected in the entire spectrum of current software development method-

ologies and tools—we have OO methodologies, analysis and design tools, and OO programming languages. Writing complex applications such as graphical user interfaces, operating systems, and distributed applications while maintaining comprehensible source code has been made possible with OOP.

Success at developing simpler systems leads to aspirations for greater complexity. Object orientation is a clever idea, but has certain limitations. We are now seeing that many requirements do not decompose neatly into behavior centered on a single locus. Object technology has difficulty localizing concerns involving global constraints and pandemic behaviors, appropriately segregating concerns, and applying domain-specific knowledge. Post-object programming (POP) mechanisms that look to increase the expressiveness of the OO paradigm are a fertile arena for current research. Examples of POP technologies include domain-specific languages, generative programming, generic programming, constraint languages, reflection and metaprogramming, feature-oriented development, views/viewpoints, and asynchronous message brokering. (Czarnecki and Eisenecker's book includes a good survey of many of these technologies [2].)



The topic of this special section is one important POP technology: aspect-oriented programming (AOP).¹ AOP is based on the idea that computer systems are better programmed by separately specifying the various *concerns* (properties or areas of interest) of a system and some description of their relationships, and then relying on mechanisms in the underlying AOP environment to weave or compose them together into a coherent program. Concerns can range from high-level notions like security and quality of service to low-level notions such as caching and buffering. They can be functional, like features or business rules, or nonfunctional (systemic), such as synchronization and transaction management. While the tendency in OOP is to find commonality among classes and push it up in the inheritance tree, AOP attempts to realize scattered concerns as first-class elements, and eject them horizontally from the object structure.

Any structural realization of a system will find that some concerns are neatly localized within a specific structural piece, while others cross multiple elements. AOP is focused on mechanisms for simplifying the

realization of such crosscutting concerns. Aspectual requirements are concerns that (for common structural decompositions) introduce crosscutting in the implementation. Examples of aspectual requirements include synchronization policies that require a whole set of operations to following a consistent locking protocol, traversals of complex object graphs that require global information, accounting mechanisms that must be notified of every chargeable action, fault tolerance mechanisms that require consistent creation of redundant copies, and quality of service concerns that require fine tuning of system priorities.

One might argue that all programming languages since Fortran have had a way of separating out concerns by creating and explicitly calling subprograms. Subprograms are a great idea. We thoroughly endorse their use. Just as OOP did not discard the ideas of block structure and structured programming, AOP does not reject existing technology. However, often the expression of a concern cannot be neatly realized by a call to a subroutine. A concern whose code becomes tangled into other structural elements becomes a mess. To ameliorate this problem, AOP offers aspects: mechanisms beyond subroutines and inheritance for localizing the expression of a crosscutting concern. AOP systems also must provide some mechanism for weaving aspects and base code into a coherent system.

Subprograms have another disadvantage. They require both knowledge and cooperation on the part of the programmers of the calling components. That is, one must know to explicitly invoke a subroutine, and know how to invoke it. AOP systems offer implicit invocation mechanisms for invoking behavior in code whose writers were unaware of the additional concerns.²

Separating the expressions of multiple concerns in programming systems promises simpler system evolution, more comprehensible systems, adaptability, customizability, and easier reuse. Several elements of AOP work toward these goals. By aggregating crosscutting concerns into aspects, AOP congeals into a single textual structure behavior that conventional programming would distribute throughout the code, both making the aspect code and the target easier to understand. Implicit invocation is a virtue in this age of increased software complexity, as domain experts for an application are unlikely to be familiar with intrica-

¹The term aspect-oriented programming is attributed to Kiczales et al. Related work includes adaptive programming, composition filters, multidimensional separation of concerns, and subject-oriented programming. The editors of this special section choose to use the term aspect-oriented programming to describe the space of programmatic mechanisms for expressing crosscutting concerns, primarily because it is a catchier, more commonly used, and less subject to ambiguous interpretation.

²Implicit invocation is a concept that in the past Robert Filman has termed obliviousness, stated as: "The distinguishing characteristic of aspect-oriented programming systems (qua programming systems) is that they provide quantification and obliviousness. Quantification is the idea that one can write unitary and separate statements that have effect in many, non-local places in a programming system; obliviousness, that the places these quantifications applied did not have to be specifically prepared to receive these enhancements." [1]

cies of specialized algorithms for distribution, authentication, access control, synchronization, encryption, redundancy, and so forth, and cannot be trusted to always invoke them appropriately in their programs. This becomes more important in an era of evolving programs, where programmers can't always anticipate the demands their programs will need to deal with in the future. Aspects themselves can be reused, and aspects can be connectors for other components.

AOP Issues

At its essence, AOP is a programming technique. Like all programming techniques, AOP must address both what the programmer can say and how the computer system will realize the program in a working system.³ Thus, a goal of AOP systems is not only to provide a way of expressing crosscutting concerns in computational systems, but also to ensure these mechanisms are conceptually straightforward and have efficient implementations.

In reading the articles in this section, readers should be sensitive to the nuances of different approaches to different problems. A major distinction among systems lies in the technology of combining programs and aspects. Clear-box approaches to AOP can examine the program and aspect internals, producing a mixture of program and aspects. On the other hand, black-box approaches shroud components with aspect wrappers. Other issues to keep in mind include:

- **How an AOP system specifies aspects.** This includes defining the join points, those places where aspect code interacts with the rest of the system; aspect parameterization, the extent to which aspects can be customized for a particular use; source encapsulation, the source-code requirements for specifying points to be joined to; and the OO dependency, the extent to which the AOP mechanism can be used in non-OO programming systems.
- **What composition mechanisms the system provides.** This includes the issues of the existence of dominant decomposition (Is there one decomposition to which aspects are applied, or are all concerns treated as equals); explicit composition languages (Does the system have a separate lan-

guage for describing which aspects are applied where? Is this a domain-specific or general language?); the visibility of aspects to each other; the symmetry of privileges between the main program and aspects; whether aspects are purely monotonic or also able to delete behavior; what mechanism is provided for resolving conflicts among aspects; and to what extent the composition and execution depend on the external state.

- **Implementation mechanisms.** This includes the static/dynamic distinction, that is, whether compositions are determined statically at compile time or dynamically as a system is running; modular compilations, whether elements of the program can be compiled separately; deployment-in-place, whether AOP mechanisms can be applied to an existing system; target representation, whether AOP mechanisms are applied to source code, byte code or object code; and verification, whether there are mechanisms for verifying compositions.
- **Decoupling.** This includes obliviousness, whether the writer of the main code be aware that aspects will be applied to it; intimacy, what the programmer has to do to prepare code for aspects; and globality versus locality, whether aspects apply to the program as a whole or only parts of it.
- **Software process.** This includes overall process, what methodology or framework the system provides for organizing the system-building activity; reusability, which aspect mechanisms enable aspect reuse; domain-specificity, whether the aspect mechanism is general or applicable to a specific domain; analyzability, whether one can analyze the performance of the aspect system; and testability, which mechanisms enable debugging aspects and systems.

The articles in this special section cover the main intellectual branches of AOP, their historical development, applications of AOP, and its enhancement toward a complete AOP software engineering process. AOP is the convergence of many independent research paths. Many cite inspiration from Dijkstra and Parnas's early expressions of the fundamental software engineering principle of decomposition [3, 4].

We begin the section with a discussion between several luminaries of the field, moderated by guest editor Tzila Elrad. This discussion delves into many aspects of AOP, taking the form of Frequently Asked Questions about AOP, and we hope you find the participants' contrasting perspectives illuminating.

Karl Lieberherr, one of the early researchers in the field, has argued "The Law of Demeter," which states: Objects should only have knowledge of closely related

³That latter goal carries a caution: Implementers of programming languages have long understood they can arbitrarily manipulate the behavior of a program by changing the interpreter of that program. For example, an interpreter that checked at each execution step for applicable aspects would provide a workable though clunky AOP mechanism. This is straightforward for a skilled reflection programmer to do in reflective programming systems. The problems with reflection as an AOP solution are that skilled reflective programmers are scarce, and, more importantly, reflective systems typically impose a large (one to two orders of magnitude) performance burden over more direct encoding.

objects. This led to work on new programming structures to separate the expression of behavioral concerns from structural ones. Lieberherr et al.'s article, "Aspect-Oriented Programming with Adaptive Methods," illustrates the use of adaptive methods to avoid tangling by abstracting over the class structure.

Using insights gained during their work on software environments and tool integration in the late 1980s, William Harrison and Harold Ossher pioneered the notion of separate specification of different class hierarchies, each implementing a concern, with subsequent composition of appropriate hierarchies to build system variants (subject-oriented programming). With Peri Tarr, they evolved this approach to allow multiple, simultaneous decompositions of the same software, and extraction of concerns from existing software. This theme is elaborated in Ossher and Tarr's "Using Multidimensional Separation of Concerns to (Re)shape Evolving Software."

Mehmet Aksit and his group at Twente University have been the earliest and most prominent proponents of filter-based approaches to AOP. In the late 1980s, the filter principle was developed to express a generic

data abstraction mechanism. This was extended to represent various types of concerns such as multiple views, synchronization, interobject communication and real-time specifications in a composable way. In this issue, Bergmans and Aksit's article, "Composing Multiple Concerns Using Composition Filters," describes how to embody aspects in explicit filters, by wrapping the filters around base components.

The AOP group at Xerox PARC came to AOP through their work on reflection and metaobject protocols. Through experience with reflection and discussions with the other groups mentioned here, they crystallized the idea that the focus on crosscutting concerns is what distinguishes AOP from previous separation of concerns technologies. The PARC group developed a series of AO languages, culminating in AspectJ—"Getting Started With AspectJ" provides a brief introduction to their system.

We have included several shorter articles illustrating particular issues in developing AOP systems and demonstrating the application of AOP in real systems. These include two assessments of the value of AOP technology: "Analyzing the Role of Aspects in Software Design" and "Does Aspect-Oriented Programming Work?"; two discussions of the applications of AOP to systems development in "Structuring Operating System Aspects" and "A Layered Approach to Building Open Aspect-Oriented Systems"; the application of AOP to virtual design in "Handling Crosscutting Constraints in Domain-Specific Modeling"; and an overview of using reflection techniques to implement aspects in "Aspect-Oriented Programming Using Reflection and Metaobject Protocols."

Links to research groups described here, and to additional groups, can be found in the Aspect-Oriented Software Development Web site [1]. **C**

Further Information

AOP is growing rapidly, and we see AOP applications in many areas such as middleware, security, fault tolerance, quality of service, and operating systems. We are also beginning to see AO applied at various stages of the software life cycle. It is not yet a fully mature discipline and we are looking forward to seeing more AO software development in research and applications.

As a burgeoning research area, there are annual workshops on separation of concerns at the European Conference on Object-Oriented Programming (ECOOP) since 1997, at the ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA) since 1999, and at the International Conference on Software Engineering (ICSE) in 1998, 2000, and 2001. A good starting point for further reading is the Aspect-Oriented Software Development Web site [1], which contains links to the foundational AOP works mentioned here, papers from the annual workshops, as well as links to over a dozen groups working on AOP. The field has matured sufficiently to generate its own independent conference: The First International Conference on Aspect-Oriented Software Development is planned for Twente, The Netherlands in April, 2002; see trese.cs.utwente.nl/aosd2002.htm. **C**

REFERENCES

1. Aspect-Oriented Software Development Web site; aosd.net.
2. Czarnecki, K. and Eisenecker, U.W. *Generative Programming: Methods, Tools, and Applications*. Addison Wesley, Boston, 2000.
3. Dijkstra, E.W. *A Discipline of Programming*. Prentice Hall, Englewood Cliffs, NJ, 1976.
4. Parnas, D.L. On the criteria to be used in decomposing systems into modules. *Commun. ACM* 15, 2 (Feb. 1972).

TZILLA ELRAD (elrad@iit.edu) is a research professor leading the Concurrent Programming Research Group in the Department of Computer Science at the Illinois Institute of Technology in Chicago.

ROBERT E. FILMAN (rfilman@riacs.edu) is a scientist at the Research Institute for Advanced Computer Science, NASA Ames Research Center, Moffet Field, CA.

ATEF BADER (abader@lucent.com) is a member of the Technical Staff at Lucent Technologies in Chicago, IL.
