

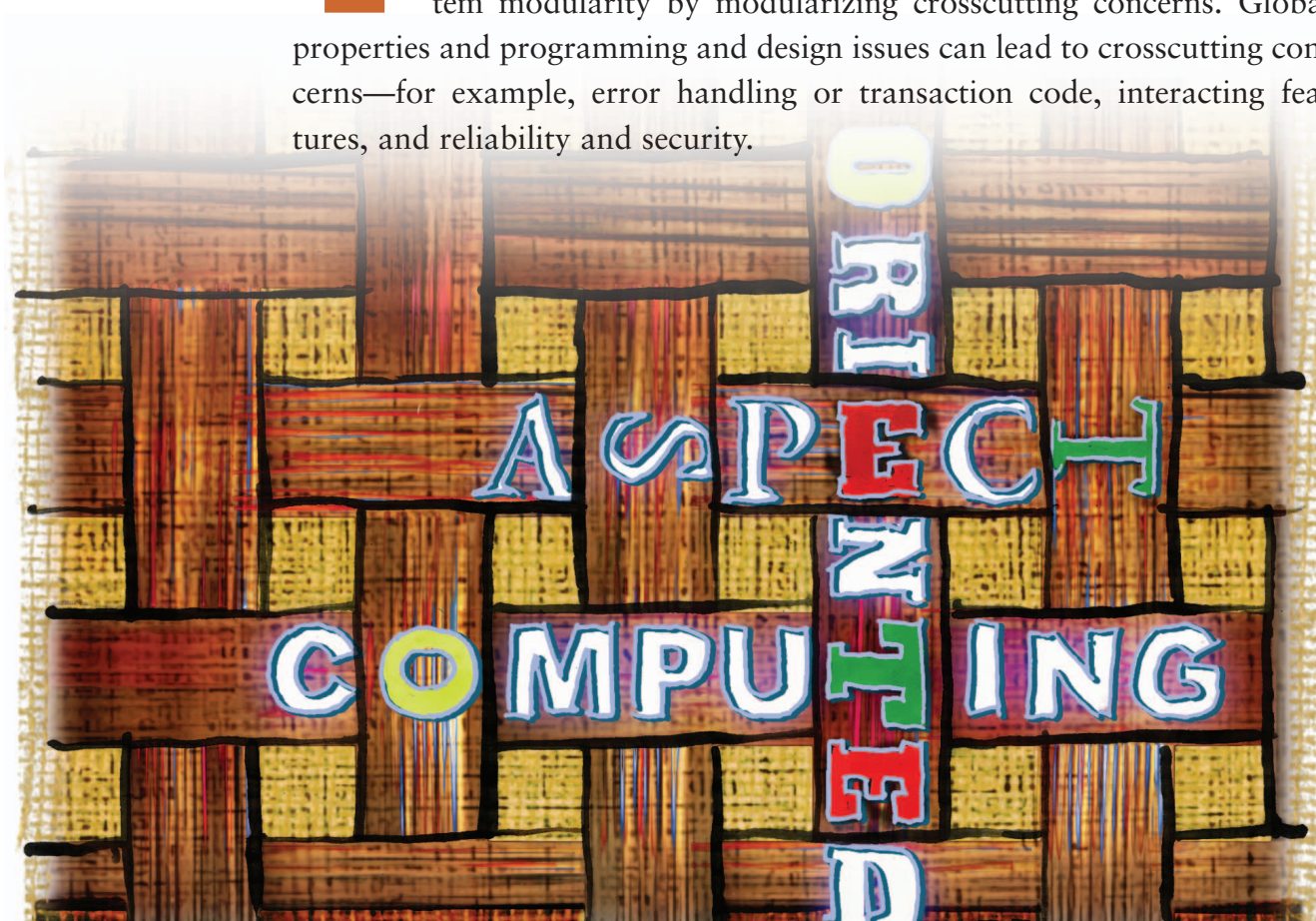
©2006 IEEE. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the IEEE.

# Aspect-Oriented Programming

**Gail Murphy**, *University of British Columbia*

**Christa Schwanninger**, *Siemens*

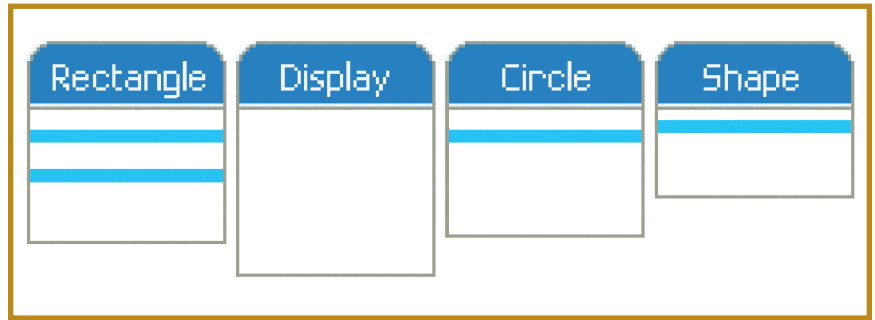
**Y**ou can measure a software system's value by its modularity.<sup>1</sup> The more modular the system, the easier it is to produce<sup>2</sup> and extend. Aspect-oriented programming<sup>3</sup> technologies aim to improve system modularity by modularizing crosscutting concerns. Global properties and programming and design issues can lead to crosscutting concerns—for example, error handling or transaction code, interacting features, and reliability and security.



Consider the improvements brought by object-oriented programming. Encapsulating data with its associated behavior in a class makes it easier for different developers to share the work needed to build the system. Developers need only communicate about and agree on the classes' interfaces. Let's take the canonical example of an object-oriented paint program designed according to the Observer pattern.<sup>4</sup> This program includes the elements painted by an abstract `Shape` class with concrete subclasses, such as `Rectangle` and `Circle`. A `Display` class makes shapes visible to the user. Developers can share the work needed to build the system by distributing work on the `Shape` hierarchy and on the `Display` class. The `Shape` hierarchy increases the system's value: you can extend it to represent UML class diagram notation to support software design editing and display, solving a different problem using the same framework.

Even when the developer chooses an object-oriented system's structure carefully, the implementation of some concerns still ends up being nonmodular, resulting in scattered and tangled code throughout the system. Common examples include logging, auditing, synchronization, and transactional support concerns. Our paint program offers a simple example: signaling an update to the display when shapes change will likely result in code scattered across classes in the `Shape` hierarchy as shown in figure 1.<sup>5</sup> AOP developers call update signaling a *crosscutting concern* because its structure inherently cuts across the classes of the `Shape` hierarchy. With the programming mechanisms available, the developer cannot simultaneously modularize both the representation of shapes and the representation of update signaling.

AOP improves this situation by supporting the modularization of crosscutting concerns.<sup>5</sup> Using an AOP language, a developer can define an `UpdateSignalling` aspect, a module that expresses code that crosscuts the system's structure and functionality. When needed, you can compose the aspect into the system. This modularization lets you independently apply the `UpdateSignalling` aspect or lets you implement different updating strategies. Then, you can configure the system with one of these strategies enabled or even omit the update functionality entirely. Just as an ordinary compiler or virtual machine implements the semantics of OO languages, an AOP compiler



**Figure 1. Crosscutting structure of display updating in the paint program. The rectangles represent the sizes of the classes in the paint program. The blue lines represent the code related to signalling a display update.**

or VM implements the AOP language's semantics using a technique known as *weaving*.

Figure 2 provides an example of an `UpdateSignalling` aspect as expressed in a popular aspect-oriented language called AspectJ.<sup>5</sup> Included in this simple example is a `Shape` class with two subclasses: `Rectangle` and `Circle`. When one of the `set` methods is

```
public class Shape {
    ...
}

public class Rectangle extends Shape {
    ...
    void setLeftUpperCorner( corner: Point ) { ... }
    void setRightBottomCorner( corner: Point ) { ... }
}

public class Circle extends Shape {
    ...
    void setRadius( length: float ) { ... }
}

public aspect UpdateSignalling {
    // This pointcut identifies all executions of any
    // method whose names start with set in Shape or
    // any of its subclasses
    pointcut displayStateChange():
        execution( void Shape+.set*(..) );

    // This advice refreshes the display every time
    // a displayStateChange join point matches
    after() returning: displayStateChange() {
        Display.refresh();
    }
}
```

**Figure 2. Modularizing the updating of a display in an AspectJ aspect.**

## Further Resources

Here, we offer some aspect-oriented programming resources.

### Languages and frameworks

Several aspect-oriented programming languages and frameworks are available for use in development, including

- **AspectJ** ([www.eclipse.org/aspectj](http://www.eclipse.org/aspectj)) is an aspect-oriented extension to Java.
- **Spring** ([www.springframework.org](http://www.springframework.org)) uses aspects to add declarative enterprise services to a J2EE framework and allows users to implement custom aspects.
- **JBossAOP** ([jboss.com/products/aop](http://jboss.com/products/aop)) provides an aspect framework for Java that includes prepackaged aspects for caching, asynchronous communication, transactions, and security.
- **AspectC++** ([www.aspectc.org](http://www.aspectc.org)) offers aspect-oriented support for C++.
- **Aspect#** ([www.castleproject.org/index.php/AspectSharp](http://www.castleproject.org/index.php/AspectSharp)) is an aspect-oriented framework for the Microsoft Common Language Infrastructure.

Many of these languages and frameworks include tool support to aid development and debugging.

### Books

Several books offer further information about the technology in general and the most mature aspect-oriented language, AspectJ:

- R. Laddad, *AspectJ in Action: Practical Aspect-Oriented Programming*, Manning Publications, 2003.
- A. Colyer et al., *Eclipse AspectJ: Aspect-Oriented Programming with AspectJ and the Eclipse AspectJ Development Tools*, Addison Wesley, 2005.
- J. Gradecki and N. Lesiecki, *Mastering AspectJ: Aspect-Oriented Programming in Java*, John Wiley & Sons, 2003.

For more on aspect-oriented ideas applied to requirements and design, see

- S. Clarke and E. Baniassad, *Aspect-Oriented Analysis and Design: The Theme Approach*, Addison-Wesley, 2005.
- I. Jacobson and P.-W. Ng, *Aspect-Oriented Software Development with Use Cases*, Addison-Wesley, 2005.

A collection of research papers about aspect-oriented technology appears in

- *Aspect-Oriented Software Development*, R. Filman, S. Clarke, and M. Aksit, eds., Addison-Wesley, 2004.

### Web sites and conferences

The Aspect-Oriented Software Association is a nonprofit organization that sponsors the annual Aspect-Oriented Software Development Conference ([www.aosd.net](http://www.aosd.net)). Many leading programming language and software engineering conferences have research sessions focused on aspect-oriented technology. Leading industry conferences such as JavaOne (<http://java.sun.com/javaone/sf>) and The ServerSide Symposiums ([www.theserverside.com/symposium/index.html](http://www.theserverside.com/symposium/index.html)) have had talks and panels on aspect-oriented technology.

called on a kind of `Shape` object, an update notification must be sent to the `Display` class to refresh. The `UpdateSignalling` aspect declares the `displayStateChange` *pointcut*, which identifies points of interest in the execution called *join points* at which to compose behavior. In this case, the join points are each execution of any `set` method in a `Shape` subclass. The *advice* in the aspect indicates that the display should refresh after any join point identified by the `displayStateChange` pointcut. In this program, all the code related to display updating is now modularized in the aspect.

Display updating is a simple example of a crosscutting concern. In this special issue, Nicholas Lesiecki's experience report demonstrates the technology's power by describing the use of several aspects in the development of a Web-based search and e-commerce interface for advertising data. The aspects served various purposes, including aiding debugging, implementing a feature, and helping to enhance the functionality of a framework used for the development. The article shows the use of AspectJ's dynamic and static join point models: the dynamic model expresses computations to be composed into the system's runtime; the static model lets a developer modify the program's static structure. In addition to describing the aspects used, the article discusses tool support and adoption issues to consider when moving to aspect-oriented technology.

The version of AspectJ described in Lesiecki's article uses a *static weaver*, meaning that it weaves aspects into the system during compilation. Context-aware environments or systems that must be highly available might benefit from *dynamic weaving*, or weaving aspects into a system as it's running. Marc Ségura-Devillechaise and his colleagues describe in their article how you can use dynamic weaving to deploy a security patch to a running Web cache. They also describe how they used their Arachne system, which provides aspect-oriented support for systems written in C, to maintain simplicity in the Web cache code while adding performance improvements and support for new protocols. The Arachne language includes support for identifying join points based on sequences of function calls. This demonstrates an important direction for aspect-oriented technology—the development of semantically meaningful and stable pointcut languages. While Lesiecki shows current state of the art, Ségura-Devillechaise's arti-

cle allows a glance at what we expect to work with in the medium-term future.

Bart Verheecke and his colleagues' article also describes the use of a dynamic aspect composition mechanism, but in the completely different domain of service-oriented architectures. A Web services management layer implemented with aspect technology allows the modularization of concerns, such as service composition, redirection, user authentication, and logging, which are otherwise scattered and tangled across client applications of Web services. They describe how the JasCo aspect language, which uses an extended version of a Java bean to hold crosscutting behavior, and a separate connector component allowed them to simplify and improve the development of a video-on-demand system.

Many AOP languages support selecting join points and modifying behavior at those join points from the aspect. Using this approach, the developer of code that will later be advised by the aspect can remain oblivious of crosscutting concerns—a useful property, but also a potential source for introducing unanticipated or even misbehavior. Whether or not obliviousness is a fundamental AOP property and how this property affects reasoning about the code in a modular way is an ongoing discussion topic within the community. William G. Griswold and his colleagues describe how introducing interfaces to describe crosscutting concerns can help expose appropriate join points to ensure that both the aspect code and the code to which the aspect applies can evolve gracefully together. They analyze how their approach aids evolution by considering changes to a version of the canonical figure editor example.

Aspect-oriented mechanisms were first developed for source code. The difficulties associated with nonmodularized crosscutting structure are not limited to implementation but also occur during requirements and design activities. It's anticipated that support for modularizing crosscutting concerns at these other stages of development can provide similar benefits to those seen at implementation and can enable better traceability across development stages and more flexible product configuration. The article by Elisa Baniassad and her colleagues describes the authors' ongoing work to bring aspect-oriented ideas to these earlier stages of the software development life cycle.

Similar to object-oriented software development 20 years ago, aspect-oriented technology

## About the Authors




**Gail Murphy** is an associate professor in the Department of Computer Science at the University of British Columbia. Her research interests include software evolution and information structure. She received her PhD in computer science and engineering from the University of Washington. She is a member of the ACM and the IEEE Computer Society. Contact her at 2366 Main Mall, Vancouver BC, Canada V6T 1Z4; [murphy@cs.ubc.ca](mailto:murphy@cs.ubc.ca).

**Christa Schwanninger** is a senior research scientist at Siemens AG's Corporate Technology division. Her research interests include software architecture, distributed object computing, patterns, software system families, and aspect-oriented software development. She received her MS in computer science from Johannes Kepler University. She is a member of the ACM. Contact her at Siemens AG, Corporate Technology, Software and Eng. Division, Otto-Hahn-Ring 6, 81739 Munich, Germany; [christa.schwanninger@siemens.com](mailto:christa.schwanninger@siemens.com).



requires us to change how we think about, design, and implement systems. In the early days of object technology, experts warned that these changes were likely too big of a leap for the average software developer. A similar discussion is ongoing among experts and vendors of aspect-oriented technology today. Learn more about each side's arguments in the Point/Counterpoint discussion on whether AOP is a technology for everyone.

**M**ore than ever, software developers must produce complex, configurable systems that must function in an ever-changing environment. In the next few years, we believe aspect-oriented technologies will play an increasing role in several additional areas: embedded systems, supporting customizability while meeting these systems' tight engineering constraints; product families, because crosscutting features can be implemented and combined according to customer needs; and ambient systems, as dynamic weaving can support runtime adaptability. 

## References

1. C.Y. Baldwin and K.B. Clarke, *Design Rules: The Power of Modularity*, MIT Press, 2000.
2. D. Parnas, "On the Criteria to Be Used in Decomposing Systems into Modules," *Comm. ACM*, Dec. 1972, pp. 1053–1058.
3. G. Kiczales et al., "Aspect-Oriented Programming," *Proc. of European Conf. Object-Oriented Programming (ECOOP 97)*, LNCS 1242, Springer, 1997, pp. 220–242.
4. E. Gamma et al., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994.
5. G. Kiczales et al., "An Overview of AspectJ," *Proc. 15th European Conf. Object-Oriented Programming (ECOOP 01)*, LNCS 2072, Springer, 2001, pp. 327–353.