An operation in an object-oriented program often involves several different collaborating classes. There are usually two ways to implement such an operation: either put the whole operation into one method on one of the classes, or divide the operation into methods on each of the classes involved. The drawback of the former is that too much information about the structure of the classes (is-a and has-a relationships) needs to be tangled into each such method, making it difficult to adapt to changes in the class structure. However, the latter scatters the operation across multiple classes, making it difficult to adapt when the operation changes.

To resolve this conflict, the Demeter project has developed the notion of an adaptive method (known in previous work as a propagation pattern [3]). An adaptive method encapsulates the behavior of
an operation into one place, thus avoiding the scattering problem, but also abstracts over the class structure, thus avoiding the tangling problem as well. To do this, the behavior is expressed as a high-level description of how to reach the participants of the computation (called a traversal strategy [4]), plus what to do when each participant has been reached (called an adaptive visitor). Both the traversal strategy and the adaptive visitor mention only a minimal set of classes that participate in the operation; the information about the connections between these classes is abstracted out. In Java, reflection provides us with an elegant way to achieve this abstraction: the class structure can be accessed directly at runtime. The DJ—for Demeter/Java—library is a Java package that supports this style of programming by providing tools to interpret a traversal strategy and an adaptive visitor in the context of the underlying class structure.

A Simple Example
The figure appearing here shows a small adaptive method written in Java using the DJ library. The purpose of the code is to sum the values of all the Salary objects reachable by has-a relationships from a Company object. It works as follows: the static variable cg is a class graph object that represents the program’s class structure. A class graph is a simple form of a UML class diagram that describes is-a and has-a relationships between classes [1]. The class structure is computed in ClassGraph’s constructor using reflection. It is used by the traverse method as a context in which to interpret traversal strategies. The traverse method starts in a given object (this in the example here) and traverses specified paths (“from Company to Salary”) executing any applicable visitor methods along the way (in this situation upon starting, to initialize the sum; upon reaching each Salary object, to add the value to the sum; and upon finishing, to build the return value).

The separation of traversal strategy (where to go), adaptive visitor (what to do), and class graph (context to evaluate in) allows the method to be reused unchanged in a set of programs. Details such as the number of classes between Company and Salary are unimportant: it could be a company with a big organizational structure (divisions, departments, work groups, and so forth) with many classes between or a company with a small organizational structure (only work groups) with few classes between.

Connection to Aspects and AspectJ
AspectJ [2] makes the following key definitions: Join points are principled points in the execution of the program. Pointcuts are a means of referring to collections of join points and certain values at those join points. Advice is a method-like construct that can be attached to pointcuts; and aspects are modular units of crosscutting implementation, comprised of pointcuts, advice, and ordinary Java member declarations.

The adaptive method sumSalaries is an aspect in the terminology of AspectJ: it is a modular implementation of a crosscutting concern. It contains a definition of a collection of join points and also contains a definition of advice for those join points. The join points are defined by a traversal strategy and the advice is defined by an adaptive visitor.

The join points are defined in an unusual way: instead of referring to points in the execution of a program given to us, we first define a traversal in terms of a class graph given to us and then we consider the join points defined by this traversal. In other words, we use a version of AOP where the pointcut definitions indirectly define a collection of join points based on information in the class graph.

The advice is also defined in an unusual way: in the Java method sumSalaries(...) different join points receive different advice while in AspectJ all join points in a pointcut get the same advice. With DJ, some points in a pointcut get no advice at all if they are not mentioned in the adaptive visitor.

Conclusion
We have highlighted how a family of crosscutting behavioral concerns can be implemented in a modularized way in pure Java using the DJ library. The modular units—known as adaptive methods—cleanly encapsulate behavior that would normally be tangled with information about the class structure or scattered across multiple classes. The behavior of the methods involves groups of participating objects, all
of which can be reached from given source objects. We have also briefly explained the core concepts of adaptive methods in the aspectual terminology of AspectJ here.

The DJ library permits Java programmers to follow the Law of Demeter in an optimal way and to experiment with aspect-oriented ideas without having to learn an extension to Java or to preprocess source files (this is an issue with off-the-shelf integrated development environments). This ease of use comes at a price: only a limited class of behavioral concerns can be expressed as adaptive methods. However, in our experience this class of behavior covers a large portion of real-world programming tasks, rendering the restriction not as onerous as it initially might seem. Lastly, the current naive implementation relies heavily on reflection, incurring noticeable slowdowns. A more sophisticated implementation is in progress.

The DJ library is available in source and bytecode form at www.ccs.neu.edu/research/demeter/DJ. A more detailed discussion of DJ, showing how to use adaptive programming methods to process XML schemas, is available at www.ccs.neu.edu/research/demeter/papers/DJ-reflection/.

REFERENCES

KARL LIEBERHERR (lieber@ccs.neu.edu) is Professor of Computer Science at Northeastern University in Boston, MA.

DOUG ORLEANS (dougo@ccs.neu.edu) is a Ph.D. student at Northeastern University in Boston, MA.

JOHAN OVLINGER (johan@ccs.neu.edu) is a Ph.D. student at Northeastern University in Boston and a researcher with SKYVA International in Medford, MA.

Research supported by Defense Advanced Research Projects Agency under agreement F33615-00-C-1694.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.