

Can Aspect-Oriented Programming Lead to More Reliable Software?

John Viega and Jeffrey Voas

Aspect-oriented programming is a novel topic in the software engineering and languages communities. AOP appears to have the potential to significantly improve the reliability of programs, particularly by modularizing error-handling policies and allowing for easier maintenance and better reuse. In this article we introduce AspectJ, the first

About the only time programmers apply abstraction to exception handlers is when they employ an exception handler to address multiple types of errors. Unfortunately, handling exceptions generically can result in unreliable and ineffective code. For example, programmers often handle otherwise unanticipated exceptions with catch-all handlers. The problem with this solution is that the programmer often has no idea how to recover gracefully from certain situations. Instead, the programmer creates exceptions that log the problem and then exit instead of recovering dynamically.

We are not arguing against generic exceptions. For example, it might be nice to handle all remote method calls that fail using `RemoteException` by retrying the operation three times before aborting. Unfortunately, such functionality would likely require that the programmer touch many different pieces of code.

Ideally, it is better to combine generic exceptions with specialized functionality (programmed on a case-by-case basis for each exception). To do so using traditional exception-writing techniques, programmers must manually change every exception, yielding much duplicate code. Another problem is maintenance of exception-handling code. For example, the intended global nature of exception logging might not be intuitive to those who will someday maintain the code. Future maintainers might add new exception handlers and forget to add appropriate logging functionality.

Crosscutting Concerns

The key issue that makes exception handling difficult from a management per-

aspect-oriented programming language, and demonstrate how you can use it to construct more reliable software.

Modularity in Exception Handling

Traditional exception-handling code suffers from many problems, the most prominent being that such code tends not to be very modular. The typical reason for this problem is that programmers treat exception handling as an ad hoc process; programmers often sprinkle handlers carelessly throughout the code and write them on a last-minute, as-needed basis, instead of designing them in from the start. Programmers rarely reuse exception-handling code or apply any abstraction to handlers whatsoever.



spective is that it is difficult to modularize exception handling using standard object-oriented abstraction techniques. Although the programmer can move exception-handling routines to utility classes, those utility classes will need to be called from many places in the code. Note that the problem here is only with exception handlers, not with exceptions themselves; exceptions are generally quite easy to inherit and reuse.

Exception handling is difficult to modularize because it is a program design feature that tends to ignore traditional class boundaries. Such features are said to be *crosscutting concerns* because they tend to cut across the traditional boundaries of abstraction (including classes and modules). Such concerns generally affect multiple objects in some way. Crosscutting concerns are harder to maintain than concerns that are easily encapsulated into objects, because code related to the concern is spread throughout the code base, making changes difficult and error-prone.

Separation of Concerns and Aspects

A fairly new movement in the programming languages and software engineering communities has called for languages that promote better *separation of concerns*. The goals are to reduce complexity and improve the reusability of software by providing appropriate abstraction mechanisms. These abstraction mechanisms provide a solution for crosscutting concerns by allowing them to be defined in a single location and applied uniformly throughout the program.

The most prominent approach to crosscutting concerns is AOP. It supports a traditional object-oriented programming model and offers several powerful additions. In AOP, traditional object-oriented programs are written in a base programming language. However, the programmer can define aspects that map to emergent crosscutting concerns in a traditional program. These aspects are modular units that avoid crosscutting.

Conceptually, aspects are separate from objects. Aspects can observe other objects and react to their behavior. In an AOP language, programmers can write code that will get called before any method call, and might just as easily write code that will get called before any method call in any public member of any public class in a set of packages.

In a way, aspects are the opposite of inherited classes. With inheritance, classes choose what functionality they wish to subsume from other objects. Aspects, on the other hand, get to choose what functionality other objects will subsume.

Exception Handling with AspectJ

The only AOP language currently seeing significant use is AspectJ, from Xerox Parc (www.aspectj.org). It is an extension of Java that adds four language constructs to support the aspect-oriented paradigm: aspects, join points, advice, and code introduction. We will introduce each construct in the context of improved exception handling.

Aspects

Much like a class, aspects are typed entities that contain functionality. However, aspects are unlike classes in that they are meant to capture crosscutting concerns to be injected into other types. Also, aspects can contain new programming elements that classes cannot.

A fairly new movement in the programming languages and software engineering communities has called for languages that promote better separation of concerns.

In the context of exception handling, programmers might wish to have a single aspect that encapsulates a system-wide exception-handling strategy. Or they might desire a single aspect for each category of exception that they wish to handle uniquely. They can even create an aspect that supplements whatever exception handling is otherwise done, forcing each thrown exception to be logged.

Join Points

Aspects cannot crosscut objects arbitrarily. AspectJ allows crosscutting only at well-defined points, such as during object construction, method entry, or (in the future) member variable access points). Thus, aspects can only introduce their supplemental functionality at these points, called *join points*. The specification for naming a join point is called a *pointcut declaration*. At present, AspectJ does not allow the user to select places where exceptions are thrown as a join point. However, the programmer can specify method entry as a join point and add code to deal with exceptions that a method raises but does not catch. It is also possible to specify the entry point of exception handlers as join points, thus allowing programmers to address any exception that is caught and ignore those that are not.

Advice

After the programmer has defined the points at which an aspect adds behavior, the behavior to be added must be defined. Such behavior is called *advice*. The contents of advice can contain anything you can put in an arbitrary Java method. The programmer can add advice before a join point runs or after it finishes running, or can force advice to run instead of the join point. If the programmer wants an “exception logging” aspect that logs exceptions that get propagated past a method exit, he or she could add *after advice* to method executions. In the behavior code, the programmer would try to catch an exception; if he or she caught one, it would be logged and reraised. Otherwise, nothing would be done.

```

aspect ExceptionPrinter {
    pointcut allMethods(): executions(* *(..));
    static after() throwing (Exception e) : allMethods() {
        System.out.println("Uncaught exception: " + e);
    }
}

```

Figure 1. An AspectJ aspect for reporting unhandled exceptions in a package.

AspectJ allows for more granularity with after advice; the programmer can explicitly write behavior that runs only when an exception is thrown or when a join point executes successfully. In these cases, both the thrown exception and the return value (if any) are easily accessible. Such functionality makes exception-handling aspects easier to write.

Code Introduction

With code introduction, programmers can add variables and methods into arbitrary types by using aspects. In the context of error handling, such functionality might be useful for keeping track of class-specific error rates.


An Example Aspect

The code in Figure 1 is a simple example of an aspect that prints all exceptions not handled by any given method in the current package. We could use this code during testing to see whether unexpected errors propagate out of a package, thereby indicating a bug in the package. We might wish to keep this code in the final version, logging the error to disc instead of printing it. And if users consent to having such logs periodically mailed back to the software maintainer (so that bugs in the software that went uncovered at testing time would get reported), maintainers would not need to rely on end users to notice and report problems.

In this example, we specify a pointcut named `allMethods`, which cuts all executions of any function in the current package. The `executions` keyword specifies that we wish to cut

method executions, and the argument is a wildcard expression declaring that we wish to cut all methods with any signature. The first asterisk indicates that matched methods might return anything, the second indicates that any method name should match. The double dot indicates that matched methods might have any number of arguments.

Our pointcut has some advice defined for it when an exception is thrown. Our advice could be specified for multiple pointcuts, which is why the pointcut's name is specified after a colon (this name could be a list of names instead).

Aspect-oriented programming definitely shows promise as a technique for building more reliable software. We hope that programmers will be able to use the technology to encapsulate all sorts of robustness features that were previously difficult to abstract. For example, we see security as an area where aspect-oriented programming could greatly ease the burden for developers. 

John Viega is a senior research associate and senior consultant at Cigital (formerly Reliable Software Technologies). He is currently the principal investigator on a DARPA grant on adding security features to programming languages using the aspect-oriented programming paradigm. He is also writing two books, *Building Secure Software* (Addison-Wesley, 2001) with Gary McGraw and *Java Enterprise Development* (O'Reilly, 2001) with George Reese. Contact him at Cigital, 21351 Ridgetop Circle, Suite 400, Dulles, VA 20166; viega@cigital.com.

Jeffrey Voas is cofounder and chief scientist of Cigital (formerly Reliable Software Technologies). He has a BSE in computer engineering from Tulane and an MS and a PhD in computer science from the College of William and Mary. He is the coauthor (with Gary McGraw) of *Software Fault Injection: Inoculating Programs against Errors* (John Wiley & Sons, New York, 1997). He serves on the editorial board for *IEEE Software*. Contact him at 21351 Ridgetop Circle, Suite 400, Dulles, VA 20166; voas@cigital.com.

IEEE Software

How to Reach Us

Writers

For detailed information on submitting articles, write for our Editorial Guidelines (software@computer.org), or access computer.org/software/author.htm.

Letters to the Editor

Send letters to

Letters Editor
IEEE Software
10662 Los Vaqueros Circle
Los Alamitos, CA 90720
cbaltes@computer.org

Please provide an e-mail address or daytime phone number with your letter.

On the Web

Access computer.org/software for information about *IEEE Software*.

Subscription Change of Address

Send change-of-address requests for magazine subscriptions to address.change@ieee.org. Be sure to specify *IEEE Software*.

Membership Change of Address

Send change-of-address requests for the membership directory to directory.updates@computer.org.

Missing or Damaged Copies

If you are missing an issue or you received a damaged copy, contact membership@computer.org.

Reprints of Articles

For price information or to order reprints, send e-mail to software@computer.org or fax +1 714 821 4010.

Reprint Permission

To obtain permission to reprint an article, contact William Hagen, IEEE Copyrights and Trademarks Manager, at whagen@ieee.org.