Join Point Encapsulation

David Larochelle
drl7x@cs.virginia.edu

Karl Scheidt[*]
Department of Computer Science
University of Virginia
kgs3c@cs.virginia.edu

Kevin Sullivan
sullivan@cs.virginia.edu

ABSTRACT

At the heart of aspect-oriented programming is the exposure of certain phenomena in the execution of one set of program elements to behavioral modifications specified by other elements. The phenomena are *join points*. The modifying elements are *aspects.* The problem that we address is that current aspect-oriented languages do not provide adequate means to control the exposure of join points for behavioral modification by aspects. Rather, these languages define certain classes of join points (e.g., method calls) and expose all instances thereof. In a nutshell, then, current languages lack *join point encapsulation mechanisms.* We map a solution space and describe one proof-of-concept design that we have implemented for AspectJ. A key feature of our design is that it is, itself, aspect-oriented: We use AspectJ's pointcut language to identity cross-cutting sets of join points to be encapsulated. The potential benefits of such a modularity-supporting mechanism include improved ease of reasoning about program behavior and the ability to assure the absence of side-effects through enforced non-interference by aspect code, and improved ease of design evolution.

1. INTRODUCTION

Aspect-oriented programming (AOP) languages enable the modular representation of cross-cutting concerns: behaviors whose constituent parts are scattered among and modify the behaviors of other modular constructs. The key to aspect languages is making certain phenomena in the execution of one modular construct accessible to modifications specified by others. The phenomena exposed are *join points.* Aspect code can modify the behavior of base code at join points [4].

[*]All authors are from the Department of Computer Science, University of Virginia, 151 Engineer's Way, P.O. Box 400470, Charlottesville, VA 22904-4740 USA. Tel. +1 434 982 2200. FAX: +1 434 982 2214.

Aspect languages define certain classes of join points as accessible for modification. AspectJ [3] join points, for example, include object initialization, method call and execution, and data field manipulation. All instances of such join points are accessible and together form a universe of join points for a system. An aspect language also provides a sublanguage used to concisely select subsets of join points called *pointcuts.* Such a *pointcut language* is at the heart of the ability of an aspect language to modularize crosscutting concerns. It enables designers to specify behavioral modifications that affect an entire system.

The problem we address is that currently aspect languages do not allow designers to constrain the accessibility of join points to aspects. There is no effective way to encapsulate join points to make them not subject to aspect-imposed modification. Thus, using the terminology of Meyer [6], all modules are necessarily *open,* and one cannot *close* or partially close them. In a sense, every module's interface is extended by a large, implicit set of events, and at any such point, behavior observation and modification are possible.

We believe that it is undesirable to make all join points necessarily subject to behavioral observation and modification by aspects. To do so unnecessarily and adversely conflates a mechanism (join points) and a policy (that they're all always visible). Our solution is to decouple them. In particular, we advocate maintaining or perhaps even broadening the set of join points in a given aspect language but then enabling the designer to impose a join-point-visibility policy as most appropriate for the given system. To do so, we provide a second kind of mechanism that we call *join point encapsulation.*

The contribution of this paper is the case for providing join point encapsulation mechanisms, a mapping of the design space for mechanisms of this kind, and a usable, proof-of-concept design and implementation for the AspectJ language. Our design and implementation leverages the design, implementation, and philosophy of AspectJ. In particular, we use the existing pointcut language and advice constructs, introducing only a new kind of advice: *join point encapsulation advice,* or *restriction advice.* Restriction advice serves to encapsulate the join points selected by a pointcut against modification by other aspects thus enabling the modular representation of the encapsulation of crosscutting sets of join points. That is, we provide an aspect-oriented join point encapsulation mechanism.

The rest of this paper is organized as follows. Section 2 presents a motivating example of the problem. Section 3 maps a design space of possible mechanisms to solve the problem. Section 4 introduces restriction advice as one specific join point encapsulation mechanism. Section 5 discusses our implementation of such a mechanism for AspectJ. Section 6 presents some discussion of our work and related work. Section 7 summarizes and concludes with a discussion of possible future directions.

2. MOTIVATING EXAMPLE

One common pattern in object-oriented programming is to have a publicly accessible method that is implemented by calling a separate private method which may have a different signature. This technique allows the public interface to be maintained while changing the private implementation, and allows multiple public interfaces (with different default arguments, perhaps) to use the same private implementation:

```
public class SomeClass {
    public void method(String a) {
        method_impl(a,null);
    }
    public void method(String a,
                       String b) {
        method_impl(a,b);
    }
    private void method_impl(String a,
                             String b) {
        // private implementation
    }
}
```

Here, `method` is implemented by calls to `method_impl` with default arguments supplied. This technique is useful because it allows the private implementation to be changed without affecting code that uses only the public interface. This allows a lot of flexibility: in a future version of the class, `method_impl` could be removed entirely, and the implementation moved to the two versions of `method`.

However, in AspectJ, even private method join points are exposed to modification by aspects. Even though `method_impl` is not a part of a public interface, aspects can still advise its call join point. This join point is thus, in essence, part of a broad, implicit interface exposed by the class to the pointcut language and thus to aspect modules. The visibility of this join point in turn creates the potential for dependencies on the private method to arise from external aspect modules.

More generally, visibility of join points has the potential to be counterproductive in several dimensions, depending on the particular circumstances of a system. First, join points can expose otherwise encapsulated implementation details (e.g., private methods). Doing this can undermine the designer's intent to use an information hiding design strategy, and the use of abstraction to limit what must be known to understand an object's behavior. Second, the accessibility of join points creates opportunities for aspects to change program behaviors in ways that the designer might not want. For example, a designer might want to hide the join point interface of an object altogether to ensure that it is possible to reason about the complete behavior of the object based only on local information (e.g., in the context of a safety-critical system, where what a module does not do can be as important as what it does do). If it is not possible to prevent aspects from modifying object behavior, then it is impossible to provide such assurances. Third, a designer might want to provide assurances that an object will not violate specified type or system invariants—assurances that can hinge on promises that an object's actions can have no unintended side-effects, as might be created by aspects.

A means for limiting what join points are exposed to modification by advice could solve these problems. For example, the join points for `method_impl` could be marked as restricted and unavailable to modification by outside advice. This restriction would preclude aspects from using parts of the implicit interface that were either meant to be impermanent or abstracted to improve comprehensibility.

A different solution to this problem would be to design an aspect language that did not expose the call join points of private methods. This solution is wholly unsatisfactory. It works precisely by undoing what aspect languages do, and it goes against a significant trend, to expose more phenomena of program execution so as to enable new kinds of modularity. Rather, we see again the need to decouple the join point mechanism from currently hard-wired policies concerning join point visibility to aspect code.

3. A DESIGN SPACE

In this section we discuss some of the important dimensions of the design space for mechanisms for control of join point visibility.

3.1 Who says what's visible?

There are several constructs in a typical aspect-oriented language that could be modified to express join point visibility restrictions. One that we considered was to have classes encapsulate their own join points. Another approach would be to have separate constructs define join point visibility.

In the end, we decided to use aspects as separate constructs for expressing join point visibility policies. This approach actually generalizes the class-based approach provided that these restriction aspects can be declared within classes. Moreover, choosing aspects as the mechanism allows crosscutting visibility policies for an entire module or system to be expressed in modular form.

3.2 How expressive are encapsulation policies?

The visibility of a given join point in a system can in principle be constrainted in many ways. Each requires a more or less expressive language for stating encapsulation policies. We discuss three. The first is to have restrictions based on the scope from which a given aspect attempts to advise the join point. Such a mechanism could, for example, allow a given module to be implemented internally in an aspect-oriented way while preventing external modules from advising its methods. The early prototype mechanism that we present in more detail in the next section makes all restrictions global.

Second, visibility restrictions could be based on the kind of advice that an aspect attempts to bind to a join point. This would allow one to specify, for example, a join point is not accessible to *around* advice, but is accessible to *before* and *after* advice. Such a mechanism could be useful if a module needs to guarantee that a certain action will surely be performed, without foreclosing the possibility of unanticipated modifications by other kinds of advice. The mechanism that we present does not currently support this level of discrimination.

Third, visibility restrictions could be made to hinge on the manner in which an aspect names a join point. It might make sense, for example, to disallow advice that matches a join point using the name of a private method, but to allow advice that attaches to the same join point using a more generic specifier, such as a wildcard or a *cflowbelow* pointcut. This level of selectivity would allow unanticipated modification without allowing the potentially undesired coupling that can occur if the names of private methods in one module are referenced in another.

The simple mechanism that we describe next does not support this kind of approach. Our mechanism has the virtue of exploiting the existing pointcut language to select join points for encapsulation. It is easy to understand and use and easy to implement. A richer mechanism as discussed here would require a meta-language for expressing properties of pointcut expressions. We intend to explore this issue in the future.

3.3 How to handle violations?

An attempt by an aspect to advise a join point encapsulated against that advice could result in a variety of behaviors. For example, the compiler could decline to advise the join point silently, issue a warning, or issue a compile-time error. The designer could also be given a choice of which of these policies to use for a particular case.

Which policy makes the most sense generally depends on the situation. If an aspect that performs logging is not able to attach to all of its desired join points, this may be acceptable and we may wish to just ignore such conflicts. However, if an aspect enforcing a locking protocol cannot attach to all of its desired join points, this could cause the program to fail, and so we may want this violation to result in a compile-time error.

Our prototype design simply declines to advise and issues warnings each time an aspect attempts to advise an encapsulated join point.

4. OUR DESIGN: RESTRICTION ADVICE

In this section, we describe the particular point in this design space that we selected as a starting point to explore the issue of join point encapsulation. In a nutshell, we introduce a new kind of advice: advice not meant to modify program behavior at a join point, but to restrict the visibility of the join point to aspects. We call this *join point encapsulation advice,* or just *restriction advice,* to identify which join points are encapsulated against aspects. This solution takes advantage of the power of AspectJ's existing pointcut language to specify sets of join points to be encapsulated.

Restriction advice specifies which join points are not available for modification by aspects. The default is to expose all join points that are not specifically restricted. This allows existing AspectJ programs to be used without modification. More importantly, it allows aspects to continue to modify classes in ways not anticipated by the original class designers, while allowing class designers to specify that certain parts of the class are private to the implementation and should not be exposed.

Restriction advice looks like any other advice. It uses the pointcut language to specify what join points it applies to:

```
pointcut private_impl():
  call(private *
       SomeClass.*_impl(..));

restrict(): private_impl();
```

The pointcut `private_impl` identifies those join points within `SomeClass` that are restricted, namely, the private `*_impl` methods that are not intended to be permanent parts of the class's interface. The `restrict` advice marks the specified join points as unavailable to modification by aspects.

When the weaver applies aspects to a class for which restrictions are given, it will attach advice only to join points that match the aspect's pointcut and do not match any of the restricted pointcuts.

For example, consider a piece of advice that attempts to modify methods in `SomeClass`, above:

```
aspect SomeAspect {
    pointcut target_methods():
        call(* SomeClass.*(..));

    before(): target_methods() {
        do_something();
    }
}
```

Because there is restriction advice matching `method_impl`, the weaver does not allow the `before` advice to attach to the restricted join points, but it may still apply to any other exposed join points in the class. When the weaver processes this aspect, the `target_methods()` pointcut effectively becomes the following:

```
pointcut target_methods():
  call(* SomeClass.*(..)) &&
  ! call(private *
        SomeClass.*_impl(..));
```

Restrict advice, like other kinds of advice, must appear in an aspect. There are two main ways we anticipate restrict advice to be used: as an inner aspect describing the restrictions that apply to the class it is contained within, or as an external, top-level aspect describing what restrictions apply to the system or package as a whole.

By using an inner aspect for restrictions, each class can specify what restrictions apply to it. This allows a class like `SomeClass` to specify what methods in it are restricted, and such restrictions could even be inherited if written correctly. This is probably the most straightforward use of restrictions, and makes the most sense when inner aspects restrict only methods in the class they are contained within.

The other alternative, placing restriction advice in top-level aspects that apply to the entire system, has interesting possibilities for modularity. This allows all of the join point visibility rules for a system to be encoded in a single place, making it possible to apply default policies to an entire project. For example, a package might set a default policy forbidding advice attachment to private methods, or to any methods that access a particular private data structure which might be removed in a future version. Because the restrictions are specified in a separate module, it makes it easy to substitute different restriction policies without changing the individual classes to which they apply.

5. IMPLEMENTATION

We implemented our restriction language in AspectJ version 1.0.3. We modified the `ajc` compiler to recognize the keyword `restrict` and parse our restrictions. Parsing occurs as it would for a `before` advice statement, except that no code body is required. We modified the advice planning pass of the compiler so that when the compiler checks each join point for matching advice, it also checks for matching restrictions. If the join point is restricted, the compiler does not create a plan for weaving the advice, as it normally would, but issues a warning to the user instead.

Consider a simple `main` for `SomeClass`:

```
public static void main(String[] args) {
    SomeClass sc = new SomeClass();
    sc.method("Hello");
}
```

In addition, consider the following aspect to trace all method calls of `SomeClass`:

```
aspect Tracer {
    pointcut allMethodCalls():
        call(* SomeClass.*(..));

    before(): allMethodCalls() {
        System.out.println("In method "+
          thisJoinPoint.getSignature());
    }
}
```

When compiled and run, this program produces the following output:

```
In method void SomeClass.main(String[])
In method void SomeClass.method(String)
In method void SomeClass.method_impl(
    String, String)
```

Now consider adding the following inner aspect to `SomeClass` (which makes use of an anonymous pointcut):

```
static aspect restrictImpl {
   restrict():
       call (* SomeClass.*_impl(..))
}
```

During compilation, we get the following warning:

```
Warning:  Restricting attachment of
 advice Planner(before()) to code at
 method-call(private void
 SomeClass.method_impl(String, String))
```

The output that is produced when the program is run is as follows:

```
In method void SomeClass.main(String[])
In method void SomeClass.method(String)
```

This shows that the compiler correctly determines that the join point is restricted, and does not attach advice to it, as it otherwise would. All other join points in the program are handled normally. The compiler issues a warning to indicate that there is a conflict between a piece of advice and a restriction, because this may indicate a programming error.

Our prototype handles restrictions for join points whose elements are known statically. It does not yet handle the dynamic case. If a pointcut includes the `cflowbelow` keyword, our mechanism cannot always determine if a given join point is included. That decision requires information that is not known until run-time. A correct implementation would exploit run-time hooks (as are already used in the AspectJ implementation) at each dynamic join point to determine if the restriction applies before calling any associated advice. At compile time it might be possible to warn of possible dynamic restriction violations. It would be relatively easy to either decline such advice silently or to raise runtime exceptions to flag their occurance.

6. RELATED WORK

6.1 Visibility Restrictions in AspectJ

Our proof-of-concept design and implementation extends the AspectJ [3] language. AspectJ provides some features for controlling access to private members of classes by aspects. The code body of an aspect's advice must respect normal Java visibility rules, and thus may not directly call private methods or access private data members. However, as we have demonstrated here, aspects are allowed to attach advice to private member methods of a class, and this provides an undesirable implicit extension of the interface of that class.

6.2 The AspectJ declare error Construct

The *declare error* construct of AspectJ has been used to enable a compiler to enforce design rules [9]. Many design rules can be described using AspectJ's powerful pointcut language to identify situations in which the design rules are violated. The declare error construct is then used to issue a fatal error if a violation exists. For example, the following code would enforce the rule that no kernel procedure may access methods outside the kernel module:

```
pointcut bad_method_call():
   call(* (!Kernel+).*(..))
   && within(Kernel+);
declare error:  bad_method_call():
   "method call from kernel class
     to non-kernel class";
```

Although the declare error statement, acting on the extensive AspectJ pointcut language, is a powerful construct and has been shown to be effective as a means of enforcing design rules, it is not powerful enough to provide the kind of restrictions that would resolve the problem described in this paper. In order for the declare error statement to be able to restrict attachment of advice to join points, it would have to be possible to describe, using the pointcut language itself, either the attachment of advice to a join point, or the matching of a join point to a declared pointcut, neither of which is possible with the existing AspectJ pointcut language.

6.3 Other Related Systems

A well known project related to our work is HyperJ [7]. Its core idea is to support multi-dimensional separation of concerns using a mechanism called Hyperspace. HyperJ provides some facilities that are similar to aspects in AspectJ. It does not provide a way to specify to what extent code in one concern space may interact with the code of another concern space.

PROSE [8] is an AOP platform based on the Java Virtual Machine that allows dynamic weaving and un-weaving. An aspect in PROSE is entirely specified in Java. It must first be compiled and given to PROSE, which will apply the aspect to the base class dynamically. PROSE does not limit the interaction between base classes and aspect classes.

Lieberherr et al. [5] have designed a language called Demeter/J. It is a high-level interface to the programming language Java that allows the user to write adaptive Java programs.

Composition filters [1] provide a set of filters that have five important characteristics: Modular extension, Orthogonal extension, Open-ended, Aspect-oriented and Declarative. Neither Demeter/J nor Composition Filters focuses on limiting the power of aspectual code.

Although most existing AOP languages provide some type of access control, we are not aware of any AOP system that has focused on limiting the implicit interfaces that are open to aspects, as we have addressed here.

7. CONCLUSION AND FUTURE WORK

In this paper we make a four-part contribution to research on the design of aspect-oriented programming languages. First, we frame the problem of join point visibility. Second, we identify join point encapsulation as a basic and needed language mechanism. Third, we identify important considerations in the design of such a mechanism. Finally, we have demonstrated the feasibility of providing such a mechanism in the context of a relatively mature aspect-oriented language, namely AspectJ. Our design is notable for exploiting existing elements of the AspectJ language design, and particularly for enabling the encapsulation of crosscutting sets of join points.

Our results are limited in several ways. First, we do not yet have extensive experience using join point encapsulation. Our lack of experience leaves several questions unresolved. One question is whether, in practice, the use of restrictions would significantly reduce designers ability to use aspects effectively. The answer depends in part on whether restrictions can be limited in scope. The mechanism we have provided might, in fact, be too blunt.

Second, our design and implementation are meant as proofs-of-concept. They could be more sophisticated in several dimensions. We view this work as a starting point for an exploration of the problem and design space. That our mechanism is fully implemented as an extension of the AspectJ compiler enables a broader set of researchers to begin such an exploration.

Nor have we addressed some more fundamental issues. One area that we have not explored seriously is that concerning meta-pointcut languages that would be needed to express encapsulation policies based on the form of pointcut by which an aspect attempts to advise a method. The tradeoffs of language complexity for expressiveness in this direction remain as open questions.

As observed at the start of this paper, the essence of aspect-oriented programming is in the exposure of certain phenomena of program execution to crosscutting modification by aspects. One of the most important decisions that an aspect language designer makes is the choice of the kinds of join points to expose.

Some researchers have speculated that the logical conclusion of the aspect paradigm lies in exposure of all execution phenomena implied by the semantics of a given programming language [2]. To the extent that such a future is realized, the importance of join point encapsulation will be magnified. The two ideas together might promise software developers the best of two worlds: the ability to modularize crosscutting concerns involving all manner of program behaviors, on one hand, and the ability to control the undisciplined use of such flexibility, on the other.

8. ACKNOWLEDGEMENTS

This work was supported in part by the National Science Foundation under grants ITR-0086003 and CCR-0092945, and by NASA Langley Research Center. We thank Gregor Kiczales for helping us to understand some of the accessibility criteria visibility restriction mechanisms might support.

9. NOTE TO REVIEWERS

We will make our AspectJ-based implementation available on request. We do plan to make it available to the research community shortly.

10. ADDITIONAL AUTHORS

Additional authors, also from the Department of Computer Science at the University of Virginia:
Yuan Wei (email: `yw3f@cs.virginia.edu`),
Joel Winstead (email: `jaw2u@cs.virginia.edu`), and
Anthony Wood (email: `wood@cs.virginia.edu`).

11. REFERENCES

[1] M. Aksit, K. Wakita, J. Bosch, L. Bergmans, and A. Yonezawa. Abstracting Object Interactions Using Composition Filters. In R. Guerraoui, O. Nierstrasz, and M. Riveill, editors, *Proceedings of the ECOOP'93 Workshop on Object-Based Distributed Programming*, volume 791, pages 152–184. Springer-Verlag, 1994.

[2] M. Jackson. Personal communication between Michael Jackson and Kevin Sullivan.

[3] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 327–353, Budapest, Hungary, June 2001.

[4] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Akşit and S. Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

[5] K. Lieberherr, D. Orleans, and J. Ovlinger. Aspect-Oriented Programming with Adaptive Methods. *Communications of the ACM*, 44(10):39–41, 2001.

[6] B. Meyer. *Object-oriented Software Construction.* Prentice-Hall, New York, London, 1988.

[7] H. Ossher and P. Tarr. Multi-dimensional separation of concerns and the hyperspace approach. In *Proceedings of the Symposium on Software Architectures and Component Technology: The State of the Art in Software Development.* Kluwer, 2000.

[8] A. Popovici, T. Gross, and G. Alonso. Dynamic weaving for aspect oriented programming. In *1st International Conference on Aspect-Oriented Software Development (AOSD)*, Enschede, The Netherlands, April 2002.

[9] M. Shomrat and A. Yehudai. Obvious or not?: regulating architectural decisions using aspect-oriented programming. In *Proceedings of the 1st international conference on Aspect-oriented software development*, pages 3–9. ACM Press, 2002.