Open Modules: Reconciling Extensibility and Information Hiding

Jonathan Aldrich
School of Computer Science
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213, USA
jonathan.aldrich@cs.cmu.edu

ABSTRACT

Aspect-oriented programming systems provide powerful mechanisms for separating concerns, but understanding how these concerns interact can be challenging. In particular, many aspect-oriented programming constructs can violate encapsulation, creating dependencies between concerns that make software evolution more difficult and error-prone.

In this paper, we introduce Open Modules, a mechanism for enforcing a strong form of encapsulation while supporting much of the extensibility provided by languages like AspectJ. Open Modules provide extensibility by allowing clients to advise the interface of a module, but enforce encapsulation by protecting function calls made within the module from external advice. A module can expose semantically important internal events to client aspects through pointcuts in its interface. The module's implementation can change without affecting client advice as long as the semantics of the methods and pointcuts in its interface are preserved. Thus, open modules preserve much of the expressiveness of existing aspect-oriented programming techniques, while providing strong encapsulation guarantees even in the presence of aspects.

1. Solution Name: Open Modules

2. Problem: Encapsulation for Aspects

In his seminal paper, Parnas laid out the classic theory of information hiding: developers should break a system into modules in order to hide information that is likely to change [10]. Thus if change is anticipated with reasonable accuracy, the system can be evolved with local rather than global system modifications, easing many software maintenance tasks. Furthermore, the correctness of each module can be verified in isolation from other modules, allowing developers to work independently on different sub-problems.

Unfortunately, developers do not always respect the information hiding boundaries of modules–it is often tempting to reach across the boundary for temporary convenience, while causing more serious long-term evolution problems. Thus, encapsulation mechanisms such as Java's packages and public/private data members were developed to give programmers compiler support for enforcing information hiding boundaries.

The central insight behind aspect-oriented programming is that conventional modularity and encapsulation mechanisms are not flexible enough to capture many concerns that are likely to change. These concerns cannot be effectively hidden behind information-hiding boundaries, because they are scattered in many placed throughout the system and tangled together with unrelated code. Aspect-oriented programming systems provide mechanisms for modularizing a more diverse set of concerns. However, few aspect-oriented programming projects have addressed the problem of providing an encapsulation facility for aspect-oriented programming.

2.1 Existing Encapsulation Approaches

The most widely-used AOP system, AspectJ, leaves Java's existing encapsulation mechanisms largely unchanged [7]. AspectJ provides new programming mechanisms that capture concerns which crosscut Java's class and package structure. Because these mechanisms can reach across encapsulation boundaries, AspectJ does not enforce information hiding between aspects and other code.

For example, Figure 1 shows how an aspect can depend on the implementation details of another module. The figure shows two different `Shape` subclasses, one representing points and another representing rectangles. Both classes have a method `moveBy`, which moves the rectangles on the screen. An assurance aspect checks certain invariants of the scene every time a shape moves. The aspect is triggered by a pointcut made up of all calls to the `moveBy` function in shapes. We assume the assurance aspect is checking application-level invariants, rather than invariants specific to the `shape` package, and therefore it is defined in a package of its own.

Unfortunately, this aspect depends on the implementation details of the `shape` package, and will break if these implementation details are changed. For example, consider what happens if the rectangle is modified to store its coordinates as a pair of points, rather than two pairs of integer values. The body of `Rectangle.moveBy` would be changed to read:

```
p1.moveBy(dx, dy);
p2.moveBy(dx, dy);
```

Now the `moves` pointcut will be invoked not only when the `Rectangle` moves, but also when its constituent points move. Thus, the scene invariants will be checked in the middle of the rectangle's `moveBy` operation. Since the scene invariants need not be true in the intermediate state of motion, this additional checking could lead to spurious invariant failures.

```
package shape;

public class Point extends Shape {
  public void moveBy(int dx, int dy) {
    x += dx; y += dy;
  ...
}

public class Rectangle extends Shape {
  public void moveBy(int dx, int dy) {
    p1x += dx; p1y += dy;
    p2x += dx; p2y += dy;
  ...
}
```

```
package assure;

aspect AssureShapeInvariants {
  pointcut moves():
      call(void shape.Shape+.moveBy(..));

  after(): moves() {
    scene.checkInvariants();
  }
}
```

Figure 1: In this AspectJ code, the correctness of the shape invariants aspect depends on the implementation of the shapes. If the implementation is changed so that `Rectangle` uses `Point` to hold its coordinates, then the invariants will be checked in the middle of a `moveBy` operation, possibly leading to a spurious invariant failure.

The aspect in Figure 1 violates the information hiding boundary of the `shape` package by placing advice on method calls within the package. This means that the implementor of `shape` cannot freely switch between semantically equivalent implementations of `Rectangle`, because the external aspect may break if the implementation is changed. Because the aspect violates information hiding, evolving the `shape` package becomes more difficult and error prone.

AspectJ is not the only system in which aspects can violate information hiding boundaries. Other aspect-oriented programming systems that support method interception, such as Hyper/J [14] and ComposeJ [15], share the issue. Even recent proposals describing module systems for AOP allow these kinds of violations [8, 4].

Clearly the programmer of the assurance aspect could have written the aspect to be more robust to this kind of change. However, the whole point of an encapsulation system is to protect the programmer from violating information hiding boundaries. In the rest of this paper, we explore a proposed module system that is able to enforce information hiding, while preserving much of the expressiveness of existing aspect-oriented programming systems.

2.2 Encapsulation Goals

What should be the goals of an encapsulation system for aspects?

The motivation for encapsulation mechanisms is to allow programmers to express their information-hiding intent to the compiler, so that the compiler can help them ensure that clients of a module do not take advantage of hidden information. Since we are working in the domain of aspect-oriented programming, it is important that the system can be used to hide information in crosscutting concerns.

The encapsulation system should also ensure that module clients do not rely on hidden information. Reynolds' abstraction theorem is a way of stating this goal more precisely: no client should be able to distinguish two modules that have the same external behavior but differ in implementation details [11]. A system with the abstraction property guarantees that clients will not be affected by changes to information that is hidden within a module.

Both of these goals are necessary to realize the potential of aspect-oriented programming. Without the flexibility to encapsulate crosscutting concerns, AOP is no better than conventional programming systems. Without the abstraction property, we can modularize crosscutting concerns, but programmers will inevitably create dependencies that violate the intended information hiding boundaries. Only with both of these goals will we be able to capture crosscutting concerns *and* evolve them locally, achieving the benefits of modularity. To date, however, we know of no system which supports both common cases of aspect-oriented programming and an abstraction property.

3. Description of Open Modules

We propose Open Modules, a new module system for aspect-oriented programs that is intended to be *open* to aspect-oriented extension but *modular* in that the implementation details of a module are hidden. The goals of openness and modularity are in tension, and so we try to achieve a compromise between them.

Previous systems support AOP using constructs like advice that can reach across module boundaries to capture crosscutting concerns. We propose to adopt these same constructs, but limit them so that they respect module boundaries. In order to capture concerns that crosscut the boundary of a module, the module can export pointcuts that represent *abstractions* of events that might be relevant to external aspects. As suggested by Gudmundson and Kiczales [6], exported pointcuts form a contract between a module and its client aspects, allowing the module to be evolved independently of its clients so long as the contract is preserved.

Figure 2 shows a conceptual view of Open Modules. Like ordinary module systems, open modules export a list of data structures and functions such as `moveBy` and `animate`. In addition, however, open modules can export pointcuts denoting internal semantic events. For example, the `moves` pointcut in Figure 2 is triggered whenever a shape moves. Since a shape could move multiple times during execution of the `animate` function, clients interested in fine-grained motion information would want to use this pointcut rather than just placing advice on calls to `animate`.

By exporting a pointcut, the module's maintainer is making a promise to maintain the semantics of that pointcut as the module's implementation evolves, just as the maintainer must maintain the semantics of the module's exported functions.

Open Modules are "open" in two respects. First, their interfaces are open to advice; all calls to interface functions from outside the module can be advised by clients. Second, clients can advise exported pointcuts.

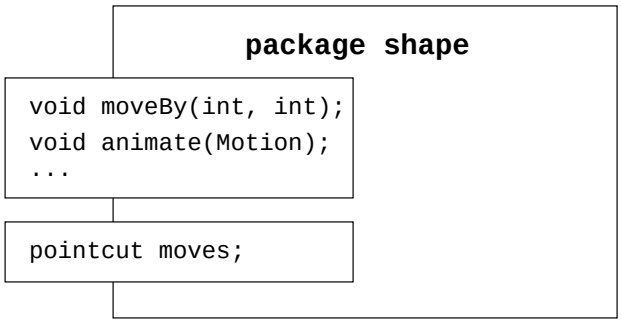

Figure 2: A conceptual view of Open Modules. The `shape` module exports two functions and a pointcut. Clients can place advice on external calls to the exported functions, or on the exported function, but not on calls that are internal to the module.

On the other hand, open modules encapsulate the internal implementation details of a module. As usual with module systems, functions that are not exported in the module's public interface cannot be called from outside the module. In addition, calls between functions within the module cannot be advised from the outside–even if the called function is in the public interface of the module.

For example, a client could place advice on external calls to `moveBy`, but not calls to `moveBy` from another function within the same module. Thus, in our system the code in Figure 1 would trigger a compile time error, because the `moves` pointcut includes calls that are within the `shape` package, violating its encapsulation.

We now provide a more technical definition for Open Modules, which can be used to distinguish our contribution from previous work:

Definition [Open Modules]: *A module system that:*

- *allows external aspects to advise external calls to functions in the interface of a module*
- *allows external aspects to advise pointcuts in the interface of a module*
- *does not allow external aspects to advise calls from within a module to other functions within the module (including exported functions).*

3.1 Open Module Examples

In Open Modules, there are two ways to revise the code in Figure 1 so that the information hiding boundaries of the `shapes` package are maintained. The first solution, shown in Figure 3, simply adds a clause to the pointcut ensuring that it only captures external calls to `moveBy`. Internal calls from within the `shape` package are not part of the pointcut. Thus, if the `Rectangle` implementation is later changed to use `Point`,the assurance aspect will not be affected.

A second possible revision is shown in Figure 4. In this example, the `moves` pointcut has been defined inside the `shape` package, and is exported so that external clients can use it as an extension point. The pointcut will still need to be changed if the implementation of `Rectangle` is changed to use `Point`. However, these two changes are now both the responsibility of the maintainer of the `shape` package, making it easier to keep the pointcut coordinated with the ordinary Java code.

```
package assure;

aspect AssureShapeInvariants {
  pointcut moves():
      call(void shape.Shape+.moveBy(..))
      && !within(shape.*);
  ...
}
```

Figure 3: The violation of information hiding in Figure 1 can be fixed by changing the pointcut so that calls from within the `shape` package are not included in the pointcut. This allows the `shape` package to be changed without affecting the assurance aspect.

```
package shape;

class Shape {
  public pointcut moves:
      call(void Shape+.moveBy(..));
  ...
}


package assure;

aspect AssureShapeInvariants {
  after(): shape.Shape.moves() {
    scene.checkInvariants();
  }
}
```

Figure 4: A second fix to the information hiding problem is to define the coumoves pointcut within the `shape` package. The maintainer of the `shape` package must maintain the semantics of this pointcut as the package evolves.

Although this second solution suggests that the pointcut for the aspect be moved into the base code, in general there will be a division of responsibility for pointcuts between the aspect and the base code. For example, the assurance aspect may not be interested in all moves, but only moves that are within the control flow of some higher-level operation. It would be inappropriate (and unnecessary) for the base code to include a pointcut that was specialized to the particular needs of the aspect. Instead, a pointcut representing generic move events can be defined in the `shape` package as described in Figure 4, and the aspect can refine the pointcut by specifying the control flow pointcut, as follows:

```
pointcut relevantMoves : shape.Shape.moves()
                         && cflow(...);
```

3.2 Semantics

This paper describes a design in progress. We have not yet worked out the semantics of Open Modules in a full aspect-oriented programming system such as AspectJ. To do so will be a challenging task, because the design principles of Open Modules may require rethinking many different aspects of the way that pointcuts and advice are defined.

As a start towards a semantic understanding of Open Modules, a companion paper gives a precise semantics for Open Modules in a core language consisting of the lambda calculus plus pointcuts and advice [1]. In addition to proving the usual type soundness properties, this paper shows that Open Modules enforce Reynolds' abstraction property. As discussed before, this property ensures that clients cannot be affected by changes to the implementation of a module, as long as those changes preserve the semantics of the module's exported functions and pointcuts.

4. Open Modules and Comprehensibility

Open Modules increase the comprehensibility of a system relative to existing aspect-oriented programming systems, because the implementation and behavior of a module can be completely understood in isolation. External advice cannot affect internal calls within the module, so it is easy to reason about control and data flow within the module. Although clients can advise exported pointcuts, this advice can be treated as a callback to a client-specified function, and then standard reasoning techniques can be used.

5. Open Modules and Predictability

Compared to other AOP systems, Open Modules make it easier to predict the results of making small changes to the implementation of a system. Because our system enforces encapsulation boundaries, changes to code within a module can only directly affect other code and aspects within that same module. As long as the semantics of the module's interface are preseved, external code and aspects will not be affected.

6. Open Modules and Evolvability

Research on both aspect-oriented programming and module systems is predicated on the belief that better ways to support information hiding will ease software evolution. The premise of this paper is that Open Modules combine the most important properties of both lines of research: the ability to capture crosscutting concerns with the ability to enforce information hiding boundaries.

7. Open Modules and Semantic Interactions

Open Modules do not solve the problem of semantic interactions between different aspects. However, our system does make the problem easier by limiting the ways in which aspects can interact.

If both an aspect and the code it advises are part of some package P, then that aspect can only interact with other aspects defined in package P. This is true because external aspects cannot advise the implementation of P, only calls to its interface.

Although external aspects can advise pointcuts exported by package P, interaction with internal aspects via these pointcuts is relatively controlled. A mechanism similar to AspectJ's aspect precedence declarations could be used to specify whether the internal aspects apply before or after external aspects.

The semantic interaction problem remains challenging for aspects defined within the same high-level module. Many tools, such as the AspectJ plugin for Eclipse, help with the semantic interaction problem by showing a view of where aspects apply to base code.

8. Expressiveness

Like the Gudmundson and Kiczales proposal on which they are based [6], Open Modules sacrifice some amount of *obliviousness* [5] in order to support better information hiding. Base code is not completely oblivious to aspects, because the author of a module must expose relevant internal events in pointcuts so that aspects can advise them[1]. However, our design still preserves important cases of obliviousness:

- While a module can expose interesting implementation events in pointcuts, it is oblivious to which aspects might be interested in those events.
- Pointcuts in the interface of a module can be defined *non-invasively* with respect to the rest of the module's implementation, using the same pointcut operations available in other AOP languages.
- A module is completely oblivious to aspects that only advise external calls to its interface.

A possible concern is that the strategy of adding a pointcut to the interface of a base module may be impossible if the source code for that module cannot be changed. In this case, the modularity benefits of Open Modules can be achieved with environmental support for associating an external pointcut with the base module. If the base module is updated, the maintainer of the pointcut is responsible for rechecking the pointcut to ensure that its semantics have not been invalidated by the changes to the base module.

8.1 Empirical Study

We hypothesize that open modules are expressive enough to support many existing uses of aspect-oriented programming. We have conducted a micro-experiment that, while not sufficient to test this hypothesis in the general case, still serves as a reality check to make sure the hypothesis is not unreasonable.

In our experiment, we examined the SpaceWar program from the AspectJ compiler distribution. At 2300 lines of code, SpaceWar is a small demonstration program, not a realistic application. However, it is the largest of the examples in the distribution, and it shows a variety of interesting uses of aspects.

Our methodology was to examine the pointcuts used for a number of different purposes, to see if those pointcuts violated module boundaries as described earlier in this paper. Since SpaceWar is not divided into packages, we used files as the module boundaries in this study.

Results. There were 11 pointcuts in the SpaceWar program, not counting the debugging pointcuts (discussed below). Of these, 4 pointcuts were compatible with Open Modules as written. For example, the following pointcut in `Display.java` repaints the display objects after each clock tick. It is compatible with Open Modules because it intercepts messages going into the Game object, rather than internal messages.

[1]We note that many in the AOP community feel "obliviousness" is too strong a term, preferring a notion of "non-invasiveness" that is compatible with our proposal. See for example posts to the aosd-discuss mailing list by Dean Wempler and Gregor Kiczales in August 2003, available at aosd.net.

```
after() ... : call(void Game.clockTick()) {
   //for each Display object ‘‘display’’ do:
       display.repaint();
}
```

Six other pointcuts would have to be moved into the target module in order to be compatible with Open Modules, because they intercept messages that were internal to that module. For example, the pointcut below is used to synchronize access to data structures in the Game object. Because `handleCollisions` is called from within `Game.java`, this pointcut would have to be defined in `Game.java` rather than in an external file.

```
protected pointcut synchronizationPoint() :
       call(void Game.handleCollisions(..))
    || call(Ship Game.newShip(..));
```

The `Ship` class already includes a semantic pointcut in its interface, shown below. This pointcut captures "commands" to the ships in the SpaceWar program, and is used by an external aspect that cancels the command if the ship is disabled. The use of this semantic pointcut in existing code supports the argument that putting pointcuts in a module interface is a natural programming idiom.

```
pointcut helmCommandsCut(Ship ship):
    target(ship) &&
        ( call(void rotate(int))     ||
          call(void thrust(boolean)) ||
          call(void fire()) );
```

It was unclear how the one remaining pointcut should be supported in our system. This pointcut ensures that objects are only "registered" with the game when they are created and destroyed. It is unusual in that it talks not only about the method being called but also the method from which the call is made. Further study will be necessary to understand how to integrate this kind of pointcut into our system.

The only concern our system definitely could not handle in a reasonable way was an extremely invasive debugging aspect. Debugging is an inherently non-modular activity, so we view it as a positive sign that our module system does not support it. In a practical system, debugging can be supported either through external tools, or through a compiler flag that makes an exception to the encapsulation rules during debugging activity.

A file containing the raw, detailed results of our study is available at the Open Modules website, http://www.cs.cmu.edu/~aldrich/aosd/.

Lessons Learned. The fact that 6 out of 11 pointcuts required moving the pointcut into the interface of the base code suggests that we should think about a way to make this easier. One possibility is to add a new method modifier stating that all calls to this method (including calls from the current module) define a pointcut by the same name as the method. This would make it easier to expose pointcuts in a way that is compatible with Open Modules.

In summary, our study of the aspects in the SpaceWar program provided preliminary evidence that Open Modules are able to capture many existing aspects with only minor changes to the way those aspects are expressed–usually moving a pointcut into the interface of the base code. In the future, we hope to test this hypothesis on more realistic applications.

8.2 Comparison to non-AOP techniques.

One way to evaluate the expressiveness of Open Modules is to compare them to non-AOP alternatives. One alternative is using wrappers instead of aspects to intercept the incoming calls to a module, and using callbacks instead of pointcuts in the module's interface. The aspect-oriented nature of Open Modules provides several advantages over the wrapper and callback solution:

- Open Modules are compatible with the *quantification* [5] constructs of languages like AspectJ, so that many functions can be advised with a single declaration. Implementing similar functionality with conventional wrappers–which do not support quantification–is far more tedious because a wrapper must be explicitly applied to each function.
- In Open Modules, a single, locally-defined aspect can implement a crosscutting concern by non-locally extending the interface of a number of modules. Wrappers cannot capture these concerns in a modular way, because each target module must be individually wrapped.
- Callbacks are invasive with respect to the implementation of a module because the implementation must explicitly invoke the callback at the appropriate points. In contrast, pointcut interfaces are non-invasive in that the pointcut is defined orthogonally to the rest of the module's implementation, thus providing better support for separation of concerns.

These advantages illustrate how the quantification and non-invasive extension provided by Open Modules distinguish our proposal from solutions that do not use aspects [5].

9. Discussion and Conclusion

In this section we discuss related work and our future plans before concluding the paper.

9.1 Related Work

Lieberherr et al. describe Aspectual Collaborations, a construct that allows programmers to write aspects and code in separate modules and then compose them together into a third module [8]. Their module system does not encapsulate internal calls to exported functions, and thus does not enforce the abstraction property.

In concurrent work, Dantas and Walker propose a module system for aspect-oriented programming [4]. Their system includes a novel feature for controlling whether advice can read or change the arguments and results of advised functions. In their design, pointcuts are first-class, providing more flexibility compared to the second-class pointcuts in systems like AspectJ. This design choice breaks abstraction and thus separate reasoning, however, because it means that a pointcut can escape from a module even if it is not explicitly exported in the module's interface. In their system, functions in the interface of a module can only be advised if this is planned in advance; in contrast, Open Modules allows advice on all function declarations in a module's interface, providing oblivious extensibility without compromising abstraction.

Open Modules is conceptually similar to the Composition Filters model, in that external advice to a module can be

thought of as a filter that acts on incoming messages. However, the most recent implementation of Composition Filters, ComposeJ, differs from our proposal in that messages sent to "this" are dispatched through filters rather than being sent directly to the receiver object, thus breaking the abstraction property enforced by our system [15].

AspectJ [7] extends Java's encapsulation mechanisms, protecting private methods from access by external aspects. However, AspectJ does not enforce abstraction, because internal calls to public methods may still be advised by external aspects. Furthermore, an aspect can get around even the limited encapsulation mechanism by declaring itself to be privileged. Thus, AspectJ's design provides only minimal enapculation, but gives programmers maximum flexibility in writing aspects.

A different approach to reasoning about code interactions in aspect-oriented programs is to provide an analysis that shows how aspects might affect source code or each other. For example, the Eclipse plugin for AspectJ includes a view showing which aspects affect each line of source code, and researchers have studied more sophisticated analyses [12, 13]. These analyses, however, do not prevent abstraction violations in the way that Open Modules do.

Clifton and Leavens propose to modularize reasoning about aspects using the concepts of observers and assistants [2]. Observers can observe a module, but not change its semantics, while assistants can change a module's behavior, but only with that module's permission. Open Modules enforce a stronger barrier between a module and its clients, because even "observer aspects" cannot depend on the internal implementation details of a module. Observers and assistants have a complementary advantage, supporting stronger reasoning about how different aspects might interact.

The technique of exporting a pointcut in a module's interface was originally proposed by Gudmundson and Kiczales as a way to ease software evolution by decoupling an aspect from the code that it advises [6]. It is also related to the Demeter project's use of *traversal strategies* to isolate an aspect from the code that it advises [9].

The name Open Modules indicates that modules are open to advice on functions and pointcuts exposed in their interface. This terminology is derived from Open Classes, a related AOP term indicating that classes are open to the addition of new methods [3].

9.2 Future Work

In future work, we plan to extend the design sketched here to a full-fledged aspect-oriented programming system such as AspectJ or Hyper/J. We also hope to perform an empirical study evaluating whether open modules are expressive enough to handle common examples of aspect-oriented programming.

9.3 Conclusion

This paper presented Open Modules, an encapsulation mechanism that provides much of the expressiveness of existing aspect-oriented programming mechanisms while enforcing information hiding boundaries. Open Modules are open in the sense that clients can place advice both on pointcuts and on functions in the interface of a module. A key contribution of our work is recognizing that in order to enforce encapsulation in the presence of aspects, it is necessary to distinguish between external calls to a module and internal calls within the module. Open Modules enable aspect-oriented programming to reach its full potential: not only separating concerns but also ensuring that programmers can separately reason about and evolve those concerns.

10. Acknowledgments

I thank the anonymous reviewers and my colleagues in the software group at CMU for their valuable feedback on earlier drafts of this material.

11. REFERENCES

[1] J. Aldrich. Open Modules: A Proposal for Modular Reasoning in Aspect-Oriented Programming. In *Foundations of Aspect Languages*, March 2004.

[2] C. Clifton and G. T. Leavens. Observers and Assistants: A Proposal for Modular Aspect-Oriented Reasoning. In *Foundations of Aspect Languages*, April 2002.

[3] C. Clifton, G. T. Leavens, C. Chambers, and T. Millstein. MultiJava: Modular Open Classes and Symmetric Multiple Dispatch for Java. In *Object-Oriented Programming Systems, Languages, and Applications*, October 2000.

[4] D. S. Dantas and D. Walker. Aspects, Information Hiding and Modularity. Unpublished manuscript, 2003.

[5] R. E. Filman and D. P. Friedman. Aspect-Oriented Programming is Quantification and Obliviousness. In *Advanced Separation of Concerns*, October 2000.

[6] S. Gudmundson and G. Kiczales. Addressing Practical Software Development Issues in AspectJ with a Pointcut Interface. In *Advanced Separation of Concerns*, July 2001.

[7] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An Overview of AspectJ. In *European Conference on Object-Oriented Programming*, June 2001.

[8] K. Lieberherr, D. H. Lorenz, and J. Ovlinger. Aspectual Collaborations: Combining Modules and Aspects. *The Computer Journal*, 46(5):542–565, September 2003.

[9] D. Orleans and K. Lieberherr. DJ: Dynamic Adaptive Programming in Java. In *Reflection 2001: Meta-level Architectures and Separation of Crosscutting Concerns*, September 2001.

[10] D. L. Parnas. On the Criteria to be Used in Decomposing Systems into Modules. *Communications of the ACM*, 15(12):1053–1058, December 1972.

[11] J. C. Reynolds. Types, Abstraction, and Parametric Polymorphism. In *Information Processing*, 1983.

[12] G. Snelting and F. Tip. Semantics-based Composition of Class Hierarchies. In *European Conference on Object-Oriented Programming*, June 2002.

[13] M. Storzer and J. Krinke. Interference Analysis for AspectJ. In *Foundations of Aspect Languages*, March 2003.

[14] P. Tarr, H. Ossher, W. Herrison, and S. M. Sutton. N Degrees of Separation: Multi-Dimensional Separation of Concerns. In *Interntional Conference on Software Engineering*, May 1999.

[15] J. C. Wichman. ComposeJ - The Development of a Preprocessor to Facilitate Composition Filters in the Java Language. Masters Thesis, University of Twente, 1999.