Selective Open Recursion: Modular Reasoning about Components and Inheritance

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

Kevin Donnelly
Computer Science Department
Boston University
111 Cummington Street
Boston, MA 02215, USA
kevind@bu.edu

ABSTRACT

Current component-based systems with inheritance do not fully protect the implementation details of a class from its subclasses, making it difficult to evolve that implementation without breaking subclass code. Previous solutions to the so-called fragile base class problem *specify* those implementation dependencies, but do not *hide* implementation details in a way that allows effective software evolution.

In this paper, we show that one instance of the fragile base class problem arises because current object-oriented languages dispatch methods using *open recursion* semantics even when these semantics are not needed or wanted. Our solution, called *Selective Open Recursion*, makes explicit the methods to which open recursion should apply. As a result, classes can be more loosely coupled from their subclasses, and therefore can be evolved more easily without breaking subclass code.

We have implemented Selective Open Recursion as an extension to Java, along with an analysis that automatically infers the necessary program annotations. We have collected data for the Java standard library suggesting that the additional programmer effort required by our proposal is low, and that Selective Open Recursion aids in automated reasoning such as compiler optimizations.

1. Inheritance and Information Hiding

In his seminal paper, Parnas laid out the classic theory of information hiding: developers should break a system into modules or components 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 component can be verified in isolation from other components, allowing developers to work independently on different sub-problems.

Unfortunately, developers do not always respect the information hiding boundaries of components—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.

While the encapsulation mechanisms provided by Java and other languages can help to enforce information hiding boundaries between an object and its clients, enforcing information hiding between a class and its subclasses is more challenging. The `private` modifier can be used to hide some method and fields from subclasses. However, inheritance creates a tight coupling between a class and its subclasses, making it difficult to hide information about the implementation of `public` and `protected` methods in the superclass. In component-based systems with inheritance, it is easy for a subclass to become unintentionally dependent on the implementation details of its superclass, and therefore to break when the superclass changes in seemingly innocuous ways.

```
public class CountingSet extends Set {
  private int count;

  public void add(Object o) {
    super.add(o);
    count++;
  }
  public void addAll(Collection c) {
    super.addAll(c);
    count += c.size();
  }
  public int size() {
    return count;
  }
}
```

Figure 1: The correctness of the `CountingSet` class depends on the independence of `add` and `addAll` in the implementation of `Set`. If the implementation is changed so that `addAll` uses `add`, then `count` will be incremented twice for each element added.

1.1 The Fragile Base Class Problem

This issue is known as the *Fragile Base Class Problem*, one of the most significant challenges faced by designers of object-oriented component libraries. Figure 1 shows an example of the fragile base class problem, taken from the literature [12, 9, 2].

In the example, the `Set` class has been extended with an optimization that keeps track of the current size of the set in an additional variable. Whenever a new element or collection of elements is added to the set, the variable is updated appropriately.

Unfortunately, the implementation of `CountingSet` makes assumptions about the implementation details of `Set`—in particular, it assumes that `Set` does not implement `addAll` in terms of `add`. This coupling means that the implementation of `Set` cannot be changed without breaking its subclasses. For example, if `Set` was changed so that the `addAll` method calls `add` for each member of the collection in the argument, the `count` variable will be updated not only during the call to `addAll`, but also for each individual `add` operation—and thus it will end up being incremented twice for each element in the collection.

The root of the problem described above (which is just one instance of the larger fragile base class problem) is that the subclass depends on the calling patterns between methods in its superclass. Object-oriented languages provide *open recursion*, in which self-calls are dynamically dispatched, allowing subclasses to intercept self-calls from the superclass and thus depend on when it makes these calls. Open recursion is useful for many object-oriented programming idioms—for example, the template method design pattern [7] uses open recursion to invoke customized code provided by a subclass. However, sometimes making a self-call to the current object is just an implementation convenience, not a semantic requirement. The whole point of encapsulation is to ensure that subclasses do not depend on such implementation details, so that a class and its subclasses can be evolved independently. Thus inheritance breaks encapsulation when implementation-specific self-calls are made.

Examples like these have led some to call inheritance antimodular. Most practitioners, recognizing the value of inheritance for achieving software reuse in component-based systems, would not go so far, but this example illustrates that reasoning about correctness is challenging in the presence of inheritance.

A number of previous papers have addressed the fragile base class problem in various ways [8, 12, 13, 2, 11]. These solutions, however, either give up the power of open recursion entirely, or expose details of a class's implementation that ought to be private (such as the fact that the `addAll` method calls or does not call `add` in the example above).

1.2 Contributions

The contribution of this paper is Selective Open Recursion, a new approach that provides the benefits of inheritance and open recursion where they are needed, but allows programmers to effectively hide many details of the way a class is implemented. In our system, described in the next section, method calls on the current object `this` are dispatched statically by default, meaning that subclasses cannot intercept internal calls and thus cannot become dependent on those implementation details. External calls to the methods of an object—i.e., any method call not explicitly invoked on `this`—are dynamically dispatched as usual.

If an engineer needs open recursion, she can declare a method "open," in which case self-calls to that method are dispatched dynamically. By declaring a method "open," the author of a class is promising that any changes to the class will preserve the ways in which that method is called.

In section 3, we describe our implementation of Selective Open Recursion as an extension to Java. We have implemented a static, whole-program analysis that annotates an existing Java program with the minimal set of "open" declarations that are necessary so that the program has the same semantics in our system. Results of applying our analysis to the JDK 1.4 standard library show that open annotations are rarely needed and that Selective Open Recursion increases the potential for program optimizations such as inlining. Section 4 discusses related work and section 5 concludes.

2. Selective Open Recursion

We argue that the issue underlying the instance of the fragile base class problem described above is that current languages do not allow programmers to adequately express the intent of various methods in a class. There is an important distinction between methods used for communication between a class and its clients, vs. methods used for communication between a class and its subclasses.

Some methods are specifically intended as callbacks or extension points for subclasses. These methods are invoked recursively by a class so that its subclasses can provide customized behavior. Examples of callback methods include methods denoting events in a user interface, as well as abstract "hook" methods in the template method design pattern [7]. Because callback methods are intended to be invoked whenever some semantic event occurs, any changes to the base class must maintain the invariant that the method is always invoked in a consistent way.

In contrast, many accessor and mutator functions are primarily intended for use by clients. If the implementation of a class also uses these functions, it is typically as a convenience, *not* because the class expects subclasses to override the function with customized behavior. The fragile base class problem described above occurs exactly when a "client-oriented" method is called recursively by a superclass, but is also overridden by a subclass.[1] Because the recursive call to the method was never intended to be part of the subclassing interface, the maintainer of the base class should be able to evolve the class to use (or not use) such methods without affecting subclasses.

The key insight underlying Selective Open Recursion is that subclasses do not need to intercept recursive calls to methods that were not intended as callbacks or extension points—they can always provide their behavior by overriding the external interface of a class. At most, intercepting recursive calls to "client-oriented" methods is only a minor convenience, and one that creates an undesirable coupling between subclass and superclasses.

We thus propose to add a new modifier, `open`, which allows developers to more fully declare their underlying design intent. An `open` method has open recursion semantics; it is treated as a callback for subclasses that will always be recursively invoked by the superclass whenever some conceptual event occurs. Ordinary methods—those without the `open` keyword—are not part of the subclassing interface. While external calls to ordinary methods are dynamically dispatched as usual, recursive calls where the receiver is explicitly stated to be the current object `this` are dispatched statically.[2] Because open recursion does not apply to methods that are not marked `open`, subclasses cannot depend on

[1] There are other instances of the fragile base class problem—for example, name collisions between methods in a class and its subclasses—that we do not consider here.

[2] If the receiver is not syntactically `this` but is an alias, we treat the call as external. This ensures that the dispatch mechanism used is easily predicted by browsing the source code.

```
public class Set {
  List elements;

  public void add(Object o) {
    if (!elements.contains(o))
      elements.add(o);
  }
  public void addAll(Collection c) {
    Iterator i = c.iterator();
    while (i.hasNext())
      this.add(i.next());
  }
}
```

Figure 2: In the first solution to the problem described in Figure 1, the developer decides not to mark either `add` or `addAll` as `open`. Thus, when `addAll` invokes `add`, the call is dispatched statically, so that `Set`'s implementation of `add` executes even if a subclass overrides the `add` method (Client calls to `add` are dispatched dynamically as usual). Thus, subclasses cannot tell if `addAll` was implemented in terms of `add` or not, allowing the maintainer of `Set` to change this decision.

```
public class Set {
  List elements;

  /* called once for every added element */
  public void open add(Object o) {
    if (!elements.contains(o))
      elements.add(o);
  }
  public void addAll(Collection c) {
    Iterator i = c.iterator();
    while (i.hasNext())
      this.add(i.next());
  }
}
```

Figure 3: In the second solution to the problem described in Figure 1, the developer decides that the `add` method denotes a semantic event of interest to subclasses, and therefore marks `add` as `open`. By doing this, the developer is promising that any correct implementation of `Set` will call `add` once for each element added to the set. Therefore, a subclass interested in "add element" events can override the `add` method without overriding `addAll`.

when they are invoked by the superclass, and the fragile base class problem cannot occur.

In our proposal, there are two choices a designer can make to solve the problem described in Figure 1. In the first solution, shown in Figure 2, the designer of the `Set` class has decided that neither `add` and `addAll` are intended to act as subclass callbacks, and so neither method was annotated `open`. In this case, subclasses cannot tell whether `addAll` is implemented in terms of `add` or not, and so the fragile base class problem cannot arise. Even if `addAll` calls `add` on the current object `this`, this call will be dispatched statically and so subclasses cannot intercept it. Note that calls to `add` from clients are dispatched dynamically as usual, so that an implementation of `CountingSet` can accurately track the element count simply by overriding both `add` and `addAll`.

In the second solution, shown in Figure 3, the designer of the `Set` class has decided that `add` represents a semantic event (adding an element to the set) that subclasses may be interested in reacting to. The designer therefore annotates `add` as `open`, documenting the promise that even if the implementation of `Set` changes, the `add` method will always be called once for each element added to the set. The implementor of `CountingSet` can keep track of the element count by overriding just the `add` function. Any changes to the `Set` class will not break the `CountingSet` code, because the implementor of `Set` has promised that any changes to `Set` will preserve the semantics of calls to `add`.

2.1 Using Selective Open Recursion

With any new language construct, it is important not only to describe the construct's meaning but also how to use it effectively. We offer tentative guidelines for the use of `open`, which can be refined as experience is gained with the construct.

We expect that `public` methods will generally not be `open`. The rationale for this guideline is that `public` methods are intended for use by clients, not by subclasses. In general, any internal use of these public methods is probably coincidental, and subclasses should not rely on these calls. There are exceptions—for example, the `add` method could be both `public` and `open`, depending on the designer's intent—but these idioms can also be expressed by having the `public` method invoke a `protected open` method. For example, instead of making the `add` method `open`, the developer could implement both `add` and `addAll` in terms of a `protected, open internalAdd` method that serves as the subclass extension point. Using this `protected` method solution is potentially cleaner because it separates the client interface from the subclassing interface.

On the other hand, we expect that `protected` methods will either be `final` or `open`. Protected methods are usually called on the current object `this`, so overriding them is useful only in the presence of open recursion. Protected methods that are not intended to represent callbacks or extension points for subclasses should be marked as `final`.

Private methods in languages like Java are unaffected by our proposal; since they cannot be overridden, open recursion is not relevant.

2.2 An Alternative Proposal

The discussion above suggests an alternative proposal: instead of adding a new keyword to the programming language, simply use open recursion dispatch semantics for all (non-`final`) `protected` methods and treat all `public` methods as if they were non-open. This alternative has the advantage of simplicity; it takes advantage of common patterns of usage, does not add a new keyword to the language, and encourages programmers to cleanly separate the public client interface from the protected subclass interface.

However, there are two disadvantages to the alternative. If, in addition to performing a service for a client, a public method also represents an event that subclasses may want to extend, the programmer will be forced to create an additional protected method for the subclass interface, creating a minor amount of code bloat. Furthermore, the proposal that makes `open` explicit is a more natural evolutionary path;

existing Java code need only be annotated with `open` (perhaps with the analysis described in Section 4), whereas in the alternative proposal `public` methods that are conceptually `open` would have to be re-written as a pair of `public` and `protected` methods.

2.3 Applications to Current Languages

Our proposal extends languages like Java and C# in order to capture more information about how a class can be extended by subclasses. However, the idea of "open" methods can also be applied within existing languages, providing engineering guidelines for avoiding problematic uses of inheritance.

The discussion above suggests that developers should avoid calling `public` methods on the current object `this`. If a `public` method contains code that can be reused elsewhere in the class, the code should be encapsulated in a `protected` or `private` method, and the `public` method should call that internal method. This guideline was previously suggested by Ruby and Leavens [11], and appears to be common practice within the Java standard library in any case. For example, the `java.util.Vector` class in the JDK 1.4.2 internally calls a `protected` method, `ensureCapacityHelper`, to verify that the underlying array is large enough—even though the `public` method `ensureCapacity` could be used instead.

Protected methods should be `final` if they don't represent an explicit extension point for subclasses. The author of a library should carefully document under which circumstances non-`final` protected methods are called, so that subclasses can rely on the semantics.

Methodological solutions like this one have the advantage that they do not change the semantics of the language. However, for a methodology to be effective, it must be followed. The advantage of Selective Open Recursion is that the keyword `open` encourages developers to make an explicit choice about the nature of the methods they define, then enforces that choice naturally through the dispatch semantics o the language. Thus, both the syntax and semantics of our proposal work together to reinforce good use of inheritance, while a methodological solution relies primarily on programmer discipline, possibly augmented with `lint`-like style checkers.

2.4 A Rejected Alternative Design

Based on the insight that the fragile base class problem arises when open recursion is used unintentionally, there is a natural alternative design to be considered. In the discussion above, we chose to annotate methods as being `open` or not; an alternative is to annotate call sites as using dynamic or static dispatch. We rejected this alternative for two reasons. First, it is a poor match for the design intent, which associates a method—not a call site—with a callback or extension point. Second, because the design intent is typically associated with methods, it would be very surprising if different recursive calls to the same method were treated differently. By annotating the method rather than the call site, our proposal helps developers be consistent.

2.5 Family Polymorphism

The fragile base class problem can be generalized to sets of classes that are closely related. For example, if a `Graph` module defines classes for nodes and edges, it is likely that the node and edge class are closely related and will often be inherited together. Just as self-calls in an object-oriented setting can be mistakenly "captured" by subclasses, calls between node and edge superclasses might be mistakenly captured by node and edge subclasses.

This paper is primarily focused on the version of the problem that is restricted to a single subclass and superclass, in part because the right solution is more clear-cut in this setting. However, some languages provide first-class support for extending related classes together through mechanisms like Family Polymorphism [6]. In this setting, our proposal could potentially be generalized to distinguish between inter-object calls that should be dispatched dynamically and those that should be dispatched statically. Further work is needed to understand how to apply our proposal effectively in this setting.

2.6 Pure Methods

A central aspect of our approach is that a class must document the circumstances under which all of its open methods are called internally. As suggested by Ruby and Leavens [11], it is possible to relax this requirement for *pure* methods which have no (visible) side-effects and do not change their result with inheritance. Since these methods have no effects and always return the same result, a class can change the way in which they are called without affecting subclasses. An auxiliary analysis or type system could be used to verify that pure methods have no effects, including state changes (other than caches), I/O operations, or non-termination.

2.7 Specification and Verification Benefits

We believe that Selective Open Recursion has potential benefits to formal specification and verification techniques. Intuitively, reasoning about non-`open` methods is easier than reasoning about `open` methods, since calls to `open` methods on `this` must be formally treated as callbacks to methods of a subclass. Reasoning about code with callbacks to an unknown function defined in a subclass is inherently more challenging than analyzing a locally-defined set of functions, because the analysis results will be dependent or parameterized by the behavior of that function. By making open recursion selective, our technique can reduce the number of callbacks that formal verification techniques must confront. We are currently working to make these intuitions more precise by proving a representation independence theorem in a formal model of selective open recursion.

3. Implementation and Analysis

We have implemented Selective Open Recursion as an extension to the Barat Java compiler [3]. Our implementation strategy leaves `open` methods and `private` methods unchanged. For each non-open `public/protected` method in the source program, we generate another `protected final` method containing the implementation, and rewrite the original method to call the new method. We leave all calls to `open` methods unchanged, as well as all calls to methods with a receiver other than `this`. For every call to a non-open method that has `this` as the receiver, including implicit uses of `this` but not other variables aliased to `this`, we call the corresponding implementation method, thus simulating static dispatch.

Our implementation of Selective Open Recursion is available at `http://www.archjava.org/` as part of the open

source ArchJava compiler.

3.1 Inference of Open Recursion

In order to ease a potential transition from standard Java or C# to a system with Selective Open Recursion, we have implemented an analysis that can automatically infer which methods must be annotated with `open` in order to preserve the original program's semantics. Of course, our system is identical to Java-like languages if every method is `open`, so the goal of the analysis is to introduce as few `open` annotations as possible. Extra `open` annotations are problematic because they create the possibility of using open recursion when it was not intended, thus triggering fragile base class problems like the example above. In general, no analysis can do this perfectly, because the decision to make a method `open` is a design decision that may not be expressed explicitly in the source code. However, an analysis can provide a reasonable (and safe) default that can be refined manually later.

In order to gain precision, our analysis design assumes that whole-program information is available. A local version of the analysis could be defined, but it would have to assume that every method called on `this` is `open`, because otherwise some unknown subclass could rely on the open recursion semantics of Java-like languages. This assumption would be extremely conservative, so much so that it would be likely to obscure any potential benefits of Selective Open Recursion.

Our analysis design examines each `public` and `protected` method m of every class C. The program potentially relies on open recursion for calls to m whenever there is some method m' in a class $C' \leq C$ that calls m on `this`, and some subclass C'' of C' overrides m, and that subclass either doesn't override m' or makes a super call to m'. The analysis conservatively checks this property, and determines that the method should be annotated `open` whenever the property holds.

3.2 Experiments

We have applied our analysis to a large portion of the Java library, namely all of the packages starting with `java` except for `java.nio` (which was more difficult to compile due to the code generation that is used in that package), and `java.sql` (which triggered a bug in our implementation). We used the JDK 1.4.2 as our codebase.

A threat to validity of this experiment is that a true determination of which methods might be open would have to make a closed world assumption, implicitly considering all possible clients of the library. The library developer might have created "hook" methods for use by future clients that are self-called and ought to be open, but which are not overridden within the library. Our analysis will not catch these methods as being open. However, we believe that because there is substantial use of inheritance and overriding within the library, our analysis should find most of the relevant open methods.

Open Annotations. There are 9897 method declarations in the portion of the standard library that we analyzed. Of these, we determined that only 246 would require `open` annotations in our system to preserve the current semantics of the standard library. This is a small fraction (less than 3%) or the methods in the library, suggesting that open annotations would be infrequently needed in practice.

In principle, it is possible that open annotations would not be needed on a codebase because that codebase was not making use of inheritance in any case. In fact, however, we found 1394 of the methods in the standard library are actually overridden, indicating substantial though not ubiquitous use of inheritance. The 246 `open` methods still make up less than 18% of the methods that were overridden.

The evidence that few `open` annotations are needed in practice supports the utility of Selective Open Recursion. Calling patterns within a class can be easily changed if few of that class's methods are open. In order to support correct usage of inheritance it is important that the ways in which open methods are called are documented [8, 12, 11], and so having fewer open methods lessens the documentation burden on implementors.

Optimization Potential. Since having few "open" annotations makes it easier for developers to reason about correctness of changes to a class, it is natural to expect that it might aid in automated reasoning—such as for compiler optimizations—as well. We tested this hypothesis on the same part of the Java library by looking at the potential for method inlining. We found that the library contains 22339 method calls, of which 6852 were self-calls. Only 716 of these self-calls were to open methods, meaning that they need to be dynamically dispatched. The remaining 6136 calls could potentially be inlined in a system with Selective Open Recursion. In standard Java, however, it would be unsafe in general to inline these calls, as Java treats all methods as implicitly open.

This data indicates that Selective Open Recursion allows 27% of all method calls in the libraries analyzed to be inlined. It is possible that some of these calls could already have been inlined because the target method is `private` or `final`, however. We are currently working to gather data that will tell us the true increase in optimization potential.

A whole program analysis could catch many of the optimization opportunities that Selective Open Recursion does, simply by observing that a particular program does not use all of the open recursion that Java supports. Whole program optimization of Java programs is complicated, however, because many programs load code dynamically, and this code could invalidate optimizations such as inlining. Because our Selective Open Recursion proposal changes the semantics of dispatch so that subclasses cannot intercept non-open self calls, it enables optimizations like inlining without the need for whole-program information.

4. Related Work

A significant body of related research focuses on documenting the dependencies between methods in a *specialization interface*. Kiczales and Lamping proposed that a method should document which methods it depends on, so that subclasses can make accurate assumptions about the superclass implementation [8]. Steyaert et al. propose a similar approach in a more formal setting [12]. Ruby and Leavens suggest documenting method call dependencies as part of a broader focus on modular reasoning in the presence of inheritance [11]. They also document a number of design guidelines that are applicable to the setting of Selective Open Recursion.

A common weakness of the "dependency documentation"

approaches described above is that they solve the fragile base class problem not by hiding implementation details, but rather by exposing them. Since the calling patterns of a class are part of the subclassing interface—and since subclasses may depend on them—making significant changes to the implementation of the class become impossible. Steyaert et al. acknowledge this and suggest documenting only the "important method calls," but the fragile base class problem can still occur unless unimportant method calls are hidden from subclasses using a technique like ours. Our work requires that calling patterns be maintained for calls to `open` methods, but does not impose this requirement for non-open methods, allowing a much wider range of implementation changes.

Bloch, Szyperski, and others suggest using forwarding in place of inheritance as a way of avoiding the fragile base class problem [2, 13]. However, as Szyperski notes, not all uses of inheritance can be replaced by forwarding because open recursion is sometimes needed [13]. Selective Open Recursion provides a middle ground between inheritance and forwarding, providing open recursion when it is needed but the more modular forwarding semantics where it is not.

Mikhajlov and Sekerinski consider a number of different ways in which an incorrect use of inheritance can break a refinement relationship between a class and its subclasses [9]. They prove a flexibility theorem showing that under certain conditions, when a superclass C is replaced with a new implementation D, then C's subclasses still implement refinements of the original implementation C. Their results, however, do not appear to guarantee that the semantics of C's subclasses are unaffected by the new implementation D, which is the contribution of our work.

Our use of static dispatch for calls on `this` is related to the *freeze* operator provided by module systems such as Jigsaw [4]. The freeze operation statically binds internal uses of a module declaration, while allowing module extensions to override external uses of that declaration. The freeze operator, however, has not been previously proposed as a solution to the fragile base class problem, nor (to our knowledge) has it previously been integrated into an object-oriented language implementation.

Some languages, including C++, provide a way to statically call a particular implementation of a method [5]. While this technique can be used as an implementation strategy for our proposal, we believe it is cleaner to associate "open-ness" with the method that is called rather than the call site, as discussed earlier.

Our solution to the fragile base class problem was inspired by our earlier work on a related modularity problem in aspect-oriented programming [1]. Just as a `CountingSet` subclass of `Set` can observe whether `addAll` is implemented in terms of `add`, a `Counting` aspect can be defined that uses advice to make the same observation. Our solution there was to prohibit aspects from advising internal calls within a class or module—just as we solve the fragile base class problem by using static dispatch to prevent subclasses from intercepting implementation-dependent calls in their superclass. In the aspect-oriented setting, we allow modules to export pointcuts that act as disciplined extension points, similar to `open` methods.

Relative to previous work, ours is the first to address the fragile base class problem by distinguishing methods for which open recursion is needed from methods for which it is not.

5. Conclusion

This paper argued that the fragile base class problem occurs because current object-oriented languages do not distinguish internal method calls that are invoked for mere convenience from those that are invoked as explicit extension points for subclasses. We proposed to make this distinction explicit by labeling as `open` those methods to which open recursion should apply. Our results mean that object-oriented component library designers can freely change more aspects of a library's implementation without the danger of breaking subclass code.

6. Acknowledgments

We thank Craig Chambers, Donna Malayeri, Todd Millstein, Frank Pfenning, and the anonymous reviewers for their feedback on earlier drafts of this material.

7. References

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

[2] J. Bloch. *Effective Java*. Addison-Wesley, Reading, Massachusetts, 2001.

[3] B. Bokowski and A. Spiegel. Barat–A Front-End for Java. Freie Universitt Berlin Technical Report B-98-09, 1998.

[4] G. Bracha. The Programming Language Jigsaw: Mixins, Modularity and Multiple Inheritance. Ph.D. Thesis, Dept. of Computer Science, University of Utah, 1992.

[5] M. A. Ellis and B. Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, Reading, Massachusetts, May 1990.

[6] E. Ernst. Family Polymorphism. In *European Conference on Object-Oriented Programming*, June 2001.

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[8] G. Kiczales and J. Lamping. Issues in the Design and Documentation of Class Libraries. In *Object-Oriented Programming Systems, Languages, and Applications*, 1992.

[9] L. Mikhajlov and E. Sekerinski. A Study of the Fragile Base Class Problem. In *European Conference on Object-Oriented Programming*, 1998.

[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] C. Ruby and G. T. Leavens. Safely Creating Correct Subclasses without Seeing Superclass Code. In *Object-Oriented Programming Systems, Languages, and Applications*, October 2000.

[12] P. Steyaert, C. Lucas, K. Mens, and T. D'Hondt. Reuse Contracts: Managing the Evolution of Reusable Assets. In *Object-Oriented Programming Systems, Languages, and Applications*, October 1996.

[13] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. ACM Press and Addison-Wesley, New York, NY, 1998.