

# Accommodating Evolution in AspectJ

Eric Wohlstadter, Aaron Keen, Stoney Jackson, and Premkumar Devanbu  
{wohlstad, keen, jacksoni, devanbu}@cs.ucdavis.edu

## 1 Introduction

The intent of Aspect-Oriented Programming is to encapsulate, reuse, and maintain concerns that crosscut modularization constructs provided in today's programming languages. As AOP solutions find their way into mainstream software development, complex systems, and libraries designed with aspects will eventually be prevalent. This new code-base will become the legacy code of the future and will be subject to the same evolutionary demands as today's code. Designers of tools and languages supporting AOP must be mindful of evolutionary pressures.

Aspect-oriented programming systems based on program transformations [BP00] use powerful pattern-matching techniques to identify and apply crosscutting concerns in software. AspectJ<sup>1</sup> [AJ], specifically, uses join points as program locations where advice may be weaved. An aspect refers to both sets of join points, called pointcuts, and the associated advice. The advice in an aspect is inserted at all join points associated with that advice.

In many cases, it may be quite reasonable to define pointcuts applicable to a specific current state of a program, such as certain particular naming conventions, etc. Therein lurks the evolutionary problem. As the code base evolves, the pointcuts defined for the original code may capture more join points than originally intended (or even imagined) [GK01]. If the AOP system does not support the evolution of aspects, then it may be cumbersome to adjust the pointcuts or the applicable advice. This issue is addressed in AspectJ through such language features as *abstract aspects*, *introduction*, and *aspect domination*. This paper evaluates these features and proposes a new, complementary feature.

## 2 Reuse in AspectJ

During the evolution of object-oriented software some of the functional and non-functional properties of the overall system may crosscut traditional module boundaries. Aspect-Oriented Programming can support reuse by allowing separate definition of aspects, which are subsequently woven into the module system. This non-invasive approach to program evolution provides desirable separation between the system's core functionality and added aspects.

Applying aspect-oriented design in a concrete implementation requires maintenance and evolution of the AOP artifacts such as aspects, pointcuts, and advice. In some situations, it may be advantageous to treat these AOP artifacts, themselves, as components and re-assemble them in different ways. AspectJ provides several mechanisms to enable the reuse of AOP constructs.

### 2.1 Abstract Aspects

In AspectJ, a programmer can declare an aspect definition to be abstract. This allows the definition of concrete advice that can be applied at abstract pointcuts. Abstract aspects do not, themselves, affect a program; their advice is only woven in when other aspects extend them to define a concrete aspects. One idea is to use abstract aspects as library code with abstract pointcuts as placeholders for the classes, methods, or fields that the aspect code could be woven into. When a programmer wants to use the library in their system they simply need to define concrete pointcuts that refer to parts of their system. Since the aspect does not make a commitment to a specific pointcut (or even a specific namespace), it can be flexibly re-used in

---

<sup>1</sup> This paper refer specifically to syntax and concepts from AspectJ 0.8beta4.

different systems. An example of this can be seen in the Tracing aspect of the AspectJ distribution [AJ]. AspectJ also allows abstract aspects to provide concrete pointcuts that can be overridden.

We now present a simple example to show why abstract aspects do not provide the necessary expressiveness to always provide non-invasive system evolution. Consider a system that starts out with one base class and several subclasses. A programmer writes an aspect to trace all method receptions to objects in the system. Now consider a newly introduced subclass, for which we would like to provide different tracing behavior from that of the rest of the system. For such a small example, one might just edit the original aspect to produce the desired behavior. However, directly modifying the existing code-base is not a scalable approach.

With that in mind we can attempt a reuse strategy by making the existing aspect abstract and extending from it (Figure 1). The sub-aspect shown overrides the previous pointcut but refers to it in the definition. This is to preserve the previous system behavior and only affect the new subclass. Our solution seems reasonable until a second subclass is added that also needs specific tracing behavior.

```
abstract aspect Tracing {
    pointcut tracePoints() : receptions(* *(..));
    before () : tracePoints() {
        //default tracing behavior }
}

aspect SubClass1Tracing extends Tracing {
    //The attempt to preserve default tracing interferes with other Tracing
    sub-aspects
    pointcut tracePoints() : Tracing.tracePoints() &&
!subClass1TracePoints();
    pointcut subClass1TracePoints() : instanceof(SubClass1) && receptions(*
*(..));
    before () : subClass1TracePoints() {
        //overridden tracing behavior
    }
}
```

**Figure 1**

One option is to make the sub-aspect abstract and continue to extend it. Doing so introduces undesired coupling between the two sub-aspect entities as well as further complicating the pointcut definitions. The extra coupling would prevent the addition and removal of the sub-aspect behaviors independently. Another approach is to add a second sub-aspect following the same idea as before. Unfortunately, these two sub-aspects can interfere with one another, e.g., canceling each other's attempt to remove advice from the base abstract aspect. The end result is that the default tracing gets applied twice to the base class and the two new subclasses get covered with the undesired default tracing.

## 2.2 AspectJ Introduction

There are times when the separation of a crosscutting concern can be implemented by injecting new fields or methods into existing types. This process is referred to as introduction and can significantly modify a type hierarchy [AJ]. It is even possible to add superclasses and interfaces to a class. A programmer can then separate different role/view behavior of a particular class into aspects and then selectively apply them, as needed, by adding an interface that represents that role. This is done in the AspectJ distribution to separate the Cloneable, Hashable, and Comparable roles of the Point class into different aspects. Introduction takes advantage of

dynamic dispatch in Java, allowing the programmer to inject methods in a base class and specialize these methods in subclasses to introduce overriding behavior. We will use this method to solve the problem of the previous section [AJU].

A new solution is presented in Figure 2. This solution relies on the AspectJ feature of pointcut arguments. Pointcut arguments allow advice to access state in the execution of a program available at a particular join point. The exact mechanism is described further in the AspectJ documentation. Advice in this example uses pointcut arguments to gain access to the currently executing object at a method reception point (the *this* argument). The advice then forwards the responsibility of determining its behavior to a helper function (`traceEntry`) introduced into the object. All of the classes in the framework will now exhibit the tracing behavior of the base class. To provide different tracing functionality for our new subclasses we define two aspects that introduce overriding methods. Notice that the new aspects do not interfere with one another, do not depend on each other, and definition of the sub-aspect is non-invasive on the tracing aspect.

Although introduction works well in this example, it does not provide a general mechanism for overriding advice. The solution given relies on the subclassing relationship for the different parts of the program that should exhibit varying advice. It has been shown that concerns may crosscut many different parts of a program's structure [KKH+01] including the class hierarchy, so a more powerful overriding mechanism is necessary. Section 3 presents an example that crosscuts the class hierarchy and a mechanism to provide non-invasive aspect evolution.

```
aspect Tracing {
    pointcut tracePoints(BaseClass x) : instanceof(x) && receptions(* *(..))
    && !receptions(void traceEntry());
    before(BaseClass x) : tracePoints(x) {
        x.traceEntry();
    }
    public void BaseClass.traceEntry() {
        //default trace behavior
    }
}

aspect SubClass1Tracing {
    public void SubClass1.traceEntry() {
        //overridden trace behavior
    }
}
```

**Figure 2**

### 2.3 Dominates

The aspects in a given environment cannot always be composed in an orthogonal fashion. The design of AspectJ provides the *dominates* keyword to give the programmer control over the order in which advice from aspects is applied. For instance, a security aspect may be required to dominate a logging aspect to prevent logging of behavior that is denied through some security mechanism. This paper identifies four ways for a dominating aspect to affect the behavior of aspects that it dominates:

1. The dominating aspect may throw an exception before the aspects it dominates.
2. The dominating aspect may fail to call `proceed()` in an around advice.
3. The dominating aspect may pass modified parameters to `proceed()` changing the values of a dominated advice's pointcut parameters.
4. The dominating aspect may modify some shared global state.

Since advice at a join point must be ordered in some linear fashion, in addition to the `dominates` keyword there are default rules for determining ordering of advice. As a consequence, an aspect cannot use methods (1) or (2) to selectively circumvent the advice from specific aspects. It may, as a result of default ordering, unknowingly skip the advice from a completely orthogonal aspect. Using methods (3) or (4) a programmer could devise a scheme for aspects to communicate with one another in order to provide overriding behavior. Such a scheme would pollute the bodies of the original advice and not clearly express the programmer's intention.

### 3 Introducing the `overrides` Keyword

In order to make explicit the cases where one aspect needs to circumvent advice from specified aspects without changing the rest of the system, we propose the use of an overriding clause. The overriding clause allows the programmer to specify a list of aspects that are overridden at a specific pointcut. The next example helps motivate the language modification and explain its use.

#### 3.1 Example Using `overrides`

```
aspect BeanTracing {
    pointcut beanPoints() : receptions(* get*()) || receptions(void
set*(..));
    before () : beanPoints() {
        //default bean tracing
    }
}

aspect SpecificBeanTracing dominates BeanTracing {
    pointcut specificPoints () : instanceof(SpecificBean) &&
BeanTracing.beanPoints();
    before () : specificPoints () {
        //overriding tracing
    } overrides BeanTracing;
}
```

**Figure 3**

Suppose, as before, that we have a framework of classes that we would like to trace. However, we now consider just JavaBeans classes, sans any subclass relationship. Using pointcuts to identify the join points of the bean `get` and `set` methods, the aspect is able to instrument advice for the tracing of beans. When a new bean is added that needs alternative tracing behavior, it may be desirable to reuse our existing aspect without modifying it. This adaptation of the existing system can be codified in an additional aspect that contains advice that overrides the advice from a named list of aspects. This allows additional adaptations to be made to the system by other aspects that do not interfere with one another, as in the case of abstract aspects.

#### 3.2 Design considerations

The previous section showed how the evolution and maintenance of aspect-oriented systems could be aided by the concept of aspect overriding. This section sketches some initial language semantics and design guidelines for the `overrides` keyword.

Section 2.3 pointed out how dominating aspects have a large amount of control over the aspects they dominate. In keeping with this precedent, an aspect is only allowed to list aspects that it dominates in overriding clauses. This prevents the possibility of two aspects overriding

each other at different join points. We haven't yet come across an example where this would be desired. Fortunately, it also precludes the case of advices that mutually overrides each other<sup>2</sup>.

It is important to note that the overriding clause does not name any specific advice to override. All advice from an overridden aspect is removed from join points where the overridden advice overlaps. Since advice is not named in AspectJ it is not possible to pinpoint certain advice to override. If pointcuts are used as the target for overriding clauses, then anonymous pointcuts [GK01] cannot be overridden. The design of the adaptation mechanism we propose is based on minimizing assumptions placed on adapted code and limiting the number of modifications to the current AspectJ language.

The replacement of new advice over the overridden advice can be done statically. This allows for IDE support to aid in the process of evolution by identifying overlapping join points. An alternative is to allow for specific aspects to be named dynamically at the point of a call to proceed. Although a dynamic mechanism is more powerful, we do not yet take a position as to whether or not such power is warranted.

### 3.3 Semantics of *Overrides*

The meaning of *overrides* can formally be defined in terms of join point sets. Consider the general aspect definition

```
aspect A dominates A1, A2, ..., An {
    pointcut p():...
    before(): p() { ... } overrides A1, A2, ..., An;
}
```

Then for each advice  $a$  in  $A1, A2, \dots, An$

$$pointcut'(a) = pointcut(a) \setminus p$$

where  $pointcut'(a)$  and  $pointcut(a)$  are, respectively, the new and old join point sets to which advice  $a$  is applied. Other *overrides* clauses that also refer to the aspect of  $a$  modify the new join point set  $pointcut'(a)$ .

In English, when the *overrides* keyword appears after some advice defined on a join point set  $P$ ,  $P$  is masked out from the join point sets of all advice in all aspects that appear in the *overrides* clause.

From this perspective, the *overrides* clause allows the overriding aspect to modify the join point sets that the advice of the overridden aspect are defined on. *Overrides* is, therefore, a predefined set manipulation. This suggests that there is a more general set manipulation mechanism that can be used to change the point cuts of old aspects when new aspects are written. Other set operations could be employed to give new aspects more control of their predecessors. We are exploring the ramifications of this more flexible mechanism.

## 4 Related Work

Aspect components [LLM99] aim to provide reuse in AOP software by dividing it into independent pluggable entities. Components are hooked together through the use of a visual builder tool or connection language. By moving connection code out of any specific components an extra amount of flexibility is gained. AspectJ, however, provides semantics similar to Java

---

<sup>2</sup> We are assuming that two aspects should not mutually dominate each other. However, [AspectJ 0.8beta4](#) does not prevent this.

where external connection languages are not used. Our approach is to follow the current design as closely as possible.

Work on feature interaction tries to solve the problem of conflicting views of object functionality. Prehofer [Preh97] suggests an approach where features can be stacked by writing lifting functions between two features. Overriding could be used to support feature composition in AspectJ by writing specialized code for join points where features conflict. Bergenti [BP00] identifies Composition Filters as an alternative to weaving for aspect-oriented programming. By changing the meta-object protocol at runtime, CF provides flexibility for dynamic aspect evolution. Java only provides introspective reflection so object model changes are not possible. Our work relates to the current status of AspectJ as a static program transformer. Future work is planned to add more dynamic features to AspectJ [AJ].

An aspect system design approach using pointcut interfaces and abstract pointcuts [GK01] produces code that is less prone to the adaptability problems described in this paper. However, it is not always possible to structure pointcuts in such a way to provide hooks for program evolution.

## 5 Conclusion

Overriding advice is a powerful tool to help programmers with the evolution and adaptation of aspect-oriented systems. In this paper, we consider a situation that requires aspects to evolve, and consider the evolution tools (abstract aspects and introduction) currently in AspectJ. We find that abstract aspects lead to undesirable coupling or interference between aspects; we also find that introduction can only handle inheritance hierarchy evolution. We propose a complementary approach, *overriding*, that affords more flexibility. As with most tools, careful design considerations should be followed to determine the right usage of overriding.

## Reference

[AGS+00] Brian de Alwis, Stephan Gudmundson, Greg Smolyn, and Gregor Kiczales. Coding Issues in AspectJ. In *Proceeding of Workshop on Advanced Separation of Concerns, ECOOP 2000*.

[AJ] AspectJ Homepage. <http://www.aspectj.org>.

[AJU] Gregor Kiczales. [users@aspectj.org](mailto:users@aspectj.org) archive messages. <http://aspectj.org/pipermail/users/>.

[BP00] Federico Bergenti and Agostino Poggi. Aspect Views as a Means to Promote Reuse in Aspect-Oriented Languages. *ECOOP Workshop on Aspects and Dimensions of Concerns, Cannes-Sophia Antipolis, France, 2000*.

[CKA01] Pascal Costanza, Gunter Kniesel, and Michael Austermann. Independent Extensibility for Aspect-Oriented Systems. In *Proceedings of Workshop on Advanced Separation of Concerns in Object-Oriented Systems, ECOOP 2001*.

[GK01] Stephan Gudmundson, Gregor Kiczales. Addressing Practical Software Development Issues in AspectJ with a Pointcut Interface. In *Proceedings of Workshop on Advanced Separation of Concerns in Object-Oriented Systems, ECOOP 2001*.

[KHH+01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm and William G. Griswald. An Overview of AspectJ. In *Proceedings of ECOOP 2001*.

[LLM99] Karl Lieberherr and David Lorenz and Mira Mezini, Programming with Aspectual Components . *Technical Report*, NU-CCS-99-01, March 1999.

[Preh97] Christian Prehofer. Feature-Oriented Programming: A Fresh Look at Objects. In *Proceedings of ECOOP97*.