# Aspects and Polymorphism in AspectJ

Erik Ernst
Dept. of Computer Science,
University of Aarhus
Åbogade 34, DK-8200 Århus N
Denmark
eernst@daimi.au.dk

David H. Lorenz*
College of Computer & Information Science,
Northeastern University
360 Huntington Avenue 161 CN
Boston, Massachusetts 02115 USA
lorenz@ccs.neu.edu

## ABSTRACT

There are two important points of view on inclusion or subtype polymorphism in object-oriented programs, namely polymorphic access and dynamic dispatch. These features are essential for object-oriented programming, and it is worthwhile to consider whether they are supported in aspect-oriented programming (AOP). In AOP, pieces of crosscutting behavior are extracted from the base code and localized in aspects, losing as a result their polymorphic capabilities while introducing new and unexplored issues. In this paper, we explore what kinds of polymorphism AOP languages should support, using AspectJ as the basis for the presentation. The results are not exclusive to AspectJ—aspectual polymorphism may make aspects in any comparable AOSD language more expressive and reusable across programs, while preserving safety.

## 1. INTRODUCTION

There are two important points of view on inclusion or subtype polymorphism in object-oriented programs, namely polymorphic access and dynamic dispatch (Figure 1).

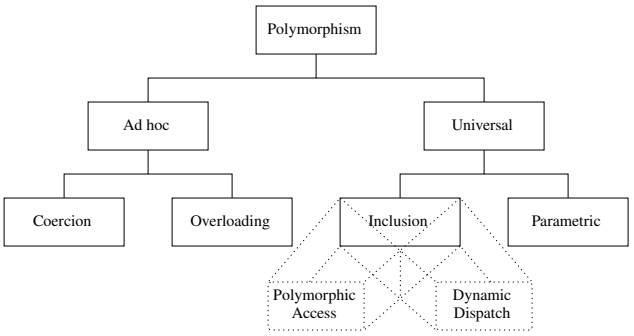


Figure 1: Polymorphism

*Supported in part by the National Science Foundation (NSF) under Grant No. CCR-0098643 and CCR-0204432, and by the Institute for Complex Scientific Software at Northeastern University.



AOSD 2003 Boston, MA USA
© ACM 2003 1-58113-660-9/03/002...$5.00

- Polymorphic access: A name denotes an object whose dynamic type is known only by an upper bound. For example, the predefined reference `self` in a method in a class `C` denotes an instance of `C` or some subclass of `C`. The essence is that the *same* code can operate on differently shaped objects because they all satisfy certain constraints (possibly made explicit in a statically declared type).
- Dynamic dispatch: When accessing an object polymorphically, a message-send may be resolved by late binding, i.e., it may call different method implementations for different dynamic receiver types. The essence is that the meaning of a name (the method selector) is determined by dynamically selecting the "best" definition from a set of *different* available definitions, namely all the method implementations in the dynamic class of the receiver and its superclasses.

In short, inclusion polymorphism is characterized by invariance in an "early" phase (e.g., a call site contains one particular message send `foo`; or an object $O$ contains a known set of features), and by dynamic variance in a "late" phase (e.g., considering all the possible implementations of `foo` for that call site; or all the possible additional features of $O$). Late binding of methods provides the flexibility of executing different behaviors at the same call site, while preserving the safety of always executing a behavior that is appropriate for the actual receiver object. For object access, we get the flexibility of being able to add new features, and the safety guarantee that known features will indeed be present dynamically. In this paper, we seek a similar kind of safe flexibility by means of polymorphism in connection with aspects.

Inclusion polymorphism is a kind of universal polymorphism, which means that the set of possible forms in the late phase is open-ended [7]. In contrast, in ad-hoc polymorphism, the set of possible forms is determined in the early phase, and one specific form is selected. For example, an overloaded operation also simultaneously denotes more than one function [21]. However, unlike dynamic dispatch, overloading is resolved at compile time and hence is a form of ad-hoc polymorphism.

In AspectJ [27, 25], object-oriented polymorphism is supported in the sense that the Java [3] programming language supports polymorphism, and AspectJ is a superset of Java. Similar connections would typically exist with other AOSD approaches. The support for polymorphism in connection with aspects is essentially ad-hoc.

For example, one can write a single implementation of an aspect, which displays polymorphic behavior in different join points. Con-

sider the abstract aspect `SimpleTracing` in Listing 1. [1]

Listing 1: SimpleTracing.java

```
package aspect;[2]
import org.aspectj.lang.JoinPoint;
abstract aspect SimpleTracing {
  abstract pointcut tracePoints();
  before(): !within(aspect..*) && tracePoints() {
    trace(thisJoinPoint);
  }
  protected abstract void trace(JoinPoint jp);
}
```

The **`thisJoinPoint`** pseudo variable (and the related pseudo variable **`thisJoinPointStaticPart`**) is bound to an object representing the current join point, thus providing access to dynamic (respectively static) information about that join point. This object can be queried with methods such as

```
thisJoinPoint.getThis()
```

or

```
thisJoinPointStaticPart.getSignature()
  .getDeclaringType().getName()
```

which will give different outcomes depending on the actual join point. This is because the pointcut (and hence the aspect) is abstract, which means that an actual aspect using the given before-advice would be a subaspect of `SimpleTracing` in which the pointcut `tracePoints` has been made concrete, and the method `trace` has been implemented. Hence, an execution of the before-advice would happen in context of an aspect instance whose type is only known by an upper bound, and **`thisJoinPoint`** is only known to be a join point.

Moreover, an aspect can define abstract or concrete methods for subaspects to redefine or use, e.g.,

Listing 2: ExecutingObjectTracing.java

```
abstract aspect ExecutingObjectTracing
  extends SimpleTracing {
  protected void trace(JoinPoint jp) {
    System.out.println("Executing object: "
      + jp.getThis());
  }
}
```

Finally, an advice and a pointcut may take arguments such as the receiver object, e.g.,

```
  after(Point p, int nval): target(p)
    && args(nval)
    && call(void Point.setX(int)) {
    System.out.println("x in " + p
      + " is now set to " + nval + ".");
  }
```

[1]To avoid explaining AspectJ basics we will assume knowledge of [25] and use some examples similar to the ones in that paper. In specific discussions about properties of AspectJ we are referring to version 1.0.6

[2]For the rest of the paper we will assume that all aspects are defined in this package.

and this means that the advice code will polymorphically access the receiver, `p`, of the intercepted invocation of `setX`.

However, all these examples of polymorphism are "inherited" from the base language. Imagine a base language without polymorphism, and they would all evaporate.

Even the notion of an abstract pointcut and associated definitions of concrete pointcuts is not truly polymorphic in the same sense as object-oriented methods. It is true that advice code such as the invocation of `trace` in Listing 1 is executed depending on a pointcut whose concrete definition is not known in `SimpleTracing`. In other words, it is not known in `SimpleTracing` when this advice code will be executed. But for each location in the program where this advice may actually be enabled, it will be considered in context of a concrete subaspect of `SimpleTracing` for which `tracePoints` can be checked statically—e.g.,

Listing 3: PointTracing.java

```
aspect PointTracing
  extends ExecutingObjectTracing {
  pointcut tracePoints():
    execution(* Point+.setX(..));
}
```

—"is this an invocation of `C.setX` where `C` is `Point` or a subclass thereof?" Note that even with dynamic pointcuts, e.g., **`cflow`**, it is known statically that it is a **`cflow`** pointcut, with a known specification, and the dynamic part is only determining whether a given expression is true or not.

# 2. AD-HOC POLYMORPHISM IN ASPECTJ

There are several points worth noting about polymorphism in AspectJ.

## 2.1 Extending an aspect

It is possible to extend an abstract aspect, but not a concrete one. The concrete aspect `PointTracing` (Listing 3) extends the abstract aspect `ExecutingObjectTracing` (Listing 2). However, `PointTracing` cannot be further extended. This is actually a minor limitation that can be explained as an enforcement of the "abstract superclass rule" [20], i.e., the rule that only the leaves of an inheritance hierarchy can be concrete. Every aspect hierarchy can be transformed into one in which all super-aspects are abstract, although this is not typically required by programming languages.[3]

## 2.2 Overriding a pointcut designator

It is possible to redefine concrete pointcuts in the subaspect. For example, the abstract aspect `ReturnValueTracing` in Listing 4,

Listing 4: ReturnValueTracing.java

```
abstract public aspect ReturnValueTracing
  extends ExecutingObjectTracing {
  pointcut tracePoints(): call(*.new (..));
  after() returning (Object o): tracePoints() {
    System.out.println("Return: " + o);
  }
}
```

[3]For example, the Sather programming language separates classes and types strictly, and supports only the 'insert-new-super' for types. However, the treatment of classes in Sather does not enforce a strict abstract superclass rule. Even in Sather it *is* possible to inherit from a concrete class.

defines a concrete pointcut `tracePoints`, which designates constructor calls. Its after-returning advice assumes the join points are constructor calls; and the concrete aspect `ConstructorTracing`,

Listing 5: ConstructorTracing.java (1)

```
public aspect ConstructorTracing
  extends ReturnValueTracing {}
```

would trace newly constructed objects. However, it is an error to put after-returning advice on a join point that does not return a value or returns a value of an incorrect type. Had the concrete aspect `ConstructorTracing` mistakenly redefined the pointcut `tracePoints` to be constructor executions rather than calls, e.g.,

Listing 6: ConstructorTracing.java (2)

```
public aspect ConstructorTracing [4]
  extends ReturnValueTracing {
  pointcut tracePoints(): execution(*.new (..));
}
```

then there wouldn't have been a return value. Thus, by overriding a pointcut designator, a subaspect might break inherited advice code. Currently in AspectJ, the compiler will complain that the return value will bind to null, but will ignore a type mismatch.

## 2.3 Overriding an advice

It is impossible to redefine advice. It is also impossible to advise an aspect's advice in order to modify an existing advice, at least not yet. The before-advice defined in `SimpleTracing` (Listing 1) is frozen. Attempting to redefine the before-advice in a subaspect `DoubleTracing` (Listing 7), for example,

Listing 7: DoubleTracing.java

```
aspect DoubleTracing
  extends ExecutingObjectTracing {
  pointcut tracePoints(): execution(* *(..));
  before(): !within(aspect..*) && tracePoints() {
    trace(thisJoinPoint);
  }
}
```

will (perhaps somewhat unexpectedly) result in the method `trace` being called twice for each join point. The two before-advice declarations in Listings 1 and 7 are identical, but in AspectJ they are considered unrelated and will be applied independently.

These examples indicate the sense in which aspects in AspectJ are not polymorphic. More can be learned by compiling the code with the `-preprocess` option. The abstract aspect `SimpleTracing` in Listing 1 is preprocessed into the abstract class `SimpleTracing` (Listing 8). Note that the method is final.

The concrete aspect `PointTracing` in Listing 3 is preprocessed into the concrete class `PointTracing` (Listing 9). Note that this is a non-polymorphic implementation of a Singleton design pattern [12].

[4]Compiling this aspect generates the warning message: "on target constructor-execution( SimpleTracing()) return value will bind to null, use 'this' pointcut to bind executing object... behavior change to fix compiler bug in 1.0.3 and earlier (warning)..."

Listing 8: Preprocessed SimpleTracing class

```
abstract class SimpleTracing {
  public final
  void before0$ajc(JoinPoint thisJoinPoint) {
    this.trace(thisJoinPoint);
  }
  protected abstract void trace(JoinPoint jp);
}
```

Listing 9: Preprocessed PointTracing class

```
public class PointTracing
  extends ExecutingObjectTracing {
  public static PointTracing aspectInstance;
  public static PointTracing aspectOf() {
    return PointTracing.aspectInstance;
  }
  static {
    PointTracing.aspectInstance =
      new PointTracing();
  }
}
```

## 2.4 Run-time pointcut designators and run-time aspect instances

To complete the picture we must also consider run-time pointcut designators such as **`this`**, **`target`**, **`cflow`**, **`cflowbelow`**, **`args`**, and **`if`**, as well as run-time aspect instance specifications such as **`perthis`**, **`pertarget`**, **`percflow`**, and **`percflowbelow`**.

Run-time pointcuts would seem to introduce the dynamism that is otherwise missing in the invocation of advice code, but they are in fact only capable of enabling or disabling the given advice. Since dynamic dispatch is concerned with a dynamic selection of *which* piece of code to execute, not just *whether* to invoke a given piece of code or not, even these kinds of pointcuts do not introduce full polymorphism into AspectJ, apart from the situations where the polymorphism is inherited from the base language.

Aspect instances also introduce a level of dynamism. However, selecting which aspect instance to use is not the same as dynamically selecting which piece of code to execute: the choice is *whether* or not to execute a piece of code that is fully determined by the aspect instance, not *which* piece of code to execute, based on the type of the object to which the advice is applied. Non-polymorphic aspectual code is a step back from OOP. In this paper, we revisit the AOP model and discuss the potential for supporting aspectual polymorphism.

# 3. LATE BINDING OF ADVICE

An interesting extension is introducing polymorphism in connection with advice. The advice language element can be extended so that the advice which is possibly applied at a given point in the execution of the program is *chosen* dynamically from a set of applicable advice declarations, similarly to dynamic dispatch where one method implementation is chosen from a set of applicable ones. We use the term *late binding of advice* to denote such a mechanism.

Consider the `IncrTracking` aspect in Listing 10. Today, the two

[5]The classes such as `FigureElement` and `Point` are assumed to be defined as in [25].

Listing 10: IncrTracking.java

```
aspect IncrTracking [5] {
  after(FigureElement fe): target(fe)
    && call(void FigureElement.incrXY(int,int)) {
    System.out.println("IncrTracking: "
      + "Moving the figure element " + fe);
  }
  after(Point p): target(p)
    && call(void Point.incrXY(int,int)) {
    System.out.println("IncrTracking: "
      + "Moving the point at ("
      + p.getX() + "," + p.getY() + ")");
  }
}
```

advice declarations in `IncrTracking`, and indeed *any* two advice declarations, would be invoked independently of each other. For each point in the execution of the program where an advice might be enabled it is determined whether or not that advice should be invoked, without considering any other advice. In fact, putting the two advice declarations in two separate aspects (Listings 11 and 12) would give the same behavior as the code in Listing 10. They only differ in the identity and number of aspect instances, which is immaterial for stateless aspects, and the ordering of advice code execution, which can be controlled using aspect domination: [6]

Listing 11: IncrTrackingFigureElement.java

```
aspect IncrTrackingFigureElement {
  after(FigureElement fe): target(fe)
    && call(void FigureElement.incrXY(int,int)) {
    System.out.println("IncrTrackingFigureElement:"
      + " Moving the figure element " + fe);
  }
}
```

Listing 12: IncrTrackingPoint.java

```
aspect IncrTrackingPoint
  dominates IncrTrackingFigureElement {
  after(Point p): target(p)
    && call(void Point.incrXY(int,int)) {
    System.out.println("IncrTrackingPoint: "
      + "Moving the point at ("
      + p.getX() + "," + p.getY() + ")");
  }
}
```

As a result we might invoke zero, one, or two of the above advice declarations. This is true both when the answer is known statically, and also when a dynamic check is required.

To avoid the case of invoking *both* of the above advice declarations, it is possible to use around-advice without calling **`proceed`** (Listings 13 and 14). By doing so, the around-advice that is invoked first will prevent the other around-advice from being invoked. This is perhaps closer to late binding of advice, and can be used in a Visitor [12] design pattern style [29]. However, it is not quite the same as late binding, because it does not cover kinds of advice other than around-advice; it does not allow the base method to be invoked;

[6] An aspect $A$ may declare that the advice in it *dominates* the advice in some other aspect $B$. This only affects the order of execution when advice declarations from both $A$ and $B$ are enabled at the same point.

Listing 13: IncrTrackingFigureElement2.java

```
aspect IncrTrackingFigureElement2 {
  void around(FigureElement fe): target(fe)
    && call(void FigureElement.incrXY(int,int)) {
    System.out.println("IncrTrackingFigureElement2"
      + ": Moving the figure element " + fe);
    // do not invoke proceed(fe);
  }
}
```

Listing 14: IncrTrackingPoint2.java

```
aspect IncrTrackingPoint2
  dominates IncrTrackingFigureElement2 {
  void around(Point p): target(p)
    && call(void Point.incrXY(int,int)) {
    System.out.println("IncrTrackingPoint2: "
      + "Moving the point at ("
      + p.getX() + "," + p.getY() + ")");
    // do not invoke proceed(p);
  }
}
```

and it lets one advice override all other advice declarations including, e.g., an unrelated before-advice with different arguments. The latter demonstrates the need for advice grouping.

## 3.1 Advice grouping

In order to be able to talk about late binding of advice it is necessary to introduce some kind of device that allows us to establish a *group* of related advice declarations. It is not reasonable to assume that all advice declarations which happen to be enabled at the same point are logically related such that it makes sense to choose one and ignore all others. We therefore need to control advice grouping explicitly. We discuss how to do this in the next sections; for now we just assume that it has been done. After having determined that an advice group as such is enabled, we choose exactly one most specific advice and invoke it, ignoring all the others in the group (they are being *overridden*). Advice declarations outside the group are being processed independently. In Listing 10 we would choose to invoke the second advice for `p` being an instance of `Point` or a subclass thereof, and we would invoke the first advice for instances of `FigureElement` or a subclass thereof except `Point` and its subclasses.

Finally, invoking the most specific advice and ignoring all others in the group is just one possible semantics for late binding of advice. With reference to the CLOS [5, 24] method combination framework we could also envision *composing* an effective advice out of several applicable advice declarations in the group.

## 3.2 Congruent arguments

One possible way to establish a group of advice declarations is to compare argument lists. Any two advice declarations with congruent argument lists would be in the same group, for some suitable definition of 'congruent.' This would probably be a poor idea; just consider how hard it would be to express that two advice declarations should actually be independent of each other even though they might be enabled at the same program point and they do have congruent argument lists. In particular, it would be hard to avoid unintended grouping of advice from different aspects, possibly written

by programmers in different organizations.

## 3.3 Advice selectors

A better way is to give names to advice. This is similar to the mechanism commonly used with methods: When a given method implementation is selected during late binding, the set of potential implementations will all have the same name (and, in languages with static overloading, also 'sufficiently similar' types of arguments). So we might introduce syntax to give names to advice, and then use it as follows:

Listing 15: IncrTracking (1)

```
aspect IncrTracking {
  after trackIncr(FigureElement fe): target(fe)
    && call(void FigureElement.incrXY(int,int)) {
    System.out.println("IncrTracking: "
      + "Moving the figure element " + fe);
  }
  after trackIncr(Point p): target(p)
    && call(void Point.incrXY(int,int)) {
    System.out.println("IncrTracking: "
      + "Moving the point at ("
      + p.getX() + "," + p.getY() + ")");
  }
  after(Point p): call(void p.incrXY(int,int) {
    p.x++;
  }
}
```

Note that the two named advice declarations (named `trackIncr`) are in the same group, but the third (traditional, unnamed) advice is in a group of its own, and advice declarations with other names than `trackIncr` are of course also in different groups. With this rule we might have `trackIncr` advice in one or more aspects having an argument of type `FigureElement` or a subclass thereof.

## 3.4 Advice signature

In order to make the intended common properties explicit, it would probably be useful to introduce the notion of an advice group signature:

Listing 16: IncrTracking (2)

```
aspect IncrTracking {
  advicegroup trackIncr(FigureElement);
  after trackIncr(FigureElement fe):
  ...  // as in Listing 15
}
```

This declares explicitly that all advice declarations which belong to the `trackIncr` group must have an argument list congruent with '`(FigureElement)`'. It is necessary to consider the notion of congruence carefully, but as a first approximation we could just assume that the first argument must be the receiver where the given type must be a subtype of the one in the signature. A congruent argument list would then have a subtype in the first argument position, and it would have identical argument types in the remaining argument positions. There is a lot of room for expansion in this area, and an obvious extension would be to dispatch on several arguments, possibly modeled after CLOS, Dylan [33] or some other language where multiple dispatch is already available.

Another interesting extension would be to use predicate dispatch. Indeed, the Fred [30] AOP language, which is based on Chambers' predicate classes [8] and Ernst et al.'s predicate dispatch [11], exhibits a higher degree of polymorphism by unifying the method and advice concepts to a single concept of 'branch.' For this paper we just assume that there is *some* definition of congruence, and that there is *some* run-time criterion which will allow us to select a single advice from the given group as the most specific one, or to compose an effective advice from one or more of the applicable advice declarations.

## 3.5 No Need for Ordinary Late Binding

The notion of late binding of advice introduces genuine polymorphism into the aspect part of the language, and it even enables us to achieve the effect of late binding of methods in a base language that does *not* support late binding of methods. Assume that the base language always select method implementations statically, but supports late binding of advice. Now consider the following definitions:

Listing 17: AddIncrXY

```
aspect addIncrXY {
  after incrXY(FigureElement fe, int x, int y):
    target(fe)
    && args(x,y)
    && call(void FigureElement.incrXY(int,int)) {
    // implementation of incrXY for FigureElement
    ...
  }
  after incrXY(Point p, int x, int y):
    target(p)
    && args(x,y)
    && call(void Point.incrXY(int x, int y)) {
    // implementation of incrXY for Point
    ...
  }
  ... // implementations for other subclasses
}
```

Along with these definitions, we would have an empty `incrXY` method in the `FigureElement` class (and no other `incrXY` methods in any of the subclasses of `FigureElement`). This empty method is bound statically, but the support for late binding of advice ensures that the appropriate behavior for the actual class of the receiver is invoked after the execution of the empty method. The effect is the same as it would have been if the base language had supported late binding of ordinary methods, and we had used them.

This approach is generally applicable, hence late binding of ordinary methods is subsumed by late binding of advice.

# 4. ENVIRONMENTAL ADVICE

An aspect displays polymorphic behavior in different aspect instances, not just in different join points. Aspects are by default **`issingleton`**, but they may be declared **`perthis`**, **`pertarget`**, **`percflow`**, and **`percflowbelow`**.

The idea behind environmental advice is that the advice which is applied depends on the aspect instance that is dynamically available at the context of a given point in the execution of the program.

Consider the following `Percflow` aspect:

Listing 18: Percflow.java

```
abstract aspect Percflow extends SimpleTracing
  percflow(tracePoints()) {}
```

and two of its concrete subaspects, `PerLine` and `PerPoint`:

Listing 19: PerLine.java

```
aspect PerLine extends Percflow {
  pointcut tracePoints():
  call(void Line.incrXY(int,int));
  protected void trace(JoinPoint jp) {
  System.out.println("PerLine instance: "
     + this);
  }
}
```

Listing 20: PerPoint.java

```
aspect PerPoint extends Percflow {
  pointcut tracePoints():
  call(void Point.incrXY(int,int));
  protected void trace(JoinPoint jp) {
    System.out.println("PerPoint instance: "
     + this);
  }
}
```

At a particular point in the execution, there are now many aspect instances (stack based) to chose from. An advice can access the innermost aspect instance by the static method `aspectOf()`, for example,

```
PerPoint.aspectOf().trace(thisJoinPoint);
```

This allows polymorphic access to an aspect instance, (even without dynamic dispatch on advice). Unfortunately, AspectJ does not permit you to access the aspect object via the abstract aspect. Attempting to write:

```
Percflow.aspectOf().trace(thisJoinPoint);
```

would yield a compile time error (saying that no method named `aspectOf` defined in `Percflow`.) Hence, this kind of polymorphism is not available today in AspectJ.

Note that the lack of polymorphic support here cannot be worked around by a simple explicit switch statement. Writing:

```
if (PerPoint.hasAspect())
  PerPoint.aspectOf().trace(thisJoinPoint);
else if (PerLine.hasAspect())
  PerLine.aspectOf().trace(thisJoinPoint);
else
  System.out.println("no Percflow aspect");
```

won't necessarily select the innermost `Percflow` instance, because a `Line` can be nested within the control flow of a `Point`. One could also try to write:

Listing 21: PerFigureElement.java

```
aspect PerFigureElement extends SimpleTracing
   percflow(tracePoints()) {
  pointcut tracePoints():
  call(void Line.incrXY(int,int))
  || call(void Point.incrXY(int,int));
  protected void trace(JoinPoint jp) {
    System.out.println("PerFigureElement "
     + "instance: " + this);
  }
}
```

but then the polymorphic access is gone, for example,

```
PerFigureElement.aspectOf().trace(thisJoinPoint);
```

will always invoke the same `trace` method.

A mechanism which supports environmental advice should allow even more flexibility by late association of the aspect instance and the client trying to access it. Such a mechanism cannot rely on having the source code available for preprocessing. (Bytecode weaving, when available, will not solve this problem.[7]) Instead, advice should be subjected to environmental polymorphism [13] and aspects need to be looked up during the late phase. The client will thus be parameterized by the environment in which it happens to run. We use the term *environmental advice* to denote such a mechanism.

Schüpany et al. [32] mention aspectual polymorphism as an interesting research area to explore, which they borrowed from Minos [31]. They refer to an environmental advice mechanism in which changes in the aspect context during run-time affects the code of aspectized methods, similar to mechanisms of component deployment which allow a container to intercept the methods of its component [28]. We have used the term aspectual polymorphism independently in an early draft of this paper [10] in a broader sense: polymorphism in connection with aspects.

# 5. OTHER AOSD APPROACHES

In this paper we have focused on AspectJ, in order to be able to analyze the concrete language constructs of a widely known AOSD language in some detail. However, the concept of polymorphism deserves consideration also in connection with other AOSD approaches.

In the *composition filters* approach [34, 1, 35], it is possible to intercept message-sends and manipulate the receiver (i.e., delegate to another object), manipulate the selector (i.e., invoke a method with a different name), and manipulate the arguments. Moreover, the concept of *superposition* allows for non-invasive application of filters to multiple objects. This provides a very general framework, capable of expressing entities similar to AspectJ aspects, among other things. We believe that various kinds of polymorphism could be implemented by means of composition filters, but also that it would imply a manual encoding of, e.g., a late binding mechanism for advice. It is not clear to which extent such a mechanism could be expressed once and for all, as a reusable library, but it would be very interesting to see such a library if it were implemented.

AspectS [15, 16] is an AOSD framework with AspectJ-like concepts which is based on the Smalltalk [14] dialect Squeak [22, 26]. AspectS supports AOP by means of method interception and general Smalltalk meta-programming. It non-invasively extends the Squeak environment [18, 17]. Unlike AspectJ which does not support run-time weaving [4], AspectS' aspects can be installed and uninstalled dynamically. When an aspect is installed or dynamically deployed, AspectS replaces the designated methods with method wrappers [6], which act as around-advice. PerspectiveS [19] is a management facility on top of AspectS which provides layers of context-depended behavior that can be activated or deactivated dynamically.

[7]From the AspectJ FAQ: "Weaving into bytecodes at both compile and load-time will definitely be provided in a future release."

Similar to the situation with composition filters, we believe that the AspectS flexible framework could support a manual encoding of various kinds of polymorphism in connection with aspects. Again, expressing aspectual polymorphism as a facility on top of AspectS and PerspectiveS is a logical follow-up to this work.

# 6. CONCLUSION

Polymorphism in connection with aspects in AspectJ is generally inherited from the base language, and the aspectual polymorphism features that we have identified and discussed are completely resolved at compile time. By definition, this makes it ad-hoc polymorphism. Moreover, the fledgling support for aspectual polymorphism in AspectJ invites further developments because it has some limitations and irregularities. Based on this analysis, we explore what kinds of genuine, dynamic aspectual polymorphism AOP languages could support.

Aspectual polymorphism can make aspects more expressive and reusable across programs, yet keep them safe. In particular, late binding of advice would be at least as powerful as ordinary late binding of methods, in the sense that a base language without late-bound methods but with late-bound advice would be able to simulate late-bound methods by a simple, general technique. Advice subjected to an environmental polymorphism mechanism would be at least as powerful as the wrapper technology in component systems (e.g., BeanContext, EJB container, JINI container, and stateless web services) in the sense that it would allow separation of nonfunctional requirements and business logic concerns [9], and in that the behavior of the component is parameterized by the runtime environment in which it happens to run.

Inclusion polymorphism is key in OOP and environmental polymorphism is key in component technology. Aspectual and environmental polymorphism would likely play a critical role in many new and unexplored issues like AOP frameworks, componentized aspects, aspectual components, and more.

## Acknowledgment

We thank the anonymous reviewers for their valuable feedback.

# 7. 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*, number 791 in Lecture Notes in Computer Science, pages 152–184, Kaiserslautern, Germany, July 26-30 1994. Springer Verlag.

[2] AOSD 2002. *Proceedings of the $1^{st}$ International Conference on Aspect-Oriented Software Development*, Enschede, The Netherlands, Apr. 2002. ACM Press.

[3] K. Arnold and J. Gosling. *The Java Programming Language*. The Java Series. Addison–Wesley Publishing Company, 1996.

[4] J. Baker and W. Hsieh. Runtime aspect weaving through metaprogramming. In AOSD 2002 [2], pages 86–95.

[5] B. Bobrow, D. DeMichiel, R. Gabriel, S. Keene, G. Kiczales, and D. Moon. *Common Lisp Object System Specification*. Document 88-002R. X3J13, June 1988.

[6] J. Brant, B. Foote, R. E. Johnson, and D. Roberts. Wrappers to the rescue. In Jul [23], pages 396–417.

[7] L. Cardelli and P. Wegner. On understanding types, data abstractions, and polymorphism. *ACM Comput. Surv.*, 17(4):471–522, 1985.

[8] C. Chambers. Predicate classes. In O. Nierstrasz, editor, *Proceedings ECOOP'93*, LNCS 707, pages 268–296, Kaiserslautern, Germany, July 1993. Springer-Verlag.

[9] F. Duclos, J. Estublier, and P. Morat. Describing and using non functional aspects in component based applications. In AOSD 2002 [2], pages 65–75.

[10] E. Ernst and D. H. Lorenz. Aspectual polymorphism. Technical Report NU-CCS-01-09, College of Computer Science, Northeastern University, Boston, MA 02115, Oct. 2001.

[11] M. D. Ernst, C. Kaplan, and C. Chambers. Predicate dispatching: A unified theory of dispatch. In Jul [23], pages 186–211.

[12] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, USA, 1995.

[13] J. Gil and D. H. Lorenz. Environmental Acquisition – A new inheritance-like abstraction mechanism. In *Proceedings of the $11^{th}$ Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 214–231, San Jose, California, Oct. 6-10 1996. OOPSLA'96, ACM SIGPLAN Notices 31(10) Oct. 1996.

[14] A. Goldberg and D. Robson. *Smalltalk–80: The Language*. Addison-Wesley, Reading, MA, USA, 1989.

[15] R. Hirschfeld. Aspect-oriented programming with AspectS. In *Proceedings of the $3^{rd}$ International Conference Net.ObjectDays, NODe 2002*, pages 219–235, Erfurt, Germany, Oct. 7-10 2002.

[16] R. Hirschfeld. AspectS—aspect-oriented programming with Squeak. In M. Aksit, M. Mezini, and R. Unland, editors, *Architectures, Services, and Applications for a Networked World*, number 2591 in Lecture Notes in Computer Science. Springer Verlag, 2003.

[17] R. Hirschfeld and D. H. Lorenz. Non-invasive on-demand changes: Using change-sets as aspects. Unpublished, 2003.

[18] R. Hirschfeld and M. Wagner. Metalevel tool support in AspectS. In *Workshop on Tools for Aspect-Oriented Software Development, OOPSLA'02*, Seattle, Washington, 2002.

[19] R. Hirschfeld and M. Wagner. PerspectiveS – AspectS with context. In *Workshop on Engineering Context-Aware Object-Oriented Systems and Environments, OOPSLA'02*, Seattle, Washington, 2002.

[20] W. L. Hürsch. Should superclasses be abstract? In M. Tokoro and R. Pareschi, editors, *Proceedings of the $8^{th}$ European Conference on Object-Oriented Programming*, number 821 in Lecture Notes in Computer Science, pages 12–31, Bologna, Italy, July 4-8 1994. ECOOP'94, Springer Verlag.

[21] J. D. Ichbiah. Rationale for the design of the ada programming language. *Sigplan Notices*, 14(6), 1979. Part B (special issue).

[22] D. Ingalls, T. Kaehler, J. Maloney, S. Wallace, and A. Kay. Back to the future: The story of squeak, a practical Smalltalk written in itself. In *Proceedings of the $12^{th}$ Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 318–326, Atlanta, Georgia, Oct. 5-9 1997. OOPSLA'97, Addison-Wesley.

[23] E. Jul, editor. *Proceedings of the $12^{th}$ European Conference on Object-Oriented Programming*, number 1445 in Lecture Notes in Computer Science, Brussels, Belgium, July 20-24 1998. ECOOP'98, Springer Verlag.

[24] S. E. Keene. *Object-Oriented Programming in Common Lisp*. Addison-Wesley, Reading, MA, USA, 1989.

[25] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In J. L. Knudsen, editor, *Proceedings of the $15^{th}$ European Conference on Object-Oriented Programming*, number 2072 in Lecture Notes in Computer Science, pages 327–353, Budapest, Hungary, June 18-22 2001. ECOOP 2001, Springer Verlag.

[26] G. Korienek, T. Wrensch, and D. Dechow. *Squeak - A Quick Trip to ObjectLand*. Addison–Wesley Publishing Company, 2001.

[27] C. V. Lopes and G. Kiczales. Recent developments in AspectJ. In S. Demeyer and J. Bosch, editors, *Object-Oriented Technology. ECOOP'98 Workshop Reader*, number 1543 in Lecture Notes in Computer Science, pages 398–401. Workshop Proceedings, Brussels, Belgium, Springer Verlag, July 20-24 1998.

[28] M. Mezini, K. Ostermann, and R. Pichler. Component models and aspect-oriented programming. `http://www.st.informatik.tu-darmstadt.de/lehre/ws01/sctoo/materials/aj-aop.pdf`, 2001.

[29] M. Nordberg. Polymorphic advice and multimethod advice, Mar. 2002. AspectJ Users Mailing List. Thu, 7 Mar 2002 18:16:25 PST, and Sat, 9 Mar 2002 04:50:36 PST.

[30] D. Orleans. Incremental programming with extensible decisions. In AOSD 2002 [2].

[31] K. Ostermann and M. Mezini. Object creation aspects with flexible aspect deployment. `http://www.st.informatik.tu-darmstadt.de/database/publications/data/minos.pdf?id=68`, 2002.

[32] M. Schüpany, C. Schwanninger, and E. Wuchner. Aspect-oriented programming for .NET. In Y. Coady, editor, *Proceedings of the First AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software*, pages 45–50, Enschede, The Netherlands, Apr. 2002.

[33] A. Shalit. *The Dylan Reference Manual: The Definitive Guide to the New Object-Oriented Dynamic Language*. Addison-Wesley, Reading, Mass., 1997.

[34] A. Tripathi, E. Berge, and M. Aksit. An implementation of the object-oriented concurrent programming language SINA. *Software Practice and Experience*, 19(3):235–256, Mar. 1989.

[35] J. C. Wichman. Composej: The development of a preprocessor to facilitate composition filters in the Java language. Master's thesis, Department of Computer Science, University of Twente, Enschede, the Netherlands, Dec. 1999.