Concerning AOP and Inheritance

Stefan Hanenberg, Rainer Unland
Dept. of Mathematics and Computer Science
University of Essen, D - 45117 Essen
{shanenbe, unlandR}@cs.uni-essen.de

Abstract

Aspect-Oriented Programming (AOP) has recently been proposed as a new paradigm for software development. It supplies mechanisms and constructs for expressing concerns separated from each other. There are already general-purpose aspect languages which offer these mechanisms on implementation level.
Aspect-oriented mechanisms can be used for changing the behavior of objects. The same results can be achieved using inheritance, well known in the object-oriented world.
This paper compares those techniques and introduces, how inheritance can be applied to AOP.

1. Introduction

Aspect-Oriented Programming (AOP) has recently been proposed as a new paradigm for software development. It supplies mechanisms and constructs for expressing concerns separated from each other.

The core elements of object-oriented approaches are objects, classes and inheritance (cf. [9]). Classes are templates from which objects can be created and define an interface for them. Inheritance (cf. [8]) is a mechanism for deriving new class definitions from existing ones.

A class inheriting from another might add additional members, or redefine members of the upper class. By redefining methods it is possible to change the behavior of derived objects. Inheritance is based on the structural descriptions of objects consisting of classes, attributes and methods. It is not possible to reuse method definitions on a finer granularity, e.g. for redefining certain statements, because the implementation of a method is hidden to the developer.

This mechanism for changing behavior is very rigid, because it is only possible to override the whole method and not parts of it.

While object-oriented programming languages use structural elements like classes, attributes and methods as extension points, general-purpose aspect languages (GPAL) like AspectJ (cf. [1], [4]) or Sally [7] introduce a new extension point: *interaction*.

This paper discusses how this new extension point can be used for changing behavior of objects, which was achieved using inheritance in the object-oriented world.

The remainder of this paper is structured as follows. Section 2 introduces interaction as new extension point, section 3 introduces our general-purpose aspect language Sally. Section 4 discusses the relationship between inheritance and the aspect-oriented extension point. Section 5 discusses the inheritance relationship between aspects, and section 5, finally, summarizes and concludes this paper.

2. Aspect-Oriented Extension Points

Aspect-Oriented Programming permits to treat different aspects or concerns separately. Adding a certain new aspect to existing code means to add cross-cutting code. In order to overcome the rigid and inflexible approach of object-oriented technology towards the change of behavior general-purpose aspect languages like AspectJ (cf. [1], [4]) or Sally [7] introduced an additional extension point: *interaction*.

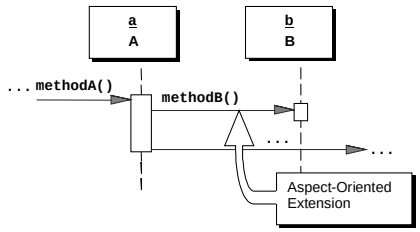

Figure 1: Extending interactions using AOP

Figure 1 shows how an instance of `A` in `methodA` interacts with other objects. The object-oriented approach would just permit to reuse the method-implementation "as is". With the aspect-oriented approach the interaction between `A` and `B` can be exploited by adding some code to this specific interaction. Whenever the specified interaction happens this additional code is executed. For this purpose the participants of the interaction and the code which is supposed to be executed need to be specified. In this context we use the term *aspectual extension*, because the origin code is extended with the aid of aspect-orientation. Whenever the code is executed, we use the term *aspectual invocation*, because the code is executed although it is not directly embedded in `A`.

In the next section we introduce the metamodel our GPAL Sally is based on and compare it shortly with AspectJ from the Xerox Palo Alto Research Center,

which is the most popular and well-established aspect language.

3. Metamodel of Sally

We will introduce the metamodel for our GPAL Sally "bottom-up" what means, that we show how interaction can be specified before we explain what aspects consist of.

3.1. Join Points

Sally defines join points as participants of an interaction and is in that way contrary to the definition from AspectJ, which introduces join points as "principled points in the execution of the program" [4].

A participant of an interaction is always a method, which is defined in Sally via a 4-tupel consisting of the class identifier, the return type identifier, the method name and the parameter type identifiers.

So, a valid join point in Sally is: `joinpoint j1 {"B", "void", "methodB", ""}`.

3.2. Pointcuts

Pointcuts specify interactions which can be used as extension points. Thereto pointcuts are combinations of join points. AspectJ defines these combinations as boolean expressions of join points. However, in Sally a pointcut is a combination of exactly 2 join points: one for the caller and one for the receiver.

For example, specifying a pointcut for the interaction between `methodA` of an instance of `A` and `methodB` of an instance of `B` can be done in Sally as follows:

```
joinpoint c {"A","void","methodA", ""};
joinpoint r {"B", "void", "methodB", ""};
pointcut p1 {c, r};
```

This defines two join points `c` and `r`, and a pointcut `p1`, which defines `c` as the caller and `r` as the receiver. Although there is an additional effort to define each named join point the advantage of this approach is that every join point can be used in arbitrary many pointcuts without redefining it. We speak of an *activated pointcut* whenever a point in the execution of a program is reached which matches a specified pointcut.

While join points and pointcuts are used to specify interactions, the action taking place at this interaction has to be specified by using pointcut methods.

3.3. Pointcut methods

Pointcut methods specify the code that is meant to be executed whenever certain interactions happen. A pointcut method, therefore, is the cross-cutting code because it may be executed at numerous execution points in the program. We use the term *pointcut method* because they are invoked whenever a certain pointcut is activated. AspectJ calls such a method an *advice*.

The declaration of a pointcut method must define at what pointcuts it is meant to be executed. So, every pointcut method refers to one or more pointcuts. Additionally, it must specify at what point in time it is supposed to be executed: a pointcut method may either be executed *before* or *after* a certain interaction happens or may even *replace* the invoked method.

In AspectJ pointcut methods are defined as follows:

```
before(): aPointcut() {
    ...do something...
}
```

This pointcut method is executed before the pointcut `aPointcut` is activated. AspectJ does not use named pointcut methods. We regard that as a disadvantage because a pointcut method is a special method which is invoked whenever a pointcut is activated. Without regarding pointcut methods as special methods it would not be possible to specify pointcuts that have a pointcut method as an participants of an interaction. That is the reason why AspectJ does not allow to declare join points which refer to pointcut methods.

To avoid such problems pointcut methods in Sally are named methods. Moreover, the metaclass `PointcutMethod` extends the metaclass `Method`. So, the Sally version of the above pointcut method definition is:

```
public void beforeAPointcut(...)
    before aPointcut {
      ...do something...
}
```

The pointcut method modifier (keyword "`before`" in our example) is an attribute of the relationship between pointcut methods and pointcuts (figure 2) and declares at what point in time in relation to the interaction a pointcut method is invoked.

3.4. Aspects

In general-purpose aspect languages aspects are constructs which contain those fragments of code, which cross-cut numerous decomposition units or modules.

On the implementation level aspects are constructs which consist (in Sally) of attributes, methods, join points, pointcuts, and pointcut methods (figure 2). In AspectJ join points are not part of the aspect, however, part of the pointcut.

Based on that model it is possible to distinguish between classes and aspects. An aspect has in addition to attributes and methods join points, pointcuts, and pointcut methods. So, the metaclass `Aspect` is supposed to extend the metaclass `Class`.

```
aspect ExtendeAB {
  joinpoint j1 {"A", "void", "methodA", ""};
  joinpoint j2 {"B", "void", "methodB", ""};
  pointcut p1 {j1, j2};
  public void beforePM(...)
    before p1 {
      ...do something...
  }
}
```

The code example above shows a valid aspect declaration for the example in figure 1. Before an instance of A in method `methodA()` sends a message `methodB()` to an instance of B the pointcut method `beforePM` is invoked. This is an aspectual extension to the interaction between A and B.

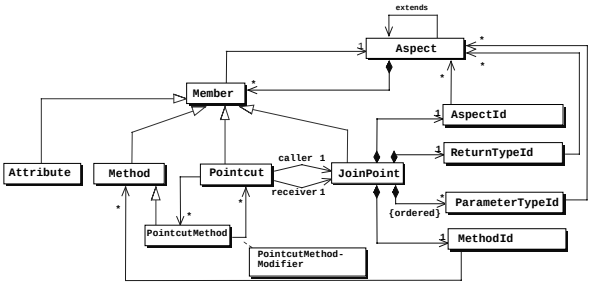

Figure 2: GPAL Metamodel

We didn't regard the parameters used in a pointcut method, because it is not necessary for the further discussion.

4. Aspectual Extension versus Inheritance

Aspectual extension is a way to change the behavior of objects. Design patterns like *template method*, *strategy* or *decorator* (cf. [2]) have the same intention and make exhaustively use of inheritance. In that way it is necessary to discuss, how inheritance relates to aspectual extension.

Based on out metamodel aspects are some "special classes" which have three additional members: join points, pointcuts and pointcut methods. So it is also necessary to discuss how the inheritance relationship between aspects is influenced by those new members. Part of this discussion is, how mechanisms like overriding methods can be mapped to aspect-oriented programming.

Aspectual extensions can be used for changing, adding or replacing behavior. In this way aspectual extension is quite similar to inheritance which allows to override and overload methods. The difference between both techniques is, that inheritance is an abstraction principle for all of its objects. So all objects created from a certain class behave all the same and in that way the behavior of an object depends on its type. In contrast to that aspectual extension based on Sally can be adopted for interaction between objects of a certain type. So the behavior of objects no longer depends only on its type, but also on the type of those objects whose messages are received. Aspectual extension doesn't introduce a new type for modifying behavior.

However, AspectJ does not allow to specify both participants of an interaction and in that way, the behavior of an object does not depend on both participants of an interaction.

Our aspect language Sally doesn't allow to introduce new attributes or methods. Other GPALs like AspectJ allow to add members using *introductions* (cf. [4]). In this section we will neglect the possibility to add new members and compare inheritance and aspectual extension regarding the possibility to adopt behavior.

4.1. Single Inheritance

In the Sally-example below the pointcut method `pcm` is executed before any arbitrary object sends the message `getX()` to an instance of A. As long as no other aspects are registered at A, all of its instances behave as defined there.

```
aspect ExtendedA {
  joinpoint j1 {"*", "*", "*", "*"};
  joinpoint j2 {"A", "Integer", "getX", ""};
  pointcut p1 {j1, j2};
  public void pcm(...) before p1 {
      ...do something...
  }
}
```

Almost the same result might be achieved creating an subclass of A and overriding method `getX()`. This assumes the existence of late binding in the underlying programming language. The following example is written in Java and assumes A to be a valid class.

```
class ExtendedA extends A {
  public Integer getX(...) {
      ...do something...
      return super.getX();
  }
}
```

The difference between both approaches is, that the aspectual extension works on the same objects of type A. It does not need an instance of `ExtendedA` to be created. Using inheritance leads to change the code responsible for creating instances of `A`. Creational patterns (cf. [2]) reduce this problem. If no such design patterns are used, the effort might be immense. So the main distinction between aspectual invocation and inheritance is, that the extensions can be done without changing the code responsible for creating objects.

The example used a *before* pointcut method. So the code is executed before the interaction between the participants takes place. Another kind of pointcut method is *after*, which enforces the method to be executed after the interaction takes place. Before and after pointcut methods can be used for adding behavior without changing the predefined implementation.

As long as the aspectual invocation does not influence the execution of the target method for example by changing the actual parameters, the usage of

before and after pointcut methods can be compared to *strict inheritance* (cf. [10]): the extension is behavior compatible but enforces the execution of some additional code.

The third kind of pointcut method is *instead*, which replaces the receiver's method implementation. Aspectual extension based on instead pointcut methods can be compared to overriding a method without referring the origin implementation. It depends on the developer if the new implementation is behavior compatible or not. The current GPALs don't supply any mechanism to guarantee the compatibility of behavior.

The example we introduced had some specific characteristics:

- the execution of the pointcut method was independent of caller's type
- the pointcut method was related to exactly one pointcut

As long as the executed code is independent of the caller inheritance can be used instead of aspectual extension for purposes of adopting behavior. If the code to be executed depends on the caller, inheritance alone is not sufficient. In this case the caller has to be identified by the receiver, so the caller has to send an additional `this`-parameter with the message. The overridden method has to decide with the aid of this parameter how to behave.

If the pointcut method is related to exactly one pointcut and that pointcut relates to exactly one join point, the result achieved is similar to single inheritance. Otherwise the pointcut method is executed at several join points. Pointcut methods executed at more than one pointcut can be compared to the usage of multiple inheritance.

4.2. Multiple Inheritance

The programming language Java which is the basis for the GPALs Sally and AspectJ doesn't support multiple inheritance for implementation reuse. For this reason we will discuss the relationship between aspectual extension and multiple inheritance on a more abstract level.

```
aspect ExtendedAB {
  joinpoint j1 {"*", "*", "*", "*"};
  joinpoint j2 {"A", "Integer", "getX", ""};
  joinpoint j3 {"B", "Integer", "getX", ""};
  pointcut p1 {j1, j2};
  pointcut p2 {j1, j3};
  public Integer pcm(...) instead p1,
                          instead p2 {
      ...do something...
  }
```

In the example above the pointcut method `pcm` is executed whenever an instance of `A` or `B` receives a message `getX()`.

This adoption of behavior can also be done creating a new subclass of `A` and `B` and override method `getX()` (figure 3).

Although using multiple inheritance seems to be similar to aspectual extension, there are a lot of differences. Multiple inheritance introduces a new type `ExtendedAB`, which combines the interfaces of `A` and `B`. However, aspectual extension just leads to a new implementation of `getX()` without changing the interface. Using multiple inheritance assumes the underlying programming language to handle name collisions (cf. [6]). Such a mechanism is not needed by aspectual extension, because is doesn't combine any methods so name collisions can not happen.

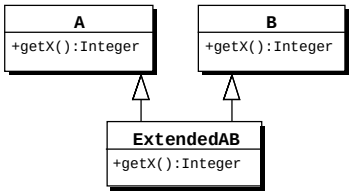

Figure 3. Multiple Inheritance

The aspect-oriented examples discussed here used concrete join points. If join points are abstract, then the registration of aspects depends on, how they are defined in subaspects. That means the aspect's implementation is done, but leaves it open to the developer what aspects should be extended. This mechanism is similar to mixin-based inheritance, what we will discuss in the next section.

4.3. Mixin-based Inheritance

A mixin is an abstract subclass that may be used to specialize the behavior of a variety of parent classes [2]. Mixin classes don't have superclasses and are not therefore structurally bound to any specific place in the inheritance hierarchy [8]. Whenever a certain class extends a mixin, it becomes part of the inheritance structure. So the place in the inheritance hierarchy is not fixed by the mixin itself, but by classes, which extend them.

Like in the sections before we neglect the fact, that mixins allow to introduce new members. Instead we concentrate on how mixins can be used to change the behavior of predefined classes.

Let's assume we have 2 classes `A` and `B`, both supplying a method `getX()` with return type `Integer`. If the implementation of `getX()` has to be replaced it might be done using single inheritance (a new subclass of `A` and a new subclass of `B`) or multiple inheritance (a new class inheriting from `A` and `B`).

The first approach has the disadvantage of redundant code, the other one combines the interfaces of `A` and `B` together in a subclass.

In this situation the usage of mixins might be more appropriate. A new mixin class implementing `getX()` has be to created and whenever a new subclass of `A` or `B` is needed, this class extends either `A` or `B` and the mixin

class. The advantage of this approach is, that there is no redundant new implementation of `getX()` and the interface of the new type is not "wasted" with both interfaces of A and B.

The same result can be achieved using aspectual extension.

```
abstract aspect ExtendedAB {
  joinpoint j1 {"*", "*", "*", "*"};
  abstract joinpoint j2 {AspectIdentifier ai}
              {ci, "Integer", "getX", ""};
  pointcut p1 {j1, j2};
  public Integer pcm(...) instead p1 {
      ...do something...
  }
}
```

The code extract above shows how an abstract aspect, containing an abstract join point `j2`, may be used to overwrite the method `getX()` of some aspects. Which aspects it extends has to be specified by the developer by extending the aspects and defining the receiver's aspect name. If the developer wants to replace `getX()` of every instance of A, he has to set the abstract aspect identifier `ai` to A.

```
aspect ExtendedA extends ExtendedAB {
  joinpoint j2 {ai="A"};
}
```

While the previous sections discussed how aspectual extension may be used instead of inheritance we made here already use of the inheritance relationship between aspects. In the next section we will discuss this relationship in detail.

5. Inheritance between Aspects

An aspect without join points, pointcuts and pointcut methods is equal to an object-oriented class. So an aspect's extension can access all resources defined in its upper aspect.

The question is, how the inheritance-relationship looks like in aspects having join points, pointcuts and pointcuts methods. We will begin our discussion with an example taken from AspectJ.

5.1. Extending Aspects in AspectJ

AspectJ allows an aspect to extend another aspect. All members defined in an upper aspect are also part of its extension. The following example is similar to a code example in AspectJ.

```
aspect Trace {
  pointcut printouts():
      instanceof(HelloWorld) &&
      receptions(void printMessage ());
  static before(): printouts() {
    System.out.println("before");
  }
}
```

Whenever an instance of `HelloWorld` receives a message `printMessage()` the corresponding before pointcut method is executed. This aspect can be extended by another aspect `Trace2`.

```
aspect Trace2 extends Trace {
  static before(): printouts() {
    System.out.println("another before");
  }
}
```

The result of that extension is, that before an instance of `HelloWorld` receives a message `printMessage()`, the pointcut method of `Trace2` and afterwards the pointcut method of `Trace1` is executed.

Although this might be the expected behavior, there is a fundamental problem: there is no chance to redefine the behavior of the pointcut method in `Trace`. In other words: a once defined static pointcut method can never be adopted anymore.

The reason for this problem is, that AspectJ doesn't treat pointcut method as "usual" methods. They do not have a name. So there is no way to decide, if the developer wants to redefine the same pointcut method, or would like to introduce another one. He also has no possibility to add an aspectual extension to the pointcut method, because therefore a method name is needed. For this reason we recommend in our metamodel, that `pointcut method` extends `method`. and realized it in Sally.

If the developer overrides a pointcut method, the overridden method will be executed instead of the method inherited from the upper aspect. So on instance level it is clear what method has to be invoked. In this way it is also possible to override static pointcut methods in Sally.

5.2. Extending Aspects in Sally

Sally allows like AspectJ to extend aspects. An aspect extending another one shares all of its resources. So attributes, methods, join points, pointcuts and pointcut methods defined in an aspect are also part of its descendent. A child of an aspect might also introduce new members.

It is possible to define abstract aspects, which can not be instantiated. Abstract aspects, which contain some predefined behavior, may contain abstract join points. So the developer extending the aspect is responsible for defining, at what join points the aspectual invocation happens.

Overriding pointcut methods is done in the same way like overriding methods in Java. A pointcut method overriding another one must have exactly the same signature.

Overriding static pointcut methods is handled in Sally as follows. Whenever a static pointcut method of an ancestor is overridden, the overriding method will be invoked instead of the overridden one. We think this

approach is more appropriate for extending aspects including static pointcut methods like the one supported in AspectJ.

As a result this approach leads to a kind of *inversion of control*: a pointcut declaration in an upper aspect is responsible for the invocation of a static pointcut method of its descendant.

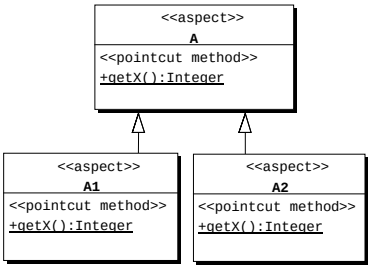

Figure 4. Overriding static pointcut methods

The example in figure 4 shows an aspect `A` having two subaspects `A1` and `A2`. The corresponding join points and pointcuts are not considered in this example. Whenever the static pointcut method `getX()` of `A` is activated the corresponding pointcut methods of `A1` and `A2` are invoked, but no longer the method of `A` itself.

6. Conclusion and further work

In this article we compared aspect-oriented extension and object-oriented inheritance. We showed, how aspectual extension can be used instead of inheritance for adopting behavior.

Besides we discussed the inheritance-relationship between aspects and especially the problem of overriding static pointcut methods. We introduced the approach used by Sally to handle static pointcut methods.

In the future we will examine, how the mechanism of method overloading can be applied to AOP.

7. References

[1] AspectJ Homepage, `http://www.aspectj.org/`, 2001

[2] Gilad Bracha, William Cook. Mixin-based Inheritance. In: Norman Meyrowitz (Ed.). *OOPSLA / ECOOP'90 Conference Proceedings*, ACM SIGPLAN Notices 25, 10, pp. 303-311, 1990

[3] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995

[4] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, William G. Griswold. *An Overview of AspectJ*. To appear in ECOOP 2001, 2001

[5] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Christina Lopes, Jean-Marc Loingtier, John Irwing. *Aspect-Oriented Programming*. Proceedings of ECOOP '97, LNCS 1241, Springer-Verlag, pp. 220-242, 1997

[6] Jorgen Lindskov Knudsen. *Name Collision in Multiple Classification Hierarchies*. In Gjessing, Nygaard (Eds.). *ECOOP '98 European Conference on Object-Oriented Programming*, LNCS 322, Springer-Verlag, pp. 93-109, 1998

[7] Sally: A General-Purpose Aspect Language, `http://www.cs.uni-essen.de/dawis/research/aop/sally/`, January 2001

[8] Antero Taivalsaari. *On the Notion of Inheritance*. In: ACM Computing Surveys, Vol. 28, No. 3, pp. 439-479, 1996

[9] Peter Wegner. *Dimensions of object-based language design*. In: N. Meyrowitz (Ed.), Proceedings of OOPSLA '87, SIGPLAN Notices 22 (12), pp. 168-182, 1987

[10] Peter Wegner . *The object-oriented classification paradigm*. In: Bruce Shriver, Peter Wegner (eds.). *Research Directions in Object-Oriented Programming*. MIT Press, pp. 479-460, 1987