# Using and Reusing Aspects in AspectJ

Stefan Hanenberg and Rainer Unland

Institute for Computer Science
University of Essen, 45117 Essen, Germany
{shanenbe, unlandR}@cs.uni-essen.de

**Abstract**. One of the major benefits of Object-Oriented Programming (OOP) is, that it supports inheritance as a mechanism for *code reuse* and *incremental modification*. Well-established object-oriented technologies like white-box frameworks make exhaustively use of and intensively rely on inheritance. Aspect-Oriented Programming (AOP) is a new technology, which supports another extension mechanism whose intension is to enable a better separation of concerns. General-purpose aspect languages (GPAL) like *AspectJ* are build upon object-oriented programming languages, and in that way it is possible to combine both extension mechanisms. This paper shows that both mechanisms are in contradiction to each other. The result is, that the advantages of inheritance are no longer available. Furthermore, incremental modifications and code reuse are not directly supported for the new language features of AspectJ. So, the application of techniques like redefining aspect-specific behavior is not possible. This paper proposes some *rules of thumb*, which allow to modify aspects incrementally and reuse aspect-specific code in AspectJ. Moreover, this paper shows how in that way the cooperation of inheritance and aspectual modification is achieved.

## 1 Introduction

Inheritance is the core mechanism of object-oriented programming (OOP). (Class-based) inheritance permits to derive new class definitions from existing ones and enables in that way *code reuse* (or *code sharing*, cf. [1]) from upper classes to its' descendants. An extension of a class can introduce new attributes and methods which are based on existing ones. Additionally, it is possible to override methods to create new classes, whose objects behave in a way that fits the needs at hand.

The importance of inheritance as a mechanism for *incremental programming* (or *incremental modification*) is widely accepted and has been discussed by numerous authors (cf. e.g. [4], [12], [13]). In contrast to the everyday sense, incremental modification realized by inheritance is not a process of *destructive change*, because there is no need to tamper with existing source code [4].

Aspect-Oriented Programming (AOP, [10]) has been emerging as a technology for expressing multiple concerns in software development. General-purpose aspect languages (GPALs) like *AspectJ* [9], which are based on object-oriented programming languages, support another mechanism in addition to inheritance. This mechanism (we termed *aspectual extension* in [7]) defines code which has to be executed whenever a certain *join point*, i.e. a principled point in the execution of the program [9], is reached. Thereto, GPALs introduce new language constructs like *pointcuts*, *advice* and *aspects* (c.f. [9], [8]). The question arises, if the advantages of code reuse and incremental modification are still available in AOP.

Applying AOP using AspectJ without any foresight leads to some trouble. There are situation, where the mechanisms of inheritance and aspectual extension are in contradiction to each other. In such situations it is necessary to apply destructive modifications within the aspect code. Before preparing the possibility to modify aspect code incrementally it is necessary to discuss what ingredients of an aspect should be modified. Furthermore, the question arises, what parts of an aspect should be reused by other aspects.

This paper proposes some *rules of thumb* which allow aspects to be incrementally modified and reused by other aspects. Thereto, section 2 states the problem of the contradicting mechanisms. The next section discusses what parts of an aspect should be reused. Section 4 introduces the rules of thumb for enabling incremental modifications and code reuse in AspectJ, afterwards the mechanism for reusing pointcuts and advices is illustrated. In section 5 the introducing example will be revised regarding the possibilities of incremental modification and code reuse. Finally, section 6, summarizes and concludes this paper.

## 2 Motivation and Problem Description

We assume to have an interface `ListModel`, which defines methods for adding either a single list item or a collection of list items to its instances. Furthermore, we would like to make all instances of `ListModel` observable. Making certain instances observable is an usual application of aspect-oriented programming. Thereto, in AspectJ an aspect has to be to defined consisting of a poincut to determine, when a new object is added to `ListModel` and a corresponding advice, which informs all observers about this change. In the following we assume to have `ListModel` and `ObservableListModel` like given below. We just focus on the definition of the pointcut `listChangedPC()`.

```
interface ListModel {
    void add(ListItem o);
    void add(Collection o);}
aspect ObservableListModel {
    pointcut listChangedPC(): ......;
    after():listChangedPC() {
      ...inform each observer}}
```

The first attempt to fulfill the given requirement is to define a poincut which is activated, whenever the method `add(ListItem)` is invoked:

```
pointcut listChangedPC():
    instanceof(ListModel) && receptions(void add(ListItem));
```

This pointcut definition has a significant disadvantage: if method `add(Collection)` uses `add(ListItem)` for every element in the collection, all observers are informed for every single item added to the list. Instead, for reasons of efficiency all observers should be informed just once after calling `add(Collection)`. In [3] this is called a *jumping aspect*: the aspect code to be executed depends on the context. Whenever `add(ListItem)` is called, the advice should be executed, but not as a result of an `add(Collection)`-invocation. If `add(Collection)` is called the advice should be executed only once.

[5] argues the other way round: if the pointcut is defined as above and method `add(Collection)` does not use `add(ListItem)` the observers will not be informed. So the aspect seems to vanish and [5] calls such a situation *vanishing aspects*.

It seems like jumping and vanishing aspects can be easily avoided. The problem is, that the definition of `listChangedPC()` does not consider, that `add(ListItem)` can be invoked by `add(ListItem)`. Or spoken in terms of [11] and [6]: it has to be considered, that `add(ListItem)` might be a *hook method* invoked from the *template method* `add(Collection)`. This can be considered as follows:

```
pointcut listChangedPC():
    instanceof(ListModel) &&
          ((receptions(void add(ListItem)) &&
              !cflow(receptions(void add(Collection))))
           || receptions(void add(Collection)));
```

The definition above guarantees that the pointcut is activated in the right way independent of how `add(Collection)` is implemented. Since, this definition is an appropriate solution for the problem of jumping and vanishing aspects in `ListModel`.

Assuming that there is a need for a subtype of `ListModel`, which declares a method for adding an array of `ListItem` to a list:

```
interface ListModelExt extends ListModel {
    void add(ListItem[] li);}
```

There are different possibilities to implement `add(ListItem[])`: it may invoke `add(ListItem)` to insert all objects to be added. In such a case all observers are invoked every time an object from the array is added and the problem we resolved before exists for another time. Maybe the new method uses `add(Collection)`, so all observers are informed rightly. Another possibility is, that the new method uses none of the former methods and no observer will be informed. Some bizarre implementations may even use a combination of `add(ListItem)` and `add(Collection)`.

So, after extending `ListModel` the following problems arise:

- if the implementations of the new method use one of the former methods, the pointcut `listChangedPC()` has to be changed. This can be done only destructively, because AspectJ does not allow to extend concrete aspects.

- if none of the former methods is used, the code for informing the observers has to be added. This could be realized by overriding the pointcut, or by adding a new pointcut and a new advice. This also can be done just by destructive modifications.

The problem arose, because inheritance and aspectual extension are contradicting mechanisms in our example: the extension realized via inheritance is no extension from the aspect's point of view. This can be regarded as a special kind of composition anomaly (cf. [2]): two concerns expressed by different mechanisms have to be coordinated.

The previous example showed that is not easy to use aspects in AspectJ, not to mention to reuse them. Because of the restricted mechanism of AspectJ concerning extending aspects (see [8] for a detailed discussion), it is only possible to modify the aspect in a non-destructive way to achieve the desired result. Nevertheless, it is possible to bypass that weakness by following some *rules of thumb*, which allow to modify aspects incrementally.

# 3 What should be reused?

In AspectJ aspects consist of pointcuts, advice, attributes and methods (neglecting introductions). The necessity of reusing attributes and methods is already widely discussed in the area of object-oriented programming and needs not to be repeated here. Instead we will focus on the reuse of pointcuts and advice.

Pointcuts are constructs which define, when corresponding advice have to be executed. There are mainly two situations where reusing pointcuts is reasonable:

- *Overriding Pointcuts definitions*: In the introducing example we showed, that a certain pointcut definition might lead to some problems when the original class will be extended. In such a situation it is necessary to override a pointcut. Furthermore we would like to keep the advantage of incremental modifications known from the object-oriented programming. So overriding pointcuts should not be done by a destructive modification, i.e. sources of a once defined pointcut should not be changed.
- *Using pointcut definitions for other aspects*: Whenever a new aspects contains advice which should be executed at the same principled points in the execution of the program it is reasonable that this aspect uses the same pointcut definition like the existing one. An example of this is, that every time an instance of `ListModel` changes, this should be logged to a file. In such a situation the pointcut definition from `ObservableListModel` could be reused.

In some way advice is the core ingredient of AspectJ. An advice is the code, which has to be executed whenever a joinpoint specified in the corresponding pointcut is reached. The question is, what part of an advice should be reused: the *body* of the advice, or also the header, which defines, when the advice will be executed (*before*, *after*, *around*). In some situations it is not relevant, if an advice is executed before or after a certain join point, so it is to be desirable to reuse only the body of an advice. On the other hand an around-advice contains the commands responsible for executing the joinpoint it surrounds, so the body of an around-advice can not be used as a before-advice.

Another question is, if it is plausible to override an advice: if there is the need to override an advice it is questionable, if the corresponding aspect should be really applied. Corresponding to our introducing example, it is reasonable to apply an advice to another pointcut. Then we would be able to use the observer-behavior in other situations without the need to redefine the corresponding pointcut.

In summary, we do not think there is a need to override advice, but to apply an advice to another pointcut.

# 4 Rules of Thumb for Reusing Aspects in AspectJ

We regard the following rules as the beginning of numerous idioms for AspectJ that will follow in the future. However, AspectJ did not reach a version 1.0 until now, so the language may change in the future in some fundamental ways, so the usage of some of the following rules will not be necessary. The rules themselves contain no information about, how to reuse aspects. Instead they are a requirement for the mechanism we will introduce in section 5.

## 4.1 Rule 1: Separated Pointcut Declarations

The problem of redefining pointcuts exists almost every time when a class, an aspect is woven to, has to be extended via inheritance. Therefore, it is necessary to separate pointcut declaration and definition from the rest of the aspect.

Whenever a new aspects is created, a corresponding abstract super-aspect has to be implemented which contains all pointcuts declarations and / or definitions needed by the aspect. Advice in the sub-aspect refer to the pointcuts defined in the super-aspect. This rule also implies, that all pointcuts must be named and must not be part of an advice-definition.

### 4.2 Rule 2: *Using Hook Pointcuts*

AspectJ does not allow to define pointcuts recursively. E.g. if there is an abstract aspect with a pointcut `listChangedPC()` it is not possible define a sub-aspect and override the pointcut like

```
listChangedPC() (): super.listChangedPC() && ......;
```

because AspectJ does not allow to refer to pointcuts in a super-aspect with a special variable like `super`.

When a pointcuts has to be defined incrementally, i.e. based on an existing pointcut, a new pointcut has to be declared and defined and afterwards it has to be assigned to the target pointcut.

For the example above we have to define a concrete pointcut like:

```
listChangedPC_1(): ...pointcut definition...;
listChangedPC():listChangedPC_1();
```

If we redefine later the pointcut `listChangedPC` based on the existing definition, we have to create a new pointcut, which uses the hook pointcut `listChangedPC_1`:

```
listChangedPC_2(): listChangedPC_1() && ...pointcut definition...;
listChangedPC() ():listChangedPC_2();
```

We recommend to use a certain naming convention which reveals, to which target pointcut a hook pointcut belongs to. Furthermore, it is necessary to document, which hook pointcut belongs to what aspect, otherwise there will be some naming conflicts with existing pointcuts.

We do not think that this rule has to be used in the future, because we expect AspectJ to support special variables to refer to pointcuts defined in a super-aspect in the future.

### 4.3 Rule 3: *No pointcut for more than one advice*

In AspectJ a pointcut might be used by more than one advice. If one pointcut is used for more than one advice, there is no possibility to adapt the behavior of a single advice. This rule is important, if a developer becomes aware of, that a once defined aspect contains more than one concern. Nevertheless there are situations where it is necessary to use the same pointcut for more than one advice (especially when *before*- and *after*-advice are used in combination). In such a situation we recommend to use the same hook pointcut and to assign it to two different target pointcuts:

```
myPC_1():...pointcut definition...;
targetPointcut1() ():myPC_1();
targetPointcut2() ():myPC_1();
```

### 4.4 Rule 4: *Concrete aspects are always empty*

If an advice, attributes and methods have to be reused, it must not be in concrete aspects, because of the pre-mentioned restrictions in AspectJ. Once an advice is within a concrete aspect, it becomes lost for any further reuse. Likewise, this rule claims than a concrete aspect does not contain any pointcut definition. This guarantees the possibility to redefine a concrete aspect's pointcut. Also all attributes and methods needed by a concrete aspect have to be defined in a super-aspects.

## 5 Reusing Poincuts and Advice

If those rules are applied, the definition of an aspect consists of at least three aspects (fig. 1):

- one abstract aspect for the pointcut declarations (`MyAspectPCDs`),
- one abstract aspect for advice, attributes and methods (`MyAspectAbstract`)
- one concrete (empty) aspect (`MyAspectConcrete`)

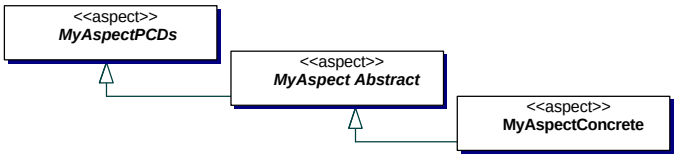


**Figure 1: Structuring aspects for reuse**

The aspects are connected by an inheritance relationship. The structure does not determine where to define the concrete pointcuts. In some situation is makes more sense to define the concrete pointcuts in a descendant of `MyAspectPCDs`, but not within `MyAspectAbstract` because of rule 1. In other situations it is more

appropriate to define concrete pointcuts in a descendant of `MyAspectAbstract`, but not in `MyAspectConcrete` because of rule 4.

The main idea behind the reuse of aspects in AspectJ is, to extend the inheritance relationship by using introductions, which are a special mechanism in AspectJ. Concretely, the possibility of introducing a new super-aspect to an already existing one is applied.

If we need to override a pointcut defined in an aspect A, we extend the aspect where the pointcut is defined in and introduce this new aspect as a direct ancestor of A's child. Rule 4 guarantees that there is always an ancestor of that aspect we would like to extend.

If we would like to reuse an aspect implementation without its concrete pointcuts, we extend `MyAspectAbstract` and create a new concrete aspect. Additionally we have to create a new aspect containing the concrete pointcuts and introduce this as the direct ancestor of the `MyAspectAbstract` extension.

## 6 Revising the example

The main problem in the introducing example was, that the once defined pointcut could not be overridden without destructive modifications. To permit such an extension, we have to apply the rules of thumb we proposed in section 4. The interface `ListModel` remains unchanged, so we focus on the definition of `ObservableListModel`.

The first rule enforces to separate the pointcut declaration from the rest of the aspect implementation. Furthermore, we would like to define the pointcut directly in that aspect and we use the second rule to allow the reuse of the pointcut definition.

```
abstract aspect ObservableListModelPCDs {
    pointcut listChangedPC_01():
      instanceof(ListModel) &&
      ((receptions(void add(ListItem)) &&
        !cflow(receptions(void add(Collection)))
       )|| (receptions(void add(Collection))));
    pointcut listChangedPC(): listChangedPC_01();}
```

Afterwards we have to define the rest of the functionality in another aspect `ObservableListModelAbstract` extending `ObservableListModelPCDs`. Rule 4 enforces to declare that aspect as abstract. Like in the introducing example we do not discuss the implementation of that functionality.

```
abstract aspect ObservableListModelAbstract
      extends ObservableListModelPCDs {
      ...implementation...}
```

The last step is to define the empty concrete aspect.

```
aspect ObservableListModelConcrete
      extends ObservableListModelAbstract  {}
```

After extending `ListModel` we have to override the pointcut, because we just want to inform every observer after `add(ListItem[])`is executed. Thereto we have to extend the aspect containing the pointcut definitions. Additionally, we have to introduce this new aspect as the ancestor of aspect `ObservableListModelAbstract`.

```
abstract aspect ObservableListModelPCDsExtension
      extends ObservableListModelPCDs {
      pointcut listChangedPC_02():
            (listChangedPC_01() &&
            !cflow(receptions(void add(ListItem[])))
            || (instanceof(ListModelExt) &&
              receptions(void add(ListItem[]))));
      pointcut listChangedPC():listChangedPC_02();
      ObservableListModelAbstract
            +extends ObservableListModelPCDsExtension;}
```

In that way we are able to modify the pointcut incrementally without the need to touch its existing source code. Furthermore, we reused the definition of the former pointcut for the new one.

## 7 Discussion and Conclusion

We demonstrated that an unprepared usage of the GPAL AspectJ leads to situation, where incremental modifications are no longer possible. The reason for it is the contradiction of inheritance and aspectual extension. In that way the whole application of Aspect-Oriented Programming (and especially AspectJ) would be questionable. We proposed four simple rules, which (when applied) fix this problem. Applying those rules permits to build reusable and incrementally modifiable aspects. Thereto we discussed for what ingredients of an aspect reuse is reasonable.

This paper leaves some issues open, which have to be discussed in the future:

1. *effect of incremental modifications*: the effect of incremental modifications on aspects is different than the modification achieved via inheritance in OOP. In AOP an incremental modification influences all classes and objects, the aspect is woven to.
2. *using hooks within advice*: a possibility to change an advice's behavior incrementally would be to add certain hooks within the advice. In such a case those hooks could be extended via inheritance or via aspectual extension. Both alternatives need to be investigated in much detail.

We regard our proposed rules of thumb as the beginning of some idioms, that will arise in the future for AspectJ. Applying those rules leads to a certain structure for aspect definitions. This structure can be modified using introductions for overriding pointcut definitions or inheritance for reusing advice. In that way the aimed reusablilty and incremental modifiability is realized.

In the future we will investigate the impact of the proposed approach for reusing larger composition units.

## References


1. Pierre America. Designing an Object-Oriented Programming Language with Behavioural Subtyping. Proceedings of REX Workshop 1990, LNCS, Vol. 489, Springer, 1991, pp. 60-90
2. L. Bergmans and M. Aksit, Composing Software from Multiple Concerns: A Model and Composition Anomalies, ICSE'2000 Workshop on Multi-Dimensional Separation of Concerns in Software Engineering, Limerick, June 2000
3. Johan Brichau, Wolfgang De Meuter, Kris De Volder. Jumping Aspects, Position paper at the workshop Aspects and Dimensions of Concerns, ECOOP 2000, Sophia Antipolis and Cannes, France, June 2000.
4. William R. Cook. A denotational semantics of inheritance, Ph.D. thesis, Brown University Tech. Rep. CS-89-33, May, 1989
5. Pascal Costanza. Vanishing Aspects, Workshop on advanced separation of concerns at OOPSLA 2000, Minneapolis, Minnesota, USA, 2000
6. Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995
7. Stefan Hanenberg, Rainer Unland. Concerning AOP and Inheritance. Workshop Aspekt-Orientierung der GI-Fachgruppe 2.1.9, Paderborn, May, 2001
8. Stefan Hanenberg, Boris Bachmendo, Rainer Unland. An Object Model for General-Purpose Aspect-Languages. To appear in: Proceedings of the Third International Conference on Generative and Component-Based Software Engineering, Erfurt, September, 2001
9. Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, William G. Griswold. An Overview of AspectJ. Appears in ECOOP 2001
10. 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
11. Wolfgang Pree. Design Patterns for Object-Oriented Software Development, Addison-Wesley, 1995
12. Antero Taivalsaari. On the Notion of Inheritance. In: ACM Computing Surveys, Vol. 28, No. 3, pp. 439-479, 1996
13. Peter Wegner, Stanley B. Zdonik. Inheritance as an incremental modification mechanism or what Like is and isn't like. Proceedings of ECOOP '88, LNCS 276, Springer-Verlag, pp. 55–77, 1988