

# Declarative Aspect Composition

István Nagy

Lodewijk Bergmans

Mehmet Aksit

TRESE group, Dept. of Computer Science, University of Twente  
 P.O.Box 217, 7500 AE, Enschede, The Netherlands  
 +31-53-489{5682, 4271, 2638}

{nagyst, bergmans, aksit}@cs.utwente.nl

## ABSTRACT

Aspect-oriented languages provide means to attach certain program units (e.g. advice, filters) to a given set of join points. It is possible that not just a single, but several units need to execute at the same join point. Aspects that specify the insertion of these units are said to "share" the same join point. Such shared join points may give rise to several issues, such as determining the exact execution order and the dependencies among the aspects. In this position paper, we outline a declarative approach that addresses this problem. We evaluate it with respect to several software engineering properties, in particular comprehensibility, predictability and evolvability.

## 1. Introduction

The so-called *join point model* is an important ingredient of every AOP language [1]. It defines certain interaction points ("hooks") where the original behaviour of the program can be modified or enhanced, typically by *superimposing*, or *weaving*, additional behaviour.

Various AOP languages use different types of unit (for example advice, filter or method) to specify additional behaviour to be executed at a given join point. However, it is possible that not just a single, but several units execute at the same join point. For this category of join points, we use the term *shared join point*, since they are shared among units defined by several aspects<sup>1</sup>.

The composition of multiple aspects at the same join point raises several questions, such as: What is the execution order of the units? Is there any other type of dependency between them? These questions are not specific to certain AOP languages but they are relevant for almost every AOP language.

This paper presents a novel, generic approach to specify the composition of aspects at shared join points in aspect-oriented programming languages. This approach is based on declarative specifications of both ordering and control constraints among aspects. In the following section (2), we will introduce an example, and use this to explain the problems of composing aspects at shared join points in detail.

The paper continues in section 3 by presenting the generic core model for specifying aspect composition; section 4 discusses the above mentioned software engineering properties. Finally, in section 5 we close this paper with conclusion.

## 2. Problem Statement

The superimposition of an advice on a particular join point involves several issues. To present the problems that can occur at a shared join point in detail, we introduce an example application. This example will be used throughout the paper.

### 2.1 The Example

The example consists of a simple personnel management system. The *Employee* class, shown in Figure 1, forms an important part of the system. In particular, we will focus on the method *increaseSalary()*, which uses its argument to compute a new salary.

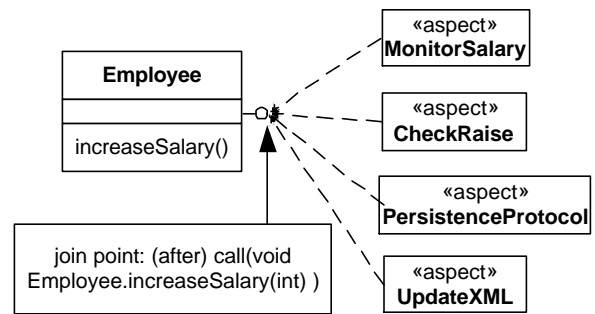


Figure 1. The Employee Class and its superimposed aspects

Our example has been constructed as a scenario that introduces new requirements at each step. Applying the principle of separation of concerns we implement each of these requirements by aspects that will be superimposed on the same join point (as well as others): *after* the execution of the method *increaseSalary()* of the *Employee* class.

Note that this is a particular example that we will implement in AspectJ only for illustrating the problems. There may be other possible implementations, and other AOP languages could have been used as well.

In this example we compose, one by one, five aspects with the *Employee* class. Each of them will be superimposed on the same join point<sup>2</sup>. In each step we show the possible problems that can occur at the shared join point. We present an analysis of these problems and formulate the requirements towards the solution of the problem.

<sup>1</sup> We will use the terminology "composition of aspects" in this paper.

<sup>2</sup> Note that not every aspect will be superimposed on the same set of join points. However, for all aspects there is a common join point which can be designated by the pointcut "`call(void Employee.increaseSalary(int))`" in AspectJ.

## 2.2 Step 1 – Monitoring Salaries

As a first step, the company introduces a logging system to monitor the change of salaries. This feature is implemented by the `MonitorSalary` aspect in Figure 2. This aspect prints a notification whenever a salary has been changed. This could include information about the employee and the type of salary change.

```
public aspect MonitorSalary{
    ...
    after(Employee person, int level):
        salaryChange(person, level){
            System.out.println("Update occurred");
            ... }
}
```

Figure 2. The advice of the `MonitorSalary` aspect

## 2.3 Step 2 – Persistence

The second requirement states that certain classes of the application should store their state in a database. The database should be updated, as soon as possible, after each state change in the object. To keep persistence separate from the application model, we use aspects to realize this requirement<sup>3</sup>.

The abstract `PersistenceProtocol` aspect contains the advice that performs the update of a persistent object:

```
public abstract aspect PersistenceProtocol
    pertarget (target(PersistentObject)){
    abstract pointcut
        stateChange(PersistentObject po);

    after(PersistentObject po): stateChange(po){
        System.out.println("Updating DBMS...");
        po.update();
        ... }
    ... }
```

Figure 3. The abstract `PersistenceProtocol` aspect

The following aspect definition applies the abstract `PersistenceProtocol` aspect to the `Employee` class:

```
public aspect EmployeePersistence extends
    PersistenceProtocol{
    /* The Employee class will implement the
     * PersistentObject interface */
    declare parents:
        Employee extends PersistentObject;

    pointcut stateChange(PersistentObject po):
        call(void Employee.increaseSalary(int))
        && target(po) && ... ;
    ... }
```

Figure 4. An implementation of `PersistenceProtocol`: `EmployeePersistence`

These two aspects together implement persistence for the `Employee` class. However, from the perspective of the join point only

<sup>3</sup> There are several issues, such as connection, storage, updating and retrieval that have to be considered when dealing with persistence. However, we will focus on updating here, because the other issues do not raise additional problems. More details about implementing persistence by aspects can be found in [5].

`PersistenceProtocol` is important for us. If the data of a persistent object changes the corresponding information should be updated in the database too (Figure 3, the advice of the aspect). Changes to the state of the object are captured by the `stateChange(PersistentObject po)` pointcut designator, which is implemented in `EmployeePersistenceProtocol`.

Note that the `MonitorSalary` aspect from the previous step and the `EmployeePersistenceProtocol` have been superimposed on the same join point. In general, aspects can be developed independently. However, once the aspects are integrated within the same application they can easily affect each other's functionality if they are superimposed on the same join point.

**Problem:** Because the database needs to be updated as soon as possible after the state change in the object, the advice of the `PersistenceProtocol` aspect has to be executed before the advice of the `MonitorSalary` aspect.

To ensure the required behaviour for the composition, it must be possible to specify the execution order of aspects at a shared join point. Some AOP languages, for example `AspectJ`, provide means to specify precedence between aspects, which implies an execution order. However, this is not true for every AOP language.

## 2.4 Step 3 – Checking Salary Raises

The next requirement states that an employee's salary cannot be higher than his/her manager's salary. Thus, a raise is not accepted if it violates this criterion. This is enforced by the `CheckRaise` aspect:

```
public aspect CheckRaise
    pertarget(target(Employee) ){
    private boolean _isValid;

    before(Employee person, int level):
        MonitorSalary.increase(person, level){
        _isValid = true;
        } // workaround for conditional execution

    after(Employee person, int level):
        MonitorSalary.increase(person, level){
        Manager m=person.getManager();
        if ((m!=null) && (m.getSalary() <=
            person.getSalary() )){
            //Warning message
            System.out.println("Raise rejected");...
            //Undo
            person.decreaseSalary(level);
            //workaround for conditional execution
            _isValid = false;
        }
    }
}
```

Figure 5. The `CheckRaise` aspect

The advice of this aspect (Figure 5) checks the new salary after the `increaseSalary()` method has executed<sup>4</sup>. If the rule is violated, a

<sup>4</sup> An alternative solution could be the prevention of an invalid raise using a *before* advice (as a pre-condition) instead of an *after* advice. However, this is not feasible in all cases; e.g. it is undesirable to repeat complex salary calculations within the advice and the main method.

warning message is printed and the salary is set back to its original value.

**Problem:** Adding the CheckRaise aspect affects the composition; if this aspect fails the PersistenceProtocol aspect should not be executed because the employee's data has not changed. That is, the execution of the PersistenceProtocol aspects depends on the outcome of CheckRaise.

As an example, Figure 6 shows the modified version of PersistenceProtocol. A new if statement has been added to check if the raise was accepted by the CheckRaise aspect.

```
public aspect PersistentProtocol
  pertarget (target(PersistentObject)){

  after(PersistentObject po): stateChange(po){
    if (CheckRaise.aspectOf(
      (Object)po).isValid()){
      System.out.println("Updating DB...");
      po.update(po.getConnection());
    }
  }
}
```

**Figure 6. The modified version of PersistenceProtocol composed with CheckRaise**

Another disadvantage of this solution is that aspects will depend on each other. That is, to realize the expected behaviour of the composition, aspects will need to refer to other aspects directly. The invocation of the isValid member of CheckRaise in Figure 6 is a typical example of such a dependency.

## 2.5 Step 4 – Updating XML Representations

The fourth requirement states that if the database is not available persistence must be implemented with XML files. For each instance of Employee, an XML file is generated. If the regular persistence does not take place (e.g. because of database connection problems), the file must be updated after each state change of the Employee object. This is realized by the UpdateXML aspect in . This aspect has one advice that calls the method that rewrites the XML file if the salary (or other data) changes.

```
public aspect UpdateXML {
  after(XMLPersistentObject po):
    stateChange(po){
      if ((CheckRaise.aspectOf(
        (Object)po).isValid()
        && (!EmployeePersistence.aspectOf(
        (Object)po).isUpdated()))
        po.toXML();
    }
}
```

**Figure 7. The UpdateXMLEmployee aspect**

In this example, XML files should be updated only if the PersistenceProtocol aspect was not able to update the database. This means that UpdateXML should also execute conditionally; only if PersistenceProtocol failed.

The execution of an aspect may depend on the outcome of other aspects. Only if the outcome of these other aspects satisfies a certain criterion, the dependent aspect is allowed to execute. To

avoid workarounds and their shortcomings, language support is needed for expressing this type of dependencies explicitly.

## 3. Core Model

The problem of shared join points is general to AOP languages. For this reason, we propose a generic solution model that can be built into various AOP languages. The aim of this model is not to be a formal foundation, but to present our approach in a concrete, language-independent way. This requires a set of minimal and sufficiently generic assumptions about AOP languages. Section 3.1 presents these assumptions, section 0 presents *composition constraints* as a means to specify composition of aspects at shared join points.

### 3.1 Basic Entities

We want to be able to map our core model to most AOP languages. It is important that we make only a few assumptions about these elements, since every assumption that we make, potentially restricts the set of AOP languages that we can map to. We focus –briefly– on two key elements of AOP models: joinpoints and the units of behavioural enhancements (cf. ‘advice’).

Our join points are points in the execution of the program either *at the place* of a primitive operation or *between* two primitive operations of the language. This is different from the AspectJ model, where only the operations, for example method calls, are themselves the join points. In AspectJ, advices must be attached *before*, *after* or *around* a join point; in our model, we distinguish these as three different join points<sup>5</sup>.

The additional behaviour that can be inserted at join points, is represented as an *action* in our model. An action has a name that identifies the action itself, and can have a return value. For the purposes of our model, we are only interested in Boolean return values. These typically indicate success (*true*) or failure (*false*) of the action<sup>6</sup>. By default, every action assigned to the join point will be executed, unless specified otherwise. The execution of actions is sequential<sup>7</sup>, that is, only one action can be executed at the same time. In the absence of ordering constraints, the order of the aspects to be executed is undefined; in this case, the programmer should be warned about possibly unexpected orderings

### 3.2 Constraints

Our solution model for composing aspects at shared join points is based on declarative specifications of constraints. Constraints define

<sup>5</sup> Note that language design issues such as ease of expression play no role here, since our model can be used regardless of the join point model shown at the language level.

<sup>6</sup> This issue is beyond the scope of this paper, but a key reason for this restriction to Boolean values is that it guarantees uniform interfaces between the actions; allowing for more freedom in choosing return types would create undesired dependencies between actions.

<sup>7</sup> Parallel execution is an orthogonal issue, unless synchronization between actions is needed. In this paper we focused on the sequential execution of aspects.

<sup>12</sup> We have identified other relevant control constraints, but limit the discussion in this paper to the principal cond constraint.

dependencies between actions. We distinguish two main categories of constraints: *ordering constraints* and *control constraints*. Ordering constraints specify a partial order upon the execution of a set of actions. Control constraints affect the execution order as well; however, they also specify conditional execution of actions.

For each constraint we will also distinguish a *hard* and a *soft* version. These two versions originate from the fact that an action does not necessarily have to execute. Soft constraints ‘tolerate’ the absence of an action. This can be important to achieve open-ended specifications. Hard constraints have stricter execution policies since they aim at ensuring the presence of an action. This may be important for the sake of safety and correctness. In the following section we will show how these constraints resolve the issues that we have identified in the problem analysis.

### 3.2.1 Ordering Constraints

Ordering constraints specify partial ordering among actions. When several actions have been superimposed upon the same join point, all these actions are assumed to execute once, in an arbitrary order. By applying ordering constraints the number of possible orders can be decreased. It is possible to reduce the number of possibilities to 1 by applying a sufficient number of ordering constraints. For example, assume that four aspects, namely MonitorSalary (M), PersistenceProtocol (P), CheckRaise (C) and UpdateXML (U), are superimposed on the same join point as shown in section 2. Without any ordering constraints, the number of possible execution orders is  $4!=24$ .

#### Constraint *pre*

The *pre* constraint specifies that one action should precede another action in the execution at a shared join point. The definition of the *pre* constraint is the following:

$pre_{soft}(x,y)$  – The order of actions is such that *x* can never be executed after the execution of *y* has occurred. (The two actions do not have to follow each other directly; other actions can be executed between them.) Besides, *y* is allowed to execute if *x* did not execute at this join point.

$pre_{hard}(x,y)$  – The order of actions is such that *x* can never be executed after the execution of *y* has occurred. (The two actions do not have to follow each other directly; other actions can be executed between them.) Besides, *y* can be executed *only* if *x* has been executed at this join point.

We use Figure 8 to illustrate the behaviour of the constraints which were applied on two actions called *x* and *y*. The topmost row of the table shows the applied constraints (currently, only the columns of *pre* are important for us). The leftmost column lists the possible values (true, false and void) that the *x* action can have after its execution. The last item is the case when the *x* action has not been executed for some reason. A cell of the table indicates if *y* is allowed to execute after the execution of *x*, according to the applied constraint and return value of *x*. We can see in this figure that the  $pre_{soft}$  constraint is not influenced by the return value: in each case *y* can execute after *x* executed. The last cell shows the case that *y* can execute even if *x* has not been executed. However, the next cell of  $pre_{hard}$  shows that *y* cannot execute in the same case.

	$pre_{soft}(x,y)$	$pre_{hard}(x,y)$	$cond_{soft}(x,y)$	$cond_{hard}(x,y)$
<b>x: true</b>	<b>y</b>	<b>y</b>	<b>y</b>	<b>y</b>
<b>x: false</b>	<b>y</b>	<b>y</b>	-	-
<b>x: void</b>	<b>y</b>	<b>y</b>	-	-
<b>x did not run</b>	<b>y</b>	-	<b>y</b>	-

Figure 8. The execution semantics of the composition constraints

In Figure 9 we demonstrate how the *pre* constraint works and decreases the number of possible orders. We use the case that we introduced in section 2 again, but show only the first letters of the actions. We assume that all four actions (C, P, M, U) are superimposed upon the same join point. In the middle column we list the constraints applied, while in the right column we list all the possible orders which are valid, given those constraints. In the first row (Case I.) we apply only one constraint specifying that PersistenceProtocol should be executed before MonitorSalary. The last six possible orders of Case I. are those cases where the execution of PersistenceProtocol and MonitorSalary are interleaved with other actions (C and/or U).

Case	Constraints	Possible Orders
I.	$pre_{soft}(P, M)$	<b>PMCU, CPMU, CUPM, PMUC, UPMC, UCPM   PCMU, PCUM, CPUM, PUMC, PUCM, UPCM</b>
II.	$pre_{soft}(P, M), pre_{soft}(P, U)$	<b>CPMU, CPUM, PCMU, PCUM, PMUC, PMCU</b>
III.	$pre_{soft}(P, M), pre_{soft}(P, U), pre_{soft}(C, P)$	<b>CPMU, CPUM</b>

Figure 9. The possible orders are decreasing as new constraints are added.

In Case II. we add a new *pre* constraint. The new constraint specifies that PersistenceProtocol should precede UpdateXml as well. By applying two ordering constraints, the number of valid orders will be reduced to six in this case. In the third row (Case III.) there are only two alternatives left, after applying three constraints. In this case only the order between MonitorSalary and UpdateXml is not fixed.

### 3.2.2 Control Constraints

Control constraints express conditional execution dependencies between actions. The general form of a control constraint is the following: “*Constraint(Condition, ConstrainedAction)*“. The *Condition* is represented by an action. Control constraints use the return value of executed actions for constraining the execution of *ConstrainedAction*. For each type of control constraint we assume that  $pre_{soft}$  relationship also holds between the arguments of the control constraints; this is motivated by the fact that the return value of *Condition* can only be used when *Condition* executes before *ConstrainedAction*.

#### Constraint *Cond*

In this paper we only discuss a single constraint type<sup>12</sup>; the *cond* constraint specifies that an action is conditionally executed according to the return value of another action. The definitions of the soft and hard version of the *cond* constraint are:

$cond_{soft}(x,y)$  – Action  $y$  can execute if  $x$  returns true or did not execute. That is,  $y$  will not execute if  $x$  returns either false or void.

$cond_{hard}(x,y)$  – Action  $y$  can execute *only* if  $x$  returns true. That is,  $y$  will not execute if  $x$  returns false or void, or if  $x$  does not execute.

The last two columns in Figure 8 illustrate how the behaviour of the two alternatives differ. For the true and false values both constraints work in the same way:  $y$  can execute only if  $x$  succeeded (i.e.  $x$  returned true). However, when  $x$  is not executed,  $cond_{soft}$  allows for the execution of  $y$ , while  $cond_{hard}$  does not allow this<sup>13</sup>.

Case	Constraints	Possible Orders		
		C: false	C:true	C did not run
IV.a	$pre_{soft}(P, M)$ , $pre_{soft}(P, U)$ , $cond_{hard}(C, P)$	<i>CPMU</i> , <i>CPUM</i>	<i>CMU</i> , <i>CUM</i>	<i>MU</i> , <i>UM</i>
IV.b	$pre_{soft}(P, M)$ , $pre_{soft}(P, U)$ , $cond_{soft}(C, P)$	<i>CPMU</i> , <i>CPUM</i>	<i>CMU</i> , <i>CUM</i>	<i>PMU</i> , <i>PUM</i>

Figure 10. Using the *cond* constraint.

The effect of the *cond* constraint is demonstrated in Figure 10. In Case IV.a we have changed the third constraint of Case III to  $cond_{hard}(CheckRaise, PersistenceProtocol)$ . Depending on the return value of *CheckRaise*, there are two sets of possible orders. When *CheckRaise* returns true (the first column of Possible Orders) the possible orders are the same as by applying the  $pre_{soft}$  constraint. However, when the return value of *CheckRaise* is void or false (the second column of Possible Orders) *PersistenceProtocol* will not be executed. The right-most column, “C did not run”, shows the difference between the hard and the soft versions: in case of  $cond_{soft}$ , *PersistenceProtocol* executes, even though *CheckRaise* has not been executed.

## 4. Software Engineering Properties

### 4.1 Comprehensibility

We define comprehensibility as the ability to understand the execution of a program and the collaboration among modules (units) of a program from the source code. Comprehensibility can be influenced by several language properties, such as the modularization of units, how these units refer to each other and where the specification of the references is placed.

Using composition constraints, the dependencies between aspects are expressed explicitly; thus, the composition of aspects can be traced back easily. Without constraints, these dependencies must be hard-wired into the body of aspects (and/or advices), which renders more difficulties in the understanding of the program.

It is also important to consider the modularization of composition constraints. If the specification of constraints is distributed over several modules (i.e. it is too fragmented), the comprehensibility of

<sup>13</sup> For the *cond* constraint, a boolean return value is desired. Hence if strong typing is applied to the return values of actions and the arguments of constraints, the void case (which is also used for all non-boolean return values) can be avoided. We deliberately included void return values as a legitimate case to make the system more flexible and applicable to a wide range of languages; either with, or without strong typing.

the program can decrease. On the other hand, if all constraints are centralized without sufficient organizational structure, this may scale up badly.

Briefly, composition constraints may both increase and decrease the comprehensibility of the composition of aspects depending on their usage. For this reason, a language should support alternative modularizations of constraints. For example the aspectJ *declare precedence* construct allows this. Our model can be mapped to languages that support this.

### 4.2 Evolvability

Evolvability is a software engineering property that helps in developing programs in an incremental way. In other words, it facilitates extending an application with new requirements, mostly by reusing previously written modules without modifying them.

In a constraint specification (using soft constraints) it is possible to refer to aspects which are not necessarily present in the system. In this case, the specification is not taken into account when the environment evaluates it. Whenever those aspects are defined by the developer and become present, the constraint specification will automatically apply to them. Thus, aspects can refer to other aspects which will be integrated with the system later. In addition, when these aspects are removed, the system will not collapse. (For this reason, we call these specifications *open-ended* specifications.) This is an important property of our model, since aspects can be developed and deployed independently from each other.

The fact that constraint specifications can be placed in any module also has a positive impact on evolvability, since it allows for the composition of aspects that are developed independently. Finally, reuse of modules, e.g. through inheritance, should incorporate the reuse of constraints. This is a language design issue that is independent from our proposed model.

### 4.3 Predictability

Predictability ensures programmers that certain properties are hold during the entire execution of a program. For programming languages there is a wide range of features that supports predictability. A simple one, for example, is to build a constraining mechanism into the language in the form of keywords, such as the *final* keyword in Java. An important technique to support predictability is to enhance the language with constructs (e.g. interfaces, abstract types, etc.) that allow for the specification and definition of contracts. In this case, the language also needs to have features, usually built into compilers, that guarantee these contracts (e.g. type-checking and other static analysis facilities). Typical examples are the *declarative completeness* in HyperJ [4] and *aspect collaboration interfaces* in Caesar [3]. In both cases the composition (i.e. weaving) is based on interfaces that act as contracts between the components to be composed together.

Control constraints have a slight negative impact on predictability. As stated in the previous section, aspects can refer to aspects that will only be integrated with the system later. Thus, a programmer can add a new aspect to the system without knowing what constraint specifications apply to that aspect. Tools might help in detecting unintended compositions and avoiding conflicts.

On the other hand, using composition constraints, designers can ensure that when certain aspects are added later to the system, they

will be integrated in predefined way. This affects predictability in a positive way.

#### 4.4 Semantic Interactions

As long as a language has no additional mechanisms or techniques that affect composition of aspects at shared join points, no semantic interactions are to be expected. In fact, our proposed mechanism explicitly *defines* how aspects must interact.

### 5. Conclusion & Discussion

Shared join points are not a new phenomena, nor specific to any AOP languages. To the best of our knowledge, shared join points have not been explicitly analysed in the literature before. In this paper, first we showed a motivating example on the issues that arise when multiple aspects are superimposed at a shared join point. We outlined our solution, which is a constraint-based, declarative approach to specify the composition of aspects. Finally, we discussed how our model affects comprehensibility, evolvability and predictability of software.

### 6. REFERENCES

- [1] T. Elrad, M. Aksit, G. Kiczales, K. Lieberherr, H. Ossher, "Discussing Aspects of AOP", *Communications of the ACM*, Volume 44, Issue 10, October 2001.
- [2] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm & W. Griswold, "An Overview of AspectJ", in *Proceedings of ECOOP 2001*, LNCS 2072, Springer Verlag, 2001
- [3] M. Mezini & K. Ostermann, "Conquering Aspects with Caesar", in *Proceedings of the 2<sup>nd</sup> international conference on Aspect-oriented software development*, Boston, Massachusetts, 2003.
- [4] H. Ossher , P. Tarr, "Multi-Dimensional Separation of Concerns and the Hyperspace Approach", in *Software Architectures and Component Technology: The State of the Art in Research and Practice*, M. Aksit (Ed.), Kluwer Academic Publishers, 2001
- [5] A. Rashid, R. Chitchyan, "Persistence as an Aspect", in *Proceedings of the 2<sup>nd</sup> international conference on Aspect-oriented software development*, Boston, Massachusetts, 2003.