

A Meta Model and Modeling Notation for AspectJ

YAN Han*
Dept. of Computer Science
Nanjing University of Science and
Technology
210094, Nanjing, China

yanhan@njust.edu.cn

Günter Kriesel
Dept. of Computer Science III
University of Bonn
Römerstr. 164, D-53117 Bonn,
Germany

gk@cs.uni-bonn.de

Armin B. Cremers
Dept. of Computer Science III
University of Bonn
Römerstr. 164, D-53117 Bonn,
Germany

abc@cs.uni-bonn.de

ABSTRACT

In order to bridge the gap between conceptual modeling and aspect oriented programming, many researchers have proposed extensions of the UML towards graphical notations for aspects. However, notations without an underlying formal semantics and syntax are not amenable to automated tool support. Tool-supported aspect modeling requires an *aspect meta model* as a basis.

In this paper we follow a bottom-up approach, focusing on a meta model for AspectJ. By tailoring UML meta classes, a Java meta model is built first and extended into an AspectJ meta model. The semantics, attributes and associations of the main aspect-related meta classes are specified. These meta classes are visualized by a set of corresponding notations, and three structural views are suggested for aspect related diagrams. The meta model is precise, pragmatic and enables implementation of aspect modeling tools. The work promises to narrow the gap between conceptual modeling of aspects and concrete implementations in AspectJ.

Keywords

aspect, modeling, meta model, AspectJ, UML

1. INTRODUCTION

In large and complex systems many concerns are interwoven. Aspect Oriented Programming (AOP) allows untangling of these concerns at the implementation level using novel languages, such as AspectJ [24], HyperJ [25], AspectC [26], AspectC++ [27], AspectC# [28], AspectR [29], AspectS [30], ComposeJ [19], LogicAJ [12], etc. AOP languages, however, need to be complemented by suitable aspect modeling concepts and tools [9, 17]. This entails the challenge to specify, visualize, construct and document aspect related artifacts. Many previous approaches focus on graphical notations for aspects. However, aspect modeling notations without an underlying formal semantics and syntax are not amenable to automated tool support. Tool-supported aspect modeling requires an *aspect meta model* as a basis.

Definition of an all-encompassing aspect meta model would be premature and counterproductive at a time when the state of the art in AOP is still characterized by a wealth of rapidly evolving concepts, languages and systems. In this paper, we therefore follow the bottom-up approach of [9], focusing on a *meta model for AspectJ* [11,24]. The meta model can be used for forward and reverse engineering between AspectJ models and AspectJ programs. The choice of AspectJ is motivated by its wide popularity, mature language design, industrial-strength tool support and by its nature as an extension of Java. A meta-model

for AspectJ necessarily includes a meta-model for Java that may be beneficial for the many other languages built as extensions of Java.

We aim at a meta model that satisfies the following requirements: 1) *Standard-compliance*: It should build on existing standards for meta modeling. 2) *Extensibility*: It should be easy to extend in order to support tools for many other extensions of Java. 3) *Minimality*: It should be as small and simple as possible, in order to ease implementation of related tools. 4) *Correctness and completeness*: It should faithfully model all structural concepts of Java and AspectJ. 5) *Visualized*: Graphical notations should be a part of the meta model for visual aspect modeling. 6) *Amenable to formal reasoning*. It should enable reasoning about semantics, syntax, notations, aspect structure, crosscut architecture, and crosscut effects. Reasoning also requires that every part is consistent with the others.

Regarding standards compliance and extensibility, an aspect meta model should certainly be based on the Meta Object Facility (MOF), the Object Management Group (OMG) standard, specifying how to define, interchange and extend meta models [23]. On one hand, the OMG promotes UML for modeling meta models and uses it in the definition of the MOF [6]. On the other hand, UML is itself based on the MOF [21, 22].

Because the latest UML version (2.0) [21, 22] does not support aspect modeling, different extensions to the UML have been proposed. Some researchers extend UML with generic crosscut elements [3,17]. Others extend it towards particular languages, such as AspectJ [1, 15, 16, 18]. All these approaches use the UML extension mechanisms, based on *stereotypes*, *tags*, and *constraints*.

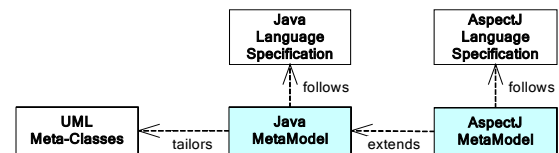


Figure 1. The AspectJ meta model extends a Java meta model built by tailoring UML meta classes

In this paper we follow a different approach, motivated by our minimality requirement. Extensions of UML are complex to implement – if we are just interested in a Java and AspectJ meta model, much implementation effort will be wasted on unneeded generality. A self-contained Java and AspectJ meta model is smaller and easier to implement. This is a big advantage, if we are interested in building a dedicated tool supporting AspectJ, rather than extending an existing UML CASE tool. The main functions of an aspect modeling tool (visual representation of crosscutting,

*This paper describes research done during the author's visit at University of Bonn, Germany.

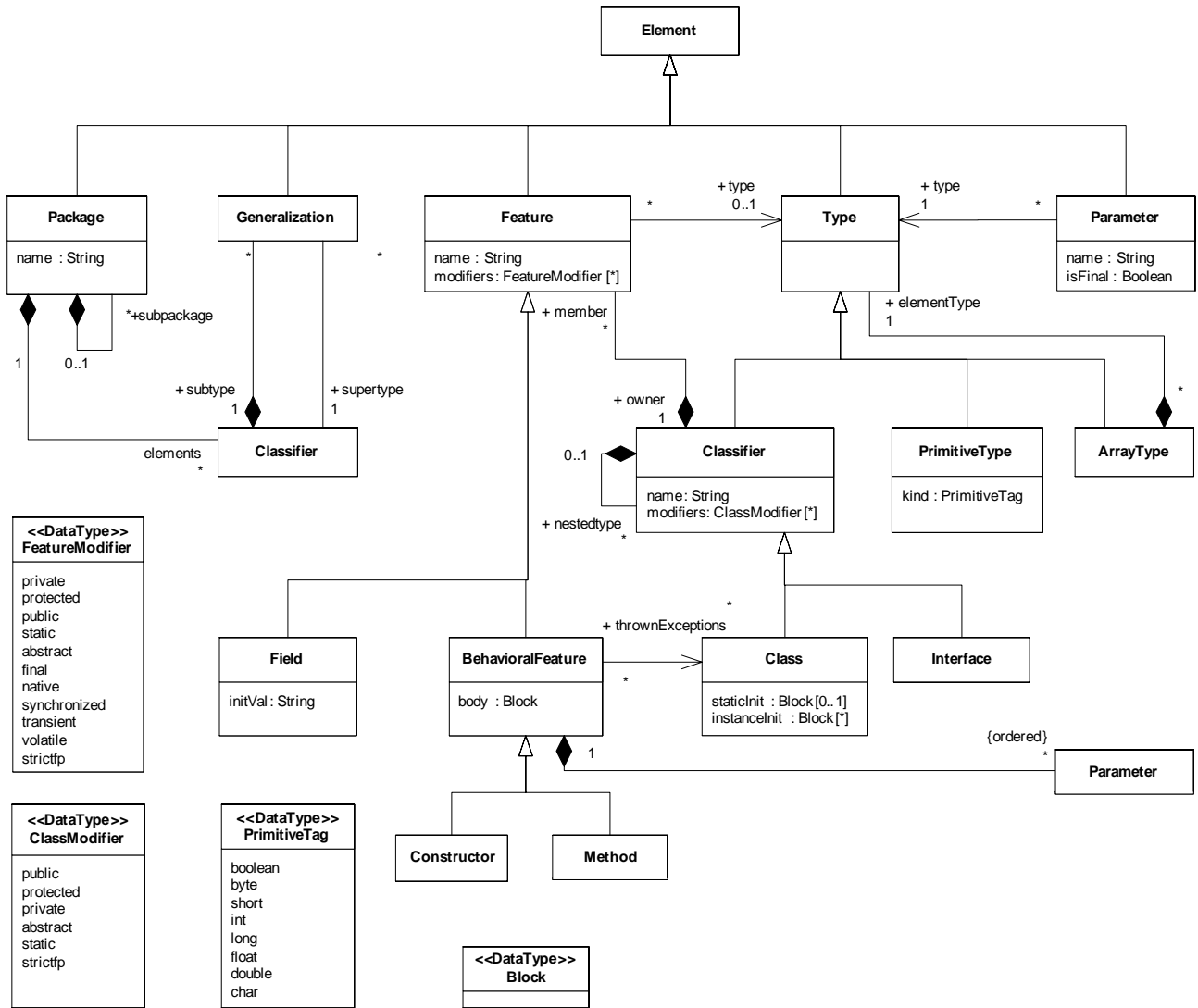


Figure 2. The Java meta model is built tailoring UML meta classes

code generation, and reverse engineering) can be performed on the simple meta model. For interoperability with other tools, it suffices if there is a mapping of the meta model to UML. However, as long as there is no *standardized* aspect meta model or aspect UML profile, interoperability is not an option, anyway. So for the time being, the arguments for a self-contained AspectJ meta model have most to recommend them.

The remainder of the paper is structured as follows. Section 2 introduces briefly the Java meta model. Section 3 specifies the five main meta classes of the AspectJ meta model. Section 4 suggests a set of graphical notations for these meta classes and three views for aspect diagrams. A modeling example using our notation is illustrated in Section 5. Section 6 discusses our proposal and section 7 discusses related work. Section 8 concludes.

2. THE JAVA META MODEL

As AspectJ is an extension of Java, we introduce first a meta model for Java, which is then extended to one for AspectJ. The Java meta model expresses the static structure of the Java

language (Figure 2). Readers are assumed to be familiar with Java and UML. Therefore, the model specification from Figure 2 is explained only briefly.

The Java meta model is not an extension but a leaner version of the UML meta model. It is built using the MOF [23] and tailors UML meta classes [22] to the Java 2 Specification [7], eliminating irrelevant generality (unnecessary attributes and associations). Most of the 15 remaining meta classes use the same names as the corresponding UML meta classes. Only 4 use Java specific names: ArrayType corresponds to MultiplicityElement in the UML meta model, Field to Property, Constructor and Method to Operation. All Java meta classes have the same semantics and graphical notation as the corresponding UML meta classes (up to eliminated attributes, associations and associated constraints). Some of them should preferably be implemented as abstract classes, such as Element, Type, Classifier, Feature and BehavioralFeature.

The Generalization class abstracts the extends and implements relationships between classifiers (classes and/or interfaces). An

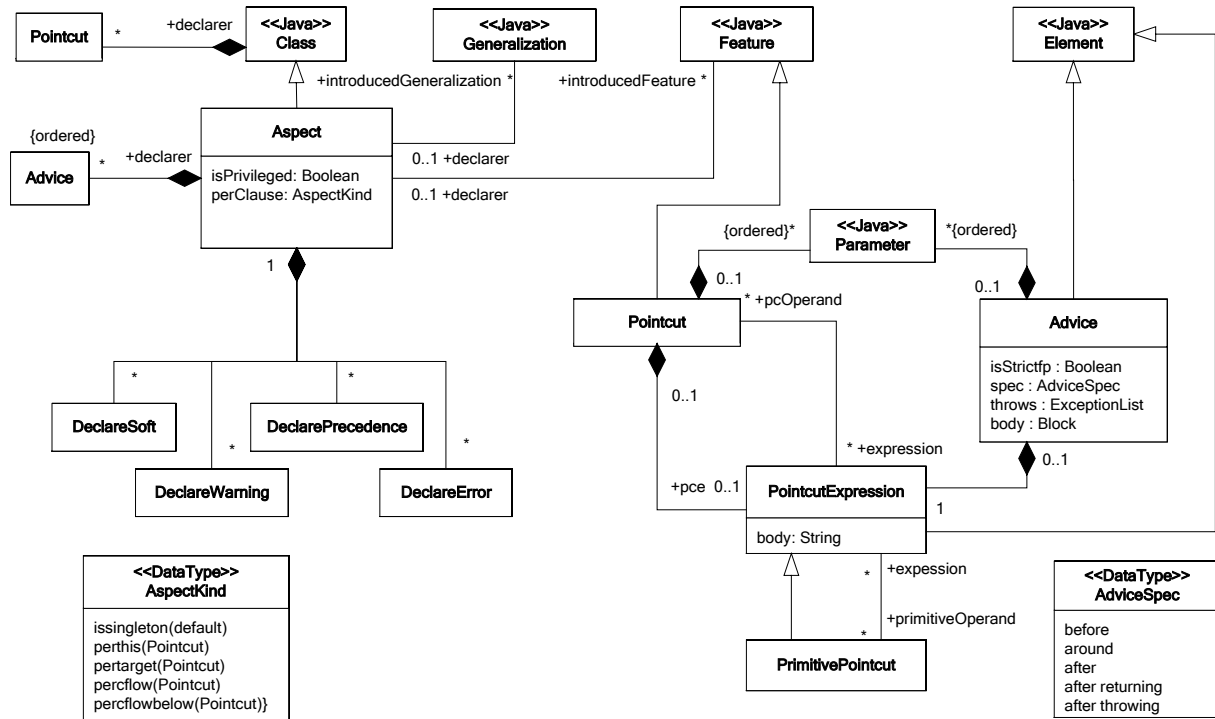


Figure 3. The AspectJ meta model. The AspectJ meta classes extend and adapt the Java meta classes

instance of it represents an inheritance relationship between a subtype and a supertype.

The association between Classifier and Feature specifies that a classifier (class or interface) may declare a set of features as its members, such as fields, constructors and methods.

A sequence of statements is abstracted as a data type Block. It is used to model the static initializer and instance initializer of a Class, as well as the body of a BehaviouralFeature. The Java meta model does not get into details on blocks. A block should be modeled using behavioral UML elements, such as state machines, interaction diagrams, etc.

The Java meta model can be used on its own for Java program modeling. It has been tested with a part of the tutorial programs from the Java 2 SDK Standard Edition, version 1.3 and 1.4.

3. THE ASPECTJ META MODEL

The AspectJ meta model is shown in Figure 3. It extends the Java meta model according to the AspectJ specification [23]. AspectJ adds to Java the concept of pointcut, advice, inter-type declaration and aspect. Pointcuts and advice affect program flow, inter-type declarations affect a program’s class hierarchy and features, and aspects encapsulate these new constructs.

To model the principles of AspectJ, we first explain the concept of *crosscut target and weaving configuration*.

Crosscut target. A type is a *structural crosscut target* if it is extended with new super-types or features by some aspect’s inter-type declarations. A type is a *behavioral crosscut target* if its behavior is modified by some aspect’s advice. In either case, the type is a *crosscut target*. Note that an aspect may also be a target.

Weaving configuration. The weaving configuration determines the relationships of a set of aspects crosscut a set of targets. There are two weaving configurations, one for the compiler *ajc* and one for the class loader *aj*. When the compiler is invoked, it gets its configuration from the command line. When *aj* is used to load a class, the class is woven with the aspects from the JARs specified in the environment variable ASPECTPATH.

The AspectJ meta-model expands three Java meta classes, Class, Generalization and Feature, with new associations (Figure 3). It adds five new meta classes: Aspect, PointcutExpression, PrimitivePointcut, Pointcut and Advice (Table 1). Their semantics, attributes and associations are specified below. For ease of presentation, related constraints will be explained informally rather than in Object Constraint Language (OCL).

Table 1. The major elements and their meta modeling

Language elements	Meta modeling for AspectJ
aspect	A subclass of <i>Class</i> from Java meta model
declare parent	An association (<i>introducedGeneralization</i>) between <i>Aspect</i> and <i>Generalization</i>
inter-type member	An association (<i>introducedFeature</i>) between <i>Aspect</i> and <i>Feature</i>
pointcut	A subclass of <i>Feature</i> , associated with <i>Class</i>
pointcut expression	A subclass of <i>BooleanExpression</i> . <i>PrimitivePointcut</i> is a subclass of it.
advice	A subclass of <i>Element</i> from Java meta model

3.1 The Class “Aspect”

3.1.1 Semantics

An aspect modularly expresses a concern that crosscuts multiple types or features. In the AspectJ meta model Aspect is a subclass of Class from the Java meta model, extended with crosscutting elements (Figure 3).

3.1.2 Attributes

Aspect inherits the features and relationships of Class with unmodified meaning. It adds the following attributes and constraints:**isPrivileged: Boolean.** If true, the aspect code may access private members of target classes. Default is false.

- **perClause: AspectKind.** It declares how the aspect is instantiated and associated. A constraint is that user code cannot instantiate an aspect type. The 5 legal values are defined in the data type AspectKind.

3.1.3 Associations

As a subclass of Class, an aspect is also a Classifier and a Type. It inherits their features, associations and constraints with almost unmodified meaning. It adds the following attributes and constraints:

- **pointcut: Pointcut[*].** A class may declare a set of pointcuts. Aspect inherits this ability.
- **introducedGeneralization: Generalization[*].** The association models a category of inter-type declarations. The Generalization class is extended with an association to Aspect. An aspect may declare Generalization instances for other types making them own the generalization thereby owning new supertypes. The subtype is the introduction *target*. The introduced superclass must be a subclass of the original superclass of the target.
- **introducedFeature: Features[*].** IntroducedFeature is an inter-type declaration. It extends Feature with an association to an aspect. An aspect may declare Feature instances for other type. The owner type is the introduction *target*. If a feature links with an aspect, it is an introduced feature; else it is a self-declared one. If the target is an interface, only non-static features can be introduced to it.
- **advices: Advice[*].** An aspect may declare a set of advices.

The other four associations from Figure 3 defining inter-type declarations are not explained in detail (e.g. DeclareSoft, etc.). Their semantics is straightforward from the AspectJ language specification.

3.2 The Class “PointcutExpression”

3.2.1 Semantics

A *join point* is a well-defined point in the execution of a program. A pointcut expression is a kind of expression that selects a set of joint points. When a thread reaches a selected join point, required context data is passed and associated advice code is run. The PointcutExpression is modeled as a subtype of Element from the Java meta model.

3.2.2 Attributes

- **body: String.** A string formed by logical operators (! NOT, && AND, || OR) and operands (Pointcuts and Primitive Pointcuts).

3.2.3 Associations

- **pcOperand: Pointcut[*].** An expression may reference many pointcuts as its operands by their names and parameters.
- **primitiveOperand: PrimitivePointcut[*].** An expression may reference more than one primitive pointcuts.
- **pointcut: Pointcut.** A PointcutExpression may be declared in a pointcut.
- **advice: Advice.** A PointcutExpression may be declared in an advice.

3.3 The Class “PrimitivePointcut”

3.3.1 Semantics

The primitive pointcuts are a set of pre-defined pointcut expressions. Each selects particular joint points. The class PrimitivePointcut is modeled as a subtype of PointcutExpression.

A primitive pointcut may pick out joint points across multiple types. Logical operators (! NOT, && AND, || OR) may be used in type patterns. The wildcard (*) may be used to match any sequence of characters in a package or type name. Use of a NOT operator or a wildcard on a type pattern creates *open* (unbounded) join point sets. In such cases only the weaving configuration limits the weaving space.

3.3.2 Attributes

- **body : String.** Inherited.

3.3.3 Associations

- **expression : PointcutExpression[*].** A primitive pointcut may be referenced by arbitrarily many pointcut expressions.

3.4 The Class “Pointcut”

3.4.1 Semantics

A pointcut is a named feature in a class or an aspect declaring 1) its name and visibility 2) a parameter list specifying what data will be exposed from runtime context 3) and if not abstract, a pointcut expression indicating which joint points will be selected. A pointcut is referenced by its name and parameters via its visibility. The purpose of declaring a pointcut is to share its pointcut expression in many advices or other pointcuts.

Pointcut is modeled as a subtype of Feature. Although it is a Feature, it cannot be introduced by any inter-type declarations. While a pointcut contains a list of parameters, its name is unique and it cannot be overloaded, unlike in the case of a behavioral feature. A class that declares pointcuts must be compiled by *ajc* rather than *javac*.

3.4.2 Attributes

- **name : string.** Inherited. A unique string in the declaring type.
- **modifiers: FeatureModifier[*].** Inherited. Legal modifier values are public, private, protected, abstract and final. If a pointcut is abstract it declares no PointcutExpression and expects to be overridden in a subspect by a pointcut with the same name and parameters. If it is final, it cannot be overridden, i.e., subspects cannot declare any pointcut with the same name as this pointcut.

3.4.3 Associations

- **declarer: Class.** The class declaring the pointcut.

- **parameters: Parameter[*].** A pointcut may contain a list of parameters to pass context data to advices.
- **pce: PointcutExpression.** If a pointcut is not abstract, it declares a pointcut expression.
- **expression: PointcutExpression[*].** A pointcut may be referenced by many pointcut expressions.

3.5 The Class “Advice”

3.5.1 Semantics

An advice is an element in an aspect declaring 1) what joint points will be picked out by a pointcut expression; 2) what data will be passed from runtime context to code by a parameter list; and 3) what code will be run when any picked joint points are reached at runtime. Advices perform behavioral crosscuts. If an advice in an aspect crosscuts a target, it is said that the aspect crosscuts the target.

Advice is modeled as an additional subtype of Element.

3.5.2 Attributes

- **isStrictfp: Boolean.** If true, the advice code is strict floating point in the sense of the Java Language Specification [7].
- **spec: AdviceSpec.** The purpose of a spec is to declare when the advice code will run. As defined by the AdviceSpec type in Figure 3, there are 5 legal spec values: before, around, after, after returning and after throwing. Different primitive pointcuts support different specs. For instance, handler(*Exception*) only supports before; set/get(*Field*) supports before, after and around; execution(*MethodOrConstructor*) and call(*MethodOrConstructor*) support all 5 specs.
- **throws: ExceptionList.** Declares what checked exceptions may be thrown by the execution of the advice body.
- **body: Block.** The code block of the advice. It will be run at the specified spec whenever a join point selected by this advices’ pointcut expression is reached during execution.

3.5.3 Associations

- **parameters: Parameter[*].** A list of parameters specifies context data exposed from joint points to the advice.
- **pointcut: PointcutExpression.** A pointcut expression specifies what joint points will be selected and what context data will be exposed.
- **declarer: Aspect.** The aspect declaring this advice.

4. NOTATIONS AND VIEWS

Visual modeling needs graphical notations for the core concepts of AspectJ. A candidate notation that falls out from our meta model is to depict aspects, pointcuts, advices, etc. at instances of the respective meta classes from Section 3. An example how a pointcut would be depicted this way is given in Figure 4(a). This approach is appealing from a tool builder’s perspective, because it requires no modification of the UML. From a user’s perspective, however, a simpler notation by a stereotype that hides the internals of the meta class is desirable (cf Figure 4(b)).

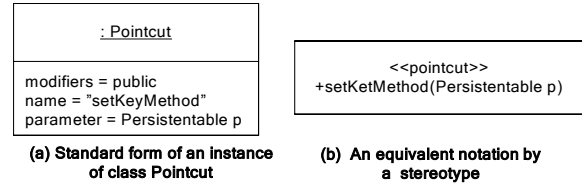


Figure 4. Comparison of two notations for pointcut

In the following, we suggest four stereotype notations for instances of the AspectJ meta classes: <<aspect>> is based on Aspect, <<pointcut>> on Pointcut, <<pce>> on PointcutExpression and <<advice>> on Advice. In spite of the notational simplification, the semantics, attributes and associations of the meta classes are preserved. These notations complement UML static structure diagrams unambiguously and without conflict to existing notations of the UML. Based on our notations, we suggest three views to support multiple perspectives into visual aspect modeling (cf. Section 4.4).

4.1 Aspect Notation

Like classes, aspects can be depicted in folded and unfolded form (Figure 5.1 and Figure 5.2). In unfolded form, aspects contain one additional compartment for intertype declarations, pointcuts and advices.

The fact that an aspect crosscuts a target is shown by letting the aspect overlap the target. An aspect may crosscut many targets (Figure 5.3), including targets shown in other diagrams. This notation provides just an abstract indication of crosscutting without specifying the precise form of crosscutting. If we want to provide more detail we can use the notation for structural or behavioral crosscutting presented below.

Structural crosscutting, such as introduced generalization (Figure 5.4) and introduced feature (Figure 5.5), is modeled as association of an aspect (as declarer) and its target (as owner). Our notation visualizes the relationship between aspects and targets as well as the result of applying aspects to their targets. For instance, in Figure 5.4) the link between the Target Type and the Super Type resulting from the aspect is immediately visible in the diagram.

Behavioral crosscutting is visualized by the notations for Pointcut, PointcutExpression and Advice.

4.2 Pointcut and PointcutExpression Notation

Every pointcut is an instance of the meta class Pointcut. Its visual representation uses the stereotype <<pointcut>> and shows the modifiers, name and parameter list (Figure 5.6). For a non-abstract pointcut, its association with a pointcut expression is shown too.

Every pointcut expression is an instance of the meta class PointcutExpression. Its simplified visual representation uses the stereotype <<pce>> and shows the body of the pointcut expression (Figure 5.6).

4.3 Advice Notation

Advice is a stereotype (Figure 5.7, 5.8 and 5.9). As advice is declared without name, we use its sequence number in its declaring aspect, its spec, a parameter list and a short text explanation to indicate an advice: [n]<spec>(paralist){explanation}. An advice associates with a pointcut expression.

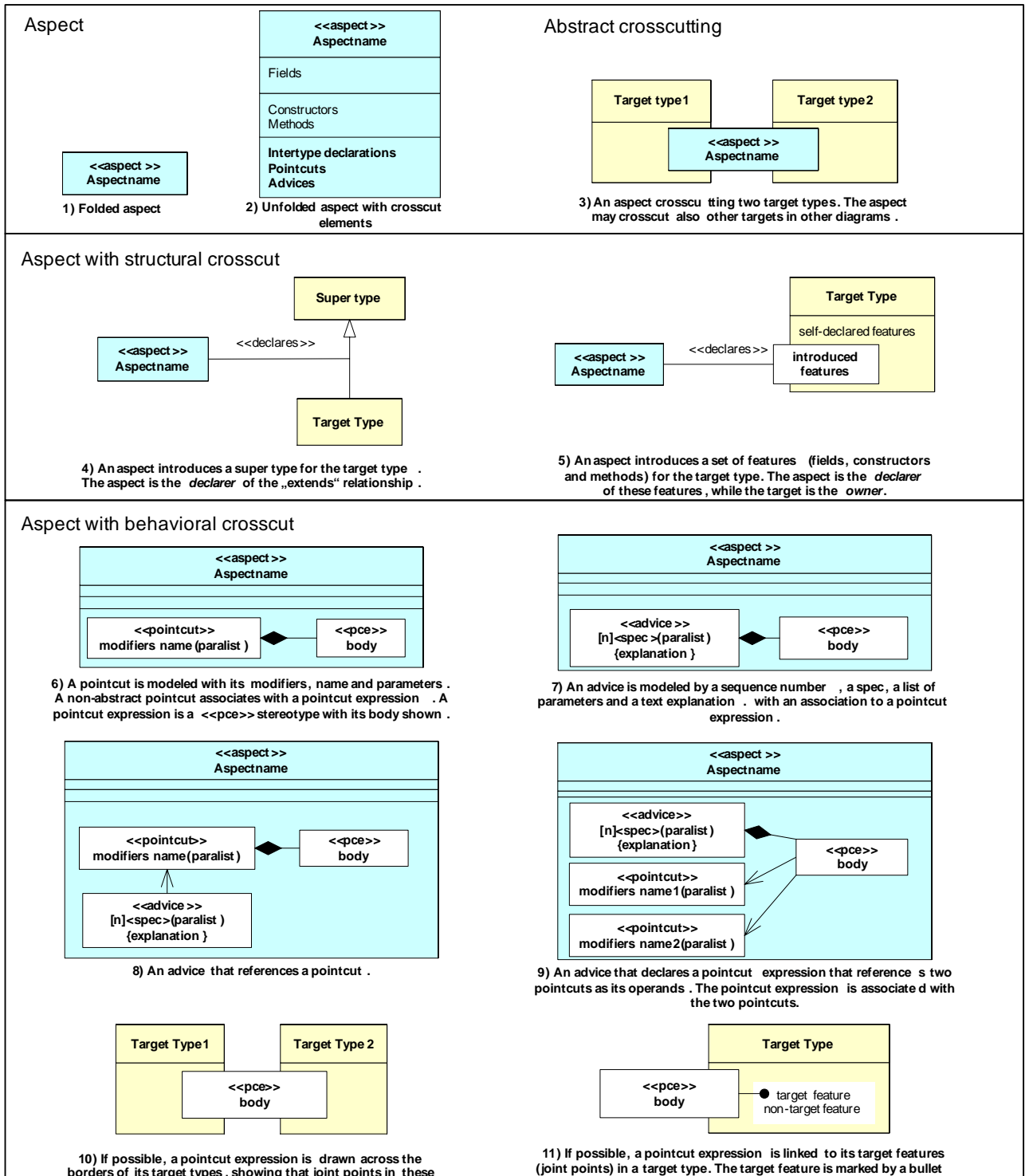


Figure 5. Graphical notation for aspects and crosscuts (abstract, structural and behavioral)

If an advice references a pointcut directly, use a link to the pointcut (Figure 5.8). If the pointcut expression references several pointcuts, use associations to these pointcuts (Figure 5.9).

A pointcut expression that references an abstract pointcut or an *open* join point set (3.3.1) cannot be connected to particular

crosscut targets. Others can be represented as in Figure 5.10 to visualize the crosscut relation to target types. As an additional level of detail the notation from Figure 5.11 can be used to specify individual target features.

4.4 Views

To model aspects from different perspectives, we suggest three views, serving different purposes during development and maintenance (Table 2). Tools should enable focusing a diagram on either of these views and quick switching of views.

Table 2. Three aspect modeling views

Views	Model content	Purpose
Aspect view	Structure and members of a particular aspect	Aspects designing and refactoring
Crosscut view	Relationships between aspects and targets	Concerns separation and aspects analysis
Target view	Crosscut effects on a particular target	Aspects testing and integration

5. A MODELING EXAMPLE

In this section we give an example of applying our aspect notation for modeling the personnel management part of a Management Information System (MIS) containing the classes Department, Employee, etc (Figure 6).

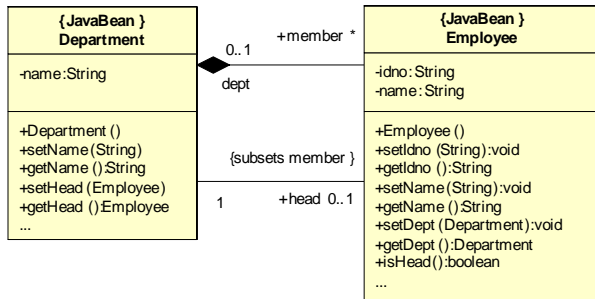


Figure 6. The example: Basic Java classes

Figure 7 is a **crosscut view** showing two independent aspects (PersonnelPersistency and PersonnelConstraints) crosscut the above classes. It additionally shows that the PersonnelPersistency aspect is a subspect of the abstract aspect PersistentAsp.

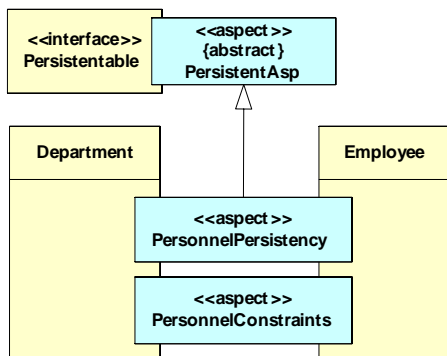


Figure 7. Example: Crosscut view

Figure 9 is an **aspect view** modeling personnel constraints as a non-functional aspect. The corresponding AspectJ code is shown below.

```

before(Department d, Employee newHead):
    execution(void setHead(Employee))&&target(d)&&args(newHead){
        if (newHead!= null && newHead.getDept()!= d)
            newHead.setDept(d);
    }
before(Employee e, Department newDept):
    execution(void setDept(Department))&&target(e)&&args(newDept){
        if (e.isHead() && e.getDept() != newDept)
            e.getDept().setHead(null);
    }

```

Figure 8. Advice code of aspect from Fig. 9

The aspect “PersonnelConstraints” is designed to enforce two constraints on the associations between Employee and Personnel: the head of a department must be a member of the department, and an employee can be a member of only one department. Let d1, d2 be departments and e1, e2 be the respective heads of department. Without constraint monitoring, calling either d1.setHead(e2) or e2.setDept(d1) will violate the constraints. The aspect shown in Figure 9 and 10 enforces these constraints.

The aspects shown in Figure 10 store and retrieve objects from a relational database using Structural Query Language (SQL). After building a database and mapping classes to tables, the aspects connect to the database; handle exceptions; load objects with associated objects; automatically maintain status information of objects (e.g. isDeleted, isChanged); commit objects to the database as part of a transaction. This is an advanced design with improved performance, association keeping, state consistency, and transaction management.

For general reuse, an abstract aspect, PersistentAsp, is designed crosscutting an interface. The subspect PersonnelPersistency ensures persistency of Department and Employee. Some methods are about objects commit at multi-level: 1) Lower-level commit is insert(), delete() and update() for every object. 2) The method commitDB() automatically determines the proper low-level commit operation based on the status information. 3) The static commit() invokes commitDB() on all objects in the pool of a class. 4) PersonnelPersistency declares an upper-level static commit() for all objects of the two classes. These methods provide powerful and flexible approaches for objects persistency.

Figure 11 is a **target view** for Department class, collecting all pointcuts from all relevant aspects. This diagram may be derived from the class diagrams (Figure 6) and relevant aspect views (Figure 9 and 10) of our application. If a type is crosscut by an aspect, all its subtypes are crosscut too. This leads to many aspects weaving on a subtype indirectly. The target view allows examining one target under multiple aspects weaving.

Furthermore, many advices crosscut a target feature. For instance, the method setHead(Employee) is crosscut by 3 pointcut expressions. By the target view, it is easier to examine whether there are duplicate pointcuts, and further improve our pointcut design. Also, the view is helpful to examine potential interferences among a set of pointcuts on a target. Similarly, the target view for the Employee class can be achieved.

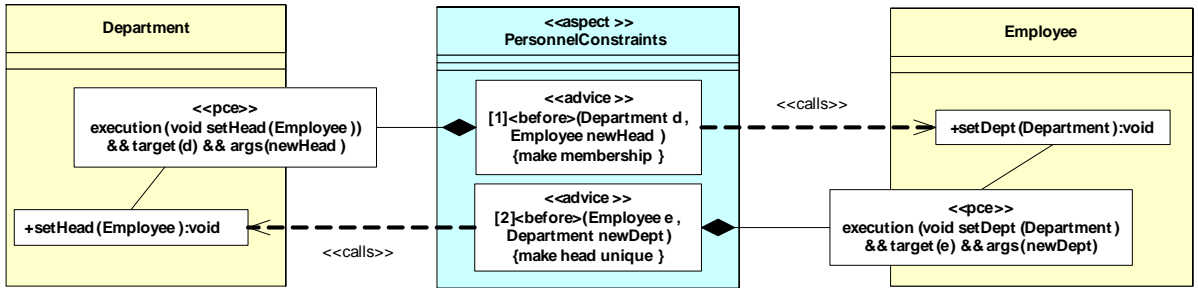


Figure 9. Aspect view: PersonnelConstraints declares two advices. Every advice declares a pointcut expression. Every expression monitors a method execution. The No1 advice ensures that before a new head is assigned to a department, the head is made a member of the department. The No2 advice ensures that before an employee changes its department, if he/she is the head of a former department, the department head should be set to NULL. The dependencies from advices code to called methods are shown, so that a circular dependency becomes immediately visible.

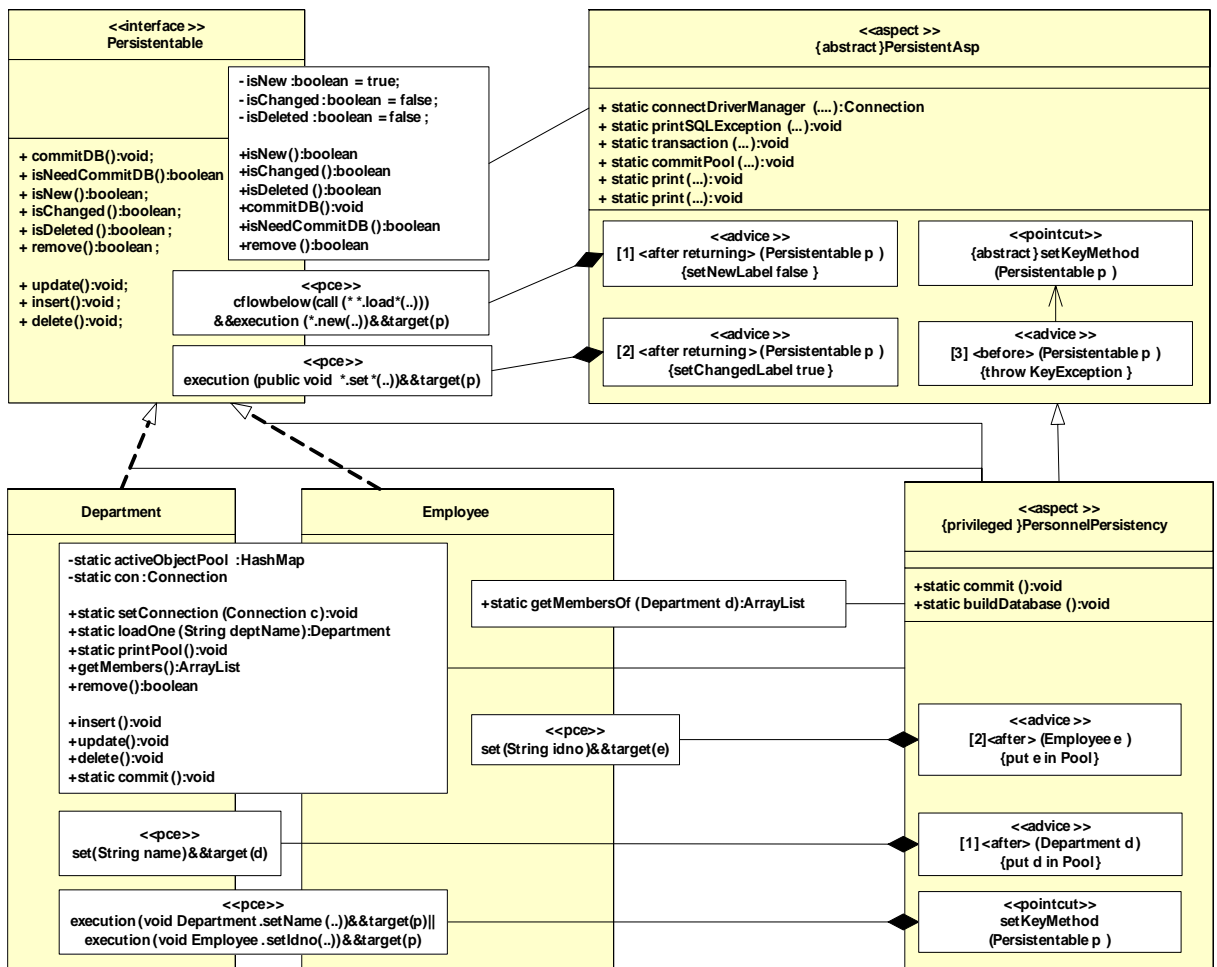


Figure 10. Aspect View: Persistency aspect. An abstract aspect PersistentAsp crosscut an interface by introducing a set of fields and implementing methods. No1 and No2 advices in PersistentAsp are for objects label setting and further for choose suitable SQL commit statements automatically. No3 advice restricts changing key fields on loaded objects by throwing exception. It references an abstract pointcut, and subsaspect will declare a concrete pointcut. The subsaspect PersonnelPersistency is for Department and Employee, making them implement the interface and provide concrete methods. An active objects pool is introduced for each class, so that created objects and loaded objects are all put in the pool, in which objects keep associations, be committed as a whole transaction, and improve performance. No1 and No2 advices in PersonnelPersistency ensure that every object be put in pool after setting key fields.

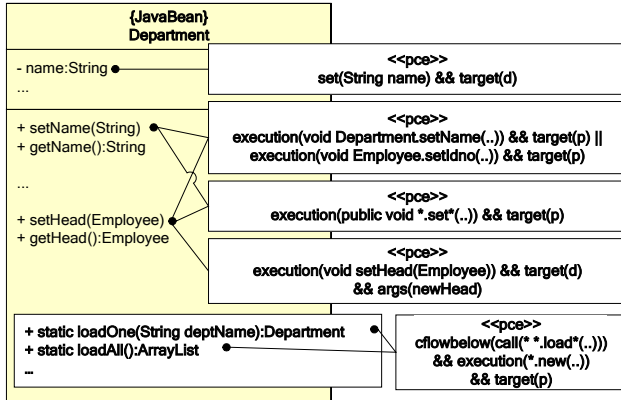


Figure 11. Target view: collecting pointcuts for Department

6. DISCUSSION

So far the AspectJ meta model has been tested with more than 17 aspect examples from the AspectJ v1.1 and v1.2 release documentation [24]. The tests indicate that the semantics and syntax are modeled correctly and completely, and the notations and views are unambiguous and non-conflicting with UML class diagrams.

The AspectJ meta model can be used for forward engineering of AspectJ programs from models and reverse engineering of models from programs. We intend to build a modeling tool based on an implementation of the proposed meta model. Implementation of the AspectJ meta model is straightforward, since the model is self-contained, minimal and extensible. It will allow modeling classes and aspects, and generating code from the models. With the tool for reversed engineering, modeling from AspectJ programs will be done automatically. Because our model is based on the MOF, interoperability with other MOF-based tools for Java and AspectJ will be guaranteed.

7. RELATED WORK

Dedic and Matula [4] propose a Java meta model which was beneficial in our initial development. Our proposal, however, is leaner. Our purpose of building a Java meta model is for AspectJ extending and crosscutting. By tailoring and combining the model we build one that is suitable for AspectJ.

There is a significant body of work on extending UML meta classes to support aspect modeling [1,3, 9,15, 16,18]. In contrast, we proposed a self-contained AspectJ meta model by tailoring UML kernel meta classes to our needs, rather than extending them.

Chavez and Lucena [3] present a meta model for Aspect-Oriented Modeling extending UML. Aspect is modeled as a direct subtype of Classifier in the UML model. Crosscutting is modeled as a new kind of Relationship between CrosscutElement and BaseElement. In our AspectJ meta model, the crosscutting relationship is not represented by one single element. Specifically targeted to AspectJ, we model structural crosscutting by the associations introducedFeature and introducedGeneralization and behavior crosscuts by the classes Pointcut and Advice (Figure 3). These concepts are visualized by a specific notation, which is lacking in [3].

Stein, Hanenberg and Unland [15, 16] model advice in AspectJ as a stereotype of Operation in the UML. However, we think this is

inconsistent. UML specifies that every Feature has its identity in its namespace. However, an advice has no name, so has no identity, thus it is not a Feature, not a Behavioral feature, not an Operation, of course. Note that the same advice may be duplicated in an aspect, and that an operation can be invoked explicitly, while an advice never be requested directly. In order to avoid this inconsistency we model advice as a subclass of Element.

The same authors [15, 16] use template collaboration of UML to specify inter-type declarations in AspectJ. Our AspectJ meta model extends a Java meta model making it simpler by extending Generalization and Feature with a *declarer* association, so as to distinguish the *owner* from the *declarer* of these elements. Thus we model intertype declarations easily.

Suzuki and Yamamoto [18] propose UML stereotypes for aspect modeling (aspect, pointcut, advice, ...). In their proposal, there are two relationships between aspects: inheritance and association. In our model association is not represented explicitly. However, other relationships between aspects can be expressed. 1) Aspect is subtype of Class thus inheriting all possible relationships between classes, including generalization. 2) An aspect may be the crosscut target of another aspect. This can be derived from the fact that an aspect introduces a Feature into a Classifier thus also into an Aspect. In the case of behavioral crosscutting the target can also be code in an aspect. 3) An aspect may reference pointcuts declared in another. 4) An advice in an aspect may call methods declared in another. 5) The precedence among aspects determines execution order of their advices on same joint points with same specs.

Katara and Katz [10] propose an aspect architecture based on the superimposition principle. It provides a concern diagram extending UML. We share the goal to model aspect architecture but approach it differently. On the one hand, our approach is less general in that it focuses on a particular platform, AspectJ [8]. On the other hand, it is more general, because it proposes a concrete meta model on which our notations and views are based.

8. CONCLUSION

Crosscutting concerns are complex themselves. Aspect modeling is significant to simplify the complexity. A meta model is necessary for formal aspect modeling. We present an AspectJ meta model to support AspectJ software modeling. A self-contained Java meta model is built by tailoring UML kernel meta classes. Then an AspectJ meta model extends Java, in which five main meta classes are specified with their semantics, attributes and associations. According to the specification, notations and views are suggested to support aspect visual modeling. A practical example illustrates how to use the meta model for AspectJ program modeling. The meta model is self-contained, straightforward, and easy to implement as a basis for AspectJ modeling tools. We hope it help bridge the gap between conceptual modeling of aspects and AOP.

9. ACKNOWLEDGMENTS

This research was performed during the visit of the first author at the University of Bonn, generously funded by Alfred Krupp von Bohlen und Halbach Stiftung. We would like to thank the members of the ROOTS group in the Dept. of Computer Science III for constructive feedback on intermediate versions of the paper.

10. REFERENCES

- [1] O. Aldawud, T. Elrad, and A. Bader: An UML Profile for Aspect Oriented Modeling, OOPSLA 2001 Workshop on Aspect Oriented Programming.
- [2] G. Booch, J. Rumbaugh, and I. Jacobson: *The Unified Modeling Language User Guide*. Addison–Wesley, Reading, Massachusetts, USA, 1 ed., 1999.
- [3] C. Chavez and C. Lucena: A Metamodel for Aspect-Oriented Modeling, *Workshop on Aspect-Oriented Modeling with UML at AOSD2002*, Enschede, The Netherlands, April 22, 2002.
- [4] S. Dedic, M. Matula: Metamodel for the Java Language, <http://java.netbeans.org/nonav/models/java/java-model.html>
- [5] T. Elrad, M. Aksits, G. Kiczales, K. Lieberherr, and H. Ossher. Discussing Aspects of AOP. *Communications of the ACM* **44**(10), pp. 33–38, October 2001.
- [6] D. S. Frankel: *Model Driven Architecture: Apply MDA to Enterprise Computing*. Wiley Publishing, Inc. 2003.
- [7] J. Gosling, B. Joy, G. Steele and G. Bracha: *The Java Language Specification*, Second Edition, Addison Wesley Longman, Inc. 2000.
- [8] E. Hilsdale and J. Hugunin: Advice Weaving in AspectJ. In K. Lieberherr (ed.): *AOSD2004 – 3rd International Conference on Aspect-Oriented Software Development*, Lancaster, UK, March 22-26, 2004. pp.26-35, ACM 2004.
- [9] M. M. Kandé, J. Kienzle and A. Strohmeier: From AOP to UML – A Bottom-up Approach. Workshop on Aspect-Oriented Modeling with UML at AOSD2002, Enschede, The Netherlands, April 22, 2002.
- [10] M. Katara and S. Katz. Architectural Views of Aspects. In M. Akşit (Ed.): *AOSD2003 – 2nd International Conference on Aspect-Oriented Software Development*, Boston, Massachusetts, March 17-21, 2003. pp.1-10. ACM 2003.
- [11] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersen, J. Palm, and W. G. Griswold: An Overview of AspectJ. In L. Knudsen (Ed.): *ECOOP2001 – Object-Oriented Programming*, Budapest, Hungary, Jun. 2001, p. 327-353, LNCS 2072, Springer 2001.
- [12] G. Kriesel, T. Rho, S. Hanenberg: Evolvable Pattern Implementations Need Generic Aspects, *Proc. of ECOOP'2004 Workshop on Reflection, AOP and Meta-Data for Software Evolution* at ECOOP 2004, Oslo, Norway, June 15, 2004.
- [13] A. Rashid and R. Chitchyan: Persistence as an Aspect. In M. Akşit (Ed.): *AOSD2003 – 2nd International Conference on Aspect-Oriented Software Development*, Boston, Massachusetts, March 17-21, 2003, pp. 120-129. ACM 2003.
- [14] J. Rumbaugh, I. Jacobson, and G. Booch: *The Unified Modeling Language Reference Manual*. Object Technology Series, Addison Wesley Longman, Reading, MA, USA, 1999.
- [15] D. Stein, S. Hanenberg, and R. Unland: An UML-based Aspect-Oriented Design Notation for AspectJ. In G. Kiczales (Ed.): *AOSD2002 – 1st International Conference on Aspect-Oriented Software Development*, Enschede, The Netherlands, April 22-26, 2002, pp.106-112, ACM 2002.
- [16] D. Stein, S. Hanenberg, and R. Unland. Designing Aspect-Oriented Crosscutting in UML. *Workshop on Aspect-Oriented Modeling with UML at AOSD2002*, Enschede, The Netherlands, April 22, 2002.
- [17] S. M. Sutton Jr. and P. Tarr, Aspect-Oriented Design Needs Concern Modelling, *Aspect Oriented Design Workshop at AOSD2002*, Enschede, The Netherlands, April 22, 2002.
- [18] J. Suzuki and Y. Yamamoto. Extending UML with Aspect: Aspect Support in the Design Phase. In Proceeding of *3rd Aspect-Oriented Programming workshop* at ECOOP'99, Lisbon, Portugal, Jun. 1999.
- [19] J. C. Wichman. ComposeJ: The Development of a Pre-processor to Facilitate Composition Filters in the Java Language. *Master's thesis, Department of Computer Science, University of Twente*, 1999.
- [20] A. A. Zakaria, H. Hosny and A. Zeid, An UML Extension for Modeling Aspect-Oriented Systems, *Second International workshop on Aspect-Oriented Modeling with UML* at UML 2002, September 30-October 4, 2002, Dresden, Germany.
- [21] Object Management Group (OMG), UML 2.0 Infrastructure Final Adopted Specification. <http://www.omg.org/cgi-bin/doc?ptc/2003-09-15>
- [22] Object Management Group (OMG), UML 2.0 Superstructure Final Adopted Specification. <http://www.omg.org/cgi-bin/doc?ptc/2003-08-02>
- [23] Object Management Group (OMG), MOF 2.0 Core Final Adopted Specification. <http://www.omg.org/cgi-bin/doc?ptc/03-10-04>.
- [24] The AspectJ Team. AspectJ Programming Guide (v1.2). In <http://aspectj.org>
- [25] HyperJ, <http://www.research.ibm.com/hyperspace/HyperJ/HyperJ.htm>
- [26] AspectC, <http://www.cs.ubc.ca/labs/spl/projects/aspectc.html>
- [27] AspectC++, <http://www.aspectc.org/>
- [28] AspectC#, http://www.dsg.cs.tcd.ie/index.php?category_id=169
- [29] AspectR, <http://aspectr.sourceforge.net/>
- [30] AspectS, <http://www.prakinf.tu-ilmnau.de/~hirsch/Projects/Squeak/AspectS>