An UML-based Aspect-Oriented Design Notation For AspectJ

Dominik Stein, Stefan Hanenberg, and Rainer Unland
Institute for Computer Science
University of Essen, Germany
{dstein | shanenbe | unlandR}@cs.uni-essen.de

ABSTRACT

AspectJ is a well-established programming language for the implementation of aspect-oriented programs. It supports the aspect-oriented programming paradigm by providing a special unit, called "aspect", which encapsulates crosscutting code. While with AspectJ a suitable aspect-oriented *programming* language is at hand, no feasible *modeling* language is available that supports the design of AspectJ programs. In this work, such a design notation for AspectJ programs is presented based on the UML. It provides representations for all language constructs in AspectJ and specifies an UML implementation of AspectJ's weaving mechanism. The design notation eases the perception of aspect-orientation and AspectJ programs. It carries over the advantages of aspect-orientation to the design level.

1. INTRODUCTION

Aspect-oriented programming (AOP) [12] is a new software development paradigm that aims to increase comprehensibility, adaptability, and reusability by introducing a new modular unit, called "aspect", for the specification of crosscutting concerns. AspectJ [2] is a programming language that supports the aspect-oriented programming paradigm by providing new language constructs to implement crosscutting code. At present, no design notation is available that appears to be appropriate for the design of aspect-oriented programs in AspectJ. The need of such a design notation is obvious. First, it would ease the development of AspectJ programs. A graphical notation helps developers to design and comprehend AspectJ programs. Further, it would facilitate the perception of aspect-orientation. A design notation helps developers to assess the crosscutting effects of aspects on their base classes. Its application carries over the advantages of aspect-orientation to the design level and facilitates adaptation and reuse of existing design constructs.

In this work, an approach is presented that extends the Unified Modeling Language (UML) [13] with the aspect-oriented design concepts as they are specified in AspectJ (in the following, the approach is referred to as the "aspect-oriented design model", or AODM for short). The approach reproduces these concepts by extending existing UML concepts using UML's standard extension mechanisms. Doing so assures an immediate understanding of aspect-oriented design models and enables rapid support by a wide variety of CASE tools. Further, the approach reproduces AspectJ's weaving process in the UML. Doing so helps to perceive the effects of aspect-orientation in AspectJ programs (e.g., tools may be developed that generate woven design models).

The remainder of this work is structured as follows. After a short overview of AspectJ and the UML, section 2 introduces UML representations for each AspectJ language construct. Section 3 describes an UML implementation of AspectJ's weaving mechanism. A new relationship is introduced to represent this weaving mechanism. In section 4, existing approaches to extend the UML with aspect-oriented design concepts are regarded with respect to their compliance with AspectJ's semantic. Section 5 concludes this paper and gives a short outlook on the oncoming work to do.

1.1 AspectJ

AspectJ [2] is an implementation of aspect-oriented programming for Java (cf. [1], [8]). It adds to Java several program elements that define modular units of crosscutting code. AspectJ provides the concept of join points and pointcuts to enable dynamic (i.e., context-dependent) crosscutting of behavior. It comes with a preprocessor that *weaves* the crosscutting code of aspects into the code of the base classes. It is part of the aspect to specify where its crosscutting code has to be woven into the base classes. In the following, the language constructs of AspectJ and their semantic are explained briefly.

Join points in AspectJ are "principled points in the dynamic execution of a program" [11] [1]. These points come to pass at several actions, such as method and constructor calls, method and constructor executions, field accesses, as well as object and class initializations. Join points can be considered as distinct points in a dynamic object call graph. In this call graph, control passes through each of those distinct points twice, once the control is passed down to the called object, and once control flows back up.

Pointcuts are sets of join points. Pointcuts are used to specify at which join points crosscutting behavior is to be executed. Pointcuts are defined in terms of pointcut designators. Some of those pointcut designators (such as **`this`**, **`target`**, **`args`**, **`cflow`**, **`cflowbelow`**, or **`if`**) select join points based on the dynamic context they come to pass in.

Advice defines code to be executed whenever a join point of a particular set of join points is reached. It is part of the advice declaration to specify this set of join points (in terms of pointcut designators). As "control passes through each join point twice"

AOSD 2002, Enschede, The Netherlands

[11] (i.e., once the communication is dispatched, and once the communication has been fulfilled) the designer needs to specify at what point in time relative to the execution of the communication (i.e., before, after, or around) the advice is to be executed.

Introductions are used to crosscut the static type structure of the classes. That is, with introductions additional class members like constructors, methods, and fields may be inserted into classes as if they were declared in the classes themselves. Further, introductions may change the classes' super-classes and super-interfaces by inserting new generalization and realization relationships into the class structure.

Aspects are "modular units of crosscutting implementation" [11] and serve as containers for pointcuts, pieces of advice, introductions, and ordinary Java members. Aspects in AspectJ are instantiated by an extraordinary instantiation mechanism. This mechanism allows to instantiate aspects per object, per control flow, or once for the global environment.

1.2 The Unified Modeling Language

The Unified Modeling Language (UML) [13] is an object-oriented design notation that provides basic building blocks to model software-intensive systems, such as *abstractions* that represent structure and behavior of a system, *relationships* that state how the abstractions relate to each other, and *diagrams* that show interesting excerpts of a set of abstractions and relationships (cf. [3]). The most important characteristics of the UML in respect to the issue tackled in this work are its extension mechanisms. These mechanisms are briefly introduced in the following.

UML's extension mechanisms provide standardized means to extend existing UML building blocks with new properties, called *tagged values*, or with new semantic, called *constraints*. Tagged values may be used to attach arbitrary information to a model element, like management information (e.g., author, due date, status) or code generation information (e.g., optimization level, container class). With constraints, new semantic can be specified for a model element.

Besides the alteration of existing building blocks, the UML may be extended with completely new building blocks that are derived from existing ones. These new building blocks, called *stereotypes*, have the same structure (attributes, associations, operations) as the base building block they are derived from. However, they may have a different semantic and may specify additional well-formedness rules or required tagged values that apply to each building block of that stereotype. Stereotypes may be used to indicate a difference in meaning or usage between two building blocks with identical structure (cf. [13]).

2. ASPECTJ'S BASIC ABSTRACTIONS

In this section, UML representations are presented for each of AspectJ's basic abstractions, such as join points, pointcuts, pieces of advice, introductions, and aspects. To do so, the semantic of these concepts is thoroughly analyzed and checked against the existing model elements in the UML.

2.1 Join Points

Join points are no distinct language construct of AspectJ. Rather, they denote abstract points in the dynamic execution of a program. Nevertheless, it is necessary to find a suitable representation for join points in the UML to visualize pointcuts (being sets of join points) and to implement AspectJ's weaving mechanism. Looking for an appropriate UML representation for join points, *links* can be identified as the one model element which represents them best.

In the UML, links serve as communication connection for stimuli. A stimulus reifies a communication between two instances that is dispatched by an action, such as an invocation of an operation, a request to create or to destroy an instance, or a raise of a (asynchronous) signal. This means that control is passed from one instance to another via communication links. Hence, links in the UML represent "principled points in the dynamic execution of a program" just like join points do in AspectJ. And just like join points in AspectJ, control passes each communication link in the UML two times, once the control is passed down to the called instance, and once control flows back up again.

However, whether a link actually represents a join point depends on the exact communication that is dispatched over the link. A link used to communicate the destruction of an instance, for example, does not represent a join point in the sense of AspectJ. AspectJ's join point model defines precisely which kind of communications promotes an ordinary link to a representation of a join point.

In the UML, some communications such as field references or field assignments do not dispatch stimuli. This means that control flow passes no link at all, and no link can be assigned to represent the respective join points. To solve this problem, in the AODM, these communications are stereotyped as "pseudo" invocations of "pseudo" operations that have no other purpose than to read or write (respectively) a specific field. Similar, no link can be identified to represent execution and initialization join points. Considering that the execution of an operation or a constructor or the initialization of an object never occurs without a (preceding) operation or constructor call, it is legitimate to use one link (i.e., the one associated with the call or create action) to represent all two (or three) join points. To represent the order in which control passes these join points, corresponding call, execute, and initialize actions are organized to an UML action sequence.

Join points may be visualized in UML interaction diagrams by highlighting *messages*. In the UML, interaction diagrams are

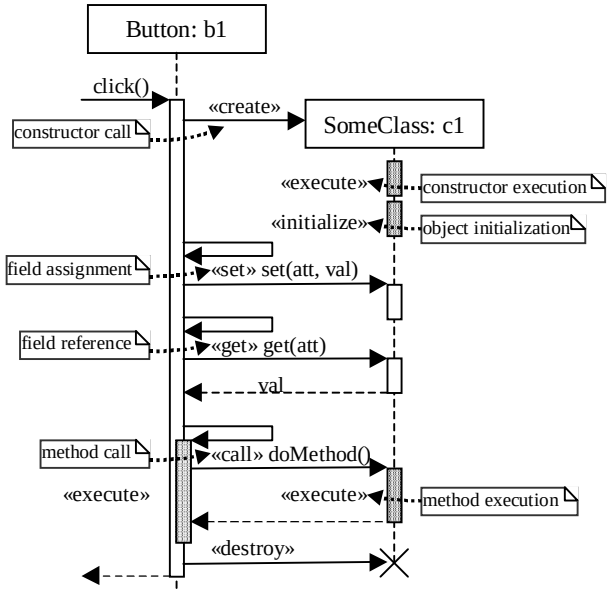

Figure 1: Indicating Join Points in Interaction Diagrams

commonly used to represent communications between instances. In these diagrams, communications are associated with messages. Communication between two instances can only take place, if the communicating instances are connected by a link. With other words, messages can only be send from one object to another, if the sending object has a reference to the receiving object. Hence, considering that messages are associated with communications and require the existence of links, it is proper to highlight messages in collaborations to indicate join points.

The notes in Figure 1 demonstrate which kind of messages may be used to indicate what kind of join points. Join points which come to pass during actions that usually do not result in communications (such as method and constructor executions, object initializations, or field accesses) are indicated by special stereotypes (see «execute», «initialize», «set», and «get» stereotypes in Figure 1; the other stereotypes «create», «call», and «destroy» are pre-defined by the UML specification).

2.2 Pointcuts

In the AODM, pointcuts are represented as operations of a special stereotype, named «pointcut» (see Figure 4 for examples). This is legitimate due to the strong structural resemblance of pointcuts to standard UML operations. Just like standard UML operations, pointcuts are features of a particular classifier (i.e., an aspect), they may have an arbitrary number of (output-only) parameters, and their declaration comprises a signature and an "implementation" (see Figure 2).

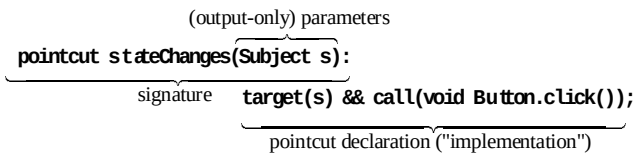

Figure 2: Structural Similarity of Pointcuts to Operations

The «pointcut» stereotype captures a new semantic and specifies several additional constraints. One of those constraints declares that operations of stereotype «pointcut» must be implemented by methods of a special stereotype that equips the standard UML Method meta-class with an additional property named "base" to hold the "implementation" of the pointcut (i.e., its declaration; see Figure 4 for an example).

2.3 Advice

Similar to a pointcut, an advice is represented as an operation of a special stereotype, named «advice» (see Figure 4 for an example). This is legitimate due to the strong structural similarity of an advice to a standard UML operation. Just like a standard UML operation, a piece of advice is a feature of a particular classifier (i.e., an aspect), it may have an arbitrary number of parameters, and its declaration comprises a signature and an implementation (see Figure 3). In contrast to a pointcut, an advice is also semantically comparable to a standard UML operation because it defines some dynamic feature that effects behavior.

However, there is a semantic difference between an advice and an operation. One important difference is, for example, that an advice does not have a unique identifier. This circumstance may cause conflicts with existing well-formedness rules of the UML stating that two operations (i.e., two pieces of advice) in the same classifier (i.e., aspect) must not have the same signature. To avoid such conflicts, the AODM supplies an advice with a "pseudo" identifier

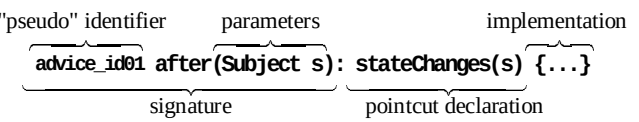

Figure 3: Structural Similarity of an Advice to an Operation

(see Figure 4 for an example). Another difference pertains to inheritance. Since in AspectJ a piece of advice has no unique identifier in the super-aspect, it cannot be overridden in the sub-aspect. The «advice» stereotype captures this semantic difference by constraining that an advice in the AODM (although having a "pseudo" identifier) cannot be overridden. Then, advice declarations in AspectJ contain pointcut declarations that specify the set of join points at which the advice is to be executed. Therefore, operations of stereotype «advice» must be implemented by methods of a special stereotype that equips the standard UML Method meta-class with an additional property named "base" to hold the pointcut declaration. Note how this proceeding coincides with the way that pointcuts are implemented in the AODM (see section 2.2). In fact, the same method stereotype is used for the implementation of both pieces of advice and pointcuts.

2.4 Introductions

In AspectJ, introductions are used to insert members (such as constructors, methods, and fields) and relationships (such as generalization/specialization and realization relationships) to the base class structure. In the UML, *templates* are the appropriate means to do the same (i.e., to introduce new model elements, such as members and relationships, to an existing design model). Templates are parameterized model elements that are used to generate other model elements by binding its template parameters to actual arguments. Templates cannot be used directly in a design model.

Since introductions in AspectJ may insert both members and relationships, the parameterized model element destined to represent introductions in the UML must be able to describe members and relationships, too. After reviewing the UML specification, parameterized *collaborations* can be identified to meet these requirements best. In the UML, collaborations are used to specify a set of instances together with their members and relationships (i.e., a structural context) and a set of interactions that describes some communication between these instances (i.e., some behavior performed within the structural context). So, collaboration templates prove to be suitable to specify structural and behavioral characteristics of introductions. The AODM specifies an extra stereotype of collaboration templates, named «introduction», to capture the particular semantic of introductions (see Figure 5 for zoomed-in and Figure 4 for zoomed-out examples).

Just like ordinary templates, collaboration templates of stereotype «introduction» need to be bound to actual arguments before they can be used in UML design models. The standard UML binding mechanism proves to be not suitable to do so, though, as it does not comply with AspectJ's weaving semantic. The UML well-formedness rules state that "a model element may participate in at most one binding as a client" (i.e., as an argument) [13]. In AspectJ, though, a class may be crosscut by multiple introductions.

Therefore, the AODM specifies a special binding mechanism for collaboration templates of stereotype «introduction». Note that introductions in AspectJ are conceptually *always* bound to (a fixed set of) actual base classes, which are specified as type pattern in the introduction declaration. Accordingly, in the AODM, template parameters of a collaboration template stereotyped with

«introduction» are required to be of a special stereotype, named «containsWeavingInstructions». That stereotype equips the standard UML TemplateParameter meta-class with a supplementary meta-attribute, named "base", to hold the type pattern that specifies the set of actual base classes to be crosscut (see Figure 4 and Figure 5 for examples). A collaboration template of stereotype «introduction» is generally considered to be implicitly bound to the actual arguments specified in that "base" expression. Thus, it is proper to use introduction templates in design models directly.

2.5 Aspects

In the AODM, aspects are represented as classes of a special stereotype, named «aspect» (see Figure 4 for examples). This is legitimate due to the strong structural similarity between aspects and standard UML classes. Just like standard UML classes, aspects serve as containers and namespaces for various features, such as attributes, operations, pointcuts, pieces of advice, and introductions. And just like them, they may participate in associations and generalization relationships.

However, there are differences between aspects and classes concerning their instantiation and inheritance mechanisms. For instance, aspect declarations in AspectJ contain instantiation clauses that specify the precise way in which an aspect is to be instantiated (e.g., per object, per control flow, or once for the global environment). Further, sub-aspects in AspectJ inherit all features from their super-aspects, yet only ordinary Java operations and abstract pointcuts may be overridden. The new «aspect» stereotype captures these semantic differences. Besides that, the stereotype equips the standard UML Class meta-class with a couple of additional meta-attributes to hold the instantiation clause, the pointcut declaration contained in that instantiation clause, and a boolean expression specifying whether the aspect (not just its introductions) may access the members of the base classes as a privileged "friend" (see Figure 4 for an example).

2.6 Example

To demonstrate the use of the design notation, Figure 4 presents a design model of the subject/observer protocol [7] as it is implemented in AspectJ in [1]. The subject/observer protocol specifies a mechanism in which a *subject* entity notifies one or more *observer* entities whenever its state changes.

In Figure 4, the interfaces "Subject" and "Observer" describe the set of operations required by the subject/observer protocol. Their implementation is realized by the introductions "Subject" and "Observer" contained in the abstract aspect "SubjectObserverProtocol" (the exact implementation is not shown; note, though, how the type patterns specified in the "base" expressions of the introductions' template parameters refer to the interfaces' names). Apart from the introductions, the aspect "SubjectObserverProtocol" contains an after advice (given the "pseudo" identifier "advice_id01") and a pointcut "stateChanges". The advice "advice_id01" implements the notification of the observers (not shown) and is executed whenever (i.e., after) a join point designated by the pointcut "stateChanges" (specified in the advice's "base" attribute) has been reached. The pointcut "stateChanges" is abstract (the pointcut's "base" attribute is not defined) and has to be overridden by sub-aspects to meet a certain application's needs. Note how the aspect is provided with additional tagged values determining how the aspect is to be instantiated ("instantiation" tag) and how the aspect may access the members of the crosscut base classes ("privileged" tag).

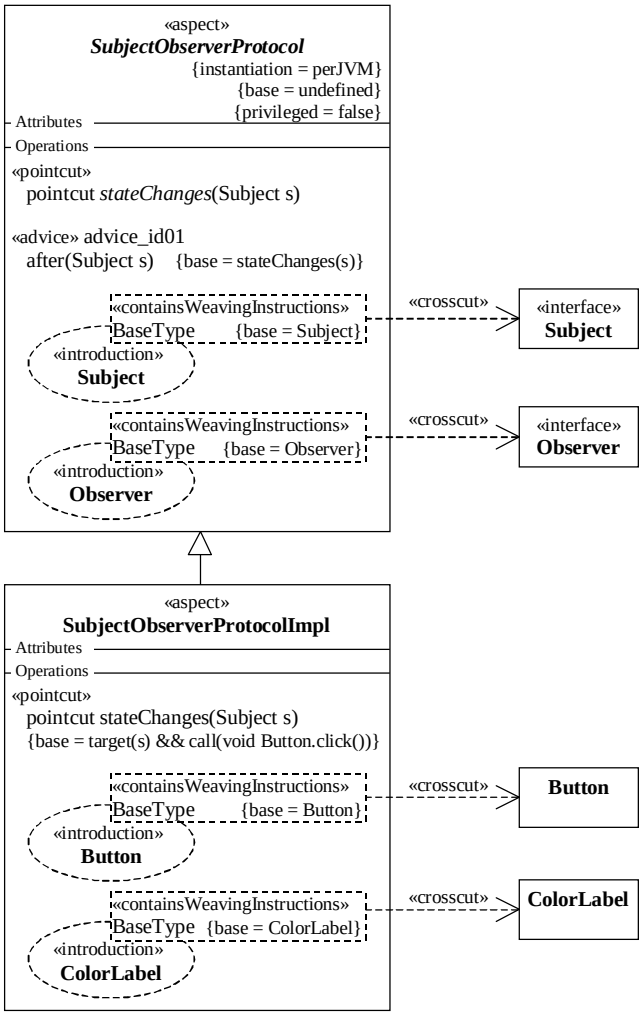

Figure 4: An Aspect-Oriented Design Model

The concrete aspect "SubjectObserverProtocolImpl" applies the subject/observer protocol to a concrete application by extending the "SubjectObserverProtocol" aspect and overriding the "stateChanges" pointcut (by (re)defining the pointcut's "base" attribute). Further, the sub-aspect specifies two additional introductions, "Button" and "ColorLabel". These introductions insert two operations (named "getData" and "update") into the "Button" and "ColorLabel" class and specify a realization relationship from the "Button" class to the "Subject" interface and from the "ColorLabel" class to the "Observer" interface. Figure 5 gives a zoomed-in view on the introductions "Button" and "ColorLabel" illustrating how this is accomplished.

In the AODM, the crosscutting effects of aspects and its components are indicated by «crosscut» relationships. This relationship is introduced at the end of the following section 3.

3. ASPECTJ'S WEAVING MECHANISM

This section presents UML implementations of AspectJ's weaving mechanism. Further, a relationship is introduced denoting the crosscutting effects of aspects on their base classes. Both the weaving mechanism and the relationship are derived from weaving instructions specified in the aspects (cf. section 2).

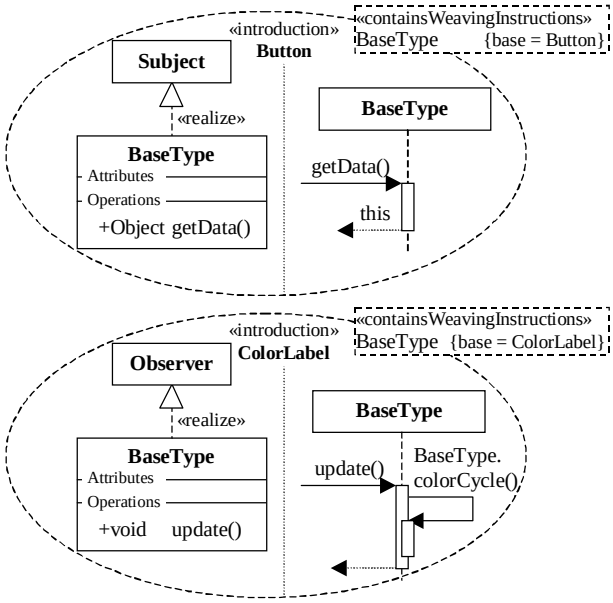

Figure 5: Design of Introductions

3.1 Weaving Advice

The AODM implements AspectJ's weaving mechanism for advice with help of collaborations. In the UML, collaborations are commonly used to describe the behavior of operations (recall that in the AODM, advice is a stereotyped operation and thus is realized by collaborations, too). For weaving purposes, the collaboration describing the behavior of the base classes' operations is split at first. Splitting always takes place at a particular join point (recall that in collaborations, join points are indicated by messages; see section 2.1). Depending on the kind of advice to be inserted, the collaboration is split before, after, or (in the case of around advice) before and after the particular join point. Then, the split fragments are composed with the collaboration describing the advice to form a new collaboration. In the UML, composition of collaborations can be accomplished by identifying and matching instances that participate in each of the collaborations to be composed (cf. [13]).

To explicitly state the order of weaving, the AODM utilizes UML use cases. In the UML, use cases are used to define a piece of behavior of a semantic entity, e.g., the operation of a class or the advice of an aspect. (Super-ordinate) use cases can be split into a set of smaller (sub-ordinate) use cases using *refinement* relationships. Further, use cases may (unconditionally) include the behavior defined in other use cases by means of *include* relationships. At last, a use case may augment the behavior of another use case by means of *extend* relationships. Extend relationships provide a condition that must be fulfilled for the extension to take place.

To represent the weaving order in the UML, the AODM refines the use case describing the base classes' operations (for example, the "click" use case in Figure 6) into three sub-ordinate use cases; one describing the behavior at the join point ("click_step2"), the others describing the behavior before ("click_step1") and after that join point ("click_step3"). Then, the AODM composes a new use case ("wovenClick") that includes the behavior (i.e., the use cases) of both the base classes' operations and the advice. In the UML, collaborations may be specified to explicitly describe how

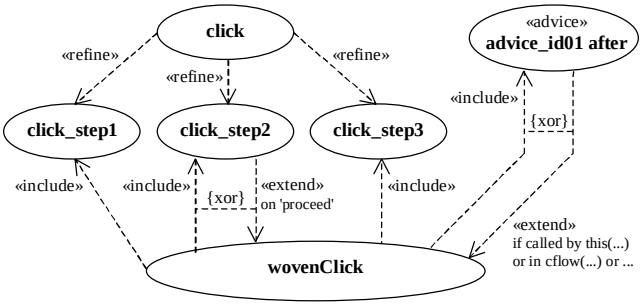

Figure 6: Weaving Advice with UML Use Cases

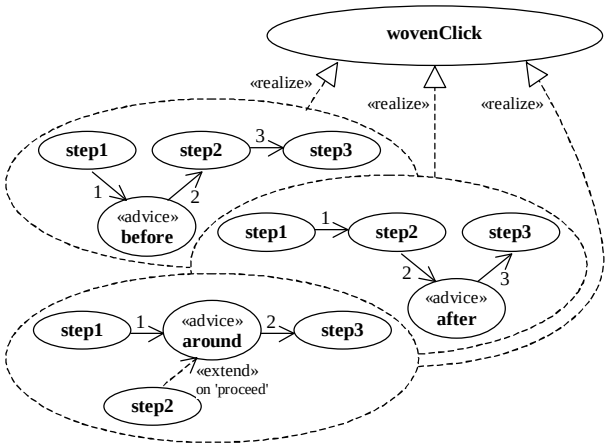

Figure 7: Specifying Weaving Order

the included use cases cooperate to perform the behavior of the including use case. Figure 7 shows three collaborations specifying how the included use cases cooperate in case of a before, after, or around advice to perform the behavior of the including use case (i.e., of the crosscut operation of the base classes).

Special regards must be given to pieces of around advice and of advice that are attached to context-based pointcuts. In these cases, the woven use case is generated by means of extend relationships that precisely specify under which circumstances the behavior of the extending use case is to be performed. If an advice is attached to a context-based pointcut, for example, the extend relationship's condition reflects on the dynamic context in which extension has to take place. For an around advice, the condition generally states that extension shall be performed only if 'proceed' is called. Figure 6 illustrates how these conditions are expressed in UML use case diagrams.

The weaving process may lead to multiple collaborations. This is particularly likely in the case of dynamic crosscutting based on a join point's current execution context (i.e., when a piece of advice is attached to a context-based pointcut). Multiple collaborations may be needed also to describe all possible flows of control through an around advice. This means no conflict with the UML specification, though, as it explicitly allows the existence of multiple collaborations for a single use case (cf. [13]).

3.2 Weaving Introductions

Just like weaving of advice, the AODM implements weaving of introductions with help of collaborations. Recall that introductions are represented in the AODM as collaboration templates of

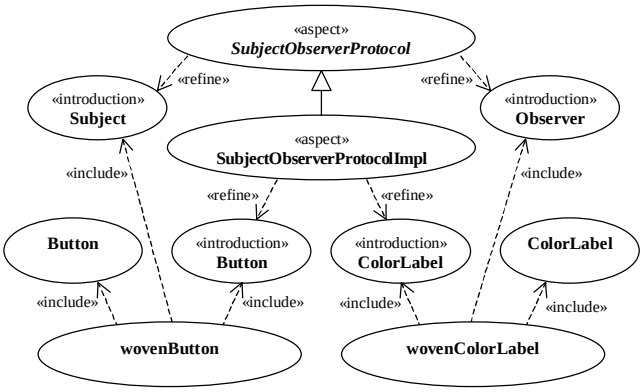

Figure 8: Weaving Introductions with UML Use Cases

stereotype «introduction». Thus, weaving of introductions is realized by instantiating the collaboration template in the base classes' namespace. Before the instantiation, the base classes (specified in the template parameter's "base" tag) are supplemented with the features and relationships specified in the collaboration template so that the design model will not be ill-formed after the weaving process.

Just like the weaving mechanism of advice, the weaving mechanism of introductions is represented in the AODM in a more abstract manner using UML use cases. In Figure 8, for example, the use cases describing the aspects are refined into sets of (subordinate) use cases each specifying the behavior of one individual introduction contained in the aspects. These sub-ordinate use cases (together with the use cases describing the base classes) are then included into new (woven) use cases describing the behavior of the woven (i.e., crosscut) base classes.

3.3 Weaving Relationship

The AODM introduces a new relationship (named "«crosscut»") to the UML to signify the crosscutting effects of aspects on their base classes (see Figure 4 for examples). This relationship is specified in imitation of the extend relationship that is already specified by the UML specification [13]. It is no special stereotype of the extend relationship, though, since extend relationships may only exist between two use cases. Crosscut relationships, however, must connect other kinds of classifiers, as well (such as classes, interfaces, and aspects).

Similar to extend relationships, the crosscut relationship is a directed relationship from one classifier (i.e., an aspect) to another classifier (i.e., a base class) stating that the former classifier affects the latter classifier (in the way that the former classifier is woven into the latter classifier). At the same time, though, the latter classifier remains independent from the former classifier (in the way that its implementation or functioning does not require the presence of the former classifier). Instead, the opposite is true. The crosscut relationship signifies that the former classifier (i.e., the aspect) requires the presence of the latter (i.e., the base class). These characteristics make (the extend relationship as well as) the crosscut relationship distinct from other relationships in the UML, such as the various kinds of dependency relationships.

The crosscut relationship states further that the former classifier (i.e., the aspect) is woven into the latter classifier (i.e., the base class) according to the weaving mechanism described above. Note that crosscut relationships and weaving instructions (specified in the various "base" tags; see section 2) are related to each other by a one-to-one mapping. So (provided with appropriate tool support), designers may specify the crosscutting effects of aspects either by drawing crosscut relationships or by specifying weaving instructions.

4. RELATED WORK

The need for a suitable design notation for the design aspect-orient programs has been recognized soon. Proposals to extend the UML have been made by Suzuki and Yamamoto [15], by Herrero et al. [10], and by Clarke et al. [4] [6]. These approaches do not always meet the semantic of AspectJ, though, which are summarized in the following.

4.1 Approach of Suzuki and Yamamoto

The first proposal to extend the UML with concepts for the design of aspect-oriented programs comes from Suzuki and Yamamoto [15]. In their approach, a new UML meta-class named "aspect" is introduced, which is related to base classes using a UML realization relationship. This proposal implies two capital difficulties.

First, Suzuki and Yamamoto merely present a notation that can be used to design introductions. It remains unclear, how pointcuts or pieces of advice are supposed to be designed with the UML and how their crosscutting effects on the behavior of the base class structure is to be illustrated.

Then, the use of a realization relationship to model the relationship between an aspect and its base classes does not quite comply with the semantic of AspectJ. In the UML, "a realization is a relationship between a specification model element and a model element that implements it" [13]. In AspectJ, though, an advice is no pure declaration of a crosscutting feature. Nor is it the duty of the base classes to implement this feature. In AspectJ, an advice does both, it declares *and* implements the crosscutting feature.

4.2 Approach of Herrero et al.

Herrero et al. [10] seek to separate the design of the object's basic behavior from its non-functional aspects into distinct design entities. These entities are related to each other by means of UML association relationships. These relationships are supplied with a "mapping expression" designating which elements in the design entity representing the base classes correspond with which elements in the design entity representing the aspects. This approach inheres some problems, too.

In the UML, an association is used to express a semantic relationship between two entities (cf. [13]) which, in the case of aspects and their base classes, could be best interpreted as an "is-part-of" or "has" relationship. The semantic of UML association relationships implies further that the members of the participating classifiers remain properties of their respective classifier. In AspectJ, though, introductions are actually injected into base classifiers. Thus, using association relationships does not appropriately illustrate the crosscutting effects of introductions on base classifiers.

In the approach of Herrero et al., AspectJ's pointcut declarations are expressed by mapping expressions, which are attached to the association relationships. In AspectJ, though, pointcut declarations are *properties* of aspects. Hence, attaching pointcut declarations to relationships does not meet AspectJ semantic. Doing so particularly hinders overriding of pointcuts.

4.3 Approach of Clarke

The conceptually most founded approach was introduced by Clarke et al. [4] [6]. She extended the UML with a new design

concept, named "composition patterns". Composition patterns are UML templates for UML packages which are bound to actual classes and operations by means of a special binding composition relationship (cf. [14]). Composition patterns are based on a special design notation for subject-oriented programming [5] [9], called the "subject-oriented design model" [4]. Although the approach originates from the field of subject-oriented programming, Clarke et al. demonstrate in [6] how composition patterns can be used to design aspect-oriented programs with AspectJ, as well. However, the way proposed there does not comply with the semantic of AspectJ in several ways.

Composition patterns imply the semantic of introductions rather than the semantic of advice. An advice in AspectJ, for instance, is executed in the aspect's scope and not in the base classes' scope. That is, within an advice, **`this`** points to the aspect and not to the base class. This means also, that an advice can only access those members of the base classes which are exposed by the pointcut (or which are ordinary Java members of the aspect owning the advice). "Aspect" classes (i.e., *pattern classes*) in composition patterns are merged with their actual base classes, though, meaning that an advice would be run in the base classes' scope.

With composition patterns, only static crosscutting can be designed. Dynamic crosscutting by means of advice is not considered. In AspectJ, an advice may crosscut a given operation depending on the dynamic context in which that operation was called. If the **`cflow`** pointcut designator is used, for example, the respective advice crosscuts a given operation only if that operation was called in the control flow of the designated operation. If the **`this`** pointcut designator is used, the advice crosscuts a given operation only if it was called from the objects of the designated class. These dynamic issues cannot be modeled with composition patterns.

In AspectJ, an advice does not only crosscut the behavior originally defined in the base classes but also the behavior inserted into it by means of introductions. The semantic of a composition pattern does not support such "recursive" crosscutting.

Then, aspects in AspectJ can contain ordinary Java members, like attributes and operations. Composition patterns, though, being stereotyped UML packages, cannot. Members of aspects cannot be declared as members of "aspect" classes (i.e., pattern classes) either since then they would be merged to the actual base classes.

At last, introductions in AspectJ know of the members of their base classes and may work on them. This semantic is not supported by composition patterns, though.

5. CONCLUSION AND FUTURE WORK

In this work, a new approach is presented which reproduces the semantic of AspectJ in the UML. It provides suitable representations for all components of an aspect (such as join points, pointcuts, pieces of advice, and introductions) as well as for the aspect, itself. These representations are extended from existing UML concepts using the standard UML extension mechanisms. The representations are supplied with supplementary meta-attributes to hold the weaving instructions. This way, aspects may be fully specified in concise units in an UML design model, thus carrying over the advantages of aspect-oriented modularity (such as higher comprehensibility, adaptability, and reusability) to the design level.

Furthermore, the approach implements AspectJ's weaving mechanism in the UML and specifies a new relationship signifying the crosscutting effects of aspects on their base classes. This way (provided with appropriate tool support), designers may specify weaving instructions as easy as connecting aspects to base classes. Relationship and weaving process specified in the AODM assist developers to assess the crosscutting effects of aspects at design time.

The design notation presented in this work has been fully specified in a more extensive writing. Next, tools have to be developed that implement this specification so that designers may soon benefit from it.

6. REFERENCES

[1] AspectJ Team. *The AspectJ Programming Guide.* http://aspectj.org/doc/dist/progguide/index.html, Sep. 2001

[2] AspectJ, http://www.aspectj.org, Ver. 1.0b, Sep. 2001

[3] Booch, G., Jacobson, I., Rumbaugh, J. *The Unified Modeling Language User Guide.* Addison Wesley, Reading, MA, 1999

[4] Clarke, S. *Composition of Object-Oriented Software Design Models*. PhD Thesis, Dublin City University, Dublin, Ireland, Jan. 2001

[5] Clarke, S., Harrison, W., Ossher, H., Tarr, P. *Subject-Oriented Design: Towards Improved Alignment of Requirements, Design, and Code*. in Proc. of OOPSLA '99 (Denver, CO, Nov. 1999), SIGPLAN Notices 34(10), 325-339

[6] Clarke, S., Walker, R.J. *Composition Patterns: An Approach to Designing Reusable Aspects*. in Proc. of ICSE '01 (Toronto, Canada, May 2001), ACM, 5-14

[7] Gamma, E., Helm, R., Johnson, R., Vlissides, J. *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1994

[8] Hanenberg, St., Bachmendo, B., Unland, R. *A Meta-Model for General-Purpose Aspect-Languages.* in Proc. of GCSE '01 (Erfurt, Germany, Sep. 2001), LNCS 2186, 80-91

[9] Harrison, W., Ossher, H. *Subject-Oriented Programming (A Critique of Pure Objects)*. in Proc. of OOPSLA '93 (Washington DC, Oct. 1993), SIGPLAN Notices 28(10), 411-428

[10] Herrero, J.L., Sánchez, F., Lucio, F., Torro, M. *Introducing Separation of Aspects at Design Time*. in Proc. of AOP Workshop at ECOOP '00 (Cannes, France, Jun. 2000)

[11] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, K., Palm, J., Griswold, W.G. *An Overview of AspectJ*. in Proc.of ECOOP '01 (Budapest, Hungary, Jun. 2001), LNCS 2072, 327-252

[12] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, Ch., Lopes, Ch., Loingtier, J.-M., Irwin, J. *Aspect-Oriented Programming*. in Proc. of ECOOP '97 (Jyväskylä, Finland, Jun. 1997), LNCS 1241, 220-242

[13] Object Management Group (OMG). *Unified Modeling Language Specification.* Version 1.3, Mar. 2000

[14] Ossher, H., Kaplan, M., Katz, A., Harrison, W., Kruskal, V. *Specifying Subject-Oriented Composition.* in Theory and Practice of Object Systems, Vol. 2(3), 1996, 179-202

[15] Suzuki, J., Yamamoto, Y. *Extending UML with Aspects: Aspect Support in the Design Phase*. in Proc. of AOP Workshop at ECOOP '99 (Lisbon, Portugal, Jun. 1999)