Towards a Common Reference Architecture for Aspect-Oriented Modeling

A. Schauerhuber  W. Schwinger  W. Retschitzegger  M. Wimmer
Women's Postgraduate College for Internet Technologies Vienna University of Technology, Favoritenstraße 9-11, A-1040 Vienna andrea@wit.tuwien.ac.at

INTRODUCTION

Aspect-oriented software development (AOSD) sometimes also called Advanced Separation of Concerns (ASoC) is a fairly young but rapidly advancing research field. AOSD aims at providing new ways of modularization in order to separate crosscutting concerns from traditional units of decomposition during software development.

Today, besides Aspect-Oriented Programming (AOP) [28], different approaches initially not proposed under the term aspect-oriented, such as Adaptive Programming (AP) [29], Composition Filters (CF) [1], Subject-Oriented Programming (SOP) [21], and Multi-Dimensional Separation of Concerns (MDSoC) [33], are now called aspect-oriented, because the term is "catchier, more commonly used, and less subject to ambiguous interpretation" [13].

From a software development point of view, aspect-orientation originally emerged at the programming level with AspectJ [2] as one of the most prominent protagonists. Not least due to the rise of model-driven development [8], however, the aspect-oriented paradigm is no longer restricted to the programming level but is also more and more stretching over other phases of the development life cycle such as requirements engineering (cf. aspect-oriented requirements engineering, e.g., [30], [24]) or design (cf. aspect-oriented modeling, e.g., [8], [12], [16], [25], [39]).

Particularly in the field of aspect-oriented modeling (AOM) there already exist several approaches, each of them having different origins and pursuing different goals for dealing with the unique characteristics of aspect-orientation. This entails not only the problem of different terminology but also led to a broad variety of aspect-oriented concepts. In several cases, concepts of aspect-oriented programming languages are simply incorporated unaltered into a modeling language failing to consider the different levels of abstraction. Applying aspect-orientation at the modeling level is not just injecting code at a certain point within a program but requires the consideration of the full spectrum of modeling concepts not present in programming languages, e.g., different views on the application's structure and behavior as provided by current modeling languages such as UML [31].

This paper contributes to a consolidation of aspect-oriented modeling by taking an initial step towards a common reference architecture that identifies the common ingredients of aspect orientation, abstracted from certain AOP languages or AOM approaches. Such a reference architecture is beneficial in three ways. First, it provides the basis for the construction of a framework of evaluation criteria allowing for a structured and programming-language independent evaluation of aspect-oriented approaches and thereby identifying their strengths and
shortcomings as demonstrated in an extended version of this paper [37]. Second, concepts of different aspect-oriented approaches can be mapped onto each other via the common reference architecture, thus acting as a kind of mediator model. Third, it could act as a blueprint in terms of a metamodel for designing a new, unified aspect-oriented modeling language.

The remainder of the paper is organized as follows: Section 2 discusses related attempts for identifying the common ingredients of aspect-orientation. On basis of this, Section 3 proposes our common reference architecture. Finally, Section 4 critically reflects on our proposal by identifying open problems and issues requiring further investigation.

2. RELATED WORK

Although there already exist several approaches in the area of AOM, to the best of our knowledge there are only a few attempts up to now, that provide a common understanding of aspect-oriented concepts at the programming level or at the modeling level. Some of them provide a dedicated reference architecture, whereas others provide a set of evaluation criteria for surveying existing aspect-oriented approaches only. The design of our reference architecture draws from all those sources.

In van den Berg et al. [40] an attempt towards establishing a common set of concepts for aspect-orientation has been made, based on previous work by Filman et al. [14]. In particular, the concepts of two AOP languages, namely AspectJ [2] and ComposeStar [9] have been examined and expressed in terms of separated UML Class Diagrams. Based on these results the initial definitions of concepts have been revised.

In Chavez et al. [5], a conceptual framework for AOP has been proposed in terms of Entity-Relationship Diagrams. Based on this conceptual framework an evaluation of four programming level approaches, namely AspectJ [2], Hyper/J [23], Composition Filters [1], and Demeter/DJ [29] is presented.

In contrast to these proposals, our reference architecture does not only consider programming level constructs, but also takes up a broader view on aspect-orientation by explicitly considering the modeling level. Thus, these attempts are only partly applicable for our reference architecture. Furthermore, with respect to [40], we provide a unified reference architecture in terms of a UML Class Diagram instead of representing the concepts of each approach in an isolated way.

In contrast to the above mentioned work, Hanenberg et al. [20] present a set of criteria that was used to evaluate four AOP languages. Mik Kersten [27] also provides a comparison of four leading AOP languages, having only AspectJ in common with Hanenberg et al. In addition Mik Kersten also investigates their corresponding development environments.

An extensive survey done by Chitchyan et al. [7], including also aspect-oriented analysis and design approaches, presents the evaluation results of 22 AOM proposals. Based on this evaluation, which categorizes the approaches into requirements, architecture and design approaches, an initial proposal for an integrated aspect-oriented analysis and design process is outlined. However, while a set of criteria has been identified, a precise definition of some of the criteria used to evaluate the approaches is missing.

Similar, but less extensive surveys on AOM have been provided by Reina et al. [35] and Blair et al. [4]. While Reina et al. [35] compare different AOM approaches with respect to four high-level criteria, Blair et al. [4] did not only focus on AOM, but compare several approaches in different phases of software development. In particular, an explicit set of criteria is provided for the phases of aspect-oriented requirements engineering, specification, and design.

The above mentioned surveys provide a valuable source, since they identify common criteria for aspect-orientation. Nevertheless, these criteria have not yet been composed into a common reference architecture. We have adopted their criteria where appropriate and refined them so that they can be applied for our common reference architecture at the modeling level.

3. AOM REFERENCE ARCHITECTURE

Applying aspect-oriented concepts, which were originally coined for the programming level (e.g. by AspectJ [2]), to the modeling level turns out to be a challenging task. This is on the one hand due to the very specific meaning of programming level aspect-oriented concepts and on the other hand due to different concepts introduced by related approaches. An example for the first issue are AspectJ’s join points which are defined as “points in the execution of the program” including field accesses, method and constructor calls [2]. This definition is too restricted for the modeling level since runtime is not the primary focus of modeling. With respect to the latter issue, an example is the concept of aspect in AOP where similar though different concepts have been introduced in other approaches, e.g., hyperslice in Hyper/J, filter in CF, and adaptive method in Demeter/DJ [5]. Consequently, instead of sticking with AOP concepts, it is rather advisable to find general definitions of aspect-oriented concepts that apply to any level in the software development lifecycle.

In order to support the process of establishing a common terminology, we primarily adopt the definitions presented in [40] but refine them to be suitable for the modeling level. Additionally, based on the surveyed approaches we extend the definitions to provide a broad base of conceptualization of aspect-orientation.

In Figure 1 our reference architecture for aspect-orientation is shown as a UML class diagram, which comprises the concepts of aspect-orientation at a higher level of abstraction. Thus, it represents an initial proposal for a conceptual model for aspect-orientation in the very sense it is asked for in [40].

In the following, the concepts of the reference architecture are described along with its major building blocks.

3.1 Concern Decomposition

The concern decomposition deals with the general decomposition of the system under development into concerns and their interrelationship.

Concern. Along with [40] we define a Concern as an interest which pertains to the system’s development, its operation or any

1 Admittedly, AspectJ also allows the introduction of adaptations with respect to the program’s structure, but join points are defined with respect to runtime only.
other matters that are critical or otherwise important to one or more stakeholders. A concern in this respect represents an inclusive term for aspect and base, which is depicted using generalization in Figure 1. We refrain from referring to crosscutting and non-crosscutting in our reference architecture since they represent interests with respect to a system at the level of requirements rather than the modeling level. On the contrary, aspect and base form a representation of concerns in a more formalized language (e.g. a modeling language or a programming language). We modeled an overlapping generalization relationship in order to point out that an aspect – in the sense of a role – might, at the same time, represent a base for other aspects, allowing for both symmetric and asymmetric approaches to decomposition [22]. A disjunctive generalization relationship would allow asymmetric approaches only.

**Base.** A base is a unit of modularization formalizing a non-crosscutting concern. In most programming and modeling paradigms the provided units of modularization allow for decomposing a system according to one dimension only, called dominant decomposition [33]. The object-oriented paradigm for example provides hierarchically ordered units of modularization (i.e. classes and methods) in terms of a vertical decomposition. Thus, it does not support horizontal decomposition, i.e., crosscutting concerns, that are typically scattered across the dominant decomposition.

**Aspect.** An aspect is a unit of modularization of a crosscutting concern with respect to another concern. Aspects are related to other aspects in three ways. First, aspects themselves may be acting as base for other aspects, i.e. an aspect is adapting another aspect (cf. overlapping). Second, an aspect might be specialized into several sub-aspects, thus refining where and how other concerns might be adapted. Third, two or more aspects might introduce adaptations to a concern in a way that causes conflicts, i.e. contradicting adaptations with respect to the same element in the model. Thus, for such aspects a conflict resolution has to be specified defining the precedence of one aspect over another. Which kind of conflict resolution is applicable depends on the particular domain. This fact is represented in the reference architecture by the abstract association class conflict resolution, which – in form of a Strategy pattern [17] – can embrace any concrete conflict resolution, (e.g. relative or absolute ordering) that might be applicable.

**Weaving.** In AOSD the composition of aspects with other concerns, which in turn are either bases or aspects, is called weaving. For our purposes, we distinguish between two ways of weaving aspects into other concerns, namely static (i.e. at design time) and dynamic (i.e. at runtime). Thereby, one aspect of a system may be statically composed with other concerns whereas another aspect may be dynamically woven, which is taken into account by an association class. The weaving relationship is navigable only from the aspect's side, meaning that the concern is oblivious [15] to possible adaptations by aspects.

**AdaptationRule.** An aspect's adaptation rules introduce adaptations at certain points of other concerns. Consequently, an adaptation rule consists of an adaptation describing how to adapt the concern and a pointcut describing where to adapt the concern. We modeled the consists-of relationships using weak aggregations, since both adaptation and pointcut might be reused in other adaptation rules.

![Figure 1. AOM Reference Architecture](image_url)

### 3.2 Language

The following concepts describe the language underlying the specification of base and aspect.

**Language.** Depending on the current focus in the software development lifecycle the language might represent for example a modeling language or a programming language. For sake of simplicity, the reference architecture is defined to cover a single language only (cf. Section 4).

**Element.** Concerns are formalized using elements of a certain language. With respect to aspect-orientation, elements serve two purposes. First they are represented as join points and thus specify where to introduce adaptations. Second, elements of a language are used for formulating an adaptation. Such elements are either structural elements or behavioral elements as depicted in Figure 1.

**StructuralElement.** Structural elements of a language are used to specify a system's structure.

**BehavioralElement.** Behavioral elements of a language are used to specify a system's behavior.

### 3.3 Adaptation Subject

The adaptation subject describes the concepts required for identifying where to introduce an aspect's adaptations.

**JoinPoint.** A join point specifies where an aspect might insert adaptations. Thus a join point is a representation of an identifiable structural or behavioral element of the underlying language used to capture a concern. In addition, join points can be either static or dynamic (cf. dynamicity attribute). Static join points are elements of a language that can be identified based on information
available at design time (e.g. method definition). Dynamic join points are elements of a language that can not be identified before runtime (e.g. method execution).

**JoinPointModel.** The join point model comprises all elements of a certain language where aspects are allowed to introduce adaptations.

**Pointcut.** A pointcut represents a subset of the join point model, i.e. the join points used for specifying certain adaptations. The selection of join points as pointcuts can be done for example by means of a query on the join point model (cf. SimplePointcuts and SelectionMethod). For reuse purposes, pointcuts can be composed of other pointcuts (cf. CompositePointcuts), which refer to the same join point model. Pointcuts are associated with adaptation rules by additionally assigning an optional relative position (cf. RelativePosition). We refrain from associating join points directly to an adaptation rule but use pointcuts instead as a level of indirection allowing for reusing join points in other adaptation rules and other pointcuts.

**RelativePosition.** A relative position may extend a pointcut when used in an adaptation rule. This is necessary since in some cases, selecting join points by pointcuts only, is not enough to specify where adaptations have to be inserted, since an adaptation can be introduced for example before or after a certain join point. The relative position is modeled as an association class, which allows to specify a different relative position for each pointcut in the context of different adaptation rules. In some other cases a relative positioning is not necessary, e.g. when a new attribute is introduced into a class the order of the attributes is insignificant. Therefore, the attribute relPos is specified being optional.

### 3.4 Adaptation Kind

The adaptation kind comprises the concepts necessary to describe an aspect's adaptation.

**Adaptation.** An adaptation specifies in what way the concern's structure or behavior is adapted, i.e., enhanced, replaced or deleted. This concept is similar to the commonly found definition of an advice which represents an artifact that augments or constrains concerns (cf. [40]) and resembles a differentiation proposed in [20] in terms of constructive (cf. enhancement), and destructive (cf. replacement and deletion) adaptation effects.

**StructuralAdaptation.** A structural adaptation comprises a language's structural elements for adapting concerns.

**BehavioralAdaptation.** Likewise to structural adaptation, a behavioral adaptation comprises a language's behavioral elements for adapting concerns.

**CompositeAdaptation.** For reuse purposes, adaptations can be composed of a coherent set of both structural and behavioral adaptations. In this respect the adaptation concept extends the general understanding of the advice concept described in [40].

### 4. CRITICAL REFLECTION

In the following, we critically reflect on our reference architecture by further discussing certain design decisions and by pointing out open issues that require further investigation. The discussion follows the reference architecture's four major building blocks.

**Dynamic weaving beneficial also at the modeling level.** In our reference architecture, weaving of aspects into base concerns is possible at different points in time, either at design time or at runtime. This design decision of the reference architecture has been driven by weaving concepts in AOP. At modeling level, it still can be argued that being able to distinguish between static and dynamic weaving of base and aspects is advantageous for two reasons. First, if the runtime semantics of the language's metamodel has been specified (which for UML is the case only for parts of the language like state machines), i.e., models are executable, dynamic weaving may happen while executing the models similarly to the way it happens at code level. Second, this distinction allows specifying what aspects need to be statically or dynamically woven into the base program during later stages of the development process.

**Adaptation Rules should be represented separately.** In AOP, adaptation rules, i.e., the specification where to adapt how (such as the pointcut-advice combination in AspectJ) were specified in an intermingled way. Some AOM approaches [10], [19] provide an adaptation rule specification that is independent from both base and aspect for reusability reasons. In [10], the authors distinguish between modeling the aspect's adaptations and modeling adaptation rules by proposing a connector metamodel for aspect-oriented composition. Furthermore, in [19], independence of linking technology (e.g. AspectJ) is achieved by introducing the connector concept to link aspect and base concerns. Along with those approaches, we clearly separate the adaptation rule from the adaptation for reasons of enhanced variability, reusability, and expressiveness.

**Appropriate language for AOM necessary.** With respect to providing appropriate abstraction mechanisms, the question arises to what extent existing non aspect-oriented languages need to be extended to sufficiently cover aspect-oriented concepts. Considering for example UML, despite of its expressive power, as commonly known, either a heavy-weight or a light-weight extension can be employed to cover aspect-oriented concepts. Whereas for a heavy-weight extension the UML metamodel itself is extended and can even be refined through sub-classing of any UML meta-class, in the light-weight case, only extensions using stereotypes are allowed which are grouped into profiles, thus fostering tool interoperability. Currently, the use of light-weight and heavy-weight extensions in existing AOM-approaches is balanced.

**Multiple languages should be considered.** Currently, for convenience of concept definition, the reference architecture is defined with respect to a single language only, i.e., both base and aspect are defined by elements of the same language. If this constraint is relaxed, different languages may be applied, first for specifying base concerns, second for specifying the adaptation and third for specifying adaptation rules. This would allow us, for example, to draw from the benefits of different domain specific languages (DSL). This raises the question to which extent these languages may be different and how much they must have in common to still allow for aspect weaving. Considering again the case of UML, it has to be investigated if it is preferable to base these languages on the same meta-metamodel, i.e. MOF [31], or if it is beneficial to bridge the heterogeneity between the bases' and aspects' languages by means of a weaving model (cf. [36]).
Join points required along two orthogonal dimensions. In Hanenberg et al. [20] join points of aspect-oriented programming languages are categorized according to two dimensions, dynamicity and feature. In contrast to that, our focus is broader in that we consider modeling level join points and in that we consider these two dimensions as being orthogonal. Consequently, join points are representations of structural or behavioral elements of a language, while at the same time, they are also modeling level representations of static or dynamic elements in a software system. Exemplifying those four categories by means of UML modeling elements, structural join points would be classes (static) and objects (dynamic), whereas behavioral join points would be activities (static) and method executions (dynamic). Admittedly, not all languages may offer elements which allow for dynamic join points as is the case with UML.

Nature of relative position is language dependent. For dynamic join points the relative position resembles a temporal specification for example before an event occurs. A typical example are AspectU’s before advices that are adaptations for dynamic join points, a technique called wrapping in [14]. For static join points the relative position is defined with respect to the element’s structure. For example, if a link is added, its relative position in terms of the participating object is specified. Consequently, the nature of a relative position depends on both the kind of element representing the join point and the kind of adaptation. Our reference architecture currently does not explicitly cope with these dependencies.

Adaptation effect should be explicit. There exists an inherent relationship between pointcuts or rather their relative position and adaptations with respect to the effect an aspect has on the base. One and the same adaptation may have an enhancement effect, a replacement effect, or a deletion effect depending on the pointcut and its relative position when used in the adaptation rule. For example the relative position before leads to an enhancement whereas in case of around the adaptation may resemble an enhancement, a replacement, or a deletion. Because of this interdependency, currently the adaptation effect is not explicitly represented in the reference architecture, although, this would be beneficial since it would create more awareness of the consequence of the aspect introduced.

5. OUTLOOK
Besides further detailing our reference architecture on basis of the issues identified in the previous section, future work heads into two different directions.

Mapping of AOM/AOP approaches to our reference architecture. One crucial activity we currently are focusing on is to demonstrate the appropriateness of our reference architecture in terms of its unification ability. For this, we intend to specify mappings to well elaborated existing AOM and AOP approaches. Such mappings could be, e.g., defined on basis of OMG’s QVT proposal [34], provided that the approach in question is based on MOF [31]. On the basis of such mapping definitions our reference architecture could also act as a pivot model translating between different aspect-oriented languages.

Development of an AOM language for context-aware Web applications. With respect to application domains for AOM, not least because of several projects in this area [26], [18], [38], we concentrate on context-aware web applications, similar to [3]. This new generation of web applications, also called ubiquitous web applications (UWA) adhere to the anytime/anywhere/any-media-paradigm and are required to be customizable, i.e. the adaptation of their services towards a certain context e.g. time, location, device, and user. Since customization can affect all parts of such applications including content, hypertext and presentation level and because the base concerns of an UWA in terms of its services should be oblivious to the need of customization, customization is regarded as a crosscutting concern, which allows to make existing web applications context-aware. We are currently investigating to which extent existing AOM approaches can be employed for the model driven development of such ubiquitous web applications, or if the development of a UML profile for AOM on basis of our reference architecture would be more appropriate.

6. ACKNOWLEDGEMENTS
This research has been partly funded by the Austrian Federal Ministry for Education, Science, and Culture, and the European Social Fund (ESF) under grant 31.963/46-VII/9/2002.

7. REFERENCES

2 In literature (amongst others [27], [5], [14]) we find different interpretations of what a join point is. The focus is on describing the join points’ properties such as dynamicity and structural & behavioral features, sometimes mixing up terms (e.g. using static as a synonym for structural).

3 While Hanenberg et al. [20] use the term "abstraction", we adhere to UML terminology [31] in that we distinguish between structural and behavioral features.


