

Aspect-Orientation from Design to Code

Iris Groher
Siemens AG, CT SE 2
Otto-Hahn-Ring 6
81739 Munich, Germany

groher@informatik.tu-darmstadt.de

Thomas Baumgarth
Siemens AG, CT SE 2
Otto-Hahn-Ring 6
81739 Munich, Germany

thomas.baumgarth@siemens.com

ABSTRACT

The AO paradigm focuses mainly at the implementation phases of the software lifecycle and is missing standardized concepts for early stages of the development lifecycle. The term *Early Aspects* refers to crosscutting properties at the requirements and architecture level and this paper addresses the separation of crosscutting concerns at the architecture design phases by offering AML (*Aspect Modeling Language*), a notation for aspect-oriented architecture design modeling that is standard UML conform. Within the notation, crosscutting artifacts are clearly encapsulated and completely kept apart from the business logic to foster their reuse. A clear separation of the AO language dependent from AO independent parts simplifies the support of a number of different AO languages and concepts. To extend the support beyond the architecture phase a code generator is presented addressing low-level design support by offering an automated mapping from design models to programming models to prevent inconsistencies among design and implementation.

Categories and Subject Descriptors

D.2.2 [Design Tools and Techniques]: Computer-aided software engineering (CASE)

General Terms

Architecture, Design, Languages

Keywords

AOSD, aspect, crosscutting concerns, early aspects, architecture design modeling, UML

1. INTRODUCTION

Separation of concerns [1] is one of the fundamental principles in software engineering. It states that a given problem involves different kinds of concerns, which should be identified and separated in order to manage complexity and to achieve required engineering quality factors such as adaptability, maintainability, extensibility and reusability. OO software development proved its usefulness regarding the separation of functional concerns of a system. Concerns that crosscut these functional decompositions do not fit equally well into the OO model and have a potentially harmful impact on engineering quality factors mentioned above. Aspect-oriented programming [2] addresses these concepts at the implementation level and offers low-level support for separation of concerns. Aspects are implemented as first-class elements that are expressed in terms of their own modular structure, thus enabling the modularization of crosscutting concerns.

Early Aspects refer to crosscutting properties at the requirements and architecture level ([3]). The term denotes aspect-orientation within the early development stages of requirements engineering and architecture design. This paper focuses on the separation of crosscutting concerns at the high level architecture and the low level design while offering an approach for aspect-oriented modeling and automated code generation. Typically, design artifacts that crosscut an architecture cannot be encapsulated by single components or packages and are typically spread across several of them and therefore also make design hard to understand and maintain. This work addresses the specification of crosscutting concerns at the architecture level in order to maintain the separation of concerns at an early stage in the software development lifecycle. Crosscutting design artifacts can clearly be encapsulated avoiding tangling and scattering.

An extension to UML [4] [5] is presented, without changing its metamodel specification, to achieve standard UML conformity. This helps developers to become acquainted with AO modeling when they are already familiar with OO modeling and UML. A key intention was to offer standard development tool support and interchangeability between various tools. UML is customized by using standard extension mechanisms only. To gain the benefits of code and design reuse of AO software, the ability to reuse aspect and business logic separately is needed. A notation is presented where aspect and business logic are completely kept apart. Thus, both are reusable and at the same time independent of the implementation technology. Within this approach it is assumed that the requirements have already been defined and specified during previous development stages.

To ease the transition from design to implementation and to offer low-level architecture design support, a code generator was developed to support automatic generation of AO code skeletons from design models. This helps developers to focus on models having the code skeletons generated automatically to gain the benefits they are used to in OOSD. Code generation improves developer productivity, ensures syntactical correctness and reduces errors when mapping a model to code. The presented UML notation in combination with the code generator makes AOSD more usable and more efficient for software development by avoiding inconsistencies among design and implementation. Developers can then concentrate on AO design having the code skeletons generated automatically.

The remainder of this paper is organized as follows: Section 2 presents shortcomings of the current state of research on aspect-oriented modeling and describes the need for AO architecture design. Section 3 describes the syntax and semantics of the developed notation. Section 4 presents the automated transition

from design models to implementation models. We conclude with a note of related work and a summary in Section 5 and 6.

2. PROBLEM STATEMENT

The architecture design is an important step within the software development lifecycle. OO design has proved its strength when it comes to modeling common behavior. However, OO design does not adequately address design artifacts that crosscut an architecture. They cannot be encapsulated by single components or packages and are typically spread across several of them and therefore also make design hard to understand and maintain. Crosscutting concerns are present during all phases of a software development lifecycle, leading to code tangling or code scattering during the implementation phase and *graphical tangling* during the design phase. AOSD is still lacking standardized concepts at the design phase that would foster the specification of crosscutting concerns at the high level architecture and low level design. Development of large software systems follows processes that all include activities like requirements engineering, analysis, design and implementations. Following a design methodology like OOD, and focusing on AOP at coding level causes a shift of paradigms between OO design and AO code. This leads to inconsistencies between design and implementation as the AO paradigm is not seamlessly supported during the early stages of the development lifecycle. To avoid the divergence of design models and code, crosscutting concerns must be identified at the requirements and architecture level and carried forward in the implementation phase. Concepts are needed for a seamless integration of AO design and implementation and will be a first step towards an integrated AO development process. To make AOSD more widely accepted, the different phases of an AOSD lifecycle have to be integrated more smoothly by supporting the AO paradigm in every phase. This work includes both, a design notation as well as a code generator for automatic code generation and validation of AO models. Supporting design models and their transition to concrete implementations makes AOSD more usable, more efficient and more accepted among software engineers.

When analyzing OO design, one can see that OO modeling tries to adopt many of the OO programming features for design and analysis. Classes, their structures, and their relationships are identified and generalization and aggregation hierarchies are built. OO design techniques are not sufficient when focusing on the AO paradigm as crosscutting concerns also make design tangled and therefore hard to understand and maintain. When developing an AO modeling approach, the following requirements are obvious:

- A sufficient notation should be simple to understand and straightforward to use for developers who are familiar with common design notations (such as UML).
- Design modeling should be supported by powerful CASE tools to improve developer productivity and to ensure syntactical correctness of the AO model.
- Design notations should support modeling according to the paradigms behind the most common AO approaches and languages.
- Models should be easy to read and offer a clear separation of concerns to avoid crosscutting concerns spanning over many design elements.

- A direct mapping between the notation and supported implementation languages should allow automatic code generation based on the design model.
- The notation should be applicable in real-world development projects and should be part of an integrated AO development process.

This work can be seen as a step towards a standardized way to capture aspects at the design phase of an AO development process. Existing approaches and prototypes are well aware of the fact that aspect-oriented modeling is a critical part of AOSD. Obviously, to obtain an AO development lifecycle, the gap between AO requirements engineering and AOP has to be filled. This work makes a contribution to the problem of bridging this gap.

3. ASPECT MODELING LANGUAGE

This work specifies an approach for AO modeling to address the specification of crosscutting concerns at the architecture level in order to maintain the separation of concerns at an early stage in the software development lifecycle. A key intention is to offer standard development tool support and interchangeability among various CASE tools, thus an extension to UML was developed without changing its metamodel specification to achieve standard UML conformity. Using UML as a modeling language improves developer productivity and offers high acceptance, as it is the industry-standard modeling language for the software engineering community. When using standard UML for aspect-oriented modeling, developers do modeling by using familiar tools and environments to gain all the benefits they are used to in OO design. UML is an extensible modeling language that enables domain-specific modeling which raises its suitability as a modeling language for supporting aspect-oriented modeling. Another important goal was to gain the benefits both of code and design reuse of AO software, including the ability to reuse aspect and base elements separately. Thus, aspects and base elements should be completely kept apart and independent of the implementation technology in order to simplify the replacement of the AO language. A clear separation of the language dependent crosscutting parts eases the support of many different AO languages and concepts. This work focuses on adopting AspectJ [6] [21] concepts for the implementation language dependent parts of AML; the support of other AO concepts (such as HyperJ [18] [19] [23]) is considered and part of some future work.

AML considers the fact that crosscutting concerns tend to affect multiple classes in a system. Since a concern itself can consist of several classes and since all of these classes may be associated with the class the concern crosscuts, the module construct for a concern should be higher-level than a class. Otherwise associations modeled on class-level would supersede the logical grouping of the classes belonging to one concern. This would make the design models hard to read and lead to graphical tangling of crosscutting concerns instead of a clear separation.

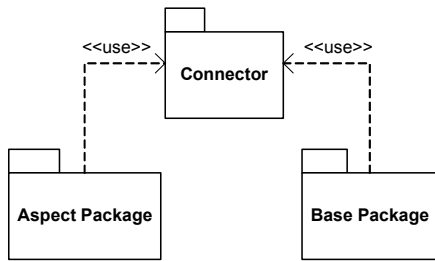


Figure 1: Package Level (De) Composition

Figure 1 provides an overview of the notation and its focus on package-level decomposition. AML includes a *base package* (containing the business logic), an *aspect package* (containing the crosscutting concern) and a *connector* to link aspects and base elements. This separation enables high reusability of the aspect and base elements since the connector is the only crosscutting element. Focusing on UML packages as a central decomposition unit leads to design models that are easy to read, as they avoid *graphical tangling*. Additionally, the connector encapsulates the underlying implementation technology (e.g. AspectJ). The aspect can be modeled independently of any design it may potentially affect. The connection between base design and aspect design is specified separately. Support of different AO technologies is therefore rather simple and straightforward, as it is only the connector's syntax that has to be changed.

The aspect package provides a graphical representation (class diagram) of the static view of a particular crosscutting concern and is, along with the base package, one of the OO parts of the AO model. The base package contains the business logic of the system and can be modeled without considering any crosscutting concern that may potentially affect the system. Similar to the aspect package, the base package can contain any valid UML model that describes the business logic of the desired system. There is no direct relationship among the aspect package and the base package; their relationship is only defined through the connector package containing the rules for the later recomposition of aspects and base elements. As AspectJ is currently the best known AO language, all connector semantics presented here have been developed according to AspectJ's connection model.

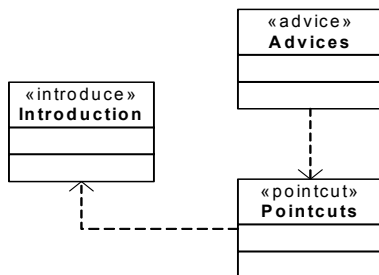


Figure 2: AspectJ specific Connector

As shown in Figure 2, the AspectJ-specific connector package can contain the following classes that conform to the concepts AspectJ offers for the specification of weaving rules:

1. The *Introduction* class, which defines the rules for AspectJ's introduction mechanism.
2. The *Pointcut* class, which defines execution points in the control flow of the program.
3. The *Advice* class, which defines the code to be executed at the pointcuts defined in the *Pointcut* class.

All classes contain operations with special semantics to specify how aspect and base elements have to be recomposed. The complete syntax of the AspectJ specific connector will not be presented here; the following example should provide a view of how the notation can be used and shows some of the most important constructs.

The example in Figure 3 shows how to model an aspect related to tracing to give some guidelines and indications on how to use our notation.

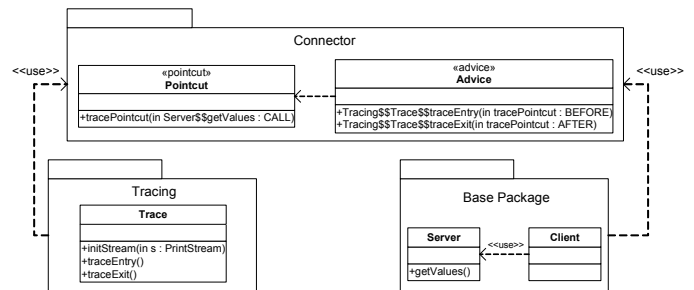


Figure 3: Tracing Design Example

Every time the user performs an invocation on the Server, the action should be traced. Both, tracing aspect and business logic, are independent from each other, no connection is modeled inside. The connector, specifying the weaving rules, includes program execution points (pointcuts) and actions performed at those points (advices). The pointcut (*tracePointcut*) is triggered every time the client invokes the *Server.getValues()* method. The action to be performed before the method call is tracing the entry of the method (*Trace.traceEntry()*) and after the method call is tracing the exit of the method (*Trace.traceExit()*). As within Java [7] dots are not allowed within operation names, it was soon discovered that dots could not be used to separate packages from classes and members. Therefore, we decided to separate them from each other using "\$\$". The "\$" character can be found quite often within AML and it has been chosen as it is rarely used by developers within class or member names.

AML is a simple and powerful notation for aspect-oriented modeling. In order to reduce errors when mapping models to code and offer low-level architecture design support, a code generator is developed which is presented in the next chapter.

4. CODE GENERATION

To extend the support beyond the architecture phase a code generator is presented addressing low-level design support by offering an automated mapping from abstract design models to programming models. This low-level architecture design support prevents inconsistencies among design and implementation and helps developers concentrate on AO design having the code skeletons generated automatically. AspectJ has been chosen to be the target language, as it is the AO language that is mainly used at present. The semantics of the connector have been designed according to AspectJ concepts including concrete mapping rules between model and code. Before generating code skeletons, the model is validated for syntactical and semantical correctness. It is even possible for developers to have the model validated without generating code afterwards.

The development of the code generator is divided into two parts (see Figure 4):

1. The *model validation* part validates an AO design model for syntactical and semantical correctness (e.g., the existence of referenced pointcuts). It is possible for developers to have the design model validated without generating code afterwards.
2. The *code generation* part generates AspectJ source code for a validated AO model.

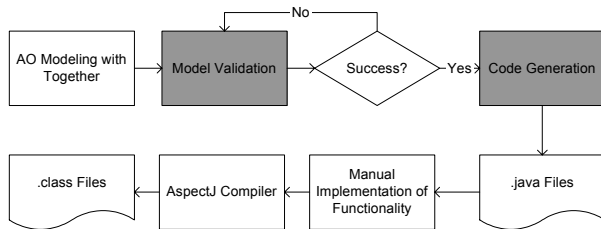


Figure 4: Flowchart of an AO Development Process

The CASE tool *Together* [22] from Borland is an enterprise development platform enabling application design, development, and deployment. It is extensible through an open Java API offering the possibility to develop custom software that plugs into the Together platform in the form of modules. The open API is composed of a three-tier interface that enables varying degrees of access to the infrastructure of Together. Altogether, Together's open API offers a lot of very powerful concepts for the manipulation of UML models and has therefore been chosen for the development of the code generator. The tool automatically validates and generates the OO parts of the model (aspect and base elements), the validation and code generation of AO parts (i.e. connector elements) is implemented as modules that plug into the Together platform.

Aspect elements and base elements map to Java source code. The aspect package and the base package are the OO parts of the notation. Connector elements map to AspectJ source code. The connector package consists of the AO part of the notation, linking aspect package and base package. To ensure syntactical and semantical correct AspectJ files that can then be compiled with an

AspectJ compiler mapping rules have been defined between the notation and AspectJ concepts.

```

public aspect TracingAspect {
    pointcut tracePointcut () :
        call ( * BasePackage.Server.getValues(..));

    before () : tracePointcut () {
        System.out.println ("Entering method...");
    }
    after () : tracePointcut () {
        System.out.println ("Leaving method...");
    }
}
  
```

Listing 1: Tracing Aspect with Copied Code

```

public aspect TracingAspect {
    pointcut tracePointcut () :
        call ( * BasePackage.Server.getValues(..));

    before () : tracePointcut () {
        Trace t = new Trace (/*parameters*/);
        t.traceEntry (/*parameters*/);
    }
    after () : tracePointcut () {
        Trace t = new Trace (/*parameters*/);
        t.traceExit (/*parameters*/);
    }
}
  
```

Listing 2: Tracing Aspect with Instantiation

Listing 1 shows the aspect `TracingAspect` that is generated when code parts of the crosscutting concern are copied into the AspectJ file. The difference between Listing 1 and Listing 2 lies in the specification of actions being performed at pointcuts. In Listing 1, the code to be executed (declared inside the aspect package) is copied into the AspectJ file, whereby in Listing 2 the relevant classes are instantiated and the appropriate methods are called. When instantiating classes (as shown in Listing 2), the appropriate constructor and method parameters have to be inserted by the user which is not necessary when copying the code. The user can choose between the two options when generating the code (copied code and instantiation).

The generation of AspectJ code is a one-time/one-way generation, possible future extensions could support roundtrip engineering including reverse engineering for aspect mining.

5. RELATED WORK

Related aspect-oriented design approaches proposed to provide support for crosscutting concerns at the architecture design level are based on Composition Patterns [12] [13] [14], Aspectual UML [10] [11] and other UML based modeling approaches [15] [16] [17].

The *Composition Pattern* approach combines UML templates with a subject-oriented model. The notation focuses on package level decomposition and “binds” crosscutting concerns with business logic classes with the help of binding relationships between the decomposed packages. The modeling language is based on standard UML extension elements like stereotypes, constraints or templates, which are supported on all standard UML conform CASE tools. The Composition Pattern notation does not provide an explicit notation for advice specifications, instead advices are expressed through state diagrams. A designer is forced to provide an additional state diagram for each execution point. While modeling the notation requires switching between object and state diagrams. The notation might be sufficient for small designs, but gets complex and hard to read for larger systems.

Aspectual UML separates the design in aspectual collaboration modules and all linking rules in a separate “connector” package. Compared to Composition Patterns, the notation enhances the separation of base classes, crosscutting concerns and binding rules in independent modules. However the UML notation of this approach introduces two new relationships on package-level (package inheritance and package adaption), which are unknown to standard UML and will be problematic to realize in existing CASE tools. With binding by delegation and advice weaving, Aspectual UML provides two powerful binding concepts, but is lacking other AO concepts like introduction and full support for all AspectJ-like join point definitions.

Many of the other modeling approaches [15], [16], [17] are based on class level decomposition. This decomposition level does not seem ideal, since often several classes are involved in one crosscutting concern. There is a danger that class level decomposition may lead to redundant notations and graphical tangling in the design models. [17] complies to standard UML, however the tight coupling of specific notations to AspectJ concepts, will make it difficult to support other aspect-oriented languages (e.g. HyperJ). [15] remains unspecific, how advice or pointcuts can be modeled. It mainly provides concepts for static crosscutting of operations. [16] provides limited modeling capabilities for crosscutting concerns e.g. advices can only be expressed through state-chart diagrams.

6. SUMMARY AND FUTURE WORK

This work addresses the AO development process from the high level architecture to the low level design by presenting an approach for aspect-oriented modeling and automated code generation. When considering the requirements defined in chapter 2, the following goals have been reached:

- An approach for high level architecture design, called AML, has been developed to enable separation of concerns at the design level of an AO development process. Within this approach it is assumed that the requirements have already been defined and specified during previous development stages.

- Since AML is UML conform, any CASE tool that supports UML modeling can be used.
- Aspects and base elements are completely kept apart; they are connected via a special language-specific connector element that encapsulates the underlying implementation technology. Any desired AO technology can be supported; it is just the connector’s syntax and semantics that have to be specified.
- Both, aspects and base elements, can be reused separately as the connector is the only crosscutting, language-dependent part. This sort of encapsulation offers a logical grouping of all classes belonging to one concern and eases the readability of design models as avoiding graphical tangling.
- To offer low-level architecture design support, a code generator has been developed to improve productivity and reduce errors when mapping model to code.

The work can be seen as a first step towards a simple and powerful modeling approach that fosters support from existing CASE tools since it is based on standard UML. AML in combination with the code generator should make AOSD more usable and more efficient for software development. The assumptions about the usefulness of the notation and the AO code generation have to be proven in the near future when using it in business development projects.

After evaluating the prototype’s features in real world development projects, some concepts may have to be added (e.g. complex relationships between aspects). Another important feature will be a complete CASE tool support including roundtrip engineering for aspect mining. As Together plans to support the development of modules offering roundtrip engineering features in the next version, this should be included in the next version of the code generator.

The connector package encapsulates the underlying implementation technology. Currently, the syntax and semantics of an AspectJ specific connector type are defined. This sort of encapsulation eases the replacement of the AO language, the support of different technologies and language concepts (such as Hyper/J [18] [19] [23]) will be part of some future work. An automated code generation for different languages is rather straightforward, too. It is only the code generator’s mapping rules that have to be changed.

There are still many issues to be solved until efficient AO development support comparable to current OO support is established. When offering an integrated development process, the gaps between the early phases and AO programming have to be filled as so far the paradigm focuses mainly at the implementation level. There is still a lot of challenging research to be done in the future until the paradigm is widely accepted and developers are aware of the benefits AOSD offers.

7. ACKNOWLEDGMENTS

We thank all researchers who explored the idea of crosscutting concerns to a state we could build on to gain a first experience in development projects, especially the group around AspectJ, Caesar [8] and Hyper/J. We also want to thank our colleagues at Siemens for their valuable feedback.

8. REFERENCES

- [1] E.W. Dijkstra. *A Discipline of Programming*. Prentice Hall, Englewood Cliffs, NJ, 1976.
- [2] G. Kiczales et al. *Aspect-Oriented Programming*. 2001
- [3] Early Aspects Homepage, <http://early-aspects.net>
- [4] J. Rumbaugh et al. *The Unified Modeling Language Reference Manual*. Addison-Wesley, Massachusetts, USA, 1999.
- [5] The Unified Modeling Language, version 1.4, <http://www.uml.org/>
- [6] AspectJ Homepage, <http://www.eclipse.org/aspectj/>
- [7] Java Homepage, <http://java.sun.com/>
- [8] CAESAR project, <http://caesarj.org/>
- [9] E. Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Massachusetts, USA, 2001.
- [10] S.Herrmann. *Composable Designs with UFA*. Submission to AOSD 2002.
- [11] S. Herrmann, M. Mezini. *Aspect-Oriented Software Development with Aspectual Collaborations*. Submission to ECOOP 2002.
- [12] S. Clarke et al. *Composition Patterns: An Approach to Designing Reusable Aspects*. Workshop on AO Requirements Engineering and Architecture Design on AOSD 2002, Enschede, Netherlands, 2002.
- [13] S. Clarke, R. J. Walker. *Separating Crosscutting Concerns Across the Lifecycle: From Composition Patterns to Aspect J and Hyper/J*
- [14] S. Clarke, R. J. Walker. *Towards A Standard Design Language for AOSD*
- [15] J. Suzuki, Y. Yamamoto, *Extending UML with Aspects: Aspect Support in the Design Phase*, Submission for Workshop at ECOOP 1999
- [16] J.L. Herrero, F. Sanchez, F. Lucio, M. Torro, *Introducing Separation of Aspects at Design Time*, Submission for Workshop at ECOOP 2000
- [17] R. Pawlak et al. *A UML Notation for Aspect-Oriented Software Design*. Workshop on AO Requirements Engineering and Architecture Design on AOSD 2002, Enschede, Netherlands, 2002.
- [18] Hyper/J, <http://www.alphaworks.ibm.com/tech/hyperj/>
- [19] Hyperspaces, <http://www.research.ibm.com/hyperspace/>
- [20] I. Jacobson et al, *Unified Software Development Process*. Addison-Wesley Professional, February 1999.
- [21] G. Kiczales. E.Hilsdale, J. Hugunin, M. Kersten, J. Palm and W. Griswold. *An overview of AspectJ*. In Proc. Of 15th. ECOOP, LNCS 2072, p. 327-353, Springer-Verlag, 2001
- [22] Together Homepage, <http://www.borland.com/together/>
- [23] H. Ossher and P. Tarr. *Multi-Dimensional Separation of Concerns and the Hyperspace Approach*. In Proceedings of the Symposium on Software Architectures and Component Technology: The State of the Art in Software Development. Kluwer, 2000.