Generating Aspect Code from UML Models

Iris Groher
Siemens AG, CT SE 2
Otto-Hahn-Ring 6
81739 Munich, Germany
Iris.Groher@fh-hagenberg.at

Stefan Schulze
Siemens AG, CT SE 2
Otto-Hahn-Ring 6
81739 Munich, Germany
Stefan.Schulze@siemens.com

ABSTRACT

This position paper presents a concept for aspect-oriented design and a seamless integration of AO design and implementation. We suggest a design notation based on standard UML which separates clearly the reusable programming language independent design of aspect code and base (business logic) code from the language dependant cross-cutting parts. Thus fostering reuse of aspect code and simplifying the replacement of the aspect-oriented implementation language. Additionally we ease the transition from design to implementation by defining the mapping from design model to implementation language and support automatic generation of aspect-oriented code skeletons from the design model.

Categories and Subject Descriptors

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

General Terms

Design, Languages

Keywords

AOSD, Aspect, AspectJ, cross cutting concerns, UML.

1. INTRODUCTION

Software design is an important step within the software development lifecycle. Object-oriented (OO) design has shown its strength when it comes to modeling common behavior; however OO design does not adequately address behaviors that span over many classes. Crosscutting requirements, such as persistence or security, including all well known problems they lead to, are present throughout the whole development lifecycle and therefore cause a reduction in the expected benefits of design. Aspect-oriented programming (AOP) addresses these problems at coding level and offers low-level support for separation of concerns as can be found e.g. in AspectJ [2][11], Hyper/J [8][9][14], LAC [7] or Caesar [12]. A lack of design support leads to a gap between design and implementation which worsens the desired results. To gain the AOP benefits at earlier stages in the software development lifecycle, similar separation capabilities must be provided also at design level.

This paper addresses the specification of crosscutting concerns at design level to maintain the separation of concerns earlier in the lifecycle. We adopted a terminology for aspect-oriented modeling (AOM) based on the core concepts presented in AspectJ [2][11] and the concepts of the design notation UFA [5][6]. Our work can be seen as a step towards a UML standardization on how to define aspects at the design phase of aspect-oriented software development (AOSD).

We present an extension to UML without changing its metamodel specification to achieve standard UML conformity. Our intention is to offer standard development tool support and interchangeability between various CASE tools, so we customized UML for supporting AOM only using its standard extension mechanisms (such as stereotypes, tagged values and constraints); see Section 2 for a detailed description. To gain the benefits of code and design reuse of aspect-oriented software, the ability to reuse aspect and base code separately is one of our key intentions. We offer a terminology where aspect and base elements are completely kept apart ; there is no direct connection between them (as proposed in UFA [5]). Thus aspects and base elements are reusable and independent of the implementation technology; for more details please see Section 3.

Our work addresses the aspect-oriented development process from design to code. Due to the fact that aspect support has been focused mainly at implementation level, we focus on design and present an automated mapping from design models to programming models. We offer validation and AspectJ code generation of models to avoid inconsistencies among design and implementation. This helps developers concentrate on aspect-oriented design having the code skeletons generated automatically to gain the benefits they are used to in object-oriented software development.

The sections of this paper are organized as follows: Section 2 describes the need for aspect-oriented design and the requirements for a notation allowing separation of concerns at design level. Section 3 describes the syntax and semantics of our notation. Section 4 presents the automated mapping between design model and AspectJ code. We conclude with a note on future work and a summary in Section 5 and 6.

Submitted for: Workshop on Aspect-Oriented Modeling with UML, AOSD, March 17-21, 2003, Boston, USA.

2. MODELING CROSS-CUTTING-CONCERNS IN THE DESIGN PHASE

Aspect-oriented software development is a new technology for separation of concerns (SOC) in software development. The techniques of AOSD make it possible to modularize crosscutting aspects of a system. AOP offers the low-level support for SOC, but there is a lack of high-level means for expressing aspects. Using AOSD in real-world development projects soon meets problems when defining aspects at the early analysis and design steps of an aspect-oriented application. Development of large software systems follows development processes that all include activities like requirements engineering, analysis, design and implementation. Following a process, such as the *Rational Unified Process* [15][10], and focusing on AOP at coding level is a paradigm shift between object-oriented design and aspect-oriented code. This leads to inconsistencies between design and implementation as the aspect-oriented paradigm is not seamlessly supported during design phase.

Aspects must be identified both at design and implementation phases to avoid their divergence. Our requirements for specifying crosscutting concerns at a higher level of abstraction are the following:

- The terminology should be simple to understand and straightforward to use for developers being familiar with design notations like the UML.
- Design modeling should be supported by powerful CASE tools to improve developer productivity and ensure syntactical correctness of the model.
- The notation should support the most common aspect-oriented approaches and languages. A direct mapping between design model and supported programming model must be possible and straightforward.
- The direct mapping between the notation and the supported implementation language should allow automatic code generation based on the design model.
- It should be applicable in real-world development projects and offer the capability of modeling large systems.

2.1 Standard UML Extension Mechanisms

UML is acquainted to be the industry-standard modeling language for the software engineering community; using standard UML improves developer productivity, offers high acceptance, broad development tool support and interchangeability among tools. Developers can model using familiar tools and environments and therefore gain all the benefits they are used to in object-oriented design. Using UML and its built-in extension mechanisms as a modeling language for supporting AOM fulfills the first and the second requirement. UML is an extensible modeling language to enable domain specific modeling. It offers a set of built-in extension mechanisms to customize the UML for a specific domain, e.g. aspect-oriented modeling. Model elements can be customized and extended with new semantics by using stereotypes, constraints, tag definitions, and tagged values. The principal extension mechanism is the concept of stereotype which provides a way of classifying model elements as if they were instances of new virtual metamodel constructs. These model elements have the same structure (attributes, associations, operations) as similar non-stereotyped model elements of the same kind [13].

2.2 Mapping AspectJ Concepts

AspectJ, an aspect-oriented extension to the Java language, is one of the most common aspect-oriented languages. Therefore we chose to adopt AspectJ concepts for the implementation language dependent parts of our notation. This enables us to automatically generate AspectJ code skeletons from a design model to avoid inconsistencies between design and implementation which adds to the seamless support of the whole aspect-oriented development process. A mapping between model and code must be straightforward and automated to enable developers to be as productive as they are used to when the developing object-oriented systems. In the future we will add further support of other implementation languages similar to AspectJ (such as AspectC++ [1], AspectR [3] and AspectS [4]) to our notation which can easily be achieved by changing the well separated implementation language dependant part of our notation and by changing the mapping rules from model to code. The support of AO concepts diverging from AspectJ (such as HyperJ [8]) should be considered and is part of some future work; see Section 5 for more details.

3. MODELING ASPECTS IN UML

An appropriate notation should consider the fact that crosscutting concerns affect multiple classes in a system. Since a concern itself can consist of several classes (e.g. security concern) and since all of these classes may be associated with the classes the concern crosscuts we suggest the module construct for a concern to be higher-level than a class. Otherwise associations modeled on class level would supersede the logical grouping of all classes belonging to one concern. This would make the readability of the design quite difficult and lead to a “graphical tangling” of cross-cutting concerns instead of a clear separation. Therefore we base our notation on the work on UFA [5][6], which suggests package level (de)composition.

We adopted the underlying concepts of UFA and added additional support for AspectJ concepts (e.g. introduction mechanisms), since UFA is not specifically designed to support AspectJ. As our goal was to leverage the power of UML and existing CASE tools we modified the syntax of UFA to achieve UML standard conformity. In the following we will present the resulting notation and its syntax.

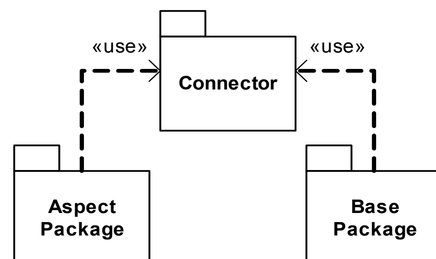

Figure 1: Package Level (De)Composition

The notation includes a base package (containing the business logic), an aspect package (containing the crosscutting concern)[1] and a connector to link aspect and base elements. This separation enables high reusability of the aspect and base package code since the connector is the only crosscutting element. Additionally the connector encapsulates the underlying implementation technology which eases its replacement (e.g. replace of AspectJ with AspectC++). The aspect can be modeled independently from any design it may potentially crosscut; the connection between base design and aspect design is specified separately from the aspect. Both aspect and base elements can be modeled using common UML elements, such as classes or interfaces.

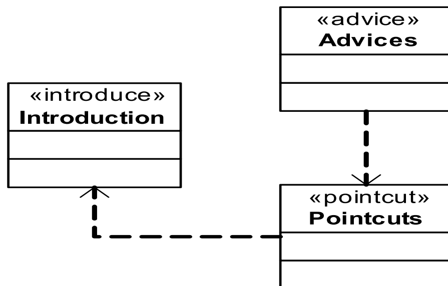

Figure 2: Connector Package

The connector includes AOP's core concepts and benefits, e.g. specifying execution points in a program, actions to be performed at those points and type-modification constructs. Every design model must use a specific connector type that maps to the AOP technologies to be used for the implementation. The connector type described in the following (see Figure 2) is designed according to AspectJ concepts.

The AspectJ connector defines pointcuts (i.e. execution points), advices (i.e. actions to be performed at pointcuts) and introductions (i.e. type-modification constructs). All these elements are defined syntactically as methods of the class they belong to. E.g. a *before* **advice** used to trace method calls at pointcut *login* would be declared as a method of the connector's *Advice* class:

<<before>> login(Tracer.trace)

The stereotype *before* indicates an AspectJ's *before* advice. Our notation contains similar stereotypes for AspectJ's *after* and *around* advices. The parameter "*Tracer.trace*" of the advice specifies the method *Tracer.trace* to execute at pointcut *login*.

The **pointcut** definition in the connector's pointcut class would be:

login(<!>com.siemens.UserManager.login)

Where the "*<!>*" construct marks pointcuts on method calls and the "*com.siemens.UserManager.login*" parameter specifies the execution point the pointcut matches to (here the *login* method).

Our notation defines further constructs for AspectJ's possibilities to intercept method executions, constructor calls, throwing of exceptions, property access etc. We also support AspectJ's wildcards for pattern matching when defining a pointcut (e.g. **.*.UserManager.** for all methods of a class called *UserManager*).

3.1 Design Example

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

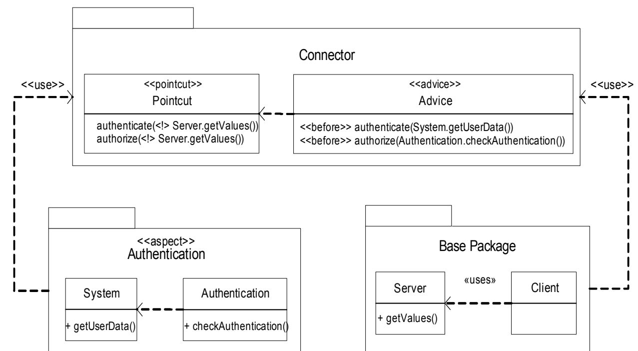

Figure 3: Security Design Example

Every time the user performs an invocation on the Server, he is authenticated. Both base package and aspect package are independent from each other, no connection is modeled inside. The connector, specifying the weaving rules, includes program execution points (pointcuts) and actions to be performed at those points (advices). The pointcuts are triggered every time the client invokes the *Server.getValues()* method, the actions to be performed before the method call are reading and checking the user data to authenticate him. Both base elements and authentication aspect are reusable and independent of the implementation technology. This example can be modeled using any CASE tool that supports standard UML.

4. GENERATION OF ASPECTJ CODE

The generation of AspectJ code should raise the acceptance of our notation as it offers an automatic mapping from design models to concrete implementations. Code generation improves developer productivity, ensures syntactical correctness and reduces errors when mapping model to code. We have chosen AspectJ to be our target language as it is the aspect-oriented language that is mainly used at present. The generation is done following concrete mapping rules between model and AspectJ concepts. As the AspectJ compiler produces class files that comply to the Java byte code specification any compliant Java Virtual Machine (JVM) can interpret the produced class files. While designing the code generator we have evaluated some possible technologies:

- Code generation based on design model data in XMI (XML Metadata Interchange).
- Code generation through a design pattern approach supported by advanced CASE tools.
- Code generation as an integral part of a CASE tool based on the APIs of the specific CASE tool.

The main purpose of XMI is to enable easy interchange of metadata between modeling tools (based on the OMG-UML) and

[1] To simplify AspectJ code generation we currently focus solely on the scenario described in Figure 1 which applies only one aspect to a base package that contains the business logic.

metadata repositories (OMG-MOF based) in distributed, heterogeneous environments [18]. A possible solution would have been to write a code generator that parses XMI and generates the aspect-oriented code. Because in this case the whole code including the object-oriented base elements (such as classes and interfaces) would have to be generated by our code generator, we opted against this solution.

The second alternative would have been to use a tool like Rational XDE from Rational [16] that allows the user to define proper patterns and code templates for these patterns. Thus we would have defined a pattern and code template per connector element. The user-defined-patterns concept of the Rational XDE showed not to be powerful enough which is mainly due to the limited code generation possibilities. It is currently not possible to influence the weaving of the user-defined-patterns code templates and the code generated by Rational XDE's code generator.

Finally we chose the third alternative and selected the CASE tool Together from Borland which is extensible through an open Java API [17]. It offers the possibility of developing custom software that plugs into the Together platform in the form of modules. A module in this case is an assembly of Java classes that offer APIs defined by Together and are registered with the CASE tool. The API is composed of a three-tier interface that enables varying degrees to access the native infrastructure. The tool automatically validates and generates the object-oriented base elements such as classes and interfaces. The aspect-oriented validation and code generation are implemented as modules.

The development of the code generator is divided into two modules:

- Model validation: validates an aspect-oriented design model for syntactical and semantical correctness (e.g. the existence of referenced pointcuts). It is possible for the user to validate a design model without generating code afterwards.
- Code generator: generates AspectJ code for a validated aspect-oriented model.

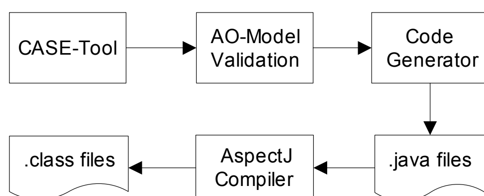

Figure 4: Code Generation Model

The AspectJ code generation is a one-time/one-way generation, the development of modules offering roundtrip engineering is not yet supported by Together's Open API. It is planned for future versions. The generated code templates can then be completed by the user and compiled using AspectJ's aspect weaver.

5. FUTURE WORK

As our work is a first step towards aspect support in design, there is still some related work to do. We will continue to refine the notation while testing it in larger development projects to get feedback concerning our work.

Future extensions of the code generator could support roundtrip engineering including reverse engineering for aspect mining. Currently Together's Open API does not support the implementation of a module for roundtrip engineering, but it is planned to be provided in the next version.

An automated code generation for other languages similar to AspectJ like AspectC++ [1] or AspectS [4] is planned. Such an extension to our notation and the code generator should be straightforward; only the code generator mapping rules should have to be specified.

Even if AspectJ is currently the best known AO language there are many other promising AO languages that are based on different concepts like HyperJ [8][9], where the system functionality is implemented in separate Hyperslices, or Caesar [12], where an aspect is regarded as a number of abstractions collaborating to realize a certain concern in the system. The authors will investigate, if their approach can also be mapped to these AO language flavors.

6. SUMMARY

Aspect-oriented software development is missing standardized concepts in the design phase. To make AOSD more widely accepted we have to offer solutions for designing cross-cutting concerns and we have to integrate the different phases of the AOSD life-cycle more smoothly by supporting the aspect-oriented paradigm in every phase.

The design notation presented here is a first step towards a simple and powerful notation that fosters support from existing CASE tools since it is based on UML. The notation in combination with the code generator we make AOSD more usable and more efficient for software development.

A first prototype of the code generator described in chapter 4 is currently under development. We expect to prove our assumptions about the usefulness of our notation and the aspect-oriented code generation in the near future using this prototype in development projects.

But there are still many issues to solve until we have as efficient development support as it is already common for object-oriented software development. The future improvements of our work concern the notation that should support more complex relationships between several aspects and it also concerns a complete CASE tool support as e.g. roundtrip engineering.

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, Hyper/J and UFA. We also want to thank our colleagues at Siemens for their valuable feedback.

8. REFERENCES

[1] AspectC++, http://www.aspectc.org/

[2] AspectJ, http://www.eclipse.org/aspectj/

[3] AspectR, http://aspectr.sourceforge.net/

[4] AspectS, http://www.prakinf.tu-ilmenau.de/~hirsch/Projects/Squeak/AspectS/

[5] S.Herrmann. Composable Designs with UFA. Submission to AOSD 2002.

[6] S. Herrmann, M. Mezini. Aspect-Oriented Software Development with Aspectual Collaborations. Submission to ECOOP 2002.

[7] S. Herrmann, M. Mezini. Combining Composition Styles in the Evolvable Language LAC. Submission for ASoC Workshop at ICSE 2001.

[8] Hyper/J, http://www.alphaworks.ibm.com/tech/hyperj

[9] Hyperspaces, http://www.research.ibm.com/hyperspace/

[10] I. Jacobson, G. Booch, J. Rumbaugh. Unified Software Development Process. Addison-Wesley Professional, February 1999.

[11] 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

[12] M. Mezini, K. Ostermann. Conquering Aspects with Caesar. To appear in Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD), March, 2003, Boston, USA.

[13] OMG Unified Modeling Language Specification, version 1.4, September 2001

[14] 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.

[15] Rational Unified Process (RUP) http://www.rational.com/products/rup/index.jsp

[16] Rational XDE™, http://www.rational.com/products/xde/index.jsp

[17] Together, http://www.togethersoft.com/

[18] XMI specification version 1.2, http://cgi.omg.org/docs/formal/02-01-01.pdf