Modeling Aspects: An Implementation-Driven Approach

Wesley Coelho
Department of Computer Science
University of British Columbia
201-2366 Main Mall, Vancouver BC
Canada V6T 1Z4
coelho@cs.ubc.ca

Gail C. Murphy
Department of Computer Science
University of British Columbia
201-2366 Main Mall, Vancouver BC
Canada V6T 1Z4
murphy@cs.ubc.ca

ABSTRACT

Model-Driven Software Development (MDSD) and Aspect-Oriented Programming (AOP) are two emerging software engineering paradigms that have developed independently. We believe that these approaches can be combined to produce better solutions to problems such as product line engineering. To integrate AOP with MDSD, techniques for modeling aspect-oriented constructs must be developed. As a first step, we consider the problem of modeling aspects in a static program structure. We believe that the limitations of passive notations inhibit the effectiveness of aspect modeling techniques that have been previously proposed. Therefore, we consider the use of interactive and dynamic software visualizations for aspect modeling. In this paper we describe an approach to modeling the influence of an aspect in an existing system.

1. INTRODUCTION

Aspect-Oriented Programming (AOP) [6] enables the modularization of crosscutting concerns that span an existing modularization of a system. A new unit of modularity called an Aspect is used to encapsulate a crosscutting concern. The benefits of improving the modularity of a program include improved comprehensibility and reusability. AOP has been applied to various programming problems including synchronization, logging, error checking and handling, persistence, as well as design issues, such as capturing design patterns [5].

AOP technology can also be used to support software product line engineering [7]. The Model-Driven Software Development (MDSD) paradigm has also been applied in this area. We believe that the different approaches supported by these technologies can be integrated to improve product line engineering, as well as solve other programming problems.

To take advantage of Aspect-Oriented Programming in Model-Driven Software Development, it is necessary to model the additional concepts introduced by AOP. As a first step, we consider the notation required to describe an aspect in an existing system. More specifically, we are interested in modeling the static structure of an aspect-oriented program in the way that a Unified Modeling Language (UML) class diagram models the structure of an object-oriented program. Although this notation describes an existing implementation, we believe our approach provides a foundation on which to understand how to model aspects a priori in a meaningful way.

Several approaches have been proposed for describing aspect-oriented concepts in a static program structure diagram [3, 8, 9, 4]. Although there is considerable variability in the proposed notations, each approach models AOP with a passive view that can be presented on paper.

We believe that the nature of the crosscutting concerns modeled by aspect-oriented constructs makes them difficult to display cleanly in a single passive view. Therefore, we consider the application of dynamic and interactive software visualizations to AOP modeling. Although this approach is not limited to one aspect-oriented programming language, we have chosen AspectJ [1] as an initial target language for our implementation-driven models.

We describe several passive notations that have been proposed for AOP modeling as well as their limitations in the following section. Section 3 describes our software-supported approach to AOP modeling. We outline our plans for future work in section 4 and conclude in section 5.

2. PASSIVE ASPECT NOTATION

The majority of notations proposed for modeling AOP constructs are based on the Unified Modeling Language (UML). Indeed, several are created using UML's extensibility constructs and are therefore valid UML. These UML-based approaches can be broadly categorized as examples of two high-level strategies: 'Aspect as Classifier' and 'Aspect Binding.'

In the first strategy, a new classifier is introduced to represent aspects in a manner similar to how a class is modeled in UML, e.g. [9, 8]. This classifier, stereotyped as an <<aspect>>, is used to encapsulate crosscutting behavior. These aspect classifiers are then associated with the program elements they affect by drawing lines connecting them.

A disadvantage of this approach is that the crosscutting concerns often affect a high number of entities in the model. To display this situation, it is necessary to draw many line connections. This could rapidly result in "graphical tangling" of crosscutting concerns and a consequent reduction in model readability [4].

Another category of aspect modeling approaches is to describe crosscutting behavior separately from the base design. A binding mechanism is then used to associate crosscutting behavior with base program elements. For example, Clarke and Walker [3] use class diagrams and sequence di-

agrams in UML templates to encapsulate crosscutting behavior. Parameters to the template and binding statements are then used to associate this behavior with the base design elements. Groher and Shulze [4] separate concerns into packages. A base package contains the model without crosscutting concerns, which are defined in a separate package. A third package is used to specify how the crosscutting functionality is bound to the base design.

An advantage of the binding approach is that it achieves a clean separation of base and crosscutting concerns. However, this separation can make it difficult to quickly understand the influence of a crosscutting concern on the base design. Parsing the bind statements and searching for the base elements and corresponding crosscutting behavior imposes a high cognitive load. This increases the time required to understand the model and introduces the possibility that important program characteristics will be overlooked.

These aspect modeling strategies highlight the conflict between demonstrating the influence of an aspect on the base design and avoiding graphical tangling. This conflict will inevitably arise when aspects are modeled with a single passive view. Since it is sometimes important to show the detailed influence of an aspect, and yet at other times it is important to abstract the influence, no passive modeling approach is likely to be ideal for this task.

These limitations of previous modeling techniques suggest that software-supported visualizations may be more appropriate for aspect modeling. The rich functionality that is possible with a software visualization can allow for dynamic views that can either show the influence of an aspect on a base design or abstract it to avoid tangling, depending on the user's changing interests. The use of color and interactive features can further improve the quality of the modeling experience. In the remainder of this position paper, we present an approach to modeling aspects with the aid of software visualization.

3. DYNAMIC ASPECT DIAGRAMS

Our approach to modeling an aspect is to provide a dynamic diagram of the effect an aspect has on a system to which it is applied when requested by the modeler. By default, the dynamic diagram displays a minimal overview of an aspect's influence. The minimal diagram can be expanded to show additional details of the target aspect's influence. Color-coding is used to link base program elements with the crosscutting behavior that is associated with them. Some elements of the diagram are constantly visible, while others become visible in response to user interaction such as mouse hovering.

As an aspect modeling example, we have selected a simple AspectJ aspect (Figure 1) and included a snapshot sketch of a corresponding dynamic aspect diagram in Figure 2. This aspect implements customer billing in an example telecommunications application.

3.1 Notation Overview

Unlike most other approaches to modeling aspects, our notation is based only loosely on UML, though there are many similarities. We have not used the UML's extensibility features but instead focused on modifying UML constructs to support aspect modeling.

```
public aspect Billing {
    public static final long LOCAL_RATE = 3;
    public static final long LONG_DISTANCE_RATE = 10;

    public Customer Connection.payer;
    public Customer getPayer(Connection conn) {
        return conn.payer;
    }

    // Caller pays for the call
    after(Customer cust) returning (Connection conn):
        args(cust, ..) && call(Connection+.new(..)) {
        conn.payer = cust;
    }

    //Connections give the appropriate call rate
    public abstract long Connection.callRate();

    public long LongDistance.callRate() {
        return LONG_DISTANCE_RATE;
    }
    public long Local.callRate() {
        return LOCAL_RATE;
    }

    // When timing stops, calculate and add the charge
    // from the connection time
    after(Connection conn): Timing.endTiming(conn) {
        long time = Timing.aspectOf().getTimer(conn)
             .getTime();
        long rate = conn.callRate();
        long cost = rate * time;
        getPayer(conn).addCharge(cost);
    }

    // Customers have a bill paying aspect with state
    public long Customer.totalCharge = 0;
    public long getTotalCharge(Customer cust) {
        return cust.totalCharge;
    }

    public void Customer.addCharge(long charge){
        totalCharge += charge;
    }
}
```

Figure 1: An example aspect that defines customer billing behavior in a telecommunications application.

The notation introduces an additional classifier to represent aspects. The intent of the aspect classifier is similar that of a UML 'class' classifier except it encapsulates additional member types and relationships necessary for modeling AspectJ programs.

The members that can be defined in an aspect classifier include inter-type declarations, `declare parents` statements, advice, and pointcuts. In AspectJ, inter-type declarations are used in aspects to introduce fields and methods on other types. Similarly, declare parents statements are used to cause a type to implement or extend another type. AspectJ advice are method-like constructs that implement crosscutting behavior. Pointcuts are statements that indicate where crosscutting behavior is to be applied.

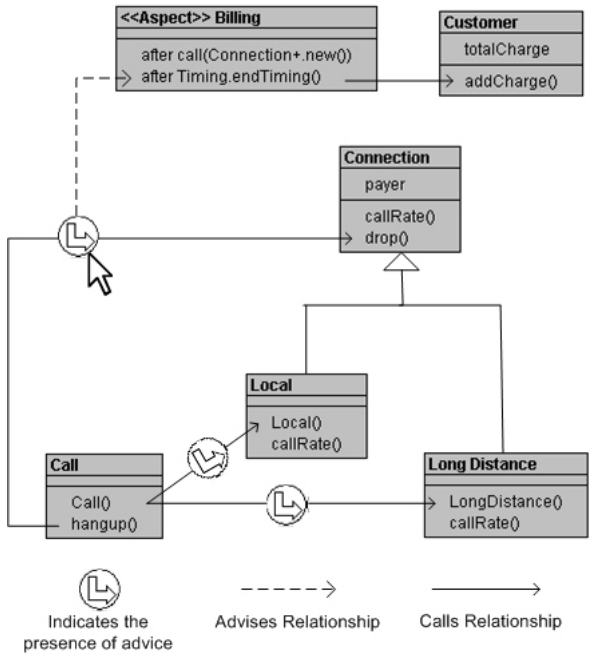

Figure 2: A snapshot sketch of the dynamic aspect diagram for the Billing aspect in Figure 1. In this sketch, the user's mouse is hovering over an advice indicator icon, which has caused a relationship to the contributing advice declaration to be drawn.

Aspect classifiers in the dynamic diagrams support two relationships not found in standard UML. The 'advises' relationship associates an advice body with the location in the static structure where its behavior applies. A call relationship is also used, which is a special case of a UML association that indicates the existence of a method call.

Our dynamic aspect diagrams use icons to indicate the presence of crosscutting behavior. For example, behavior that is executed after (as opposed to before) a point in the base program's execution is indicated with the arrow icon shown in Figure 2. Additional icons are used to depict other AspectJ constructs.

Relationship granularity is an important difference between UML class diagrams and our dynamic aspect diagrams. In the UML, relationships connect classifiers to other classifiers. For example, a class might aggregate another class. In dynamic aspect diagrams, we also include member-level relationships. Method calls from a method in one class to a specific method in another class can be drawn. Similarly, advises relationships terminate at the signature of the advice that provides the behavior rather than the aspect containing the advice.

When aspect diagrams are first generated, they display an initial overview of an aspect's influence. If the model user wishes to see more details, a '+' operator is available to expand the diagram. The expanded diagram typically includes several more relationships or members that are likely to be relevant.

3.2 Initial Diagram Generation

The initial diagram of an aspect or set of aspects is generated by reading certain members in the aspect code and adding a representation of their impact on the program to the diagram. The members that are considered during the initial diagram are those that are directly involved in applying changes to the base program. Thus, inter-type declarations, `declare parents` statements, and the effects of advice are considered while private fields and methods of the aspect are not included in the initial diagram.

3.2.1 *Inter-type Declarations*

Inter-type declarations in AspectJ are statements in aspects that declare fields or methods to be part of another type. These statements directly impact the base program and are therefore always included in the initial aspect diagram generation. The declared members are shown as members of the target classifier where they have been introduced.

3.2.2 `declare parents` *Statements*

AspectJ `declare parents` statements cause one type to implement or extend another. When such a statement is encountered in the aspect, the relevant subtype and supertype are added to the diagram if not already present. These types are then connected by the appropriate 'extends' or 'implements' relationship.

3.2.3 *Advice*

AspectJ 'advice' are method-like constructs that implement crosscutting functionality. In dynamic aspect diagrams, advice are shown as a member in the aspect that declares them. More importantly, program elements closely associated with the static shadow of an advice are also included in the diagram.

The static shadow of an advice is the set of all locations in the static program structure where the advice body might be applied at run-time. For example, consider a `call` advice that advises the call sites at which a specified method is invoked. In this case the call itself is being advised. We show this by adding a calls relationship from the advised call site to the target method. The calls relationship is decorated with an icon indicating that it is being advised (e.g. the arrow icon in Figure 2). In some cases, this icon could be color-coded to correspond to the advice body that advises it. An 'advises' relationship is also drawn from this icon to the advising advice signature. However, this connection is only shown when the user's mouse hovers over it to avoid graphical tangling.

3.3 Abstraction and Examples

Despite the ability to hide 'advises' relationships until they are requested, dynamic aspect diagrams could quickly become cluttered with program elements that are influenced by the aspects the user is investigating. For example, consider a performance profiling aspect that counts the number of times each method in the program is called. The complete influence diagram for such an aspect would include all methods in the system. For any non-trivial program, the resulting diagram would be excessively cluttered with classes and methods.

However, the inclusion of all of these methods is not required for understanding the effect of the aspect. Therefore, it is possible to introduce a level of abstraction. A single aggregate classifier and method can be used to represent the set of all classifiers and methods that are advised by the aspect. In the resulting profiling aspect diagram, an aspect's advice can be shown to advise the call between two methods that abstract a large underlying set of actual methods advised. If desired, the user can query the aggregated classifiers or methods to retrieve a list of the actual elements they represent. This approach can significantly reduce the diagram's complexity.

To aid program comprehension, it may also be beneficial to show an example of the abstracted entities and relationships. This concrete example is displayed along with the abstracted entities that represent all other instances of the element being displayed.

3.4 Automated Diagram Expansion

The initial dynamic aspect diagram provides an overview of how the system is affected by the selected set of aspects. In some cases, the user may wish to know additional details of the aspects' influence. However, it may not always be clear what additional program elements and relationships are important.

We believe that the initial aspect influence diagram can be expanded automatically to show additional relevant information about the aspect and its impact on the system. For example, it may be informative to show the user additional members of the aspect or other classifiers in the diagram. Additional relationships may also be relevant. In particular, calls made from an advice body may be a key part of the concern specified by the aspect. In Figure 2, the 'calls' relationship between the second member (an advice) is an example of a diagram addition resulting from a request to expand the diagram.

Dynamic aspect diagrams are expanded to show additional details by invoking a '+' operator. This operator takes as input the existing diagram and adds to it several additional details of the target aspects' influence. The '+' operator can be invoked repeatedly until the desired level of detail is reached.

Not all additional elements are likely to be equally relevant. Therefore, when the '+' operator is invoked it must rank possible additions according to their potential information value. Possible additions are ranked by their element or relationship type as well as other characteristics such as their parent's rank. For example, a method call from an advice body to a method introduced on another class will have a higher rank than a class's private field. The '+' operator may also elect to show additional details by replacing aggregate entities with the previously abstracted concrete classifiers, members, or relationships.

4. FUTURE WORK

We are currently developing a tool that will generate dynamic aspect diagrams for AspectJ. The tool is implemented as a plug-in for the extensible Eclipse software development environment [2]. This tool will use icons, color-coding, and 'hovering' to assist with the aspect influence presentation. We will also provide an implementation of the '+' operator as well as basic diagram abstraction support. An early prototype of this tool will be used to investigate heuristics for default diagram generation as well as diagram expansion.

When a functioning prototype has been developed, we intend to perform an initial evaluation of this approach by conducting a small-scale study. The study will investigate whether users can more effectively implement a software modification task using the tool rather than with traditional software engineering tools alone. If use of the tool is associated with better change task implementations, then this will support the hypothesis that dynamic aspect diagrams are an effective technique for modeling aspects.

5. CONCLUSION

We have shown that passive notations for modeling aspects either do not adequately show the influence of aspects or have a tendency to be encumbered by graphical tangling. Since this tradeoff is inevitable with single-view, passive modeling approaches, we suggest that software visualizations may be more effective. We presented dynamic aspect diagrams as a technique for modeling the influence of an aspect on an existing software system. Dynamic aspect diagrams use color, user interaction, and dynamic expansion to model aspects. We believe that this approach clearly displays the effects of an aspect while avoiding graphical tangling.

6. REFERENCES

[1] AspectJ. http://www.eclipse.org/aspectj/, August 2004.

[2] Eclipse. http://www.eclipse.org/, August 2004.

[3] S. Clarke and R. J. Walker. Composition patterns: An approach to designing reusable aspects. In *International Conference on Software Engineering*, pages 5–14, 2001.

[4] I. Groher and S. Schulze. Generating aspect code from UML models. In F. Akkawi, O. Aldawud, G. Booch, S. Clarke, J. Gray, B. Harrison, M. Kandé, D. Stein, P. Tarr, and A. Zakaria, editors, *The 4th AOSD Modeling With UML Workshop*, 2003.

[5] J. Hannemann and G. Kiczales. Design pattern implementation in java and aspectj. In *Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 161–173. ACM Press, 2002.

[6] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Akşit and S. Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

[7] T. Kishi and N. Noda. Aspect-oriented analysis for architectural design. In *Proceedings of the 4th international workshop on Principles of software evolution*, pages 126–129. ACM Press, 2002.

[8] D. Stein, S. Hanenberg, and R. Unland. Designing aspect-oriented crosscutting in UML. In *AOSD-UML Workshop at AOSD '02 (Enschede, The Netherlands, Apr. 2002)*.

[9] J. Suzuki and Y. Yamamoto. Extending UML with aspects: Aspect support in the design phase. In *ECOOP Workshops*, pages 299–300, 1999.