ActiveAspect: Presenting Crosscutting Structure

Wesley Coelho
Department of Computer Science
University of British Columbia
Vancouver, BC, Canada
coelho@cs.ubc.ca

Gail C. Murphy
Department of Computer Science
University of British Columbia
Vancouver, BC, Canada
murphy@cs.ubc.ca

ABSTRACT

Developers must often deal with concerns that crosscut a program's structure. Understanding the crosscutting structure may help a developer understand and work with the concern. Current tools for presenting crosscutting structure suffer either from graphical complexity or a mismatch between the presentation and the underlying program structure. We introduce a concern presentation approach that overcomes these problems through a combination of automatic abstraction and interactive features that enable a developer to investigate relevant details. We sketch how the ActiveAspect tool we are developing implements this approach for presenting the crosscutting structure described by aspects in AspectJ.

1. INTRODUCTION

Many concerns that a developer must understand and manage at the code level involve crosscutting structure. For example, many modification tasks involve changing code spread across multiple classes and methods [15, 16]. Several approaches exist to help a developer identify the crosscutting structure relevant to a concern [4, 5, 8, 10, 13, 17].

Approaches to present crosscutting structure use one of two techniques: tree views or structure diagrams. Tree views use screen space effectively but are not well suited to presenting structure with multiple, often non-hierarchical, kinds of relations. Existing structure diagrams (e.g., UML class diagrams) can intuitively present such structural patterns but often exhibit substantial graphical complexity.

Existing tools that use these approaches are further limited by using a static display of information. A fixed subset of the program structure is presented and cannot be easily adjusted to meet the developer's requirements, often resulting either in information overload or a lack of needed detail.

To overcome these problems, we are investigating a structure diagram presentation approach that combines automatic the abstraction of structure with user interaction. The automatic abstraction enables important structure to be highlighted while avoiding unnecessary graphical complexity. The user interaction features allow developers to investigate details as required in the context of the presented structure. The diagrams can also be expanded automatically to show additional context associated with a concern.

There are several ways to obtain a subset of the program structure belonging to a concern. Currently, we focus on the presentation of crosscutting concerns as captured by aspects in AspectJ [1]. However, we believe our approach generalizes to present concerns described in other ways, for example, as a concern graph [16].

We are developing the ActiveAspect tool as a preliminary implementation of our approach. ActiveAspect is implemented as a plug-in for Eclipse [3] that presents the crosscutting structure of an aspect when requested by the developer.

In the next section we review existing techniques for presenting crosscutting program structures and discuss their limitations. In section 3 we describe our approach to presenting concerns defined by AspectJ aspects. We conclude in section 4.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Workshop on Modeling and Analysis of Concerns in Software (MACS 2005)
16 May 2005, St. Louis, MO, USA
Copyright 2005 ACM 1-59593-119-8/05/05 ...$5.00.

2. EXISTING PRESENTATIONS

Most of the current approaches for showing crosscutting structure are based either on tree displays or graphical node-link structure diagrams.

2.1 Tree Views

Several tools use tree views to present crosscutting concerns [4, 5, 8, 13]. These views display program elements according to a hierarchical relationship and are rooted at an element of interest. Child elements can be displayed as required by expanding nodes along an arbitrary path of the tree.

A representative example of the tree view approach is found in the AspectJ Development Tools (AJDT) [4]. AJDT is a plug-in for the Eclipse environment that provides developers with information about the crosscutting relationships associated with AspectJ aspects. This information is presented primarily using a number of aspect-aware tree views such as an outline view, inheritance hierarchy, call hierarchy, and package explorer[1].

[1] AJDT also provides gutter annotations that show local crosscutting information for a particular program element as well as the Aspect Visualizer, which provides a SeeSoft [11] view indicating the extent to which source files are affected by an aspect.

Using tree views to present concerns has several drawbacks. Since each tree presents a single hierarchical relationship, it can be necessary to view multiple trees to investigate all relevant relationships for a program element. The developer must manually find a program element of interest in several potentially stacked tree views and synthesize the information provided in each. This task is further complicated by the need to mentally disentangle program elements relevant to the concern from all other elements that are also shown in the tree views[2].

Another important drawback that is common to all tree-based concern presentations is the mismatch between the structure of the tree view and the underlying program structure it represents. In general, trees are hierarchies of a single relationship type. They are therefore best suited to presenting a single program relationship that is strictly hierarchical, such as inheritance. However, the structure of a concern is not hierarchical, but rather a directed graph of program elements and relationships that may be cyclic. This mismatch between presentation structure and program structure requires the developer to mentally perform the mapping from display structure to actual program structure. Furthermore, structural patterns involving multiple relationship types become virtually impossible to detect. These limitations of tree views impose a cognitive load on the developer that increases the time and effort required to understand a concern and raises the likelihood that relevant information will be overlooked.

2.2 Static Structure Diagrams

For the purpose of this discussion we broadly define static structure diagrams as node-link diagrams where the nodes are program elements such as methods and classes and the links are program relationships such as method invocations or inheritance. We further specify that the program information in the static structure diagram is known at compile time. This definition includes but is not limited to UML class diagrams.

Static structure diagrams improve on tree-based presentations because they show directly the structure of a concern. Structural patterns can be identified easily and multiple relationship types can be accommodated. This allows developers to explore a concern by looking at a single view.

There are several tools that can be used to produce a static structure diagram of a concern [2, 6, 7, 9]. Unfortunately, these diagrams typically suffer from excessive graphical complexity when used to present crosscutting structures that influence many parts of a system. In this case the result is a diagram that is cluttered with nodes and relationships, making it difficult for the developer to determine which part of the structure is relevant for their current task.

Several aspect-oriented modeling approaches have been proposed that are designed specifically for modeling crosscutting concerns captured by aspects. These approaches can be broadly categorized as examples of two high-level strategies: *aspect 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 [17, 18]. 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 from aspects to the entities they influence. This results in the same graphical complexity problems exhibited by existing structure diagramming tools.

The second aspect modeling strategy is to describe crosscutting behavior separately from the base design [10, 12]. A binding mechanism is then used to associate crosscutting behavior with entities in the base program. For example, Clarke and Walker [10] use class diagrams and sequence diagrams 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.

An advantage of the binding approach is that it achieves a clean separation of crosscutting concerns. However, this separation can make it difficult to quickly understand the influence of a crosscutting concern. Parsing the bind statements and searching for the base elements and corresponding crosscutting behavior is difficult for the developer; increasing the time required to understand the concern and the possibility that important information will be overlooked.

3. ActiveAspect

Our concern presentation approach uses a concept we call *active models* to retain the benefits of existing structure diagrams while avoiding excessive complexity. Active models are created automatically from a description of the crosscutting structure of interest when requested by a developer. Active models use a variety of user-interaction and automated features to enable a developer to explore the crosscutting structure in a view.

To investigate the active model concept, we are developing ActiveAspect, an Eclipse plug-in that presents crosscutting structure described in an AspectJ aspect. A developer invokes ActiveAspect on a particular aspect using a context menu in the IDE. This action creates an active model and displays a graphical representation of the model in a view. The presented structure diagram notation is similar in appearance to a UML class diagram, although the semantics may differ.

Active models are produced as a result of three operations. The *projection* operation produces an initial model by selecting the important crosscutting program structure influenced by the aspect of interest. The *abstraction* operation is then applied to reduce the complexity of large concerns. Finally, developers may repeatedly invoke the *expansion* operation to insert additional detail or context as required for comprehension of the concern.

3.1 Projection

ActiveAspect uses declarations in aspect source files to select the important parts of the crosscutting structure to form the initial content of an active model. The aspect itself is an important part of the concern, as are the scattered program entities it directly influences. The particular aspect members considered during this initial projection are those that are directly involved in applying changes to the base program. Thus, the structure associated with the effects of advice, inter-type declarations, and `declare parents` state-

[2]The Mylar tool [14] for Eclipse that is currently under development has the potential to mitigate this problem by only showing items with a high degree-of-interest function value.

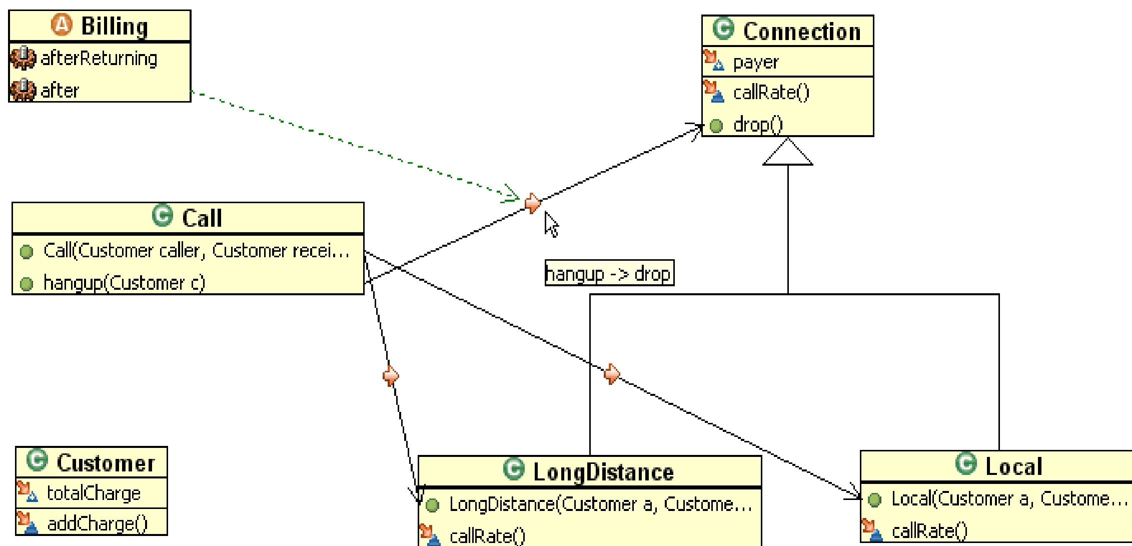

Figure 1: A diagram of an active model showing the crosscutting structure of a concern as projected from a Billing aspect in the telecommunications example program distributed with AspectJ.

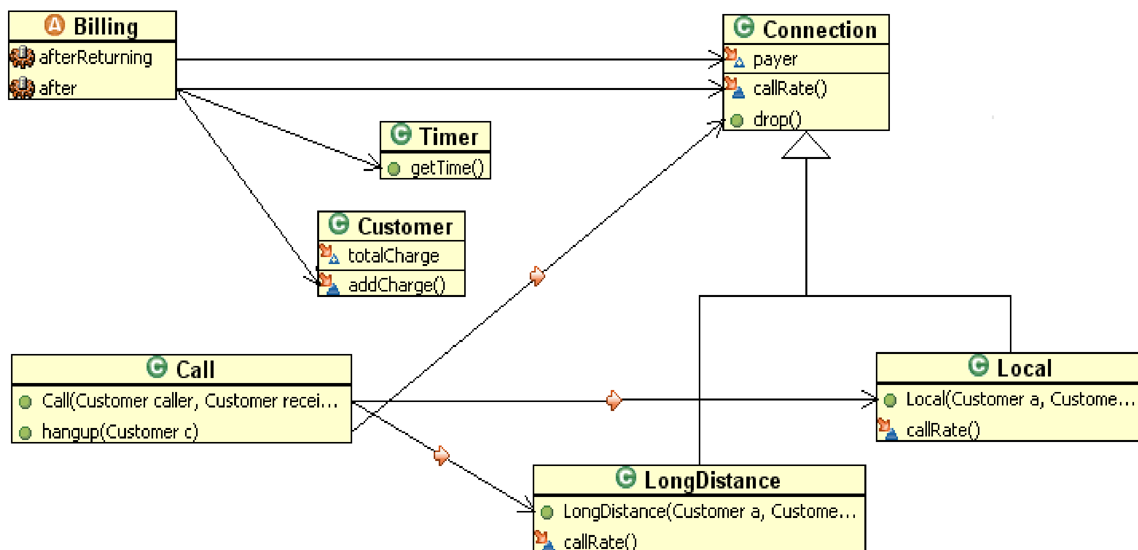

Figure 2: This diagram shows the model presented in Figure 1 after expansion. The diagram now shows several additional method calls from the Billing aspect. The Timer class has also been added.

ments are included in the initial model.

The active model resulting from this concern projection stage will typically hide some crosscutting relationships to reduce graphical clutter. The presence of such relationships are indicated using icons. Hovering the mouse over these icons causes the hidden relationships to appear. Figure 1 shows an example of an active model as the user hovers over an icon that indicates the presence of crosscutting.

3.2 Expansion

The initial active model provides an overview of a concern structure. Often, though, additional context is needed. For example, a developer may need to know the callers of an advised or introduced method. Calls made from an advice body may also be useful for understanding a concern.

The developer may not always know what additional program elements and relationships are most important in the current context. ActiveAspect automatically expands the concern structure to show additional related structure when requested by the developer, saving the developer from issuing many individual queries. The structure is expanded incrementally by invoking the expand operation until the desired level of detail is reached. The elements added to the model during an expansion are those that have the highest potential importance to the developer. The potential importance is computed as a function of the element or relationship type and other characteristics such as the parent type or number of associated relationships. Figure 2 displays the concern from Figure 1 after it has been expanded to show additional calls and references.

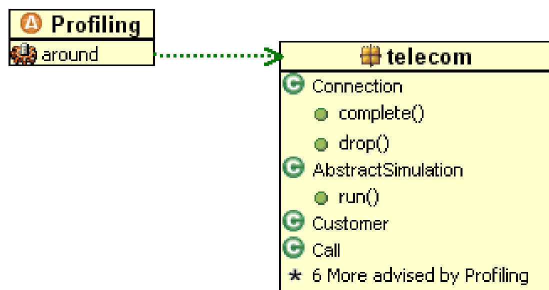

Figure 3: A diagram of an active model of a profiling aspect where many program elements have been abstracted.

3.3 Abstraction and Examples

Concerns may involve many program elements and relationships. For example, consider a performance profiling concern that counts the number of times each method in a program is called. A complete diagram of such a concern would include all methods in the system. For any nontrivial program, the resulting diagram would be excessively cluttered with classes and methods.

However, the inclusion of all of these methods and classifiers in a structure presentation model is not required for understanding the concern. Therefore, we apply automatic abstraction techniques to aggregate the classifiers, members, and relationships to control the complexity of the diagram. To ensure the aggregation is comprehensible, we retain a small number of concrete examples in the produced diagram. These concrete examples are displayed along with the abstracted entities that represent all other similar instances of the concrete element.

Figure 3 shows a diagram of an active model of a profiling concern that involves many methods and classes. The abstracted view shows several classes in a package that are part of the concern. Three concrete examples of methods affected by the concern are shown. The remaining classes and methods involved have been abstracted.

4. CONCLUSION

Existing approaches for presenting crosscutting concerns either do not adequately display cyclic structures with multiple relationship types, or suffer from excessive graphical complexity. To overcome these problems, we are investigating the *active model* approach which provides the directness of the structure diagram approach while avoiding excessive graphical complexity through automated abstraction and focused user interaction. ActiveAspect is a preliminary implementation of our approach that presents the crosscutting structure described by AspectJ aspects. We are currently tuning ActiveAspect's heuristics and user interaction so that it can be used to evaluate the effectiveness of active models. Our goal is to enable developers to be able to cost-effectively understand and work with crosscutting program structure.

5. REFERENCES

[1] AspectJ. http://www.eclipse.org/aspectj/, February 2005.

[2] Borland Together. http://www.borland.com/together/, February 2005.

[3] Eclipse. http://www.eclipse.org/, February 2005.

[4] Eclipse AspectJ Development Tools Project. http://www.eclipse.org/ajdt/, February 2005.

[5] FEAT: An Eclipse Plugin for Locating, Describing, and Analyzing Concerns in Source Code. http://www.cs.ubc.ca/labs/spl/projects/feat/, February 2005.

[6] Omondo. http://www.omondo.com/, February 2005.

[7] Rational Software Architect. http://www-306.ibm.com/software/awdtools/architect/swarchitect/, February 2005.

[8] The Concern Manipulation Environment Project. http://www.eclipse.org/cme/, February 2005.

[9] The SHriMP Project. http://www.shrimpviews.com/, February 2005.

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

[11] S. G. Eick, J. L. Steffen, and J. Eric E. Sumner. Seesoft-a tool for visualizing line oriented software statistics. *IEEE Trans. Softw. Eng.*, 18(11):957–968, 1992.

[12] 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.

[13] D. Janzen and K. D. Volder. Navigating and querying code without getting lost. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 178–187. ACM Press, 2003.

[14] M. Kersten and G. C. Murphy. Mylar: a degree-of-interest model for IDEs. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development (to appear)*.

[15] 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.

[16] M. P. Robillard and G. C. Murphy. Concern graphs: finding and describing concerns using structural program dependencies. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 406–416. ACM Press, 2002.

[17] 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)*.

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