Presenting Crosscutting Structure with Active Models

by

Wesley Coelho
B.Comm., The University of Victoria, 2000

A THESIS SUBMITTED IN PARTIAL FULFILMENT OF
THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
in
The Faculty of Graduate Studies
(Computer Science)

THE UNIVERSITY OF BRITISH COLUMBIA
August 2, 2005
© Wesley Coelho, 2005
Abstract

When performing development tasks, such as modifications and debugging, developers must often understand and manipulate source code that crosscuts a software system's structure. These tasks are made more difficult by limitations of the two approaches currently used to present crosscutting structure: tree views and structure diagrams. Tree views force the developer to manually synthesize information from multiple views; structure diagrams quickly suffer from graphical complexity. We introduce an active model as a means of presenting the right information about the crosscutting structure to a developer at the right time. An active model is produced as a result of three operations—projection, expansion, and abstraction. Combined with particular user interaction features during display, these operations enable a view of the model to be presented to the developer without suffering from the complexity of existing approaches. We have implemented an active model tool, called ActiveAspect, for presenting crosscutting structure described by AspectJ aspects. We report on the results of a case study in which the tool was used effectively by two subjects to implement a modification task to a non-trivial AspectJ system.
Contents

Abstract ................................................. ii

Contents ................................................ iii

List of Tables ........................................ vi

List of Figures ......................................... vii

Acknowledgements .................................... viii

1 Introduction ........................................ 1
   1.1 Motivation ........................................ 1
   1.2 The Active Model Approach .................... 1
   1.3 Thesis Statement ............................... 4
   1.4 Overview ......................................... 4

2 Existing Approaches ............................... 5
   2.1 High-level Views ................................ 5
   2.2 Tree-based Views ............................... 5
   2.3 Static Structure Diagrams ..................... 9
      2.3.1 Round-trip Engineering Tools ............ 9
      2.3.2 Aspect-oriented Modeling .................. 10

3 Active Models ..................................... 11
   3.1 Active Model Structure ....................... 13
   3.2 Graphical Notation and User Interaction ...... 13
   3.3 Projection ...................................... 16
   3.4 Expansion ...................................... 19
   3.5 Abstraction .................................... 19
## Contents

4 ActiveAspect ....................................... 22  
  4.1 Presenting AspectJ Constructs ............... 22  
    4.1.1 Aspects ................................ 22  
    4.1.2 Advice .................................. 23  
    4.1.3 Inter-type Declarations ................ 24  
    4.1.4 Declare Parents Statements ............ 24  
  4.2 Active Model Heuristics ...................... 25  
    4.2.1 Projection Heuristics .................. 25  
    4.2.2 Expansion Heuristics .................. 25  
    4.2.3 Abstraction Heuristics ................ 26  
  4.3 Supplementary Views ........................... 28  
    4.3.1 Declaration View ....................... 29  
    4.3.2 Detail View .............................. 29  
  4.4 ActiveAspect Implementation ................. 30  
    4.4.1 The AspectJ Abstract Structure Model . 30  
    4.4.2 Model Operators ......................... 32  
    4.4.3 User Interface ............................ 33  
    4.4.4 Performance .............................. 35  

5 Evaluation ......................................... 37  
  5.1 Case Study ..................................... 37  
    5.1.1 Setup .................................... 37  
    5.1.2 Results .................................. 38  
    5.1.3 Limitations ............................... 44  
  5.2 AJDT Comparison ................................. 44  

6 Discussion ......................................... 46  
  6.1 Generalizeability ............................... 46  
  6.2 Local Expansion ................................. 49  
  6.3 Scalability ..................................... 49  
  6.4 Diagram Layout .................................. 50  

7 Conclusion .......................................... 51  
  7.1 Research Contributions ....................... 52  
  7.2 Future Work .................................... 52  
    7.2.1 Interaction Technique Validation ...... 52  
    7.2.2 Degree-of-Interest Guided Heuristics ... 53  
    7.2.3 Forward Engineering and Refactoring ... 54
List of Tables

4.1 References added during expansion ................. 26
4.2 Element and relationship example values ............ 27

5.1 Structure event log entries .......................... 39
5.2 Summary of recorded structure information events. .... 40
5.3 Eclipse and AJDT views needed by participants ........ 44
List of Figures

2.1 High-level structure visualization .............................................. 6
2.2 AJDT and Eclipse tree views ...................................................... 7

3.1 Active model work flow .......................................................... 11
3.2 Active model data flow ........................................................... 12
3.3 Active model data structure ...................................................... 14
3.4 Active model notation ............................................................ 15
3.5 Active model abstraction notation ............................................. 17
3.6 Projection of the Billing aspect .................................................. 17
3.7 An active model diagram after expansion ..................................... 20
3.8 An active model diagram showing abstraction ............................... 21

4.1 Screenshot of the ActiveAspect plug-in for Eclipse ....................... 23
4.2 AspectJ-specific notation ......................................................... 24
4.3 Declaration view ................................................................. 29
4.4 Detail view .............................................................................. 30
4.5 ActiveAspect architecture ......................................................... 31
4.6 Model-view-controller relationships .......................................... 34

5.1 Screenshot of the display for the case study aspect ....................... 43

6.1 Active concern graph model ...................................................... 47
6.2 Expanded active concern graph model ....................................... 47
Acknowledgements

I would first like to thank my supervisor, Gail Murphy. Early on, Gail provided me with invaluable guidance in clarifying and developing my best research interests, while steering me clear of dead-end research paths. When problems arose, her sound judgment kept the research moving along smoothly. Gail always responded to my questions and provided feedback on my work with surprising timeliness. Her advice and suggestions dramatically improved the quality of my papers and presentations. As the research progressed, Gail generously supported my travel to present the work. Furthermore, her leadership style helped make the whole experience enjoyable. I was fortunate to have Gail as a supervisor and I am very grateful for her dedication.

I would like to thank Mik Kersten for his assistance in modifying the AspectJ compiler to make the implementation of this research possible. Mik was also able to find time to share his life-saving expertise with Eclipse on several occasions.

Several members of the Software Practices Lab provided feedback on my work, answered my questions, and pointed me to relevant references. In particular, I would like to thank Brian de Alwis, Jonathan Sillito, Chris Dutchyn, Trevor Young, and Ed McCormick. It was great to have them around. I would also like to give special thanks to Eric Wohlstadter for reviewing this thesis as Second Reader.

I am grateful for partial funding from NSERC (Natural Sciences and Engineering Research Council of Canada) and a University of British Columbia Graduate Fellowship that supported this work.

I would like to thank my family for their generous support and for always making my education a priority.

Finally, I would like to dedicate this thesis to Alison. Thanks for waiting for me.
Chapter 1

Introduction

1.1 Motivation

Many tasks performed by a software developer involve code that crosscuts the software system’s structure. For example, modification tasks often involve changing code spread across multiple classes and methods [18]. Most debugging tasks involve understanding execution slices that cut across system modules [23]. Various approaches have been proposed to help identify the crosscutting code of interest for a task (e.g., [20, 23, 24]). These approaches are similar in that they identify program elements and relations of interest for a task; they differ in how the elements of interest are determined from largely manual, static analyses (e.g., [20]) to largely automatic dynamic analyses (e.g., [24]). We refer to the program elements and relations identified by these approaches as crosscutting structure.

In contrast to the variety of ways of identifying crosscutting structure, only two approaches have been proposed to present the details of crosscutting structure: tree views, and structure diagramming approaches (Section 2). Each of these approaches has significant limitations. It typically takes multiple tree views to present crosscutting structure, placing the burden on the developer to manually synthesize the information of interest from multiple places. Existing structure diagram approaches suffer from graphical complexity when applied to even a moderate amount of crosscutting structure, making it difficult for the developer to find information of interest.

1.2 The Active Model Approach

To overcome these limitations, we have developed the active model approach that automatically determines important crosscutting structure information from a description of crosscutting structure and displays that information as a structure diagram. Our approach uses several interactive features to elide
structure from a displayed view until a developer needs the information. The combination of these features overcomes the limitations of previous approaches by providing the information of interest in one view while avoiding unnecessary graphical complexity.

An active model is a data structure that represents crosscutting program structures for the purpose of displaying them in a view. Three operations are used to produce and iteratively manipulate an active model: projection, expansion, and abstraction. When invoked by a developer, the projection operation selects a representative overview of the crosscutting program structure of interest. The expansion operation inserts additional structure incrementally when it is requested by a developer to aid investigation. When the desired structure is too large to be displayed cleanly, an abstraction operation is automatically applied to reduce complexity while retaining information content.

Active models are displayed as a diagram in a view that is intended to be used interactively. Developers interact with the view to reveal additional relationships, source code, and other information related to particular parts of the diagram of interest. The presence of additional information is often indicated by icons in the diagram.

To investigate our approach, we developed a tool called ActiveAspect that uses the active model approach to present crosscutting structure as described by an AspectJ [17] aspect. We chose to initially evaluate the active model approach in the context of a system written in AspectJ because the crosscutting structure is clearly stated, allowing us to focus on our approach and not on the identification of the crosscutting structure.

An active model will only be useful to a developer who needs to understand and manipulate crosscutting structure if the model presents the right information to a developer at the right time. To provide an initial assessment of whether our approach meets this goal, we undertook an exploratory case study in which two subjects used the ActiveAspect tool for a modification task on a non-trivial system. Using ActiveAspect, the two subjects were able to find the crosscutting structural information needed to make progress on the assigned task. We report on the results of the study both quantitatively and qualitatively. We also report on a comparison of accessing the

---

1 No tool can always provide exactly the right information at the right time but through a case study we found that our ActiveAspect tool, which provides active models for AspectJ aspects, can provide mostly the right information at mostly the right time. We use the stronger form of the sentiment to make our intent clear.
structural information needed by the subjects using the state-of-the-art tree views provided by the AspectJ Development Tools (AJDT) [2]. In short, the active model approach performed as expected, presenting the information directly rather than requiring the subjects to synthesize information from multiple views.
1.3 Thesis Statement

This thesis evaluates the following hypothesis:

Important parts of crosscutting program structures can be presented directly in a diagram view without excessive complexity using a combination of user interaction and abstraction techniques driven by a set of heuristics.

1.4 Overview

We begin by presenting an overview of crosscutting structure presentation approaches and their limitations in Chapter 2. Chapter 3 describes the interactive and automated techniques of our active model approach to presenting crosscutting structure. An implementation of this approach for AspectJ called ActiveAspect is presented in Chapter 4. We report on the results of a case study where developers used ActiveAspect to implement a change task in Chapter 5. Chapter 6 provides a discussion of several issues raised by the approach. We review our contributions and discuss several future research directions in Chapter 7.
Chapter 2

Existing Structure Presentation Approaches

Several approaches have been developed for presenting crosscutting program structure at various levels of abstraction. The main categories of existing approaches include high-level views, tree-based views, and static structure diagrams.

2.1 High-level Views

SeeSoft-like [9] views are commonly used to present high-level overviews of crosscutting structure [2, 7, 11]. These views display a row of rectangles that represent files containing source code involved in the crosscutting structure of interest. Within each rectangle, color-coded horizontal lines are used to represent lines of source code that are part of the structure (Figure 2.1).

These views allow developers to quickly determine the amount of code of interest as well as the extent to which the code is scattered among the modules of a system. However, since none of the program's declarations are visible as text, the information provided is insufficient for directly supporting development tasks such as program modifications and debugging.

2.2 Tree-based Views

Several tools use tree views to present crosscutting structure (e.g., [2, 3, 7, 15]). These views display program elements according to a hierarchical relationship that is rooted at an element of interest. For example, the tree view labeled "Hierarchy" in Figure 2.2 shows the inheritance relationships among several classes. Typically, nodes in the tree remain collapsed until the details about the children of the node are required, at which time the child nodes may be expanded.
Chapter 2. Existing Approaches

Figure 2.1: A high-level visualization of crosscutting structure provided by AJDT. The rectangles represent classes and horizontal lines represent lines of source code that are part of the crosscutting structure.

One advantage of tree views is their simplicity. The Hierarchy view in Figure 2.2 clearly shows which classes extend the Connection class. Another advantage of this approach is that the view contains mostly text and can concisely display a relatively large number of elements. This is aided by the use of scroll bars and by allowing developers to collapse parts of the tree structure that are not relevant.

A significant disadvantage of using tree views is that tools (e.g., \[2, 3, 7\]) typically present only a single relationship type in a view, such as containment or inheritance. Since crosscutting structures involve multiple relationship types, multiple tree views are required. When crosscutting structure information is spread across multiple views, the burden is on the developer to mentally integrate the information from multiple views to construct a mental picture of the crosscutting structure.

As an example of this limitation, we describe the use of the state-of-the-art AspectJ Development Tools (AJDT) \[2\] plug-in for Eclipse to investigate the crosscutting structure associated with a Billing aspect in a sample telecommunications program distributed with AspectJ. The source code for the Billing aspect is included in Appendix A. The Outline view in Figure 2.2 shows declarations in the aspect that describe crosscutting structure, such as the highlighted afterReturning(Customer) declaration. To view the program structure affected by this declaration, a developer refers to the
Figure 2.2: AJDT and Eclipse tree views displaying part of the crosscutting structure associated with the Billing aspect in a telecommunications sample program distributed with AspectJ.
Chapter 2. Existing Approaches

Cross References view. This view identifies two constructor calls whose call sequence can be investigated in the Call Hierarchy view. Finally, a developer may discover inheritance relationships between the identified constructor calls using the Hierarchy view. The developer is required to open the appropriate tree view on the right elements in order to produce these trees and then manually synthesize the information they present.

Another disadvantage of the multiple tree view approach is that the views are frequently re-used to show the results of a new query. This causes the previously visible structure to disappear. Therefore, developers are required to remember key parts of the program structure as they discover it. This requirement to memorize structure further complicates the task of synthesizing information from several views.

jQuery [15] is a source code browsing tool that attempts to overcome these problems by presenting program structure with multiple relationships in a single tree view. The structure displayed in the view is determined by a set of queries issued by the developer using a logic language or built-in queries. Although JQuery presents the structure of interest in a single view, the synthesis problem still exists. When multiple relationship types are defined for a particular program element, that element will appear several times in the tree view. The developer must then find the multiple nodes that represent a single program element and mentally integrate them in order to discover the complete set of relationships of different types that are defined for that element.

The heart of the problem with tree views is that the structure of a tree-based presentations does not match the structure of the crosscutting information being presented. A tree is well-suited to presenting a single program relationship that is strictly hierarchical, such as inheritance. However, crosscutting structures consist of a directed graph of program elements and relationships that may include cycles. This mismatch between program structure and presentation structure requires the developer to mentally map between the two. This mismatch also makes it difficult to detect structural patterns involving multiple relationship types, such as the constructors of all subclasses of a particular class being advised. Thus, the developer using tree views is burdened with a complex translation and integration task that makes it difficult to quickly grasp important parts of a crosscutting structure.
Chapter 2. Existing Approaches

2.3 Static Structure Diagrams

The alternative to presenting crosscutting crosscutting structures as a tree view is to present the structure directly as a node-and-link diagram. In these diagrams, nodes represent program elements such as methods and classes and links represent relationships such as inheritance or method invocations. Although there are various notations, these diagrams are typically similar to UML class diagrams [19]. Existing approaches in this category are static in two ways: The information presented is extracted from source code without running the program, and the display presents a fixed subset of the program structure.

This approach improves upon tree views by directly presenting structures as they exist in the program. This avoids the need for the developer to mentally map from the presentation structure to the actual program structure. Structure diagrams naturally accommodate multiple relationship types and make it easier to identify structural patterns.

Approaches in this category include commercial round-trip engineering tools as well as two aspect-oriented modeling approaches.

2.3.1 Round-trip Engineering Tools

Several existing round-trip engineering tools produce object-oriented structure diagrams from code [1, 5, 6]. These tools typically allow developers to make changes to the diagram and then automatically update the source code to reflect those changes.

Although these tools are typically used to present a subset of a program’s structure, they are not well suited to presenting a crosscutting subset of a program structure. Consider, for example, displaying just the subset of a program structure that performs event logging. This crosscutting structure could involve nearly all classes in a system. The presentation of this structure by an existing round-trip engineering tool would contain each of the affected classes with some number of links between them representing, perhaps calls or associations between the classes. The resulting diagram would be excessively complex for any non-trivial application.
2.3.2 Aspect-oriented Modeling

Several aspect-oriented modeling approaches have been developed specifically for diagramming crosscutting structures (e.g., [8, 12, 21, 22]). These notations can be classified as belonging to one of two strategies that share similar advantages and limitations.

In the first strategy, a new classifier\(^1\) is introduced to represent aspects similar to how a class is modeled in UML (e.g., [21, 22]). A classifier for an aspect is then associated with the program elements it affects by connecting lines between the involved classifiers. This strategy displays crosscutting structure directly, but suffers from graphical complexity as it adds more connections to already complicated diagrams of object-oriented structure.

In the second strategy, crosscutting structure is described separately from the base structure (e.g., [8, 12]) and a binding mechanism is used to associate the two. The separation of the two structures can reduce graphical complexity. However, the separation makes it more difficult to understand the combined structure. A developer must parse the bind statements and combine the two structures mentally. Any automated means of presenting the results of these parse and bind operations would again result in an overly complex node-and-link diagram.

The limitations of these approaches demonstrate the tradeoff between directly showing crosscutting structure and avoiding graphical complexity. We believe developers would benefit from a presentation approach that directly shows crosscutting structure with minimal load on the developer while at the same time limiting graphical complexity.

---

\(^1\)The term *classifier* refers to encapsulation constructs such as classes, interfaces, and aspects.
Chapter 3

Active Models

In our active model approach we want to retain the benefits of node-and-link diagrams while mitigating their main weakness of graphical complexity. We achieve these benefits through a combination of user interaction techniques and operations that display a representative subset of a structure of interest to a developer. In this chapter we describe the general active model approach using specific examples from a concrete implementation of the approach called ActiveAspect. We provide details of ActiveAspect in Chapter 4.

The active model approach requires a description that identifies the crosscutting structure of interest. A structure description must contain the program elements of interest, such as methods and classes, as well as the relationships between them, such as method calls and inheritance. Examples of structure descriptions include concern graphs captured by the FEAT tool [3] and the crosscutting structures encapsulated by aspects in aspect-oriented languages.

A developer accesses an active model by requesting a projection of a given crosscutting structure description. The projection operation selects the important structure to display for the active model. As part of this process, the abstraction operation may reduce the complexity of the produced structure diagram; the abstract elements and relationships introduced by

Figure 3.1: Active model work flow. The developer invokes the projection operation to produce an active model that is automatically abstracted before being displayed. The projected model is used as input when the developer invokes the expansion operation. Expanded models are automatically abstracted prior to display.
Chapter 3. Active Models

this operation are kept in the active model. When the developer requires more detail about the crosscutting structure, the expansion operation is used to add detail to the model and diagram. As with the initial projection, the abstraction operation may again be used to help control the complexity of information in the diagram and model. This work flow is illustrated in Figure 3.1.

The content of an active model is displayed as a design-like diagram view in the developer's integrated development environment. The diagram view presents the active model structure in a manner similar to that of a UML class diagram, although the semantics of the notation differs (Section 3.2). Detailed structure information present in the model may not be visible in an initial view; the developer interacts with the view to investigate specific details as required. Figure 3.6 shows an example of a diagram view of an active model.

To fully explain the meaning of an active model and how one is produced, we need to clarify some terms. As shown in Figure 3.2, an active model presents crosscutting structure described by a crosscutting structure description that refers to the software system's source code. An active model may also use information obtained directly from the system's source code, or a model of the source code's structure. The active model consists of program elements and relationships referred to by the crosscutting structure description; it also may contain abstract elements and relationships that are not stated explicitly in the source code. A developer interacts with the active model through a structure diagram that displays part or all of the active model in a view.

![Figure 3.2: Active model data flow. The view displays the active model, which contains structure information from the given structure description as well as the neighboring source code.](image)
3.1 Active Model Structure

Active models are represented using the data structure shown in Figure 3.3. This structure consists of model elements and the relationships between them. Model elements can represent several kinds of program elements such as classes, methods, and fields. Model relationships represent program relationships such as inheritance and method invocation.

In this model, elements can have zero or more relationships associated with them. Model relationships must have a source element, but the target of a relationship may be either a ModelElement or another ModelRelationship. This allows the model to represent crosscutting structure that affects elements and relationships. For example, in aspect-oriented languages crosscutting behavior may be invoked when one method calls another. In this case, a program element representing the crosscutting behavior has a relationship to the affected method call relationship.

Elements and relationships in the model both extend from ModelEntity, which stores state required by all elements and relationships in the model. Each model entity stores a "value" that is assigned and used by the active model operators. For example, the expansion operator sets this value to represent the likelihood that a developer will find a particular new relationship to be of interest. Another important state common to all model entities is the visibility state. Elements and relationships may be present in the model for analysis but only those set as visible will be presented to the developer in a view.

A key feature of this model is that both elements and relationships may aggregate a number of other elements or relationships. In this case the aggregating entity exists in the model but not in the structure of the source code it represents. These aggregate entities are used to abstract several concrete elements or relationships in the program structure to be presented.

3.2 Graphical Notation and User Interaction

The structure diagram notation used to display active models is similar to a UML class diagram. However, there are several important differences. Furthermore, while UML diagrams are static documents, our approach is intended to support development tasks and limit graphical complexity.

UML class diagrams only support relationships between classifiers. Active
Figure 3.3: UML class diagram of the active model data structure.
models support these classifier relationships as well as relationships between individual members of a classifier. For example, a method call in a particular method that invokes a specific method of another class can be specified. Furthermore, one endpoint of a relationship may be a member while the other is a classifier. This more expressive notation allows lower level structural details to be presented to support development rather than design tasks.

Another important difference between our notation and that of a UML class diagram is that our notation maps directly to concrete programming language constructs rather than abstract design-level entities. For example, relationships drawn between members map directly to concrete relationships at the programming level, such as method invocations and field references. In a UML class diagram it is only possible to specify a general association between two classes. The use of such concrete, language-specific semantics further improves the utility of the active model notation for assisting with real development tasks.

One way in which an active model limits graphical complexity is by hiding some relationships until the user indicates that they are interested in viewing them. For example, by hovering over a program element in the diagram the user indicates that it is of interest. This causes previously omitted relationships connecting to the program element to temporarily displayed. Icons are used to indicate where additional relationships are present. These icons can be associated with both program elements and relationships, as shown in Figure 3.4.
A user action, such as hovering, can also trigger the display of more detailed information about a particular element. For example, the ActiveAspect tool provides supplementary views that display source code and other details when members or relationships are hovered (Section 4.3).

Active models also support notation for expressing abstractions. The abstractions aggregate similar groups of classifiers, members, and relationships. Several classifiers with common characteristics can be displayed using a single aggregate classifier. The members of an aggregate classifier represent the individual classifiers being aggregated. When the classifiers have a common parent package, the parent package name is used as the title. The classifier labeled 'd' in Figure 3.5 is an example of an aggregate classifier.

Members of a classifier can also be abstracted. This applies both to aggregate package nodes as well as regular classes. In this case, an aggregate member is used to represent a number of concrete members. The aggregate member is labeled to indicate the common properties of the elements it represents (e.g. Figure 3.5 a, b, and c).

Finally, aggregate relationships may also be used in active model diagrams to represent a number of concrete relationships. These aggregate relationships are typically used to represent multiple relationships with the same source or target element. Aggregate relationships are visually distinguished from regular relationships by using thicker lines. When an aggregate relationship connects to the title area of a classifier, this indicates that it applies to multiple members of the classifier (e.g. Figure 3.5 g). Many-to-many aggregate relationships are drawn between the title areas of the source and target classifiers.

3.3 Projection

The initial content of an active model is produced by the projection operation, which selects a subset of the given structure description to be included in the model. The operation attempts to select parts of the structure that are likely to be of interest to the developer. The potentially interesting structure is determined by a set of heuristics that identify particular elements based on their characteristics and structural properties.
Figure 3.5: Active model abstraction notation. a. An aggregate member representing 4 methods invoked by ClassInPackage.method2(). b. Represents 3 methods that implement the interface IInterface. c. Represents 6 'getter' or 'setter' methods. d. A package classifier whose members are classes belonging to the package package. e. A method defined in ClassInPackage. f. Represents 4 additional classes in the package. g. An aggregate call relationship representing several calls from method() to targets in package. h. An aggregate call relationship representing several calls from method2() to methods in the Class class.

Figure 3.6: A diagram of an active model of the Billing aspect listed in Appendix A. The developer is hovering over an icon indicating that the call from hangup() to drop() has additional structure associated with it. This causes the dashed arrow to appear, which identifies an advice affecting the call.
The following list provides examples of element and relationship characteristics that may be used by projection heuristics:

- Element Kind: Classifier or Member
- Member Kind: field, method, constructor
- Member Visibility: public, protected, private
- Classifier Kind: Class, Interface, Aspect
- Relationship Kind: Inheritance, method call, field reference

The following examples of structural properties may also be considered:

- Number of source or target call relationships defined on a method
- Number of subclasses extending a particular class
- Number of overriding methods declared for a particular method
- Number of members defined in a classifier
- Whether a method references only local members or if it makes external references

These characteristics and structural properties allow heuristics such as:

- Select public methods that are called by methods in several different classifiers
- Select methods that are frequently overridden, as well as the overriding methods
- Select classifiers where many of its members are in the structure description

The heuristics also determine which relationships are to be permanently displayed or only shown when the user interacts with a specific part of the diagram. Relationships may be selected to be displayed only upon user interaction if they are identified by a heuristic that matches structure that is likely to be of relatively lower interest to a developer.

Figure 3.6 shows a snapshot of the active model for the Billing aspect introduced in Section 2.2 after the projection operation. The user's mouse is hovering over an icon that indicates the presence of additional structure. This causes a relationship (dashed in the diagram) to be drawn.
Chapter 3. Active Models

3.4 Expansion

The expansion operation is invoked by the developer to add additional structure to an active model. There are two sources of additional structure. The first is program elements and relationships present in the structure description but not yet included in the model. Adding this kind of structure serves to add more detail to the model. The second source of structure is program elements and relationships in the program source code that neighbor the structure in the model. Adding this kind of structure serves to add context to the model so developers can see how it interacts with other parts of the program.

The expansion operation works by finding all neighboring structural elements and adding them to the active model. Initially, the visibility state of this new structure is set to false (in the ModelEntity state described in Section 3.1). The operation also sets the value variable of the elements and relationships in the new structure to represent the likelihood that they will be of interest to the developer. The potential interest to a developer is determined by structural properties such as those described in Section 3.3. A small number of elements and relationships with the highest values are then selected to become visible to the developer during each expansion operation.

The intent of this operation is to allow the developer to focus on understanding the crosscutting structure, rather than on forming queries to access the information, as is the case in other tools.

Figure 3.7 shows the result of applying the expansion operation to the Billing aspect. The operation has added the Timer class as well as several method calls made from the Billing aspect. These additional relationships are intended to help a developer understand how the aspect works.

3.5 Abstraction

Crosscutting structures are frequently too large to be drawn cleanly in one diagram. The abstraction operation automatically reduces the complexity of the model after projection and expansion operations by aggregating classifiers, members, and relationships. To ensure the diagram retains detailed information, a small number of concrete examples of the aggregated information remain visible in the diagram.

Classifiers are aggregated when there are several that share a common
property, such as inheriting from a common superclass, or containing a reference to the same member of another class. A set of heuristics is used to identify such structural patterns where aggregation may be beneficial. The aggregated classifiers appear in a package node labeled with the name of the most specific package that contains each of the aggregated classifiers, as shown in figure 3.5. To serve as an example and display additional detail, a small number of the classifiers are selected to remain in the model outside of the aggregate node. Classifiers with high-valued structural properties are selected as examples.

When there are many members within a classifier or aggregate classifier, the abstraction operation represents members with common properties using aggregate members. For example, accessor methods, or methods that implement a particular interface may be aggregated. In the extreme case when there are too many members with no known common property, they may be still be aggregated under a node labeled "n more methods," for example. Classifiers typically contain several concrete members that serve as detailed examples, followed by aggregate members.

Aggregation can also be applied when there are many relationships with the same source or target element. In this case, several relationships are represented by a single aggregate relationship. This aggregate relationship connects to the single source or target member on one end, and the title of the classifier (or aggregate package node) that contains multiple start or end
points of the relationship.

Figure 3.8 shows a diagram from an active model of the profiling structure in the telecommunications example program. Although the profiling affects all members and classes in the system, the diagram shows only a few examples with the remaining structure aggregated. The dark relationship arrow is an aggregate arrow that represents a number of relationships connecting to elements in the telecom package.
Chapter 4

ActiveAspect

ActiveAspect is a tool we have developed that implements the active model approach to present crosscutting structure defined in an AspectJ aspect. ActiveAspect is implemented as an Eclipse plug-in that developers invoke to display and explore the structure associated with a particular aspect of interest while working on a development task.

In this chapter we describe how crosscutting structures are specified in AspectJ and the corresponding representation of these structures in active models and their graphical notation. We then describe the heuristics we used in instantiating the active model approach to support the presentation of crosscutting structures defined in AspectJ. Finally, we provide details of the ActiveAspect tool's implementation.

4.1 Presenting AspectJ Constructs

ActiveAspect presents crosscutting structure expressed as an aspect in AspectJ. AspectJ aspects are similar to classes but also support additional declarations that are used to describe crosscutting structure. The declarations that describe this structure are “advice,” inter-type declarations, and declare parents statements. In this section we describe how aspects and their crosscutting structure declarations are represented in active models and the corresponding diagram notation.

4.1.1 Aspects

In an active model, an aspect is represented as a classifier in the same way as a class or interface. In the diagram, aspects are differentiated from other classifiers by an 'A' icon. In addition to fields and methods, aspects can have pointcuts and advice as members. Advice are method-like constructs that implement crosscutting behavior. Pointcuts are used to describe the points in a program's execution where advice behavior may be executed. In the
Figure 4.1: Screenshot of the ActiveAspect plug-in for Eclipse

diagram, advice are denoted by a cog icon and pointcuts by a lightning bolt icon. These icons are consistent with those used in AJDT. Figure 4.2 shows some of the notation used to present aspects and other AspectJ constructs.

4.1.2 Advice

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 [13]. Advice advise these locations and therefore have advises relationships connecting them with parts of the structure where they might apply.

When a method execution is advised, this is represented in the active model by a member-to-member advises relationship between the advice and the advised method. Advice can also insert additional behavior at a particular method call (or field reference), in this case the advises relationship connects the advice with the relationship representing the advised call.
Advice often advise a number of parts of the program structure and are therefore displayed on mouse hover in the diagram to avoid unnecessary complexity. Icons are used to indicate parts of the structure that are advised. Hovering the icon temporarily draws the advises relationship, revealing the advice that applies at that location.

### 4.1.3 Inter-type Declarations

AspectJ inter-type declarations are statements in aspects that declare fields or methods to be part of another type. The declared members are modeled and displayed as part of the target classifier rather than in the declaring aspect because this allows the combined effect of inter-type declarations and advice on a particular classifier to be displayed clearly. Furthermore, it is natural to reason about an inter-type declaration as part of the target type because it runs in the scope of that type.

*Introduces* relationships are drawn from the declaring aspect classifier to the inter-type declaration. These relationships are shown on hover and icons indicate which members are inter-type members.

### 4.1.4 Declare Parents Statements

AspectJ *declare parents* statements cause one type to implement or extend another. The notation for these statements is simply an inheritance relationship between the subtype and supertype. However, this relationship
Chapter 4. ActiveAspect

is annotated with an icon to indicate that it is declared by an aspect. Hovering over the icon draws a relationship indicating the declaring aspect.

4.2 Active Model Operation Heuristics

To instantiate and manipulate an active model for a given type of structure description, it is necessary to define the heuristics to be used by the projection, expansion, and abstraction operations. In this section we describe the heuristics used by ActiveAspect to produce active models for the structure described in an AspectJ aspect. These heuristics were tuned by applying ActiveAspect to sample AspectJ programs.

4.2.1 Projection Heuristics

In ActiveAspect, the projection operation selects program structure that is directly influenced by the aspect of interest as well as the aspect itself. The crosscutting structure directly influenced by the aspect is determined by its advice, inter-type declarations, and declare parents statements. The effect of each such statement in the given aspect is added to the active model.

For each advice in the aspect describing the structure to be presented, the advice and the structure associated with the static shadow of the advice is added to the model. This structure can include methods whose executions are advised or advised method call relationships as well as their source and target methods. Inter-type members declared in the aspect are added to the model as members of their target types. For each declare parents statement in the aspect, the specified subtype and supertype are added as well as an inheritance or realization relationship between them. Figure 3.6 shows a diagram of the Billing aspect after these projection heuristics have been applied to produce the model.

4.2.2 Expansion Heuristics

The ActiveAspect expansion operation adds additional method calls and field references to the model. The targeted members and their parent classifiers are also added to the model as required. ActiveAspect currently adds the two highest-valued references during each expansion operation. Target elements of these references are also added if they were not previously in the model.
Chapter 4. ActiveAspect

<table>
<thead>
<tr>
<th>Order</th>
<th>Relationship</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Method call from advice body</td>
</tr>
<tr>
<td>2</td>
<td>Reference from inter-type method</td>
</tr>
<tr>
<td>3</td>
<td>Reference to inter-type method</td>
</tr>
<tr>
<td>4</td>
<td>Reference from advice body to field in another type</td>
</tr>
<tr>
<td>5</td>
<td>Reference to inter-type field</td>
</tr>
</tbody>
</table>

Table 4.1: These reference types are added to the active model during expansion in the listed order.

Reference values are assigned according to a simple ordering based on their characteristics. Our expansion model expands reference types in the order listed in Table 4.1.

For the programs we used to tune the heuristics, we consistently found that the most important references to be displayed are method calls made from advice bodies. Advice bodies implement the crosscutting behavior of an aspect and therefore the calls made by them are particularly useful for understanding the aspect’s effect.

After calls made by advice, we believe that calls made by inter-type methods are likely to be relevant for similar reasons. Inter-type methods implement crosscutting behavior and it is important for developers to see how they interact with the system. In general, we believe that field references are lower-level details but are more likely to be of interest if they are referenced from outside the class in which they are declared.

Figure 3.7 shows the structure identified in Figure 3.6 after the expansion operation has been applied.

4.2.3 Abstraction Heuristics

Abstraction of classifiers in ActiveAspect is guided by the advice in the aspect of interest. Substantial parts of an aspect’s crosscutting structure are often related through the advice that applies to them. ActiveAspect takes advantage of this relationship by selecting a small number of concrete examples from large advice shadows and aggregating the remaining structure.

Examples are selected based on a simple model of the element characteristics that suggest they may be of interest to the developer. In our model, the value of an element is determined by its type as well as the number and
Chapter 4. ActiveAspect

<table>
<thead>
<tr>
<th>Elements</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Advice</td>
<td>10</td>
</tr>
<tr>
<td>Classifier</td>
<td>5</td>
</tr>
<tr>
<td>Pointcut</td>
<td>5</td>
</tr>
<tr>
<td>Inter-type Method</td>
<td>5</td>
</tr>
<tr>
<td>Inter-type Field</td>
<td>5</td>
</tr>
<tr>
<td>Constructor</td>
<td>3</td>
</tr>
<tr>
<td>Method</td>
<td>2</td>
</tr>
<tr>
<td>Field</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Relationships</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Advises</td>
<td>5</td>
</tr>
<tr>
<td>Introduces</td>
<td>4</td>
</tr>
<tr>
<td>Uses Pointcut</td>
<td>4</td>
</tr>
<tr>
<td>Extends</td>
<td>3</td>
</tr>
<tr>
<td>Realizes</td>
<td>3</td>
</tr>
<tr>
<td>References</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 4.2: Element and relationship values used to select examples during the abstraction.

type of its relationships. For example, inter-type methods that are called by one or more advice bodies are assigned high values while a field that is only referenced once is unlikely to be selected as an example.

The values assigned to elements and relationships by ActiveAspect for example selection are listed in Table 4.2. Advice are considered central to understanding the presented aspect structure and are therefore assigned the highest value. Fields are often lower-level implementation details and are unlikely to be selected as examples. Relationship values were selected so that elements with several relationships of the types listed will be selected before elements of any type that have few relationships. Elements with many relationships are more likely to be an important part of a crosscutting structure.

When there are many members in a classifier, ActiveAspect aggregates members that have similar characteristics. For example, methods advised by the same advice or introduced by the same aspect may be grouped. Similarly,
methods implementing a particular interface can be represented by a single aggregate method. The following characteristics are used by ActiveAspect to aggregate members:

- accessor methods
- methods implementing a particular interface
- members introduced by a particular aspect
- members advised by a particular advice
- methods making calls that are advised by a particular advice
- members with low values based on structural properties

ActiveAspect also uses the relationship aggregation feature of the active model notation to aggregate advises relationships. When there are many members of a classifier advised by the same advice, they can be represented by a single aggregate advises relationship. In this case, the aggregate relationship connects from the advice to the name of the classifier containing many advised elements.

Figure 3.8 shows an active model of a profiling structure that involves many methods and classes. The abstracted model shows several classes in a package that are part of the structure. Three concrete examples of methods in this structure are shown. The remaining classes and methods involved have been abstracted. An aggregate relationship indicates that the advice advises many program elements in the telecom package.

4.3 Supplementary Views

A key principle of the active structure presentation approach is to provide a structural overview and allow the user to interact with it to investigate specific details of interest. A simple way of providing additional kinds of detailed information is to use Eclipse views that are visible alongside the main structure view.
4.3.1 Declaration View

When interacting with the ActiveAspect structure view, developers can gain an intuition of the system's behavior via structural properties such as the names of invoked methods. It is often necessary to confirm this intuition by reading source code. If the code is viewed in an editor that replaces the structure view, however, the structural context of the source code is no longer visible.

To reduce the possibility of losing structural context, ActiveAspect provides a Declaration View that shows the source code for program elements that are hovered with the mouse in the diagram. This allows the developer to quickly find and read scattered pieces of source code in the context of their crosscutting structure. A screenshot of the Declaration View is shown in Figure 4.3.

![Declaration View Screenshot](image)

Figure 4.3: Declaration view showing the source code of an advice declaration.

4.3.2 Detail View

Due to aggregation, not all of the program structure that is part of the presented structure is visible at all times. If the selected example is insufficient, the Detail View can be used to show individual elements or relationships corresponding to an aggregate entity. For example, hovering over an aggregate member labeled "5 more called by save()" will display the five aggregated methods in the detail view. Similarly, hovering over an aggregate relationship will list the source and target of each individual relationship it represents.
Figure 4.4 shows the result of hovering over an aggregate member representing several classes.

```java
public class TimingSimulation extends AbstractSimulation
public class BillingSimulation extends AbstractSimulation
public class BasicSimulation extends AbstractSimulation
public aspect Timing
public aspect Billing
public class Timer
```

Figure 4.4: Detail view showing several classes aggregated by an abstract package member.

### 4.4 ActiveAspect Implementation

ActiveAspect is written in AspectJ and implemented as a plug-in for the Eclipse development environment, consisting of approximately 7000 lines of code and 100 classes, interfaces, and aspects.

ActiveAspect also relies upon other external components including the AspectJ compiler, the Graphical Editor Framework (GEF) for Eclipse, and the Draw2D drawing framework. Figure 4.5 provides an overview of the main architectural components of the ActiveAspect implementation.

ActiveAspect obtains all program structure information from an extended version of a data structure produced by the AspectJ compiler called the Abstract Structure Model (ASM).

A set of operations implemented by the ActiveAspect plug-in create and manipulate active models using information from the ASM. The active models are then rendered on screen using the Eclipse Graphical Editor Framework and the Draw2D drawing framework.

#### 4.4.1 The AspectJ Abstract Structure Model

The AspectJ compiler produces an Abstract Structure Model (ASM) during each compile. This data structure represents the program elements and relationships in the compiled AspectJ program. For example, aspects and
Figure 4.5: An architectural overview of the ActiveAspect implementation. Boxes represent key architectural components and open-head arrows identify important dependencies between them. Closed-head arrows show which ActiveAspect components extend external components or frameworks.

their advice are in the model, as are the relationships between advice and the program elements they advise. It is this structure model that the AJDT uses to display crosscutting structure in tree views and gutter annotations.

ActiveAspect also uses the ASM as a source of all program structure information. This avoids the need to parse Java source code or use Eclipse search functionality to determine the structure. However, the information contained the ASM is not sufficient for producing active models. Therefore, we extended the AspectJ compiler code that constructs the ASM so that the following additional program elements and relationships are included in the model:

- Method invocation sites for calls that are not advised and relationships from these sites to the called methods
- Field reference sites that are not advised and relationships from these sites to the referenced fields
Chapter 4. ActiveAspect

- Relationships between a `declare parents` declaration and the super types it declares on another type
- Relationships between advised field-set statements and the fields that they set

When a developer creates an active model of an aspect, ActiveAspect finds the corresponding aspect element in the extended ASM. The children of this aspect element in the ASM correspond to the declarations in the aspect's source code. Relationships in the ASM can then be retrieved for these elements to determine which parts of the program structure they affect. The entire active model is produced by navigating the ASM in this way.

4.4.2 Model Operators

Given the name of an aspect whose structure is to be presented, the projection operator finds the element representing the aspect in the ASM and produces the initial active model by iterating over its declarations and adding their effect as described in Section 4.1.

The expansion operator traverses the portion of the active model that is visible to the user and adds any neighboring elements and relationships. Initially, these neighboring entities are not visible to the user but become part of the active model so that they can be assigned a relevance value based on their type and structural properties (As described in Section 4.2.2). The expansion operator then iterates over the newly added elements and selects the $n$ highest ranked to become visible to the developer.

The active model abstraction operator is implemented as a set of three sub-operators that manipulate the model to achieve a subtask of the overall abstraction. These sub-operators perform classifier abstraction, member abstraction, and relationship abstraction.

The classifier abstraction sub-operator runs first and it uses advises relationships to guide the aggregation of classifiers. The first step is to find advice shadows where there are a number of classes affected. For each such shadow, a small number of classes are selected as examples and thus flagged as un-abstractable so they will not be aggregated by subsequent operators. The remaining classes in the shadow are removed from the model and re-added as children of an aggregate package classifier. This aggregate classifier is labeled with the name of the most specific package that contains each of the collapsed classifiers.
The member abstraction sub-operator finds classifiers with more than \( n \) members and applies an ordered list of abstraction rules to replace concrete members with aggregate member elements until the number of elements is \( n \) or less. An example of a member abstraction rule is: “Group all methods that contribute to the implementation of a particular interface.” Member abstraction rules are applied to group members in the order listed in Section 4.2.3. The intent of this ordering is to begin by abstracting members that are likely to be of lower value to the developer. If no common properties are found for grouping but there are still too many members, then the last step is to collapse members that have low importance values.

Finally, the relationship abstraction sub-operator is applied. This sub-operator finds instances where all or most of the members in a classifier have relationships with the same source element. When this case is found, an aggregate relationship from the source to the classifier is added to the model and each underlying concrete relationship is marked as abstract. The current implementation of ActiveAspect performs relationship abstraction for advises relationships, though it is trivial to extend this to also include method invocations and introductions.

### 4.4.3 User Interface

ActiveAspect uses the Eclipse Graphical Editor Framework (GEF) [4] to produce the user interface that developers use to view and interact with the active model. GEF is a generic framework that supports the development of interactive interfaces for an arbitrary set of objects in a program. These objects typically model a particular domain and are called *domain objects* in the GEF documentation.

GEF supports a model-view-controller (MVC) architecture that allows the domain objects to be displayed and manipulated on screen yet retain a de-coupled implementation that does not include user interface logic. The *model* in the MVC architecture refers to the arbitrary domain objects. The *View* is a set of objects called *figures* that define the on-screen appearance of the domain objects. Each domain object that is to be displayed has a corresponding figure object. Figures are created by extending a drawing framework called Draw2D. The *Controller* component of the MVC architecture is a set of objects called *EditParts* that coordinate between corresponding model and figure objects. The EditPart is responsible for updating the figure when properties of the domain object change. The EditPart also accepts user
Figure 4.6: This diagram shows examples of classes from three parallel data structures in ActiveAspect that assume the role of model, view, or controller in the MVC pattern required by GEF. Arrows indicate dependencies and diamond-shaped connectors show that the classes on the top row aggregate instances of the classes below.

input from the figure, which may result in a change to the domain object.

The domain objects in ActiveAspect are the elements and relationships in the active model, whose corresponding classes are shown in Figure 3.3. Each type of element and relationship (e.g. class, method, inheritance) has a corresponding figure object that defines its on-screen appearance and accepts user input. A set of EditParts coordinate between corresponding sets of figures and domain objects. Figure 4.6 shows an example of the relationships between the domain objects, EditParts, and figures in the ActiveAspect implementation.

The Eclipse development environment uses a plug-in architecture that allows it to be extended to support new functionality. ActiveAspect uses this feature of Eclipse to contribute the GEF active model interface as well as the detail view and a menu option to invoke ActiveAspect when the user right-clicks an aspect. The declaration view is included in the Eclipse distribution and can be invoked to display source code when the user hovers over declarations in the model.
4.4.4 Performance

Since ActiveAspect is intended to be a lightweight tool for quickly investigating key structure information, it is important that the active models can be generated and manipulated without delay. ActiveAspect’s performance depends on the performance of the projection, expansion, and abstraction operations.

The performance of the ActiveAspect projection operation is determined by the number of declarations in the aspect to be presented and the cost of retrieving the crosscutting structure associated with these declarations from the ASM. Due to the implementation of the ASM, its hierarchy of elements must be searched to find each element of crosscutting structure affected by a declaration. Thus the performance of the projection operation is $d \times r \times s$ where $d$ is the number of declarations in the aspect to present, $r$ is the average number of crosscutting relationships per declaration, and $s$ is the number of program elements in the program and ASM.

The expansion operation traverses the active model in order to add neighboring structure. The neighboring structure for a particular element in an active model is available through the ASM. The cost of retrieving the neighboring structure is $e \times r \times s$, where $e$ is the number of nodes to be expanded, $r$ is the average number of neighboring elements, and $s$ is the number of program elements in the ASM, which must be searched to find each neighboring element. In addition to adding new structure, the expansion operation is also responsible for assigning a value to new entities based on structural properties (Section 4.2.2) and selecting entities to be included in the portion of the model that is visible to the user. Therefore the ActiveAspect implementation makes three passes to expand, assign values, and select elements to be made visible. Thus the cost of the expansion operation is $3n + e \times r \times s$, where $n$ is the number of elements in the active model.

The abstraction operation makes two passes over the active model to abstract members of classifiers, and relationships between members. The declarations of the aspect being presented are also traversed to search for large advice shadows that merit abstraction of the target classifiers. During the member abstraction pass, the abstraction operation will iterate over members in each classifier several times in search of common properties that may be abstracted (Section 4.2.3). However, there are typically a relatively small number of members in a classifier and member abstraction is only performed as needed. Therefore the most significant cost of the abstraction
operation is 2n.

An active model represents a subset of a concern description, which typically represents a small fraction of a program structure. Thus the number of elements in an active model is expected to be low enough that traversals over the model do not substantially impact performance. Therefore, the most significant cost is the time required to search the ASM for neighboring structure. Relationships in the current ASM implementation use string handles to identify the element they refer to. Thus the ASM must be searched for the actual element with a matching handle. If the ASM could be modified such that relationships directly referenced elements, neighboring structure could be found in constant time, which would result in a substantial performance improvement.

A potential performance limitation of the ActiveAspect implementation is that active models are not incrementally updated after an incremental compile. After compiling, the active model may no longer match the program structure and it is necessary to project a new model from the most recently produced ASM.

A more important limitation of the ActiveAspect implementation that affects performance relates to difficulties that were encountered in extending the ASM. The ASM was extended to include method call relationships between call sites and the target method. However, we were unable to determine the file and line number of the target method, which was required to find the corresponding element in the ASM. A workaround for this problem is to search the ASM for the required method based on its name, file name, and other properties. However, this requires a search over the ASM for each method call in a program. This is likely to result in performance problems when constructing an ASM for a large-scale system. Therefore, this bug may need to be addressed, perhaps with assistance from the AspectJ development team, in order for ActiveAspect to scale to support large-scale systems.

Despite these limitations, ActiveAspect does not exhibit significant performance delays when applied to the AspectJ source code for ActiveAspect itself. Using a 1.8GHz notebook computer with 768 MB of RAM, an aspect with 30 declarations can be projected, expanded, and abstracted with no noticeable delay. Full compilation using the standard ASM averaged 7.6 seconds over 10 trials, compared with 7.7 seconds when using the extended ASM required by ActiveAspect.
Chapter 5

Evaluation

Active models attempt to focus a developer's time and effort on understanding the crosscutting structure of interest, rather than on moving between views, disentangling complicated diagrams, and searching through source code. To be successful in meeting this goal, an active model must select the information of highest interest to the developer most of the time, and must make it straightforward to get to additional detailed information when needed. As an initial determination of whether active models have these features, we performed a case study of the use of ActiveAspect by others on a software modification task, and we compared the information found with ActiveAspect for this task to what can be found with AJDT, the current state-of-the-art tool for displaying crosscutting structure.

5.1 Case Study

The case study involved a change task to the AspectJ code for the ActiveAspect tool. ActiveAspect comprises more than 100 classes and approximately 7000 lines of code. The task was to enable a user to make a non-sticky relationship (shown only on hover) sticky by clicking on it. We replicated this case study with two different subjects. In each case, the subjects attempted to implement the task using ActiveAspect. We were interested in whether the heuristics in ActiveAspect were effective in selecting information of interest to present, and whether the desired information was presented when it was needed by the developer.

5.1.1 Setup

To ensure we did not bias the study to a case for which ActiveAspect was particularly well-suited, we asked two graduate students to suggest a range of possible changes to the tool. We limited these students to suggestions that involved one of eight features implemented, at least in part, by an aspect. The
students suggested 7 additions; we selected the *click-to-stick* task randomly from this list. This task involves gaining an understanding and modifying the *OnDemandRelationships* aspect, that contains 30 declarations. A solution to this task by an author of this paper required edits to three locations in the source code. The ActiveAspect model used by the author to complete the change task involved over 100 program elements and relationships. The chosen task thus involves a non-trivially sized crosscutting structure.

The two subjects who participated in the study are graduate computer science students who had not previously used ActiveAspect. To familiarize them with active model concepts, each subject was given an overview of the approach and of the use of the ActiveAspect tool. To ensure comprehension of key concepts, subjects were asked to use ActiveAspect to answer four questions about the crosscutting structure of the *Billing* aspect (Section 2.2). Subjects were able to answer these questions correctly with only a small amount of assistance or clarification from the investigator.

We gave each subject a starting point—the name of the aspect describing the structure associated with the change task—because we found in a pilot run that the task was too difficult for a developer without any prior knowledge of the system to implement in a short period of time. We also gave each subject a sketch of the overall user interface design to focus on the use of ActiveAspect, rather than on an investigation of the GEF user interface framework architecture on which the tool is based [4].

Subjects were then asked to perform the modification task using ActiveAspect as a source of crosscutting structure information they required. If the needed structure could not be found, subjects were to use other features of the development environment. While implementing the task, subjects were asked to describe aloud the nature of any crosscutting structure they discovered or wished to know. These instances were logged by the experimenter, who also recorded whether the information was found using ActiveAspect. Table 5.1 shows an excerpt from the log recorded during an experiment (We explain the second and third columns in section 5.1.2).

### 5.1.2 Results

We analyzed the data gathered from the study quantitatively to assess how much of the crosscutting structure needed by the subjects was presented in the active model. This assessment helps answer the question of whether the active model has the right information. We rely on a qualitative analysis to
assess whether the active model was presenting the information at the right time.

**Quantitative Analysis**

Both subjects were able to use ActiveAspect as the primary source of the crosscutting structure information they required as they implemented the change task. A total of 16 and 13 events were logged for each subject, respectively. For 10 (63%) and 11 (83%) of the events, respectively, the structure information of interest was found in the active model.

The logged information events can be classified as either *structure queries* or *noticed structure*. A *structure query* event occurs when the subject needs explicitly to know something about the structure, such as asking “What are the callers of showOnDemandReisO?” (Subject 1). A *noticed structure* event occurs when the subject finds structural information believed to be important but for which the subject was not explicitly looking, such as “I see that setHiddenO is introduced from the aspect” (Subject 2).

While implementing the modification, 12 and 7 structure queries were logged for each subject, respectively. Subject 1 was able to satisfy 50% of the 12 queries using the active model; Subject 2 was able to satisfy 86% of their queries using the active model. Some queries, such as those involving inter-type declarations and shadows of advice associated with the aspect of interest, were found easily. With some effort, through the use of the expansion operation, queries regarding method calls to and from parts of this structure were also satisfied. The active model did not provide answers to

---

### Table 5.1: Excerpt from a log of recorded structure information events.

<table>
<thead>
<tr>
<th>Structure</th>
<th>Event Type</th>
<th>In Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Callers of setHidden()</td>
<td>Query</td>
<td>Yes</td>
</tr>
<tr>
<td>Two advice call showOnDemand()</td>
<td>Noticed</td>
<td>Yes</td>
</tr>
<tr>
<td>Type hierarchy of AbstractGraphicalEditPart</td>
<td>Query</td>
<td>No</td>
</tr>
<tr>
<td>Call hierarchy for mouseEntered()</td>
<td>Query</td>
<td>No</td>
</tr>
</tbody>
</table>
Chapter 5. Evaluation

<table>
<thead>
<tr>
<th>Subject</th>
<th>S1</th>
<th>S2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total recorded structure events</td>
<td>16</td>
<td>13</td>
</tr>
<tr>
<td>Total number of events satisfied by ActiveAspect</td>
<td>10 (63%)</td>
<td>11 (85%)</td>
</tr>
<tr>
<td>Number of structure queries</td>
<td>12</td>
<td>7</td>
</tr>
<tr>
<td>Structure queries satisfied by ActiveAspect</td>
<td>6 (50%)</td>
<td>6 (86%)</td>
</tr>
<tr>
<td>Useful structures noticed</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>Useful structures noticed in ActiveAspect</td>
<td>4 (100%)</td>
<td>5 (83%)</td>
</tr>
</tbody>
</table>

Table 5.2: Summary of recorded structure information events.

queries that involved structure related only distantly to the task. Examples include a query about the members of the Figure class from the drawing framework used to implement the tool, and a request for the call hierarchy for the mousePressed method, which is invoked by the drawing framework. These types of queries partially account for the lower percentage of required structure found by Subject 1 using the active model. There was a single logged case in which the answer to a subject’s structure query could not be answered using the model but would clearly have contributed to their understanding. This query by Subject 1 was “What is the type hierarchy for AbstractGraphicalEditPart?” The answer to this query would help determine the best place in the hierarchy for declaring a new field needed for the task.

In addition to structure that was explicitly investigated, the subjects reported noticed structure, structure they deemed relevant but for which they did not actively search. Both subjects discovered such structure in 4 and 6 events, respectively. For Subject 1, the active model was the source of the noticed structure in all 4 cases. The relevant structure information noticed by Subject 2 was found in the active model in 5 (83%) of cases. Table 5.2 summarizes the structure information events from the log for each subject.
Qualitative Results

Figure 5.1 shows a diagram of the active model for the OnDemandRelationships aspect after the expansion operation has been applied six times. The presented structure includes all three classifiers that were edited by the subjects: ModelRelationship, MemberEditPart, and OnDemand Relationships. For the ModelRelationship class, four members for controlling the visible state of a relationship that are introduced through inter-type declarations are shown. In the MemberEditPart class, the displayed mouseEntered method is similar to the mousePressed method that is a required part of the code to understand for the change task. When the mouseEntered method is hovered, an arrow appears indicating the presence of an advice that applies to that method. Both subjects’ solution for the change task involved writing an advice very similar to the one shown. Thus the diagram was able to show simultaneously much of the structure that was useful in implementing the task.

Subjects used the expansion operator primarily to look for callers of a method of interest. This approach was successful in nearly all cases. However, both subjects found this feature awkward to use because the operation needed to be performed several times before the relationship of interest appeared. When the relationship did appear, the diagram had often become cluttered with relationships in which the subject was not interested at the time. Furthermore, some relationships added by the expansion operation were shown only when hovering over an associated member. The subjects found this behavior confusing because they could not see the result of an expansion. They both suggested that expanding relationships associated with a particular member of interest would likely improve the ability of the expansion operation to provide the information of interest.

The active model investigated in the study included an aggregate element advised by the OnDemandRelationships aspect (top-center in Figure 5.1). This aggregate node presents one or more methods in each of four classes. Neither of the subjects’ logged events involved any of these classes. We consider the aggregation successful because it prevented the display of three unnecessary classifier nodes. The abstraction operation also aggregated two accessor methods in the ModelRelationship class. At one point during the task, Subject 2 wished to see the callers of one of the methods, although this was not required for implementing the change task. In this case, the abstraction caused the call arrows to connect to a single aggregate member,
which made it difficult to determine the call targets.

Overall, then, within a reasonable number of expansion steps, the active model included most of the crosscutting structural information needed by the subjects. The abstraction introduced in the model and diagram helped to elide information that would otherwise have cluttered the diagram. Some refinement on the operation of the expansion operation is needed to make it easier for developers to focus the model to provide detail on the parts of the model of most interest.
Figure 5.1: A screenshot of the active model diagram for the structure associated with the change task subjects were asked to implement. The expansion operation has been applied six times and the user is currently hovering over the mouseEntered(...) method.
5.1.3 Limitations

Our case study results rely upon the log of events created by the experimenter as the subjects were performing the task. This log may be incomplete because the subjects found it difficult to state precisely the structure in which they were interested. The log may also be incomplete because subjects did not always volunteer information, rather the experimenter had to prompt for whether a query about structure was occurring. The possibility thus exists that there is structural information not provided by ActiveAspect that we did not consider in our analysis. If such situations occurred, we believe they were mostly limited to localized information a subject could glean from the source code as we prompted the subjects for their query when they accessed crosscutting structural information presented in other Eclipse views.

5.2 AJDT Comparison

AJDT is the most commonly used tool for presenting crosscutting structure described in an AspectJ aspect. To further evaluate whether ActiveAspect presents the right structural information within fewer views, we compare where the information of interest to the developers would be found in the views provided by AJDT.

The information needed by the subjects can be obtained using the AJDT outline, AJDT call hierarchy, and Eclipse JDT type hierarchy tree views. To satisfy the events we logged during the study, subjects 1 and 2 would have needed to open 11 and 6 views, respectively. Table 5.3 details this information.

The need to use several views requires a developer to synthesize information between views. For example, the key structure to understand to

<table>
<thead>
<tr>
<th>View</th>
<th>S1</th>
<th>S2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Outline</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>Call Hierarchy</td>
<td>7</td>
<td>4</td>
</tr>
<tr>
<td>Type Hierarchy</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Total Views Re-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>quired</td>
<td>11</td>
<td>6</td>
</tr>
</tbody>
</table>

Table 5.3: Number of Eclipse or AJDT tree views required to satisfy logged information events.
complete the modification task was an advice on `mouseEntered` methods, another advice on `mouseExited` members, and a method invoked by these advice to set relationship visibility in the diagram. Using AJDT and Eclipse views to discover this structure requires separate investigation of the shadows of each advice using the outline or two cross reference views, as well as a callee hierarchy for each advice\(^1\). In contrast, this information can be obtained from an active model simply by hovering over the advice of interest and noting the symmetry of their shadows and that they both invoke the same method.

Many of the information events in the log could be satisfied using the outline view for the `OnDemandRelationships` aspect. However, this view shows all 30 declarations in the aspect. When fully expanded, 70 program elements and their relationship to the aspect are visible. Due to the hierarchical presentation, a developer must synthesize information even within this view. Finding advice with the same shadows, for instance, requires expanding child nodes and comparing the results manually. Understanding that classes containing some of the advised elements realize a common interface requires similar digging through the nodes and synthesizing information. In contrast, ActiveAspect clearly shows the pattern (Figure 5.1).

\(^1\)We have assumed this feature to be available for the purpose of this analysis although the current version of AJDT does not yet support callee hierarchies for advice.
Chapter 6

Discussion

The case study presented in the previous section provides initial evidence to support our claim that useful active models of crosscutting structure can be produced automatically. However, several open questions remain.

6.1 Generalizeability

One question is whether the active model approach can be applied to present other means of identifying crosscutting structure. To sketch what would be required, we consider the steps involved in building an active model tool for concern graphs [20] as captured by the FEAT [3] tool. A concern graph describes crosscutting structure as program elements and relationships drawn from a program model. The FEAT tool supports the creation of concern graphs from the elements and relationships comprising a Java program. A similar set of steps would be needed to create an active model tool for concerns described with CME [7].

The first consideration is what elements and relationships to support in the active model. In this case, these elements and relationships would be the same as those in a structural representation of a Java program. A subset of the elements and relationships in ActiveAspect would suffice. The second consideration is the notation used to display a diagram from the active model; here also, a subset of the ActiveAspect notation would suffice. Third, we must define heuristics for projection, which might be selecting only public members and their relationships in the concern graph. Fourth, we must choose expansion heuristics; they may be similar to ActiveAspect’s, such as adding inter-class method calls first, followed by field references. Finally, heuristics for abstraction must be set, such as if a method calls similarly named methods in $N$ other classes, they should be aggregated with only one call selected as an example.

Figure 6.1 shows a mock-up of an active model diagram for a concern graph. In this example, an abstraction aggregates several classes that extend
Figure 6.1: Mock-up of an active model diagram for a concern graph.

Figure 6.2: Mock-up of the active model diagram in Figure 6.1 after expansion.
AbstractEditPart by placing them in a package classifier labeled with the common parent package, editparts. Two classes in this package classifier are represented by a single aggregate member. Figure 6.2 shows the same active model after it has been expanded to include additional structure.
Although creating a new active model tool is not insignificant, we believe the effort in selecting appropriate heuristics is reasonable as the developer is answering questions within a framework, rather than attempting to reduce a graphical clutter problem with no structure.

6.2 Local Expansion

A second question is whether a general expansion operation is appropriate, or if the expansion should be directed by the developer. We had chosen to investigate a general expansion operation as a means of reducing the need for a developer to break out of the flow of their work to access more information to form a specific query. However, both subjects in our case study commented that they most often needed to see additional relationships for a particular program element rather than for the whole model. Each subject expanded repeatedly until relationships appeared for the element of interest. These operations required too much time, and the subjects were often not interested in the other relationships that were added to the model. Another problem was that some newly added structure was immediately set to be only shown on mouse hover. This made it difficult to determine where new structure was added. These comments suggest an approach in which a specific member of classifier can be used as the focus for which more relationships are desired. This approach would have more quickly satisfied each of the method call structure queries logged during the study.

6.3 Scalability

A third question is whether the abstraction operation is sufficient to allow the approach to scale to handle most crosscutting structures with which developers will need to work. One indication that the approach will work with moderately sized crosscutting structures comes from the structure used in the task for the case study. As described earlier, this crosscutting structure includes well over 100 program elements and relationships.

The ability of our approach to handle large crosscutting structures rests largely on the effectiveness of the abstraction operation. We believe useful abstraction heuristics can be stated because the intent of the approach is to provide detailed, focused information. With each display of the model,
we can choose to elide structure that is not the focus of the current part of the investigation by the developer. Our approach is more closely aligned with a fish-eye view [10] of the structure of interest, than of earlier reverse engineering approaches (e.g., [14]), which attempted to abstract an overview of the structure. The reverse engineering approaches suffer from a difficulty in producing meaningful names for abstracted entities. By remaining close to the source, and by providing examples of the aggregation, we remove the need to automatically produce such a name, and hence do not suffer from this problem.

6.4 Diagram Layout

The initial implementation of ActiveAspect does not include an automatic diagram layout feature. Instead, classifiers are arranged in a grid and developers are required to manually rearrange them to improve the layout. We believe this is acceptable for an early prototype that explores our approach. However, an industrial-strength version of the tool would require a more sophisticated layout feature.

The diagram layout algorithm required for tools such as ActiveAspect would be difficult to develop due to the unique requirements of the tool. One requirement is that the UML conventions that inheritance relationships appear vertically and reference relationships be horizontal. A further complication is that the aspect is a central node in the graph that typically has more relations defined on it than other nodes. This means that the diagram will likely need to be arranged relative to a centered aspect, or one that is in the top left corner. Another more significant complication is that the diagram layout must support the dynamic addition of new classifiers and relationships without the ability to predict which of these elements will need to be added.
Chapter 7

Conclusion

Existing approaches for presenting crosscutting structure include tree views, which require developers to manually synthesize information across multiple views, and static structure diagrams, which tend to suffer from excessive graphical complexity. We have introduced active models as an approach that addresses these problems by presenting the right subset of crosscutting structure at the right time. The right information is determined through automatic projection and abstraction operations that select elements and relationships likely to be of interest and that abstract those elements and relationships to control the diagram complexity when too many similar cases occur. The information is presented at the right time through a combination of a user-driven expansion operation that adds detail to the model, and interaction features that show some information only on demand by the user.

Active models differ from existing modeling approaches by allowing a developer to focus on a slice of the model of the entire system. Active models differ from existing visualization approaches by considering how to present a small, focused, detailed view of program structure.

We have demonstrated that the active model approach can be realized by developing the ActiveAspect tool, which implements the approach to present the crosscutting structure associated with an AspectJ aspect. A case study of two developers using ActiveAspect showed that the majority of the structural information needed to implement a change task can be provided by ActiveAspect through user interaction and operations driven by simple heuristics.

There are a number of ways in which the active model presentation approach can be improved and extended to present other kinds of structure descriptions and support other development tasks. We believe that, by addressing shortcomings of existing approaches, active models hold promise for improving the way developers work.
7.1 Research Contributions

This thesis makes three contributions:

- it introduces the active model approach as a means of presenting crosscutting structure to a developer;
- it describes how the approach was applied to produce a tool, called ActiveAspect, for displaying crosscutting structure as defined by an AspectJ aspect, and
- it presents initial evidence that the approach can scale to present non-trivial crosscutting structure effectively to a developer.

7.2 Future Work

The structure presentation approach supported by the active model approach leads to several interesting directions for future work. These include improvements to the tool itself as well as exploring other applications of the techniques supported by ActiveAspect.

7.2.1 Interaction Technique Validation

ActiveAspect proposes an interaction technique where additional relationships are provided as the user hovers over particular elements of the diagram. This is one of several possible interaction techniques that are possible. We believe it is necessary to study other interaction techniques to determine which is the most effective for model exploration. In particular, we wish to compare the following approaches:

- hover to view relationships, click to lock the relations on-screen
- hover to highlight relationship endpoints
- click to toggle relationship visibility
- show all relationships of a particular type selected by the user
- show all relationships
These interaction techniques can be evaluated by asking subjects to answer structure-related questions using a specific interaction technique. The subjects' performance can then be compared in search of productivity differences across the techniques.

7.2.2 Degree-of-Interest Guided Heuristics

The ActiveAspect heuristics rely solely on structural properties of the program. It would also be interesting to consider individual developers' interaction history to assist in determining which parts of a program structure are likely to be of interest.

The Mylar tool [16] assigns a degree-of-interest (DOI) value to program elements and relationships navigated by a developer. Frequently visited elements have high DOI values, indicating that the element is relevant for a particular task.

ActiveAspect could take advantage of these DOI values to guide the heuristics during the projection, expansion, and abstraction operations. For example, the projection operation could use DOI values to help select the most relevant elements to appear in the initial model. The operation should still perform a standard structural analysis to identify the core structure, however the DOI values could then be used to include additional structure known to be of interest to the developer in order to show how the projected structure relates.

During the expansion operation, the DOI values could be used to favor the addition of elements known to be of interest to the developer. Furthermore, the abstraction operator could select concrete member examples with high DOI values as well. For classifier abstraction, it would be helpful to avoid the aggregation of classifiers whose members have a high total DOI.

A limitation of this approach is that it assumes the developer has previously worked with a portion of the structure to be presented by the tool. This may not always be the case, but DOI information from other developers working on a similar task may be available. Furthermore, it may be possible for DOI values to be recorded quickly during a new development task so that they are available to guide the active model tool after some interaction has occurred.
7.2.3 Forward Engineering and Refactoring

The current implementation of ActiveAspect, while interactive, does not support changes to the underlying source code. However, we believe there are some AspectJ-specific development tasks that can be performed at the level of abstraction provided by the ActiveAspect active model view.

For example, the view could be used for refactoring existing functionality so that it is encapsulated by a new aspect. To begin the refactoring, a developer would project the crosscutting structure of interest (as captured by a concern graph, for example) so that its structure is displayed in a diagram. The developer would then create a new empty aspect classifier that would also be placed in the diagram. The refactoring would then be accomplished through several interactive features such as those listed below:

- Dragging a method body onto an aspect creates an advice body with the same behavior as the previously existing method.
- Dragging an arrow from an advice body to parts of the structure adds the corresponding join point to the shadow of the advice.
- Dragging an arrow from the aspect to a member in another class transforms the member to an inter-type member declared in the aspect.
- Dragging an inheritance arrow between types creates a corresponding declare parents statement in the aspect.
- Deleting arrows reverses the 3 operations above.
- Delete advises arrows to remove a join point from an advice shadow.

This approach would allow developers to maintain a high-level perspective of the relevant program structure while performing a refactoring task.

7.2.4 ActiveObject

We believe the active model approach can be applied as a structure presentation approach even when there is no previously existing concern description. In this case, the structure description is constructed as required when the user identifies a program element as a starting point.
Once a starting point element is selected, the most relevant structure associated with that element could be projected in the diagram view. The relevant structure would be determined by heuristics such as those described for the ActiveAspect projection operation and could also be guided by the DOI values described above. The resulting diagram could then be expanded and abstracted much as an active model in ActiveAspect. We call this variation of our approach \textit{ActiveObject}.

Although this approach is not suitable for presenting crosscutting structures, it has the advantage that a previously existing concern structure is not required. Therefore, this approach may ease adoption and evaluation of a subset of the techniques implemented by ActiveAspect.
Bibliography


package telecom;
/**
 * The Billing aspect deals with... billing.
 * How much money did each connection cost?
 * How much money did each call cost?
 * How much money is being debited to a customer?
 *
 * Billing can depend on many things, such as timing, the type of
 * the connection, some special discounts the customer has,
 * special features, etc. In here, it depends only on timing and
 * on the type of the connection.
 */
public aspect Billing {
    // precedence required to get advice on endtiming
    // in the right order
    declare precedence: Billing, Timing;

    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;
}