Presenting Crosscutting Structure with Active Models

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

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

ABSTRACT

When modifying or debugging a software system, among other tasks, developers must often understand and manipulate source code that crosscuts the system's structure. These tasks are made more difficult by limitations of the two approaches currently used to present details of crosscutting structure: tree views and structural 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 crosscutting structure to a developer at the *right* time. An active model is produced as a result of three automated 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.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques—*Object-oriented design methods, User interfaces*; D.2.6 [**Software Engineering**]: Programming Environments—*graphical environments, integrated environments, interactive environments*

Keywords

Program structure, design views, structure presentation, aspect-oriented programming, AspectJ

1. INTRODUCTION

Many tasks performed by a software developer involve code that crosscuts a system's structure. For example, modification tasks often involve changing code that is spread in a disciplined way across multiple classes and methods [18]. Most debugging tasks involve understanding execution slices that cut across system modules [26]. Various approaches have been proposed to help identify the crosscutting code of interest for a task (e.g., [22, 26, 27]). 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., [22]) to largely automatic dynamic analyses (e.g., [27]). 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.

To overcome these limitations, we have developed the *active model* approach that automatically determines important crosscutting structure information and displays that information as a structure diagram. Our approach uses several interactive features to elide structure from a displayed view until it is needed (Section 3). 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.

Three automated operations are used to produce an active model: projection, expansion, and abstraction. The projection operation selects a representative overview of the crosscutting program structure of interest to a developer. 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 applied to reduce complexity while retaining information content.

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 [1] aspect (Section 3). 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

AOSD 06, March 20–24, 2006, Bonn, Germany

the crosscutting structure. The approach can be applied to other languages and representations of crosscutting structure and in this paper we also sketch how our approach can be used to present crosscutting structure represented as a concern graph [22] (Section 5).

An active model is only useful to a developer 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 (Section 4). Using ActiveAspect, the two subjects were able to find the crosscutting structural information needed to make progress on the assigned task. We analyzed and report on the results of the study both quantitatively and qualitatively. We also report on a comparison of accessing the structural information needed by the subjects using the state-of-the-art tree views provided by the AspectJ Development Tools (AJDT). In short, the active model performed as expected, presenting the information directly rather than requiring the subjects to synthesize information from multiple views.

This paper 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.

2. EXISTING APPROACHES

Existing approaches for presenting the details of crosscutting program structure are based either on tree-based displays or node-and-link structure diagrams. Although SeeSoft-like [12] views and tree maps can be used to present a high-level overview of crosscutting (e.g., [8, 14]); we do not include a detailed discussion of these tools as they lack detailed information that is necessary for most development tasks and thus serve a different purpose than our approach.

2.1 Tree Views

Several tools use tree views to present crosscutting structure (e.g., [3, 4, 7, 17, 20]). These views display program elements according to a hierarchical relationship that is rooted at an element of interest. 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.

To explain some of the advantages and limitations of tree views, we describe the use of the state-of-the-art AspectJ Development Tools (AJDT) [3] plug-in for the Eclipse [2] development environment to investigate crosscutting structure associated with the `Billing` aspect in a sample telecommunications program distributed with AspectJ (Figure 1).

One advantage of tree views is their simplicity. The Outline view in Figure 1 clearly shows which calls are advised by a particular piece of advice in the aspect (the children of the expanded `afterReturning` node). A significant disadvantage of using tree views for presenting crosscutting structure is that developers must often synthesize information across multiple views. For example, a relationship between

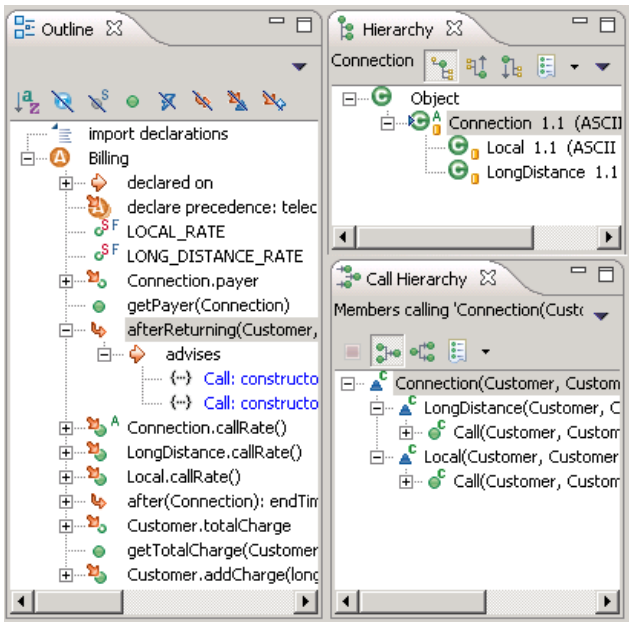

Figure 1: AJDT and Eclipse tree views displaying part of the crosscutting `Billing` aspect structure.

the calls that are advised becomes apparent only after considering information in the Hierarchy view on the top right of Figure 1. A developer must synthesize information across these views to build a model of the program's crosscutting structure. Another disadvantage of tree views is that they are frequently replaced (re-used) for new queries, causing developers to have to remember key parts of the structure as it is discovered.

The heart of the problem is that the structure of a tree-based presentation 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 structure consists of a, potentially cyclic, directed graph of program elements and relationships. 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.

2.2 Static Structure Diagrams

Static structure diagrams, such as UML class diagrams [19], are node-and-link diagrams where the nodes are program elements (e.g., classes and methods) and the links are program relationships (e.g., method invocations and inheritance). These diagrams can directly show crosscutting program structure in a single view, accommodate multiple relationship types, and make it easier to identify structural patterns.

Several existing round-trip engineering tools produce object-oriented structure diagrams from code(e.g., [6]). All too often, the diagrams produced by these tools quickly become cluttered with nodes and relationships. Consider, for example, displaying just the subset of program structure that performs event logging in a system. This crosscutting structure could involve nearly all classes in the system. The

presentation of this structure by an existing round-trip engineering tool would contain each of the affected classes with some number of relationships, such as associations, between them. The resulting diagram would be excessively complicated for any non-trivial application.

One possibility to overcome this graphical complexity problem is to introduce new notations specifically for modeling crosscutting program structure. Several notations have been proposed for modeling such structure associated with an aspect [24, 25, 9, 15]. The approaches taken in these notations apply one of two high-level strategies: aspect classifiers or aspect bindings.

In the first strategy, a new classifier[1] is introduced to represent aspects similar to how a class is modeled in UML [24, 25]. 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 [9, 15] 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.

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 section we describe the general active model approach using specific examples from a concrete implementation of the approach called ActiveAspect.

An active model is a data structure that represents a part of a program's structure, such as classes, methods, and inheritance relationships, for the purpose of displaying it in an interactive class diagram-like view. The initial content of an active model is selected automatically from a given description of the crosscutting structure of interest. Examples of structure descriptions include concern graphs captured by the FEAT tool [4] and the crosscutting structures encapsulated by aspects in aspect-oriented languages. As shown in Figure 2, an active model may also contain structure obtained directly from the source code in addition to the source code structure identified by the structure description.

A developer creates an active model by requesting a projection of a given crosscutting structure description (Figure 3). The projection operation selects the important structure to include in the initial 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 this operation are kept in

[1]The term *classifier* refers to encapsulation constructs such as classes, interfaces, and aspects.

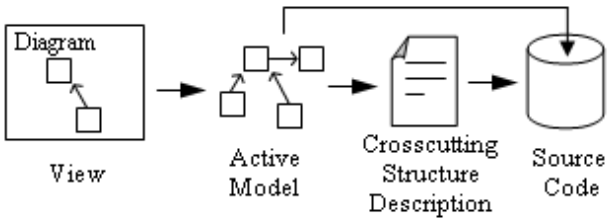

Figure 2: Active model data flow.

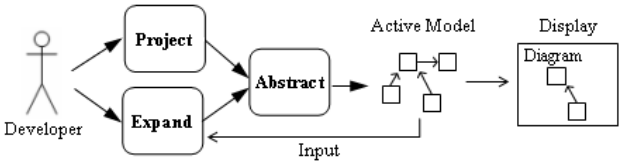

Figure 3: Active model work flow.

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.

The content of an active model is displayed as a design-like diagram view. Figure 4 shows an example of a diagram view of an active model. 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. There are many interaction models that can be supported by the diagram view. For example, hovering over a particular method of a class can cause method calls from that method to temporarily appear in the diagram. Clicking on the method could cause these relationships to be permanently displayed or can cause the display of the source code for the method.

Our ActiveAspect tool implements the active model approach to present crosscutting structure defined in an AspectJ aspect. ActiveAspect is packaged as a plug-in for Eclipse, and implementation details can be found in [10]. Developers invoke ActiveAspect on an aspect of interest to investigate its crosscutting structure.

3.1 Active Models for AspectJ

We consider the basis of an active model as the information that can appear in a UML class diagram extended with calls relationships. ActiveAspect extends this basic model and diagram view with constructs for AspectJ.

3.1.1 Aspects

In an AspectJ active model, an aspect is represented as a classifier in the same way as a class or interface. In addition to fields and methods, aspects can have pointcuts and advice as members. In the diagram, aspects are differentiated from other classifiers by an 'A' icon; pointcuts and advice also have special icons. Figure 4 (top left) shows an aspect with two advice denoted by cog icons.

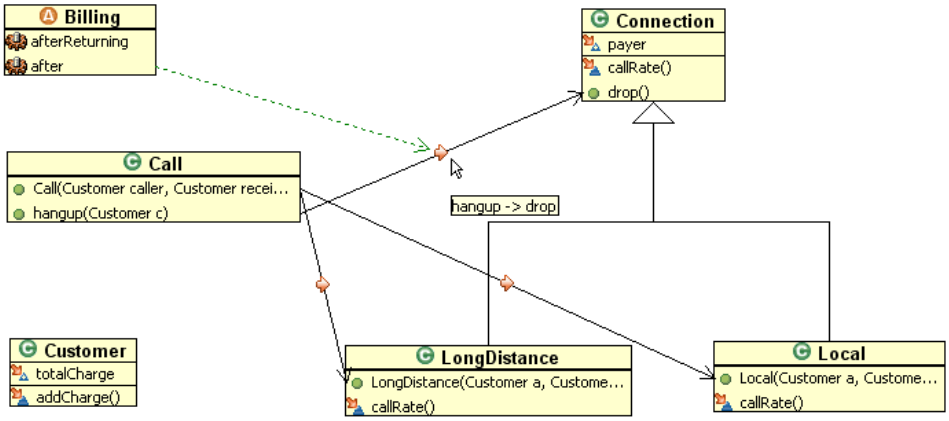

Figure 4: A diagram of an active model of the `Billing` aspect.

3.1.2 *Advice*

When a method execution is advised, this is represented in the active model and corresponding diagram by a member-to-member *advises* relationship between the advice and the advised method. When a method call or field reference is advised, the advises relationship connects the advice with the relationship representing the advised call or field reference.

Advice often advise a number of parts of the program structure and are therefore interactively displayed by hovering over affected parts of the diagram to avoid unnecessary complexity. Icons are used to indicate parts of the structure that are advised. In Figure 4, the user is hovering over an icon that indicates that the method call is advised. This causes the dashed advises relationship to be temporarily drawn, revealing the advice that applies at that location.

3.1.3 *Inter-type Declarations and* `declare parents`

The members declared by inter-type declarations 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. Active models include introduces relationships connecting the inter-type declaration and the declaring aspect.

In the diagram, introduces relationships are drawn from the name of the declaring aspect classifier to the inter-type declaration. These relationships are shown by hovering over the name of the declaring aspect or the declared inter-type member. Icons are used to indicate which members are inter-type declarations. For example, the `payer` field and `callRate()` method in the `Connection` class in Figure 4 are inter-type members.

AspectJ `declare parents` statements are modeled and diagrammed by inserting the declared inheritance (or realizes) relationship between the target type and it's new supertype. The inheritance relationship is annotated with an icon to indicate that the relationship is declared by an aspect. Hovering over the icon temporarily draws a relationship indicating the declaring aspect.

3.2 Projection

The initial content of an active model is produced by the projection operation which uses a set of heuristics to determine the most important elements and relationships of the crosscutting structure. The operation selects elements and relationships based on type and on structural properties, such as the number of relationships to and from a particular element. The heuristics can also determine which relationships are to be permanently displayed in the diagram or shown only while hovering over its endpoints.

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, the advice is added as a member of the aspect 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 4 shows a diagram of the `Billing` aspect after these projection rules have been applied to produce the model. 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.

3.3 Expansion

The expansion operation is invoked by the developer to add additional structure to an active model which provides either more detail about the structure already present in the model, or additional structural context. The operation considers program elements and relationships neighboring those that are currently visible in the diagram. Since some structural elements are more important than others, all possible additions are ranked by an importance value, which mea-

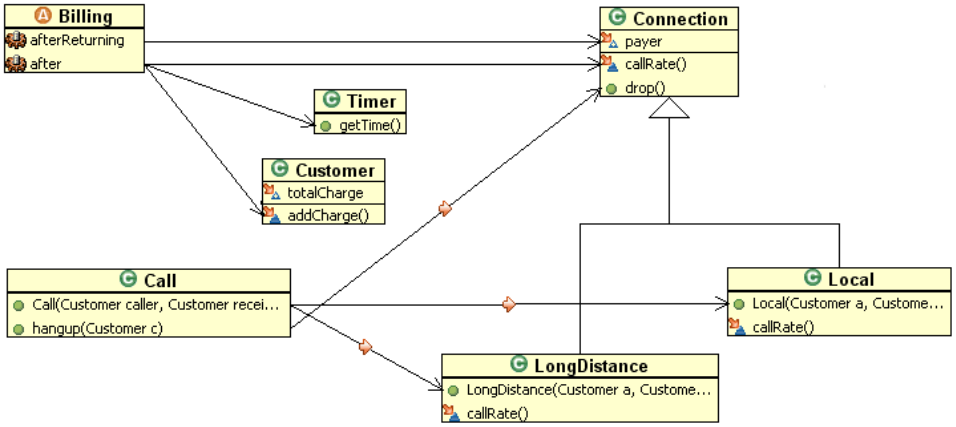

Figure 5: A diagram from the active model of `Billing` after the expansion operation has been applied.

Table 1: These reference types are added to the active model during expansion in the listed order.

Order	Relationship
1	Method call from advice body
2	Reference from inter-type method
3	Reference to inter-type method
4	Reference from advice body to field in another type
5	Reference to inter-type field

sures the likelihood that the addition will be of interest to the developer. Importance values are computed as a function of the element or relationship type and other structural information in the source code, such as the parent type and number of associated relationships. The expansion operation selects the top ranking entries to add to the model and diagram. The intent of the 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.

The expansion operation in ActiveAspect adds additional method calls and field references to the model. The classifiers containing referenced methods or fields are also added as required. ActiveAspect uses a simple ranking as the importance values as shown in Table 1.

For the programs we used to tune the heuristics,[2] 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.

[2]Telecom and SpaceWar sample programs in the AspectJ distribution and AspectTetris developed by Gustav Evertsson (http://www.guzzzt.com/coding/aspecttetris.shtml).

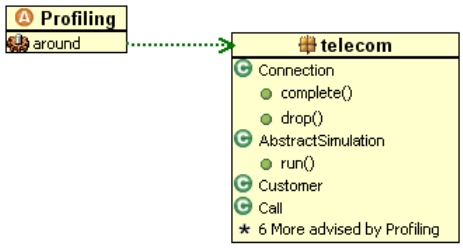

Figure 6: A diagram of an active model of a profiling structure where many elements have been abstracted.

Figure 5 shows the result of applying the expansion operation to the Billing aspect twice using ActiveAspect. The operation has added several method calls made from the advice in the `Billing` aspect because these relationships have the highest ranking (Table 1). The `Timer` class and `getTime()` method were added to the model because `getTime()` is the target of one of the new relationships.

3.4 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 using structural information from the source. To ensure the diagram retains detailed information, a small number of examples of the aggregated information are included. Examples are selected based on their importance value, which is a function of their type and the type of their relationships.

The diagram in Figure 6 shows that the around advice in the `Profiling` aspect advises a number of elements aggregated by the `telecom` package classifier. The members of the `telecom` package classifier are *examples* of classes, and methods in those classes, that are advised by `Profiling`. The last member in the `Profiling` package classifier aggregates six more classes that are advised by `Profiling`.

Hierarchical structure in a Java program can be used to abstract related classes; a package classifier can be introduced into the model and diagram with several classes retained as examples. When hierarchical structure in the source cannot help directly, we look for a shared property, such as implementing a common interface, then a non-source related aggregate member may be introduced into the class (e.g., the aggregate member may be named: n methods implementing `IInterface`). In the extreme case, we may have several elements that are not obviously related, if they are of low importance, we sometimes choose to create a new aggregate member that simply groups these elements (e.g., the aggregate member may be named: n more methods).

In ActiveAspect, abstraction of classifiers 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, as shown in Figure 6.

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 importance value of an element is given by the value assigned to its element type plus the sum of the values assigned to the relationships connecting to the element.

The values assigned to elements and relationships by ActiveAspect for example selection are listed in Table 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 do not have 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.

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.

Several other approaches for abstracting structure diagrams have been proposed (e.g. [11, 21]). However, these techniques do not provide detailed examples and the abstraction operations are limited to operating on the higher level semantics available in UML class diagrams.

4. 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.

Table 2: Element and relationship values used to select examples during the abstraction.

Elements

Type	Value
Advice	10
Classifier	5
Pointcut	5
Inter-type Method	5
Inter-type Field	5
Constructor	3
Method	2
Field	1

Relationships

Type	Value
Advises	5
Introduces	4
Uses Pointcut	4
Extends	3
Realizes	3
References	2

4.1 Case Study

The case study involved a change task to the AspectJ code for our own ActiveAspect tool. ActiveAspect comprises more than 100 classes and approximately 7000 lines of code. We chose to use our own tool as the target because it was the largest available AspectJ system that did not use aspect libraries, which are not yet supported by the tool. The system was developed before it was determined that the code would be used in the case study and the aspects were written without regard for their potential use as concern descriptions.

The task was to enable a user to make a relationship that is shown only on hover permanently visible by clicking on either of its endpoints. 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.

4.1.1 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

Table 3: Excerpt from a log of recorded structure information events.

Structure	Event Type	In Model
Callers of setHidden()	Query	Yes
Two advice call showOnDemand()	Noticed	Yes
Type hierarchy of AbstractGraphicalEditPart	Query	No
Call hierarchy for mouseEntered()	Query	No

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 were graduate computer science students who were familiar with AspectJ syntax but did not have substantial AspectJ development experience. Neither subject had 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 2.1). 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 [5].

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. The experimenter, who is an author of this paper, logged these instances, including whether the information was found using ActiveAspect. Table 3 shows an excerpt from the log recorded during an experiment (We explain the second and third columns in section 4.1.2.1).

4.1.2 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.

4.1.2.1 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 `showOnDemandRels()`?" (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 `setHidden()` is introduced from the aspect" (Subject 2).

During the implementation of 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 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 4 summarizes the structure information events from the log for each subject.

4.1.2.2 Qualitative Analysis.

Figure 7 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 `OnDemandRelationships`). 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.

Table 4: Summary of recorded structure information events.

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

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 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 7). 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, 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 helped to elide information that would otherwise have cluttered the diagram. Some refinement on the way the expansion operation is invoked is needed to make it easier for developers to focus the model to provide detail on the most relevant program structure.

4.2 Comparison With AJDT

AJDT is the most commonly used tool for presenting crosscutting structure described by 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.

Table 5: Number of Eclipse or AJDT tree views required to satisfy logged information events.

View	S1	S2
Outline	3	1
Call Hierarchy	7	4
Type Hierarchy	1	1
Total Views Required	**11**	**6**

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).

The need to use several views requires a developer to synthesize information between views. For example, the key structure to understand to 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.[3] 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 7).

4.3 Limitations

Our case study results rely upon the log of events created by the experimenter as the subjects performed 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

[3]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.

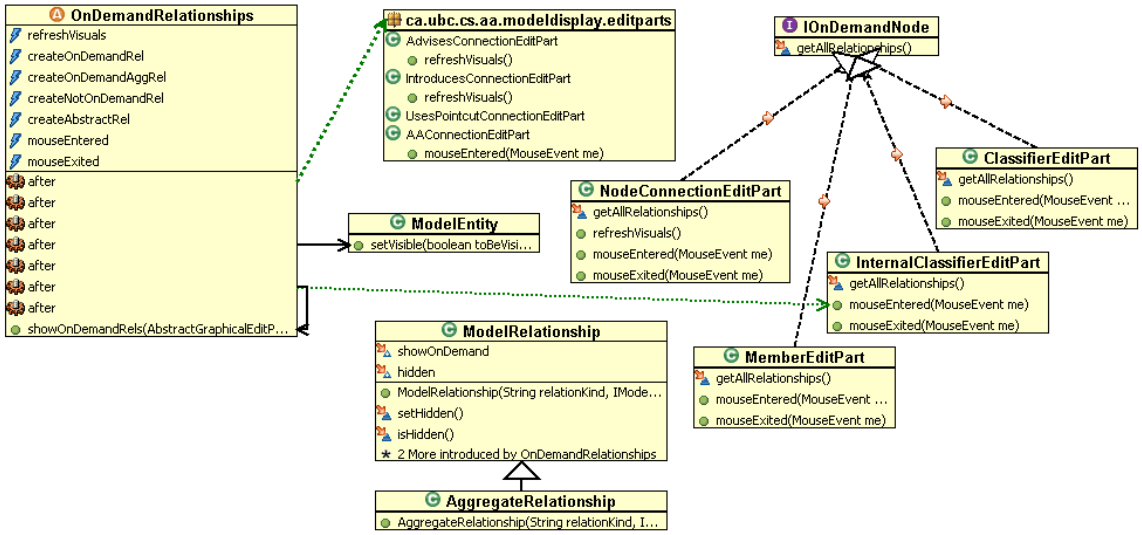

Figure 7: A snapshot of the active model diagram for the structure associated with the case study change task after six expansions. The user is currently hovering over the `mouseEntered(..)` method.

the subjects for their query when they accessed crosscutting structural information presented in other Eclipse views.

5. DISCUSSION

Although further validation is necessary, the case study presented in the previous section provides initial evidence that useful active models of crosscutting structure can be produced automatically. However, several open questions remain.

5.1 Alternative Structure Descriptions

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 [22] as captured by the FEAT [4] tool.[4] 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 relationships in the concern graph. Fourth, we must choose expansion heuristics; there may be similar to ActiveAspect's, such as adding inter-class method calls first, followed by field references. Finally, rules 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. Although creating a new active model tool is not insignificant, we believe the effort in selecting appropriate rules and heuristics is reasonable as the developer is answering questions within a framework, rather than attempting to reduce a graphical clutter problem with no structure.

[4]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 structure comprising a Java program. A similar set of steps would be needed to create an active model tool for concerns described with CME [7].

5.2 Improving the Expansion Operation

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 a focused expansion approach in which a classifier or specific member of a classifier is used as the focus for which more relationships are desired. This approach is similar to that supported by Relo [23] although Relo does not automatically manage complexity through abstraction. Focused expansion would have more quickly satisfied each of the method call structure queries logged during the case study and has been implemented in a later version of ActiveAspect.

5.3 Alternative Presentations

A third question is whether ActiveAspect's effectiveness could be improved through alternative active model presentation and interaction techniques. We consider ActiveAspect to be an experimental user interface for active models and there are many variations that could be implemented.

For example, member highlighting could be used in place of relationship arrows and the click-to-stick functionality implemented in the case study appears to be a promising improvement. There are also many possible variations for the visual cues used to indicate the presence of crosscutting structure. Further study is required to investigate and validate these alternatives.

5.4 Scalability

There is a question as to 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 rules 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 [13] of the structure of interest, than of earlier reverse engineering approaches (e.g., [16]), 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. SUMMARY

Existing approaches for presenting crosscutting structure include tree views, which require developers to manually synthesize information across multiple views, and existing 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 believe that active models hold promise for breaking developers out of source code only views of their systems.

7. ACKNOWLEDGMENTS

The authors would like to thank Mik Kersten for his assistance with the AspectJ compiler and Chris Dutchyn, Kevin Sullivan, and Miryung Kim for commenting on earlier drafts of this paper. This work was supported by a UBC graduate fellowship and NSERC.

8. REFERENCES

[1] AspectJ. http://www.eclipse.org/aspectj/, 2005.
[2] Eclipse. http://www.eclipse.org/, 2005.
[3] Eclipse AspectJ Development Tools Project. http://www.eclipse.org/ajdt/, 2005.
[4] FEAT: An Eclipse Plugin for Locating, Describing, and Analyzing Concerns in Source Code. http://www.cs.ubc.ca/labs/spl/projects/feat/, 2005.
[5] Graphical Editor Framework. http://www.eclipse.org/gef/, 2005.
[6] Omondo. http://www.omondo.com/, 2005.
[7] The Concern Manipulation Environment Project. http://www.eclipse.org/cme/, 2005.
[8] Asbro. http://www.cs.manchester.ac.uk/cnc/hendrik/asbrohome.php, 2006.
[9] S. Clarke and R. J. Walker. Composition patterns: An approach to designing reusable aspects. In *Proc. of ICSE*, pages 5–14, 2001.
[10] W. Coelho. Presenting Crosscutting Structure with Active Models. Master's thesis, University of British Columbia, 2005.
[11] A. Egyed. Semantic abstraction rules for class diagrams. In *Proc. of ASE*, page 301, 2000.
[12] S. G. Eick, J. L. Steffen, and J. Eric E. Sumner. Seesoft-a tool for visualizing line oriented software statistics. *IEEE Trans. Softw. Eng.*, 18(11):957–968, 1992.
[13] G. W. Furnas. Generalized fisheye views. In *Proc. of CHI*, pages 16–23, 1986.
[14] W. G. Griswold, J. J. Yuan, and Y. Kato. Exploiting the map metaphor in a tool for software evolution. In *Proc. of ICSE*, pages 265–274, 2001.
[15] I. Groher and S. Schulze. Generating aspect code from UML models. In *AOSD Workshops*, 2003.
[16] D. Hutchens and V. Basili. System structure analysis: Clustering with data bindings. SE-11(8):749–757, 1985.
[17] D. Janzen and K. D. Volder. Navigating and querying code without getting lost. In *Proc. of AOSD*, pages 178–187, 2003.
[18] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proc. of ECOOP*, pages 220–242. 1997.
[19] OMG. The Unified Modeling Language Specification. Version 1.4, 2001.
[20] J. Pfeiffer, A. Sardos, and J. R. Gurd. Complex code querying and navigation for aspectj. In *Proc. of Eclipse Technology eXchange*, 2005.
[21] F. D. Racz and K. Koskimies. Tool-supported compression of uml class diagrams. In *Proc. of UML*, 1999.
[22] M. P. Robillard and G. C. Murphy. Concern graphs: finding and describing concerns using structural program dependencies. In *Proc. of ICSE*, pages 406–416, 2002.

[23] V. Sinha, R. Miller, and D. Karger. Interactive exploratory visualization of relationships in large codebases for program comprehension. In *Proc. of Eclipse Technology eXchange*, 2005.

[24] D. Stein, S. Hanenberg, and R. Unland. Designing aspect-oriented crosscutting in UML. In *AOSD Workshops*, 2002.

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

[26] M. Weiser. Programmers use slices when debugging. *Commun. ACM*, 25(7):446–452, 1982.

[27] N. Wilde and M. C. Scully. Software reconnaissance: mapping program features to code. *Journal of Software Maintenance*, 7(1):49–62, 1995.