Tools for Exploring and Editing Crosscutting Concerns

by

Doug Janzen

BComm(Hons), University of Manitoba, 1994

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF
Master of Science

in

THE FACULTY OF GRADUATE STUDIES

(Department of Computer Science)

We accept this thesis as conforming
to the required standard

The University of British Columbia

July 2004

© Doug Janzen, 2004
In presenting this thesis in partial fulfillment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the head of my department or by his or her representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Doug Janzen
Name of Author (please print)

July 21, 2004
Date (dd/mm/yyyy)

Title of Thesis: Tools For Exploring and Editing Cross-cutting Concerns

Degree: M.Sc. Year: 2004

Department of Computer Science
The University of British Columbia
Vancouver, BC Canada
Abstract

Typical programming languages allow only a single decomposition of a program into source files. This results in concerns that are difficult to work with because they cut across the primary decomposition. Current development environments do not provide adequate support for working with crosscutting structure because multiple tools are required to perform even simple explorations, forcing the developer to remember how parts of the code are related to one another, and causing disorientation by having to switch back and forth between multiple views. Furthermore editing of crosscutting structure must be done within the context of the single decomposition provided by the source files.

In this thesis we present the design and implementation of two prototype tools for working with crosscutting structure. With the JQuery prototype developers can construct many kinds of browsers by writing queries over program structure, and then incrementally extend these browsers to explore complex relationships between various code elements. With JQuery developers are able to perform complex explorations in the context of a single view, reducing disorientation. Their exploration path is explicitly maintained, thus reducing cognitive overhead. A case study was conducted that provides preliminary evidence for the usefulness of JQuery in performing a realistic development task involving crosscutting concerns.

The Decal prototype supports the editing of crosscutting structure through the use of virtual source files (VSFs) that provide two mutally crosscutting points of view. In the classes view each VSF contains all the declarations related to a particular class. The modules view provides an alternate decomposition where each VSF may contain declarations from multiple classes. Each VSF is generated from a single common representation of the program structure so that edits to a VSF in one view will be reflected in VSFs from the other view. By using Decal developers are not restricted to editing programs according to the single primary decomposition, but are able to choose either one of two decompositions that is most suitable to the particular task at hand.
Contents

Abstract ii

Contents iii

List of Tables vi

List of Figures vii

Acknowledgements viii

1 Introduction 1

1.1 Motivation .................................. 1
1.2 Tools for Working With Crosscutting Concerns .......... 3
1.3 The JQuery Prototype .......................... 6
1.4 The Decal Prototype .......................... 7
1.5 Research Contributions .......................... 8
1.6 Overview ................................ 9

2 Exploring Crosscutting Concerns 10

2.1 Introduction ................................ 10
2.2 Illustrating Example ............................. 14
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.3 The JQuery Tool</td>
<td>17</td>
</tr>
<tr>
<td>2.3.1 Getting Started</td>
<td>17</td>
</tr>
<tr>
<td>2.3.2 The Query Language</td>
<td>17</td>
</tr>
<tr>
<td>2.3.3 Contextual Menu Structure</td>
<td>19</td>
</tr>
<tr>
<td>2.4 Case Study</td>
<td>20</td>
</tr>
<tr>
<td>2.4.1 The Task</td>
<td>22</td>
</tr>
<tr>
<td>2.4.2 Summary of Task Progress</td>
<td>22</td>
</tr>
<tr>
<td>2.4.3 Case-study Examples</td>
<td>24</td>
</tr>
<tr>
<td>2.4.4 Case-study Conclusions</td>
<td>28</td>
</tr>
<tr>
<td>2.5 Implementation of JQuery</td>
<td>31</td>
</tr>
<tr>
<td>2.5.1 Customizing JQuery</td>
<td>32</td>
</tr>
<tr>
<td>2.6 Related Work</td>
<td>35</td>
</tr>
<tr>
<td>2.7 Conclusion</td>
<td>38</td>
</tr>
<tr>
<td>3 Editing Crosscutting Structure</td>
<td>40</td>
</tr>
<tr>
<td>3.1 Introduction</td>
<td>40</td>
</tr>
<tr>
<td>3.2 Motivating Example</td>
<td>44</td>
</tr>
<tr>
<td>3.3 The Decal System</td>
<td>47</td>
</tr>
<tr>
<td>3.3.1 The Modules View</td>
<td>48</td>
</tr>
<tr>
<td>3.3.2 The Classes View</td>
<td>51</td>
</tr>
<tr>
<td>3.3.3 Motivating Example Revisited</td>
<td>53</td>
</tr>
<tr>
<td>3.4 The Decal Database</td>
<td>55</td>
</tr>
<tr>
<td>3.4.1 Developing a Schema</td>
<td>55</td>
</tr>
<tr>
<td>3.4.2 Impact of Schema Design on the Language</td>
<td>57</td>
</tr>
<tr>
<td>3.5 A Practical Editing System</td>
<td>57</td>
</tr>
<tr>
<td>3.5.1 Editing Semantics</td>
<td>58</td>
</tr>
</tbody>
</table>
List of Tables

2.1 Some predefined predicates in the query language. .................. 20
2.2 A sample of the ways in which nodes in the tree can be extended... 21
List of Figures

2.1 Exploring Figures Implementation in JHotDraw. .................. 15
2.2 An early exploration of the Jin code. .......................... 25
2.3 Searching for the use of images in the Jin code. ................. 26
2.4 Exploring the BoardManager class. ............................ 27

3.1 The modules view crosscuts classes and the classes view crosscuts modules. ............................................ 43
3.2 The Extract-Edit-Absorb Cycle ................................. 44
3.3 Most natural OO decomposition for a simple AST implementation . 45
3.4 A decomposition of a simple AST implementation with open classes 47
3.5 Example VSF: The ast module .................................. 49
3.6 Example VSF: The printing Module ............................ 50
3.7 Example VSF: The Number class ............................... 52
Acknowledgements

I would like to thank my supervisor, Kris De Volder, for his help and stimulating discussions. I would also like to thank Emily, Taija and Matthew for their support and understanding during my studies.

DOUG JANZEN

The University of British Columbia
July 2004
Chapter 1

Introduction

1.1 Motivation

Many of the current approaches to software engineering view the development process as being an evolutionary cycle that continues throughout the lifetime of a piece of software [7]. This is in contrast to the classical waterfall approach where the stages of development are partitioned into a strict chronological order that begins with requirements gathering and ends with deployment and maintenance. Under the evolutionary approach software artifacts undergo continual revision and expansion as new requirements are discovered, bugs are found and the external environment changes.

One of the primary goals of software engineering is to assist developers in dealing with this dynamic environment, where a large part of their work is to change existing code rather than to write new code. One of the ways in which this can be done is through better separation of concerns [36]. To this end programming languages provide mechanisms for dividing programs into modules that represent particular design decisions, features or pieces of functionality which can be generally
referred to as concerns. Modularizing concerns helps during evolution because developers do not have to deal with the program in its entirety every time they want to make a change. They can focus on just the modules that relate to their task and the compiler can provide some assurances that implementation details will not have far reaching effects. Modules also provide a means for encapsulating generic functionality so that code can be reused across multiple projects.

However, not all concerns can be easily modularized. When a designer chooses a decomposition of a program into modules, he or she does so with the intent of making the expected evolution tasks easier for developers to perform. In practice, finding a decomposition that supports all required evolution tasks is often impossible. In some cases this is due to new requirements or environmental changes that could not have been predicted and so were not planned for. In other cases the programming language used to implement the software does not provide adequate means to modularly express the tangled web of interactions that exist between all the relevant concerns. This leads to the existence of concerns whose implementations are scattered across multiple modules. This mismatch between the chosen decomposition and the required programming tasks is often referred to as the tyranny of the dominant decomposition [42] and is one of the main motivations for aspect-oriented programming (AOP) [30].

Aspect-oriented programming builds on top of object-oriented programming by introducing new forms of modularity. AOP languages provide more flexibility in choosing a decomposition through the use of aspects which define both state and behaviour that can be woven into the object-oriented structure of a program. While this improves the coverage of decompositions over evolution tasks it does not provide a complete solution because there is still only a single decomposition available. AOP
approaches make it easier to modularize concerns that were previously scattered amongst object-oriented classes. At the same time though, they tend to scatter the implementation of classes across aspects. This makes some tasks easier to perform at the expense of making others more difficult.

1.2 Tools for Working With Crosscutting Concerns

The process of carrying out an evolution task involves several activities. In this thesis we focus on two activities that can be described as “exploration” and “editing”. The exploration activity involves looking at the source code of the program and using tools to understand what parts need to be changed and what the implications of the changes would be. The editing activity involves making changes to the source to bring about the desired changes to the behaviour of the program.

Current integrated development environments (IDEs) provide a variety of tools for assisting with these activities, such as browsers and search tools for exploration, and smart editors that provide code assist, refactoring support, quick fix, and other tools to support the editing of code. These tools are typically designed for tasks that have to do with the object-oriented structure of programs and provide little support for dealing with crosscutting concerns.

In this thesis we present two prototype tools that support each of these two activities. The JQuery tool combines the advantages of hierarchical code browsers with those of query tools to better support the exploration of crosscutting concerns. The Decal tool lets developers edit their code using virtual source files that support two mutually crosscutting points of view.

To see why such tools might be useful consider the kinds of tools currently available to developers.
One kind of tool that is ubiquitous in current IDEs is the hierarchical browser. Hierarchical browsers come in many flavours, such as package browsers, call graphs browsers, class hierarchy browsers, etc. These browsers display hierarchical structure using tree views where each element in the tree represents a program element, such as a field, method or class. Users can expand a node in the tree to reveal all the other elements that are connected to that node by some kind of relationship, such as inheritance or method invocation. By expanding nodes users can reveal the elements they are interested in and see how they relate to each other by looking at their positions in the tree.

The main limitation of these views is that they represent only one kind of relationship between program elements (or in some cases a small number of different kinds of relationships between program elements). To browse across some other kind of relationship one has to switch to another browser and in doing so the relationship between the collection of elements being explored is lost. The path taken from an element visited early in the exploration to an element visited late in the exploration is not maintained because the browsers are not able to represent all the various relationships needed to connect the two elements. As the developer backtracks and tries different paths, he or she must maintain the history of the exploration in his or her head. This may not be difficult for small tasks, but becomes problematic when one considers that developers may be performing several different tasks simultaneously and over long periods of time.

Query tools provide an alternate means of exploration. Most IDEs include simple text search, or grep-like tools. More advanced query tools include specialized query languages that work over program databases. Using these tools developers can construct complex queries that bring together elements using a wide variety
of relationships that would not be possible with a hierarchical browser. The results of these queries can be used to explore code by visually representing them as hyperlinked structures with links to the corresponding source code.

However, it is often not possible to write a query that captures everything a developer wants to know with regards to a particular evolution task. The typical pattern of usage for query tools is to continually refine a query or to write a series of related queries that help the developer as he or she slowly comes to understand the task at hand. As with hierarchical browsers, the problem here is that the exploration path is not maintained from one query to the next.

Another limitation of both hierarchical browsers and query tools is that when it comes time to edit the source code they are only useful as navigational aids. The views they provide are not effective in the sense that they do not allow one to effect changes to the structure of the program by manipulating them. They will typically provide a means to jump to the source code for elements represented in their views, but they do not allow an evolution task to be carried out in the context of the structure that the tools provide. Editing must always be performed in the context of the object-oriented structure in which the source files are kept. The editors themselves provide many tools that make use of the structure of the program to help with renaming and moving elements as well as more sophisticated refactorings, but the structure of the text presented in the editors is restricted to the single decomposition that exists in the source files. To edit a collection of elements that crosscuts this structure one must open all the files that contain these elements and then jump back and forth between them to make the necessary changes.
1.3 The JQuery Prototype

To address the limitations of current exploration tools as discussed in the previous section we developed the JQuery prototype, a query-based code browser. JQuery combines the advantages of hierarchical browsers and query tools by providing a generic mechanism for constructing tree-view browsers from queries and by allowing users to incrementally extend these browsers using additional queries. At the core of JQuery is a logic database of program information that is kept synchronized with the source files for a program. Users can type in queries, or select from a set of predefined queries, which JQuery will execute and then construct a tree-view browser from the query results. This functionality was developed in an earlier prototype called QJBrowser [37]. JQuery extends QJBrowser by also letting users incrementally extend their view. With JQuery the initial browser is just the starting point for exploration. When a user finds an element of interest and wants to continue exploring based on a relationship that is not represented in the current view, he or she can select the node and create a subtree that includes the desired relationship. For example if the user is browsing the inheritance hierarchy of a code base and wants to extend the view with class creation information he or she can select a class element and create a subtree of all the methods that create an instance of that class. In this way the user can incrementally extend the view to explore crosscutting structure.

JQuery offers several advantages over traditional browsing and querying tools. With JQuery, developers are not limited to the set of browsers built in to the IDE they are using, but can create their own browsers tailored to the specific application or task at hand. By incrementally extending these browsers developers can explore relationships that are not easily encoded in a single query. By explic-
itly maintaining the developer’s exploration path. JQuery eases the cognitive burden of trying to remember how all the elements in their exploration are connected to one another. With JQuery, users can perform complex explorations involving both querying and hierarchical browsing within a single view, thus reducing the disorientation caused by switching back and forth between multiple views.

1.4 The Decal Prototype

To address the limitations of current editing tools, we developed the Decal prototype that lets developers edit their code from two mutually crosscutting points of view using virtual source files. Like JQuery, Decal is built on top of a program database, but instead of generating browsers, it generates editable text that represents two different points of views on the single underlying program structure. The two views in Decal are called the modules view and the classes view. Each view is similar to the single decomposition of a program that one might find in a traditional programming language. However, Decal allows developers to edit both views simultaneously and edits made to one view will be reflected in the other view.

The classes view in Decal is similar to the decomposition of programs into classes that one would find in a typical object-oriented system. Each virtual source file (VSF) in the classes view contains all the declarations related to a particular class. The modules view is similar to the more operational decomposition that one would find in a language that supports open classes. Each VSF in the modules view may contain declarations related to multiple classes. The modules view allows for declarations to be organized according to more subjective criteria as defined by the user. The classes view crosscuts the modules view in a similar way that the modules view crosscuts the classes view. Both views contain the same program
declarations, just organized in different ways. Furthermore, both views are effective, in that editing the text of a VSF will cause the underlying program structure to be modified. The role of each view will be explained in more detail in Chapter 3.

1.5 Research Contributions

The main contribution of this thesis is to present the design and implementation of two prototype tools that extend the state of the art in exploration and editing of object-oriented code. Specifically, this thesis shows a way to design and implement:

- a source code browser which combines the advantages of a hierarchical browser with the flexibility of a query tool.
- a programming system that supports the simultaneous editing of two mutually crosscutting views.

We claim that the design of JQuery improves on that of similar tools by providing an explicit, unbroken representation of the exploration path taken by developers, thus allowing them to remain oriented within their task. Furthermore, JQuery makes it possible to complete a complex exploration task within a single view, thus reducing disorientation caused by switching between multiple views.

This thesis presents a case study that uses JQuery to perform a realistic development task. A series of examples are presented that show how JQuery was used to complete small subtasks from within a single view. The case study seems to indicate that the typical usage pattern for JQuery is to start with relatively simple queries and then incrementally expand the view until the target of the search is found. The case study also highlighted some possible improvements that could be made to the tool.

8
Some additional contributions related to the Decal prototype are that we identify a number of issues and problems that are specific to a system that supports crosscutting effective views. We believe that reporting on these issues and discussing the particular approach used in Decal will be a valuable contribution to the design and implementation of similar systems in the future.

In summary, JQuery and Decal provide better tool support to developers for working with crosscutting structure in source code than exists in current IDEs. JQuery supports the exploration of crosscutting structure through the use of a query-based code browser. Decal supports the editing of crosscutting structure through the use of virtual source files. One of the assumptions underlying this work is that these two tools could be combined to produce a seamless exploration and editing experience that provides better support for working with crosscutting structure in legacy object-oriented code than is currently available.

1.6 Overview

In this chapter we present the main motivation for this work and outline the approach taken. In Chapter 2 we introduce the JQuery prototype and describe how it supports exploration of crosscutting structure. Chapter 3 describes the Decal prototype and how it supports editing of crosscutting structure. Chapter 4 provides some concluding remarks.
Chapter 2

Exploring Crosscutting Concerns

In this thesis we focus on the exploration and editing activities of program evolution. This chapter presents the prototype tool, JQuery, for exploring crosscutting structure in Java source code. The text of this chapter represents a large portion of a previously published paper [27] with only minor changes made to adapt it to the context of this document.

2.1 Introduction

Consider the scenario of a software developer wanting to reuse part of a particular application's code base because it contains functionality she needs in another application she is developing. The developer will need to track down the potentially scattered pieces of code that constitute the desired functionality and refactor the code to bring them together into one or more modules. This can be a challenging task because not only are the parts of the code she is trying to identify scattered
across several modules, they are also tangled up with each other and with the rest of the code through many different types of relationships.

In a good development environment developers may have at their disposal a wide variety of exploration, visualization and navigation tools that may assist them in this task. Established integrated development environments today may provide tools such as a package browser, a class hierarchy browser, a call graph browser (e.g. [3]) and a variety of different search engines. Research prototypes may go even further and provide powerful specialized query languages (e.g. [9, 45, 13, 14]) and sophisticated visualization tools (e.g. [34, 21, 41]).

In this chapter we focus on how to combine the advantages of hierarchical code browsers and query tools, in terms of how they help developers with exploration, by providing a powerful means to navigate the source code.

A hierarchical browser is a tool that supports navigation based on particular kinds of relationships. For example, a class hierarchy browser allows navigation along inheritance relationships whereas a call-graph browser allows navigation along static calling dependencies. The typical browser's interface is a tree view with collapsible and expandable nodes. When a node is expanded, subnodes reveal other elements that are connected to it through some specific relationship. The advantage of hierarchical browsers is that they provide an explicit map of the navigation paths. Also, the history of the user's exploration is captured in the collection of nodes that were expanded.

Unfortunately, these browsers are specialized and limit exploration to particular types of relationships. Consequently, when developers want to navigate the code across different kinds of relationships they are forced to switch between different browsers. Switching between tools is disorienting by itself, and also has the
disadvantage that there no longer is an unbroken representation of the exploration path. Instead, the path is divided into fragments spread across multiple disconnected views. As a result developers lose track of their current position with respect to the exploration task.

Compared to typical browsers, tools based on query languages and program databases (e.g. [9, 45, 13, 14]) provide more flexibility in terms of the relationships that can be explored with them. Developers can construct queries using complex combinations of relationships. Queries allow the extraction of useful information and can also be used for the purpose of constructing source code navigation views. For example, the results of a query can be turned into a navigational aid by visually representing it as a hyper-linked structure, with links to the corresponding places in the code that match the query. In general, it is not possible to formulate a single query that finds everything the user is interested in pertaining to a task. Consequently, exploration using a query tool usually follows a pattern of writing a query, browsing the results, writing another query, analyzing more results, and so on. A drawback of this approach is—once more—that the exploration path connecting the queries gets lost along the way.

This chapter presents JQuery, a prototype code browsing tool. JQuery is a browser tool implemented on top of an expressive logic query language. The main contribution of this chapter is that it shows a way to design and implement a source code browser which combines the advantages of a hierarchical browser with the flexibility of a query tool. Specifically, our design succeeds in combining the following advantages into a single tool:

- Like any hierarchical browser, our tool provides an explicit representation of the exploration paths followed by the developer.
• Like a query tool, it supports directed searches for specific subsets of elements of a code base according to some criteria specified by a query.

• Like a query tool, it supports exploration in terms of a broad range of relationships between code units.

Compared to other browsers and query based tools, we claim that our design results in a tool that reduces the cognitive burden associated with exploration in the following ways:

1. It provides an explicit, unbroken, representation of the exploration paths. This helps a developer to retain a sense of orientation within the context of an exploration task.

2. The flexibility of the tool to explore a broad range of different relationships and queries within a single integrated view greatly reduces the need to switch between tools and views. Therefore the disorientation caused by switching views is greatly reduced.

JQuery extends an earlier prototype, QJBrowser, which was discussed in a previous paper [37]. QJBrowser constructs a navigation tree from a single logic query and a list of query variables. We have shown how this method provides enough flexibility to define many different kinds of useful hierarchical browsers. A view can be based on a user-specified query, performing a directed search. Alternatively it can be based on a predefined query, resulting in a generic type of navigation tree. For example, it is possible to provide predefined queries that yield views similar to a typical package browser, a class hierarchy browser or a view organizing code units based on specific JavaDoc tags attached to them (e.g., a browser based on Author tags).
The novelty in JQuery is that the initial tree only serves as a starting point for the exploration process. To support continued exploration, a JQuery tree can be incrementally refined by the developer. At each node in the tree the developer may wish to explore further and may choose to extend the current view with a new subtree. The subtree shows the results of a selected query that finds code units connected to the selected unit through some relationship of interest. It is this feature of JQuery that provides the additional flexibility required to avoid switching views and scattering an exploration path across multiple disconnected views.

2.2 Illustrating Example

In this section we present an example that illustrates how JQuery would be used for a typical exploration task. Our example is a fictional scenario in which a developer is exploring the JHotDraw [1] code base. Like most drawing applications, JHotDraw lets users draw a variety of figures: rectangles, circles, lines, etc. Suppose the developer wants to add a feature that operates on figures and therefore, she would like to find out how figures are implemented and to find an example of a class that manipulates figures.

Figure 2.1 shows a screenshot of JQuery at the end of the exploration task. We now explain step by step how the developer reached this situation.

To start using JQuery a developer can type a query or choose a specific browser to find starting points for her exploration task. In this example the developer has chosen to start with a generic package browser which groups Java classes and interfaces according to the packages in which they are declared. She navigates down into the CH.ifa.draw.figures package and there she discovers a number of classes with names that correspond to different types of figures. Assuming there is a
Er

common base type for figures she decides to examine the supertypes of ElbowHandle. In JQuery, the browser's view can be extended to reveal relationships not yet shown. Right-clicking on a node brings up a contextual menu of node-specific relationships. Our developer right-clicks on ElbowHandle and selects “supertypes”.

Often, a developer will be interested in seeing (and perhaps editing) the source code associated with a node in the tree. In JQuery, double clicking on a node brings the corresponding source code into view in the editor pane. Our developer uses this functionality to inspect the source code of the supertypes of ElbowHandle and concludes that they do not appear to be what she is looking for. She then retraces her steps and tries listing the supertypes of EllipseFigure instead. Among these she finds an interface called Figure which is what she was looking for.

Having found the Figure interface, the developer wants to know what operations can be performed on Figures, so she expands the view with the list of methods it defines. Finally, to find examples of classes that use Figures she decides to list all the places in the code that make calls to addFigureChangeListener().

This simple scenario illustrates how an exploration task may involve following several different types of relationships back and forth between elements of a code base. In this example, the developer navigated along relationships induced by declaration nesting, subtyping and method calls. This example illustrates how the developer was able to complete the task without the disorientation of switching between tools or views. The JQuery view also reduces the cognitive burden placed on the developer by helping her to remain oriented by showing how previously explored elements relate to the current element.
2.3 The JQuery Tool

In the preceding section an example illustrated a typical usage scenario. In the following subsections we provide some additional details about the functionality and usage of the tool.

2.3.1 Getting Started

The starting point for exploration is a browser whose view is defined by a query and a list of variables. The JQuery user interface allows a developer to either type this query directly or select a predefined one.

The query determines what elements to display in the browser and the list of variables determines how to organize them in a tree. This method of defining hierarchical views was introduced in the QJBrowser prototype and its utility was discussed in detail in a previous paper [37]. This method allows for the definition of many different kinds of useful general-purpose browsers. It can also be used for performing directed searches that produce views with a small number of elements of particular interest.

For a user unfamiliar with the query language or unwilling to make the effort to compose a complex query, a few predefined generic browsers are provided as presaved queries. These can be selected by simply clicking on one of the tabs at the top of the JQuery view.

2.3.2 The Query Language

JQuery is built on top of TyRuBa [16, 15], an expressive logic programming language. TyRuBa is similar to Prolog [17]. The expressive power of the logic programming language provides the flexibility to express complex queries and to use
rules to define higher-level relationships. We assume basic familiarity with Prolog and do not explain the details of the TyRuBa syntax here. JQuery’s query language is basically TyRuBa augmented with a library of predefined predicates that allow querying for code units and the various relationships between them.

Table 2.1 lists a sample of the predefined predicates in the query language. There are several predicates that go beyond the basic elements and relationships that exist in a Java program. The method(?M, tag, ?Tag, ?Value) predicate retrieves the value of JavaDoc tags attached to method declarations. The error() predicates provide access to the location and severity of compilation errors. To find dependencies at the class level there is the refType(?Ref, ?Caller, ?Callee) predicate that finds references to all fields and methods contained in a particular type.

The predicates in the query language follow the convention that predicate names correspond to the type of an object and the parameters correspond to, respectively, an object reference, an attribute name or relationship name, and a value. For example the following query finds all classes ?C who’s name property is HelloWorld.

class(?C, name, HelloWorld)

Note that TyRuBa has non-standard lexical conventions for the denotation of variables and constants. In TyRuBa, symbols starting with a “?” are variables. This is convenient because Java identifiers denoting class, field and method names can be used as constants.

The following example shows a more complex query. When used with the variable list ?P, ?C, ?I, ?CM this query produces a browser that groups all the methods in a class by the interface to which they belong.
JQuery is implemented as an Eclipse [25] plug-in and its query language is integrated with the Eclipse development environment in such a way that all objects displayed in a JQuery tree view are automatically "hyperlinked" to the code. When a node is double-clicked the corresponding source code is brought into view in the Eclipse source code editor pane. Specifically, in the above example, the variable $CM$ would become bound to an object that represents a hyperlink to the actual line of code where the corresponding method is declared.

### 2.3.3 Contextual Menu Structure

The contextual menu associated with a particular node is specific to that node and contains a list of all the ways in which the tree can be extended at that node. The structure of the menus and the corresponding queries can be fully configured by an expert user. JQuery comes with a default configuration file that provides access to a number of useful relations between code units. The default configuration is summarized in Table 2.2. In the implementation section (Section 2.5) we will
<table>
<thead>
<tr>
<th>Predicate</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>package(?P)</td>
<td>True if ?P is a package.</td>
</tr>
<tr>
<td>package(?P, name, ?N)</td>
<td>True if package ?P has name ?N.</td>
</tr>
<tr>
<td>type(?T)</td>
<td>True if ?T is a type.</td>
</tr>
<tr>
<td>type(?T, name, ?N)</td>
<td>True if type ?T has name ?N.</td>
</tr>
<tr>
<td>type(?T, field, ?F)</td>
<td>True if type ?T contains field ?F.</td>
</tr>
<tr>
<td>type(?T, method, ?M)</td>
<td>True if type ?T contains method ?M.</td>
</tr>
<tr>
<td>type(?T1, type, ?T2)</td>
<td>True if type ?T1 contains inner type ?T2.</td>
</tr>
<tr>
<td>type(?T, modifiers, ?M)</td>
<td>True if type ?T has modifiers ?M, where ?M is a list.</td>
</tr>
<tr>
<td>type(?T1, super, ?T2)</td>
<td>True if type ?T1 has super type ?T2.</td>
</tr>
<tr>
<td>type(?T, tag, ?Tag, ?Val)</td>
<td>True if type ?T has a JavaDoc tag ?Tag with value ?Val</td>
</tr>
<tr>
<td>class(?C, creator, ?M)</td>
<td>True if an instance of class ?C is created in method ?M.</td>
</tr>
<tr>
<td>method(?M, returnType, ?RT)</td>
<td>True if method ?M has return type ?RT.</td>
</tr>
<tr>
<td>method(?M, paramType, ?PT)</td>
<td>True if method ?M has a parameter of type ?PT.</td>
</tr>
<tr>
<td>method(?M, exception, ?ET)</td>
<td>True if method ?M throws an exception of type ?ET.</td>
</tr>
<tr>
<td>error(?E, message, ?M)</td>
<td>True if error ?E is described by message ?M.</td>
</tr>
<tr>
<td>error(?E, severity, ?S)</td>
<td>True if error ?E has severity ?S.</td>
</tr>
</tbody>
</table>

Table 2.1: Some predefined predicates in the query language.

discuss how the menu structure can be configured by an expert user.

### 2.4 Case Study

In order to assess JQuery's usefulness we conducted a simple case study using JQuery to perform a realistic development task.
<table>
<thead>
<tr>
<th>Node Type</th>
<th>Relationships</th>
</tr>
</thead>
<tbody>
<tr>
<td>Packages</td>
<td>Classes</td>
</tr>
<tr>
<td></td>
<td>Interfaces</td>
</tr>
<tr>
<td></td>
<td>All Types</td>
</tr>
<tr>
<td>Classes</td>
<td>Methods</td>
</tr>
<tr>
<td></td>
<td>Fields</td>
</tr>
<tr>
<td></td>
<td>Subtypes</td>
</tr>
<tr>
<td></td>
<td>Supertypes</td>
</tr>
<tr>
<td></td>
<td>Imports</td>
</tr>
<tr>
<td></td>
<td>Constructors</td>
</tr>
<tr>
<td></td>
<td>Creators</td>
</tr>
<tr>
<td></td>
<td>References Methods</td>
</tr>
<tr>
<td></td>
<td>References Types</td>
</tr>
<tr>
<td></td>
<td>Calls to this type</td>
</tr>
<tr>
<td>Methods</td>
<td>References Methods</td>
</tr>
<tr>
<td></td>
<td>References Types</td>
</tr>
<tr>
<td></td>
<td>References Fields</td>
</tr>
<tr>
<td></td>
<td>Calls to this method</td>
</tr>
<tr>
<td></td>
<td>Signature</td>
</tr>
<tr>
<td>References</td>
<td>Caller</td>
</tr>
<tr>
<td></td>
<td>Callee</td>
</tr>
<tr>
<td></td>
<td>Caller's callers</td>
</tr>
<tr>
<td></td>
<td>Callee's callees</td>
</tr>
</tbody>
</table>

Table 2.2: A sample of the ways in which nodes in the tree can be extended.
2.4.1 The Task

The task we chose was to extract the user interface code from a chess program called Jin [2] and put it into our own application as the front end for a simple AI module. Jin is a client for the Internet Chess Club (ICC), a server that allows people to play chess against each other through the Internet. Jin contains no AI code, but contains a lot of extra code that handles features of the ICC, such as chatting with other users and searching for players on the Internet.

Although most of the code we were interested in was contained in a single package — a fact we only discovered during the experiment — there were numerous dependencies on the rest of the code. There was no clearly defined interface for using the code in the context of another application.

The Jin code base was small enough to make a good first study, yet large enough that the task would be difficult to complete without the aid of a tool. Jin has 151 classes containing 1207 methods for a total of 24,482 commented lines of code.

We recorded how we actually used the tool by taking screen shots of the browser and taking note of the queries we used as the task progressed.

2.4.2 Summary of Task Progress

Our initial approach was to use JQuery to separate the code into two parts: the part we wanted to keep and the part we wanted to throw away. We started by searching for pieces of code that could be immediately put into either of those two categories. These early searches involved two main techniques. First we explored using standard types of browsers as a starting point. For example by browsing a packages view we were able to identify several packages that had to do with handling
communication with the Chess Club server. Second, we used more directed queries to search for particular elements of the code. For example we searched for methods that took parameters of type Image. Since the rest of the application made little use of graphics we were able to hone in on the part that had to do with drawing the board.

After finding some pieces of code that we knew could be deleted or kept, we tagged these pieces of code using a custom JavaDoc tag. By using JQuery to find elements with specific tags, we were able to bring those pieces of code together into a single convenient view. We used this view to explore the relationship of the tagged pieces of code with the rest of the code. Our intent was to continually expand the amount of code that was tagged until we had tagged everything.

However we soon found that while some pieces of code were easy to classify others were not. Some pieces of code had dependencies on both the parts we wanted to delete and the parts we wanted to keep. At this point we abandoned our method of tagging code and started making some modifications. We started using JQuery to help us understand pieces of code that we weren’t sure what to do with and to help resolve errors after making changes. Our use of JQuery now involved searching for very specific code elements, such as a single class or method, and then extending the view using combinations of call graph, class membership and inheritance relationships.

After making a series of relatively small modifications we were able to plug the Jin code into our own application by writing a class that implemented both our own UI interface and a single Jin interface. The final use of JQuery was to help with cleaning up the Jin code to remove as much unnecessary code as possible. In the end we were able to eliminate 65% of the Jin code.
2.4.3 Case-study Examples

Before moving on to drawing some more general conclusions from the study, we start by presenting some concrete examples that illustrate some of the advantages of using JQuery. The examples in this section are actual situations we encountered while we were performing the case-study task. The screenshots shown in this section were adapted from the collection of screenshots taken as a means of recording our progress in the case-study task.

Example 1

The first example is one of our earliest searches, the corresponding screenshot is shown in Figure 2.2. At the start of this search, we knew nothing about the code and we wanted to find out how the GUI was constructed. Figure 2.2 is an example of using a directed query to find a specific starting point in the code from which to explore. In this case the query found all the main methods. We were initially interested in finding out where all the windows and menus were created so we started exploring the call graph. We soon found a class called JInFrame, listed all its methods and started looking at the code. Exploring in and around the JInFrame class gave us some understanding of how the visual elements of the application fit together.

To accomplish the same thing using standard tools we would have had to use a search or query tool, a call graph browser and a class browser. Switching between the three different tools introduces additional overhead and is disorienting. For example, to explore the call graph from the main method, we would have had to open a call graph browser and then navigate to the main method before we could find out what methods were called by main. To find out what methods JInFrame
has we would then have had to open a class browser and navigate to JinFrame.

With JQuery we could concentrate on moving through the code without the
distraction and disorientation of having to open new windows and explicitly initialize
each view with the context of our previous view.

Example 2

While using the Jin application we noticed that there was very little use of graphics
outside the board window. This observation led us to query for methods that took
Image objects as a parameter. Figure 2.3 is a screenshot that was taken part way
through our exploration of the results of this query.

This example shows one way in which the tree-view of a search can be useful.
Each query result represents the starting point for exploring the code. JQuery
allowed us to explore each result in as much depth as we wanted without losing
track of which results we had explored and which ones we hadn’t. To do this using
other tools we would have had to keep the original results open in one window,
while exploring individual results in other windows. JQuery kept track of the initial search results and all our exploration of them in a single view.

Example 3

In our next example, we wanted to focus more specifically on the code to be extracted. We had already identified the BoardManager class as containing most of the functionality we were interested in. We needed to find out how to properly use this class.

Figure 2.4 shows how we discovered the key to controlling the Jin chess board. The BoardManager communicates with the JinConnection interface using an event system. To respond to user moves and display computer moves we needed to create a class that implemented JinConnection.
What is interesting about this example is that the relationship between the BoardManager class and the JinConnection interface was explicitly captured by JQuery during the exploration process. While looking at the source code for JinConnection we could see that the reason we were interested in it was that one of the classes that implemented it generates a GameStartEvent for the BoardManager.

Maintaining this context throughout the exploration process was important because it relieved us of the need to remember how the piece of source code we were looking at fit into the larger exploration task. We did not have to worry about getting sidetracked because we could always refer back to the JQuery window to reorient ourselves.
2.4.4 Case-study Conclusions

This study is of a preliminary nature and its main goal was to get a quick assessment of whether or not the ideas behind the JQuery prototype are sound. Overall, the preliminary results were positive and indicate that the ideas behind the tool’s design are indeed sound.

The task we chose was relatively complex and involved a series of many small subtasks. As such we did not expect to be able to complete the entire task without ever switching views. We expected that JQuery would allow us to perform an exploration subtask mostly without switching views. In general, this expectation was confirmed. For example, one technique we used was to delete a section of code as soon as we were sure we did not want to retain it for our own application. Then we would analyze the resulting compilation errors and try to resolve them. Resolving the errors generally involved two choices: either to add code to satisfy the dependency or to delete the code containing the error. In order to make this decision we typically started a JQuery view to explore and understand the code surrounding the error. We were typically able to complete this exploration subtask without switching views.

The typical usage pattern was that we started by writing a fairly simple query and then explored deeper by incrementally extending the view and inspecting source code. Although the query language is very expressive and can support complex queries, our experience indicates that typically writing “the” query that finds directly what we are looking for is either impossible or impractical. In our experience this was because we usually did not know exactly what we were looking for until we found it. Even if we did have a more precise idea of what we were looking for, expressing an accurate, often complex query to find it is usually too
hard to be practical. Consequently, it was more cost effective to start from a fairly simple query and repeatedly extend our view until the target was found. Example 1 shows a typical scenario in this respect.

Usually the exploration process involved switching back and forth between reading source code and extending the view. It is thus essential that JQuery provided an easy way to access the source code from the tree view.

Summarizing, we found JQuery was useful in the completion of our task and supported us in the following ways:

- By repeatedly expanding a view, we could construct chains of steps across a heterogeneous set of relationship types.
- We could easily get access to the source code from the tree view.
- We could visually retrace our steps in the tree. This made it easier to understand our current position in relationship to previously visited elements.
- Occasionally we wanted to backtrack and start exploring in another direction. The tree-view naturally supported that.

While our overall experience with the tool was positive, there were some issues that were perceived as clearly hampering the tool's effectiveness.

One problem is the simple graphical representation of the exploration paths. When the tree is expanded many levels deep, it tends to become too wide and too cluttered to fit in the JQuery pane. To get an overview it is necessary to scroll the view horizontally as well as vertically. This is cumbersome and makes it harder to understand the connection between elements separated by many levels in the tree. Note that, while this strongly suggests that our method of visualizing the exploration
history can be improved, the connection between elements at great distance from each other in the tree would be orders of magnitude harder to reconstruct in a typical IDE where there would be no representation of the exploration paths at all.

While the ability to perform directed searches with the query language was useful (e.g., see the example in Section 2.4.3), the logic query language was hard to use for complex queries. This is true even for developers reasonably familiar with the query language. We noticed that in practice we tended to formulate only fairly simple queries. Subsequent incremental view expansion led to the actual targets of a search. This suggests that either the query language is too hard to use, or there is no real need for end-users to perform complex queries.

Related to the issue of what queries are used to find starting points for exploration, is the question of what kind of navigation steps are most practically useful. Using JQuery, we occasionally came into a situation where a step we wanted to take was not provided in the menus, thus creating a "blockage" on the navigation path and forcing us to make a detour. Such detours break-up the intended path and cause a certain amount of disorientation. We could argue that these problems could be avoided by reconfiguring the tool, adding the missing queries. However, it is not clear whether adding "missing links" systematically will eventually converge to a manageable set of intuitive navigation steps.

Another problem—of a more technical nature—concerns integration of the tool with the rest of Eclipse. The tool would benefit from a tighter integration with the editor. For example, it would be useful to be able to follow hyperlinks in the editor and have these navigation steps be automatically reflected in the JQuery view. Currently only navigation steps taken from within the JQuery view itself are recorded in the tree. Consequently we had to accept the minor nuisance of having
to force ourselves to make all navigation steps via the JQuery tree in order to avoid losing navigation history.

2.5 Implementation of JQuery

JQuery consists of 25 classes and 4621 lines of non-blank lines of code, not including the code for the query engine. It is implemented in Java as a plug-in to the Eclipse Platform [25], an open source IDE. Facts in the JQuery database are generated dynamically by making calls to the underlying Eclipse API. This has the advantage that queries can be run on code immediately after changing it. There is no need to perform an intermediate step of generating a database from source code as Eclipse performs this step automatically. Furthermore, Eclipse comes with an excellent incremental compiler that is very tolerant of errors. Contrary to many other systems based on querying of static analysis information, running queries against a code base which contains compilation errors does not pose a problem in JQuery.

A disadvantage of using the Eclipse API to generate facts dynamically is that it increases the running time of queries. Retrieving call graph information from the Eclipse API is an expensive operation. Queries that process large amounts of call graph information can easily take several minutes to run. Also, the query engine cannot take advantage of its normal indexing mechanism to speed up search times. In a new version of JQuery we will investigate the effect of storing some facts in the logic database and keeping them synchronized with changes to the code, rather than generating the facts dynamically with every query execution.

A limitation of our current implementation is that query results are not updated automatically when the source code changes. Queries must be rerun in order to refresh the display. A production version of JQuery could make use of an
incremental update algorithm, such as [22], to improve the usability of the tool.

2.5.1 Customizing JQuery

The JQuery tool was designed to support as much flexibility as possible in the different types of relationships between code units that can be explored with it. Making JQuery flexible in this respect seemed especially important because it is precisely the rigidity of conventional browsers that forces a user to switch between browsers. To avoid this problem, JQuery has been made configurable so that it is easy to extend the relationships supported by the tool. An expert developer can add additional queries to the contextual menus that support the incremental expansion of a view at any given node.

This is how the configuration mechanism works: when a developer right clicks on a node, the tool launches a query to determine what relationships apply to that node and what menu items should be shown in the menu. Thus, the structure of the menu can be configured by providing a configuration file containing logic rules that match these queries. To be able to extend the menu structure, a user needs to be familiar with 1) the logic query/programming language embedded in JQuery, and 2) how to structure the logic rules that define the menu items. The query language was already described in Section 2.3.2. We briefly describe the procedure for adding a menu item below.

Adding a Relation-Query Menu Item

All nodes in a JQuery tree view are generated from the results returned by queries to the underlying query engine. The JQuery menus can be customized by constructing queries and adding them to the database in the form of special rules.
When a user selects a node the JQuery tool determines the structure of the context menu by running a query `menuItem` to determine what menu items are associated with that node. Thus, adding an item to the context menu can be accomplished by adding an appropriate `menuItem` rule to a configuration file. The `menuItem` rule has three parameters: the selected node, the label to be displayed in the menu, and the name of the rule that implements the corresponding query. Typically, the body of the rule checks the selected node to see if the corresponding query is applicable to it. The second and third parameter of a matching rule are used to create a menu item for the contextual menu.

For example, let's say we want to add a menu item that lists all the listeners of an event generating class. The `menuItem` rule only applies when the selected node is a class that has some kind of `addListener()` method.

```
menuItem(?O, "Listeners", menuFindListeners) :-
    class(?O, method, ?M),
    // Finds all the methods ?M of class ?O.
    // Fails if ?O is not a class.
    method(?M, name, ?N),
    // Finds the name ?N of method ?M
    match(?N, /add.*Listener/).
    // Matches the name ?N to a regular expression
```

Here, the second parameter, Listeners, is the label that will appear in the context menu and the third parameter, `menuFindListeners`, is the name of a rule that we must still write. It will perform the query whose results will appear below the selected node.

The next step is to implement the `menuFindListeners` rule. This rule must
have two parameters: one that gets bound to the selected node, and one that is a list containing the results of the query. This list represents a single path in the resulting subtree. The actual subtree is constructed by merging all the paths that are returned by the query. The use of a list here provides JQuery with a fixed interface — a two parameter predicate — while still allowing queries to return paths of arbitrary length. This facilitates the construction of subtrees of arbitrary depth and structure.

menuFindListeners(?O, [?C]) :-
    class(?O, method, ?M),
    // Finds all the methods ?M of class ?O.
    // Fails if ?O is not a class.
    method(?M, name, ?N),
    // Finds the name ?N of method ?M.
    match(?N, /add.*Listener/),
    // Matches the name ?N to a regular expression.
    refMethod(?Ref, ?Caller, ?M),
    // Finds all the methods that call ?M.
    class(?C, method, ?Caller).
    // Finds the classes to which those methods belong.

This rule is similar to the menuItem rule, but adds some extra terms to find references to the addListener() methods of the selected class.

The configuration mechanism described above is important because it would be impossible to anticipate all possible relationships that a developer might be interested in. Customized menu items can make use of highly specific coding conventions and design patterns, enabling developers to explore their code using more high-level relationships induced by them.
As can be seen from our example, the configuration mechanism is relatively complex and we do not expect end-users to configure the tool. However, if sufficiently motivated, expert users could provide libraries of rules that encapsulate various coding conventions and design-pattern-specific relationships and views. For example, if a tool like JQuery were to become widely available, it would be conceivable that library vendors would provide a set of library-specific rules.

2.6 Related Work

JQuery does not directly support working with crosscutting concerns, but rather supports navigation and exploration of code in a general way. We believe this kind of support is useful in general, but is particularly useful in the context of crosscutting concerns, because crosscutting concerns imply a need to explore a complex and tangled web of relationships between scattered elements of a code base. In the remainder of this section we discuss how JQuery relates to other tools for supporting exploration of code, regardless of whether or not they claim explicit support for working with crosscutting concerns.

JQuery derives much of its flexibility and functionality from the expressive power of the underlying query engine. The idea of using structural queries — in a logic language or another sufficiently powerful query language — as a basis for constructing software development tools is not new. Some examples of other systems based on structural source code querying are SOUL [45], ASTLog [14], GraphLog [13], Coven [10] and Stellation [11]. SOUL is a logic query language integrated with the Smalltalk development environment. ASTLog is a logic query language for querying C++ abstract syntax trees. GraphLog is a logic based graphical query language in which both queries and query results are represented as Graphs. Coven
and Stellation are software configuration management tools, equipped with an SQL-like query language for the retrieval of software units. In all these tools, software queries can be used by developers in the process of exploring code. However, these tools do not typically provide explicit support for exploration in terms of chains of related queries. A notable exception is the Ciao [9] system which we will discuss separately.

There are numerous tools (e.g. Rigi [34], SHriMP [41], Ciao [9], SVT [21] and GraphLog [13]) that provide different ways to visualize the structure of a software system. Some of these tools were already mentioned as query-based tools and we don’t discuss them again. Rigi [34] is a reverse engineering tool that starts by generating complex graph views from the original source code. It then provides tools to iteratively refine these views into higher level representations of the subsystems. SVT [21] is a configurable software visualization framework that relies on Prolog as a configuration language.

All of these tools help in understanding and exploring software systems, but generally they tend to focus on the visualization of the structure of the software. To this end they may provide very sophisticated graphical views and user interfaces. In comparison, the hierarchical views provided by JQuery are relatively primitive. However JQuery is different from most software visualization tools in its emphasis on providing a representation of the structure of an exploration process rather than the software. For example, consider the SHriMP tool. It also strives to help a developer to remain oriented. SHriMP offers different ways to organize and navigate source code and can create sophisticated graphical views of the system. However, these views do not capture the history of an exploration process. SHriMP helps a developer remain oriented by providing context information in terms of one's
location within the graph, but not in terms of the path taken to navigate to that location. Thus, it is possible to see where you are but not how or why you got there.

The Ciao [9] system is interesting because it provides some kind of representation of the exploration history in the form of a "navigation graph". Each node in the navigation graph corresponds to a query that generates a specific view on the system. The edges of the navigation graph represent historic dependencies between query views. However, the nodes in the navigation graph only show the type of query that was run and the corresponding graph is shown in a separate window. To reconstruct the structural relationships that connect different queries on a path, one must compare their corresponding views.

The FEAT tool [38] supports working with crosscutting concerns by letting developers incrementally build-up an explicit representation of a concern. A crosscutting concern in FEAT is represented by a set of scattered code-units identified by a developer as being part of the implementation of that concern. FEAT allows developers to browse the elements of a given concern, and to incrementally add additional units of code to the concern. To facilitate this process of building up a concern representation FEAT offers queries to find code units that have incoming or outgoing structural dependencies on elements already in the concern. JQuery and FEAT are largely complementary. JQuery focuses on supporting exploration of code by representing the history of the exploration process, whereas FEAT focuses on representing crosscutting concerns in code. There is some overlap in the functionality of both tools. FEAT provides some support for the exploration process needed to build-up a concern representation, but it does not present the user with an explicit representation of the exploration paths. JQuery on the other hand does not provide explicit support for capturing the representation of a concern. Developers
may use custom JavaDoc tags as an ad-hoc representation of FEAT-like concerns.

2.7 Conclusion

In this chapter we presented the JQuery prototype. JQuery intends to enhance a developer’s ability to perform tasks involving crosscutting concerns by providing better support for carrying out exploration tasks involving a complex web of relationships between scattered elements of a code base.

JQuery’s design goal is to combine the advantages of query based tools and hierarchical browser tools. Specifically, from query tools we want to retain the ability to perform directed searches, and the flexibility to explore code in terms of many different kinds of relationships. The hierarchical browser interface on the other hand provides an explicit representation of the exploration history in terms of exploration paths and queries performed.

Using the JQuery tool a developer can start an exploration process with a query. The result of the query is used to define an initial browser view that serves as a starting point for an exploration process. The developer may navigate the tree and extend it at will by requesting additional queries to be added as subtrees of specific nodes of interest. In so doing, the shape of the tree provides an explicit representation of the history of the exploration process in terms exploration paths and queries performed.

We claim that our design reduces the cognitive burden associated with an exploration process, by helping a developer to remain oriented. Our tool reduces the need to switch between different views. This avoids the disorientation caused by switching views and keeps an unbroken representation of the whole exploration path.
A case study was performed to test the soundness of our design. Overall, the study confirms that JQuery's design is sound. We found that we could complete many subtasks involving chains of exploration steps and some backtracking without switching views. The representation of the exploration history was found to be helpful in keeping one's orientation while performing an exploration task.
In this thesis we focus on the exploration and editing activities of program evolution. This chapter presents the prototype tool, Decal, for editing crosscutting structure using virtual source files. The text of this chapter represents a large portion of a previously published paper [28] with only minor changes made to adapt it to the context of this document.

3.1 Introduction

Aspect-oriented software development [30] addresses the issue of crosscutting concerns and how they can be more cleanly modularized and dealt with by developers. There are many different approaches towards this goal. What all the approaches have in common is that they attempt to provide explicit representations of crosscutting structure in software.

Language based approaches provide programming language extensions such as open classes, aspects, pointcuts and advice that allow concerns which cut across classes to be captured modularly. Some examples of this are systems like As-
pectJ [29], Caesar [33], and Aspectual Collaborations [31].

Tool-based approaches on the other hand make crosscutting structure explicit by constructing views on top of the code. In this way tools can make crosscutting structure which is implicit in the code explicitly visible and accessible to the developer. Examples of tool-based approaches are FEAT [38], AJDT [4], JQuery [27], AspectBrowser [44] and Stellation [11]. These tools can be loosely divided into two camps. On the one hand there are tools which work on legacy OO systems and try to show how implicit crosscutting concerns exist within the object-oriented program (examples: FEAT, JQuery, AspectBrowser, Stellation). On the other hand there are tools which try to do exactly the opposite: they try to produce views that recover the object-oriented structure of the program which has been made implicit by the introduction of aspect-oriented features in the language (example: AJDT for AspectJ).

In both cases the fact that one of the views is explicit in the code and the other views are generated by tools implies a different level of support for working with those views. The view which has been made explicit in the code can be edited directly by editing the code. We call such a view an effective view. For example, for all programming languages which are based on textual syntax and the use of source files to store programs, the source code of a program represents an effective view, because editing the source code directly affects program structure. On the other hand, the views produced by most tools are non-effective because typically these views cannot be edited in such a way that the actual program structure is affected by the edits to the view. Instead, the typical usage profile for tool-based crosscutting views is that they serve as documentation or navigational aids for developers. But in order to make effective changes to program structure developers must edit the
actual source code.

In this chapter we present the Decal prototype. Decal is a tool that lets developers work simultaneously with two mutually crosscutting and effective source code views. Decal is atypical in that the views it produces are not derived from source code, they are source code. In some sense, Decal reverses the roles of tool-based views and source code as found in most tools. A typical tool produces a view based on source code. Decal maintains an internal, structured representation of the program and, from this, views are generated in the form of source code.

The current prototype is limited to two crosscutting views only, but this could easily be extended. In section 3.7 we will discuss some possible extensions. For now we limit ourselves to a brief introduction to the two views supported by the current prototype.

The first of the two views is called the modules view. This view provides a decomposition of program structure in terms of modular units which crosscut classes in a way that is similar to open classes. The second view is called the classes view and is a more traditional object-oriented view, showing a decomposition of the program structure in terms of classes. The classes view crosscuts the modules view in a similar way that the modules view crosscuts the classes view. Every element in the program belongs simultaneously to both the classes view as well as the modules view. Both the modules view and the classes view are effective and editing the textual representation of either view impacts program structure. Consequently, when changes are made to one view, these changes are also automatically reflected in the other view. Figure 3.1 depicts this mutually crosscutting relationship between the two views.

To accomplish this bi-directional causal connection between the two views,
Figure 3.1: The modules view crosscuts classes and the classes view crosscuts modules.

Decal views are represented as *virtual source files* (VSFs). The term "virtual source file" was introduced in Stellation [11]. VSFs are dynamically generated ASCII files which can be browsed and edited by the developer with commonly available text editors. To be able to generate the VSFs dynamically and have changes from one view be reflected in the other view, Decal internally maintains a single common representation of the program structure. After editing a VSF, the developer can commit the file back to Decal. The system will then translate the changes the developer made into corresponding changes to the common representation. At this point, the VSF is "absorbed" back into the system and ceases to exist. This extract-edit-absorb cycle is depicted in Figure 3.2.

Both the modules view and the classes view in Decal are, on their own, similar to conventional decompositions of program structure into source files. The classes view provides an effective, textual view similar to that of traditional object-oriented source files. The modules view provides an effective, textual view similar to that of a language that supports open classes. The contribution of this chapter is to show that these mutually crosscutting effective views can be supported simultaneously.

An additional contribution this chapter makes is to identify a number of
issues and problems that are specific to a system that supports crosscutting effective views. We believe that the insights gained from the design and implementation of Decal, and the ways in which we have addressed them are valuable for the future design and implementation of similar systems.

The rest of this chapter is structured as follows. In the following section, we present a motivating example. This example illustrates why it is desirable to provide an effective view for classes as well as modules at the same time. Section 3.3 presents the Decal system and its two different views. Section 3.4 describes the Decal database and issues surrounding its design. Section 3.5 looks at issues that arise when editing virtual source files. Section 3.6 describes the implementation of Decal. Section 3.7 discusses how the ideas explored in our prototype might be applied to a more realistic language. The last three sections describe future and related work and end with some concluding remarks.

3.2 Motivating Example

The example presented here is derived from a similar example in Tarr et. al. [42]. What the example tries to illustrate is that no matter how well designed a pro-
Figure 3.3: Most natural OO decomposition for a simple AST implementation

Program may be and no matter how carefully developers may have considered the decomposition of their program to anticipate future evolution, the choice of any decomposition makes an implicit tradeoff, making certain tasks easier at the expense of making other tasks harder.

The example is phrased in the context of an implementation for an environment for working with programmatic expressions. The example is strongly simplified for presentation purposes. The class diagram in Figure 3.3 depicts a simplified version of the subsystem for representing expression abstract syntax trees. It depicts the most natural way to represent an AST in terms of a hierarchy of classes. This is also most naturally mapped onto an implementation in an object-oriented language such as Java, mapping every “class box” in the diagram to a compilation unit in the implementation language.

This natural object-oriented decomposition makes it hard to add new fea-
tures to the system that require the implementation of new operations on AST structures. For example, suppose we wanted to extend our implementation with a pretty-printing feature. This would require the addition of one or more printing related methods to most, if not all, of the AST classes. Good designers may have anticipated the need for such extensions and included a Visitor pattern [18] in their design. The Visitor pattern comes at a price however because it introduces additional complexity into the program. In many ways the Visitor pattern is just a way of reifying a more procedural abstraction as an object/class, so that it can be represented with the available decomposition mechanism of classes.

Some languages, such as MultiJava [12] and AspectJ [29] provide the concept of open classes which leads to more flexibility in this regard. With open classes it is possible to declare methods outside of the class that they “belong to”. Thus a developer may simply add additional operations on AST structures in a separate module and need not modify the original source code of the AST classes. With open classes, one may achieve a decomposition which has a modular structure similar to that of the Visitor pattern, but without the complications of implementing procedural abstractions in terms of closed classes. This situation is depicted in Figure 3.4.

Unfortunately, even the extra flexibility of open classes ultimately is not sufficient. When developers choose to structure modules around certain operations such as printing, type checking, etc. they are gaining more ease to maintain these types of abstractions, but at the same time they inevitably make it harder to do other types of extensions because the implementation of classes is scattered across multiple modules. This makes it harder to perform tasks that more naturally align with the class structure of the program. For example, suppose a future extension requires
adding a new type of expression node to the AST. This task will be complicated because it will require making additions to several modules to implement all the required functionality for the new class, such as printing, type checking, etc.

Decal addresses this problem by letting developers alternate between editing the program in the modules view or the classes view. In the following sections we will explain each view in terms of the example from this section. At the end of the section we will show how the classes view eases the extensions of the AST implementation with new classes, while at the same time program structure is decomposed in terms of modules that crosscut those classes.

3.3 The Decal System

At the core of the Decal system is a simple object-oriented language with support for open classes. To keep the prototype lightweight and suitable for quick exploratory experiments we wanted to keep the language as simple as possible. The Decal lan-
guage therefore supports only a minimal subset of object-oriented features. Specifically, it supports single inheritance with the ability to override operations, but does not support advanced features such as interfaces, overloading, static members and constructors\(^1\).

To this basic object-oriented core language, Decal adds support for a simple form of open classes (but does not support multiple dispatch as in MultiJava). We consider open classes the simplest "flavor" of aspect-oriented programming \(^2\). Our current prototype does not support more advanced aspect-oriented language features such as pointcuts and advice.

The addition of more advanced object-oriented features would make the design and implementation of several components of the Decal system technically harder, but we do not believe it should pose fundamental problems. The situation is not as clear-cut for adding additional aspect-oriented features such as support for pointcuts and advice. In Section 3.7 we discuss how the choices made in designing and simplifying the Decal language may affect the generalizability of our approach both in terms of object-oriented as well as aspect-oriented features.

3.3.1 The Modules View

The modules view in Decal allows programs to be edited according to a modular structure that may crosscut classes. Declarations related to a particular class may be spread across multiple modules, and each module may contain declarations related

\(^1\) We assume reliance on ordinary methods and programming conventions for initializing objects as is also the case in, for example, Smalltalk\(^{[20]}\).

\(^2\) Some people might argue that open classes belong in the camp of traditional object-oriented languages because open classes do not include a mechanism of implicit invocation. What set of features makes a language aspect-oriented (or object-oriented for that matter) can of course be debated. Our view is that they are a simple form of aspect-orientation because they explicitly provide a mechanism to support a form of crosscutting modular structure.
module ast {

    public class Expression {
    }

    public class Number extends Expression {
        public int value;
        public void initialize(int value) {
            this.value = value;
        }
    }

    public class BinaryOperator extends Expression {
        public Expression left;
        public Expression right;
        public String op;
        public void initialize(Expression left, String op, Expression right) {
            this.left = left;
            this.op = op;
            this.right = right;
        }
    }

    public class Plus extends BinaryOperator {
    }

    public class Minus extends BinaryOperator {
    }

}

Figure 3.5: Example VSF: The ast module

to more than one class. Figure 3.5 shows the ast module from our running example. It contains the “bare bones” declaration of several classes to represent a simple AST structure.

Decal modules are a means to group declarations related to a specific concern. Each declaration can be marked public to indicate that it is visible outside the module or private to indicate that it is visible only within the same module.

Besides declaring classes of its own, a module may also introduce additional
module printing {

    import ast;

    Expression {
        public void print(Printer out);
    }

    Number {
        print {:
            out.printInt(value);
        :}
    }

    BinaryOperator {
        print {:
            left.print(out);
            out.printString(op);
            right.print(out);
        :}
    }
}

Figure 3.6: Example VSF: The printing Module

declarations into classes that are publicly defined in other modules. Figure 3.6 shows a printing module that provides an additional printing-related operation for the classes defined in the ast module. As with open classes, using the modules view in Decal allows programmers to work with concerns that crosscut the class structure in a modular way. All code related to the printing concern are brought together in the printing module. Similarly, it is possible to add operations related to semantic checking functionality to all the AST classes in a separate module VSF as well (this is not shown to save space).

Decal is similar to Java minus a number of features such as interfaces, static members, overloading and constructors. Besides these simplifications, there are a few other differences that are worth mentioning.
First, Decal makes a distinction between "operations" and "methods". We will have more to say on the reason for this distinction when we discuss program representation in Section 3.4. For now we merely explain the nature of the difference.

An operation represents an operational abstraction. For example, the printing module provides a declaration for an operation called print on Expression. A method declaration on the other hand provides a specific implementation for an operation on a specific class. A method declaration provides the method body that is to be executed in the event of an invocation of the operation on some specific type of object. Therefore, generally an operation is implemented by one or more method declarations. Methods are dispatched at run time according to the standard single dispatch semantics of overriding and inheritance.

In our example the print operation is implemented by two methods, for the Number and BinaryOperator classes respectively. Note that an operation declaration is where the signature of the operation resides, but that it is not repeated in the method declaration which only requires the name of the operation to identify it unambiguously (since there is no overloading in Decal). An alternative VSF syntax might redundantly repeat the operation signature with the method declaration, for the sake of readability. However, in this version, the syntax was designed to resemble the database representation as closely as possible and redundant information in the syntax is factored out, just as it is in the database representation.

3.3.2 The Classes View

The classes view in Decal contains the same declarations that are present in the modules view except that they are organized according to class membership rather than module membership. A VSF can be generated for each class in the system.
public class ast.Number extends ast.Expression {

    module ast {
        public int value;
        public void initialize(int value) {
            this.value = value;
        }
    }

    module printing {
        print {
            out.println(value);
        }
    }

    module checking {
        check {
            ...
        }
    }

    module evaluating {
        eval {
            ...
        }
    }
}

Figure 3.7: Example VSF: The Number class

Each VSF contains all the declarations related to the chosen class grouped in blocks according to which module they are declared in. Figure 3.7 shows what the VSF looks like for the Number class from our example.

Note that in conventional languages the boundaries of information hiding are typically defined in terms of lexical position. In Decal, where every declaration occurs simultaneously in two textual locations, a class VSF and a module VSF, this alignment of scope with textual nesting only really happens in the modules view. This is because we can think of the modules view as the “primary” view, which is most like the only textual view one would get if Decal was a conventional
programming system. The classes view is conceived of as a secondary, derived view which is generated from the primary view. However, because both views are mutually effective this distinction between primary and secondary view is not all that clear. Scoping rules however, is one way in which there is still a clear qualitative difference between the nature of the two views. Classes, unlike modules, do not define a name space in Decal and do not have import statements. The names referred to in the code in the class VSF in Figure 3.7 therefore are to be interpreted in relation to the modules they belong to and to their respective import statements. For example, the print method and the value field are defined in the same class. However this alone is not sufficient to make a reference like this.value legal. It is further required that the name value can be resolved in terms of the import statements of the printing module (as it was shown in Figure 3.6). Similarly, it would be legal for other modules to declare additional fields called value on the Number class. These fields would be considered as distinct. This is especially useful for private fields, providing modules with a safe way to add additional state to classes without having to worry about accidental name conflicts with extensions defined by other modules on the same class.

3.3.3 Motivating Example Revisited

Now let us return to the motivating example again. The explanations in the preceding sections already show how the modules view allows extending classes with new functionality, for example to modularize behavior such as printing, or checking across multiple classes. Let us now examine how the classes view aids in extending the system with an additional AST class, even when the system has been decomposed around behavior oriented modules as shown above.
For example, assume we needed to add a new class `UnaryOperator`. This can be conveniently accomplished in the classes view. Indeed, it can be done in a way very similar to what we might do if the program was written in a style similar to the most natural class-based decomposition shown in Figure 3.3. What we can do is — inspired by the assumption that `UnaryOperator` is most similar to `BinaryOperator` — extract the `BinaryOperator` class VSF and use it as a template. We create a new VSF for `UnaryOperator` and copy-paste the contents of the `BinaryOperator` class into the new VSF. Then we edit it at will into what we need. This works very well, because the `BinaryOperator` class will already have all the right pieces of functionality from various crosscutting modules in place, telling us exactly what functionality we also need to implement for a `UnaryOperator` and providing us with some convenient code snippets to base our implementation on. The task of adding a new AST class would be significantly harder in the modules view. This is true even in a fancy development environment that provides some additional non-effective tool-based views that show a view recovering the class structure. This is so because although a non-effective class view will help by telling us what we need to do, and where we need to do it, we would still need to add snippets of code in multiple different modules across the system. In Decal we get spared from this because we can directly edit the classes view and even meaningfully copy-paste code from one class VSF into another class VSF.

We can extend this example a little further. Assume that after considering the copy-pasting solution, we do not like the amount of code duplication that was introduced as a result. So we want to refactor this code and introduce a common superclass to capture the similarity between unary and binary operator expressions. This refactoring task is also more easily carried out on the class-based decomposition.
We can do the refactoring by working with only three class VSFs, the class VSF for the new class, plus VSFs for the two classes to be refactored. It would be significantly harder to do this refactoring in the modules view because the functionality to be factored out into the superclass is scattered across many modules.

3.4 The Decal Database

To support multiple crosscutting effective views Decal uses a database as a single common representation of the program, from which all views are generated on-demand. We have chosen an RDBMS over other alternatives, not for any particularly significant reason, except perhaps that it is the most standard and commonly available type of database. In this section we provide some information about the design of the Decal program database.

3.4.1 Developing a Schema

The database schema of Decal is basically a fairly straightforward mapping of the structure of Decal's modules view into database tables. Some aspects of the schema design are non-obvious and, we believe, key to making Decal work. We refrain from discussing all the details of the schema but provide some information here about some key issues and design decisions.

Granularity

The core object-oriented structure of the database is made up of tables to represent classes, fields, operations, methods and the relationships between them. A key choice that we made in designing this part of the schema was to store method bodies as blocks of text, rather than explicitly representing every statement and expression
as a separate fact. What is necessary for generating Decal VSFs is the relationship between high-level declarations, not the details of the AST structure of method bodies. A further motivation for this choice was the experience of the OMEGA project [32] which reported serious performance problems due to the number of queries required to generate the text of even simple programs.

Object IDs

The database representation makes an explicit distinction between the identity of an entity (module, class, operation, method, etc.) and its name. This is important because it is possible that several distinct entities exist that have the same name. For example multiple classes called TraversalState might be defined as helper classes in different modules for printing, checking etc. in the AST example. For convenience and efficiency (to avoid performing name lookup each time), references to program entities in database tables are represented in terms of some globally defined non-ambiguous object IDs (OIDs).

Soft keys

Another key feature of the schema design is the notion of "soft keys". In order to support incremental name resolution and error tolerance, an extra level of indirection was introduced to represent references that use names. For such references, rather than storing a direct link to an OID, a so called "soft-key" is stored. A soft key points to a table entry that stores what information is needed to resolve the reference and caches the OID of its target once it becomes resolved.

Soft keys provide a form of intentional name capture. Even if a reference can be resolved immediately, the target of the reference may later be deleted or
renamed. By retaining the name used to resolve a reference, Decal can ensure that the reference can be resolved and unresolved many times as the program evolves while retaining, to some degree, the original intention of the programmer.

3.4.2 Impact of Schema Design on the Language

It is interesting to note that the design of Decal's database schema has had an impact on the language design. In particular, in the process of designing the schema for the representation of method declarations we realized that information about method signatures would be represented redundantly. Standard normalization practices in database design suggest that this information should be factored out into an additional table. Refactoring the database tables in this fashion, we also decided to make the language itself reify the notion of an operation explicitly. Though we don't think this is essential to make our approach work, it leads to a cleaner correspondence between the database schema and the syntax of the language. As a result, the implementation of generation/absorption of VSFs from/to the database is more straightforward.

3.5 A Practical Editing System

In this section we will look at Decal's editing system. When we say editing system we mean more than just a tool for editing the text of a VSF. The editing system includes everything required to support the extract-edit-absorb cycle. The editing system of Decal, which is characterized by a complete separation between the editing format and storage format of programs, introduces some new issues that do not arise in traditional programming systems, where the editing format and the storage format are one and the same. In the following subsections we elaborate on some of those
issues and how we have addressed them in Decal.

3.5.1 Editing Semantics

In a conventional programming system, where programs are stored as text files, and where developers can edit the stored program representation directly, the meaning of edits is unambiguously defined. However, when the text being edited is part of a virtual source file the meaning of the edits may not be as clear.

An important design decision that we made is that declarations can belong to only one module. For a brief moment we deliberated about supporting overlapping modules, so that one piece of information (e.g., a method declaration) would be represented simultaneously in multiple modules. We found this idea very appealing, because it would make it possible to support modules that not only crosscut classes, but also modules that crosscut each other. However, this raised the issue that the editing semantics of module VSFs are no longer intuitively unambiguous. For example, if a declaration can belong to more than one module VSF at the same time then it is not clear when a developer deletes a declaration if the intent is to delete the declaration from this module only, or from the system as a whole. Assuming that declarations belong to only one module naturally gives VSFs an editing semantics very similar to regular source files, i.e., it is intuitively clear that deleting declarations must remove them from the program altogether.

We call the property that a piece of information is represented in at most one VSF within a particular view, the disjointness property. It is a desirable property because it allows an editing semantics of VSFs which is intuitive, in that it is designed to behave similarly to what would be expected of "real" source files. Note that the converse property, completeness — that every piece of information is repre-
sented in at least one VSF — is not as significant in defining the semantics of edits. It does however affect the usefulness of a view since it determines what information can be manipulated through it. This is not to say that incomplete views (views which do not contain every piece of information in a program) are necessarily less useful (they are more abstract in a sense).

In the case of ambiguity, it is of course possible to explicitly assign a semantics one way or another, or to design a mechanism to let developers explicitly state their intent. However it should be clear that the design of an editing semantics that would feel intuitive to developers becomes significantly more difficult in the absence of the disjointness property. Note that a similar problem of confusion around editing semantics does not arise because of overlapping VSFs that are in different views. The fact that the semantics of an edit is defined in one view automatically defines what it means in the other view as well.

We believe that studying mechanisms to deal with ambiguity in editing semantics is an interesting and important topic for future research. Indeed, resolving this issue is a necessary condition for supporting a more complex set of aspect-oriented language features, as will be discussed in Section 3.7.2. However we considered it outside the scope of this thesis.

### 3.5.2 Name Resolution

The generation of multiple views requires that the database provide the necessary connections to determine what VSFs (module or class) a specific element belongs to. The availability of that information in the database is dependent on the successful resolution of names. For example, in Figure 3.6 the identifier "Number" refers to the class `Number` as defined in the `ast` module. To generate the class VSF for class
the system must resolve all references to number from all the modules that add members to that class. This dependence on name resolution implies that in a system that supports multiple views, name resolution needs to be done as early as possible. Therefore, in Decal name resolution takes place each time a changed VSF is saved, rather than at compile time as is usual in traditional languages.

3.5.3 Error Tolerance

An additional complication arises because an incremental name resolution mechanism, as described above, needs to be tolerant of errors and incompleteness. During the process of constructing a program developers often leave parts of the program in an inconsistent state. This is not due to bad programming, but is simply a consequence of the fact that programmers can only do one thing at a time. It is often convenient for a programmer to work on one part of a program and refer to elements that he or she intends to define later on. Or sometimes a developer may wish to delete several parts of a program that contain references to each other. Thus it is highly impractical to force all names to be resolved before a VSF can be saved.

As mentioned earlier, we use a representation for name references that we call a soft key. Soft keys retain all the information needed to resolve a reference. This includes the name used to make the reference and the module from which the reference is being made. If the reference can be resolved then the OID of the target is also stored. As elements are added to the database, unresolved soft keys are checked to see if they refer to the new element. When an element is deleted, soft keys that point to the element have their OID removed, but retain all the information needed to resolve the reference again as new elements are created. The use of soft keys gives Decal VSFs a similar kind of flexibility that regular source files have while at
the same time allowing the kinds of queries that depend on direct cross-referencing information.

Another place where tolerance of errors becomes an issue is when checking the syntax of VSFs at the time they are saved. It would be impractical to expect developers to remove all syntax errors just to be able to save their work. However, some degree of syntactic correctness is necessary for Decal to be able to map regions of text back to the database. To minimize the impact of this requirement Decal does not force method bodies to be syntactically correct before they are saved. This can be easily supported thanks to our choice to store method bodies as blocks of text. Also, we have designed Decal’s method declaration to have its body delimiters easy to recognize, by adding an additional colon next to the outer braces. This makes it easy to identify the textual area of the body without having to parse what is inside them.

3.5.4 Temporal Continuity of Views

In a conventional programming system, it is trivially true that a file saved at one time, when revisited at some later time, will look identical. However, in a system based on VSFs, this property does not automatically hold. The exact layout of a VSF depends on how it is generated. Since it is regenerated on each successive visit, the VSF layout may differ each time. We call this the issue of “temporal continuity of views”.

The fact that the contents of a VSF may change between visits is a necessary property of the system, because it is what allows changes in one view to be reflected onto the other view as well. However, changes in a VSF between visits also have the potential for causing disorientation. This is because there is typically a lot of
freedom in the use of indentation, whitespace and the order of declarations. This freedom is actively used by developers. It is therefore desirable that VSFs should look and feel as much as possible like "real" files, preserving not just the semantic content, but also non-semantically meaningful attributes of their textual layout.

In the current implementation of Decal we did not spend a great deal of effort on this issue. Our choice of representation for method bodies ensures that their formatting is retained in the database exactly as the programmer typed them, including comments. We also provide limited support for preserving comments outside of method bodies. Each element in Decal can have a Javadoc style comment attached to it that is stored with the element in the database. Other kinds of comments, such as single line comments, are not allowed outside method bodies as it is not always possible to determine which element they should be attached to in the database.

The order of declarations and white space between declarations are not currently retained. Instead we settled for maintaining a consistent use of white space and ordering (alphabetic). This guarantees some continuity: there will not be huge discrepancies in layout unless substantial edits are performed. Nevertheless, this implementation is clearly far from optimal since it takes away most of the freedom that developers have in conventional text-based editing systems in constructively using the layout of the source text.

Even in the limited setting where we have used Decal for toy examples, we found the "jumping text" behavior (on saves) to be disorienting. Therefore we believe that temporal view continuity is important for practical usability. We also believe it is technically feasible to make VSFs feel nearly the same as real files, by storing more information about their layout in the database. A good solution is
not trivial to implement and requires detailed consideration about what additional information to store, but we believe it is mostly a technical implementation issue.

3.6 A Prototype Implementation

The user interface for our prototype is implemented as an Eclipse [25] plugin. Users can browse the contents of the database using two simple views that show the inheritance hierarchy of classes and the list of modules. Selecting a class or module from one of these views opens the corresponding VSF in the editor.

At the core of Decal is an API for manipulating Decal programs. This API uses a relational database called HSQLDB [24] to actually store the program information. HSQLDB is a lightweight embeddable database that can be distributed as part of Decal thus eliminating the need for users to install and maintain a full database management system. Its use of JDBC would make it relatively easy to switch to a more industrial strength database in the future.

The generation of virtual source files is built as a layer on top of the core API. This layer generates text and maps changes back to the database independently of how the text is edited. This makes it easier to integrate Decal into a variety of programming environments.

The prototype is still in an experimental stage and is not yet available to the general public. We are currently developing a third view whose VSFs correspond to proper Java source code suitable for compilation. Since Decal's name resolution mechanism is incompatible with Java packages all VSFs in this view are written to a single directory and a name mangling scheme is used to avoid name collisions. Thus it is not a VSF that can be edited and saved back to the database, but will allow us to compile and run programs written in Decal with a standard Java compiler.
Similarly we have done only a small amount of exploratory work on type checking Decal programs. Although type checking such a system is not a trivial issue, we did not think the implementation of the type checker would be a substantial contribution compared to other languages that support open classes, such as MultiJava and AspectJ. For this reason we decided to defer this work and focus on crosscutting effective views.

3.7 Supporting More Realistic Languages

The Decal prototype shows how multiple views can be implemented on a simplified language, but to be of practical value, our methods must be applicable to more realistic languages. Adding support for standard OO features would be a good first step, but the real power of multiple views is in their application to aspect-oriented languages.

3.7.1 Adding More OO Features to Decal

To extend Decal to include more advanced OO features such as interfaces, static members, constructors, overloading etc, the first thing that must be done is to design a more extensive database schema to represent the additional features. This is mostly a technical issue not a fundamental one.

Second, the additional features make the design of the Decal module system more complex. MultiJava has already proven that it is possible to support open classes in a clean way in conjunction with a rich set of core object-oriented language features, adding both open classes and multi-methods to Java in a way that is backwards compatible. Thus, the design of the module system is technically complex, but doesn’t present any unsolvable problems.
Third, we have to consider potential complications that might arise with respect to editing semantics. Assuming that we design the language from the point of view of the modules view and consider the classes view as secondary, then the editing semantics of the modules view is automatically clearly defined. We also do not foresee any real complications with editing semantics of the generated classes view because adding conventional OO features only introduces new or more elaborate declarations that can be associated with a specific location in the class structure of the program. Therefore, they would not introduce information overlap within the classes view. As discussed before, the disjointness of different VSFs within a view implies that editing semantics can be unambiguously and intuitively defined to mimic the semantics of real source files.

3.7.2 Adding More Aspect-Oriented Features

Besides considering the extension of Decal to support a more complete set of conventional object-oriented features, we may also wish to consider extending its arsenal of features to express crosscutting in the language. For example, we may wish to include mechanisms for pointcuts and advice instead of just the simpler notion of open classes.

Presumably, the modules view would be augmented to show the system from an aspect-oriented point of view. Whereas the classes view would show where the aspects apply in terms of the class-based decomposition.

We believe this will pose more fundamental challenges than the addition of object-oriented features because the expressiveness of pointcuts and advice make the connection between the two views fundamentally more complicated. Intuitively, open classes provide only a very simple form of crosscutting, in the sense that any
declaration in the modules view can be interpreted as applying to a single location in the classes view. This property, which guarantees an easy translation between the two views, is destroyed by the introduction of an expressive pointcut language. Pointcuts allow the application of advise to a large number of locations in the classes view, or even dynamically, in terms of properties that cannot necessarily be associated directly with static locations at all. This raises some issues with the generation of the classes view, such as how to represent dynamic advice. Additionally, independent of its dynamic nature, the fact that advice may apply to many locations in terms of the classes view destroys the disjointness property of the classes view: a single advice body may occur in multiple places in the classes view. This implies that we will have to tackle some complications with the editing semantics: for example, what does it mean to edit, add or delete advice from within the classes view?

3.8 Related Work

The main contribution of this chapter is to show how it is possible to construct an editing system that lets a developer alternate between editing a system through either one of two mutually crosscutting, effective views. This work is most closely related to tool-based aspect-oriented approaches that support the creation of crosscutting views on top of source code. Decal distinguishes itself from most of these other tools in that it produces effective textual views. In contrast, most other tools provide views which are non-effective. In such tools the main utility of the view is to serve as documentation and to support navigation. Some examples of tools which fit this description are AspectBrowser [44], JQuery [27], FEAT [38] and AJDT [4]. Of these tools, AJDT is the only one that is in the same camp as Decal, generating
views that help developers to recover object-oriented structure which has been made implicit by the introduction of more aspect-like modularity in the language.

While the majority of tools provide only non-effective views, there are some notable exceptions. The oldest one is probably MasterScope in the Interlisp [43] development environment. In MasterScope, a developer may request the generation of a textual view by writing a query that identifies what declarations in the program are of interest. The declarations which match the query are returned in a textual, editable view and can then be edited as a group. A similar mechanism was provided more recently by the Stellation [11] system from which we adopted the idea of VSFs. In both cases, the views are textual and effective. The main difference is that in Decal, views are first class and have a well-defined intuitive semantics for all possible edit operations, whereas in Stellation and Interlisp, the views are arbitrary groupings of declarations and the editing system does not attach a specific semantics to the grouping of elements into a view. This has consequences for the semantics of additions and deletions of declarations to/from views which — in Stellation or MasterScope — don’t have a clearly defined and intuitive editing semantics. Our example provided in Section 3.3.3 illustrates the importance of this difference. The example relied on copying and pasting declarations between different class VSFs in order to effectively move or copy them from one class to another. This requires that insertion and deletion of declarations has the appropriate semantics with respect to the classes view in which these operations take place. Consequently, as far as the authors understand, this example would not work in either Stellation or MasterScope.

The Smalltalk Envy [35] programming environment is similar to Decal in a number of ways. The role of source code in Smalltalk differs from most other pro-
gramming systems and is similar to Decal in that program structure is not stored as source code but in a more structured format, as a coarse grained object-oriented data structure. Smalltalk Envy also has a mechanism for packaging of applications, similar to Decal modules. The main difference is that Envy does not provide textual editable views of classes or application packages as a whole. Instead, views are created by, and manipulated through different GUI source code browsers. These browsers provide some level of effectiveness by providing refactoring and restructuring commands in menus. Decal on the other hand preserves the “illusion” of source files and lets developers edit source file views of classes or modules as a whole. We believe the GUI browser approach has advantages as well as disadvantages over a VSF-based editing metaphor. Moreover, they are complementary, in the sense that GUI browsers can be added to an editing environment based around VSFs, just as many modern IDEs provide browsers for real source files. Some advantages of GUI based refactoring and restructuring commands is that they allow developers to express their intent more clearly. This is one way of resolving ambiguity in editing semantics. Nevertheless, we believe it is hard to design and implement a “complete” set of GUI tools to support the refactoring-type of editing operations. Textual manipulation of source code seems to be still what developers always need to revert to when specific refactoring operations are not supported directly in the GUI. This is also true for Envy. Although the lower-level text editing operations do not capture the intent of the developer directly, all edits can be accomplished by copying, pasting and editing text. Moreover, editing programs in terms of source files is what the majority of developers are already most familiar with.

Intentional Programming (IP) [39], like Decal also uses a rich data structure to store programs and provides different kinds of views to edit this structure. How-
ever IP views can be graphical as well as textual. Furthermore, its main purpose is to facilitate language extensions of various kinds, not the creation of crosscutting views (though this might be a potential application of IP). The complexity of coordinating the interactions between several language extensions and visual editors make IP a much more ambitious project than ours. The Decal approach consequently is more lightweight. It uses a much coarser grained representation of programs, and requires only limited support from some fairly simple and cheap tools: a standard SQL engine, standard text editors and some simple VSF parsers and VSF code generators.

HyperJ [42] and its notion of multi-dimensional separation of concerns are closely related. In HyperJ’s terminology, Decal could be characterized as “a system that supports two orthogonal dimensions of program decomposition simultaneously”. Decal modules are similar to HyperSlices. The main differences are the following. First of all, Decal modules are true modules and have true support for encapsulation and information hiding (public and private), whereas HyperSlices have no mechanisms to hide internal structure\textsuperscript{3}. Second, HyperJ focuses more on the composition of HyperSlices rather than the generation of views from existing program structure. To this end HyperJ provides a very complex mechanism of composition rules. In contrast, the composition rules for modules in Decal are embodied by a straightforward mechanism of name binding.

The concept of mixin layers [40] proposed by Smaragdkis and Batory are similar to Decal modules. The mixin layers approach focusses primarily on the complexity of composition of families of systems whereas our focus is not on the

\textsuperscript{3}HyperJ supports public and private to be used in HyperSlices, but these relate to the class structure and have no bearing on whether something is visible or not to another HyperSlice composed with it.
composition of modules into multiple systems but rather on supporting the editing of a single system simultaneously from multiple points of view. It is noteworthy that in a recent paper [6] Batory et. al. report they implemented and used an “un mixin” tool that allows a system composed from mixin layers to be edited directly, and the edits to be mapped back into the corresponding mixin layers. They report that the ability to edit from both a composed and uncomposed point of view was tremendously useful. They do not however provide any details or insights about issues such as editing semantics, temporal continuity of views, etc.

We used MultiJava’s open classes and AspectJ’s inter-type declarations as models for designing our module system. Both languages support more advanced kinds of modularity than just open classes. MultiJava includes support for multi-methods and AspectJ supports pointcuts and advice. We have already discussed in Section 3.7 what the possible implications of adding such features to Decal are.

The Decal extract-edit absorb cycle depicted in Figure 3.2 bears a remarkable similarity to the idea of roundtrip engineering as supported by many CASE tools [8, 26]. The main difference, as we see it, between Decal and such roundtrip engineering tools is that Decal transforms between (textual) representations that are both at the same level of abstraction – implementation – whereas CASE tools transform between representations at different levels of abstraction, for example connecting design to implementation or architecture to design.

In his work on trying to define a theoretical framework for Automated Roundtrip Engineering (ARE) [5], Assmann has proposed a theoretical definition of an ARE system and a classification of AOP systems in the context of roundtrip engineering systems. In Assmann’s terminology, Decal is not an ARE system, i.e., it does not provide a means to automatically generate an unweaver from a weaver.
However, the Decal editing system’s extract and absorb functions could be interpreted as hand crafted instances of what Assmann calls a bi-directional weaver, or “Beaver”. Although Assmann provides good arguments why such bi-directionality is desirable, he does not provide a clear insight on how it can be achieved in practice.

Decal has some similarities with integrated environments as proposed by Garlan [19] and Herrmann and Mizini [23] where a common data structure is shared by a collection of tools. In these systems each tool is given its own view of the common data structure and is also able to keep its own data structures that are unique to its operation. Such integrated environments could be used to produce a variety of effective views, including something like Decal. However their focus is on the problems of tool coordination and integration, and they do not address the specific problems of working with virtual source files such as editing semantics.

3.9 Conclusion

We have presented the Decal system, an unconventional programming system which provides two mutually crosscutting, effective, textual views. One view, called the modules view lets developers decompose and edit program structure in a way similar to open classes. The second view, called the classes view, presents a more traditional object-oriented decomposition of the system into classes. Developers can alternate arbitrarily between the two views. Both views are presented as a collection of virtual source files (VSFs) and can be edited by developers to effect changes to the system. Both the modules view and the classes view in Decal are, on their own, similar to conventional decompositions of program structure into source files. The main contribution of this chapter is to show that both views can be supported simultaneously, as effective views on the same program.
The design and implementation of a system which simultaneously supports several textual, crosscutting, effective views poses some unique challenges. Little or no work exists in the literature on how such systems should be designed and implemented. Thus, another contribution this chapter makes is to identify some of the issues that arise in the design and implementation of such a system, and to describe specific principles and techniques we have used to address those issues.

Below, we provide a brief overview of the most important issues that were touched upon in different parts of the chapter, and summarize what has been done in the context of Decal to address those issues.

**Program Representation** A system that supports multiple views must perform early name resolution, be tolerant of errors, and have clearly defined editing semantics. Some form of a database is necessary to allow for efficient querying of program information. The Decal database uses a representation that is tolerant of unresolved references while at the same time allows efficient querying.

**Editing Semantics** In a system where the storage format and the editing format of programs are separated, it becomes an issue to define the meaning of edits in terms of changes to the stored representation of the program. In Decal this issue is addressed by respecting the disjointness property which makes it possible to define editing semantics that mimick those of "real" source files.

**Temporal Continuity of Views** In a system where source files are generated on the fly the layout of the generated text can change between editing sessions, resulting in a sense of disorientation for the developer. We believe it is technically feasible to design VSF generators that produce stable views by adding explicit information to the database about the textual structure of VSF files.
at the moment they are saved. We have not implemented this in the current
version of Decal but leave it as a topic for future research.

**Incrementality and Tolerance of Errors** To be able to generate one view from
another a minimum of information about the structure of the program needs
to be updated incrementally as developers edit VSFs. To support the typical
editing patterns of developers this process must support a certain amount of
error tolerance. When saving a VSF Decal tolerates syntax errors in method
bodies by parsing and storing them as single tokens. Decal is also tolerant
of unresolved names and supports incremental name resolution through the
mechanism of soft keys in its database representation.

While the issues and solutions presented here stem only from the experience
in designing and implementing a single prototype, and therefore do not necessarily
generalize to all systems, we believe that our experience recorded in this chapter will
prove a valuable starting point for the design of other systems that support multiple
crosscutting effective views simultaneously.
Current approaches to software engineering view the development of software as an evolutionary process that involves the continual modification and extension of a program throughout its lifetime. The evolutionary nature of software development means that a significant portion of development time is spent making modifications to existing code rather than on writing new code. This is made difficult by the fact that the required changes often involve understanding and modifying code that is scattered across many different parts of a program.

While programming languages can help ease these difficulties by providing software designers with greater flexibility in choosing a decomposition that best supports the expected tasks, ultimately they do not provide a complete solution because a single decomposition is often insufficient to support all evolution tasks. When crosscutting concerns exist in a program a single decomposition is not sufficient because some concerns will always be scattered across the primary decomposition. Thus some tasks will be more difficult to perform than others unless this crosscutting structure can be explicitly represented and worked with.

In this thesis we presented two prototype tools that support the exploration
and editing of crosscutting structure in source code.

The JQuery tool assists exploring code through the use of query-based code browser that allows for the creation of many different kinds of browsers and for incremental extension of browsers. The goal of JQuery is to combine the advantages of hierarchical code browsers and query tools such that complex exploration tasks can be carried out from within a single view and that the exploration path is explicitly maintained.

Our claim is that, compared to similar tools, JQuery reduces a developer’s cognitive overhead during exploration because they do not have to remember how the elements they navigate to are related. This relationship is shown through the path that connects the various elements in the browser tree. Furthermore, we claim that JQuery reduces disorientation during exploration because developers do not have to switch between views.

A case study was performed in which JQuery was used to perform a realistic development task. We found that we were able to complete many small subtasks from within a single view and that the exploration path maintained by JQuery was useful in terms of backtracking and maintaining orientation.

The Decal system allows developers to edit their code from two different points of view using virtual source files. The classes view provides a set of VSFs that represent the decomposition of a program into classes. Each VSF in this view contains all the declarations to do with a particular class. The modules view represents an alternate decomposition that is similar to what one would get with open classes. Both views are generated from a single common database so that edits to one view are automatically reflected in the other view.

This thesis showed how these two views can both be supported simulta-
neously and provided a discussion of the issues involved in implementing such a system.

We discussed issues surrounding the representation of the program in a database such as the problems encountered in performing early name resolution. The use of soft keys in this case provides a level of indirection that allowed us to separate the maintenance of data integrity from that of program correctness. The level of granularity that we chose in the database schema allowed us perform efficient querying while still being tolerant of errors. We also discussed how the schema effected our design of the Decal language in terms of making a distinction between operations and methods.

The issue of editing semantics is not as straightforward in a system of virtual source files as it is with physical source files because saving a VSF involves a transformation from editing format to storage format, rather than a simple dump of an editing buffer to disk. This transformation must be intuitive to the user so that they will be able to understand what effect their edits will have on the underlying program. Decal addresses this issue through the use of the disjointness property which ensures that every element has at most one location within a particular view. In doing so, Decal is able to mimic the semantics of physical source files.

In a system of dynamically generated multiple views it is possible for users to become disoriented when the contents of their views change drastically from one editing session to the next. We have described this as the temporal continuity of views. Decal addresses this issue to a limited extent by enforcing consistent renderings of VSFs, but there is much that can be done in this area to preserve things like formatting, arbitrary orderings of elements, and free form single line comments. We leave the implementation of this for future work.
A final issue that we described in this thesis is that of the tolerance of errors. As programs are developed in an incremental fashion they are often left in an inconsistent state. Syntax errors and unresolved names are often saved by developers to be fixed later. This can be problematic when the program is stored in a database because databases like to enforce strict integrity on their data. Decal can tolerate syntax errors in method bodies because it stores them directly as text, and can tolerate unresolved names through the use of soft keys.

The two prototypes are meant to serve as a starting point for further research into the area of working with crosscutting concerns. The JQuery tool has already been demonstrated at several conferences and has received interest from the research community. The Decal tool is more experimental. Due to its use of a toy language it would not be practical to test Decal on a real system without a significant amount of basic engineering work to add all the language features that one would expect from a complete object-oriented system. However we expect the ideas presented in this thesis to be helpful in the design of other tools that implement virtual source files.
Bibliography


