Navigating and Querying Code Without Getting Lost

Doug Janzen and Kris De Volder
Department of Computer Science
University of British Columbia
2366 Main Mall
Vancouver BC Canada V6T 1Z4
dsjanzen,kdvolder@cs.ubc.ca

ABSTRACT

A development task related to a crosscutting concern is challenging because a developer can easily get lost when exploring scattered elements of code and the complex tangle of relationships between them. In this paper we present a source browsing tool that improves the developer's ability to work with crosscutting concerns by providing better support for exploring code. Our tool helps the developer to remain oriented while exploring and navigating across a code base. The cognitive burden placed on a developer is reduced by avoiding disorienting view switches and by providing an explicit representation of the exploration process in terms of exploration paths. While our tool is generally useful, good navigation support is particularly important when exploring crosscutting concerns.

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. [4, 19, 7, 8]) and sophisticated visualization tools (e.g. [14, 12, 18]).

AOSD 2003 Boston, MA USA

In this paper 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 an effective 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 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. [4, 19, 7, 8]) 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 paper 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 paper 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 [16]. 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. 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 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 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 `Figure`s 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.

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.

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

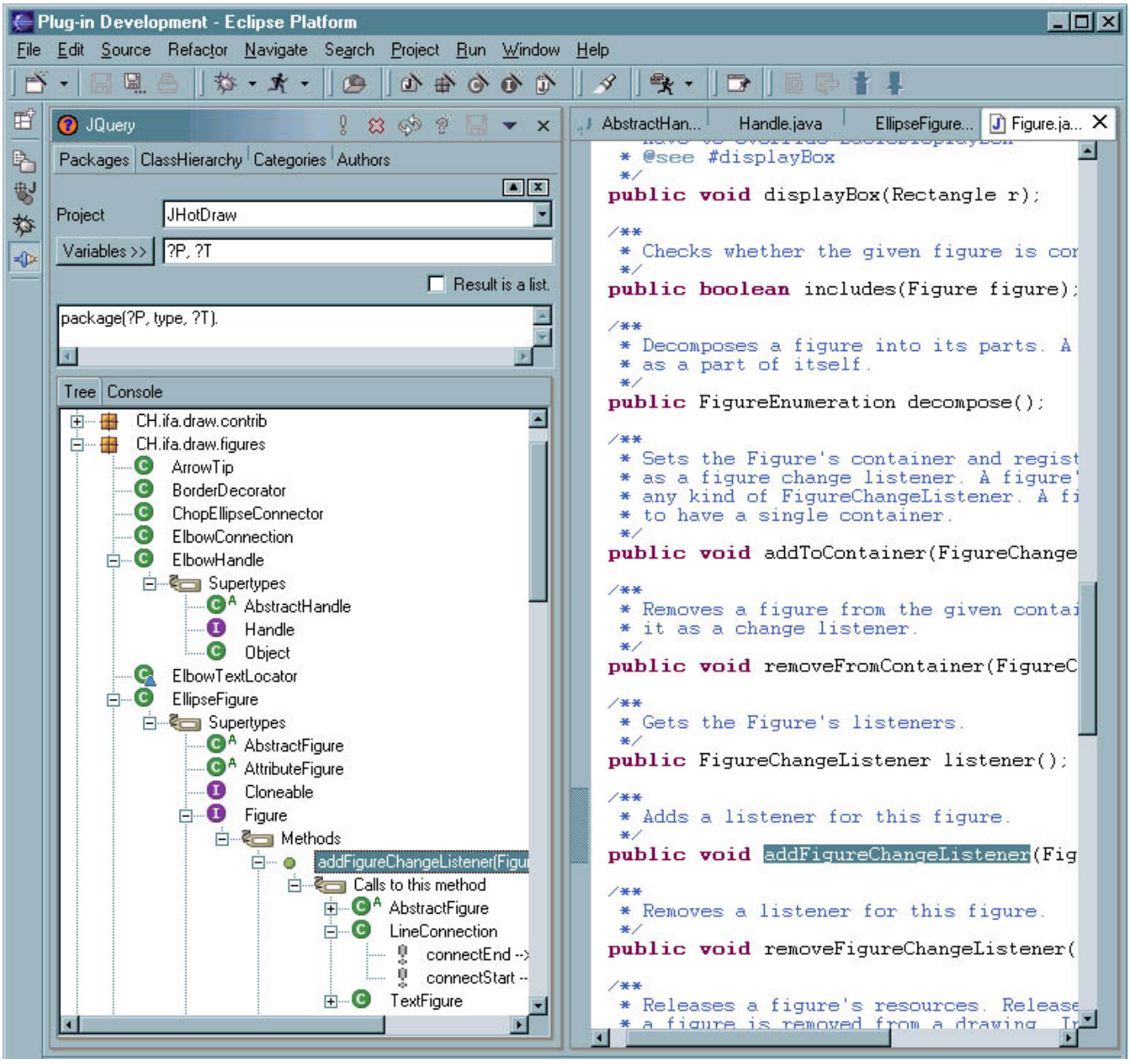

Figure 1: Exploring Figures Implementation in JHotDraw.

was introduced in the QJBrowser prototype and its utility was discussed in detail in a previous paper [16]. 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.

3.2 The Query Language

JQuery is built on top of TyRuBa [10, 9], an expressive logic programming language. TyRuBa is similar to Prolog [11]. 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 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.

```
package(?P, class, ?C),
class(?C, super+, ?I),
interface(?I, method, ?IM),
method(?IM, signature, ?IS),
method(?IM, name, ?IN),
class(?C, method, ?CM),
method(?CM, signature, ?CS),
method(?CM, name, ?CN),
equal(?IS, ?CS),
equal(?IN, ?CN).
```

JQuery is implemented as an Eclipse [15] 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.

Predicate	Description
package(?P)	True if ?P is a package.
package(?P, name, ?N)	True if package ?P has name ?N.
package(?P, type, ?T)	True if package ?P contains type ?T.
type(?T)	True if ?T is a type.
type(?T, name, ?N)	True if type ?T has name ?N.
type(?T, field, ?F)	True if type ?T contains field ?F.
type(?T, method, ?M)	True if type ?T contains method ?M.
type(?T1, type, ?T2)	True if type ?T1 contains inner type ?T2.
type(?T, modifiers, ?M)	True if type ?T has modifiers ?M, where ?M is a list.
type(?T1, super, ?T2)	True if type ?T1 has super type ?T2.
type(?T, tag, ?Tag, ?Val)	True if type ?T has a JavaDoc tag ?Tag with value ?Val
class(?C1, extends, ?C2)	True if ?C1 extends class ?C2.
class(?C, implements, ?I)	True if class ?C implements interface ?I.
class(?C, creator, ?M)	True if an instance of class ?C is created in method ?M.
method(?M, returnType, ?RT)	True if method ?M has return type ?RT.
method(?M, paramType, ?PT)	True if method ?M has a parameter of type ?PT.
method(?M, exception, ?ET)	True if method ?M throws an exception of type ?ET.
method(?M, tag, ?Tag, ?Val)	True if method ?M has a JavaDoc tag ?Tag with value ?Val
refMethod(?R, ?Cler, ?Clee)	True if ?R is a reference from method ?Cler to method ?Clee.
error(?E, message, ?M)	True if error ?E is described by message ?M.
error(?E, severity, ?S)	True if error ?E has severity ?S.

Table 1: Some predefined predicates in the query language.

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. In the implementation section (Section 5) we will discuss how the menu structure can be

configured by an expert user.

Node Type	Relationships
Packages	Classes
	Interfaces
	All Types
Classes	Methods
	Fields
	Subtypes
	Supertypes
	Imports
	Constructors
	Creators
	References Methods
	References Types
	Calls to this type
Methods	References Methods
	References Types
	References Fields
	Calls to this method
	Signature
References	Caller
	Callee
	Caller's callers
	Callee's callees

Table 2: A sample of the ways in which nodes in the tree can be extended.

4. CASE STUDY

In order to assess JQuery's usefulness we conducted a simple case study using JQuery to perform a realistic development task.

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.

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.

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.

4.3.1 Example 1

The first example is one of our earliest searches, the corresponding screenshot is shown in Figure 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 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

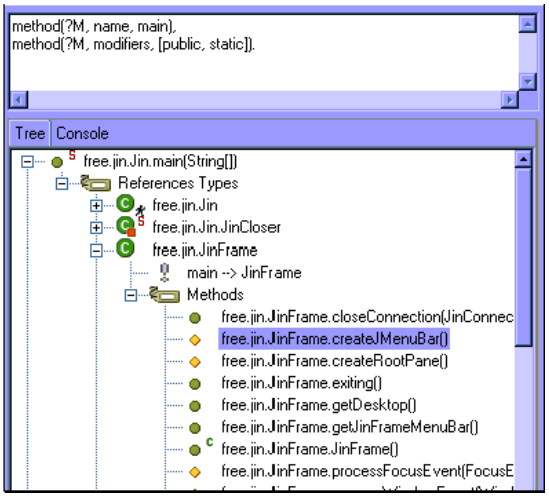

Figure 2: An early exploration of the Jin code.

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.

4.3.2 *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 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.

4.3.3 *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 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

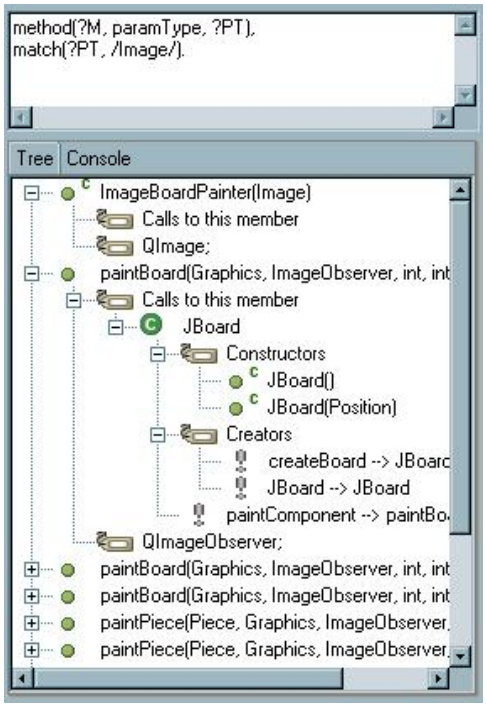

Figure 3: Searching for the use of images in the Jin code.

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.

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.

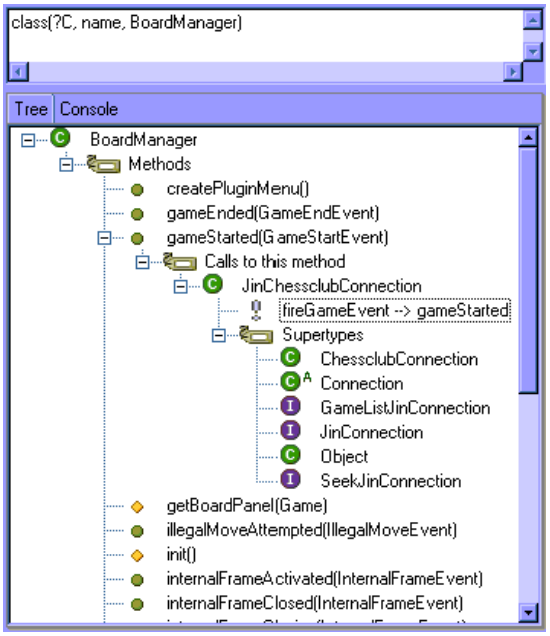

Figure 4: Exploring the `BoardManager` class.

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 *4.3.1*), 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. We will have more to say about this issue in the future work section (Section 7).

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.

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 [15], 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 [13], to improve the usability of the tool.

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 3.2. We briefly describe the procedure for adding a menu item below.

5.1.1 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.

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 [19], ASTLog [8], GraphLog [7], Coven [5] and Stellation [6]. 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 [4] system which we will discuss separately.

There are numerous tools (e.g. Rigi [14], SHriMP [18], Ciao [4], SVT [12] and GraphLog [7]) 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 [14] 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 [12] 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 [4] 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 [17] 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.

7. FUTURE WORK

Possible directions for future work address some of the problems and issues discussed in Section 4.4.

One possibility is to investigate the applicability of more sophisticated visual representations for the exploration history. For example using "fish-eye" views of a graph-based rather than tree-based representation might improve the effectiveness of the tool even further.

Another avenue of exploration concerns the query language and the question of what would be a good query language for a tool like JQuery. As noted before, our initial experience seems to indicate that complex queries are hard to write and it is typically easier to use simple queries and navigate from there. Since it is unclear whether or not the specifics of the query language (Prolog-like) is the cause of this, the issue could be addressed in a number of ways. Assuming that complex queries are useful, but too hard to write in the current language, the issue could be addressed by trying to develop an expressive yet more intuitive query language. Assuming that the specifics of the query language are not the cause, we could address the issue by providing developers with a much simpler and restricted query mechanism: text based grep searches and other simple queries only. Either way, the fundamental questions that need to be answered are: "What kinds of queries are most useful to developers?" and "How do we most intuitively support developers to perform those queries?". Since navigation steps are also a kind of query, the answers to these questions also tie in closely with the issue of which navigation steps are most

useful and how developers should be supported to choose the next step on an exploration path.

8. CONCLUSION

In this paper 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.

Acknowledgments

This work was supported in part by Object Technology International, NSERC and the University of British Columbia. We thank Jonathan Sillito, Gregor Kiczales, Chris Dutchyn, Hidehiko Masuhara and Gail Murphy for their valuable comments, insights and stimulating discussions which have greatly contributed to this paper.

9. REFERENCES

[1] JHotDraw. http://www.jhotdraw.org/, 2002.

[2] The Jin Chess Server. http://www.hightemplar.com/jin/, 2002.

[3] The Source Navigator™ IDE. http://sources.redhat.com/sourcenav/, 2002.

[4] Yih-Farn R. Chen, Glenn S. Fowler, Eleftherios Koutsofios, and Ryan S. Wallach. Ciao: A graphical navigator for software and document repositories. In *Proc. Int. Conf. Software Maintenance, ICSM*, pages 66–75. IEEE Computer Society, 1995.

[5] Mark C. Chu-Carroll and Sara Sprenkle. Coven: brewing better collaboration through software configuration management. In *Proceedings of the eighth international symposium on Foundations of software engineering for twenty-first century applications*, pages 88–97. ACM, 2000.

[6] Mark C. Chu-Carroll, James Wright, and David Shield. Aspect-oriented programming: Supporting aggregation in fine grained software configuration management. In *Proceedings of the tenth ACM SIGSOFT symposium on Foundations of software engineering*, pages 99–108. ACM, November 2002.

[7] M. Consens, A. Mendelzon, and A. Ryman. Visualizing and Querying Software Structures. In *Proceedings of the 14th International Conference on Software Engineering*, pages 138–156, May 1992.

[8] R.F. Crew. Astlog: A language for examining abstract syntax trees. In *Proceedings of the USENIX Conference on Domain-Specific Languages*, Santa Barbara, California, October 1997.

[9] Kris De Volder. Tyruba website. http://tyruba.sourceforge.net.

[10] Kris De Volder. *Type-Oriented Logic Meta Programming*. PhD thesis, Vrije Universiteit Brussel, Programming Technology Laboratory, June 1998.

[11] P. Deransart, A. Ed-Dbali, and L. Cervoni. *Prolog: The Standard*. Springer-Verlag, New York, 1996.

[12] Calum A. McK. Grant. *Software Visualization In Prolog*. PhD thesis, Queens College, Cambridge, December 1999.

[13] Ashish Gupta, Inderpal Singh Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *Proceedings of the 1993 ACM SIGMOD international conference on Management of data*, pages 157–166. ACM Press, 1993.

[14] H. Muller, K. Wong, and S. Tilley. Understanding software systems using reverse engineering technology. In *The 62nd Congress of L'Association Canadienne Francaise pour l'Avancement des Sciences Proceedings (ACFAS)*, 1994.

[15] Eclipse website. http://www.eclipse.org/, 2001.

[16] Rajeswari Rajagopolan and Kris De Volder. Qjbrowser: A query-based approach to explore crosscutting concerns. Submitted to IWPC 2003.

[17] Martin P. Robillard and Gail C. Murphy. Concern Graphs: Finding and describing concerns using structural program dependencies. In *Proc. of International Conference on Software Engineering*, 2002.

[18] M.-A. D. Storey, C. Best, and J. Michaud. Shrimp views: An interactive and customizable environment for software exploration. In *Proc. of International Workshop on Program Comprehension (IWPC '2001)*, 2001.

[19] Roel Wuyts. Declarative reasoning about the structure of object-oriented systems. In *Proceeding of TOOLS USA '98 Conference*, pages 112–124. IEEE Computer Society Press, 1998.