

JQuery: finding your way through tangled code

[A Demonstration Proposal]

Andrew Eisenberg, Kris de Volder
Department of Computer Science
University of British Columbia
Vancouver, BC, Canada 604-822-1290
{ade, kdvolder}@cs.ubc.ca

ABSTRACT

JQuery is a flexible, query-based source code browser, developed as an Eclipse plug-in. A JQuery user can define his or her own top-level browsers on-the-fly by formulating logic queries and running them against the source code. Alternatively, the user can choose from a variety of pre-written browsers, and use them as-is or modify them to suit specific needs. In this manner, JQuery provides the developer with a wide variety of crosscutting as well as non-crosscutting views within a single tool. Elements in the tree can then be queried individually in the same manner allowing further exploration of the complex web of relationships that exist between scattered elements of code, without the distraction of switching tools or losing the context of the original query.

1. RELEVANCE TO AOSD AND PROBLEMS ADDRESSED

The shared goal of Aspect-Oriented Software Development (AOSD) researchers is to provide developers with the necessary arsenal of tools and techniques to deal effectively with crosscutting concerns in the development of software [6]. AOSD covers a broad range of approaches that deal with the problem at different levels and stages of the development cycle. At the code level, this includes both new programming language mechanisms as well as tool based support to work with crosscutting concerns in code.

Integrated development environments (IDEs) provide multiple ways to browse and navigate source code based on various structural relationships between code units. This kind of support, by allowing developers to view and explore structural relationships that connect code units scattered across different files, helps a developer to work more effectively with tangled and scattered code. For example, in addition to browsing by package and class (implicit in the file system) the Eclipse IDE [7] provides the means to browse source code by class hierarchies, regular expression matches, field

accesses, method calls, etc.

However, there are some inherent limitations in the way that current IDEs support these views:

Loss of context A typical exploration task will involve multiple searches and following different types of connections between code units. This will force developers to switch from tool to tool and window to window, causing significant disorientation due to loss of context.

Composition Because different types of views are provided by separate tools, their capabilities cannot be combined.

Tailorability Since the set of default browsers is built-in to the IDE. It is difficult or impossible for developers to define their own customized browser, or tailor the built-in ones to their specific needs.

Our tool, JQuery [4] is a generic code browser, addressing all three problems. JQuery is highly tailorable, allowing developers to define their own browsers by writing queries against a logic database representation of their code-base. Furthermore, the context menus and primitive queries can be tailored for a specific purpose by altering a configuration file. Because browsers are defined as logic queries, and the query language supports composition of logic expressions using standard logical connectives, it is possible to combine features of existing views into new views. Finally, JQuery avoids loss of context by allowing the developer to add a sub-view by executing further queries on any element of an existing view. This way, explorations involving multiple searches can be completed within a single window.

2. TOOL DESCRIPTION

JQuery has been implemented as an Eclipse plugin with the idea that it should have tight integration with an existing IDE so that it can be particularly useful to developers.

It uses the TyRuBa [2] logic programming language to power its searches. TyRuBa is similar to Prolog [3] in that they both provide the capability to programmatically query a logic database. A description of the language is beyond the scope of this demonstration, but [1] contains a useful tutorial.

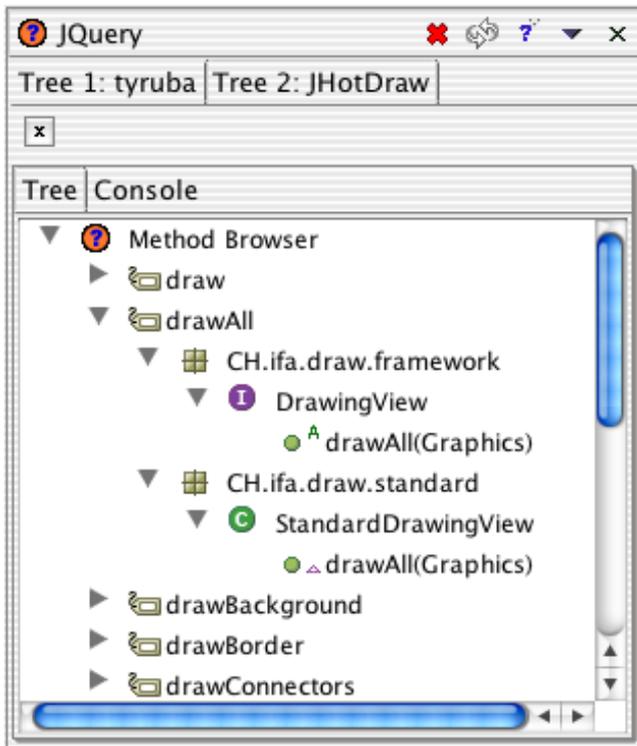


Figure 1: Displaying a top-level, customized method browser query of JQuery.

Before JQuery can query a code base, the code must be parsed and put into the TyRuBa database, taking advantage of Eclipse APIs for parsing the java abstract syntax tree. The database only needs to be created once per instance of Eclipse because source code change events are sent directly to the database which updates itself on the fly.

The results of a query are displayed in a results tree, part of an Eclipse view. Any of the results nodes can be built upon by performing a sub-query, generating a new sub-tree emanating from that node.

3. USING JQuery

Figure 1 shows the main view of JQuery showing the results tree of a customized “Method Browser” top-level query. The “Method Browser” query is one of the standard queries included with the tool that returns all methods in the code base sorted first by method name, package, class, and then method. This customized version displays only methods that contain the regular expression “*draw*”.¹ The standard, non-customized version can be accessed by using the context menu. After that, the query can be edited and customized in the Query Dialog (see Figure 2, explained in Section 3.2). The results nodes act as hyperlinks into the code; double-clicking on any node will open an editor with the code related to that node highlighted and ready to edit.

¹JQuery recognizes the syntax of the Apache Regexp Package [8]. This regular expression matches all strings that begin with the word “draw”.

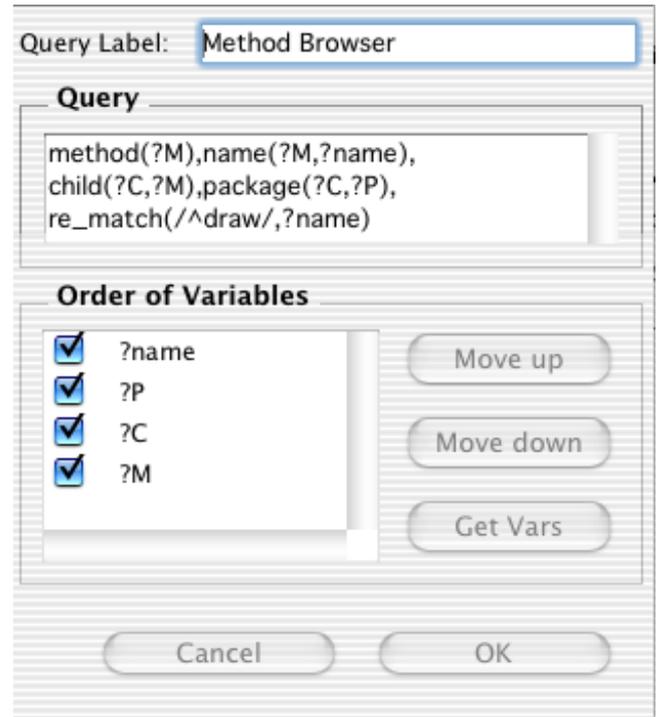


Figure 2: The query dialogue box for the customized Method Browser. In addition to specifying the query, users can choose which variables to display, and in what order they should appear.

3.1 Context Menus

By right-clicking on any of the results nodes, a context menu is displayed. Figure 3 shows the context menu specific for method nodes. The menu items are customized for the selected node. Since the selected node is a method node, the menu contains queries specific to methods such as “Calls” and “Field Accesses”. Clicking on an item in the context menu will execute that sub-query. Or, instead of choosing a pre-defined query, the user can choose to create a custom sub-query by clicking on the “New Sub-Query...” item. The sub-query will generate a new sub-tree beneath the results node that it is associated with. Figure 4 shows the results of executing the “Incoming Calls” sub-query.

3.2 Editing a Query

A developer can edit the underlying query by right-clicking on any query node and selecting the “Edit Query” item. This opens the Query dialogue shown in Figure 2. This dialog shows the underlying query of “Method Browser” of Figure 1. Here the user can edit the TyRuBa code or the variables to display in the results tree.

The top part of the Query dialogue window contains the Query box which stores the query in TyRuBa syntax. In this query, all predicates are connected by a logical “and”.² The predicates *method*(?M), *child*(?C,?M), and *package*(?C,?P) represent relationships between code units.

²in TyRuBa syntax, a comma “,” represents a logical “and” and a semi-colon “;” represents a logical “or”.

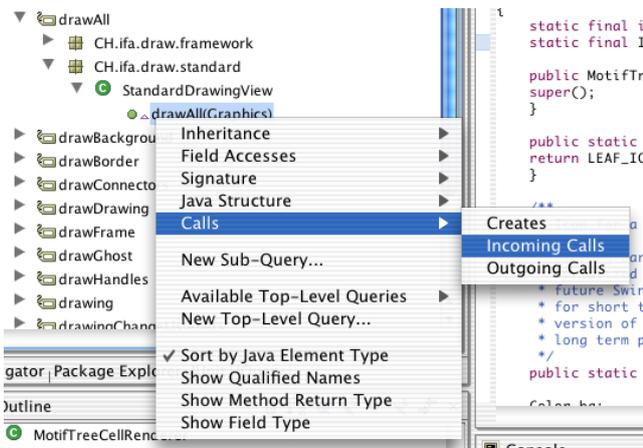


Figure 3: A context menu, specific to a class node.

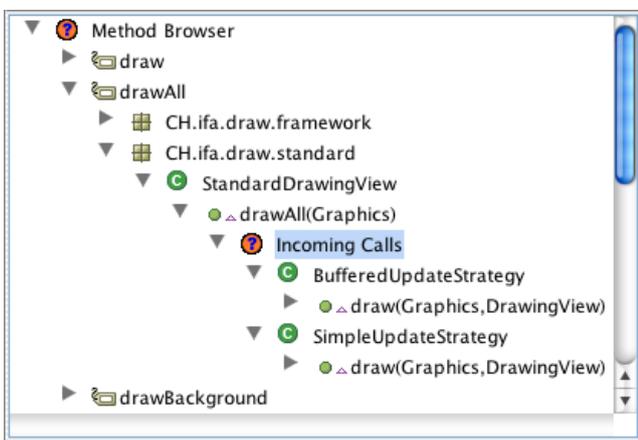


Figure 4: The results of executing the Incoming Calls sub-query

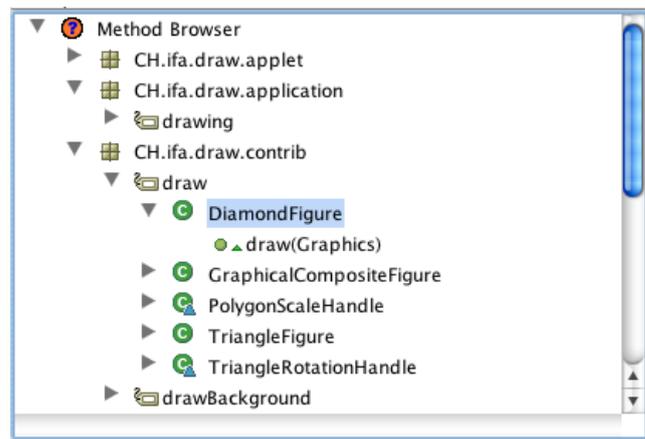


Figure 5: Changing the order of variables displayed alters the results tree.

The predicate $name(?M, ?name)$ represents the relationship between a code unit (in this case a method) and its name. And the $re_match(/^draw/, ?name)$ predicate represents a match between a regular expression and a string. An English equivalent of this query is: “Show all methods, ?M, whose name is ?name, and is a child of ?C in package ?P and ?name begins with the word ‘draw’.”

In the Order of Variables box, underneath the Query box, the developer controls which variables to display and in what order they should be used to be used to the construct successive levels of the resulttree.

In Figure 2, for example, there are four variables. Correspondingly, the results tree of this query has a depth of four (with the “Method Browser” query node as the root). Since the query returns all sets of code units that can be bound to the four variables, the results are ordered such that every path from root node to leaf node represents one set that satisfies the query. All of the code units that bind to the top variable, ?name, are at a depth of 1 in the tree. Similarly, all of the nodes at a depth of 2 are the code units that bind with ?P, or the packages. At a depth of 3 are the nodes representing classes, and the nodes at a depth of 4 correspond to all the methods whose names begin with “draw”.

Figure 5 shows the result of moving the ?name variable down one level. Although the number of leaves of the results tree remains the same, the paths to the leaves are different. This new results tree shows all methods that begin with “draw” ordered by package first. We can now easily determine which “draw” methods exist in which packages.

4. UNIQUENESS OF DESIGN AND IMPLEMENTATION

Because of the lack of generic code browsers in Eclipse and the difficulty of relating current searches to historical searches, code browsing is typically spread amongst multiple views and with previous searches contained in a history that is not immediately accessible. The developer can easily lose his or her way, forgetting the original rationale

for the browsing. Furthermore, the browsers provided with Eclipse are each specific to one type of browsing task. Many browsing or search questions cannot be directly answered or answered at all by existing browsers. For example, one of the standard sub-queries of JQuery on a class is to find all methods inherited from a superclass. As of Eclipse 2.1.1. This search cannot be performed directly using the standard Eclipse tools.

JQuery provides three advantages over existing code browsing techniques. First, because JQuery can inherently support multiple types of browsing, complicated searching tasks can be performed all within the same tool, decreasing the disorientation associated with switching tools to complete a single task. Second, because there is a rich logic programming language that forms the basis of all queries, a wide variety of searches can be performed using only JQuery. Lastly, it is extensible. Users can edit a configuration file to add their own queries to the JQuery menus.

5. WHAT THE AUDIENCE WILL SEE

The demonstration will begin with a brief PowerPoint presentation of the ideas and motivation behind JQuery, why it is useful, and how it is relevant to AOSD.

After that, the audience will see a live demonstration of the tool, showing how to use it to perform a series of typical development tasks. Each successive task will make more and more sophisticated use of the tool.

The first task will be an exploration of the code base of *JHotDraw* [5], locating information from scattered places in the code needed to perform a small change task. This task will exercise the most basic functionality of the tool, demonstrating how JQuery can help maintain search context even within complex, multi-stepped searches, because a JQuery view can retain multiple browsers within one window.

The second task will entail refactoring a piece of the *JHotDraw* codebase, requiring multiple similar changes to different locations throughout the code. In particular, we will formulate a custom query that isolates the scattered code units that must change in order to complete the task. As the presenter makes the changes and refreshes the query, the JQuery's results tree diminishes, implying that the change task is being completed. When the list is empty, the change task is complete. By using custom queries to complete a task, we will show how the underlying query language of JQuery is flexible enough to solve diverse problems. This task will show how we can compose the functionalities of existing views to create a more specific view to solve our exact problem.

The final example will take advantage of *JHotDraw* specific naming conventions and coding idioms to create browsers and navigation menus. The presenter will supply custom top-level queries and context menus to help decompose and navigate the system in terms of higher-level concepts of the *JHotDraw* code base. For example, we will show how to define abrowser in terms of the visitor design pattern as it is specifically implemented in the *JHotDraw* code base.

We will end the demonstration with a question and answer

session, focusing on the types of tasks JQuery can help with.

6. HARDWARE AND PRESENTATION REQUIREMENTS

The JQuery demonstration does not require any special hardware other than a projector and a screen. The presenter will come with a laptop prepared for the talk.

7. REFERENCES

- [1] K. De Volder. Tyruba website. <http://tyruba.sourceforge.net>.
- [2] K. De Volder. *Type-Oriented Logic Meta Programming*. PhD thesis, Vrije Universiteit Brussel, Programming Technology Laboratory, June 1998.
- [3] P. Deransart, A. Ed-Dbali, and L. Cervoni. *Prolog: The Standard*. Springer-Verlag, New York, 1996.
- [4] D. Janzen and K. D. Volder. Navigating and querying code without getting lost. In *Aspect-Oriented Software Engineering*, pages 178–187. ACM, 2003.
- [5] JHotDraw. <http://www.jhotdraw.org/>, 2002.
- [6] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proc. of European Conference on Object-Oriented Programming (ECOOP)*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer Verlag, 1997.
- [7] Eclipse website. <http://www.eclipse.org/>, 2001.
- [8] Apache jakarta regexp. <http://jakarta.apache.org/regexp/index.html>, 1999.