JQuery - A tool for combining query results and a framework for building code perspectives

by

Lloyd Markle

B.Sc. (Hons.), The University of Western Ontario, 2006

A THESIS SUBMITTED IN PARTIAL FULFILMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

Master of Science

in

The Faculty of Graduate Studies

(Computer Science)

The University of British Columbia

(Vancouver)

August, 2008

Abstract

In this dissertation we identify two problems with current integrated development environments (IDEs) and present JQuery as a tool to address these issues.

The first problem is that IDE views answer low level questions and do not provide a mechanism to combine results to answer complex higher level questions. Even relatively simple questions force the developers to mentally combine results from different views.

The second problem is that IDEs do not provide an easy way to create perspectives on project specific concerns such as naming conventions or annotations. Most IDEs do offer support for creating custom perspectives but the effort required to create a perspective is considerably more than the benefit a custom perspective provides.

JQuery is an Eclipse plugin which generates code views using an expressive query language. We have redesigned JQuery to support a number of new user interface (UI) features and add a more flexible architecture with better support for extending the UI.

To address the first problem, we have added multiple views to JQuery where each view supports drag and drop of results, selection linking, and regular expression search. These features enable a user to combine results from different views to answer more complex higher level questions.

To address the second problem, we can leverage the fact that JQuery is built on an expressive query language. Through this query language we are able to define project specific concerns such as naming conventions or annotations and then create views and perspectives for these concerns through the JQuery UI.

Table of Contents

Appendices

List of Tables

List of Figures

Acknowledgements

I would like to thank my supervisor Kris De Volder for his guidance and direction while working on JQuery and also for his insight and feedback while writing this dissertation. I would also like to thank Mark Greenstreet for reviewing my dissertation and the valuable input he provided. I am thankful for Rick Chern, Immad Naseer, and Alex Bradley for their comments and criticisms of Chapters 4 and 5 and the improvements they helped me make.

I would like to thank UBC for supporting this work and for a wonderful environment to work in. Thank you to everyone in the SPL lab here at UBC. I learned many things through conversations with you and I appreciate all of your insights into my own work.

Lastly, I am grateful for my family and friends for their support and encouragement throughout this process. A special thank you to my fiancé Laura, who has patiently supported me throughout my studdies at UBC and in particular the last half of my work.

Chapter 1

Introduction

1.1 Thesis

In this dissertation we present our enhancements to the JQuery tool [16] and provide preliminary evaluations of the tool with two claims:

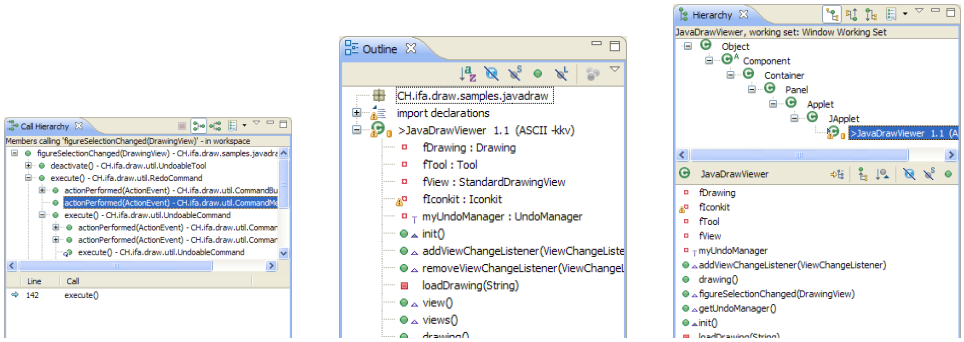
1. JQuery users can answer more complex queries through the user interface than with current integrated development environments.

2. Building collections of code views (perspectives), that take advantage of project specific naming conventions and/or project specific annotations, with JQuery is less work than building perspectives in current integrated development environments.

1.2 Problem

The modern integrated development environment (IDE) is designed to be a text editor with graphical displays of the project files that are being manipulated. These displays are called views and answer queries about the static structure of the code in a way that makes common navigation tasks easier. We have identified two weaknesses with modern IDEs.

The first weakness is the difficulty in combining results from multiple views in modern IDEs. Sillito *et al.* performed a user study to determine the kinds of questions IDE users ask as they work [27]. We can see from their work that users often ask questions which require combining results from different views in the IDE. Although an IDE may answer low level questions such as "what methods are in this class?" or "who calls this method?"directly with a particular view, the query "who calls this class?" is difficult to answer directly because we cannot combine the set of results in one view with the question being asked in the other. In this case, there is no way to take the set of methods in the "what methods are in this class?" view and ask "who calls these methods?" on that set.



(a) Call Hierarchy View (b) Outline View (c) Type Hierarchy View

Figure 1.1: Three common Java views in Eclipse.

The second weakness is that perspectives on project specific concerns are difficult to develop. Java and most newer languages have introduced consistent coding styles and naming conventions. Modern IDEs such as Eclipse [21] have shortcuts for naming conventions (e.g., search shortcuts for camel case names) but not support for navigation in their code perspectives. Adding support to the perspectives would require building extensions to the IDE. For example, a perspective designed to navigate only the classes that implement a listener interface and make calls to setter-methods would require a significant amount of work because a developer would have to extend the IDE with a new view for this purpose.

1.3 Motivation from Eclipse

This section elaborates on the problems discussed in Section 1.2 by using concrete examples from a modern IDE, Eclipse. We have chosen Eclipse [21] as representative for modern IDEs because it has a wide variety of source code views and it provides the user with many facilities for extension and customization. The arguments we make in this dissertation would apply to other modern IDEs as well though most IDEs do not offer as many views as Eclipse nor the same level of customization.

Eclipse offers several views for the Java programming language such as the “Package Explorer View”, “Type Hierarchy View” and “Call Hierarchy View” as shown in Figure 1.1. Each of these views is designed to answer

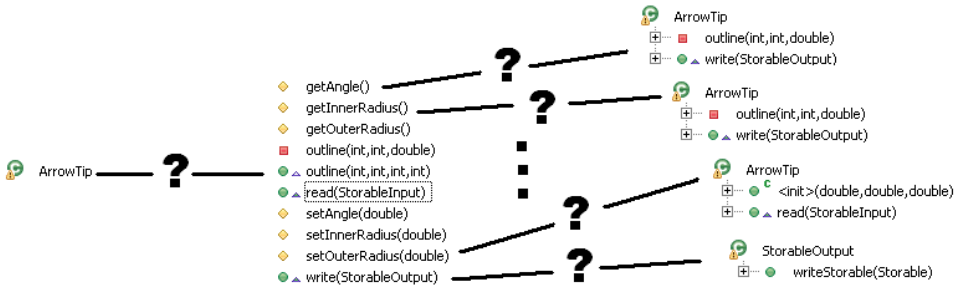


Figure 1.2: Solving "Who calls this class?" without JQuery.

questions such as "what packages are in my system?", "what are the subclasses of this class?", or "what methods call this method?". Each Java view is useful on its own to answer the specific question or questions for which it was designed. These views all support some level of customization to the question they answer (e.g., filters to only display public members or change ordering of results) and display the answer as results in a hierarchical tree structure.

The first problem with these standard Eclipse views is they are not capable of combining information to answer higher level questions. Answering the question "who calls this class?" requires being able to create a set of results and execute one of the existing queries on that set. Instead of asking the class for calls to its methods, we follow the sequence illustrated in Figure 1.2. First we must ask the class for its methods and then ask "who calls you?" on each method. Even this is not an effective solution because for each "who calls you?" query, the view updates and we lose the previous set of results. In this case, we cannot answer the "who calls this class?" question because we cannot combine results from different views.

More complicated queries such as "are there any methods with 'get' in the name being called from any listener classes?" cannot be answered either. To answer such a query we have to find a set of getter methods and see if any of those methods are being called from listener classes. In Eclipse, this kind of searching must be performed manually by the user. The user must find all the getter methods, examine their call hierarchies individually, and inspect any places where the method is called to see if the class which is calling the method is part of the listener pattern.

The second problem we encounter in Eclipse is the lack of support for project specific concerns. Eclipse offers no browsing support for project specific higher level structure defined through coding idoms. For example,

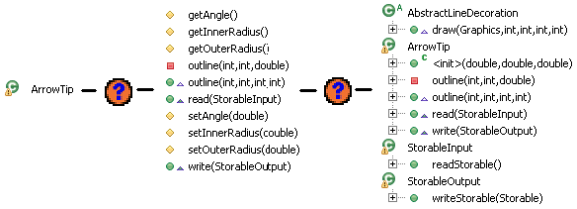


Figure 1.3: Solving "Who calls this class?" with JQuery.

camel case class names that include hints to the purpose of the class such as `EventListener`, `TreeSet`, `HashMap` are easy to read and we naturally conclude that these classes represent some form of listener, set, and map respectively. The information provided by naming conventions is useful to the developer and is indeed a part of the code but the perspectives are designed for browsing the language and not the project specific concerns these conventions express.

Another example of project specific information is Java annotations. Annotations often provide useful information to the developer and indeed are a part of the code but are not a part of Eclipse views. Annotations can be applied to elements in the code and used to express arbitrary information about the elements they mark. Some common annotations are supported by Eclipse (e.g., `@Deprecated` or `@Override`) but annotations can contain information for deployment time processing tools which cannot be handled by Eclipse. For example, frameworks including Castor [17] and JDBC [29] that allow a user to specify database schema inside their annotations cannot be displayed by Eclipse because this information varies from project to project.

1.4 JQuery

We extended JQuery to address these problems. JQuery is an Eclipse plugin with an expressive query language designed to display results from code queries in a tree view [16]. The JQuery version 3.14 offered a single tree view to dislpay answers to user questions selected from a context sensitive menu. Our primary contributions to JQuery are extendeding the UI to allow a user to position multiple JQuery tree views around the Eclipse workbench, adding a search box to each view, and introducing ways to combine results from different views using drag and drop and view linking.

These new UI features allows a JQuery user to perform more complex

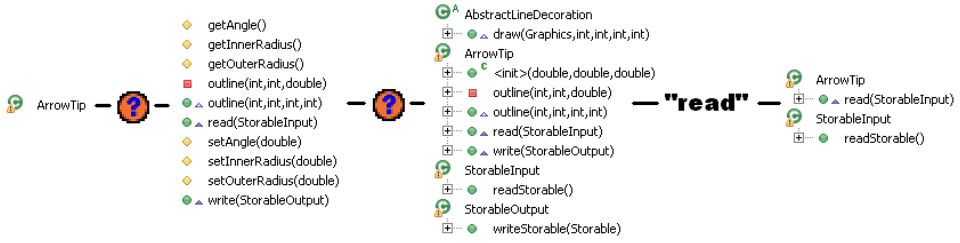


Figure 1.4: Solving "What 'read' methods call this class?" with JQuery.

queries by combining results from different views. For example, Figure 1.3 illustrates how the question "who calls this class?" can be answered in JQuery. The JQuery user first creates an incoming calls view, then queries to find the methods of a class and drops the set of methods onto the incoming calls view to answer the question.

Users can also now make the result set more narrow by applying a text search on the view to display only the results which match the word "read" as seen in Figure 1.4. This enables them to answer the question "are there any 'read' methods in this set of methods?". Narrowing a set of results can also be very helpful for developers to show relevant information for a task in one view [2, 18].

In addition to combining results from multiple queries, JQuery users can now define code perspectives composed of JQuery views, called JQueryScapes, by taking advantage of the flexible UI and expressive query language. JQuery users create JQueryScapes by positioning their JQuery views around the perspective and linking views together or linking to the editor. For example, the Java Browsing perspective from Eclipse can be created using JQuery and linking views to the code editor.

JQuery users are able to create JQueryScapes by taking advantage of the underlying expressive query langage JQuery uses. Advanced users can tweak the code representation and define custom UI menus and icons by using the underlying query language. JQuery views are created and positioned through the UI but the contents are a reflection of the code representation defined in a set of configuration files. Because JQueryScapes are easy to create, users can create them for project specific concerns in the code including annotations and naming conventions.

1.5 JQuery Users

Throughout this dissertation we will be making use of the terms developer and user. Both terms refer to software developers, but for our purposes we will describe the user as someone who will make use of the JQuery plugin and the developer as an Eclipse plugin developer making use of either the JQuery framework or the Eclipse API to build their own plugins.

Furthermore we will make a distinction between a JQuery user and an advanced user. The JQuery user will use the JQuery user interface (UI) to answer questions about the code. The advanced JQuery user will be considered an expert with the underlying query language and be able to extend the existing JQuery code representation by writing additional rules and logic in that query language to build JQueryScapes.

1.6 Contributions

In this section we will summarize our contributions to JQuery and describe our preliminary evalutions of these new features. We have added significant extension to JQuery 3.14 to make it more suitable for future research by:

1. Adding support for multiple JQuery views as described in Section 2.2.2. A full description of the JQuery UI and its new features will be provided in Chapter 2.

2. Improved the responsiveness of the UI and improved the integration with Eclipse. A full description of these changes will be provided in Section 3.1.

3. Refactored JQuery into two plugins to create a clear seperation between the frontend user interface code and the backend database code. A full description of the backend is provided in Chapter 3.

In addition to these contributions to JQuery, we have performed a preliminary evaluation of these new features of JQuery in Chapters 4 and 5.

In Chapter 4, we will define a sample set of queries and show how a JQuery user can answer more complex queries through the user interface than with current IDEs.

In Chapter 5, we will show how an advanced JQuery user can build perspectives, that take advantage of project specific naming conventions

and/or project specific annotations, with less work than building perspectives in current IDEs by performing a case study to demonstrate the amount of work required to create perspectives in JQuery and explain why a similar study in our representative modern IDE, Eclipse, is beyond the scope of this work.

Both validations will relate the new features we added to JQuery to those offered by Eclipse as a representative of modern IDEs. Eclipse is a good representative of IDEs, in particular for Java IDEs, because it provides a large number of code views and good support for customization. It is important to compare JQuery to a flexible IDE because JQuery is designed to permit a user to easily create and customize their views. Eclipse also has an extensive API which allows developers to add their own custom views to the IDE.

1.7 Summary

In this chapter we have identified two problems with modern IDEs and introduced our extensions to JQuery to address those issues. The remainder of the dissertation will describe the JQuery tool in detail and we will show how to use JQuery to validate our two claims.

Chapter 2 will give a user level perspective on JQuery by introducing key UI and configuration concepts. Chapter 3 will give the details of the JQuery implementation and provide documentation for a developer interested in extending our work. Chapter's 4 and 5 will the first claim of this thesis through the use of a few sample complex queries. will validate the second claim of this thesis and illustrate how JQueryScapes can be used as project specific code browsing perspectives. Chapter 6 will discuss other related work. Chapter 7 will conclude the dissertation and provide a road-map for future work on the JQuery plugin.

Chapter 2

JQuery

In this chapter we present instructions for using the JQuery tool. These instructions will introduce key concepts used in validating our claims made in Chapter 1. We will not be validating claims here but instead we will be introducing JQuery terminology and techniques relevant for later chapters. We will provide instructions for features of JQuery version 3.14 [16] and our extended JQuery version 4 in this Chapter with Section 2.1 outlining our technical contributions to the JQuery UI.

In Section 2.2 we will provide instructions on using the JQuery user interface (UI). This section will target the average JQuery user who will use the UI to answer questions about their code. We will see how a JQuery user is able to combine results from different queries to answer more complex queries. In particular Chapter 4 will build upon the work in this section to validate our first claim, JQuery users can answer more complex queries through the user interface than with current integrated development environments.

Section 2.3 will provide instructions for the advanced JQuery user to customize the JQuery views. An advanced user is familiar with the underlying query language and will be able to extend the JQuery code representation and add custom menus and icons to the UI using that query language. In particular Chapter 5 will use the material covered in this section to validate our second claim, that an advanced JQuery user can build collections of code views (perspectives), that take advantage of project specific naming conventions and/or project specific annotations, with less work than building perspectives in current IDEs.

For the interested Eclipse developer and for the sake of completeness, Chapter 3 provides a detailed description of the JQuery implementation.

2.1 Technical Contributions

This section will describe our technical contributions to JQuery. Consult Section 2.2 for more information on the terminology used here.

The JQuery 3.14 UI [16] consisted of a single tree view and a context-sensitive menu to guide the query process. Users were able to select a result node from the tree view and expand the tree by querying a particular result. We have preserved this interface in JQuery 4 but have added several new features.

Our extensions to JQuery consist of allowing a user to create multiple JQuery tree views where each view has its own regular expression filter box. We have also designed and implemented ways for these views to interact through dragging query results and dropping them onto other views or linking the selection of one view to another. Apart from these new features we have also improved the integration between JQuery and Eclipse by offering support for most Eclipse features on the JQuery menu and by allowing some JQuery features to appear on Eclipse menus.

The remainder of this Chapter will discuss how to use both the new and old features of JQuery and provide an introduction into how to customize the JQuery UI.

2.2 User Interface

In this section we will provide instructions on the JQuery UI for an average user. It is intended to be an overview of how JQuery answers code questions while covering, in detail, important features such as the regular expression search box, drag and drop, and view linking which are used to combine results from different queries. These features will become important in Chapter 4 as we validate the first claim of this thesis in.

The JQuery UI is designed to be an "all-in-one" browsing solution for Java. It displays answers to user questions about source code in a tree structure. Questions about packages, classes, call structure, inheritance, etc. are all supported by a single JQuery tree view out of the box. A sample JQuery view showing the outline of a class can be seen in Figure 2.1.

The nested tree structure of a JQuery view closely resembles the way existing Eclipse views already work. The difference is that Eclipse views do not provide a way to specify the question being asked. Rather, each view is hard coded to display answers to a few specific questions. For example, the Eclipse Package Explorer view displays results to the question "what projects, packages, files and classes are in my system?" through a nested structure where classes are nested inside files, files are nested inside packages, and packages are nested inside projects. In JQuery, we provide a way in the UI for the user to select the question the view will answer rather than hard

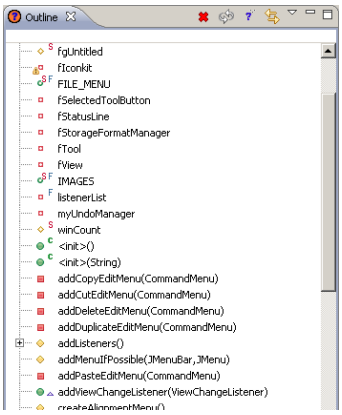


Figure 2.1: Sample JQuery view.

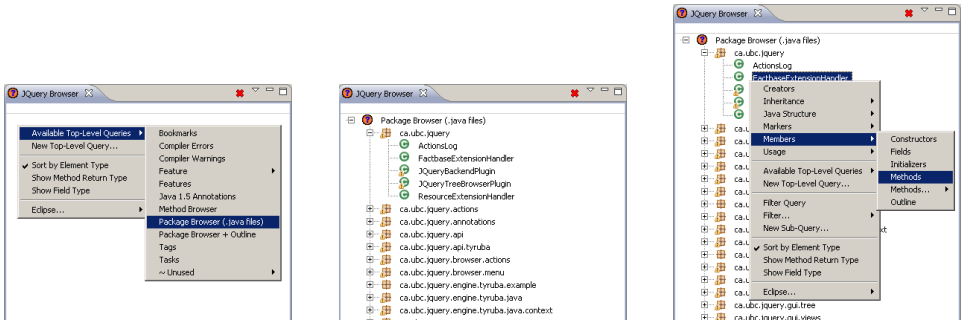
coding the question.

To ask a question in JQuery, the user can select one of the menu items available on the popup context menu as shown in Figure 2.2(a). A node representing the selected question, also known as a *query node*, will appear in the tree viewer and the answers will be nested inside the tree structure. Figure 2.2(b) shows the results to the "Package Browser" query where each package is nested below the query node and each class is nested below the package it belongs to, in much the same way as Eclipse.

The user is also able to ask questions about existing results in the view using the same context menu technique. A user can select an existing result in the view, open the context menu, and the menu will automatically display appropriate potential questions for the selected result. Figure 2.2(c) shows the menu with appropriate questions for the selected class, in this case, questions about finding the methods, fields, and constructors of the class.

The context menu also illustrates an important distinction between two types of questions in JQuery: the *targeted* question and the *top level* question. The *targeted* question requires a target and is designed to provide information about that target. For example, the "Methods" query node in Figure 2.3 is an example of *targeted* question because it asks "what are the methods of *this* class?".

A *top level* question on the other hand has no target. It could be considered to be asking a question about the entire code base rather than a particular element. In Figure 2.3, we would say that the "Package Browser" question is a *top level* question because it displays all packages and classes



(a) Selecting a query from the context menu

(b) Executing a *top level* query

(c) Context menu for a class result

Figure 2.2: How to ask questions and view answers in JQuery.

in the system.

The process of selecting a question from the menu and further exploring the results using more targeted questions can be repeated as often as needed. The results of this repeating technique can be seen in Figure 2.3. In this figure, the user first asks the system for all packages and classes and then refines the search by asking for the list of methods of the `FactBaseExtensionHandler` class. This technique lets the user ask specific questions and investigate specific answers which will help them browse their code better [25]. A few iterations of this process however can lead to some difficulty viewing results as the number of results displayed in a view may be quite large.

We have provided two solutions to the problem of a large results tree, a regular expression search box and the ability to have multiple JQuery views. Both these solutions allow the user to reduce the number of nodes in the tree while still displaying all of the important results. The regular expression search will hide all results that don't match the expression. Multiple copies of the same view allow a user to view two different sections of the results tree that would not normally fit in a single view. We will also see how having multiple views also enables users to combine results from different queries.

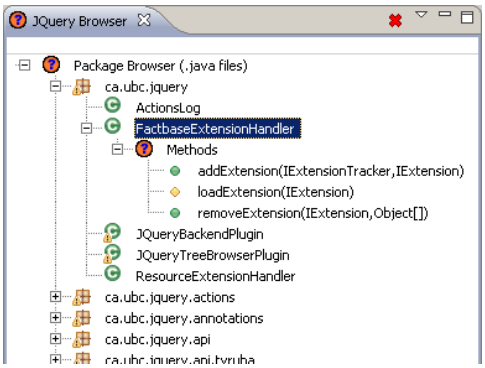


Figure 2.3: JQuery tree view with the top level "Package Browser" question and the targeted "Methods" question in JQuery.

2.2.1 Regular Expression Search

After a user refines a search by asking a few targeted questions, the JQuery view displaying the answers can become cluttered. New to JQuery 4 is a regular expression search box to address this issue. The regular expression search box is a small text box at the top of each JQuery view to help reduce the number of results displayed. Because JQuery views all share the same interface, the user only has to learn one searching mechanism which can then be used in any custom view that they create. For example, a user can apply a text search to a set of classes or a set of packages the same way they apply a text search to methods.

The logic for the search expression is straightforward. *If* any ancestor of a node matches the expression, *then* the descendants also matches and likewise *if* any descendant of a node matches the expression, *then* the ancestors also match. Figure 2.4 shows how this general purpose text search mechanism is flexible enough to allow a user to take a packge explorer view, and search for a particular package name *or* particular class name in the same expression.

2.2.2 Multiple JQuery Views

Another method to reduce the number of results inside a view is to create copies of the view to display different sections of the result. As a new feature to JQuery 4, the user is able to create multiple copies of a view and change the contents of each view independently. There are two kinds of JQuery tree

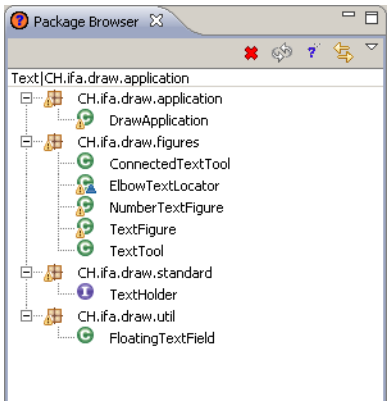


Figure 2.4: The JQuery regular expression search box.

view, the standard JQuery tree view and the query-rooted view.

The standard view can answer both top level questions and targeted questions. This straightforward "point-and-query" view is designed primarily to answer top level questions and further refine their answers using targeted questions. The standard view contains a stop button to tell JQuery to stop finding results to a computationally expensive question. The user is able to make copies of the standard view by selecting the **Clone View** menu item from the JQuery view menu.

The query-rooted view, unlike the standard view, can only display answers to targeted questions. This view has a much closer resemblance to existing Eclipse views because it contains a single question at its root and displays answers to that question. The flexibility of this view is the fact that it can be created from any query node. This means that any of the questions available on the context menu can be used to create a query-rooted view.

The query-rooted view has the same interface as the standard view. This includes support for asking questions about the nodes in the view and the stop button to prevent a computationally expensive question from disrupting the user interface. There are additional buttons at the top of this view to refresh the views contents, change the question it is answering, and link the query target to results from other views.

To create a query-rooted view, the JQuery user must drag a query node from an existing JQuery view and drop that node onto the Eclipse workbench. Query-rooted view can be created using both top level questions and targeted questions. The query node is placed at the root of the tree

and answers to the question are displayed in the view. Once created, the query-rooted view will behave the same way as any regular Eclipse view and can be resized and moved as needed.

Query-rooted views provide a single-click way to change the target for the query node at the root of the view and then update the view contents. Currently, we have two methods to change the target: (i) drag and drop and (ii) view linking. We will look at both in detail.

2.2.3 Drag and Drop

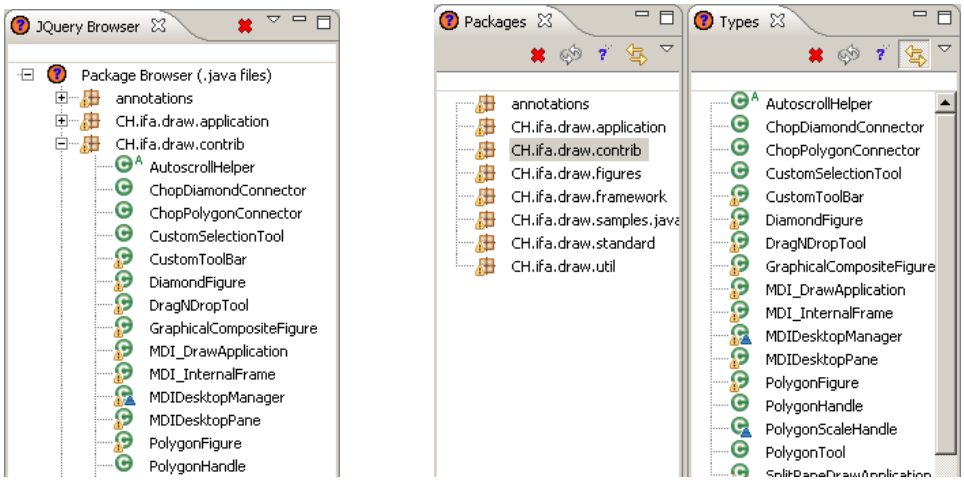
New to JQuery 4, the multiple views provided by JQuery now allows for a single-click way to set the target of a targeted question, drag and drop. The JQuery user can drag a JQuery result node from any other JQuery view and drop that result onto a query-rooted view to set the target. The view will automatically update to display the answers to the question as if it had been asked about the dropped target. For example, the user can drop a method onto a view with an incoming calls query node at its root and display all the incoming calls to that method.

JQuery can handle more advanced cases too because the drag and drop mechanism is flexible. A user can drop a set of methods from a particular view and display all the incoming calls for that set of methods. Users can even mix types and drop several classes or a combination of classes and methods onto the view. For example, dropping such a mix of classes and methods onto an incoming calls query node will display the incoming calls to the set of methods and classes that were dropped onto the view.

2.2.4 View Linking

New to JQuery 4 is view selection linking as an alternative to drag and drop to set the target for a query-rooted view. A JQuery user is able to link the target in a particular view to the selection from another. For example, in Figure 2.5(b) we have a package view and a types view with the types view linked to the package view. Whenever the user selects a package, the types view will update to display the types that are defined within the selected package or packages.

The linking feature allows a JQuery user to *flatten* the default tree structure into several linked views. Instead of visualizing the results in a layered tree structure as shown in Figure 2.5(a), the user can *flatten* the display such that each layer is represented in a linked view as in Figure 2.5(b). Both the tree structure and the flat structure present the same information but the



(a) Standard tree view package explorer

(b) Flattened package explorer split into two linked views

Figure 2.5: Hierarchical and flattened browsing comparison.

flattened version has the advantage of displaying only one type of data per view. In this case, one view displays packages and the other view displays types.

In addition to linking to other JQuery views, the linking feature also supports linking to the source code editor to display information about the code the user is editing. For example, a user can create a view to show the incoming calls to the method the editor is displaying. In a similar fashion, the user can also create a view to display information about the context of the editor selection. For example, a user can create a JQuery view to show the methods defined in the class the editor is displaying.

We repeat for clarity that all JQuery query-rooted views have the same interface, and views linked to the editor are no exception. This means a user can drop query targets onto the view for quick viewing and then, when the editor is reactivated, the view will return to displaying information from the editor selection. The user can then use the context menu to ask questions about result nodes that are in the view and browse the answers in place in that view.

Linking to the source code editor introduces some additional options for the user. The user may want to use the link information as a target for the

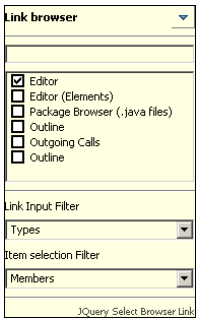


Figure 2.6: The link browser dialog in JQuery. This particular linking strategy (link target and filters) will make this view behave the same as the Eclipse Outline view.

query node at the root of the view or use the link information to select items in the view. More specifically, users may want to view query results based on the items in the code that they are editing *or* they may want items to be selected from the list of results based on the editing context.

To address these two concerns, the view linking system has two types of filter: (i) the input filter and (ii) the selection filter. Both filters are accessible from the link browser dialog shown in Figure 2.6.

Input filter: The input filter is used to filter potential query targets. If we consider the outline view in Figure 2.1, we may not want to outline a package or a method but instead only outline a class. In this case the user applies a filter to the input such that the query node at the root of the view only executes on the classes that are input to the view.

Selection filter: The selection filter is used to select items which are displayed in the view. As an example, the Eclipse Outline view selects a method or field in the view based on the editing context. The JQuery selection filter is defined such that any items that pass the filter and are displayed in the view, will be selected. A JQuery user can achieve the Eclipse Outline view effect using JQuery's selection filters.

We claimed, in Chapter 1, that the combination of features described in this section enable JQuery users to answer more complex queries through the UI than with current IDEs. We will validate this claim in Chapter 4. We offer this section as an introduction to help understand how regular

expression search, drag and drop, and view linking enable a user to combine query results to answer more complex queries.

2.3 Customization

In this section we will show how an advanced JQuery user is able to customize JQuery. We will introduce the important concepts needed when validating the second claim of this dissertation, that an advanced JQuery user can build collections of code views (perspectives), that take advantage of project specific naming conventions and/or project specific annotations, with less work than building perspectives in current IDEs, in Chapter 5. We will focus on the expressive power of the query language and how to customize the queries that JQuery executes.

The advanced JQuery user can customize many parts of JQuery because JQuery is configured through an underlying query language. Every item on the JQuery menu represents a query written in an expressive query language. All the results displayed in the tree structure are based on the results returned from a query. In fact, even the items on the menu are generated using the same query language.

Customizing JQuery includes making changes to the UI as well as adding higher level structure to the code representation through logic rules. Most of the UI is customizable in JQuery including the queries on the context menu, the filters to apply to view links, and the icons and labels in the results view. We will look at each of these UI customization possibilities and provide examples to illustrate how an advanced user can customize both the JQuery UI and the factbase.

We must first understand how JQuery configures its UI and factbase before we can begin making any customizations. Section 2.3.2 will describe how the JQuery UI and factbase are configured using rules and how JQuery displays results from queries. Section 2.3.3 will cover modifying to the UI and Section 2.3.4 will cover modifying the factbase by adding new rules.

2.3.1 From Prolog to TyRuBa

Throughout this section we will assume the reader has some familiarity with logic languages and in particular with the TyRuBa programming language [12]. TyRuBa uses a syntax similar to Prolog [30].

Prolog defines a set of facts over which a user can query. Query expressions consist of predicates joined together with logical AND and OR operators using , and ; respectively. Logical rules can be added to combine

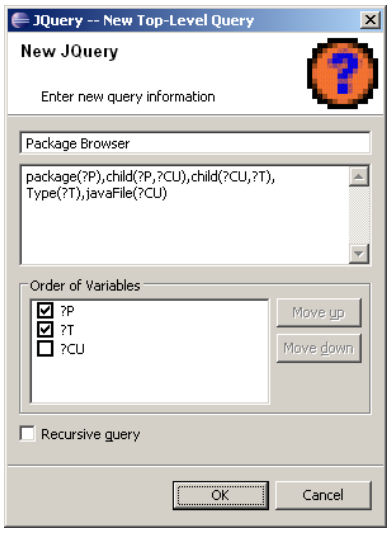


Figure 2.7: The JQuery query edit dialog defining a package browser query.

facts by declaring the predicate and its parameters followed by a `:-` with a logic expression to define what values those variables can receive.

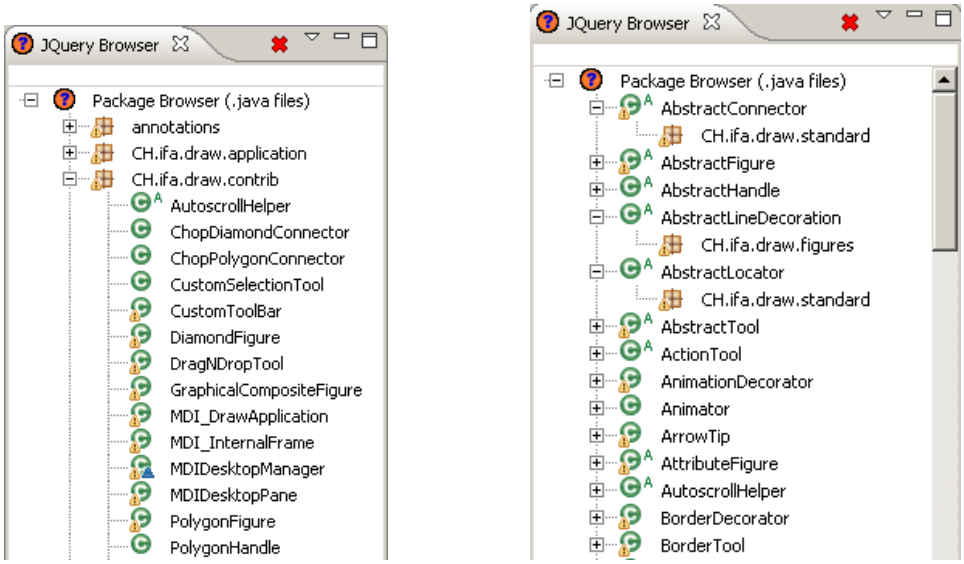
The most notable difference between TyRuBa and Prolog is that each variable name in TyRuBa is proceeded by a `?`. In JQuery, we will occasionally see a variable which is proceeded by a `!` meaning that the variable is designated an input variable. TyRuBa also defines a mode for each predicate to help optimize the query execution. More information about TyRuBa and its syntax can be found on the TyRuBa website [12].

2.3.2 Configuring JQuery

In this section we will introduce how the JQuery UI and factbase are configured and discuss how JQuery generates tree views from queries [24].

The advanced user is able to write queries in addition to the predefined queries available on the context menu. Figure 2.7 shows the JQuery query edit dialog accessible from the "New Top-Level Query" context menu item. This dialog helps advanced users compose and name their own query nodes or edit existing query nodes. The query edit dialog contains help for ordering the results from the query as well as auto-completion to help new users become familiar with the available predicates defined in JQuery.

Once a query is written and a variable ordering chosen, JQuery executes



(a) Package browser view with standard variable ordering `?P,?T`

(b) Package browser view with custom variable ordering `?T,?P`

Figure 2.8: Two different variable orderings of the package browser query and their resulting tree displays.

the query and displays the results. The ordering of variables determines the tree structure that will be displayed to the user. For example, consider the "Package Browser" query with the standard variable ordering `?P,?T`:

```
package(?P),child(?P,?CU),child(?CU,?T),
Type(?T),javaFile(?CU)
```

This query executes and finds all packages and classes in the system so that JQuery can generate a view. In this case, the tree displays the query node, with packages nested inside the query node, followed by the classes nested in the appropriate packages as we can see in Figure 2.8(a). Reordering the variables to `?T,?P` will display the classes first, then the packages as shown in Figure 2.8(b). Any variables that are not specified in the variable ordering will not be displayed in the results tree.

Writing queries and ordering variables are just two examples of how the JQuery query language permits an advanced user to customize JQuery.

Most of the JQuery UI is configured through a set of files which are loaded when JQuery is initialized. These configuration files are essentially TyRuBa programs that define a set of predicates and rules to configure the context menu, view filters, and icons and labels that are displayed in the tree viewer. There are also logic rules to configure a high level structure on the factbase to make queries easier to write.

The set of configuration files that JQuery loads can be found in the Eclipse preference pages under JQuery/Tyruba Backend. By default JQuery loads all its configuration information from the `rules` directory of the plugin but the default configuration files can be changed using the preferences dialog. The next section will discuss the details of these predefined queries through examples taken from the existing JQuery configuration files.

2.3.3 Modify the User Interface

There are three kinds of customizations possible for the JQuery UI. The advanced JQuery user can customize the menu items on the context menu, the filters defined for view linking and the icons and labels of result nodes. We will look at each one in detail in this section and consider examples taken from the JQuery configuration files in the `rules` directory of the plugin.

The first customization we will look at is how to add a custom query to the context menu. Everytime the JQuery context menu is displayed we take the selected item from the view and execute a query to find the available menu items for that selection and generate the context menu accordingly. Top level queries, such as the "Package Browser" query, use the `topQuery` predicate to define query items on the menu while the rest of the query items on the menu are generated from the `menuItem` rule. This is an example `menuItem` rule taken from JQuery's `menu.rub` configuration file:

```
menuItem(?this,["Calls","Incoming Calls"],
    "incomingCalls(!this,?M,?Ref),child(?C,?M)", ["?C","?M","?Ref"])
    :- Callable(?this).
```

This is the definition of the Calls/Incoming Calls menu item and it has four arguments. The first argument is bound to the selection in the UI and is used to determine when the rule applies. In this case, the rule applies to anything that is a callable. The second argument specifies the menu path. The third argument is a string representing the query to execute while the fourth argument is a list of strings representing the default variable ordering.

We can make this rule apply to types as well as callables by changing the last line in the sample to `":- Callable(?this); Type(?this)."`. The

new rule states that this menu item will appear if `?this` is a callable *or* a type. We are able to change the query, the menu path, or the default variable list of this menu item by editing these three lines of code.

If we consider the third argument of this rule, the query string, we will notice that there is an input variable `!this`. The `!this` variable has a special meaning for JQuery and it is the `!this` variable that makes this query a targeted query. When the query executes, the `!this` variable is bound to the target of the query. For menu items, the `!this` variable will be bound to results selected in the tree before executing the query.

Adding a top level query follows a similar pattern except top level queries always apply so we skip the first parameter. Top level queries apply no matter what the view selection is and they will always appear at the root of a view. Top queries should never contain a `!this` variable as they never apply to a specific target. The existing definitions for top level queries can be found in the `topQuery.rub` file.

The second customization to the UI is defining custom input and selection filters for view linking. In this case, we wish to define a filter with a user friendly name that will appear in the link view dialog. The `filter.rub` file contains several existing filters which are similar to this:

```
updateTargetFilter("Methods","Callable(!this)").
```

The first argument is the user friendly name and the second is the actual filter. A filter in JQuery is essentially a targeted query which is executed on the target before the query executes. In the example above, only values that are callable will be accepted and pass through the filter. If we wanted a filter which only accepts constructors, we can modify the code above to be similar to this:

```
updateTargetFilter("Constructors","constructor(!this)").
```

The third customization to the JQuery UI is to define custom icons and labels for the results view. Icons and labels are as straightforward as adding new link filters and menu items. Examples of icon and label defintions can be found in the `LabelProvider.rub` file.

In order to customize the icons which are displayed in the results view, a user must define additional rules for the `baseImage` predicate. This predicate takes one input argument and returns the name of the image for that argument. For example, a user can define a custom icon for classes with the text 'Listener' in their name by adding a rule such as (assuming there is a `listener.gif` file inside the `JQuery/icons` directory):

```
baseImage(?X,‘‘listener.gif’’) :- type(?X),re_name(?X,/Listener/).
```

To define custom labels for a result in a view, a user must define (in much the same way they define rules for icons) additional rules for the `label` predicate. This single rule generates all labels for results in the tree view. For example, to generate a label “Listener Type” for any type with the text ‘Listener’ in it’s name, we can use a rule such as:

```
label(?X,‘‘Listener Type’’) :- type(?X),re_name(?X,/Listener/).
```

One final note is that the TyRuBa language requires a mode definition for predicates before they can be used. We have managed to avoid mode definitions here because we are using predicates that have already been defined. We leave the discussion of these definitions to [11] but will provide an example of a mode definition and instructions on how to add new configuration files in the next section.

2.3.4 Creating new Rules

In this section we will show an advanced user how to add new configuration files to the existing JQuery factbase. These configuration files will enable the user to make project specific predicates that can be used to create JQueryScapes. Adding new configuration files to JQuery enables an advanced user to truly take advantage of the expressive power of the JQuery query language.

Additional files must be added to the list of configuration files loaded by JQuery. We add these files through the preferences dialog in Eclipse by going to Preferences/JQuery/Tyruba Backend and adding the configuration file to the list. After clicking **OK** or **Apply** the configuration files will be reloaded and the custom configuration will be enabled. If there are any syntax errors in the configuration file they will be caught and displayed when the files are loaded.

After successfully adding a new configuration file, we’re finally able to define predicates for project specific concerns in the code. This can be as easy as defining a predicate to match a naming convention or a certain type of annotation. We will give examples of how to achieve this kind of customization by defining logic rules in JQuery.

The advanced user is able to extend the representation of the Java code inside the factbase by adding logic rules. For example, we can define a predicate `listener`, that will return the set of all types that are a part of a listener pattern. As mentioned at the end of Section 2.3.3, we need to define

a mode for this predicate. The rule and mode definition might be similar to this:

```
listener :: Element
MODES
    (FREE) IS NONDET
END

listener(?X) :-
    type(?X),subtype+(?X,?Y), name(?Y,Listener),Type(?Y) ;
    type(?X),re_name(?X,/Listener/).
```

The mode definition states that this predicate has one argument, an `Element`, and that it supports that argument being an unbound variable. We do not cover the specifics of this mode definition here but instead refer the interested reader to the TyRuBa documentation [11].

The `Listener` rule will return all classes that are subtypes of the Listener interface *or* have the word 'Listener' in their name. Rules can be defined for many different high level concerns in the code inlcuding annotations as we will see in Chapter 5. In Chapter 5 we use logic rules similar to the listener rule above to define crosscutting subjects based on annotations in the code.

Anytime we add a rule similar to the `listener` rule we should add support for the auto-complete feature of the query editor. To do this, we have to add an additional `pasteSnippet` predicate. For the `listener` rule it may look something like this:

```
pasteSnippet("listener(?X)", "Binds ?X to a type in the Listener pattern").
```

In this section we have seen an introduction to configuring the JQuery factbase. An advanced user can define arbitrary high level concerns using rules and then develop a JQuery UI to match the information they provide. Everything from icons and labels to the queries that are displayed on the menu are configurable through the logic language driving the JQuery UI.

Advanced users can define any number of custom rules. These rules can then be combined and written into queries in the UI to answer questions and create perspectives based on project specific concerns. The work in this section should serve as an introduction to important concepts that will be used when we validate the second claim of this dissertation, that an advanced JQuery user can build collections of code views (perspectives), that take advantage of project specific naming conventions and/or project specific annotations, with less work than building perspectives in current integrated development environments, in Chapter 5.

2.4 Summary

In this chapter we have introduced the JQuery UI and have shown examples of how to customize the JQuery UI and the JQuery factbase for project specific concerns. This chapter serves as an overview of the features in JQuery appropriate for understanding how JQuery can combine results from multiple queries and how JQuery can create perspectives designed for project specific concerns.

Section 2.2 introduced several key JQuery UI mechanisms that will help when we validate our first claim in Chapter 4, JQuery users can answer more complex queries through the UI than with current IDEs.

Section 2.3 introduced several key JQuery configuration mechanisms that are important to understand when we validate our second claim in Chapter 5, that an advanced JQuery user can build collections of code views (perspectives), that take advantage of project specific naming conventions and/or project specific annotations, with less work than building perspectives in current integrated development environments.

Chapter 3

JQuery Implementation

In this chapter we will describe in detail our implementation of the JQuery tool. For completeness, we will also provide a brief technical comparison between JQuery 3.14 [16] and our JQuery 4.

JQuery 3.14 is an Eclipse plugin which served as the foundation on which we built JQuery 4. JQuery 3.14 consists of a tree view which generates content by executing logic queries against a database of code facts. JQuery 3.14 is responsible for generating the factbase, providing queries for the user to execute, and a tree view to display results to queries.

JQuery 4 performs the same tasks as JQuery 3.14 and is responsible for generating a factbase, providing queries for the user to execute, and providing a tree view to display query results. The factbase structure of JQuery 4 remains relatively unchanged from version 3.14 and the majority of our work has been focused on redesigning the user interface as described in Chapter 2 and dividing JQuery 3.14 into two plugins with an API for communication.

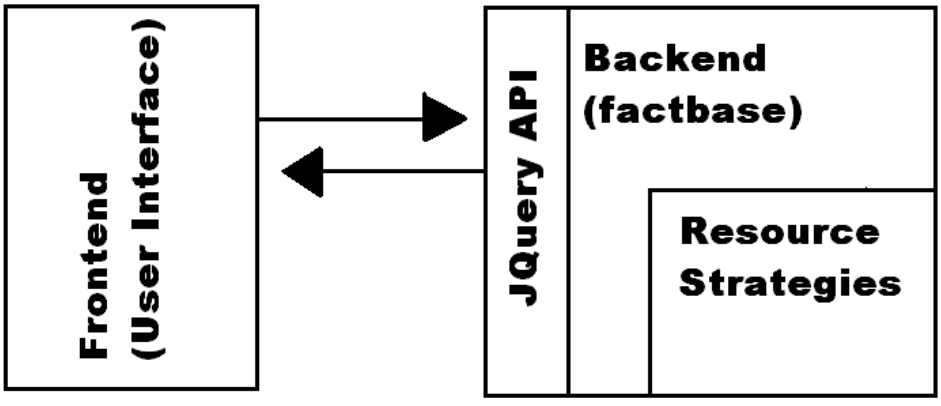


Figure 3.1: Overall design of JQuery.

JQuery 4 is now refactored into two separate plugins, a frontend and a backend, as shown in Figure 3.1. From a users perspective, these two

plugins represent a divide between the JQuery factbase and the JQuery user interface. In order to achieve this separation, we implemented an API to communicate between the plugins.

The frontend is primarily the JQuery tree view which answers questions for the user. The frontend gives JQuery its ability to execute a wide variety of queries through the UI. Section 3.1 will cover important implementation details of the JQuery user interface. For information on using the JQuery UI and any of its features, please refer to Chapter 2.

The JQuery backend provides an implementation of the public API on top of a database. The backend provides a set of queries that the frontend executes to generate its views. Plugins which use the JQuery API can take advantage of the fact that all information will be in the same format, allowing for a rich set of interactions between JQuery views as we saw in Section 2.2. Section 3.2 will describe how to use the JQuery API as an information source for other plugins and Section 3.3 will discuss how to implement the JQuery API on top of a database backend.

The backend also contains a skeleton framework for resource extensions which permit a developer to add their own custom facts to the JQuery database. Section 3.4 will look at resource extensions to the JQuery backend and explain how to use this skeleton framework.

3.1 Frontend

In this section we will only describe the improvements made to the JQuery UI since the 3.14 release of JQuery. We will not describe in detail the features the UI offers as these have already been covered in Section 2.1. As a reminder, we have contributed the ability for a user to create multiple JQuery tree views, the ability to compose results from different queries through drag and drop and selection linking, and we have added a regular expression filter to every JQuery view.

In addition to new UI features, our JQuery implementation of the tree view has undergone several performance improvements since JQuery 3.14. We have moved several computations to separate threads to prevent the UI from being disrupted. Below is a list of tasks that are now performed in the background so as not to disrupt the users experience:

- Factbase updates
- Query execution
- Result tree building

- Result node labels
- Result node icons

It is important to note that scheduling this number of threads adds a performance penalty to the overall execution of queries but we feel that most users will not notice the penalty because the editing experience is not disrupted. In fact, these changes have caused a noticeable improvement in responsiveness when building the results tree because we no longer need to wait for icons and labels to be generated before we display results.

Beyond performance improvements we have also included a number of new features to improve the integration between Eclipse and JQuery. We have enabled the Eclipse context menu for any JQuery result node. This means that JQuery users now have access to Eclipses' powerful team features and refactoring tools. We have also added "show-in" views for items from Eclipse views. The user selects an item from an Eclipse view and is now able to create a JQuery view with that item so they can begin their queries from that starting point. This allows users to create JQuery views from anywhere inside their Eclipse workbench.

3.2 Using the JQuery API

This section will serve as a guide to developers interested in using the JQuery backend to build their own UI plugins. Currently, we have only built the JQuery tree view on top of the API but in this section will explore several code samples taken from the JQuery tree view and discuss how they can be used to build a custom UI plugin.

To gain access to the JQuery factbase, an Eclipse plugin must depend on the `ca.ubc.jquery.backend` plugin. After this dependency has been added the developer will have access to any of the classes defined inside the `ca.ubc.jquery.api` package which provide direct access to the JQuery factbase.

Access to the JQuery API classes occurs primarily through calls to `public static` methods in the `JQueryAPI` class. For the developer, having the API as a series of static calls greatly simplifies usage because they do not need to initialize the API or pass an API object from method to method. Most methods in the API create or execute queries while the methods in Table 3.1 provide access to configuration options.

Below we provide three examples for using the JQuery API and an overview of JQuery update targets. The first two examples illustrate how

Table 3.1: Configuration methods in the JQuery API.

Method	Description
`void installDefinition(File)`	Adds the given definition file to the database.
`void removeDefinition(File)`	Removes the given file from the database defintions.
`void installResource(String, JQueryResourceStrategy)`	Installs the given resource strategy under the given name. This forces the defintion files to be reloaded.
`void removeResource(String)`	Removes the named resource strategy from the backend. This forces the defintion files to be reloaded.
`JQueryFactBase getFactBase()`	Returns the current active factbase. Factbases have other methods to assist with reloaded defintions files and forcing a refresh of the entire factbase.
`JQueryFactBase selectFactBase()`	Causes a dialog to appear to assist a user to select which factbase they wish JQuery to use.

to execute queries using two different methods. The last example illustrates how to use JQuery to build up a context menu by using the menu query. The overview of update targets provides insight into the part of the backend that allows views to update automatically as we saw with view linking and drag and drop (Section 2.2.4 and Section 2.2.3). Together these descriptions should provide most developers with sufficient detail to being working with the JQuery API.

There are two methods for executing a query in JQuery. Both methods execute the same query and return the same results but have different performance specifications, and in some situations a developer may prefer to use one method over the other.

The first method is a straight query execution where the developer specifies an order for the query variables and each result contains a binding for the specified variables. This method has the advantage of giving the developer complete control over how the results are represented and accessed.

The code snippet in Figure 3.2 executes a query using the first method. Initially we create an empty `JQueryResultSet` to capture the results from the query. We create the query using the API and then execute it but because

```
JQueryResultSet rs = null;
try {
    JQuery q = JQueryAPI.createQuery("method(?m)");
    rs = q.execute();

    while (rs.hasNext()) {
        JQueryResult r = rs.next();
        System.out.println("method: " + r.get("?m"));
    }
} catch (JQueryException ex) {
    ex.printStackTrace();
} finally {
    if (rs != null) {
        rs.close();
    }
}
```

Figure 3.2: Sample code for creating and executing a query using the JQuery API.

we have not set any chosen variables, the default behaviour is used and all variables in the query will be selected. The `execute()` method returns a `JQueryResultSet` which will iterate through the solutions to the query. In some cases, we may not want to visit all results but still need to notify the query engine that the result iterator is no longer needed thus we have to call `close()` anytime we generate a `JQueryResultSet`. Each item in the `JQueryResultSet` is a `JQueryResult` from which we can request the values of variables directly and display their results. In this case, the query will simply print the factbase representation for each method declared in the system.

The second method of query execution returns a graph representation of the results. The order of the variables specified determines the structure of the graph that is generated. The developer traverses the graph by accessing the neighbours of each node. The graph result method returns all results for a variable in a single call which provides a higher level structure on top of the `execute()` method.

The code snippet in Figure 3.3 illustrates how to obtain a graph of the results from a query. The above code will print the factbase representation of each method declared in the system with a tree-like indentation structure (in this case, only one level of indentation).

Figure 3.4 clarifies the difference between the two query methods we have provided. In Figure 3.4(a) we can see that each result contains a binding for all the variables at once. In Figure 3.4(b) we instead receive all the values for a particular variable with each call to `getNeighbours()` and then

```
try {
    JQuery q = JQueryAPI.createQuery("method(?m)");
    JQueryResultGraph g = q.getGraph();

    JQueryResultNode[] x = g.getNeighbours();
    for (int i = 0; i < x.length; i++) {
        printNeighbours(x[i], 1);
    }
} catch (JQueryException ex) {
    ex.printStackTrace();
}

private void printNeighbours(JQueryResultNode n, int indent) {
    JQueryResultNode[] x = n.getNeighbours();
    System.out.println(n);

    for (int i = 0; i < x.length; i++) {
        for (int j = 0; j < indent; j++) {
            System.out.print("  ");
        }
        printNeighbours(x[i], count + 1);
    }
}
```

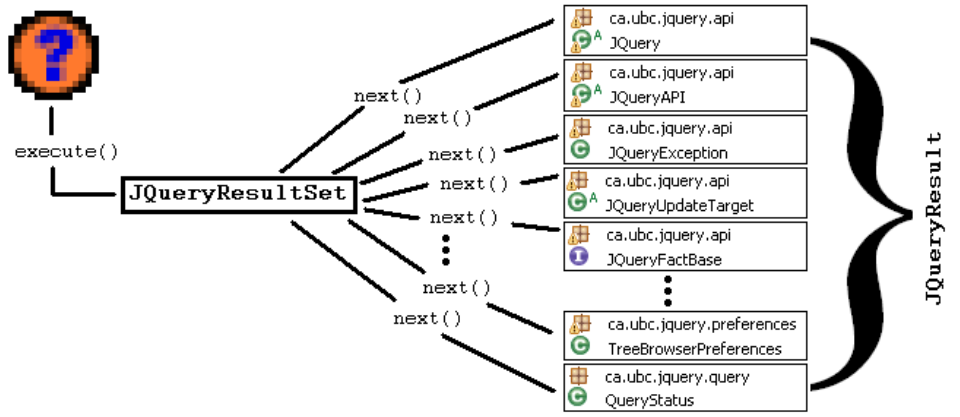
Figure 3.3: Sample code for creating a graph of query results using the JQuery API.

we can request the neighbours of that node. In both cases, the ordering of the results depends on the ordering of the variables specified before the query was executed.

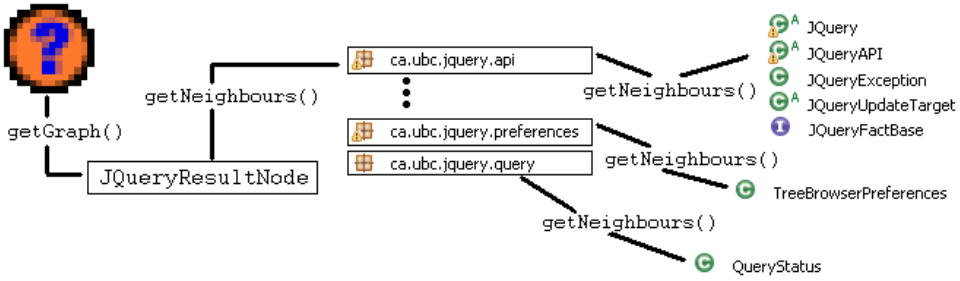
Generating a menu in JQuery requires another query execution technique. The menu query returns an iterator with a fixed variable order. All menu queries in JQuery return a consistent set of variables so we have designed this iterator to contain methods which access each of those variables. There are methods to access the menu path, the menu query, and the chosen variables.

The code snippet in Figure 3.5 can be used to get a list of possible queries available for a given target set of results. This is the query that is used to generate the context menus for the JQuery tree view. In this case, `targets` is of type `Object[]` and represents a users selection from the tree view. The `JQueryMenuResults` is also an iterator but `next()` does not return an element because the `JQueryMenuResults` object itself represents the current result. In this example, we first obtain the queries that are relevant to the selected node and then add them to the context menu.

The last important detail for a developer is update targets. Update



(a) Visualization of `execute()` method



(b) Visualization of `getGraph()` method

Figure 3.4: Execution of the JQuery package browser query with default variable ordering `?P,?T` using both query execution methods.

```
try {
    JQueryMenuResults rs = JQueryAPI.menuQuery(targets);
    for (; rs.hasNext(); rs.next()) {
        String[] path = rs.getPath();
        String query = rs.getQuery();
        String[] vars = rs.getChosenVariables();

        Action action = new DoSubQueryAction(view, path[path.length - 1],
                                              query, vars);
        createPath(menu, path, action, GROUP_NODE_SPECIFIC);
    }
} catch (JQueryException e) {
    JQueryTreeBrowserPlugin.error("Error occurred while retrieving " +
            "available sub-queries: ", e);
}
```

Figure 3.5: Sample code for generating a context menu using the JQuery API.

targets provide a way for developers of different views to link their query targets to a commonly accessible source. As a central point of access, update targets are defined in the API which makes all update targets available to every plugin which uses the JQuery API.

The API methods `createUpdateTarget(String)` and `getUpdateTarget(String)` are used to create an update target and to request an existing update target respectively. After obtaining an update target, a developer can implement the `JQueryEventListener` interface and use the API method `addListener(JQueryEventListener)` to listen for updates to a target. Anytime an the update target `setTarget(Object)` method is called, the listeners will be notified so that they can respond accordingly.

The update target method is what drives the view linking feature described in Section 2.2.4. Every view contains an update target which is updated whenever a selection changes. When a view is linked, it means that it is listening for updates on a specific update target.

In this section we have covered three examples and an overview of the `JQueryUpdateTarget` class to provide a developer a way to access the JQuery backend through the API. For these examples, we have used the default TyRuBa backend but the actual details of the query language and variable syntax depends on the particular backend being used. The Java code we have provided however will remain the same (except for portions with query strings) regardless of the API implementation being used.

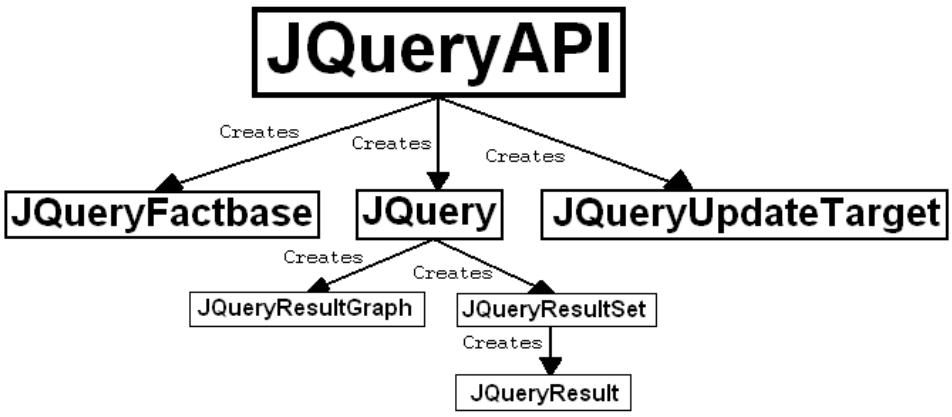


Figure 3.6: Overview of key JQuery API classes.

3.3 Defining the JQuery API

In this section we will describe the JQuery backend from a developers point of view and describe how to implement a custom backend using the JQuery API. The JQuery backend provides a single common interface on top of a database. The JQuery tree view uses the API to generate its views and we have designed the API to provide information for other UI plugins. We have also designed the JQuery API to be a wrapper around different database implementations such that users can use a different database depending on their individual needs while still using the powerful JQuery UI.

In Figure 3.6 we have provided an overview of the JQuery backend structure and layout. The `JQueryAPI` class provides most of the functionality, including a set of factory methods for other API objects and acts as a central access point to the backend. Other important classes include the `JQueryFactBase` to control the dataset, the `JQueryUpdateTarget` as a way to link query results, and the `JQuery` class to create and execute queries. We will look at these three classes in detail.

3.3.1 `JQueryAPI` class

The `JQueryAPI` class is responsible for all access to the API and provides a series of `public static` methods for access to the database and creation of other API classes. To implement a custom backend all that is required is to create an Eclipse extension point as shown in Figure 3.7 and subclass the

```
<extension
      id="ca.ubc.jquery.backend.api.tyruba"
      name="Tyruba backend"
      point="ca.ubc.jquery.backend.factbase">
   <API
         JQueryAPI="ca.ubc.jquery.api.tyruba.JQueryTyRuBaAPI">
   </API>
</extension>
```

Figure 3.7: Eclipse extension point declaration for the TyRuBa JQuery backend.

`JQueryAPI` class.

The `JQueryAPI` class contains a set of factory methods to create other API objects. This means that other classes in the API are required to be subclassed because the constructors are `protected` to keep them from being initialized outside of the `JQueryAPI` class. The classes that need to be subclassed will become obvious as the API methods are implemented and instances of these classes need to be created for the custom backend. Most developers will need to subclass `JQuery`, `JQueryResultSet`, `JQueryResult`, `JQueryFactBase`, `JQueryException`, and `JQueryUpdateTarget`.

The JQuery API object needs to be fully initialized by its constructor such that a user can make calls to the `public static` methods. To achieve this a default database should always be initialized in case the backend plugin is unable to restore the database from a previous JQuery session. The default database need not store any facts but acts as a place holder until the user decides on a database.

The backend provides direct access to the query language but a few specific queries must be implemented in order to complete the API definition. Some of these queries are specifically implemented as methods in the API to make designing new user interface plugins less dependent on the actual query language. A backend must supply definitions for all the queries in Table 3.2 for the current tree view frontend to function properly.

Currently we have implemented two versions of the JQuery API on top of two databases. One uses TyRuBa [13] for it's query language and factbase, and the other uses JTransformer [19] described in detail in Appendix A. Because both backends are wrapped by the JQuery API, they are both capable of driving the JQuery tree view. From the perspective of the first time user they would be indistinguishable because both store similar facts and have the same queries accessible from the menu. Only a user who is familiar with the query language would detect the difference.

Table 3.2: Queries available in the JQuery API as required by the tree view frontend. All of these queries are available in both the TyRuBa backend and JTransformer backend.

Query	**API Method**	**Description**
Top Query	`topLevelQuery()`	Returns the set of top queries.
Context Menu Query	`menuQuery(Object[])`	Returns the set of context sensitive menu queries.
Filter Menu Query	`filterQuery(Object[])`	Returns the set of context sensitive menu filters.
Predicates Query	`queryPredicates()`	Returns the set of predicates defined in the API.
View Filter Query	`(new JQueryUpdateTarget()) .getFilters()`	Returns the set of input and selection filters for view linking.
Location from Item	`getFileElement(Object)`	Returns the source location for the given item.
Item from File	`getElementFromFile( String,int,int, Set Context,Set Element)`	Returns the source code objects from the given file, offset and length. The source context and source elements are returned as parameters.
Label Query	`getElementLabel(Object)`	Returns a string representing the user friendly label for the given item.
Image Query	`getElementImage(Object)`	Returns an image for the given item.
Element Type Query	`getElementType(Object)`	Returns the type of the given item as a string.
Category Query	`getIntProperty( Object,‘‘category’’)`	Returns an int to sort elements by type inside the tree view.
Return Type Name	`getStringProperty( Object, ‘‘returnTypeName’’)`	Returns the type name of the return value of the given method as a string.
Field Type Name	`getStringProperty( Object, ‘‘fieldTypeName’’)`	Returns type name of the given field as a string.

3.3.2 `JQueryFactbase` class

The `JQueryFactbase` class represents a particular set of facts that the user can query. In general, every instance of the JQuery API will have to manage multiple factbases. For example, the TyRuBa backend uses a factbase for each combination of working sets. The factbase class is also responsible for managing the configuration files that are loaded.

3.3.3 `JQuery` class

The abstract `JQuery` class is responsible for executing queries against the database and preparing result iterators for the user. The task of executing queries is not trivial as it partially requires managing the results before they arrive. A developer must specify the order in which the results will be returned to provide space for optimization in the query. In this version of JQuery we have added support for recursive queries hanlded through the `getGraph()` method of the `JQuery` class instead of the `execute()` method.

We have already described how to use the two methods of executing a query in Section 3.2. Here we will discuss how to implement the `execute()` and `getGraph()` method as illustrated by Figure 3.4.

Calling the `execute()` method executes the query and the `JQuery` class creates a `JQueryResultSet` to provide results to the user as shown in Figure 3.4(a). The `JQueryResultSet` acts as an iterator and should only generate results as needed. Each result in the iterator is a `JQueryResult` which provides accessor methods to variables, by name, from the set of query results. Not all variables need to be accessed but a developer needs to be able to access multiple JQueryResults at a time so they must at least have knowledge of the value for each variable chosen from the query. When a user is finished with the `JQueryResultSet` they must call `close()` to notify the query engine that they are finished with the results.

It is important to note that recursion is not supported by this method of execution. The query class supports setting one variable as the recursive variable but it is the responsibility of the developer to repeat the execution process to achieve recursion. The `getGraph()` method, on the other hand, does handle recursion.

Fortunately the graph structures are written in a generic format and make use of the `execute()` method from the `JQuery` class. This way any implementation of the backend should already have a working version of the result graph classes. A developer is welcome to implement their own custom subclasses of these classes to optimize the process for their backend but the

```
<extension
        id="ca.ubc.jquery.resource.javafiles"
        name="JavaFiles"
        point="ca.ubc.jquery.backend.resource">
    <Definition
        File="rules/initfile.rub">
    </Definition>
    <Resource
        JQueryResource="ca.ubc.jquery.engine.tyruba.java.JavaFileStrategy">
    </Resource>
</extension>
```

Figure 3.8: Sample resource extension point definition for the JQuery backend.

default implementation should suffice.

The above description of the JQuery API and associated classes should serve as a guide to implementing a backend in JQuery. We have covered the `JQueryAPI` class, the `JQueryFactbase` class, and the `JQuery` class with its two methods of execution. The next section will discuss several other classes that are still a part of the JQuery API but are responsible for generating the factbase rather than finding results.

3.4 Extensions

This section will discuss how to develop a resource extension for the JQuery backend. We have already seen how to implement a custom JQuery backend in Section 3.3 and will focus this section on extensions to the backend.

The JQuery backend can be extended by adding support for additional resources. Figure 3.1 shows that the resources are actually inside the backend and thus depend on the database used. In the future, we intend to generalize this mechanism and provide a more modular structure for resource extensions so that a particular extension will work on all backends by passing through another section of the JQuery API. For now we will only look at how to define resource extensions for the TyRuBa backend.

Implementing a resource requires creating an Eclipse plugin and extending the `ca.ubc.jquery.resource` extension point as seen in Figure 3.8. The `<Resource>` portion of this definition identifies the `JQueryResourceStrategy` class which is the class that manages the resource. The `<Definition>` portion of the XML file specifies any additional configuration files the resource strategy may need to add new predicates or rules associated with the new facts.

The `JQueryResourceStrategy` class defines a directory for any needed images, identifies files to be parsed, and generates a parser for each file. A resource strategy may require new icons to display in the results tree. The resource strategy has a method to return a path to additional images for the frontend. The rest of the methods in this class are implemented in a straightforward manner.

Once the methods are implemented, the JQuery API will take each working set and pass them through the resource strategy to identify relevant files. Once all the files have been identified, the API will request a parser from the resource strategy for each file it found useful. The `makeParser(IAdaptable, JQueryResourceManager)` method creates a resource parser object for the given adaptable and uses the `JQueryResourceManager` to add any dependencies the parser may identify.

The `JQueryResourceParser` class handles generating the facts and putting them into the JQuery database. Each parser has a name which generally provides some representation for the resource which it generates facts from. Resource parsers also contain a `parse()` method which is called by the TyRuBa backend to generate the facts for the particular resource.

3.4.1 XML Extension

In order to test the framework we developed an XML resource strategy for the TyRuBa backend. The actual implementation of the XML resource stategy is around 300 lines of Java code [1] and an additional 60 lines of TyRuBa code [2] to generate menu structures, labels, and icons for Eclipse `plugin.xml` files and ant build files.

An important reason why the XML extension can remain small is that Java and Eclipse already have good support for parsing XML. We implemented enough rules to provide an outline similar to what Eclipse provides for Ant build files and an outline similar to what Eclipse provides for plugin definition files. As far as we could tell, the Eclipse Outline view is the only view that has special handling for the XML files and we were able to create a similar view within hours of creating the XML parsing extension for JQuery.

[1] Including comments and whitespace.
[2] Including comments and whitespace.

3.5 Summary

In this chapter we have covered the details of the JQuery implementation. We have provided details so that the interested developer is able to create their own UI plugins using the JQuery factbase. We have also provided a detailed overview of the classes inside the JQuery API such that a developer may implement this API on top of their own database. This chapter is intended to be used as a reference for the interested developer and is here for completeness of the work. The next two chapters will validate the claims made in Chapter 1.

Chapter 4

Complex Queries

In this chapter, we validate our first claim. That is, we will show that JQuery users can answer more complex queries through the UI than with current IDEs. To validate this claim, we will make use of the JQuery UI as discussed in Section 2.2 and show how it is able to answer a few sample complex queries. We will be using Eclipse as representative of current IDEs because Eclipse provides a large number of code views for Java and good support for customizing the questions those views answer. Similar arguments would apply to other IDEs.

Recall that in Chapter 1 we discussed how modern IDEs are unable to combine information between views. Without a way to combine information from different views, the developer is forced to mentally combine that information which is hard [10]. This is an important point because it can always be argued that Eclipse, and most other IDEs, offer a plugin framework to allow a developer to build their own code views. These code views could, in theory, answer a question which would normally require combining information between two or more existing views.

For the sake of simplicity, we will ignore plugin extensions to Eclipse for two reasons: (i) the high cost of creating an extension to an IDE versus defining a view in an expressive query language [32] and (ii) even if there was a viewing extension for a particular query, a user may want to combine information from this view with the information from other existing views to answer a new query. Since users cannot combine information from existing views, there will always be questions that cannot be answered.

The remainder of this chapter is divided into two sections. Section 4.1 will provide an overview of how JQuery is able to combine results from multiple queries by explaining the UI query semantics. Section 4.2 will provide several examples of complex queries that JQuery is able to answer.

4.1 Query Semantics

For the sake of completeness, this section will provide a description of the JQuery UI query semantics. This is to ensure that the reader, who may

never have used JQuery, is aware of what kinds of complex queries JQuery is able to answer.

JQuery queries produce a result which is a collection of objects that represent various code artifacts such as classes, methods, fields, etc. Recall from Section 2.2 that a there are two kinds of queries: queries with an argument called the "target" and queries without an argument that are instead applied to the entire factbase. For example, a query without a target could produce the set of all classes defined in the current project or the set of all classes whose names match a user-supplied regular expression. An example of a targeted query, a user could select a particular class as a target, and get the set of methods defined in that class as a result.

Earlier versions of JQuery, like most other code browsing tools, applied queries to individual targets. By extending JQuery to support multiple views, a user can now apply a query to a collection of results from previous queries. This enables a user to perform queries that are not possible in existing code browsing tools.

Using the JQuery interface, a user can select one or more result objects (e.g. classes or methods) from one or more previous queries. Conceptually, the JQuery UI provides a buffer of selected objects, and the user can set, clear, or add to this buffer with simple point-and-click operations. The contents of this buffer can then be "dropped" onto an exisiting query view (i.e. a window created by JQuery to display query results). When this is done, the contents of the buffer replace the previous target of the query, and the query is performed on each object in the the buffer. The result of the query is the union of the results for each object in the buffer, and JQuery orders these results and displays them in a tree-like fashion just as it would for a simple query. Thus, queries such as "what classes call methods in class X?" can be answered by first performing a query to find the methods defined in class X; then selecting these methods; and finally dropping these methods into a query view for the query "what classes call this method?". We describe this process in greater detail below by describing several queries that are representative of queries that a real programmer might want to perform that can be done easily in JQuery but that are not supported by other modern IDEs such as Eclipse.

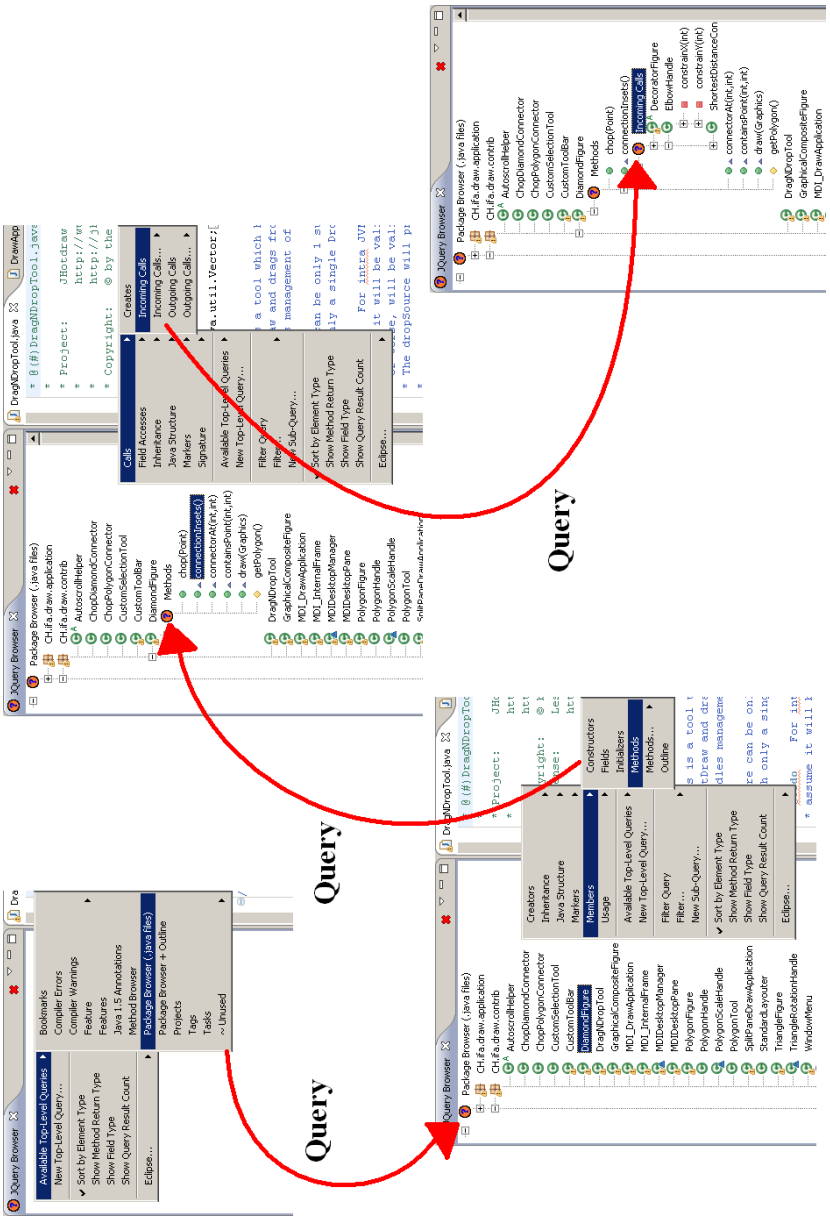


Figure 4.1: How to execute several queries in JQuery. Every query we execute is represented as an orange query node in the tree view.

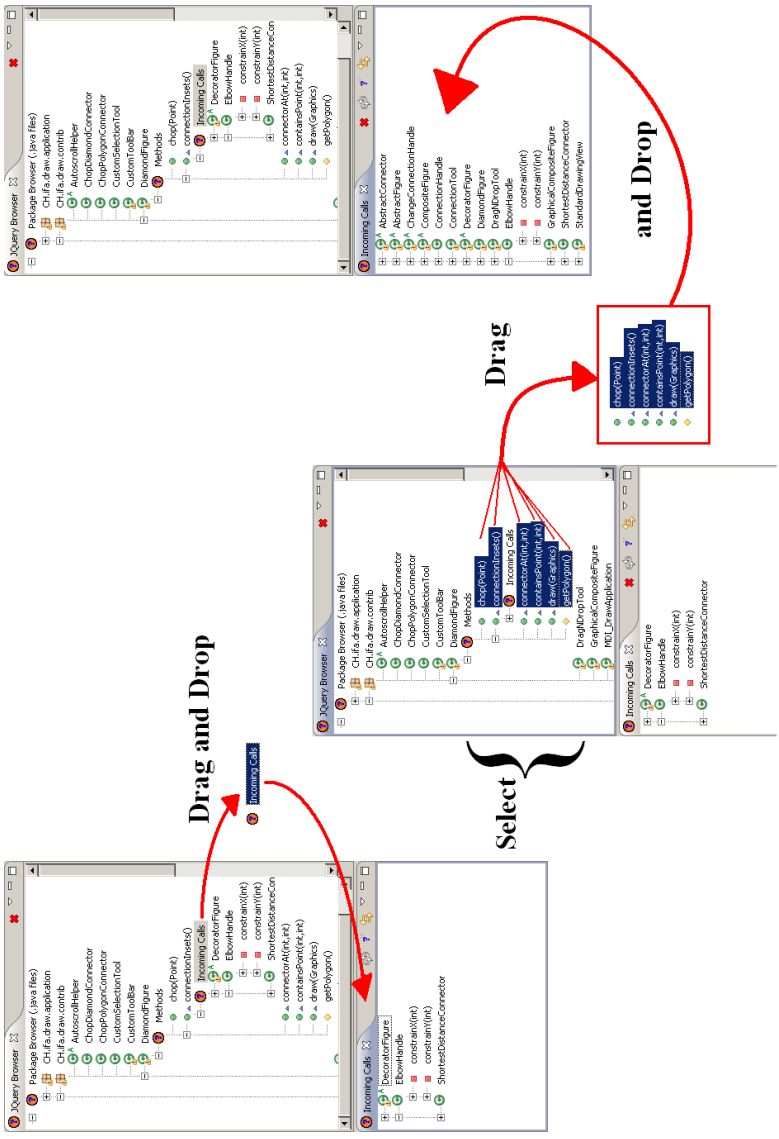


Figure 4.2: We combine query results by dragging result nodes from any other JQuery view and dropping them onto a query-rooted view, in this case, the incoming calls view. The incoming calls view now displays the union of the incoming calls to the dropped set of methods.

4.2 Validation

In this section we present four examples of complex queries and explain how to answer each query using JQuery's UI. We have illustrated the solution to the first question to help clarify the process. These queries are based on the questions modern IDEs were not able to answer as identified in [2] and [27]. For each example query we also provide a description of why our representative modern IDE, Eclipse, is unable to answer the query.

4.2.1 Who calls this class?

To answer this first query, we will assume the user starts with nothing but a single, empty JQuery view. The remainder of the questions in this chapter will begin with some information already displayed in the JQuery view.

The user is first required to pick the class which they want to see the calls to. In JQuery, we can see all the classes, sorted by package, by right-clicking on an empty view and chosing the Package Browser Top Level query as shown in the top left corner of Figure 4.1. An orange query node will appear with the name of the query and the results tree can be expanded and navigated. These orange query nodes represent a query over the JQuery factbase and can be edited if needed. The predefined queries built into the JQuery menu system cover the queries most Eclipse users execute thus we will not need to actually edit a query in this example. Instead, we navigate through the result nodes in the tree and find the class we wish to see incoming calls to.

Once the class result node has been found, we can query for the methods of that class by selecting the node, right-clicking to open the context menu, and selecting the methods query as shown in the bottom left corner of Figure 4.1. An orange methods query node appears and nested below this query node are the result nodes representing the methods this class defines. The last query we need to execute is the incoming calls query.

To query for the incoming calls to a method, we select a method result nodes, right-click and choose the incoming calls query as shown in the top right corner of Figure 4.1. As before, an orange query node appears to represent the query we selected and similarly the results to the query are nested in the tree structure below the query node. The final result to all this querying is displayed in the bottom right corner of Figure 4.1.

So far what we have seen is not unlike what Eclipse can do. The difference is that in Eclipse, the information we have queried for (classes, methods, calls) would be displayed in separate views. One view for classes and meth-

ods, and one view for call structure. In JQuery, we have combined each of these queries into a single view but in order to answer the above question, we need to combine results from the methods query and the incoming calls query so we will actually need to use multiple views. In this case, we want to see all the incoming calls to the set of methods that is displayed.

To create a view in JQuery, all that is required is to select a query node, and drag and drop that node onto the Eclipse workbench and the new query-rooted view will appear. The view will have the query that was dragged positioned at the root of the view. The real benefit from this approach is that if the query at the root has a parameter, we can set the value of that parameter by dragging one or more nodes from a JQuery view and dropping them onto the query-rooted view.

In our example, we drag the incoming calls query node to create an incoming calls view as shown in the top left of Figure 4.2. The parameter for the view in this case, is the method we are displaying the calls to. To answer the first complex query, "Who calls this class?" we simply have to set the parameter for the query-rooted view to be the set of methods the class defines. By performing this drop operation, the incoming calls query defining the view is executed on each method from the set and the view displays the union of each set of results.

In the middle of the example in Figure 4.2, we select the set of methods the `DiamondFigure` class defines and drop them onto the incomings calls view. The far right side of Figure 4.2 shows the results of this drop operation and the answer to the first complex query.

As we have already mentioned, in Eclipse, we are not able to answer this query. The Call Hierarchy view, which displays incoming calls, can only display the the incoming calls to a single method at a time. We could actually open the Call Hierarchy view on each method individually but the user is required to mentally combine the results.

4.2.2 What methods does this type hierarchy define?

The second query can also be answered by combining information between JQuery views. We start by selecting a type from the package browser, open the context menu and select the Hierarchy View query to see a type hierarchy for that type. We then select a type from the hierarchy and select the methods query from the context menu and drag the new query node to create a query-rooted view. Now we select all types defined in the hierarchy view and drop them onto the methods query-rooted view and we will see all the methods defined in this type hierarchy.

In Eclipse however, we are unable to answer this query. We can use the Type Hierarchy view to display the type hierarchy for a single class which conveniently displays the methods defined for the selected class, but it does not allow us to select multiple classes.

4.2.3 What methods return instances of these types?

The third query can also be answered by combining information between JQuery views. For this query, we apply the existing JQuery Return Type query by selecting it from the context menu. We use the Return Type query node to create a query-rooted view by dropping the node onto our Eclipse workbench. Finally, we drop the set of types we wish to query onto that view and are able to see all methods that return instances of those types thus answering the original query.

In Eclipse however, we are unable to answer this query. We start by querying the entire source code for any references to a particular type. From here we have to manually search the list of references to find where a particular type was returned from a method. We have to repeat the query and inspection process for each type in the set we are interested in.

4.2.4 Are there any getter methods being called from any listener classes?

The fourth query can also be answered by combining information between JQuery views. For this query, we start by finding all classes that implement the `Listener` interface. That is, we query the `Listener` result node from a package browser and find its inheritance structure. Then we query for the methods of a class and drag that query node to create a query-rooted view. Now we select all the classes that implement `Listener` and drop them onto that view. Finally, we create a query-rooted view to display outgoing calls and drop all the methods onto that view and search the view for all methods which begin with 'get' using the search expression '`^get`'.

Like the other three queries, we are unable to answer this query in Eclipse. We need several pieces of information but Eclipse can only provide minimal support in this case. We begin a search for the classes that implement the listener interface using the Type Hierarchy viewer. The Type Hierarchy viewer outlines the classes and we are able to view call hierarchies for each method by selecting the method and opening the Call Hierarchy view. One by one we inspect the call hierarchies and manually search for method names starting with 'get'. Since the type hierarchy viewer only dis-

plays the outline for a single type at a time, we have to repeat the process for each type in the listener hierarchy.

4.3 Summary

In this chapter we have validated the first claim of this dissertation. We have shown that JQuery users can answer four specific complex queries through the UI which we are unable to answer with current IDEs.

Chapter 5

JQueryScapes

In this chapter we will validate the second claim of this thesis. We will argue that an advanced JQuery user can build perspectives, that take advantage of project specific naming conventions and/or project specific annotations, with less work than building perspectives in current IDEs. In Section 2.3 we saw how to customize JQuery and we will make use of that knowledge in this chapter.

To make our discussion more concrete, we will be comparing JQuery to Eclipse, as a representative modern IDE. We have chosen Eclipse as a representative IDE because it contains a large number of existing code views for Java and a good framework for customization.

To evaluate the work involved in both cases, ideally we would create a JQueryScape and a similar Eclipse perspective and compare the work we invested in each but this would not be a fair comparison because it depends on our knowledge of both Eclipse APIs and JQuery APIs. We could similarly perform a user study with the same intent but with so many factors involved in selecting candidates, such a study is beyond the scope of this work.

Rather than performing a direct comparison to get a sense for the amount of work in each case, we will instead consider that even a minimal Java tree view in Eclipse consists of over 100 lines of code[3] while the Eclipse Java Outline view is closer to 950 lines of code. A similar outline view in JQuery is 3 lines of code.

Not only is the code smaller, but since most Eclipse code views are tree views, we designed JQuery to build tree views and thus it provides a very focused API for this purpose. JQuery itself provides 68 logic predicates and 22 types to configure the contents of every tree view. A typical Eclipse plugin could use hundreds of API calls. For example, JQuery uses over 747 methods from 272 types to implement the framework for creating its tree views.

Given that a direct comparison between Eclipse and JQuery is beyond the scope of this work, we instead provide a case study in Section 5.1 to

[3]Lines of code for all Java plugins in this chapter were counted using the Eclipse Metrics 1.3.6 plugin avaiable at: http://metrics.sourceforge.net/

illustrate that a reasonably complex JQueryScape can in fact be created in JQuery with a small amount of work.

5.1 Case Study

In this section we presenet the JQueryScape we created for SubjectJ [5] to illustrate the amount of work involved in creating a JQueryScape. SubjectJ is an extension to the Java programming language which uses annotations to define modularization structure. We will show that JQuery provides the advanced user the ability to browse the high level subjects these annotations define through defining a JQueryScape for SubjectJ.

5.1.1 SubjectJ

SubjectJ is an extension to Java which uses annotations to define code modules. Java has a standard object-oriented modularization where related segments of code are contained inside methods inside classes inside packages. SubjectJ allows the developer to annotate related code from different classes and packages and compose them into a subject module. Subjects are often used to define a structure that crosscuts the normal class and package structure of Java.

SubjectJ consists of a set of five annotations to describe the subject modules of which two are relevant to our discussion. The `@Subject` and `@Export` annotations allow the SubjectJ user to describe the subject module and its interface. The developer adds `@Subject` annotations to all types, methods, and fields that should be a part of the subject module. `@Export` annotations are used to define the interface of the module and are similarly added to types, methods, and fields. We have added support for these two annotations to JQuery to create this JQueryScape.

Figure 5.1 shows what the SubjectJ annotations look like inside the Java code. The SubjectJ developer can annotate Java elements, such as types, methods, or fields, with a `@Subject` annotation to tell the SubjectJ compiler that this element belongs to one or more subjects. Any code elements that are not annotated as a part of a particular subject are implicitly placed in the `None` subject.

To add support for these two SubjectJ annotations we needed to add only one additional configuration file to JQuery. Because SubjectJ uses the same syntax as Java with some additional annotations, we needed only to add semantics to the annotations and then we can define views and perspectives using the information the annotations provide.

```
import annotations.Subject;

@Subject( { "Refactor", "Modularize" })
public class Remod {
    @Subject( { "Refactor" })
    public int classCount;
    @Subject( { "Modularize" })
    public int moduleCount;

    public void save() {}

    @Export( { "Modularize" } )
    public void open() {}
}
```

Figure 5.1: Sample SubjectJ source code.

5.1.2 SubjectJ and JQuery

To add SubjectJ support to JQuery, we need JQuery to understand the concept of a subject and how subjects are defined. We wrote a custom configuration file and added it to the JQuery configuration files to allow querying over SubjectJ subjects. First we needed to actually define what a subject was and find the code elements that belong to a subject. To achieve these goals, we wrote the following three rules:

```
subjectMarked(?S,?CodeElement) :-
    hasAnnotation(?CodeElement,?A),name(?A,Subject),
    attribute(?A,value,?S),String(?S).

subject(?S,?CodeElement) :-
    subjectMarked(?S,?CodeElement).
subject(?S,?CodeElement) :-
    (method(?CodeElement);field(?CodeElement);constructor(?CodeElement)),
    NOT( subjectMarked(?,?CodeElement) ),equals(?S,"None").

subject(?S) :- subject(?S,?).
```

From here we are able to define higher level structure on top of the subjects. In particular, SubjectJ annotations must follow certain design rules. For example, subjects are not supposed to reference any fields or methods outside their subject unless they are exported to that subject. We can define an error rule displaying violoations to this constraint as follows:

```
depends(?X,?Y,?Loc) :-
    calls(?X,?Y,?Loc);
    accesses(?X,?Y,?Loc);
    overrides(?X,?Y),sourceLocation(?X,?Loc).
```

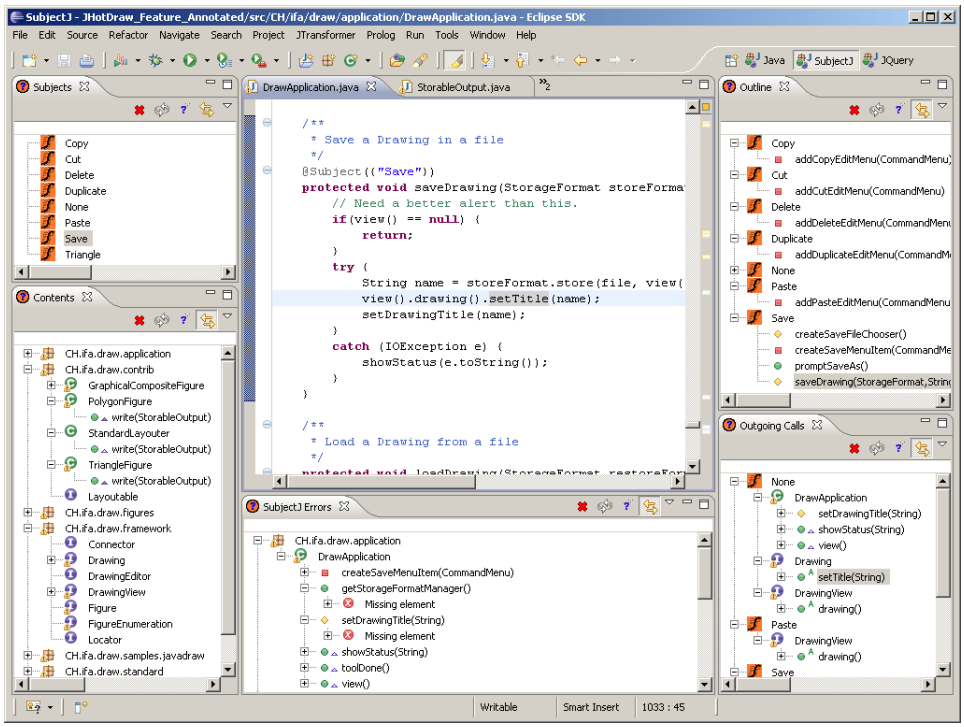


Figure 5.2: A JQueryScape for browsing subjects in JHotDraw.

```
subjectError(?S, "Missing element", ?missing, [?loc]) :-
    subject(?S,?E), depends(?E,?missing,?loc),
    NOT( subject(?S,?missing);export(?missing,?F) ).
```

The last customizations to JQuery applied to the UI. We modified the UI to support querying over subjects and added menu items specific to SubjectJ. For example, we wrote a menu item to query elements of a subject. We also modified the rule which generates icons to display icons for subjects when they appear as a result in the tree view.

The single TyRuBa configuration file we added consists of 36 rules and around 210 lines of code[4]. With our menus defined and SubjectJ rules ready, we proceeded to build the JQueryScape using only the SubjectJ menu queries and the default JQuery menu queries.

The JQueryScape in Figure 5.2 was created using the JQuery UI as described in Section 2.2. We used the SubjectJ top queries and browsed

[4]Lines of code including comments and whitespace

our project by subject instead of the normal Java package structure. We obtained the entire perspective by querying and dragging nodes to create query-rooted views. We were required to edit the original JQuery queries (such as the outline and outgoing calls query) to include SubjectJ predicates.

Each view in this JQueryScape contains some information that is relevant to the subject selected in the top left view which displays the subjects defined in the system. The bottom left view outlines the packages and classes that are in the selected subject. The view on the bottom displays customized error messages for the selected subject. The view on the top-right displays the outline of the current class with methods and fields categorized by the subject to which they belong while the view on the bottom-right displays the outgoing calls to the selected method also categorized by subject.

As a point of interest, we feel the JQueryScape we created is a reasonably complex perspective. The Contents and SubjectJ Errors view are both linked to the selection in the Subjects view. The Outline and Outgoing calls views are both linked to the editing context. Four out of the five views are displaying a customized code model. Even with the complexity of linking and a custom code representation this JQueryScape was created in a matter of minutes after the appropriate configuration file had been written.

The creation of this JQueryScape, including the TyRuBa configuration file, required little more than a day's worth of work. In fact, the SubjectJ configuration file consists of less than 200 lines of code[5]. We were not required to consider any of the complex Eclipse API nor write a significant amount of code because of JQuery's focused design. Such a complex perspective would require a lot of work to create in Eclipse given that even a single Eclipse view is generally hundreds of lines of Java code.

It is important to remember that we are not evaluating the usefulness of this perspective but instead using it as an example of the amount of work in creating a perspective for project specific concerns with JQuery. It also illustrates the relatively small amount of work required to create such a perspective in JQuery versus Eclipse due to JQuery's focused design.

5.2 Summary

In this chapter we have validated the second claim of this thesis, that an advanced JQuery user can build perspectives, that take advantage of project specific naming conventions and/or project specific annotations, with less work than building perspectives in current IDEs. Although making a direct

[5]TyRuBa configuration file lines of code include whitespace and coments.

comparison between these two options is beyond the scope of this work, we demonstrated that through creating multiple linked JQuery views and taking advantage of the declarative query language on top of which JQuery was built, we were able to create a code perspective leveraging information from project specific annotations.

Chapter 6

Related Work

In this chapter we will present the related work. We have divided the related work into two distinct categories to match our two distinct claims. Section 6.1 will look at related tools for source code queries and navigation. Section 6.2 will discuss related tools which provide support for project specific code perspectives. We will explain how JQuery is unique in its own way in each of these categories.

6.1 Tools for Code Queries and Navigation

There are many existing tools to help developers navigate code and answer questions about the code structure. Most IDEs offer views to answer queries on the source code and help developers navigate their projects. We will describe several other IDE extensions and querying tools in this section and show how JQuery is unique in its ability to combine results from different queries through the UI to answer more complex queries.

To answer complex queries, we could use a tool such as Ferret [2]. Ferret is an Eclipse plugin which, like JQuery, supports a number of extensions to its query engine through plugins and provides a number of complex queries out of the box. Although both JQuery and Ferret answer similar kinds of queries, Ferret differs from JQuery in that these queries are defined through extension points rather than through the UI. With JQuery, we provide support for combining results from different views to help users answer complex queries through the UI rather than through plugins and extensions.

QJBrowser [24], a predecessor to JQuery, and the older version of JQuery [16] both present a tool for creating views generated using a query language. Both of these tools allow a user to write queries in an expressive query language and hide the details of the query language behind the UI. Since each tool only supported a single view at a time, users must actually capture results in the query string to combine results from different queries. For our version of JQuery, we have built on top of these works by adding support for multiple JQuery views and interactions between those views which enables

users to combine results from different queries in the UI to answer more complex queries.

Another tool for querying about concerns in source code is FEAT [26]. FEAT allows a user to specify the contents of a feature and begin querying for related portions of the concern using a JQuery-like menu system. Unlike JQuery, there is no means for configuration or extension so the queries that are available in FEAT are hard coded into the system implying that the relationships that a user can browse become limited by what the tool supports. With JQuery, we provide a flexible mechanism to configure the relationships that can be viewed by allowing users to configure the queries the menu displays.

Other projects, for example Relo [28] and Lost [23] provide tools for querying code through the UI and provide tree-like structures to browse the results. Like JQuery, Relo allows for a user directed exploration of the source code by querying relationships between code entities such as classes and methods. It also allows for an in-place editor to help visualize not only a concern, but make changes to it. Lost is a querying tool very similar to QJBrowser, an ancestor to JQuery, except that it uses a query language similar to AspectJ's pointcut language. Neither tool provides a way to combine results from different code views to answer more complex queries.

In addition to these UI tools there are a number of tools for querying source code. These do not offer good UI support for browsing results but rather have focused on designing an efficient and easy to use query language.

An example code querying tool is JTransformer. JTransformer [19] is an Eclipse plugin which transforms a Java program into a collection of Prolog facts designed to allow a user to make changes to a Java program by applying transformations to the Prolog facts and then rebuilding the source code. Alternatively a user can use the JTransformer fact base to perform code queries using Prolog or build thier own user interface plugins on top of the factbase JTransformer provides. JTransformer, unlike JQuery, does not provide an interface for querying but rather a text console for typing and executing queries. In Appendix A we present the JQuery backend we have implemented on top of JTransformer, explain the details of how it is implemented, and descuss the subtle differences between JTransformer and the default TyRuBa implementation of the backend.

Similar code querying projects CodeQuest [15] and more recently SemmleCode [22] offer an SQL-like language to query over a database of facts. Both projects have good support for large code bases, an easy to learn query language, and variety of views on the results of their queries. Unlike JQuery, they do not allow interaction between views or allow a user to incrementally

refine their query by selecting a node and querying in place. The focus of these projects has been an easy to learn query language with UI support for the query language and scalable database implementation while the focus of JQuery has been on designing a flexible user interface to hide the details of the underlying query language and factbase. We see it as an interesting research project to construct a JQuery backend based on the SemmleCode database.

Other projects such as SOUL [33] and ASTLOG [9] aim to provide a query engine for source code but do not provide a UI on top of that language. SOUL is a declarative programming language for code queries over SmallTalk, Java, and C and is designed for code queries in much the same way as JQuery without a UI. ASTLOG [9] is a query language for abstract syntax tress (ASTs) using a Prolog based query language. ASTLOG was designed as a language for finding syntactic properties or design violations in the AST. Both of these projects contribute interesting query languages but do not offer a UI to hide those details.

The tools examined in the section represent the tools which are designed to answer code queries. Some are text based query engines while others provide UIs for executing their queries. None of these tools provide support for combining query results in the UI as we provide with JQuery.

6.2 Tools for Project Specific Perspectives

In this section we will look at other tools which support customizing code perspectives for project specific concerns. We have provided JQuery with as many features as possible and targeted the user interface as the main source of configuration. JQuery provides more flexibility user interface than these existing tools while requiring a relatively small amount of configuration.

Of all the UI tools that exist SVT [14] is the only one which uses a query language to generate its UI like we do with JQuery. SVT contains a more complete customization mechanism than JQuery by using a single backend which enables them to support more configuration of the facts generated. The configuration mechanism of SVT is more powerful than what we offer in the JQuery backend because it can make calls directly to an underlying C++ implementation. In JQuery we limit the configuration to defining logic rules and facts.

Though SVT is more configurable than JQuery, it does not provide interactions between its views. Though SVT also provides a much richer set of views over the source code including graphs, charts, in addition to hier-

archical structures like trees, it does not provide a rich set of interactions between these different views. JQuery does provides interaction between its views and we feel that this is an important distinction which allows JQuery to build a perspective, as opposed to code views (though JQuery is certainly capable of creating code views).

While SVT provides a wide variety of code views, there are numerous other examples of code visualization tools that produce graphs of the code structure. We have considered Ciao [3], Dagger [4], GraphLog [8], and Rigi [20]. These tools are not designed to be navigation tools but rather display the structure of the code. Each has a flexible mechanism for defining the graph to be displayed though only Rigi supports building a portion of the query which generates the graph in the user interface. Though they have not been applied to project specific concerns, they are built around a general query mechanism which would allow visualizing this kind of concern but not navigating that structure.

Apart from research tools, there is also a number of real-world tools which provide support for project specific concerns. Our representative modern IDE, Eclipse, contains a number of ways to customize its perspectives for project specific concerns. Eclipse working sets provide a way to group resources from the workspace into a named collection. Most Eclipse views support filters based on the items that are inside these working sets. If filters were applied to all Eclipse views one could think of that as a project specific perspective as only information about classes inside the working set would be displayed. There are however three problems with using working sets for project specific concerns:

1. Eclipse working sets do not have fine enough granularity to meets the needs of annotations because working sets consist of a selection of files or resources while an annotation can be applied to individual code elements such as methods and fields.

2. Working sets must also be managed by the developer which adds an additional burden.

3. Working sets cannot define higher level structure in the code but merely relations between files.

Another way to generate a perspective based on naming conventions in Eclipse is to use view filters. Eclipse views have a filtering mechanism built into them which allows matching results to be hidden from the view. The JQuery mechanism is quite different in that it hides results that do not

match and so it behaves much more like a search mechanism than a filter. Furthermore each Eclipse view can only contain one filter while in JQuery we can clone a view and change the filter to display different portions of the same result.

To create perspectives specifically designed for project specific annotations we can use the Eclipse JDT-APT [31]. The JDT-APT is an annotation processing framework for Eclipse and was designed to "provide a build-time, source-based, read-only view of program structure". In that sense it is very similar to JQuery as JQuery provides a build-time, source-based, read-only view of program structure through its factbase. JDT-APT constructs Java files and classes which Eclipse can then process while JQuery performs logic calculations on facts generated from annotations. Though JDT-APT provides good support for annotations, JQuery provides a more general mechanism as it is able to display information about other project specific concerns in conjuction with annotations.

Another plugin for customizing IDE perspectives is Mylyn (formerly Mylar [18]). Mylyn is a plugin which monitors the users work and filters views based on the items that are accessed most frequently. Mylyn also provides the user a way to specify a task and thus build up information about interesting items while they work and only display those items in the perspective. Mylyn is different from JQuery in that it defines code structure based on the users browsing trends while JQuery lets the user declare their own structures over the code through an expressive query language. Furthermore Mylyn does not provide any way to generate views but instead provides a filtering mechanism for the existing views.

There are also tools to construct multiple views on source code. We have considered Stellation [7], SHriMP [1], and Coven [6] as examples of such tools as they provide an IDE-like experience by displaying multiple views on the source code. Stellation is a framework for creating aggregates of a project similar to Eclipse working sets but at a finer granularity (method level instead of file level). SHriMP provides is a framework for zooming and navigating large code views but unlike JQuery it does not provide an expressive query language to build those views. Coven is designed to display source code editors over specific concerns and unlike JQuery, it does not provide much support for navigation but instead provides a good multi-user code editor. Each of these tools provide the user with ways to view project specific concerns in the code by providing a particular abstraction on top of the source code. We feel that JQuery is unique because it provides a programming language to let advanced users define their own abstraction.

6.3 Summary

Though there are many existing projects related to JQuery, they do not provide a flexible enough interface for answering questions nor do they provide a concise configuration mechanism for defining project specific perspectives.

Chapter 7

Conclusions

In this dissertation we have introduced a new version of the JQuery tool which was designed to address two specific problems with current IDEs.

The first problem is that the views inside the IDE are designed to answer low level questions but users may need to combine information from those views to answer higher level questions [27]. IDEs do no support combining information between views and therefore are unable to answer more complex higher level questions.

The second problem is that code perspectives are too difficult to customize for project specific concerns. IDEs usually offer very little support for project specific concerns such as naming conventions or annotations but instead offer extension frameworks so that plugin developers can build their own perspectives. These extension frameworks provide the necessary features but building a perspective for project specific concerns is too costly.

JQuery addresses these problems by allowing users to combine information between views and by building each view on top of an expressive query language. In Chapter 2 we provided a users perspective on the JQuery UI features including the regular expression search box, view linking, and drag and drop. Also in Chapter 2 we provided an advanced users perspective on configuring the JQuery UI and code representation using the expressive query language. In Chapter 3 we provided a developers perspective on JQuery including details about how to write a custom database for JQuery and how to build UI plugins on top of the JQuery database.

In addition to introducing the new JQuery tool and its design we have validated two distinct claims:

1. JQuery users can answer more complex queries through the user interface than with current integrated development environments.

2. Building collections of code views (perspectives), that take advantage of project specific naming conventions and/or project specific annotations, with JQuery is less work than building perspectives in current integrated development environments.

To validate the first claim, we defined a sample set of complex queries. We used this set to show that JQuery is able to answer more complex queries through its UI than the representative modern IDE, Eclipse, in Chapter 4.

To validate the second claim we argued through our experience the amount of work required to create a perspective in Eclipse and the work required to create a JQueryScape. Though we did not determine any specific measurements for the work in each case, we provided evidence to support our claim that a JQueryScape would likely take less work to develop than an Eclipse perspectives. We illustrated the work invovled in creating a JQueryScape through a case study in Chapter 5.

7.1 Future Work

We have made substantial contributions to the JQuery implementation which have opened the door for new ways to answer queries through the UI. Though it seems useful to use an expressive query language to build views, we would like to avoid teaching a user a new language to browse their code and instead would like to see future work involving ways to improve the JQuery UI. Here are some possible directions for future research:

Better filtering mechanism. JQuery result filters are currently applied through the context menu like queries but we would like to improve this mechanism. Perhaps drag and drop filtering or applying filters through toolbar buttons rather than the context menu would be appropriate.

Composition of filters. Currently JQuery will compose filters using a conjunction but a user may actually which to compose filters using other means. We would like future work to design UI mechanisms to allow finer control over how filters are applied to query results.

Query generated UI. Currently the menus, icons, and labels are the major portions of the UI generated by queries. We would like to see JQuery support tool bars or other dialogs generated using the expressive query language. We could take this idea so far as to declare the entire JQueryScape through the query language.

In addition to IDE features, we would like to see JQuery move beyond tree viewers. Eclipse produces code markers, syntax highlighting, and hover popup information that could all be generated using JQuery. We expect that a good direction for future work would be to concentrate on providing a much richer UI for JQuery than a tree view.

UI support for categorization of results. Advanced users are able to define their own categories or ways to sort their results. We wish to make this more accessible to a user who is not familiar with the underlying query language. This would allow them to build their own categories through the UI and navigate results based on those categories, assisting them to answer more advanced queries through the UI.

UI support for query aggregation. We would like to see support for users to compare results from different views uing user defined aggregation. This would include taking the difference of two results sets or performing counting operations on results. Again this support can be added by an advanced user but we would like to see it instead implemented in the UI to be accessible for most users.

Bibliography

[1] Shrimp views: An interactive environment for exploring java programs. In *IWPC '01: Proceedings of the 9th International Workshop on Program Comprehension*, page 111, Washington, DC, USA, 2001. IEEE Computer Society.

[2] Brian De Alwis. *Supporting Conceptual Queries Over Integrated Sources of Program Information*. PhD thesis, University of British Columbia, Software Practices Lab, April 2008.

[3] Y.-F. R. Chen, G. S. Fowler, E. Koutsofios, and R. S. Wallach. Ciao: a graphical navigator for software and document repositories. In *ICSM '95: Proceedings of the International Conference on Software Maintenance*, page 66, Washington, DC, USA, 1995. IEEE Computer Society.

[4] Yih-Farn Chen. Dagger: a tool to generate program graphs. In *UNIX'94: Proceedings of the USENIX Applications Development Symposium Proceedings on USENIX Applications Development Symposium Proceedings*, pages 2–2, Berkeley, CA, USA, 1994. USENIX Association.

[5] Rick Chern. Reducing remodularization complexity through modular-objective decoupling. Master's thesis, University of British Columbia, Software Practices Lab, April 2008.

[6] Mark C. Chu-Carroll and Sara Sprenkle. Coven: brewing better collaboration through software configuration management. In *SIGSOFT '00/FSE-8: Proceedings of the 8th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 88–97, New York, NY, USA, 2000. ACM.

[7] Mark C. Chu-Carroll, James Wright, and David Shields. Supporting aggregation in fine grained software configuration management. In *SIGSOFT '02/FSE-10: Proceedings of the 10th ACM SIGSOFT symposium on Foundations of software engineering*, pages 99–108, New York, NY, USA, 2002. ACM.

[8] Mariano Consens, Alberto Mendelzon, and Arthur Ryman. Visualizing and querying software structures. In *ICSE '92: Proceedings of the 14th international conference on Software engineering*, pages 138–156, New York, NY, USA, 1992. ACM.

[9] Roger F. Crew. Astlog: a language for examining abstract syntax trees. In *DSL'97: Proceedings of the Conference on Domain-Specific Languages on Conference on Domain-Specific Languages (DSL), 1997*, pages 18–18, Berkeley, CA, USA, 1997. USENIX Association.

[10] Brian de Alwis and Gail C. Murphy. Using visual momentum to explain disorientation in the eclipse ide. In *VLHCC '06: Proceedings of the Visual Languages and Human-Centric Computing*, pages 51–54, Washington, DC, USA, 2006. IEEE Computer Society.

[11] Kris De Volder. Tyruba language reference. http://tyruba.sourceforge.net/tyruba_language_reference.html.

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

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

[14] Calum A. McK. Grant. *Software Visualization in Prolog*. PhD thesis, Queens' College, Cambridge, December 1999.

[15] Elnar Hajiyev, Mathieu Verbaere, and Oege de Moor. Codequest: Scalable source code queries with datalog. In Dave Thomas, editor, *ECOOP'06: Proceedings of the 20th European Conference on Object-Oriented Programming*, volume 4067 of *Lecture Notes in Computer Science*, pages 2–27, Berlin, Germany, 2006. Springer.

[16] Doug Janzen and Kris De Volder. Navigating and querying code without getting lost. In *Aspect-Oriented Software Engineering*, pages 178–187. ACM, 2003.

[17] java.net. castor-annotations: Castor annotations home page. https://castor-annotations.dev.java.net/.

[18] Mik Kersten and Gail C. Murphy. Mylar: a degree-of-interest model for ides. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, pages 159–168, New York, NY, USA, 2005. ACM.

[19] Günter Kniesel, Jan Hannemann, and Tobias Rho. A comparison of logic-based infrastructures for concern detection and extraction. In *LATE '07: Proceedings of the 3rd workshop on Linking aspect technology and evolution*, page 6, New York, NY, USA, 2007. ACM.

[20] Hausi A. Müller, Scott R. Tilley, and Kenny Wong. Understanding software systems using reverse engineering technology perspectives from the rigi project. In *CASCON '93: Proceedings of the 1993 conference of the Centre for Advanced Studies on Collaborative research*, pages 217–226. IBM Press, 1993.

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

[22] Semmle code website. http://semmle.com/, 2008.

[23] J.-Hendrik Pfeiffer, Andonis Sardos, and John R. Gurd. Complex code querying and navigation for aspectj. In *eclipse '05: Proceedings of the 2005 OOPSLA workshop on Eclipse technology eXchange*, pages 60–64, New York, NY, USA, 2005. ACM.

[24] Rajeswari Rajagopalan. Qjbrowser - a query based approach to explore concerns. Master's thesis, University of British Columbia, Software Practices Lab, September 2002.

[25] Martin P. Robillard and Wesley Coelho. How effective developers investigate source code: An exploratory study. *IEEE Trans. Softw. Eng.*, 30(12):889–903, 2004. Member-Gail C. Murphy.

[26] 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.

[27] Jonathan Sillito, Gail C. Murphy, and Kris De Volder. Questions programmers ask during software evolution tasks. In *SIGSOFT '06/FSE-14: Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 23–34, New York, NY, USA, 2006. ACM.

[28] Vineet Sinha, David Karger, and Rob Miller. Relo: helping users manage context during interactive exploratory visualization of large codebases. In *eclipse '05: Proceedings of the 2005 OOPSLA workshop on Eclipse technology eXchange*, pages 21–25, New York, NY, USA, 2005. ACM.

[29] Frank Sommers. Upcoming features in jdbc 4. http://www.artima.com/lejava/articles/jdbc_four3.html, 2005.

[30] Leon Sterling and Ehud Shapiro. *The art of Prolog: advanced programming techniques*. MIT Press, Cambridge, MA, USA, 1986.

[31] Eclipse JDT-APT Team. Jdt-apt project: Annotations processing in eclipse. http://www.eclipse.org/jdt/apt/main.html.

[32] Kris De Volder. Jquery: A generic code browser with a declarative configuration language. In Pascal Van Hentenryck, editor, *PADL*, volume 3819 of *Lecture Notes in Computer Science*, pages 88–102. Springer, 2006.

[33] R. Wuyts. Declarative reasoning about the structure of object-oriented systems. In *TOOLS '98: Proceedings of the Technology of Object-Oriented Languages and Systems*, page 112, Washington, DC, USA, 1998. IEEE Computer Society.

Appendix A

JTransformer

As a product of redesigning the JQuery plugin we are able to implement the JQuery API on top of a database and run the JQuery UI on that database. Here we present the internals of the JTransformer [19] implementation of the JQuery API and the differences between it and the TyRuBa implementation. Although TyRuBa is a Prolog-like language and JTransformer uses pure Prolog [30], they both contain their own separate rule definitions and fact generating procedures and both store data in different formats.

Our implementation of the API simply provides an adapter for the Prolog queries and converts the results into JQuery results. We were able to provide a complete implementation of the JQuery API in approximately 1,200 lines of Java code [6]. 500 lines of Prolog [7] were required to supply the necessary definitions to make JTransformers fact structure more closely resemble the fact structore of JQuery.

The JTransformer facts are structured differently from TyRuBa JQuery. JTransformer was designed to support code transformations by making changes to the factbase and generating the code from the facts. The factbase is composed primarily of predicates with a larger number of arguments (5 or 6) than JQuery TyRuBa's predicates (3 or 4). To make the JTransfomer factbase more closely resemble the TyRuBa version, we provided a series of rules to convert the JTransformer predicates into a more JQuery friendly format. Because Prolog is so similar to TyRuBa the conversion was straightforward with the exception of ordering the JTransformer predicate to make the queries execute efficiently.

To make JTransformer compatible with the existing JQuery tree view, we were required to create definitions for the queries in Table 3.2. Once we had completed defining all of the API queries, we were able to use the JQuery frontend on top of the JTransformer backend. The JTransformer version is, to the average user, indistinguishable from the TyRuBa version.

One note of convenience for developers is that JTransformer has its own

[6] Lines of code were counted using the Eclipse Metrics 1.3.6 plugin avaiable at: http://metrics.sourceforge.net/

[7] Including comments and whitespace.

Table A.1: Brief description of classes implemented for the JTransformer backend.

Class	Difficulties
`JQueryJTransformerAPI`	Implementation of API factory methods and one JTransformer specific method to grant access to underlying Prolog database.
`JQueryJTransformerException`	Exception class thrown from the JTransformer backend.
`JTransformerFactBase`	Wrapper around the JTransformer Prolog interface. JQuery uses the concept of a named Prolog interface as its factbase.
`JTransformerQuery`	Wrapper around the JTransformer interface for querying Prolog and examining the results.
`JTransformerQueryResult`	A single result from a JTransformer query.
`JTransformerQueryResultSet`	A set of results from a JTransformer query. Because JTransformer doesn't (currently) support returning results in an iteratable format this structure actually stores all results from the query at once unlike the TyRuBa version.
`JTransformerUpdateTarget`	Provides support for update targets on top of the JTransformer backend.

utilities to build and manage the factbase. This greatly simplifies our implementation of the JTransformer backend becaues we do not need to manage changes and updates to the working sets as we had to do with TyRuBa. It does however have the disadvantage of making the factbase difficult to extend and customize but we are currently investigating solutions to this problem.

Table A.1 is a list of the classes implemented as well as a brief explanation of how they were implemented on top of JTransformer. Only a small part of the API needed to be implemented because JTransformer, as we mentioned above, manages and builds its own factbase.