QJBrowser: A Query-Based Browser Model

Rajeswari Rajagopalan and Kris De Volder
The Univerisity of British Columbia
201-2366 Main Mall
Vancouver BC, V6T 1Z4
+1(604)8221209
{kdvolder,raji}@cs.ubc.ca

ABSTRACT

Many development tasks are not local to a single modular component of a system but involve changes across many different modules. To carry out such a task a developer needs to understand many different kinds of relationships that exist between different parts of their code-base. In this paper we present QJBrowser, a query-based browser tool. Using QJBrowser, a developer can dynamically create many different kinds of browsers that select and organize elements of a code base in terms of many different kinds of criteria and semantic or structural relationships. Compared to state-of-the-art integrated development tools, that typically come with a limited set of built-in code browsers, our query-based browser model offers a much more dynamic and flexible way to browse code. On the one hand, this allows developers to explore the code base more directly in function of their immediate needs to explore specific relationships between specific parts of their code. On the other hand, the browser model enables the definition of code base specific browsers that reveal more high-level, code-base specific relationships and concepts which would otherwise be hard to discover in the code.

Keywords

Programming environments, multi-dimensional separation of concerns, aspect orientation, crosscutting, source code querying, code browsing.

1. INTRODUCTION

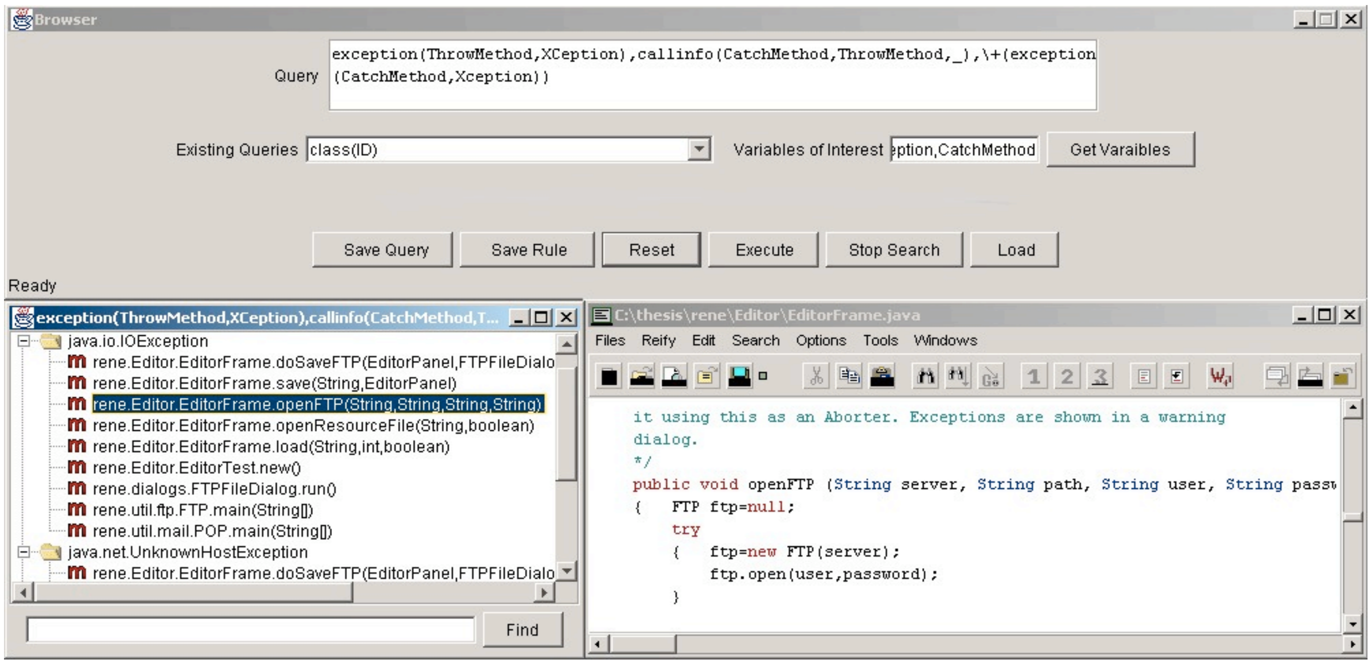
Many tasks developers face on a daily basis cut across the modularity of the system and involve exploration and changes across many different parts of the code base. Such tasks are said to involve crosscutting concerns. Tasks involving crosscutting concerns are hard because they require the developer to understand a complex web of relationships that exists between scattered elements of the code base. A good code browsers indirectly assists a developer to deal with these kinds of tasks. Code browsers organize the elements of a code base in several different ways, providing a developer with different ways to view and navigate a code base in terms of different structural and semantic relationships induced by the programming language's semantics. For example, a modern Java IDE might provide a developer with a package browser, a class hierarchy browser and include embedded within the source code editor. In this way an IDE helps a developer to navigate and explore a code based more effectively. Indirectly this improves the developer's ability to deal with crosscutting concerns in the code.

In this paper we present a source code browsing tool called QJBrowser. QJBrowser is intended to further exploit the idea that various types of code browsers can indirectly support developers to deal with crosscutting concerns. The construction of our prototype was motivated by the following considerations:

1. The number of different code browsers developers can use in an IDE is limited by what the IDE developers have chosen to provide. Consequently, a specific code browser typically provides a specific type of organizational view which provides specific navigational pathways and reveals specific kinds of relationships between elements of the code base (e.g. inheritance between classes for a class hierarchy browser).
2. In principle, more types of browsers could be supported by IDE developers. However, there is a practical limit on the number of browsing tools that can be developed and shipped with an IDE. Firstly because building a new tool for each potentially interesting view is very costly and impractical. Secondly because every added tool adds to the overall complexity of the IDE.
3. There is no obvious limit to the number of kinds of browsers that developers may find useful at one time or another. Different kinds of properties and relationships can be used for organizing elements in a browser (e.g. method names, static types, exception flow, inheritance, method calls, etc.) Many of these can be used in combination leading to a combinatorial explosion of potentially useful code browsers. In section 3.1 we will show an example that illustrates the idea of a combinatorial explosion of useful browsers.
4. Views that are specific to an application, a library, a framework, a software development company etc. can be very useful. For example, a browser that is aware of the naming conventions used within a specific framework could organize code base elements in terms of concepts and relationships that are specific to that particular framework. It would be hard to build such code-base-specific browsers into a general purpose IDE.

All of the above considerations inspired us to believe that an environment offering a tool that allows developers to flexibly define their own browsers would further enhance the way that integrated development environments already support developers indirectly to deal with crosscutting concerns.

A key issue in the design of such a tool is the trade-off between flexibility and simplicity. An effective tool offers a browser definition mechanism that is conceptually simple and, at the same time, flexible enough to allow the creation of broad set of useful browsers. QJBrowser establishes a good trade-off



```
exception(ThrowMethod,XCeption),callinfo(CatchMethod,ThrowMethod,_),\+(exception
(CatchMethod,Xception))
```

Existing Queries: class(ID)

Variables of Interest: eption,CatchMethod

exception(ThrowMethod,XCeption),callinfo(CatchMethod,T...

```
java.io.IOException
    rene.Editor.EditorFrame.doSaveFTP(EditorPanel,FTPFileDialo
    rene.Editor.EditorFrame.save(String,EditorPanel)
    rene.Editor.EditorFrame.openFTP(String,String,String,String)
    rene.Editor.EditorFrame.openResourceFile(String,boolean)
    rene.Editor.EditorFrame.load(String,int,boolean)
    rene.Editor.EditorTest.new()
    rene.dialogs.FTPFileDialog.run()
    rene.util.ftp.FTP.main(String[])
    rene.util.mail.POP.main(String[])
java.net.UnknownHostException
    rene.Editor.EditorFrame.doSaveFTP(EditorPanel,FTPFileDialo
```

C:\thesis\rene\Editor\EditorFrame.java

```
it using this as an Aborter. Exceptions are shown in a warning
dialog.
*/
public void openFTP (String server, String path, String user, String passw
{   FTP ftp=null;
    try
    {   ftp=new FTP(server);
        ftp.open(user,password);
    }
```

Figure 1.1: QJBrowser

between the two. In QJBrowser, hierarchically organized browser views are defined by means of queries against the code base. A query determines what elements will be shown by the browser as well as the specific relationships and properties that will be used to organize them. This mechanism is conceptually simple and at the same time flexible. It provides a cost-effective way to define new views. The cost of defining a new view involves little more than the formulation of a query.

The rest of the paper is organized as follows. In the next section (section 2), we introduce QJBrowser and describe its underlying ideas using a single concrete example. In section 3, we present more examples that highlight the utility of QJBrowser. Section 4 describes some preliminary experience using the tool on a small development task and section 5 compares the tool to other similar work. Section 6 outlines the limitations of the tool and presents some ideas for future work. Finally, section 7 summarizes the paper.

2. QJBROWSER

QJBrowser, short for "Query Java Browser", is a proof-of-concept prototype. Presently, QJBrowser is implemented in Java and supports two query languages, namely TyRuBa [19][20] and Sicstus Prolog [17], both of which are based on first-order predicate logic. We limit the discussion in this paper to Sicstus Prolog, which is a commercial implementation of standard Prolog.

Strictly speaking, QJBrowser is not actually a browser, but is a tool, which provides mechanisms that allows users to create their own browsers. The definition of a browser consists of two parts:

1. A *selection criterion*, which determines what elements are part of the browser's view. The selection criterion is a query that is executed against a source model. *The source model* is a suitable representation of the source code that can be queried by an expression in some kind of *query language*.
2. An *organization criterion,* which specifies how to organize the query results in the browser.

2.1 EXAMPLE: EXCEPTION HANDLER BROWSER

Before discussing the selection criterion and the organization criterion in more detail, we present a concrete example of how the tool works and how it can be used to define useful browsers. This example concerns exception handling which is notorious for being difficult to manage in Java [16], partly because of its crosscutting nature [13]. We will introduce QJBrowser's user interface, and show how a developer can use it to define a browser that allows her to conveniently view the methods in her system that handle certain exceptions. Without such a tool, finding out where a particular exception is handled would require laborious exploration of the source code.

QJBrowser's user interface presents a developer with a dialog box from which it is possible to launch different browsers by inputting the parameters that define it. A screenshot of the QJBrowser tool is shown in Figure 1.1. The textbox captioned "Query" is where the query representing the selection-criterion is entered. The entry in the box named "Variables of interest" represents the organization criterion. It determines how the query results will be organized in a tree; we will explain the rationale behind representing query results as trees later. By clicking in the tree view, the developer can expand or collapse nodes. By double-clicking on a node, she can open a source editor with the cursor positioned near the corresponding code element, (provided the node has source code associated with it). To assist the developer in composing queries, QJBrowser provides a menu from which useful expressions can be selected and appended to the query box.

In our example, the developer is interested in finding out where certain exceptions are handled in the code. Unfortunately, the exact location of the `catch` statements in the program is not explicit in the current version of the tool's source model. However, the developer realizes that methods where exceptions get handled can nevertheless be deduced by correlating the static callgraph information with information from exception declarations in method signatures, both of which are stored in the source model. The developer enters the following Prolog query in the ``Query box":

```
exception(ThrowMethod,XCeption),
callinfo(CatchMethod,ThrowMethod,_),
\+(exception(CatchMethod,XCeption))
```

For a reader unfamiliar with Prolog, this may require a little bit of explanation. The identifiers starting with upper case letters (`ThrowMethod`, `XCeption`, etc.) are variables, which will be bound to values as a result of query execution. A "," stands for logical conjunction and a "\+" stands for logical negation.

Thus, this query will find all possible combinations of values for the variables `ThrowMethod`, `CatchMethod` and `XCeption` such that `ThrowMethod` is a method that throws the exception `XCeption`, and `CatchMethod` is a method that calls `ThrowMethod` but does not itself declare throwing `XCeption`. In other words, `CatchMethod` will be bound to methods where exceptions disappear from the static call graph. These are methods where those exceptions are handled in the system.[1]

In the "Variables of interest box" the developer enters a list of the query variables she is interested in. This constitutes the organization criterion for the browser. The order in which the variables are entered determines how different elements will be organized in the resulting browser. For example entering a list `XCeption,CatchMethod` will result in a browser that categorizes methods according to the exceptions they handle (shown in Figure 2.1).

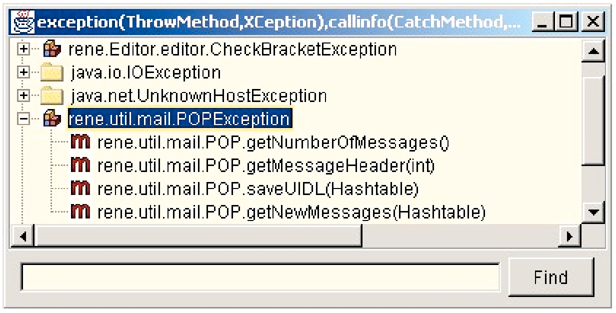

exception(ThrowMethod,XCeption),callinfo(CatchMethod,...
rene.Editor.editor.CheckBracketException
java.io.IOException
java.net.UnknownHostException
rene.util.mail.POPException
rene.util.mail.POP.getNumberOfMessages()
rene.util.mail.POP.getMessageHeader(int)
rene.util.mail.POP.saveUIDL(Hashtable)
rene.util.mail.POP.getNewMessages(Hashtable)

Figure 2.1: Exception Browser - Flavor 1

On the other hand, entering `CatchMethod,XCeption` as the organization criterion, categorizes exceptions according to the methods in which they are caught (shown in Figure 2.2).

[1] For the sake of simplicity, this example does not take into account the inheritance relationships that may exist between exceptions. The query could be elaborated using a `subtype` predicate to make it more precise.

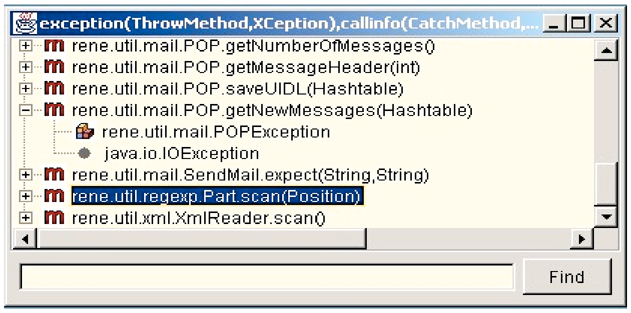

exception(ThrowMethod,XCeption),callinfo(CatchMethod,...
rene.util.mail.POP.getNumberOfMessages()
rene.util.mail.POP.getMessageHeader(int)
rene.util.mail.POP.saveUIDL(Hashtable)
rene.util.mail.POP.getNewMessages(Hashtable)
rene.util.mail.POPException
java.io.IOException
rene.util.mail.SendMail.expect(String,String)
rene.util.regexp.Part.scan(Position)
rene.util.xml.XmlReader.scan()

Figure 2.2: Exception Browser - Flavor 2

2.2 Selection Criterion

The selection criterion is defined as a query that is run against the source model. QJBrowser's source model consists of logic facts extracted from the source code by a static analysis tool. Our current implementation uses a modified version of the AspectJ type checker for discovering simple static information, such as inheritance relationships, calling dependencies, etc. Table 2.1 gives an overview of primitive queries that are supported by the current implementation of the source model.

Note that the table only lists the primitive queries, which can be used to compose more complex derived queries using standard Prolog syntax for logic conjunction, disjunction and negation. An example of a composite query was given in the previous section. We will give one more example here. Suppose that we want to find all exceptions that are explicitly propagated by methods of the class, `testpackage.Foo`. We could get this information using the following query:

```
method(FooMethod,'testpackage.Foo'),
exception(FooMethod,FooExc)
```

The query is composed of two parts combined by a logic conjunction (denoted by a ","). Notice how the first part is a query derived from, but not identical to, the query format `method(Met,Cls)`, as given by the second entry in Table 2.1. This query is more specific than the one listed in the table, because it has the name of a particular class rather than a logic variable as its second parameter. This query will therefore find only the methods that are declared in the class '`testpackage.Foo`'.

This example illustrates a general principle of Prolog expressions: all the entries in the table can be used in multiple ways, depending on whether each one of its parameters is specific or not. For example a `subtype` query can be used in four different ways: 1) to find all super types of a specific type. 2) to find all subtypes of a specific type 3) to find all pair-wise combinations of a super type and a subtype; and 4) to test whether a certain type is a subtype of another specific type. Each of these four "modes" of the `subtype` predicate can be used in the composition of a more complex query.

Primitive Query Format	Description
`class(Cls)`	Find all classes `Cls` declared in the system.
`method(Met,Cls)`	Find all pairs `Met,Cls` where `Cls` is a class (or interface) declared in the system and `Met` is a method

Primitive Query Format	Description
	decelerated in that class.
exception(Met,Exc)	Find all pairs Met,Exc where Met is a method declaration in the system and Exc is an exception declared to be thrown by Met.
callinfo(Caller, Callee,Line)	Find all pairs Caller,Callee where Caller is a method declared in the system and Callee is a method called by Caller (according to the static call graph). Additionally, Line will be bound to a reference to the actual source-code line where the call occurs.
member(Mem,Cls)	Find all pairs Mem,Cls where Cls is a class declared in the system and Mem is a member (variable, method or constructor) decelerated in that class.
subtype(Sub,Sup)	Find all pairs Sub,Sup of class or interface types declared in the system, such that Sub is a subtype of Sup.
Modifier(Dec,Mod)	Find all pairs Dec,Mod where Dec is a declaration (for a class, interface, method, constructor or variable) in the system and Mod is a modifier (public, private, protected, …) attached to that declaration.
shortname(Dec,Nam)	Find all pairs of Dec,Name where Dec is a declaration (for a class, interface, method, constructor or variable) in the system and Nam is the short name for the declared entity: the unqualified name for a class or interface or a field or the selector name for a method.
constructor(Met, Cls)	Find all pairs Met,Cls where Cls is a class declaration in the system and Met is a constructor method declaration.
type(Fld,Typ)	Find all pairs Fld,Typ where Fld is a field declaration in the system and Typ is the declared type of Fld.
callgraph(StartMethod,CalledList)	Finds a list of methods CalledList such that each method in the list is transitively reachable from the method StartMethod.

Table 2.1: Some primitive queries supported by the source model

2.3 ORGANIZATION CRITERION

The organization criterion is an ordered list of variables occurring in the selection criterion. The idea underlying the organization criterion is that it defines a way to project a multi-dimensional space that models the results of executing the selection-criterion query, onto a tree. Because this concept is hard to explain in abstract terms, we will explain it in terms of the exception browser example given in section 2.1.

Typically, executing a logic query produces a set of solutions. Each solution in the set consists of bindings for the variables in the query. In our example, there were three variables in the query: ThrowMethod, CatchMethod and XCeption. Every solution to the query will bind a reference to a method to ThrowMethod, and a reference to another method to CatchMethod. It will also bind a reference to an exception to XCeption. Thus, we can think of each solution as a 3-tuple composed of three references: two to methods and one to an exception. The query results can thus be represented as points in a three dimensional space and each one of the variables corresponds to an axis or dimension of that space.

In our example, we were only interested in two out of the three variables: CatchMethod and XCeption. By omitting one of the variables from the organization criterion we are implicitly reducing the dimensionality of the result space from 3 to 2. Now, we just have tuples of CatchMethod and XCeption, which can be thought of as a set of points in a two-dimensional space. On one axis, we plot all possible values for the variable XCeption. On the other axis, we plot all possible values for the variable CatchMethod. A point with coordinates (CatchMethod-a,XCeption-x) is marked, if the method CatchMethod-a handles exception XCeption-x.

As our example shows, in principle, neither of the axes is more important than the other. However, for projecting the result space onto a tree, the user specifies an explicit order on them. This order defines how the units on the axes are categorized in the tree. For example, considering dimensions X and Y, if the order imposed on them is (Y,X), then the units on X-axis are categorized according to the units on Y-axis. Similarly, if the order on the axes is (X,Y), then the units on the Y-axis are categorized according to the units on the X-axis.

It is evident from Figure 2.1 and Figure 2.2 that, even though the same data is being displayed, changing the order of the variables changes the shape of the tree dramatically. The actual points in the result space that can be reached by navigating the tree are the same, but the access paths differ. We can think of alternative projections as a way to look at the query results from a different perspective. Note that, in our exception example, the perspectives in both Figure 2.1 and Figure 2.2 are useful. Each one reveals different kinds of information more clearly. In Figure 2.1, it is easy to find out all places where a particular exception is being handled in the system but it is not easy to find out a list of exceptions that are handled by a particular method. The latter is more easily found in the browser in Figure 2.2.

3. EXAMPLES

In this section, we will present two sets of examples that illustrate the usefulness of QJBrowser. The first set of examples shows general-purpose views. The second set of examples shows how we can make use of code-base-specific knowledge to define code-base-specific views.

3.1 GENERAL-PURPOSE BROWSERS

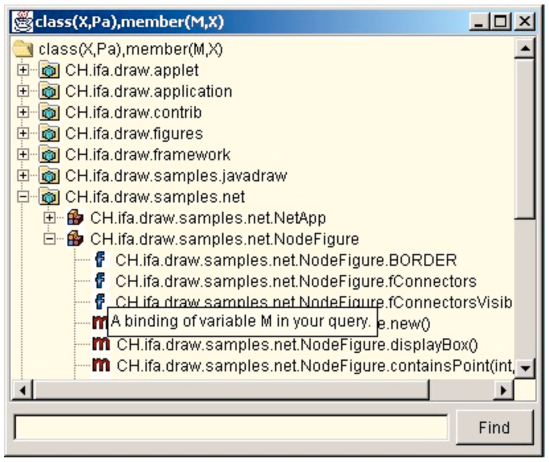
The examples in this section show how we can use many kinds of information to define general-purpose views on the code base. We call these views "general-purpose" because their definitions do not require code-base-specific knowledge and therefore they are potentially useful in browsing any kind of code base.

3.1.1 A CONVENTIONAL CLASS BROWSER

Our first example is the definition of a conventional class browser. It offers a view typical of any ordinary class browser, with packages, classes and members displayed in a hierarchy, in that order.

Selection:
`class(Class,Package),member(Member,Class)`

Organization : `Package, Class, Member`



```
class(X,Pa),member(M,X)
class(X,Pa),member(M,X)
+ CH.ifa.draw.applet
+ CH.ifa.draw.application
+ CH.ifa.draw.contrib
+ CH.ifa.draw.figures
+ CH.ifa.draw.framework
+ CH.ifa.draw.samples.javadraw
- CH.ifa.draw.samples.net
   + CH.ifa.draw.samples.net.NetApp
   - CH.ifa.draw.samples.net.NodeFigure
      f CH.ifa.draw.samples.net.NodeFigure.BORDER
      f CH.ifa.draw.samples.net.NodeFigure.fConnectors
      f CH.ifa.draw.samples.net.NodeFigure.fConnectorsVisib
      m A binding of variable M in your query. e.new()
      m CH.ifa.draw.samples.net.NodeFigure.displayBox()
      m CH.ifa.draw.samples.net.NodeFigure.containsPoint(int,
Find
```

Figure 3.1: Conventional Class Browser

This class browser is an all-round tool, suitable for many purposes, but not specifically attuned to the particularities of a specific task or code base. In most IDEs, this is the only kind that is available. With QJBrowser, in contrast, it is possible to customize the class browser by editing its view definition. The remaining examples in this paper will show, amongst others, several customized class-browser-like views.

3.1.2 EXCEPTION BROWSERS

The exception browser discussed in the beginning of this paper is only one of many similar browsers that can be defined around the theme of exceptions. Our next example shows how we can obtain a "family" of useful browsers by combining a class-browser-like view with organization based on exceptions. All browsers in this family share the same selection criterion but have different organization criteria.

Selection:
`class(Class,Package),member(Method,Class),`
`exception(Method,Exception).`

There are many possible organization criteria for this selection criterion, each defining a somewhat different browser. The total number of possible browsers that can be obtained by selecting different variables and reordering them is:

$$\binom{1}{4}.1!+\binom{2}{4}.2!+\binom{3}{4}3!+\binom{4}{4}4!=64$$

It should be noted that not all 64 variations are (equally) useful. For one thing, there is a natural order on the variables `Package`, `Class` and `Method`. Putting these variables in a different order does not result in a very useful view, because it will not impose any meaningful organization. It is worth pointing out here the distinction between a method and its name. We use the word "method" to refer to the identity of a specific method declaration in the code. So, while it may be useful to organize classes by the names of the methods they contain, organizing classes by their actual methods (as opposed to method names) is not very useful since a method declaration is only part of a single class declaration. In a way, the `Package`, `Class` and `Method` variables do not constitute orthogonal dimensions because there is a functional correlation between them: each method belongs to only a single class, and each class to single package. The `Exception` dimension however is orthogonal to `Package`, `Class` and `Method` dimensions in the sense that given exception can be thrown in many different packages, classes and methods. This means that in the organization criterion `Exception` can be positioned independently of `Package`, `Class` and `Method`. Taking these and some other considerations into account we have reduced the number of actually useful views to the 13 shown in Table 3.1.

Organization #	Useful organization criteria
1.	E P C M
2.	P E C M
3.	P C E M
4.	P C M E
5.	E P M
6.	E C M
7.	P E M
8.	P M E
9.	C E M
10.	C M E
11.	E M
12.	C E
13.	M E

E = Exception
P= Package
C= Class
M= Method

Table 3.1: A list of useful organization criteria for `class(C,P),member(M,C), exception(M,E).`

We will only explicitly discuss the first 4 variations, which use all the variables in the query. These four are presumably the most useful ones. They are also characteristic of the others. Each one of the four resulting browsers differs from the others only in the way it shows how propagation of exceptions crosscuts the organization of methods into classes and packages.

Organization 1 lists all exceptions propagated in the system. Opening an exception node will reveal a structure similar to the conventional class browser except that it only shows packages, classes and methods in which that exception is declared to be thrown. This browser allows the developer to quickly find all the places where a particular exception is thrown throughout the system.

Whereas organization 1 shows crosscutting of exceptions at a systemic level, organization 2 shows crosscutting of exceptions at the level of packages. On opening a package node, a list of exceptions declared to be thrown in that package is shown. Opening an exception node reveals classes and methods belonging to the corresponding package, much like an ordinary class browser. However, only those classes and methods in the package that propagate the exception corresponding to the expanded node are revealed.

Similarly, organization 3 shows crosscutting of exceptions at the level of classes. Organization 4 may, at first, appear to be less useful because it requires the developer to descend all the way to the level of individual methods to find out what exceptions are thrown. However, it does provide a useful exception-oriented view on classes and packages because it only shows a "filtered" package-class-member hierarchy where entities that do not propagate any exception are culled out.

3.2 Code-Base-Specific Browsers

In this section we discuss examples that highlight the utility of QJBrowser in allowing developers to define browsers, which are specific to a particular code base. In other words, in these examples, a browser's definition is inspired by some specific knowledge, which is closely linked to a particular code base. For this purpose, we shall consider a Java GUI framework for graphics, called JHotDraw [11]. JHotDraw provides several elements such as *Tools*, *Menus* and *Applications* for drawing and manipulating different *Figures*. The package includes some sample applications/applets that use these elements for different purposes, for example, a Network editor, a PERT editor etc.

Naturally, the aforementioned concepts — application, tool, menu, figure, etc. — and the relationships between them also play an important role in the JHotDraw code base. Specific bits of knowledge about how these concepts are implemented and how important relationships between them are expressed in the code will be the basis for the examples in this section.

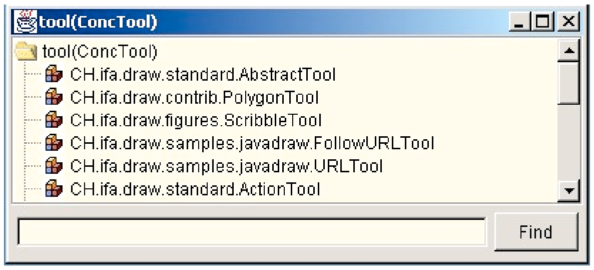
3.2.1 Tools Browser

The first browser shows all the "tools" in JHotDraw. Every tool in JHotDraw implements an interface called `Tool`, either directly or indirectly. This knowledge about how JHotDraw tools are implemented can be easily translated into a Prolog query for finding all tool classes. The resulting query constitutes the selection criterion for our first simple browser:

Selection:

```
shortname(ToolInterface,'Tool'),
subtype(Tool,ToolInterface),class(Tool).
```

Organization: `Tool`



```
tool(ConcTool)
├ CH.ifa.draw.standard.AbstractTool
├ CH.ifa.draw.contrib.PolygonTool
├ CH.ifa.draw.figures.ScribbleTool
├ CH.ifa.draw.samples.javadraw.FollowURLTool
├ CH.ifa.draw.samples.javadraw.URLTool
├ CH.ifa.draw.standard.ActionTool
```

Figure 3.2: Tool Browser

Because the notion of what constitutes a tool is a generally useful concept in JHotDraw's code base, and because tools will also play a role in the other JHotDraw-specific browsers shown in the remainder of this section, we will make a reusable abstraction that defines it. We can use Prolog's abstraction mechanism, namely rules, for this purpose. To this end, we define the following rule:

```
tool(X) :-
   shortname(ToolInterface,'Tool'),
   subtype(X,ToolInterface),class(X).
```

After defining this rule, we can use the query `tool(X)` to find all classes representing JHotDraw tools (see Figure 3.2). This defined abstraction constitutes a user defined extension to the query language, which is very useful because it improves the readability of the queries as well as the ease with which queries can be composed.

3.2.2 Tool Creation Browser

To explore the tools actually created in the sample applications/applets, in addition to knowing how tools are implemented in the package, we must also know how and where they are instantiated. By convention, JHotDraw applications/applets have a method usually called "createTools" which instantiates the tools to be used in that application/applet. We can define a rule to locate all `createTools` methods in the code, as follows:

```
createMethod(Method,Application) :-
       shortname(Method,'createTools'),
       method(Method,Application).
```

The instantiation is accomplished by simply invoking the constructor of the corresponding tools. We use this useful bit of knowledge, in conjunction with the rule defined above, to add another rule that defines the relationship between an application/applet and the tools it creates.

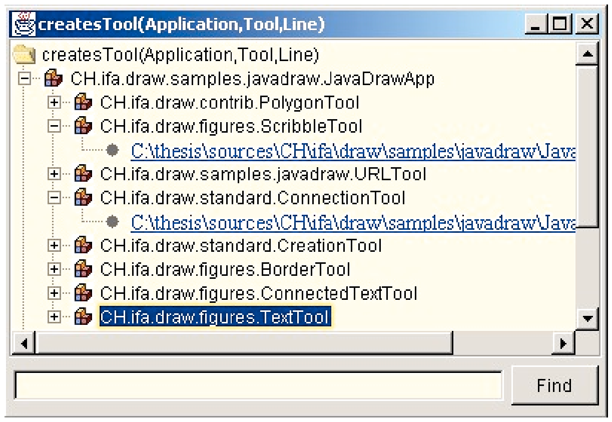
```
createsTool (Application,CreatedTool,Line) :-
     tool(CreatedTool),
     createMethod(CreateMethod,Application),
     constructor(CreatedTool,Cons),
     callinfo(CreateMethod,Cons,Line).
```

We can then use this rule, as explained below, to define a "tool-creation" browser (Figure 3.3 and Figure 3.4).

Selection:
`createsTool(Application,Tool,Line)`

Organization: `Application, Tool, Line`

Figure 3.3 shows the corresponding browser. It shows a hierarchy that lists the tools created by each application/applet in the code base. The last variable, `Line`, will appear as a hyperlink to the precise location in the code where the tool's constructor is being called. This view makes it easy to find out all the tools created by a particular application/applet in addition to the precise location of the tool creation call in the corresponding application/applet.

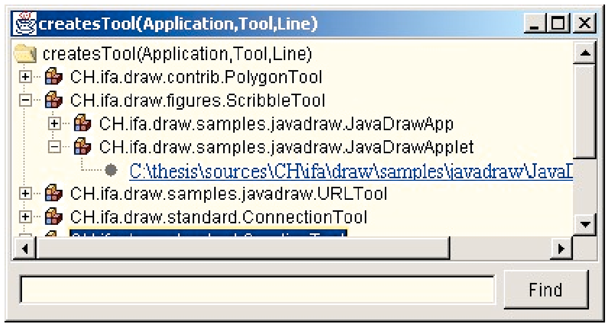


```
createsTool(Application,Tool,Line)
createsTool(Application,Tool,Line)
⊟ CH.ifa.draw.samples.javadraw.JavaDrawApp
   ⊞ CH.ifa.draw.contrib.PolygonTool
   ⊟ CH.ifa.draw.figures.ScribbleTool
        C:\thesis\sources\CH\ifa\draw\samples\javadraw\Java
   ⊞ CH.ifa.draw.samples.javadraw.URLTool
   ⊟ CH.ifa.draw.standard.ConnectionTool
        C:\thesis\sources\CH\ifa\draw\samples\javadraw\Java
   ⊞ CH.ifa.draw.standard.CreationTool
   ⊞ CH.ifa.draw.figures.BorderTool
   ⊞ CH.ifa.draw.figures.ConnectedTextTool
   ⊞ CH.ifa.draw.figures.TextTool
```

Figure 3.3: Tool Creation Browser – Flavor 1

Swapping the order of the first two variables in the organization criterion provides another useful view using the same query. This view is complementary to the previous one, making it easy to find out all the applications/applets that create a particular tool.

Organization: `Tool, Application, Line`



```
createsTool(Application,Tool,Line)
createsTool(Application,Tool,Line)
⊞ CH.ifa.draw.contrib.PolygonTool
⊟ CH.ifa.draw.figures.ScribbleTool
   ⊞ CH.ifa.draw.samples.javadraw.JavaDrawApp
   ⊟ CH.ifa.draw.samples.javadraw.JavaDrawApplet
        C:\thesis\sources\CH\ifa\draw\samples\javadraw\JavaI
⊞ CH.ifa.draw.samples.javadraw.URLTool
⊞ CH.ifa.draw.standard.ConnectionTool
```

Figure 3.4: Tool Creation Browser - Flavor 2

3.2.3 Figure Browsers

In this example, we define browsers around yet another JHotDraw-specific concept, namely "Figure". Figures are graphical objects that can be drawn and manipulated in applications/applets using appropriate tools. We want to produce different views that show the relationships between figures, tools and applications. To define an appropriate selection criterion, we need to make explicit some knowledge about the specifics of the JHotDraw's code base, by defining rules about them. First of all, there is the knowledge that classes representing "figures" are identifiable because they implement an interface called `Figure`. We express this knowledge as a rule:

```
figure(X):-
   shortname(Fig,'Figure'),
   subtype(X,Fig),class(X).
```

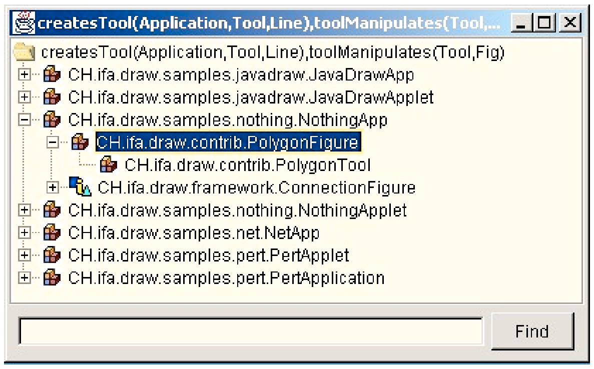
The connection between a figure and a tool that operates on the figure is also apparent in the code base. All tool classes in the code base encapsulate the figures that they can manipulate, as their data members. We turn this knowledge into the following rule:

```
toolManipulates(Tool,Figure) :-
    tool(Tool),field(Field,Tool),figure(Figure),
    type(Field,Figure).
```

We can now define several interesting views, which would reveal the figures used by the different applications/applets. The following selection criterion is the basis for several useful variations of "Figure Browser". We only show one of the possible variations here.

Selection:
`createsTool(Application,Tool,Line),`
`toolManipulates(Tool,Fig)`

Organization: `Application, Figure, Tool`



```
createsTool(Application,Tool,Line),toolManipulates(Tool,...
createsTool(Application,Tool,Line),toolManipulates(Tool,Fig)
⊞ CH.ifa.draw.samples.javadraw.JavaDrawApp
⊞ CH.ifa.draw.samples.javadraw.JavaDrawApplet
⊟ CH.ifa.draw.samples.nothing.NothingApp
   ⊟ CH.ifa.draw.contrib.PolygonFigure
        CH.ifa.draw.contrib.PolygonTool
   ⊞ CH.ifa.draw.framework.ConnectionFigure
⊞ CH.ifa.draw.samples.nothing.NothingApplet
⊞ CH.ifa.draw.samples.net.NetApp
⊞ CH.ifa.draw.samples.pert.PertApplet
⊞ CH.ifa.draw.samples.pert.PertApplication
```

Figure 3.5: Figure Browser

3.3 Examples Summary

The examples described in the preceding two sections illustrate some interesting points.

The first example illustrates how projecting two or more orthogonal organizational dimensions onto a tree can produce a family of useful views. Each view shows a different perspective on how elements in the different dimensions relate to one another. A combinatorial explosion of useful views results because there are many kinds of information that can be used to define organization along different dimensions, for example: packages, classes, method names, exceptions, calling dependencies, static type signatures, implemented interfaces, etc.

The second example illustrates how particularly interesting browsers can be defined using application-dependent knowledge. Such browsers organize specific kinds of code units based on high-level relationships between code base specific concepts (in the example: relationships between Applications, Tools and Figures). Since these relationships are often expressed in scattered places of the code base they would be rather hard to discover otherwise.

4. PRELIMINARY EXPERIENCE

In the previous sections, we discussed the utility of QJBrowser using examples. In this section, we discuss some preliminary experience using QJBrowser.

QJBrowser is a prototype that has not yet been used for any realistic software development. So it is too early to make any definite claims about its practical usability. However, to get some preliminary indications of the practical viability of our approach we tried performing two small development tasks on some software packages that we downloaded from the Internet.

While conducting the tasks, we took notes about the steps taken, the queries used and the reason for formulating those queries.

In the first task, our goal was to gain some general understanding of the structure and organization of JHotDraw [11], a Java GUI framework for technical and structured graphics. JHotDraw consists of 148 classes, 490 methods and a total of approximately 16000 lines of code. We chose JHotDraw because its code is known to rely heavily on some well-known design patterns [6]. We considered it an ideal test case because, despite its use of a number of good design principles, it is complex and understanding it requires the developer to identify and understand the various relationships that exist among the scattered elements in the code.

The second task was more directed. It involved making a change to the QJBrowser package itself by replacing its simple editor with a more sophisticated one downloaded from the Internet, called JE. JE has 236 classes, 786 methods and a total of approximately 13000 lines of code. This task consisted of changing JE appropriately to make it useable as an editor for QJBrowser. An editor for QJBrowser would have to provide a way to reify a file that was edited, so that the internal database maintained by QJBrowser can be kept up-to-date. The most essential part of the change task was to find out how to add a menu item in the editor to invoke this function. In addition, some parts of QJBrowser had to be changed to unplug the old editor and plug in JE.

For both tasks, we began by running the application and studying its external behavior before examining the source code. The next step was to search for the application entry points using a simple logic query for finding methods named `main`. Subsequently, we elaborated this query to define a view showing all methods that were transitively reachable (using the `callgraph` predicate) from the respective main methods[2].

From this point on, both experiments started to diverge. Nevertheless, in both cases, there was a tendency to formulate directed queries inspired by the results of the previous queries and a desire to further explore specific aspects in more detail.

Overall, we had a relatively positive experience with the tool although we did notice some usability issues with it.

Some things that were experienced as positive were:

- The ability to obtain different perspectives on query results by reordering variables: In the first task, we wanted to get an overview of the class hierarchy in JHotDraw. For this purpose, we used the primitive query `subtype(C,P)` and the two possible organization criteria, namely `C,P` and `P,C`, resulting in two different perspectives on the system. While the former perspective helped us in identifying the ancestors of particular classes, the latter helped us in getting a more conventional inheritance view.
- The ability to formulate specific queries inspired by the preceding results: Upon inspecting our notes, we found that we often formulated new queries to further explore some pivotal elements revealed by the preceding query. For example, from the class-hierarchy view described above, we found that only one class in JHotDraw, namely `CommandMenu`, derived from `JMenu`, the class that represents *menu* in Java's swing package. This led us to formulate more queries to explore the role of `CommandMenu` further, which gave us an overall understanding of the way menus are implemented in JHotDraw.
- The ability to repeatedly edit the selection criterion to reveal more or less information. For example, in the second task, in order to update QJBrowser's source model after editing a source file, we needed to get at the reference of the "file being edited", as maintained by the editor. By means of queries, we discovered that JE maintains the current file as a field, namely `CurrentFile` in the class `EditorFrame`. Our next immediate goal was to find all public methods that return this field. First, we queried for all the methods that accessed the field. The resulting view was not instantly helpful because it showed all the methods that accessed the field and not just the ones that returned it. To reduce the number of results, we refined the query to match only those methods that returned a type that was equivalent to that of `CurrentFile`. Finally, we refined the query even further because we were only interested in methods that had public access.

One of the problems that we noted with the tool was that, although the tool offers some support for composing queries by providing useful query fragments in a pop-up menu, composing queries that had the desired effect was not always easy and required some trial-and-error. Another issue encountered was related to the performance of the query engine. The execution of a query can take anywhere from a fraction of a second to a few minutes, depending on the complexity of the query and the number of results. Sometimes we lost patience waiting for all the results to be computed. Rather than wait for query execution to complete we would abort its execution and inspect a partially generated view.

5. RELATED WORK

5.1 QUERY-BASED TOOLS

In this section, we survey some approaches and systems that, like QJBrowser, have an expressive query language as an essential component in assisting developers to perform software engineering tasks. Of all the tools discussed here, some explicitly point out the usefulness of a query-based approach in support of aspects and multi-dimensional separation of concerns.

SOUL [21] is a logic meta-language that was developed for querying a Smalltalk image. ASTLOG [4] is a similar system developed for querying C and C++ abstract syntax trees. SOUL and ASTLOG are both query languages, whereas QJBrowser is not a query language, but uses a query language as a component to define navigable trees.

Coven [1] and the Gwydion [18] project provide a query mechanism for defining groups of editable code fragments. In Coven these are called "Virtual Source Files" and in the Gwydion project, they are called "Sheets". Both names refer to

[2] Because the current version of the tool does not support the definition of "recursive" browsers, the callgraph was collapsed into a single level of the tree. See also section 6 for more discussion about this limitation of the browser model.

a similar concept: a "virtual" set of related code fragments identified by a query. In these approaches, queries are used as a mechanism for selecting flat sets of code units, unlike the hierarchically organized results in QJBrowser.

GraphLog [3] is a logic query language in which queries and query results are represented as directed graphs. This eases the formulation of queries. GraphLog represents query results as directed graphs but does not provide the kind of support for viewing results from different angles by specifying organization criteria. A graphical query language like GraphLog could potentially be used in QJBrowser, instead of Prolog or TyRuBa, to facilitate the composition of queries.

Semantic Visualization Tool (SVT) [7] is a framework providing primitives for visualizing and browsing any kind of data present in program databases. Conceptually, SVT is by far the most similar to QJBrowser. In SVT, navigation and visualization primitives are defined as Prolog predicates. The program source code and runtime data are represented as Prolog facts in files. Data queries are used to generate specific views. Many of the underlying ideas in QJBrowser and SVT are highly similar. However, QJBrowser and SVT make different kinds of tradeoffs in terms of the degree of flexibility and ease of defining tools/views. SVT is more accurately characterized as an implementation platform for all kinds of visualization tools. Because SVT has a different goal than QJBrowser it offers much greater flexibility in the definition of a tool. However, defining a SVT tool requires considerably more effort than defining a QJBrowser view. Whereas QJBrowser just requires a selection criterion and an organization criterion, the configuration of a SVT tool involves defining views, view contents, view contexts, visual components, menus, actions, reactions, visual objects and content types.

5.2 ASPECT-ORIENTED TOOLS

In this section, we discuss how QJBrowser relates to some other research prototypes that were specifically designed to support crosscutting concerns, aspects and multi-dimensional separation of concerns.

HyperJ[14] supports the notion of a multi-dimensional concern space. HyperJ's notion of multi-dimensionality is highly similar to the multi-dimensionality of query results in QJBrowser. HyperJ supports encapsulation and composition of crosscutting modules called "hyperslices" and "hypermodules" but only has limited support for dynamic definition of hyperslices. QJBrowser, on the other hand, does not support encapsulation and composition but supports very dynamic definition of browser views.

Aspect Browser (AB) is a tool for assisting evolutionary changes by making code relating to a global change feel like a unified entity[8]. Although, this is also one of the goals of QJBrowser, there are some differences in the way both these tools are built and operate. First off, Aspect Browser uses a much weaker query language based on lexical pattern matching (like grep). Secondly AB visualizes query results using a *map metaphor*. That is, the query results are represented spatially in a map using a coloring scheme, indexing, folding and zooming, a "You are here" pointer etc. In QJBrowser, query results are represented as navigable trees with collapsible nodes. Each representation has its own pros and cons. However, no formal studies have been done to prove the benefits of one representation over the other.

Aspect Mining Tool (AMT) [9] extends Aspect Browser's query language to include type-based queries in addition to lexical matching. Although, it does not use an extensive map metaphor like AB, it does use a coloring scheme similar to AB to distinguish different concerns. QJBrowser differs from AMT in the way it visualizes query results. Another difference is that QJBrowser offers a more flexible query language than AMT. In addition, more kinds of information—such as calling dependencies, field references, constructor calls, application-specific semantics etc.—can be used in the queries.

Concern graphs [15] are used for abstracting the implementation details of a concern and showing the relationships among the different parts of a concern explicitly. In QJBrowser developers define views of interest using a flexible and expressive query language. In FEAT, developers add individual code fragments to their concern one by one. Both tools are to a large degree complementary in functionality. Whereas QJBrowser has a more powerful query language and allows intentional specification of organizational views, FEAT supports an extensional specification of individual concerns.

5.3 INTEGRATED DEVELOPMENT ENVIRONMENTS

Commercial IDEs such as VisualAge, Forte, JBuilder, etc. typically have a limited set of tools, such as class browser, package browser, class hierarchy browser etc. built in to them. These tools offer useful views that are mostly in line with the notion of modularity as defined by the supported programming language. In comparison, QJBrowser offers much greater flexibility and provides a way to define many different kinds of crosscutting and non-crosscutting views.

Eclipse [5] is an open extensible IDE with API's for plugging in a variety of development tools. Developers can build their own extensions and tools and integrate them seamlessly with the core IDE. To some extent, this allows them to customize the environment. Nevertheless, it requires significant effort, since the developers would have to build full-fledged plug-in tools in accordance with Eclipse specifications. In comparison, defining a view with QJBrowser merely requires formulating a query.

Because Eclipse is an extensible development environment, which comes with a high quality set of core Java development tools, it is an interesting idea to develop QJBrowser itself as an Eclipse plug-in. We are currently pursuing this idea and are developing Eclipse plug-in version of QJBrowser.

6. LIMITATIONS AND FUTURE WORK

QJBrowser offers a very expressive general-purpose query language. The disadvantage of this is that complex queries may take a long time to compute. Optimizing the query engine and building views more incrementally can improve the responsiveness of the tool. The new prototype we are building as an Eclipse plug-in will try to address some of these performance issues.

It is not possible with the current tool to define recursive browsers. For example a class hierarchy browser or a browser which allows a user to recursively descend deeper and deeper into a callgraph cannot be defined. This is a consequence of the choice to keep the browser definition process as simple as possible. Although it is possible to approximate recursive browsers by "flattening" them in different ways, the definition of recursive views would likely be a useful extension of the tool. We are currently considering this for the new version. It is not yet clear how to most easily support recursive browsers and whether it would be worth the extra complications to the browser-definition process.

Another limitation of QJBrowser's current implementation is that the user has to be familiar with logic programming to formulate queries. This requirement does not match well with the typical skills of an object-oriented software developer. One possible approach to alleviate the problem, could be to investigate better GUI support for editing queries. Another approach is to define a more intuitive syntax for logic queries, for example, graphical syntax as in GraphLog[3]. A third possibility is to look at non-logic query languages such as SQL, which are typically more familiar to developers.

Another idea for future research concerns extending the range of information that can be used in the query language. The current source model includes only facts about the source code that can be derived by simple static analysis. However, it can extended easily to incorporate information from a variety of other sources, such as information from dynamic analysis, JavaDoc comments, version management tools etc.

Finally, practical experience using the tool is still limited. The current implementation is only a prototype, and as such, it is of limited use in real software development settings. The Eclipse version of the tool will try to address this issue and to make it possible to conduct more realistic experiments and user studies.

7. SUMMARY

In this paper, we presented QJBrowser, an early prototype of a query-based browser tool that allows developers to dynamically define their own browsers. The browser's hierarchical view is obtained by seeing the result of a logic query as a multi-dimensional space, which can be projected onto a tree in several different ways. The definition of a browser consists of two parts: a query — the selection criterion — and an ordered list of variables occurring in the query — the organization criterion. Each variable can be thought of as representing an organizational dimension. The organization criterion defines how to combine multiple organizational dimensions into a single hierarchical view. When two or more orthogonal dimensions are involved, changing the relative order of the corresponding variables in the organization criterion will produce different useful perspectives. We can think of this as looking at a query result from a different angle.

The usefulness of a tool like QJBrowser was illustrated by means of two sets of examples. These examples illustrated that many kinds of information, including information derived from code-base-specific knowledge, can be used to define organizational structure. Because there are many kinds of information that can be used (e.g. packages, classes, method names, exceptions, static types, calling dependencies, object creation, inheritance, etc.) and because many of them are orthogonal to one another, the number of useful browsers that can be defined is virtually unlimited.

Integrating a tool like QJBrowser would enhance the ability of developers with crosscutting concerns in their code. Current IDE's indirectly support developers in dealing with crosscutting concerns by providing them with a limited set of good general purpose navigation and exploration tools. QJBrowser would enhance this ability by providing a single tool that can be used to dynamically define a large number useful general-purpose as well as code-base specific browsers with relative ease. This would enhance the IDE's ability to support developers in exploring the complex web of relationships that exists between various scattered elements of a code base.

8. ACKNOWLEDGMENTS

This work was supported by funds from the University of British Columbia (UBC), Object Technology International (OTI) and the National Science and Research Engineering Council of Canada (NSERC). We thank Jonathan Sillito and Gail Murphy for their insightful comments on a draft of this paper. We thank our colleagues and fellow students for many interesting discussions that have contributed to the QJBrowser tool and to this paper.

9. REFERENCES

[1] M. C. Chu-Carroll and S. Sprenkle. in Proceedings of the eighth international symposium on Foundations of software engineering for twenty-first century applications (November 2000), ACM SIGSOFT Software Engineering Notes, Volume 25 Issue 6. ACM Press, 88-97.

[2] Siobhán Clarke and Robert J. Walker. "Separating Crosscutting Concerns Across the Lifecycle: From Composition Patterns to AspectJ and Hyper/J." Technical report UBC-CS-TR-2001-05, Department of Computer Science, University of British Columbia, Vancouver, Canada, 23 May 2001.

[3] M. Consens and A. Mendelzon. GraphLog: A Visual Formalism for Real Life Recursion. in Proceesings of PODS, pages 404-416, 1990.

[4] R. F. Crew. ASTLOG: A language for examining abstract syntax trees. in Proceedings of the First Conference on Domain Specific Languages, pages 229–242, Oct. 1997.

[5] Eclipse Platform Technical Overview. Object Technology International, Inc, July 2001.

[6] E. Gamma, R. Helm, R. Johnson and J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Reading, MA: Addison-Wesley, 1995.

[7] Calum A. McK. Grant. Software Visualization In Prolog. Ph.D Dissertation. Queens' College, Cambridge. December 1999.

[8] W.G. Griswold, Y. Kato and J.J. Yuan. Aspect Browser: Tool support for Managing Dispersed Aspects. First Workshop on Multi-dimensional Separation of Concerns in Object-oriented Systems, OOPSLA 1999.

[9] J. Hannemann and G. Kiczales. Overcoming the Prevalent Decomposition of Legacy Code. Workshop on Advanced Separation of Concerns at the International Conference on Software Engineering (Toronto, 2001).

[10] W. Hürsch and C. V. Lopes. Separation of Concerns. Northeastern University technical report NU-CCS-95-03, Boston, February 1995.

[11] JHotDraw. http://www.jhotdraw.org/.

[12] G. Kiczales, J. Lamping, A. Mendhekar, et al., Aspect-Oriented Programming. in Proceedings of European

Conference on Object-Oriented Programming(ECOOP), June 1997.

[13] C. V. Lopes and M. Lippert. A Study on Exception Detection and Handling Using Aspect-Oriented Programming. In Proceedings of 22nd International Conference on Software Engineering. Limmerick, Ireland. ACM Press. June 2000.

[14] H. Ossher and P. Tarr. Multi-Dimensional Separation of Concerns using Hyperspaces. IBM Research Report 21452, April 1999.

[15] M. P. Robillard, G. C. Murphy. Concern Graphs: Finding and Describing Concerns Using Structural Program Dependencies. in Proceedings of ICSE 2002. (To appear).

[16] M. P. Robillard and G. C. Murphy. Designing Robust Java Programs with Exceptions. in Proceedings of the eighth international symposium on Foundations of software engineering for twenty-first century applications (November 2000). ACM SIGSOFT Software Engineering Notes, Volume 25 Issue 6, pages 2-10, ACM Press, 2000.

[17] SICStus Prolog. http://www.sics.se/sicstus/.

[18] R. Stockton and N. Kramer. The Sheets Hypercode Editor. Technical Report 0820, CMU Department of Computer Science, 1997

[19] K. De Volder. Type-Oriented Logic Meta Programming. Ph.D Dissertation. Vrije Universiteit Brussel, Programming Technology Lab*,* 1998.

[20] K. De Volder. TyRuBa. http://tyruba.sourceforge.net/.

[21] R. Wuyts. Declarative Reasoning about the Structure of Object-Oriented Systems. in Proceedings of TOOLS USA '98. pages 112-124, IEEE Press, 1998.