QJBrowser - A Query Based Approach to Explore Concerns

by

Rajeswari Rajagopalan

B.S., University Of Madras, 1999

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

Master of Science

in

THE FACULTY OF GRADUATE STUDIES

(Department of Computer Science)

We accept this thesis as conforming
to the required standard

The University of British Columbia

September 2002

Abstract

This dissertation presents a query-based browsing tool called QJBrowser that can assist developers in working with crosscutting concerns. Although there is no apparent limit to the number of different kinds of crosscutting views of source code that are potentially interesting to developers, many existing browser tools are capable of producing only a limited set of pre-defined views. This is because the logic to locate and display code units is typically pre-programmed into these tools and users have only limited control over it.

QJBrowser addresses this problem by providing a mechanism by which developers can *dynamically* define interesting views. The goals of QJBrowser are the following:

- It must be configurable enough to define a multitude of different kinds of views on source code.
- It must be simple enough so that a developer can define views *on demand.*
- It must provide an interface that is familiar to software developers.
- The query language that it provides must be extensible.
- Finally, it must provide assistance for the exploration of crosscutting concerns in source code.

Besides presenting the motivation and concepts of QJBrowser, this dissertation intends to provide evidence by using examples and observations from preliminary experience, that QJBrowser in fact meets these aforementioned goals.

Contents

List of Tables

List of Figures

Acknowledgements

First and foremost, I am extremely grateful to my supervisor Dr. Kris De Volder. He offered me guidance, support and ideas whenever I needed them. I thank my husband Narayan Krishnamoorthy and my parents for being very supportive of me throughout my Masters program. I thank Dr. Gail C. Murphy and my colleague Jonathan Sillito for their valuable comments on the paper that was written on this work. I am also thankful to Dr. Alan Wagner for agreeing to review my dissertation. Last but not the least, I would like to thank the Department of Computer Science of the University of British Columbia for their excellent facilities that I could use for this research.

RAJESWARI RAJAGOPALAN

The University of British Columbia
September 2002

Chapter 1

Introduction

Another common error is to combine two simple functions into one component because the functions seem too simple to separate. For example, one might be tempted to combine synchronization with message sending and acknowledgment in building an operating system ....If one later encounters an application in which synchronization is needed very frequently, one may find that there is no simple way to strip synchronization out of message sending routines. - D. L. Parnas [26]

It has been known from the early days of software engineering that modularization, if done properly can improve comprehensibility and maintainability of programs significantly. The modularization techniques used in the early seventies involved decomposing a program recursively into steps and making each major step in the program, a module. Parnas proposed the concept of "information hiding" as a criterion for modularization of programs in his classic paper [25]. According to him, at the inception of program development, one must list the major design decisions that are likely to change in the future and hide each such decision inside a module. In this way, the issues/concerns addressed by each module in a program are *cleanly separated* and well localized. Therefore, future changes to any design decision will not require invasive changes throughout the program. Unfortunately, concerns in a program are frequently mutually dependent and overlapping. Therefore, in practice, clean separation of *concerns*, is seldom if ever achievable. This is the root cause of the problem that we address in the dissertation. We present the problem, our motivation, our thesis statement and our approach to address the problem in this chapter.

1.1 Background

In this section, we describe some of the terms and concepts essential for following the motivation behind this dissertation. These include the terms *concerns, crosscutting* *concerns, tangling and scattering of code* etc..

1.1.1 What are concerns?

In software engineering, although the term *concern* is very widely-used, it is not clearly defined. In the context of this thesis, we think of a concern as any goal, concept or task to be accomplished. [21] categorizes concerns as *logical* and *physical* concerns. In this thesis, we preserve the same distinction between the two kinds of concerns:

Physical Concern Any concern that is a part of an actual system, such as a hardware or software unit, is a physical concern.

Logical Concern Any conceptual consideration for the system, such as issues, features etc., are logical concerns.

According to this definition, a concern may exist in any phase of the software lifecycle. For example, a feature in the Requirements Analysis stage such as *Logging* or *Distribution* is a concern. A high-level concept like a *Design pattern* [13] in the Design or Implementation stage is also a concern. Considering an Object-oriented (OO) software system, these are some examples of the second kind of concerns, namely logical concerns, since they do not exist in the system as a "unit". On the other hand, a *class* in an OO system is a physical concern existing as a concrete unit in the source code, trying to accomplish a certain task.

In the remainder of this dissertation, whenever we refer to the term concern generally, we mean both types of concerns. When needed, we make a clear distinction between them.

1.1.2 What are crosscutting concerns?

Separation of concerns is the process of identifying, encapsulating and manipulating parts of software that are relevant to a particular concern [24]. Clean separation of concerns can enhance software trace-ability, maintainability, comprehensibility, changeability etc.

Object-orientation has greatly improved separation of concerns in software systems. However, achieving good separation of concerns in practice is still very hard because:

- Concerns are mutually dependent and overlapping. The code that has to implement *Distribution* might need to perform *serialization, error-handling, synchronization* etc. as well.

- It is not always possible to express design-level concerns as code units using abstractions provided by programming languages. Requirements deal with features, while code deals with modules. Modules might not map one-to-one to features stated in the requirements. As a result, a single feature may be implemented in more than one module or a single module may implement parts of more than one feature. The former property is called scattering and the latter, tangling. [22]

- Concerns may change over time. There are several reasons for this, such as addition or deletion of functionality, changes in the environment in which the software operates, mistakes learned from experience and the need to restructure code to remedy them.

As a result, some concerns crosscut natural system modularity. Such concerns are called crosscutting concerns. Classic examples of crosscutting concerns are *distribution, synchronization, exception/error handling, serialization* and *logging.*

Crosscutting concerns result in implementations that are not only tangled but also scattered across multiple modules in the system [22]. Code scattering and tangling make it more difficult to comprehend, modify and maintain object-oriented software systems. For instance, when making a change in software, since all code units related to the change are not present in the same module, it is difficult to ensure that:

- modifying one part of the system does not render it inconsistent with the rest and

- all parts of the system that are affected by the change are modified.

Therefore tools that assist developers in working more effectively with crosscutting concerns are highly desirable.

1.2 Motivation and Thesis Statement

In the preceding section, we stated that it is desirable to have tools that support developers when faced with crosscutting concerns. Integrated Development Environments are the most widely-used kind of tools developers use today.

Current state-of-the-art IDEs already help developers to deal with crosscutting concerns by providing effective tools to explore a code base. An IDE provides a set of programming-language-aware tools that offer different kinds of *views* that help in exploring certain structural and semantic relationships between pieces of code.

Our work was motivated by the following observations about modern IDEs:

1. They offer different kinds of views using different tools built into them. As such the number of views is limited by what the developers of the IDE chose to provide.

2. In principle an IDE can support more types of views by providing more built-in tools. However, there is a practical limit on the number of tools that can be developed and shipped with an IDE. Building a new tool for each potentially interesting view is very costly and impractical. Besides, every added tool adds to the overall complexity of the IDE.

3. The views offered by the different tools of an IDE are typically closely related to the modularity mechanisms of the underlying programming language. For example, a modern IDE for an Object-oriented programming language might include a *Class Browser* and a *Class-Hierarchy Browser* that allow developers to view and navigate code in terms of classes and inheritance relationships.

   However, many different kinds of concerns may be relevant to developers at different times [24]. Each of these concerns may benefit from different kinds of views that may even crosscut the natural modularity of the system. Since such crosscutting views do not align well with the modularization mechanisms of the language for which a typical IDE is built, most IDEs do not offer tools to support such views.

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 that are specific to that framework. It is hard to build such code-base-specific tools into an IDE.

Therefore it is not possible with most modern IDEs to *dynamically* obtain a variety of customized, crosscutting views of a system, although such views can greatly help a developer in working with crosscutting concerns. This is largely because of the limited configurability offered by the tools built into the IDEs. We think that a generic tool that can be configured with general or application-specific parameters to generate many different views "on demand" would enhance an IDE's capability to explore crosscutting concerns in a code base.

One key issue in the design of such a tool is the trade-off between flexibility and simplicity. An effective tool offers a configuration mechanism that is conceptually simple and, at the same time, flexible enough to allow the creation of a broad set of useful views. Some IDEs like Eclipse offer a high degree of customizability at the expense of ease of customization. Eclipse is an open extensible IDE that can be extended by writing plug-ins in Java [33] programming language. In Eclipse, a developer who wants a customized view would have to write a tool in adherence to Eclipse's plug-in API, compile it and integrate it with the core IDE. This might involve significant effort and may not be useful for defining views dynamically.

This dissertation presents a prototype tool called QJBrowser that we built to validate the following thesis statement:

A query-based browsing tool with an extensible query language can be conceptually simple yet configurable enough to dynamically generate a wide variety of interesting views that can help a user in exploring crosscutting concerns in her software.

1.3 QJBrowser

QJBrowser is a tool that can be configured to generate views relevant to a user's interests using *queries* against a *source model.* At this point, it is sufficient to know that a source model is a database of information about source code, extracted automatically and capable of being augmented by the user. The queries are written in an expressive language and select elements that constitute a view. The view generated by the tool is organized as a tree and is navigable like any conventional browser. This configuration mechanism is conceptually simple and at the same time flexible enough to generate a multitude of views. It provides a cost-effective way to define new views because defining a view involves little more than the formulation of a query.

The query entered by the user is called the *selection criterion* for the view. It is a query against a semantically-rich source model containing different types of information such as static types, calling dependencies and inheritance relationships. The source model is automatically generated by a source code analysis tool. It can also be appended to by the user specifying information that cannot be derived directly by the analysis tool. For example, design rules, conventions and patterns can be established by the user and queries can be formulated to examine their presence in source code. Therefore, in QJBrowser, queries use not only a variety of data about the source code, but also user-defined information that can describe application- or domain-specific semantics. In this way, QJBrowser stands out to be a tool that is highly configurable.

Apart from the selection criterion, another element that goes into the formation of a view is the *organization criterion.* It is a mechanism for the user to specify the order in which the elements in the view must be organized. It is simply a comma-separated list of query variables[1] representing the different elements that will form the generated view. We will describe this in greater detail in Chapter 2.

The organization criterion is another factor contributing to QJBrowser's configurability. The tool does not yield a flat set of results, but a neat browsable view, the organization of which is also configured by the user herself. In spite of the high configurability, the tool is still relatively simple to use. A new view is generated with just two parameters: a query and a list of variables in the query.

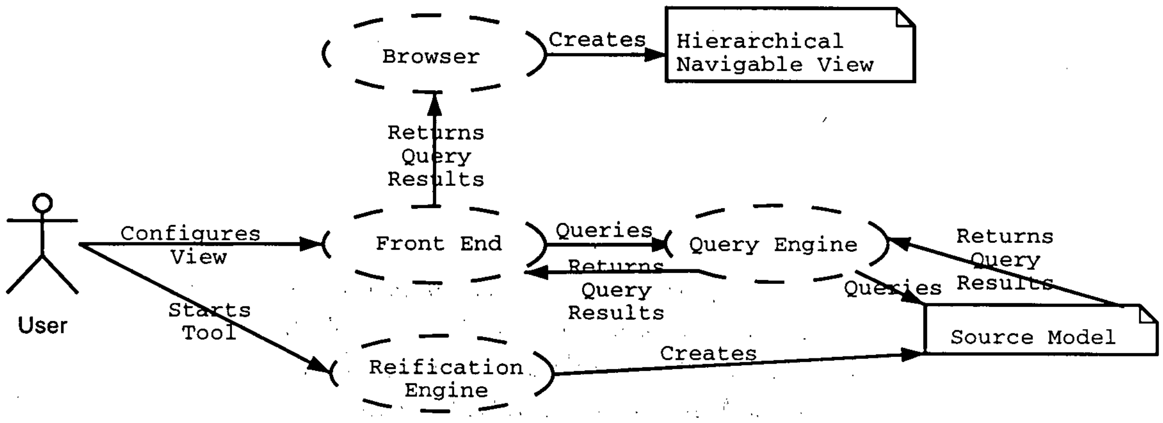


Figure 1.1: Process of Using QJBrowser

The entire process of using QJBrowser to generate views is illustrated in Figure 1.1.

1.4 Validation

This dissertation will provide validation for our thesis statement in two ways.

First, we will present a number of example scenarios that use QJBrowser. These examples are meant to illustrate that the tool achieves the right kind of trade-off between simplicity and configurability. They show that a wide variety of views can be defined with relative ease. Some of the views are similar to views offered by the tools included in a traditional IDE. Other examples show crosscutting views as well as views that are specific to a particular code base. Although the examples allow us to provide some conceptual arguments in support of the thesis, by themselves they provide little insight into how the tool would be used in practice.

[1]A query variable is an identifier starting with an uppercase letter. Please refer to appendix B.

Therefore, as a second part of the validation, we discuss two simple case studies using the tool for two development tasks. These provide some preliminary indications about the practical usability of the tool. Both the examples and the case studies are intended to illustrate that QJBrowser can produce different crosscutting views of a system, thereby helping developers in dealing with crosscutting concerns effectively.

1.5 Summary

In this chapter, we outlined the motivating ideas behind QJBrowser. QJBrowser was motivated mainly by the limitations of modern IDEs to provide customized or crosscutting views on their code base. QJBrowser addresses this issue by allowing developers to configure a view with a query. This assists developers in dealing with crosscutting concerns by allowing them to explore a code base more effectively.

1.6 Dissertation Overview

The rest of this dissertation is organized as follows. Chapter 2 introduces the concepts of QJBrowser and demonstrates the usage of the tool using a few basic examples. Chapter 3 provides support for our thesis by presenting more examples as well as observations from our own experience with the tool. Chapter 4 discusses other related approaches to deal with crosscutting concerns. Finally, chapter 5 summarizes the dissertation. It also lists the limitations and ideas for further improvement of the tool.

Chapter 2

QJBrowser

QJBrowser is a prototype tool that we developed in order to validate our thesis. We presented our thesis in Chapter 1. In this chapter, we explain QJBrowser in detail. We discuss the fundamental concepts behind it followed by examples of how to use it. We also outline the design decisions that we made while building the tool. Finally, we discuss its implementation in brief.

2.1 Design Goals of QJBrowser

We built QJBrowser to validate our thesis, which we presented in section 1.2. Our thesis forms the basis for the design goals of the tool. We present the design goals of the tool below:

- It must be a query-based browsing tool.

  Users should be allowed to browse their source code by defining their views using queries. In addition, the views obtained must have a conventional *browser-like* interface that is familiar to most developers. The tools offered by most modern IDEs that are very widely used for working with program concerns provide a simple tree-shaped view, which can be clicked on to navigate the code. This is the kind of user interface that we mean by a *browser-like* view.

  Some visualization tools offer very complex views, such as views consisting of graphs and arcs. These views may be interesting for a number of development tasks. However, graphical visualization of program data is out of the scope of our tool. We target the domain of code browsing tools with a conventional browser-like user interface that could potentially be plugged into an IDE to replace a number of specific tools that offer specialized views pre-programmed into the tool.

- It must be configurable enough to generate a wide variety of interesting views.

  The tool must not limit users to only viewing source code units connected by a pre-defined set of patterns. It must let users incorporate a wide range of interesting information about the underlying system in defining their views.

- The query language that it offers must be extensible.

  A query language that is not extensible tends to limit the kinds of views that can be obtained from the tool.

- It must be simple to use.

  By simplicity, we do not mean the complete absence of a learning curve. We intend our tool to be used easily to generate views "on demand". We limited the scope of our tool to browsing rather than graphical visualization simply because configuring the appearance and contents of a fancy graphical view (that is common in visualization tools) might involve more effort than a simple hierarchical browser view. We feel that configurability of the tool must not be achieved at the expense of its ease-of-use. Therefore, we intend our tool to strike a balance between configurability and simplicity.

- It must help a user work with crosscutting concerns in her software.

  The ultimate aim of the tool is to assist developers by further enhancing the ability of program development tools such as IDEs to deal with crosscutting concerns in software.

These goals served as our guidelines while designing and building the tool.

2.2 Basic Concepts

There are some important concepts of QJBrowser that one must know before working with the tool. The goal of this section is to discuss them briefly. The next section will describe their role in QJBrowser with examples.

QJBrowser is a tool that helps a user in obtaining a number of crosscutting views of her system. To obtain a view, the user must define it appropriately.

There are two parameters in the definition of a view in QJBrowser:

Selection Criterion - It is a query that selects the entities that should be part of the view.

Organization Criterion - It defines how the entities in the view must be organized in relation to one another. In the current version of the tool, it is simply a comma-separated list of query variables.

The selection criterion query is executed by a query engine (See section 2.5.3) against a *source model.* The source model contains information about the underlying system. In the current version, the source model is generated automatically by static analysis of source code during initialization. It contains information such as static types, calling dependencies and inheritance relationships. A user can add new information to it using the user interface of the tool. This will be discussed in sections 2.3.3 and 2.5.2. The expressions available in the query language for writing selection criterion queries is dependent on the data in the source model.

The next section concretizes these concepts by discussing the usage of QJBrowser using examples. The examples are based on the current implementation of QJBrowser. Hence understanding them would require some knowledge of the language offered by the tool for writing queries. The query language used in the prototype tool is Prolog [2]. A reader who is not familiar with the concepts and syntax of Prolog is referred to Appendix B for a brief introduction.

2.3 QJBrowser: Working

The section aims to show the reader how to use QJBrowser for the following set of tasks:

- Configuring the tool using a simple query.
- Configuring the tool using a complex query.
- Extending the source model.
- Configuring the tool for an application-specific view.

To this end, we provide four examples centered on exception propagation and handling in Java code. Exception propagation and handling mechanisms are notorious for being difficult to manage in Java, partly due to their crosscutting nature [27]. Therefore, if one needs to alter the exceptions or their propagation/handling mechanisms in code, she might have to make invasive changes to several modules. Tracing the exception flow or locating the different exception handlers, in order to effect the necessary changes in them, would require laborious exploration of source code.

In our first example, our aim is to show how a developer can define a view using selection and organization criteria. In this example, she is interested in only the propagation of exceptions in her system. We will describe how she defines a view that displays the exceptions propagated in her system along with the methods that propagate them. We will also show how specifying different organization criteria results in different "perspectives" of a view.

Our second example will build on the the first example to define a view that locates the exception handlers in a system. The goal of this example is to show the reader how to combine queries to define complex views.

The third example describes how to extend the tool's source model with extra information, which could be domain-specific. Lastly, our fourth example discusses how to define views that are specific to an application or domain.

2.3.1 Example 1: Configuring QJBrowser using a simple query

The goal of this example is to demonstrate the usage of selection and organization criteria for defining a view. In this example, our developer wants to find methods that throw exceptions in her system. As we said before, in QJBrowser, a view is defined using two parameters:

- Selection criterion: A query that selects the relevant elements.
- Organization criterion: A comma-separated list of variables in the query, which dictates the *categorization* of elements in the view.

Our developer must first frame her selection criterion query. The reader might remember that this query depends on the information in the source model. In the current version of the tool, the source model has a relation named `exception` that connects an exception to the method that throws it. Its structure is `exception(ThrowingMethod, ThrownException)`. For an analysis of the different kinds of relations in the source model, please refer to section 2.5.3.

Our developer can use this query as her selection criterion:

```
exception(ThrowMethod, XCeption).
```

The result of the execution of this query is a set of pairs, each containing bindings for `ThrowMethod` and `XCeption`, such that the method bound to `ThrowMethod` throws the exception bound to `XCeption`.

```
ThrowMethod:  testpackage.Foo.doA()
XCeption:  testpackage.exceptions.X
ThrowMethod:  testpackage.Foo.doA()
XCeption:  testpackage.exceptions.M
ThrowMethod:  testpackage.Foo.doB()
XCeption:  testpackage.exceptions.X
ThrowMethod:  testpackage.Foo.doB()
XCeption:  testpackage.exceptions.Y
ThrowMethod:  testpackage.Foo.doD()
XCeption:  testpackage.exceptions.Y
ThrowMethod:  testpackage.Foo.doF()
XCeption:  java.rmi.RemoteException
         ....................
```

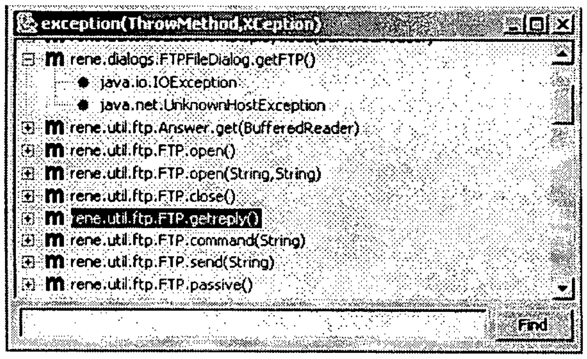
Figure 2.1: Flat Set of Results for Exception Propagating Methods Query

A selection criterion query by itself just produces a flat set of results. In this example, that would be a listing of names of all the exception-propagating methods followed by all the propagated exceptions, as shown in Figure 2.1. It does not impose any organization on the results, and hence might be very difficult to understand, especially if the result set is large.

Besides, as previously stated, one of the goals of QJBrowser is to generate a view that resembles a conventional browser. In a conventional browser, elements are organized hierarchically as a tree. In order to organize the results as a tree and abstract and hide data until needed, QJBrowser uses the second view-definition parameter, namely organization criterion.

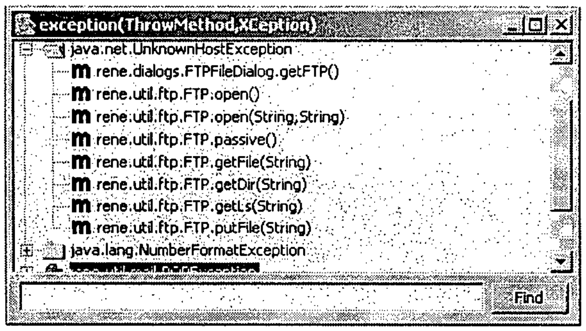
In the current implementation, the organization criterion is just a comma-separated list of query variables. The developer can specify the variables in any order and may choose to leave out some of them. The order in which the variables are specified determines how the elements in the resulting view will be classified and grouped.

For this example, one possible organization criterion is (`ThrowMethod,XCeption`). This criterion would *categorize* exceptions according to the methods that propagate them, as shown in Figure 2.2(a). By categorization, we mean how the elements in a view are *grouped* together. For example, in a traditional class browser, methods and member variables are categorized according to the classes in which they are defined. Similarly in a file system browser, files are


exception(ThrowMethod,XCeption)

- rene.dialogs.FTPFileDialog.getFTP()
  - java.io.IOException
  - java.net.UnknownHostException
- rene.util.ftp.Answer.get(BufferedReader)
- rene.util.ftp.FTP.open()
- rene.util.ftp.FTP.open(String,String)
- rene.util.ftp.FTP.close()
- rene.util.ftp.FTP.getreply()
- rene.util.ftp.FTP.command(String)
- rene.util.ftp.FTP.send(String)
- rene.util.ftp.FTP.passive()

(a)


exception(ThrowMethod,XCeption)

- java.net.UnknownHostException
  - rene.dialogs.FTPFileDialog.getFTP()
  - rene.util.ftp.FTP.open()
  - rene.util.ftp.FTP.open(String,String)
  - rene.util.ftp.FTP.passive()
  - rene.util.ftp.FTP.getFile(String)
  - rene.util.ftp.FTP.getDir(String)
  - rene.util.ftp.FTP.getLs(String)
  - rene.util.ftp.FTP.putFile(String)
- java.lang.NumberFormatException

(b)

Figure 2.2: Exception Handler Browsers

categorized according to the directories in which they reside.

Another useful organization criterion for the same query is (`XCeption,ThrowMethod`). This will result in a view that categorizes methods according to the exceptions that they handle (shown in Figure 2.2(b)).

Since both views display the same data, we can think of these two organizations as a way to view the query results from different perspectives. Note that, the perspectives in both Figure 2.2(a) and Figure 2.2(b) are useful. Each one reveals different kinds of information more clearly. In Figure 2.2(b), it is easy to find out all locations where a particular exception is being thrown in the system but it is not easy to find out a list of exceptions that are thrown by a particular method. The latter is more easily found in the browser in Figure 2.2(a). We call this generated view a *browser* because it has all characteristics of a typical source-code browser, including the ability to expand/collapse nodes, navigate to the source code corresponding to a node etc.

2.3.2 Example 2: Configuring QJBrowser using a complex query

This example aims to show how a user can combine queries using operators provided by the query language to define views that cannot be defined using simple queries. Some views cannot be defined using simple queries because of either their inherent complex nature or the unavailability of suitable predicates in the source model that can capture them. In this section, we describe a view that requires certain information not available in the current implementation of the source model and how a developer can instead use a combination of the available information to define that view.

In this example, our developer wants to view where exceptions are handled in her system. In the ideal case, she would use a query `catch(Method,XCeption)`, where `catch` is a relation in the source model that makes explicit the exact location of catch statements in a program. This query would bind the variables `Method` and `XCeption` to pairs of values such that, in each pair, the method bound to `Method` catches the exception bound to `XCeption`. Unfortunately, in the current implementation of the source model, such a relation is not available[1].

Therefore our developer has to follow a less straightforward method to define her view: In Java code, any *checked exception* that is thrown by a method must be declared in the method's signature. Hence, when a method does not throw an exception that reaches it in a calling sequence, one can say that it catches/handles that exception[2]. Our developer can use this knowledge to define her view.

The query `exception(ThrowMethod,XCeption)`, in our last example, locates the methods that propagate exceptions in a system. One could refine it in conjunction with static callgraph information available in the source model. The query thus formulated is shown below. Please refer to Table A.1 for the exact meaning of the relations in the query.

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

The execution of the above query results in a stream of solutions that are triples of values that match the variables in the query. In each solution, the first two goals of the query make sure that the value bound to `ThrowMethod` throws an exception bound to `XCeption` and is called by a method bound to `CatchMethod`. The third goal filters out from the set of bindings for `CatchMethod`, the methods that propagate the exception `XCeption` instead of handling it. Therefore any solution bound to `CatchMethod` will be the methods that handle exceptions in the system.

2.3.3 Example 3: Extending the source model

One of the merits of QJBrowser is the extensibility of its source model. The range of information that can be incorporated in user queries can be enhanced by augmenting

[1]It however would not be very difficult to make our static analysis tool add such a relation to the source model. Whenever a catch clause is encountered in the code, the tool would write the exact location and the name of the exception being caught, to the source model.

[2]It should be noted this method holds good only for checked exceptions. Detection of unchecked exceptions would require runtime analysis of the code, which is not available in the current version of the tool.

the source model with extra information.

In the current implementation, there are two ways of extending the source model:

Adding facts - For example, consider the source model in Figure 2.3. One can extend it with a new fact/relation such as

```
factory(MazeFactory).
```

This relation states that the class `MazeFactory` obeys the design pattern *Factory* [13]. Similarly, any programmer annotation about a piece of code can be introduced into the source model using new facts.

Adding rules - Rules can be thought of as abstractions in the source model that name user queries. For example, we can define a new rule called `exception-handler` that will abstract the selection criterion query of our last example, as follows.

```
exception-handler(CatchMethod,XCeption) :-
    exception(ThrowMethod,XCeption),
    callinfo(CatchMethod,ThrowMethod,_),
  \+(exception(CatchMethod,XCeption)).
```

As can be seen, the name `exception-handler`, relates two variables in the query, namely `CatchMethod` and `XCeption`, which are the two interesting variables for the exception handling example. The goal `exception-handler(X,Y)` is satisfied *only if* the query that it abstracts is satisfied.

```
class(MazeFactory).
method(makeMaze,MazeFactory).
method(makeWall,MazeFactory).
exception(makeMaze,MazeTypeNotKnownException).
exception(makeMaze,MazeComponentTypeNotKnownException).
exception(makeWall,WallTypeNotKnownException).
```

Figure 2.3: Sample Source Model

A rule defines an extension to the query language. Such an extension provides a convenient shorthand for long and complex queries and can serve as an abstraction for high-level concepts like *design patterns, design rules, conventions* etc.

Please refer to section 2.5.2 for a discussion of the user interface component of the prototype tool that aids in editing the source model.

2.3.4 Example 4: Configuring QJBrowser for an application-specific view

The example discussed in section 2.3.2 was used to generate a general-purpose exception handler browser. The view and its definition parameters (selection and organization criteria) were not restricted to a single application. However, it is possible to generate views specific to an *application* or *domain* using code-base-specific queries.

For example, let us suppose that the developer in our example is working on a *drawing application* that uses a number of *graphics tools.* If in her application, all graphics tools have a certain *signature* - say for instance, all of them derive from an abstract class called `Tool` and implement certain specific *listener* interfaces - our developer can then use this knowledge to form an abstraction that defines a *Tool.* She can then use this abstraction in her exception-handler query to find the exception handler methods in only her tool classes, as shown below.

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

```
exception(ThrowMethod,XCeption),
callinfo(CatchMethod,ThrowMethod,_),
tool(Tool),context(CatchMethod,Tool)
\+(exception(CatchMethod,XCeption)).
```

We will discuss more such code-base-specific browsers in section 3.1.

2.4 Design Decisions

In the preceding sections, we described the concepts of QJBrowser and showed how they can be used to define multiple crosscutting views of source code. While designing and building the tool, we made a number of decisions with the goals of the tool in mind. The following are two of our most important design decisions:

- We chose to use Prolog as the query language for QJBrowser. Prolog offers the full expressive power of a Turing-complete programming language for defining new predicates. This choice was made to support the goal that the query language provided must be extensible.

- The organization criterion in QJBrowser is a very simple list of variables separated by commas, which specifies the categorization order of the elements in a view. It should be noted that not letting users configure every aspect of the view restricts configurability to some extent. However, one of our goals is to keep the tool simple enough to generate a number of views on demand. Therefore, we decided to keep the organization criterion simple.

2.5 Implementation

In this section, we discuss the implementation aspects of QJBrowser. The implementation of the tool is in Java. The tool's query engine uses Prolog for its search strategy. We also developed a version of the tool whose query engine uses another logic programming language called TyRuBa [34] instead of Prolog. The version of the tool that uses TyRuBa as its query language was not used very much in our validation phase and hence this dissertation talks only about the Prolog version.

In this section, we discuss the important components of the tool followed by some of the highlights of our implementation.

2.5.1 Components Overview

The core components of QJBrowser are shown in Figure 2.4. The source model is a database of information about the underlying system. The *reification engine* transforms source code into an appropriate format in the source model. The *query engine* reads the source model to answer queries typed by users. The front end of the tool is made up of the following components -

user interface - The *user interface* allows users to configure the tool with the selection and organization criteria for their views. It also lets users communicate with the source model editor component described below.

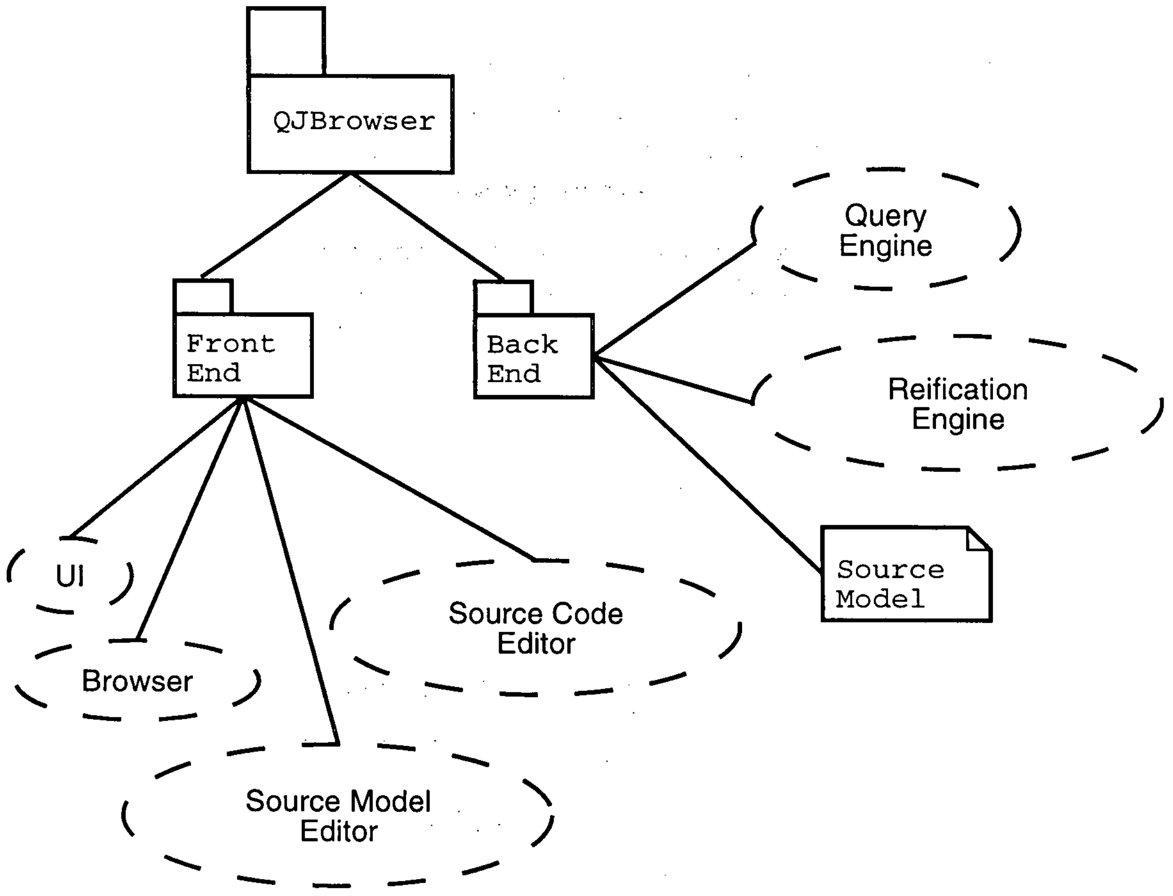


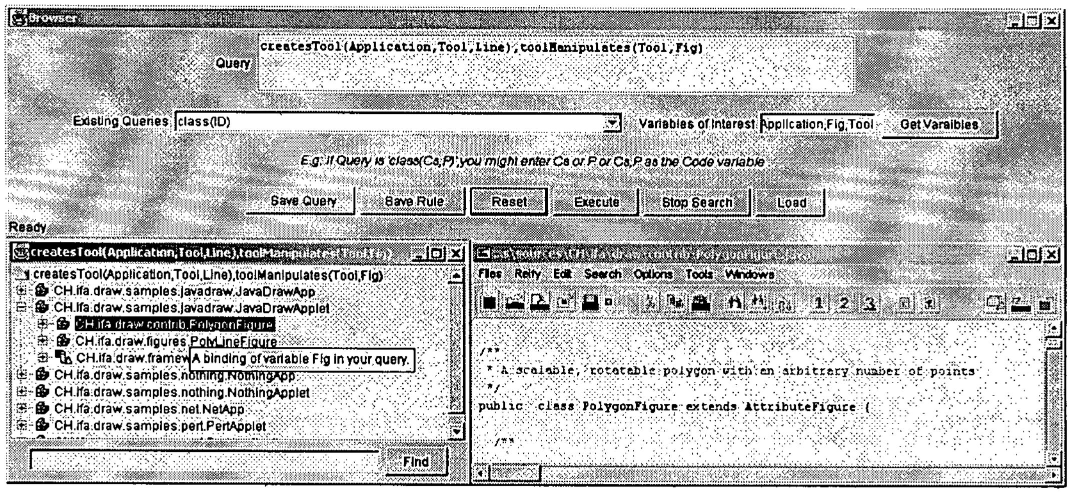
Figure 2.4: Core Components of QJBrowser

source model editor - The *source model editor* provides a way for users to augment the source model with additional knowledge.

source code editor - The *source code editor* is used to edit the source code of the system being browsed.

browser - The *browser* component adds the features of a conventional browser, such as navigability, to the generated views.

In the following sections, we describe each of these components in detail.


Query: createsTool(Application,Tool,Line),toolManipulates(Tool,Fig)

Existing Queries: class(ID)

Variables of Interest: Application,Fig,Tool

E.g: If Query is 'class(Cs,P)',you might enter Cs or P or Cs,P as the Code variable

Ready

```
createsTool(Application,Tool,Line),toolManipulates(Tool,Fig)
CH.ifa.draw.samples.javadraw.JavaDrawApp
CH.ifa.draw.samples.javadraw.JavaDrawApplet
    CH.ifa.draw.contrib.PolygonFigure
    CH.ifa.draw.figures.PolyLineFigure
    CH.ifa.draw.framew
CH.ifa.draw.samples.nothing.NothingApp
CH.ifa.draw.samples.nothing.NothingApplet
CH.ifa.draw.samples.net.NetApp
CH.ifa.draw.samples.pert.PertApplet
```

A binding of variable Fig in your query.

```
/**
 * A scalable, rotatable polygon with an arbitrary number of points
 */
public  class PolygonFigure extends AttributeFigure {

  /**
```

Figure 2.5: QJBrowser: User Interface

2.5.2 Front End

The front end of the tool is responsible predominantly for interfacing with users. This section discusses components that constitute the front end of the tool.

User Interface

A screenshot of QJBrowser's user interface is shown in Figure 2.5. It consists of a dialog box from which it is possible to launch different views by entering the parameters that define them. The text-box named *Query* is where the query representing the selection criterion is entered. The entry in the box named *Variables of interest* represents the organization criterion. The specification of organization criterion is optional in the current version of the tool. When a user does not specify an organization criterion, the tool automatically considers all the query variables as interesting and displays them in the order that they appear in the query.

A developer can expand or collapse nodes by clicking in the tree view. By double-clicking on a node, she can open a source code 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 drop-down menu box named *Existing Queries* from which useful expressions can be selected and appended to the *Query* box. These expressions are obtained during the initialization of the tool by parsing the source model. Apart from the ones loaded automatically by the tool, the user can load other files containing additional data by clicking on the *Load* button.

The button *Save Rule* interfaces with the *Source Model Editor* component, which is described in section 2.5.2. Its purpose is to add extra information to the tool's source model. The button *Save Query* provides a mechanism to store frequently used queries in the *Existing Queries* box. It is especially useful in the case of long derived queries that are formed by combining two or more simple queries.

Source Model Editor

The *source model editor* component is responsible for augmenting the source model with user-defined information. As explained in section 2.3.3, there are two ways to extend the source model in the current implementation:

1. By adding facts
2. By adding rules

Users can communicate with the source model editor using a button named *Save Rule* in the user interface.

A rule is made up of two parts: a *head* and a *body*. The head of a rule is the combination of the name of the rule and the interesting variables in it. The body of a rule can be thought of as the query that the rule abstracts.

A rule is saved by typing the query that it abstracts in the box named *Query*. Users must supply a name for the rule in a dialog box that pops up on clicking the *Save Rule* button. She might optionally specify the interesting variables of the rule in the *Variables of Interest* box. For example, entering the text `a(X),b(Y),c(Y,Z)` in the *Query* box, the text `Z,X` in the *Variables of interest* box and the text *somename* in the dialog box that pops up on clicking the *Save Rule* button, the following rule is saved in the source model:

```
somename(Z,X):- a(X),b(Y),c(Y,Z).
```

For saving a new fact to the source model, one can type the fact in the *Query* box and click on the *Save Rule* button. The dialog box that asks for a name pops up and should be ignored/canceled.

Browser

The *browser* component transforms the results of a selection-criterion query onto a tree view and provides searching and navigation capabilities for the generated view.

The browser component takes the output of the query engine, which is a set of bindings for the interesting variables in the query and renders them as a tree in accordance with the organization criterion.

A node in a tree serves a "link" to the location of the corresponding element in the source code. When a user double-clicks on a tree node, the browser component opens up an editor for her, with the cursor positioned exactly near the definition of the element in the code. In this way, the generated view closely resembles a conventional browser interface, in line with our design goals (see section 2.1).

The browser component also offers a lexical pattern matching search utility within the tree structure of the view. A lexical pattern entered in a box at the bottom of the generated view is searched in all the tree nodes and the results of the search are brought into focus one-by-one.

Source Code Editor

The *source code editor* of our prototype tool is a modified version of a third-party open-source editor called JE [19]. It has a comprehensive set of features for source code editing, including sophisticated features such as syntax coloring, code beautification, spell checking etc.

Originally, we built a very simple editor for QJBrowser with no bells or whistles. Later we replaced this editor with JE. To accomplish this change task, we used QJBrowser itself to locate the code units that needed to be modified in both JE and QJBrowser. This will be described in detail in Chapter 3.

2.5.3 Back End

The back end components of QJBrowser deal mainly with the formation and querying of the source model. This section discusses the individual components of the back end in detail.

Reification Engine

The *reification engine* is responsible for the creation of the tool's source model. It is a static-analysis tool that uses a modified version of AspectJ parser to parse and type-check the source code and create the corresponding *meta data* on a storage medium. The meta data are simply facts about the source code, compatible with the query language used.

The reification engine is triggered during program initialization as well as every time a user clicks on the "Reify File" menu item in the source code editor (see section 2.5.2).

Query Engine

The query engine is responsible for executing user queries against the source model. The query engine of our prototype uses SICStus Prolog [2] and a library called Jasper [17] for interfacing with native Prolog code.

Source Model

Source model is a database of information about the underlying system. By database, we mean just a collection of data; it can be represented as a set of flat files or a relational database or any other kind of representation. The current implementation uses a set of flat files.

In our prototype, the source model is a collection of Prolog relations. It is generated automatically during startup by the reification engine component. Users can augment it using the source model editor and the corresponding user interface component (see section 2.5).

In the current implementation of the source model there are essentially four kind of relations:

- Relations about entities in the code: These include facts that establish the entities in the source code. They are unary relations with complete names of the entities that they establish as their only argument. By complete name, we mean the name that identifies the entity within the context of the underlying system. Facts like `class('thisPackage.thisClass')` and `method('thisPackage.thisClass.thisMethod')` are examples of this kind of relations.

- Relations connecting entities with their properties: These are binary relations that connect entities with their properties. For example, a fact like `context('thisPackage.thisClass.thisMethod', 'thisPackage.thisClass')` establishes that the *context* of the entity `thisPackage.thisClass.thisMethod` is `thisPackage.thisClass`. Here the property established is *context.* Other examples of properties include modifiers of an entity, exceptions thrown by methods, static type of member variables etc.

In these relations, the first argument represents the entity and the second represents the entity's property. Thus, for a relation `modifier`, the fact `modifier('thisPackage.thisClass','public')` can be read as `'public'` is a modifier of `'thisPackage.thisClass'`.

- Relations that establish mutual connections: Relations like `callinfo` and `fieldaccessinfo` establish the relationship between two interacting entities in the code. Such relations typically have three arguments: the first two representing the respective entities and the third representing the exact location in the code where the relationship is expressed.

- Arbitrary user-defined relations: These include relations that users add to the source model. They may have any number of arguments and establish arbitrary information. Recall from section 2.3.3 that there are two ways of adding information to the source model: adding facts and adding rules.

For a listing of relations in the source model, please see Table A.1.

2.5.4 Discussion

In designing the tool, we spent considerable effort to make it flexible and extensible. The following are the highlights of our design and implementation:

- Our design acknowledges the general principle *"Program to an interface, not an implementation"*. Therefore, the different components of the tool actually implement *interfaces* that are needed by their client objects. The client objects do not directly hold references to the components directly. Instead, they refer to the corresponding interface classes. This reduces the coupling between objects. The components can be easily replaced without modifying their client code. The actual component instantiation is done in a *Factory* [13] class that is configured using a configuration file.

- The entire system is made up of a number of small component objects. A configuration file is responsible for naming the components to be used in the final system. The configuration file for the system can be thought of as a little *meta-program*, which defines the components that must be part of the tool. This type of assembly of the system allows mix-and-match between components and improves the overall changeability of the system.

- The user interface of the system and the functionality of the user interface are cleanly separated using Observer pattern [13]. Therefore, the interface and the implementation can be changed independent of each other.

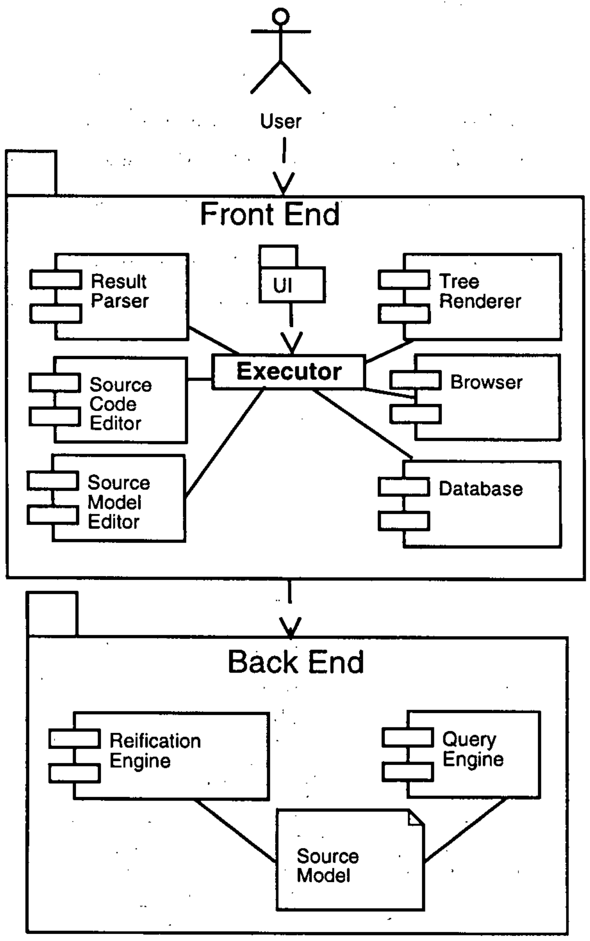


Figure 2.6: QJBrowser: System Architecture

- The architecture of the system is strictly layered. The classes at a lower level in the architecture provide services to the classes at a higher level. The overall architecture of the system is shown in Figure 2.6. Layering is a tried and tested architecture advocated by Dijkstra [35] and has been analyzed in literature before [8, 35, 36].
- As much as possible, each class implements only the functionality that it needs to. This reduces the problem of code tangling in the system. In addition, components that have distinct responsibilities can be replaced independent of each other. For example, instead of having a single class parse the query engine's output, compose it appropriately and render it on a view, our tool uses

a `ResultParser` class to parse the query results, a `ResultComposer` class to compose them according to user's organization criterion and a `TreeRenderer` class to render them as a tree. It should however be noted that this results in a trade-off in terms of the number of little classes in the system. To be more precise, since each class has fewer responsibilities, the system has more classes. There are more interactions between classes which might impact the overall performance of the system as well its comprehensibility.

2.6 Summary

In this chapter we discussed our prototype tool QJBrowser in detail. We outlined its concepts, demonstrated its usage and discussed its design and implementation.

QJBrowser is a tool that generates views based on the criteria supplied by a user, namely a *selection criterion* and an *organization criterion*. The selection criterion is a query written in an expressive query language using a variety of information, such as inheritance relationships, calling dependencies, static types etc. It selects the elements that should constitute the generated view. The organization criterion projects the results of the execution of this query onto a hierarchical browser-like view.

Overall, it was evident from this chapter that QJBrowser is a query-based browsing tool that offers an extensible query language to define crosscutting views on the system. We saw that the configuration mechanisms provided are simple enough to generate views on demand. The next chapter will provide evidence supporting our contention that such a tool can be used to obtain a wide variety of views that can help a user explore crosscutting and non-crosscutting concerns in software.

Chapter 3

Validation

This chapter is intended to provide evidence supporting our thesis that QJBrowser, a configurable browsing tool with an extensible query language, can offer conceptually simple mechanisms to create different kinds of views that can help a user work with crosscutting concerns.

This chapter is organized as follows: In section 3.1, we illustrate the utility of QJBrowser by outlining some example views generated by it. This section will show that QJBrowser achieves its goals of being simple and configurable enough to generate many kinds of different crosscutting as well as non-crosscutting views dynamically. This is followed by section 3.2, which is a discussion of our own preliminary experience with the tool. Section 3.2 will describe how we used the views generated by QJBrowser in accomplishing two of our simple development tasks.

3.1 Examples

This section focuses on example views generated by QJBrowser. These example views are intended to illustrate the following:

- QJBrowser can be configured to generate a **wide variety** of crosscutting and non-crosscutting views.
- QJBrowser can be configured using simple mechanisms. The configuration process is not very complex and can be done **on demand**.

The examples discussed in this section are of two types: General-purpose and Application-specific. By general-purpose views, we mean the kind of views that are applicable to any software system. The other kind of views, application-specific views, are obtained using application-specific data and may not be applicable to all software.

General-purpose views are the kind of views that can obtained with most development tools. Unlike most development tools, the general-purpose views that can be obtained using QJBrowser are not limited to a pre-defined set.

Application-specific views capture a user's knowledge about a domain or application. This kind of views is particularly interesting because it is not possible to obtain such views with a typical development tool that offers a pre-defined set of views. Besides, an application-specific or a task-specific view can be more "in tune" with a user's task than a general-purpose view offered by a development tool.

Our aim in using these two kinds of views in this section is to illustrate that QJBrowser is generic enough to encompass most of the standard views offered by typical development tools and configurable enough to produce views that are tailored to specific user tasks. We show in this section that it is possible to obtain many different kinds of views - both general-purpose and application-specific - and different perspectives of the same view using QJBrowser. Additionally, several of the views are crosscutting and reveal information which would be hard to discover with a traditional browser.

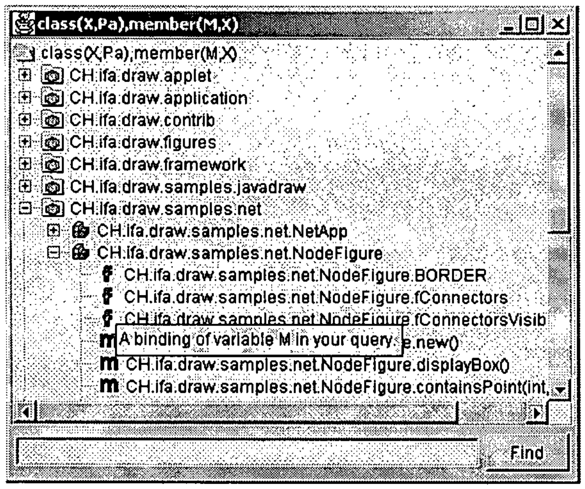
3.1.1 General-Purpose Views

This section starts with a very simple view that can be obtained in most of the modern development tools : a *class browser*. A class browser is a familiar view to most developers and is found in almost all IDEs. This will be followed by an *exception browser* example, where we discuss a class-browser-based organization of the exceptions propagated in a system. That is, we organize exceptions based on the classes in which they are thrown, in a class-browser-like view. We also discuss a number of possible useful variations for the same view. Our aim in doing so is to illustrate that QJBrowser can be used not only to generate a wide range of different views based on different data, but also a multitude of views based on the same data.

Conventional Class Browser

This example demonstrates that QJBrowser can be used to define a view that is very familiar to most developers: a *class browser*. A class browser can be found in most development tools like IDEs. These tools typically provide a fixed class view with packages, classes and members displayed in a hierarchy, in that order.

We can define a class browser in QJBrowser using the following view-definition parameters:



```
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,
```

Figure 3.1: Class Browser View

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

The view generated for this set of parameters is shown in Figure 3.1. It is possible to obtain a number of perspectives of the same view although some of them might not be very natural or useful. For example, it is possible to obtain a view that leaves out packages in the hierarchy by using the organization criterion, `Class,Member`. It is also possible to obtain various flavors of the class browser: One such flavor is a class browser that shows exceptions propagated in a system. This browser can be obtained by simply refining the query for an ordinary class browser, with extra information. The next subsection discusses this view in detail.

Exception Browser

The exception browser discussed in the section 2.3.1 is only one of many similar browsers that can be defined around the theme of exceptions. The example in this section shows how one can obtain with QJBrowser, 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. The selection criterion for this kind of browsers is shown below:

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

It must be noted that this query builds on the query stated in our last example. Thus, configuring QJBrowser to produce different types of browsers need not involve complex operations. It can be a simple editing operation incorporating the appropriate predicates in the selection criterion query.

There are many possible organization criteria for this selection criterion, each defining a somewhat different view. The total number of possible views that can be obtained by selecting different variables and reordering them is calculated by using the formula $\sum_{i=1}^{n} nP_i$, where nP_i represents a permutation of length i from a set of n elements.

Since in this example, there are 4 variables, the above formula computes to:

$$4P_1 + 4P_2 + 4P_3 + 4P_4 = \frac{4!}{(4-1)!} + \frac{4!}{(4-2)!} + \frac{4!}{(4-3)!} + \frac{4!}{(4-4)!} = 64$$

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. In a way, we can say that the `Package`, `Class` and `Method` variables do not represent orthogonal entities. We explain this further in the following paragraphs.

- Typically, executing a logic query produces a set of solutions. Each solution in the set can be thought of as a n-tuple composed of bindings for the n variables in the query. The query results can thus be represented as points in a n-dimensional space in which each of the variables corresponds to an axis or dimension.

  In principle, none of the axes is more important than another. 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 the X-axis are categorized according to units on the 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.

  In our example, the variables `Package,Class and Member` have a natural hierarchical categorization. A package is made up of classes and class is made up of members. When modeling them on a n-dimensional graph, they are not independent of one another and hence do not form orthogonal dimensions. They can rather be represented using a single dimension, with one or more classes mapping on to a package and one or more members mapping on to a class. Whereas, the variable `Exception` is orthogonal to `Package`, `Class` and `Method` variables. This means that `Exception` can be positioned independently of `Package`, `Class` and `Method` in the organization criterion.

Organization#	Useful organization criteria
1.	EPCM
2.	PECM
3.	PCEM
4.	PCME
5.	EPM
6.	ECM
7.	PEM
8.	PME
9.	CEM
10.	CME
11.	EM
12.	CE
13.	ME

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

Figure 3.2: Useful organization criteria for exception browser.

Some variations of the exception browser can be easily obtained by hiding parts of other variations. For example a view with organization criterion `Package, Class, Exception` can be obtained by expanding the view generated by the criterion `Package, Class, Exception, Method` only up to the third level. Similarly, any variation that leaves out the variable `Exception` might not represent a class browser that deals with exceptions propagated in a system.

Taking these and other similar considerations into account we have reduced the number of actually useful views to the 13 shown in Figure 3.2. We will only explicitly discuss the first 4 variations in detail, which use all the variables in the query. These four are presumably the most useful ones. Each one of the four resulting views differs from the others only in the way it shows how exceptions crosscut 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 a 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 in the system where a particular exception is thrown.

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, it should be noted that 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 with respect to crosscutting, 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.

This example shows that QJBrowser can display crosscutting from different perspectives. There may be several variations possible for the same view and each of the variations might be useful in depicting crosscutting at a different level. QJBrowser is configurable enough to capture a wide range of these (subtly different) variations, while a conventional browsing tool is not.

With QJBrowser, one can get general-purpose views, similar to the views offered by a conventional browsing tool. In addition to that, one can define a view that is specific to a code base using data specific to that code base. The next subsection discusses these kind of views in detail.

3.1.2 Code-base-specific browsers

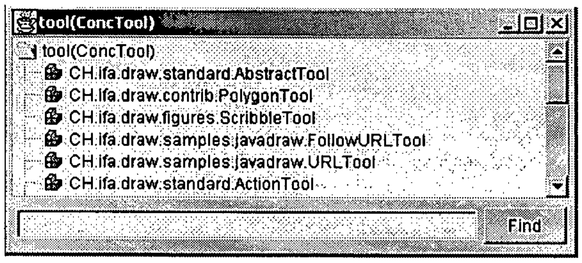
In this section we discuss examples that illustrate the utility of QJBrowser in allowing developers to define views specific to a particular code base. In other words, in these examples, a browser's view-definition is inspired by some specific knowledge, which is closely linked to a particular code base and may not hold true for any other code base.

For this purpose, we shall consider a Java GUI framework for graphics, called

JHotDraw [12, 20]. JHotDraw provides several elements such as *tools, menus and handles* 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.*) also play an important role in the JHotDraw code base. Specific bits of knowledge about how these concepts are implemented in JHotDraw will be the basis for the examples in this section.

Tools Browser



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 3.3: Tools Browser View

The browser in this example presents a view that 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 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`

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, namely a *rule* (see Appendix B) that defines it:

```
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 3.3). This rule is very useful as it improves the readability of the queries as well as the ease with which queries can be composed, in the following examples.

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 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).
```

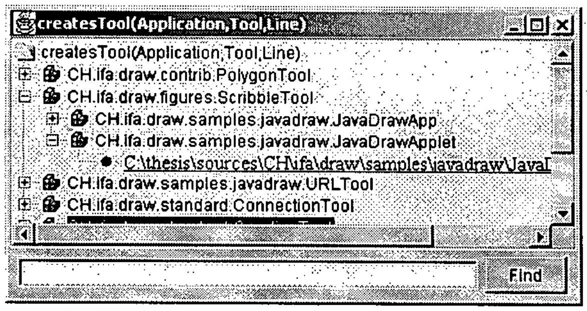
The instantiation is accomplished by simply invoking the constructor of the corresponding tools. We use this useful bit of knowledge, in conjunction with the rule defined above, to add another rule that defines the relationship between an application/applet and the tools it creates.

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

We can then use this rule, as shown below, to define *tool-creation* views (Figure 3.4 (a) and (b)).

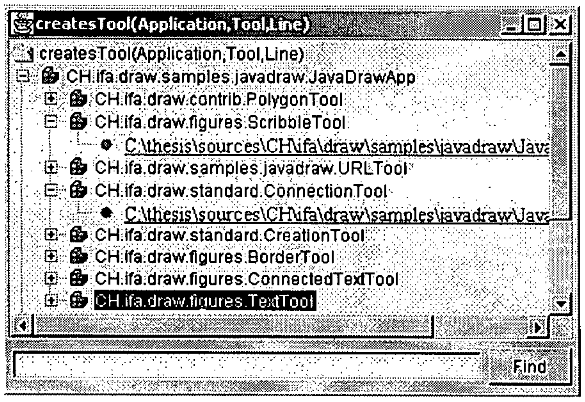
Selection:	`createsTool(Application,Tool,Line).`
Organization:	`Tool, Application, Line`

The browser shown in Figure 3.4(a) is the result of the above definition. It shows a hierarchy that lists the applications/applets in a code base that create particular tools. 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



```
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
```

(a)



```
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
```

(b)

Figure 3.4: Tool Creation Browsers

it easy find out all applications/applets that create a particular tool in addition to the precise location of the tool creation call in the corresponding application/applet. Note that it would be hard to gather the same information with a traditional browser because the creation methods where the relationships between a tool and the corresponding applications are expressed, are scattered across the different application classes.

Swapping the order of the first two variables in the organization criterion provides another useful view (Figure 3.4(b)) of the same data. This view is complementary to the previous one, making it easy to find out all the tools created by a particular application/applet. The definition parameters for this view are shown below:

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

Figure Browser

In this example, we define browsers around yet another JHotDraw-specific concept, namely *Figure*. Figures in JHotDraw are graphical objects that can be drawn and manipulated in applications/applets using appropriate tools.

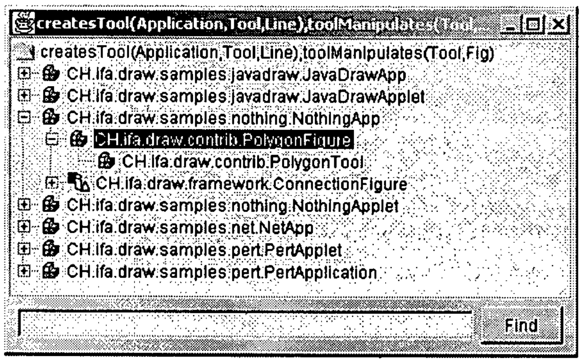
We want to produce different views that show the relationships between figures, tools and applications. To define an appropriate selection criterion, we need to make explicit some knowledge about the specifics of the JHotDraw's code base, by defining rules about them.

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).
```



```
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 View

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 a family of *Figure Browsers*. We only show one of the possible variations in Figure 3.5.

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

The examples in this section illustrated how interesting views can be defined using application-dependent knowledge. Such views can organize specific kinds of code units based on high-level relationships between code-base-specific concepts. Since the expression of these relationships are scattered throughout the code base they would be rather hard to discover by a developer using a conventional browser.

3.2 Experience

The last section illustrated that QJBrowser is configurable enough to produce a wide range of views on demand. The examples provided conceptual evidence that QJBrowser can indeed be used to work with crosscutting concerns by providing a way to generate a number of crosscutting views on the source code.

The examples only provide limited insight into the practical usability of the tool. This section is intended to illustrate the utility of QJBrowser in practice. We provide an account of our experience with the tool for performing two actual development tasks. We describe how the views offered by QJBrowser helped us in exploring our concerns during the tasks.

We would like to mention that "exploring" crosscutting concerns is a rather fuzzy goal. When we say that a tool like QJBrowser can help a user work with crosscutting concerns, we in fact mean that it can offer views that can help a user locate relevant software units in one or more steps.

In this section, we describe our preliminary experience with QJBrowser for performing two tasks:

- Comprehending the overall structure of a software package.
- Making a change to an existing application.

While performing the tasks, we recorded informal notes about the procedure as well as our general observations. The nature of the study itself was very preliminary. Therefore it does not provide unquestionable evidence regarding the usability of the tool. It however does provide support for our contention that one can use such a tool to explore a number of crosscutting concerns that arise during software development/maintenance. We could not perform more formal studies in the limited time that was available to us.

In the first task, our goal was to gain some general understanding of the structure and organization of JHotDraw [12], 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 a few well-known design patterns [13]. 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 various structural 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 perform the tasks required of an editor for QJBrowser, besides making changes to QJBrowser to accommodate JE.

An editor for QJBrowser would have to provide a way to update the source model of the tool with the changes made by editing. The most essential part of the change task was to find a way to provide this functionality in the editor by using a GUI component that was consistent with the ones already used by the editor. 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`:

Selection:	`shortname(Main,'main'),`
	`method(Main,EntryClass)`

Organization:	`EntryClass,Main`

The view defined by these parameters displayed all the classes that provided an entry into the application. 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. For this we just refined the previous query with extra information asking for the calling sequence of the `main` methods:

Selection:	`shortname(Main,'main'),`
	`method(Main,EntryClass),`
	`callgraph(Main,CalledMethods).`

Organization:	`EntryClass,Main,CalledMethods`

These parameters defined a view that consisted of all operations that could potentially be performed after the application is started. 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. Appendix C lists some of the major steps taken during each task. The reader is referred to it for more details. Here we provide an interpretation of the results of our experience.

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 [30] 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 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 pull-down 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.

3.3 Summary

This chapter provided support for our thesis that a tool like QJBrowser can help a user in working with crosscutting concerns by offering many different kinds of crosscutting views. We provided example views for sample scenarios and also discussed our practical experience with the tool.

In essence, QJBrowser offers a configurable query-based mechanism to define a multitude of views. A user can define a wide variety of views that are general-purpose or specific to an application. She can create a new view using a simple editing operation. We illustrated this by using examples. We also reported on two simple case studies that provide some preliminary insights into the practical usability of the tool.

Chapter 4

Related Work

In previous chapters, we described QJBrowser and how it can be used to deal with crosscutting concerns in source code by defining views using an expressive and extensible query language. In this chapter, we discuss some of the existing work similar to QJBrowser and how our research differs from them.

We discuss three kinds of work that have some relation to ours:

1. Commonly used integrated development tools that offer a pre-defined set of browsing capabilities.

2. Tools that can be configured to produce different views.

3. Tools that explicitly provide ways to deal with crosscutting concerns.

We choose to discuss exactly these kinds of tools for the following reasons:

- QJBrowser was motivated by the drawbacks in most modern development environments (see section 1.2 for details). Therefore, it is only logical to discuss some IDEs and show how QJBrowser can enhance their capabilities.

- QJBrowser is not the first tool to be configurable enough to generate a wide variety of views of source code. Several existing tools are configurable and they, like QJBrowser, establish a trade-off between configurability and simplicity. We discuss these tools since comparing the trade-offs made by each of them would help to establish the place of QJBrowser in relation to similar configurable tools.

- QJBrowser's ultimate purpose is to support the exploration of crosscutting concerns in source code. Some tools such as Aspect-oriented programming languages are also intended to address the same goal. We discuss such tools in order to compare the support each methodology (linguistic, browser-support etc.) offers to deal with crosscutting in code.

4.1 Integrated Development Environments

In this section, we discuss the most widely used kind of tool support for working with program concerns: Integrated Development Environments (IDEs). Typically, IDEs provide a suite of tools that target different aspects of program development. For example, an IDE might provide tools like *editor, class browser, class hierarchy browser, resource browser, file browser, debugger* and *compiler.*

Commercial IDEs such as Visual Studio [23], VisualAge [3], Forte [11] and JBuilder [18] provide only limited support for *exploring* crosscutting concerns. They provide a few standard set of browsing tools, such as class browser, package browser, class hierarchy browser etc. 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 views that may embody concepts that crosscut the basic modularity of the supported programming language.

It must be mentioned that the focus of an IDE is not simply browsing source code. Generally, IDEs are targeted towards providing a comprehensive set of tools for program development. Whereas, QJBrowser is a browsing tool that does not by itself offer any other functionality such as compiling, debugging etc. that are common to a typical IDE. Therefore, QJBrowser cannot be seen as a counterpart or replacement for an IDE, but it is complementary to it. A tool like QJBrowser can possibly be plugged in to an IDE to replace the different browser tools offered by the IDE.

Eclipse [9, 10] 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, thereby, greatly customizing the environment. Nonetheless, this requires significant effort, if one wants to produce a new view that is not offered by the existing set of tools in the Eclipse tool set, one has to develop a full-fledged plug-in tool in accordance with the Eclipse specifications. In comparison, defining a view with QJBrowser merely requires typing in the parameters that define the view.

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. Currently, the people at the Software Practices Laboratory in the University of British Columbia are investigating this idea.

4.2 Query-Based Tools

In IDEs, the views offered are typically pre-programmed into a fixed set of tools. The reasons for why this is not so desirable were discussed in section 1.2. We stated that a tool that can instead be configured by a user with the logic for generating different kinds of views is desirable.

There are some existing tools that can be configured to produce views of source code. One of the commonly used techniques to configure a view is by writing a *query*, which selects the elements that have to be shown in the view. Queries are essentially a way to identify elements related by a pattern/theme. For example, the commonly used search utility *grep* takes a lexical pattern/regular expression as a query that identifies the elements to be displayed. These elements may be present in different modules and the query output lets a user view them together. This in itself provides a way to deal with crosscutting code, since elements belonging to crosscutting code elements can be viewed as as unit.

In this section, we survey some tools that use queries as the basis for dealing with crosscutting in code.

4.2.1 Aspect Browser

Aspect Browser (AB) is a tool intended to assist evolutionary changes by making code relating to a global change feel like a unified entity [14]. An *aspect* in AB is defined as a pair consisting of a textual pattern and a color. When an aspect is enabled, the display of any program text matched by the pattern is highlighted with the aspect's corresponding color.

Aspect browser was built to help programmers in dealing with crosscutting changes to software. QJBrowser also has this as one of its goals. However, there are some important differences between the two tools. Firstly, Aspect Browser uses a much weaker query language based on lexical pattern matching (like grep). Although regular expressions are easier for typical programmers to frame than logic language queries, they are not Turing equivalent like a logic programming language. Secondly AB visualizes query results using a map metaphor. That is, query results, are represented spatially in a map using 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.

4.2.2 Aspect Mining Tool

> ...Each compilation unit (i.e. class) itself is represented as a collection of horizontal strips that correspond to the relevant lines of source code. The tool allows user-defined queries based on type usage and regular expressions, displaying matching lines in specific colors. If a line matches more than one criterion, it will be separated into two or more differently colored parts... [15]

Aspect Mining Tool (AMT) [15] 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. In QJBrowser, users define *browser-like views* using queries. Whereas, in AMT user-defined queries identify and color lines of source code that match a certain criteria.

AMT is very good for identifying *aspects* in a global view of the code base. Since it uses colors to distinguish patterns in the code base, it is easy to spot crosscutting aspects in a system. This may not be apparent using QJBrowser.

However, QJBrowser is a *browser-like navigable* tool that can be configured to view different crosscutting units on demand. These units could exhibit a wider variety of patterns among them than simple textual or type-based patterns; in other words, QJBrowser is a tool that can allow the exploration of source code using views that can be characterized by a wide range of syntactic as well as semantic queries.

4.2.3 Concern graphs

> ...we introduce the Concern Graph representation that abstracts the implementation details of a concern and makes explicit the relationships between different parts of the concern. The abstraction used in a Concern Graph has been designed to allow an obvious and inexpensive mapping back to the corresponding source code. To investigate the practical tradeoffs related to this approach, we have built the Feature Exploration and Analysis tool (FEAT) that allows a developer to manipulate a concern representation extracted from a Java system, and to analyze the relationships of that concern to the code base... [28]

Concern graphs [28] are used for abstracting the implementation details of a concern and showing relationships among the different parts of a concern explicitly. In the proof-of-concept tool FEAT, developers add individual code fragments to their concern one by one. The query language used in FEAT assists a developer in incrementally locating new fragments to add to the representation of a concern.

FEAT and QJBrowser 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. For the purpose of identifying concerns, FEAT supports six pre-defined set of queries. This is in sharp contrast with QJBrowser, which provides an extensible query language for expressing relationships among code elements that are connected to a user concern.

4.2.4 Smalltalk Object Unification Language

> ...The common denominator in the sketched problems is the incapability to express high level structural information in a computable medium that is then used to extract implementation elements. To solve this problem we introduce a logic programming language as meta-language to express and reason about the structural information of software systems... [37]

Smalltalk Object Unification Language (SOUL) [38, 37] is a logic meta-language that was developed for expressing and querying structural information about programs. It is a logic programming language that can reason about a base language program written in Smalltalk. It is based on Prolog, but provides specific extensions that can be used to query object-oriented systems.

SOUL was built as a meta-language to *express high-level structural information in a computable medium that can be used to extract implementation elements.* The language itself is extensible by using *facts* and *rules.* It was designed to provide evidence to the contention that a logic meta-language can be used to reason about and extract system structure in a base language independent way.

Whereas SOUL is a logic meta-language to query software, QJBrowser is a browsing tool that *uses* a logic meta-language to allow browsing of source code in different ways.

4.2.5 ASTLOG

> ...We desired a facility for locating/analyzing syntactic artifacts in abstract syntax trees of C/C++ programs, similar to the facility grep or awk provides for locating artifacts at the lexical level. Prolog, with its implicit pattern-matching and backtracking capabilities, is a natural choice for such an application. We have developed a Prolog variant that avoids the overhead of translating the source syntactic structures into the form of a Prolog database; this is crucial to obtaining acceptable performance on large programs... [7]

ASTLOG [7] is a system similar to SOUL for querying C and C++ abstract syntax trees. It can be used to perform syntax level analysis of source code. Similar to SOUL, ASTLOG was intended to be a language for querying rather than a customized navigation tool like QJBrowser.

It must however be noted that both SOUL and ASTLOG use more sophisticated implementation strategies than QJBrowser. They do not require an explicit translation phase to transform source code into a database that can be queried. Rather, they allow users to directly query the Smalltalk image and C/C++ AST structures respectively. This provides advantages in terms of performance and eliminates the need to synchronize two separate representations of the source code. These techniques are potentially also useful for a tool like QJBrowser and may be adopted in future versions of the tool.

4.2.6 Coven and Gwydion

In Coven, source files are treated as collections of separate program fragments. Programmers can dynamically organize these fragments into new organizations, corresponding to new decompositions of a system, into source files. The approach lets users write queries to dynamically generate new organizations by assembling individual fragments into *virtual source files* (VSFs).

Gwydion's approach is similar to Coven's in considering each source file as a collection of program fragments called *sheets*. Sheets provide linear textual display of the code, and users may directly edit the displayed text in order to modify the program. A query in Gwydion is simply a search that returns some subset of the currently defined program fragments.

In both Coven [4] and the Gwydion [32], queries are a mechanism for selecting flat sets of code-units, unlike the hierarchically organized results in QJBrowser. The tools themselves are targeted towards source code repository management. Whereas, QJBrowser is browsing tool and does not perform any source code management.

4.2.7 GraphLog

GraphLog [6] is a logic query language in which queries and query results are represented as directed graphs. Queries are represented as graph patterns. Edges in queries represent edges or paths in the program database. This visual formalism might ease the formulation of queries. The key difference between GraphLog and QJBrowser is that the former is a query language and not a tool by itself. It could potentially be used in QJBrowser, instead of Prolog or TyRuBa, to facilitate the composition of queries.

4.2.8 Semantic Visualization Tool

Semantic Visualization Tool (SVT)[1] is a framework providing primitives for visualizing and browsing any kind of data present in a code base. 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 different set of goals than QJBrowser, it offers much greater flexibility in the definition of a tool. Nevertheless, 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.

4.3 Alternative Modularization Approaches

In the last section, we discussed tools that used *queries* to produce crosscutting views of code. With these tools, it is possible to view and explore concerns that crosscut each other. However, it is not possible to separate the crosscutting code and make them into individual compilation units.

There are other approaches that support crosscutting concerns by allowing programmers to code crosscutting units of a system and weave them together appropriately. Two such popular approaches are the *Hyperspace* approach [24] and *Aspect-oriented* programming [22] approach. We discuss these two approaches in this section.

4.3.1 Hyperspaces

> ...We use the term multi-dimensional separation of concerns (MDSOC) to refer to flexible and incremental separation, modularization, and integration of software artifacts based on any number of concerns. It overcomes limitations of existing mechanisms by permitting clean separation of multiple, potentially overlapping and interacting concerns simultaneously, with support for on-demand re-modularization to encapsulate new concerns at any time. . . Hyperspaces are our approach to achiev-

ing MDSOC. Hyperspaces also provide a powerful composition mechanism that facilitates non-invasive integration, adaptation, and "plug-and-play"... We have defined a tool, called Hyper/J ^{TM}, which provides support for hyperspaces in Java^{TM}... [16]

HyperJ [16] is a tool that provides support for multiple decompositions and compositions of a system using a static file called *Hypermodule.* A hypermodule is composed of a number of *Hyperslices.* A hyperslice is created using a combination of a predefined set of pattern-matching expressions that name the entities that constitute the hyperslice.

Whereas, HyperJ supports on-demand re-modularization and executable composition of a system, QJBrowser supports the browsing of crosscutting views that are created dynamically. To this end, QJBrowser provides an expressive query language and a familiar user interface. Defining a crosscutting view requires as little effort as typing its definition parameters.

In contrast, in HyperJ dynamic decomposition of a system requires writing concern mappings that map concerns to their individual code units and forming hyperslices using these mappings. The final system is obtained by creating *hypermodules* by appropriately combining the hyperslices. This might be fairly more complex for a developer than a simple editing that defines a crosscutting view of a system.

Nevertheless, HyperJ supports encapsulation and composition of crosscutting modules into executable units. QJBrowser only supports viewing of concerns in code, but not encapsulation and composition of concerns, since its goal is to be a "browsing tool" that can assist developers in exploring crosscutting concerns effectively.

4.3.2 AspectJ

...AspectJ^{TM} is a simple and practical extension to the Java ^{TM} programming language that adds aspect-oriented programming (AOP) capabilities. AOP allows developers to reap the benefits of modularity for concerns that cut across the natural units of modularity. In object-oriented programs like Java, the natural unit of modularity is the class. In AspectJ, aspects modularize concerns that affect more than one class... [22]

AspectJ is one of the early linguistic approaches to crosscutting concerns. It provides language support for coding concerns that affect more than one module.

Whereas QJBrowser provides dynamic support for working with crosscutting concerns, with AspectJ, one must code aspects, weave them with the source code and compile the resulting code to work with crosscutting. Likewise, since AspectJ is a new programming language based on a new paradigm, its adoption might take longer than a browsing tool like QJBrowser.

However, just as we stated in the last section, QJBrowser is only a browsing tool. It does not aim to produce executable code for crosscutting units. AspectJ and HyperJ provide support for modularizing crosscutting units and executing them. The advantage of QJBrowser is predominantly its ability to allow developers to work more "on demand" with crosscutting concerns than possible with linguistic approaches like HyperJ and AspectJ.

Chapter 5

Conclusion

This dissertation presented the design and implementation of QJBrowser, a query based tool that can assist in the exploration of crosscutting concerns in source code. We showed how QJBrowser can be configured dynamically to create multiple crosscutting views of source code. We illustrated using examples and our own experience with the tool that our implementation indeed confirms to the goals we stated for a tool that can support our thesis, in the beginning of the dissertation:

- It must be a query-based browsing tool. Users should be allowed to browse their source code by defining their views using queries. In addition, the views obtained must have a conventional *browser-like* interface that is familiar to most developers.
- The query language that it offers must be extensible so that it does not limit the kinds of views that can be obtained from the tool.
- It must be configurable enough to generate a wide variety of interesting views.
- It must be simple to use. The tool must be used easily to generate views on demand. We limited the scope of our tool to browsing rather than graphical visualization simply because configuring the appearance and contents of a graphical view can quickly get overwhelming to be used on the fly.
- It must help a user work with crosscutting concerns in her software.

5.1 Limitations and Future Work

QJBrowser is only a proof-of-concept prototype that we developed to advance our thesis. In its current state of implementation, it has several limitations:

1. The queries used are in Prolog. One factor to consider here is that the tool's intended audience are Object-oriented developers who might not be very familiar with logic languages. One possible approach to address this issue 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 [6]. A third possibility is to look at non-logic query languages such as SQL, which are typically less expressive, but might be more familiar to developers.

2. Sometimes the performance of the tool is poor. This is especially true when the query result set or the code base is large. Enhancing the tool's performance can be done by performing optimizations like caching of query results, incorporating better search strategies for query execution etc. It must be mentioned that TyRuBa, the query language that we used for the second version of QJBrowser, uses some optimizations like this. In the TyRuBa version of QJBrowser, it is a lot faster to execute a query that has been encountered by the query engine before, than running a query for the very first time.

3. Formulating queries that have the intended effect is difficult. In our experience, we had to continuously refine queries to get the result we needed.

4. Some tree views might not be very intuitive. For example, when an inheritance tree is displayed one would expect the results to be recursively nested. That is, if A is a parent of B which is a parent of C, then one would like C to be nested inside B and B to be nested inside A. But in the current implementation, both B and C will be nested inside A, since both are the subclasses of A. The same is true for other recursive queries like callgraph.

   This limitation is a direct result of our decision to keep the configuration of a view as simple as possible. If the configuration mechanism is made more elaborate, this limitation could be overcome. For instance, if it were possible to express in the view-definition parameters that `subclass` is a recursive query and the display must reflect that fact, the class-hierarchy view obtained with QJBrowser could be made much more intuitive. However, one of QJBrowser's goals is to strike a balance between configurability and simplicity. Making the configuration process more elaborate might it make it more difficult to use the tool.

5. Another idea for future research concerns extending the range of information that is automatically extracted from the code base by the tool. The current source model includes only facts about the source code that are automatically

extracted by simple static analysis of source code. However, it can be extended easily to incorporate information from a variety of other sources, such as information from dynamic analysis, JavaDoc comments, version management tools etc.

6. Apart from these limitations of the implementation itself, the studies we performed with the tool are quite preliminary. They do not provide conclusive evidence with respect to the usability of the tool. We could not perform more formal user studies in the limited time we had.

Bibliography

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

[2] J. Andersson, S. Andersson, K. Boortz, M. Carlsson, H. Nilsson, J. Widn, and T. Sjland. The SICStus emulator. Technical Report T91:15, Swedish Institute of Computer Science., 1991.

[3] L. A. Chamberl, S. F. Lymer, and A. G. Ryman. IBM VisualAge for Java. *IBM Systems Journal*, 37(3), 1998.

[4] M. C. Chu-Caroll and S. Sprenkle. Coven: Brewing better collaboration through software configuration management. In *Foundations of Software Engineering (FSE)*, November 2000.

[5] H. Coelho and J.C. Cotta. *Prolog by Example.* The MIT Press and Springer-Verlag, 1988.

[6] Mariano P. Consens and Alberto O. Mendelzon. GraphLog: A visual formalism for real life recursion. In *proceedings of the Nineth ACM SIGACT-SIGMOD Symposium on principles of Database systems*, pages 404–416, 1990.

[7] R. F. Crew. ASTLOG: A language for examining abstract syntax trees. In *Conference on Domain-Specific Languages*, 1997.

[8] P. J. Denning and R. L. Brown. Operating systems. *Scientific American*, 251(3):80–90, 1984.

[9] The Eclipse platform: Technical overview. Technical report, Object Technology International Inc., July 2001.

[10] The Eclipse IDE. Available online at http://www.eclipse.org.

[11] Forte for Java. Available online at http://wwws.sun.com/software/.

[12] E. Gamma. The JHotDraw framework.
Available online at http://www.jhotdraw.org.

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

[14] W. G. Griswold, J. J. Yuan, and Y. Kato. Exploiting the map metaphor in a tool for software evolution. In *Proceedings of the 2001 International Conference on Software Engineering*, May 2001.

[15] J. Hanneman and G. Kiczales. Overcoming the prevalent decomposition of legacy code. In *Workshop on Advanced Separation of Concerns at the International Conference on Software Engineering (ICSE)*, May 2001.

[16] Hyper/j ^{TM}: Multi-dimensional separation of concerns for Java. Available online at http://www.research.ibm.com.

[17] Intelligent Systems Laboratory, Swedish Institute of Computer Science. *Mixing Java and Prolog - Jasper. SICStus Prolog User's Manual.*, 3.7.1 edition, October 1998.

[18] JBuilder. Available online at http://www.borland.com/jbuilder/.

[19] JE—just an editor.
Available online at http://mathsrv.ku-eichstaett.de/MGF/homes/grothmann/je/.

[20] R. E. Johnson. Documenting frameworks using patterns. In *Proceedings of the OOPSLA '92 Conference on Object-oriented Programming Systems, Languages and Applications*, pages 63–76, October 1992.

[21] Stanley M. Sutton Jr. and I. Rouvellou. Concern space modeling in Cosmos. In *2001 Conference on Object-Oriented Programming, Languages and Systems*, October 2001.

[22] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Videira Lopes, J. Loingtier, and J. Irwin. Aspect oriented programming. In *ECOOP '97 — Object-Oriented Programming 11th European Conference*, June 1997.

[23] M. Kirtland. Introducing Visual Studio 97: A well stocked toolbox for building distributed apps. *Microsoft Systems Journal*, 12(5), May 1997.

[24] H. Ossher and P. Tarr. Multi-dimensional separation of concerns using hyperspaces. Technical Report 21452, IBM, April 1999.

[25] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, December 1972.

[26] D. L. Parnas. Designing software for ease of extension and contraction. In *Proceedings of the 3rd international conference on Software engineering*, 1978.

[27] M. Robillard and G. C. Murphy. Analyzing exception flow in Java programs. In *Lecture Notes in Computer Science*, volume 1687, pages 322–337. Springer-Verlag, September 1999.

[28] M. P. Robillard and G. C. Murphy. Concern Graphs: Finding and describing concerns using structural program dependencies. In *24th International Conference on Software Engineering (ICSE)*, May 2002.

[29] P. Saint-Dizier. *An Introduction to Programming in Prolog.* Springer-Verlag, 1990.

[30] John Sands. News: JFC: An in-depth look at Sun's successor to AWT. *JavaWorld: IDG's magazine for the Java community*, 3(1), January 1998.

[31] L. Sterling and E. Shapiro. *The Art of Prolog.* The MIT Press, 1994.

[32] R. Stockton and N. Kramer. The Sheets hypercode editor. Technical Report 0820, Dept. Computer Science, Carnegie Mellon U., 1997.

[33] Sun Microsystems. Java RMI. http://java.sun.com/products/jdk/rmi/.

[34] K. De. Volder. *Type-Oriented Logic Meta Programming.* PhD thesis, Vrije Universiteit Brussel, 1998.

[35] Dijkstra E. W. The multiprogramming system for the EL X8 THE. circulated privately, June 1965.

[36] W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack. HYDRA: The kernel of a multiprocessor operating system. *Communications of the ACM*, 17:337–345, June 1974.

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

[38] R. Wuyts. *A Logic Meta-Programming Approach to Support the Co-Evolution of Object-Oriented Design and Implementation.* PhD thesis, Vrije Universiteit Brussel, 2001.

Appendix A

Source Model

In this appendix, we list some important relations available in the current version of the tool's source model.

`class(Cls)`	Find all classes Cls declared in the system.
`field(Fld)`	Find all fields Flds declared in the system.
`method(Mtd)`	Find all methods Mtd declared in the system.
`context(Ent,Cont)`	Find all pairs Ent,Cont where Ent is an entity (class, method or field) in a system and Cont is its context.
`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 etc) attached to that declaration.
`shortname(Dec,Name)`	Find all pairs of Dec,Name where Dec is a declaration (for a class, interface, method, constructor or variable) in the system and Name is the short name for the declared entity: the unqualified name for a class, interface, field or selector name for a method.
`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.

`arg(Met,Arg)`	Find all pairs Met,Arg where Met is a method declaration in the system and Arg is an argument of that method.
`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.
`fieldaccessinfo(Field, Method,Line)`	Find all pairs Field,Method where Field is a Field declared in the system and Method is a method that accesses the field Field in the code. Additionally, Line will be bound to a reference to the actual source-code line where the field is accessed.
`fieldchangeinfo(Field, Method,Line)`	Find all pairs Field,Method where Field is a Field declared in the system and Method is a method that changes (uses the field in the LHS of an expression) the field Field in the code. Additionally, Line will be bound to a reference to the actual source-code line where the field is changed.
`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.
`method(Met,Cls)`	Find all pairs Met,Cls where Cls is a class (or interface) declared in the system and Met is a method declared in that class.
`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) declared in that class.

`callgraph(Startmethod,Graph)`	Find all pairs Startmethod,Graph where Startmethod is a method declared in the system and Graph is the transitive closure of the methods that gets called starting from Startmethod.
`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.
`constructor(Met, Cls)`	Find all pairs Met,Cls where Cls is a class declaration in the system and Met is a constructor method declaration.

Table A.1: Source Model of QJBrowser

Appendix B

Prolog

Prolog is a logic programming language that provides for the representation of a subset of first-order predicate calculus. It is very expressive and has a simple and intuitive syntax. Prolog queries are composed of one or more goals. In its simplest form, a query is just a single goal that has to be satisfied by the Prolog engine. The result of a goal can be positive or negative based on information in the internal database of the Prolog engine.

```
class(thisClass).
method(thisMethod).
method(thatMethod).
context(thisMethod,thisClass).
context(thatMethod,thisClass).
throws(thisMethod,thisXCep).
```

Figure B.1: A simple source model

For instance, let us consider the source model shown in Figure B.1.

Every line in the figure represents a *fact* about the source code. For this source model, the goal `method(thisMethod)` will yield a positive result since the source model has a fact that names the term `thisMethod` to be a method. Likewise, the goal `method(someMethod)` will yield a negative result.

Logical variables are identifiers starting with uppercase alphabets. The goal `a(X)`, for example, has a logical variable named `X`. The solution to this goal is a set of values for `X` for which the relation `a` is true. If the relation `a` is `method` and the source model is the one shown in Figure B.1, then the query engine will assign the values `thisMethod` and `thatMethod` to `X`. The process of assigning a value to a variable is called the instantiation or binding of a value to that variable.

Goals can be combined to form *derived queries* using logical operators: ',' and ';', representing logical conjunction and logical disjunction respectively. In such a case, the query execution will attempt to satisfy the goals one-by-one, starting from left to right.

To be more precise, consider a simple conjunction:

$$\texttt{context(X, thisClass), throws(X, thisXCep).} \tag{B.1}$$

Let us see the sequence in which Prolog executes this query for the source model shown is Figure B.1. Prolog tries to satisfy the first goal `context(X,thisClass)` first. This results in two value bindings/substitutions for the variable X - `thisMethod` and `thatMethod` - and hence a fork in the execution as shown in Figure B.2.

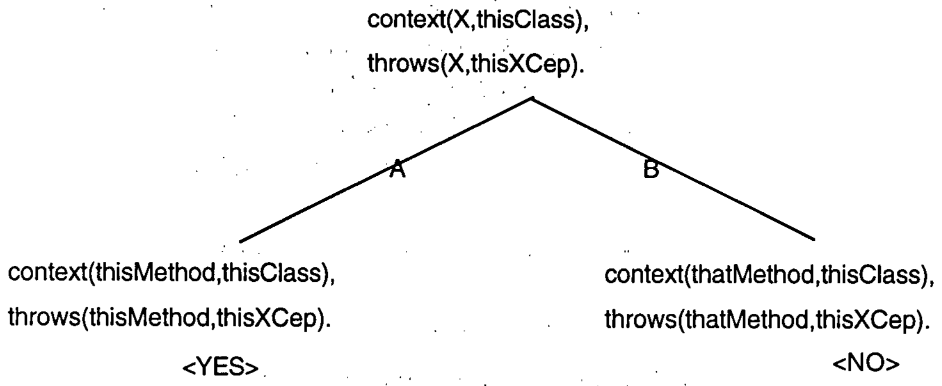

context(X,thisClass),
throws(X,thisXCep).
A
B
context(thisMethod,thisClass),
throws(thisMethod,thisXCep).
<YES>
context(thatMethod,thisClass),
throws(thatMethod,thisXCep).
<NO>

Figure B.2: Execution path for the query (B.1).

Following fork A, we can see that the variable X has been substituted by `thisMethod`. With this substitution, Prolog tries to satisfy the next goal, namely `throws(thisMethod,thisXCep)`. This goal succeeds as the source model has a fact that matches it. Since there is no more goal to match, Prolog engine concludes that `thisMethod` is a valid binding for the variable X and returns it as an answer to the query.

Along fork B of Figure B.2, the variable X has been substituted by `thatMethod`.

Prolog tries to satisfy the next goal with this substitution. It fails because there is no fact in the source model that matches it exactly. Therefore this fork is an unsuccessful branch. Whenever Prolog encounters an unsuccessful branch, it *backtracks* to the previous node and follows an alternative path. Since in this case there are no more paths left at the previous node, execution stops with just one binding for the variable X, namely `thisMethod`.

Let us now consider a query that uses a disjunction to join two goals as shown below:

$$\texttt{context(X,thisClass);throws(X,thisXCep).} \tag{B.2}$$

A disjunction offers an alternative execution path for the query engine. The execution path for the query B.2 is shown in Figure B.3.

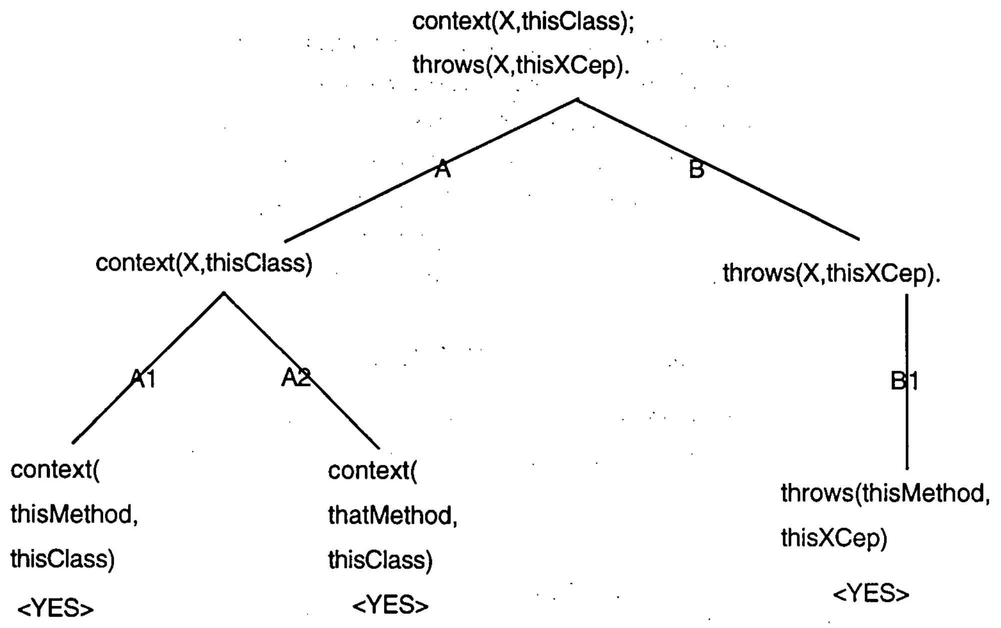


Figure B.3: Execution path for the query (B.2).

For more information on Prolog, please refer to [31], [29] and [5]

Appendix C

Experience - Results

In order to gain some experience with the tool and test its applicability to explore software concerns, the author of the tool performed two simple development tasks with QJBrowser, involving non-trivial software packages downloaded from the Internet.

During both tasks, the author took informal notes about the steps she performed and the queries used. In the first experiment, the author's goal was to understand the overall structure and organization of a software package called JHotDraw of which she had no prior knowledge. In the second experiment, the goal was more specific, involving a specific change task - an open-source editor called JE must be modified to incorporate new functionality and be plugged in as an editor for QJBrowser.

This appendix provides an abridged account of the informal notes taken by the author during both tasks. We provide only an abridged version of the notes taken by the developer, mainly because of space constraints. Additionally, the account provided is simply a paraphrasing of the notes taken by the author. For a more focused discussion of the notes, please refer to section 3.2.

C.1 Comprehension Task

The first step that the author took was to compile and run JHotDraw to get a first impression of what the application does – It created a window with a menu, toolbar and other GUI components. It was possible for the user to draw different shapes using these components.– After the author had some hands-on experience with the tool, she was ready to start exploring its source code.

The paragraphs below provide an account of the steps taken by the author to explore the structure of the package's source code using QJBrowser. They list the definition parameters for some of the major views generated during that task along with a brief description for them.

i.	**Selection:**	`shortname(M,'main'),method(M,C), callgraph(M,G).`
	Organization:	`C,M,G`

This view displayed all the methods named `main`(application-entry methods) in the application and their corresponding callgraphs. The developer noted that four classes in JHotDraw had application entry points and all the application entry points were followed by a very similar call sequence.

Using the navigation feature of the generated view, the author could easily navigate to the source code of the methods in the call sequence. She noted that in the call sequence, an instance of a class called `DrawApplication` was created, which in turn created a window, menus and a tool bar. Therefore, the developer decided to explore the `DrawApplication` class, the menus and the toolbar further.

ii.	**Selection:**	`subclass(C,P),(P =='javax.swing.JMenu'; P == 'java.awt.Menu').`
	Organization:	`P,C`

This view showed all the classes that could act as menus. An interesting detail that was found in this step was that all Menu components in the software were *swing* [30] components. The developer also found that there was a class called `CommandMenu` that was a subtype of `JMenu`. Since her immediate subgoal was to understand the implementation of menu functionality in the package, she decided to explore `CommandMenu` further.

iii.	**Selection:**	`shortname(C, 'CommandMenu'), callinfo(A1,A2,_), context(A2,C).`
	Organization:	`A2,A1`

This view displayed all units of the software that called any of the methods in the `CommandMenu` class. The author wanted to explore the `CommandMenu` class, since she found it to be the only class that derived from `JMenu` and hence a core part of the menu implementation in JHotDraw.

From the result of this query, the developer observed that the methods that created different menus in the `DrawApplication` class, for example `createEditMenu`, `createMenuItem`, `createDebugMenu`, `createColorMenu` etc., invoked a method called `add` in `CommandMenu`. This method took a `Command` object as its parameter.

Suspecting that there might be a *command* [13] design pattern involved in this interaction, she wrote a number of rules that abstracted the structural relationships of the participants of the *command* design pattern. She found that the `Command` class was in fact a participant in the Command pattern.

However, it should be noted that most design pattern instances cannot be *distinctly* and *unambiguously* defined using the structural relationships among their components. For example, aggregation/association can be implemented in a variety of ways - A class can have an aggregate in the form of a *Vector* of *generic objects*. It might not be possible to know the actual type of the objects in the vector until runtime. Such cases might result in *false negatives* while trying to detect patterns using just static information.

Similarly the structure of a pattern itself is very broadly defined. Therefore, some classes that are not intended to implement a pattern, but possess a resemblance to its structure might be detected wrongly.

iv.

Selection:	`subclass(C,P),P == 'javax.swing.JToolBar'.`
Organization:	`P,C`

Having understood the implementation of menu functionality in the package to a reasonable degree, the author decided to shift her focus to other GUI components like *tool bars*. To this end, she constructed a view with the parameters listed above. For constructing this view, the author used her knowledge that in java's swing package, the class `JToolBar` is used for representing a *toolbar*. The resulting view showed that there was a class called `CustomToolBar` that inherited from JToolBar, that could potentially be used for implementing the application's toolbar functionality.

v.

Selection:	`shortname(C, 'CustomToolBar'), callinfo(A1,A2,_), context(A2,C).`
Organization:	`A2,A1`

This view was used to find where any of the methods in `CustomToolBar` was called. To the surprise of the author, it was not called anywhere in the application, at least directly[1]

The developer checked her initial callgraphs and found a method called `createToolPalette` that actually created a `JToolBar` instead of a `CustomToolBar`. This toolbar was configured by a method called `createTools`. However, `createTools` did not create any particular *Tool* button (a toolbar is typically made of a number of buttons that can be clicked). Hence, the developer decided to find out if that was the only `createTools` method in the suite or if some subclasses overrode it.

vi.

Selection:	`shortname(D,'DrawApplication'), subclass(C,D), shortname(M,'createTools'), method(M,C).`
Organization:	`C,M`

All the subclasses of `DrawApplication`, which were also the classes containing the application entry points, contained their own versions of `createTools` method. Each of them created instances of a class called `Tool` and added the resulting tools to the toolbar. The developer wanted to explore the `Tool` class further by investigating its class hierarchy.

vii.

Selection:	`shortname(P, 'Tool'), subtype(C,P), code(C,U).`
Organization:	`P,C`

This view showed a bunch of classes that represented the different tools in the suite, like `TextTool, CreationTool, PolygonTool` etc. The developer skimmed the source code of these classes and found that they encapsulated the logic for manipulating the entities that they created.

These steps are only a subset of what the author followed in understanding the software suite. Since the notes were quite informal and some steps were minor or a refinement of the major ones listed here, we do not discuss every step in detail here.

[1]It should be noted that the above query would not reveal any call sequence generated by reflection.

C.2 Change Task

The second experiment involved making a change to the QJBrowser package itself by replacing its very simple editor with a more sophisticated one downloaded from the Internet, called JE. In essence, this task consisted of changing JE appropriately to make it perform the tasks required of an editor for QJBrowser. An editor for QJBrowser would have to provide a way to update the source model of the tool with changes introduced by the editing operation.

Hence, the most essential part of the change task was to find a way to provide this functionality in the editor using a GUI component that was consistent with the ones already used by the editor. In addition, some parts of QJBrowser had to be changed to unplug the old editor and plug in JE. In this section, we only discuss the steps taken by the author to figure out the places where JE had to be changed to accommodate QJBrowser-specific functionality.

To modify parts of a software, a user would generally require a good degree of understanding of its working. Therefore, the first step our developer took was to compile and run the application in its "stand-alone" mode. She found that the application used a toolbar and menus as the primary GUI components. Hence, she wanted to provide the QJBrowser-specific functionality in the editor using a new menu and a toolbar item. We shall describe how she added her new menu in the editor in the following paragraphs.

i.

Selection:	`shortname(M,'main'),method(M,C),` `callgraph(M,G).`
Organization:	`C,M,G`

Using this view, the developer traced the callgraph of the single application entry point in the software. She looked for all methods in the callgraph that had names that were suggestive of creating a menu or adding a menu to the menu bar. There were methods like `makeFileMenu`, `makeEditMenu` etc, that could potentially be the ones creating/adding menus in the application. These methods were part of a class called `EditorFrame`. She inspected their source code and found that they indeed created and added new menus to the *menu bar* of the application. The *menu bar* was passed as a parameter to them in the menu creation protocol.

ii.

Selection:	`shortname(E,'EditorFrame'),` `subtype(E,P).`
Organization:	`E,P`

The developer's immediate subtask was to find a way to get at the reference to the application's menu bar so that she could add her own menu to it. Since the menus were created and added by methods in the `EditorFrame` class, the developer decided to explore this class a bit further.

She used the view above to locate the parent classes of `EditorFrame`. This view revealed that `EditorFrame` inherited from a class called `java.awt.Frame`. Knowing that `java.awt.Frame` had a method called `getMenuBar` that could be used to add a new menu to the existing menu bar, the developer set out to find how the menu actions were handled in the suite.

iii.

Selection:	`shortname(M,'makeFileMenu'),` `callgraph(M,P), method(P,C),` `subtype(C,'java.awt.event.ActionListener')`
Organization:	`C,M,G`

This view showed all methods that belonged to a class that was a subtype of `java.awt.event.ActionListener` and were called in the menu creation sequence. The developer was aware that `java.awt.event.ActionListener` represented the interface responsible for handling menu item actions.

She found from the result of this query that a new instance of a class called `MenuItemActionTranslator` was created for handling menu actions.

iv.

Selection:	`shortname(M,'actionPerformed'),` `shortname(C,'MenuItemActionTranslator'),` `method(M,C),callgraph(M,A)`
Organization:	`M,A`

This view was defined to examine the course of the `actionPerformed` method, the method that is a part of the signature of `java.awt.event.ActionListener`, in the `MenuItemActionTranslator` class. It showed that this method called just one method called `doAction` in an interface called `DoActionListener`.

v.

Selection:	`shortname(P,'DoActionListener'),` `subtype(C,P), shortname(M,'doAction'),` `method(M,C), \+(modifier(M,'abstract').`
Organization:	`C,M`

Having found that a method called doAction might be responsible for handling the menu actions, the developer decided to explore the definitions of this method in the classes that implemented DoActionListener. She used the parameters listed above for this purpose, and found that EditorFrame, the class that created the different menus and added them to the menu bar, had a definition of this method and hence handled menu actions.

So the developer decided to subclass EditorFrame, add a method called makeReifyMenu in the menu creation sequence, and override doAction to handle her new functionality appropriately.

vi.

Selection:	`member(A,C), shortname(A,N), (atom_concat('save',_,N); atom_concat('load',_,N)).`
Organization:	`C,A`

The next step was to get hold of the absolute name of file being edited in order to update the source model with the relevant data. Using the query above, our developer looked for methods that saved or loaded a file to find out how a reference to the name of the corresponding file was maintained in the class. From examining the code of the resultant methods, the developer found that a member called CurrentFile belonging to class EditorFrame was used to refer to the file being edited.

vii.

Selection:	`shortname(C,'CurrentFile'), fieldaccessinfo(M,C,_).`
Organization:	`M`

Selection:	`shortname(C,'CurrentFile'), modifier(C,Mod).`
Organization:	`Mod`

Using these queries, the developer found that the variable, CurrentFile, had package access and there were no inspector methods that returned its value. Hence, it was virtually impossible to get at the current file without modifying the original JE package.

viii. The developer decided to alter the original package to provide a new public method in the class `EditorFrame` that would return the value of the `CurrentFile` variable. She also subclassed `EditorFrame` and provided her own `doAction` method that would trigger the *reification engine* to update the source model with current changes.

C.3 Summary

This section presented the major steps and observations while performing two simple development tasks using QJBrowser. It should be noted that most of the views described in this section are highly tailored to the user's needs. They use parameters that are specific to the application being dealt with.

This section provided evidence that it is possible to use QJBrowser to explore concerns, like comprehending and modifying software, in realistic situations. However, since we did not perform any detailed user studies with the tool in the limited time available to us, it is not possible to predict the usability of the tool from an end-user perspective.