Supporting Conceptual Queries Over Integrated Sources of Program Information

by

Brian de Alwis

M.Sc., University of British Columbia, 2003
B.Math., University of Waterloo, 1996

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

Doctor of Philosophy

in

THE FACULTY OF GRADUATE STUDIES
(Computer Science)

The University of British Columbia
(Vancouver)

April 2008
© Brian de Alwis
Abstract

A software developer explores a software system by asking and answering a series of questions. To answer these questions, a developer may need to consult various sources providing information about the program, such as the static relationships expressed directly in the source code, the run-time behaviour of a program recorded in a dynamic trace, or evolution history as recorded in a source management system. Typically, a developer consults these sources through a variety of software exploration tools. Despite the support afforded by these tools, developers often struggle to find the necessary information to answer their questions and may even become disoriented, where they feel mentally lost and are uncertain of what they were trying to accomplish.

This dissertation advances a thesis that a developer’s questions, which we refer to as conceptual queries, can be better supported through a model to represent and compose different sources of information about a program. The basis of this model is the sphere, which serves as a simple abstraction to provide a unified view of a source of information about a program. Many of the software exploration tools used by a developer can be represented as a sphere. Spheres can be composed in a principled fashion to form a single queryable knowledge-base suitable for answering conceptual queries, such that information from a sphere may replace or supplement information from a different sphere. Using our sphere model, for example, a developer can use dynamic runtime information from an execution trace to replace information from the static source code to see what actually occurred.

We have implemented this model in a configurable tool, called Ferret. We have used the facilities provided by the model to implement 36 conceptual queries identified from the literature, blogs, and our own experience, and to support the integration of four different sources of program information. Establishing correspondences between similar elements from different spheres allows a query to bridge across different spheres in addition to allowing a tool’s user interface (UI) to drive queries from other sources of information. Through this effort we show that sphere model broadens the set of possible conceptual queries answerable by software exploration tools.
Through a small diary study and a controlled experiment, both involving professional software developers, we found the developers used the conceptual queries that were available to them and reported finding Ferret useful. In the controlled experiment we found that any effects were dominated by the task and the strategies employed by the individual developers. During the analysis of the study results, we found it useful to apply the Task Load Index (TLX) for assessing task difficulty; this index has not been previously used in evaluation of software development tasks. We also introduce and show the usefulness of distance profiles for assessing the degree to which programmers remain on-track. The challenges we faced in conducting this study suggest that more research is needed into understanding the characteristics of tasks that make developers perceive them differently before future evaluations of this type are attempted.
# Table of Contents

Abstract .................................................................................................................. ii

List of Tables .......................................................................................................... viii

List of Figures .......................................................................................................... ix

1 Introduction ......................................................................................................... 1
   1.1 Overview of Related Work .......................................................................... 6
   1.2 Overview of Approach ................................................................................ 7
      1.2.1 Spheres: A Model to Compose Sources of Program Information ...... 7
      1.2.2 Ferret: An Implementation of the Sphere Model ............................ 9
   1.3 Evaluation ..................................................................................................... 10
   1.4 Organization of the Dissertation .................................................................. 11

2 Disorientation During Software Exploration .................................................. 13
   2.1 What is Disorientation? .............................................................................. 14
      2.1.1 Effects of Disorientation .................................................................. 14
      2.1.2 Causes of Disorientation .................................................................. 15
   2.2 Disorientation in Software Development .................................................. 15
      2.2.1 Study Format ..................................................................................... 17
      2.2.2 Overview of Study Results ................................................................ 19
      2.2.3 Characterizing Disorientation ............................................................. 23
      2.2.4 Factors Contributing to Disorientation ............................................. 24
      2.2.5 Strategies for Remaining Oriented .................................................... 27
      2.2.6 Study Validity .................................................................................... 29
   2.3 Relating Disorientation and Exploration of Software .............................. 30

3 The Sphere Model ............................................................................................... 32
   3.1 The Binary Relational Algebra ................................................................. 32
   3.2 Spheres ......................................................................................................... 33
   3.3 Representing a Software Analysis Tool as a Sphere .................................. 34
3.4 Composing Spheres  .............................................. 35
  3.4.1 Composing Sphere Relations  ............................ 35
  3.4.2 Composing Sphere Elements  .............................. 37
  3.4.3 Tracking Fidelity ........................................... 39
3.5 Expressing Conceptual Queries  .................................. 41
3.6 Summary .......................................................... 41

4 The Ferret Tool ....................................................... 42
  4.1 Using Ferret ...................................................... 42
    4.1.1 Requesting Queries ....................................... 43
    4.1.2 Investigating Query Results ............................. 44
    4.1.3 Configuring Ferret ....................................... 47
  4.2 Implementation of the Model ................................ 47
    4.2.1 Spheres .................................................... 48
    4.2.2 Conceptual Queries ...................................... 49
    4.2.3 Composing Spheres and Resolving Relations  ....... 50
    4.2.4 Supporting Correspondences ......................... 50
    4.2.5 Example of Resolving a Relation .................. 51
  4.3 Extending Ferret ............................................... 53
    4.3.1 Adding a Sphere ......................................... 54
    4.3.2 Adding a Conceptual Query ............................ 55
    4.3.3 Adding a Sphere Composition Function ............ 55
    4.3.4 Adding a Possible Correspondence ................ 56
  4.4 Summary .......................................................... 57

5 Evaluation .......................................................... 60
  5.1 Claim: Expressiveness of the model ....................... 61
  5.2 Claim: Relevance ............................................. 61
    5.2.1 Results .................................................. 63
  5.3 Claim: Improves Developer Effectiveness ............... 67
    5.3.1 The Experiment .......................................... 67
    5.3.2 Results .................................................. 77
    5.3.3 Discussion .............................................. 87
  5.4 Claim: Efficient Implementation .......................... 90
  5.5 Summary .......................................................... 92

6 Discussion .......................................................... 93
  6.1 The Sphere Model .............................................. 93
    6.1.1 Is This Model Necessary? .............................. 93
    6.1.2 Considering Other Composition Functions .......... 94
<table>
<thead>
<tr>
<th>C.1.1</th>
<th>Procedures Manual</th>
<th>139</th>
</tr>
</thead>
<tbody>
<tr>
<td>C.1.2</td>
<td>Autosave Task Description</td>
<td>142</td>
</tr>
<tr>
<td>C.1.3</td>
<td>Settings-Reloading Task Description</td>
<td>145</td>
</tr>
<tr>
<td>C.1.4</td>
<td>Ferret Tutorial</td>
<td>148</td>
</tr>
<tr>
<td>C.1.5</td>
<td>JQuery Tutorial</td>
<td>151</td>
</tr>
<tr>
<td>C.1.6</td>
<td>Suade Tutorial</td>
<td>154</td>
</tr>
<tr>
<td>C.2</td>
<td>The task Load Index (TLX) Questionnaire</td>
<td>156</td>
</tr>
<tr>
<td>C.2.1</td>
<td>TLX Questionnaire</td>
<td>157</td>
</tr>
<tr>
<td>C.2.2</td>
<td>TLX Factor Comparison Page</td>
<td>158</td>
</tr>
<tr>
<td>C.3</td>
<td>Distance Profiles</td>
<td>159</td>
</tr>
<tr>
<td>C.3.1</td>
<td>JQuery</td>
<td>159</td>
</tr>
<tr>
<td>C.3.2</td>
<td>Ferret</td>
<td>159</td>
</tr>
<tr>
<td>C.3.3</td>
<td>Suade</td>
<td>160</td>
</tr>
<tr>
<td>D</td>
<td>UBC REB Certificates of Approval</td>
<td>161</td>
</tr>
<tr>
<td>D.1</td>
<td>REB Certificate B04–0130</td>
<td>162</td>
</tr>
<tr>
<td>D.2</td>
<td>REB Certificate B06–0201</td>
<td>163</td>
</tr>
<tr>
<td>D.3</td>
<td>REB Certificate H06–80201</td>
<td>164</td>
</tr>
</tbody>
</table>
List of Tables

2.1 Summary of tasks observed with reported complexity ........................................ 20
2.2 General summary of observations and responses .............................................. 21

3.1 Examples of some possible elements and relations in a Java-based software system ................................................................. 35

4.1 Assigned categories for Ferret’s conceptual queries .................................... 44
4.2 Conceptual queries currently defined for Ferret ........................................... 58

5.1 Ferret usage over the course of the diary study ........................................ 65
5.2 Use of the conceptual queries from Study #1 ........................................... 65
5.3 Overlap between Ferret’s static Java conceptual queries and the queries directly supported by JDT, jQuery, and Suade .................................................. 70
5.4 Allocation of developers per treatment group ........................................... 72
5.5 ANOVA table for TLX  ............................................................................. 78
5.6 ANOVA table for unique program elements viewed ..................................... 81
5.7 ANOVA table for percentage of solution program elements correctly identified ................................................................. 87
5.8 Benchmark timings for use of Ferret for two different sphere configuration ................................................................. 92

6.1 Examples of other possible spheres .......................................................... 102

B.1 Implemented relation definitions in Ferret ................................................. 129
# List of Figures

1.1 Illustration of the burdens in mapping and scopings of concrete queries. .................................................. 3
1.2 Illustration of the burdens in composing concrete queries. .............................................................................. 4
1.3 Illustration of the burden in reasoning across tools. ............................................................................................ 5
1.4 The typical Ferret view in Eclipse showing the results of three conceptual queries about a Java interface named MCallEvent. .......................................................... 10

2.1 Two examples of environmental supports provided by Eclipse. ................................................................. 28

4.1 The typical Ferret view in Eclipse. ...................................................................................................................... 43
4.2 The context menu for a Eclipse Java Development Tools (JDT) query result in the Ferret view. ..................... 45
4.3 Ferret supports clustering results of conceptual queries. ................................................................................... 46
4.4 The Ferret sphere configuration wizard, used to compose one or more spheres. ........................................ 48
4.5 The correspondence broker network in Ferret. ................................................................................................. 52
4.6 The Ferret domain model. ............................................................................................................................... 54
4.7 A PDE query whose result is expressed as a frame. ......................................................................................... 55

5.1 The typical JQuery view in Eclipse. .................................................................................................................. 68
5.2 A typical Swade view in Eclipse. ...................................................................................................................... 69
5.3 Differences in TLX scores, separated by tool and task order, and by task-order only. .................................... 78
5.4 Within-subject differences in number of unique program elements visible to developers, separated by task and tool (jittered). .................................................. 80
5.5 Distance profiles for several users for the two tasks. ......................................................................................... 82
5.6 Comparisons of the percent time looking at elements directly relevant to the tasks. ................................... 84
5.7 Between-subjects percentage of solution program elements respectively selected, visible, and identified by the developers. .................................................. 85
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.8</td>
<td>Within-subject differences in the percentage of solution elements correctly identified, separated by tool and task.</td>
<td>86</td>
</tr>
<tr>
<td>6.1</td>
<td>Placing conceptual queries in context.</td>
<td>96</td>
</tr>
<tr>
<td>B.1</td>
<td>The JDT domain model.</td>
<td>131</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

As soon as we attempt to diagram software structure, we find it to constitute not one, but several, general directed graphs, superimposed one upon another. The several graphs may represent the flow of control, the flow of data, patterns of dependency, time sequence, name-space relationships. – Brooks [11], pp 185–186

Modern software systems are complex and continuously evolving [4]. Before undertaking a change to any realistic software system, a software developer must spend a large amount of time exploring and re-familiarizing themselves with the system to determine and understand the parts of the system relevant to that task, and to gauge the impact of the change on the behaviour of the system [86, 88]. This exploration and refamiliarization process is the dominant activity of a change task [16] and was estimated in one case to have accounted for 47% of the total time taken to complete a task [73, p.475].

A developer explores a software system by asking and answering questions about the system [84, 88]. To answer his questions, a developer may need to consult various sources providing information about the program, such as the static relationships expressed directly in the source code, the run-time behaviour of a program recorded in a dynamic trace [92], the bugs corrected in some part of the source code as described in the problem reports database [18], or the evolution history of the system as recorded in the source control management system [99]. A developer uses a variety of tools to help answer their questions; each tool typically presents
information from a particular information source. We call the questions that are answered directly by tools concrete queries. Despite the support afforded by current tools, developers often struggle to find the necessary information to answer their questions [59, 79, 84, 88].

In their study of the questions asked by developers, Sillito et al. [84] observed that many of the questions asked by a developer are either poorly or only indirectly supported by the concrete queries provided by tools. This mismatch between the questions the developers want to answer, which we refer to as conceptual queries, and the questions they are directly able to answer using the available tools (concrete queries) causes a developer to spend time and effort decomposing his conceptual queries into a form that can be answered using the available concrete queries. A developer must also synthesize the results from the concrete queries executed to answer their particular conceptual query. In reflecting on the underlying causes of disorientation that we have identified from our own study of software developers at work (discussed in Chapter 3), we have identified three distinct cognitive burdens that arise in answering a conceptual query.

1. **Mapping and Scoping**: Currently the developer is responsible for mapping a conceptual query onto the available concrete queries provided by a tool and in scoping the possibly broad results to just those of interest. For example, to search for the program locations in a Java [38] program where a particular exception is thrown using the commonly available searching tools in the Eclipse Java Development Tools (JDT), a developer must search for references to the exception type (mapping) and then must manually examine the results (scoping) to distinguish just those references performing a throw (Figure 1.1). Specifically, a developer must pick out the references of interest from those involving variable declarations, method declarations that throw the exception, subclasses extending the exception, statements which include a cast to the exception or that check the type of a variable is the exception type, or methods that catch an exception of the type.

---

1The Eclipse Java Development Tools (JDT) is an integrated set of editing, debugging, and source maintenance tools for Java development. (http://www.eclipse.org/jdt/ verified 2007/12/03)
2. *Composing Results:* A developer may need to *compose* the results from multiple concrete queries to answer a conceptual query. This composition often involves merging the sets of results produced by the different concrete queries to answer the original conceptual query. More difficult are situations where a developer must use the results from one query to drive another query [84]. This situation is analogous to a relational join as supported by database systems [19], but must be performed manually by the developer and the developer must carry the context of both queries in their head. For example, answering a conceptual query about where instances of a particular Java interface are created requires searching for the instantiators of each of the interface's implementing classes (Figure 1.2). This example might occur when changing some Java code from using the type Set from the Java Collections library to use the type Bag instead.

3. *Integrating and Reasoning Across Different Tools:* A developer may be required to *integrate and reason across* several sources that provide information about a program. This particular burden builds on the two burdens identified above with additional complications from using different information sources with different representations. This burden takes two forms. The first form arises when two (or more) information sources have some overlap in the types of

---

**Figure 1.1:** Illustration of the burden in mapping and scoping the results of concrete queries; a ‘C’ is a class and an ‘M’ is a method. Finding the location throwing a particular exception requires two steps: (1) find all methods and classes referencing the exception type (the labelled arcs), and (2) manually examine each result to eliminate non-throws.
Figure 1.2: Illustration of the burden in composing concrete queries; an ‘I’ is an interface, a ‘C’ is a class, and an ‘M’ is a method. Finding the instantiators of an interface requires two steps: (1) find the interface’s implementing classes, and for each class (2) find the methods instantiating the implementing class.

information represented. For example, when a dynamic execution trace is available from a particular run of the program, a developer can choose to consider the program either from its actual run-time behaviour as represented by the trace, or from the possible behaviours as represented in the source code. Here the developer must choose between two descriptions of the program’s behaviour that differ in the precision provided by their concrete queries [83].

The second form of this burden arises when two (or more) information sources maintain different internal representations for what a developer would plausibly consider to be the same element. The developer is then responsible for establishing and carrying this correspondence when making further queries. Establishing such correspondences is trivial for the developer when there is an exact one-to-one relationship between the tool representations; for instance, it is easy to map between representations of a method declaration in two static Java cross-reference tools. Other correspondences, such as when an element bears some similarity to another or when an element corresponds to a group of elements, can be more difficult to make. For example, in examining a dynamic execution trace, a developer wishing to reason about the callers of a method defined on a Java interface must extend his reasoning to the callers of the implementations of that method (Figure 1.3).
Figure 1.3: Illustration of the burden in reasoning across tools; the stylized ‘m’ represents a method in a different tool. Finding the callers of an interface method from a dynamic trace requires (1) finding the interface method’s implementations in the static Java code, then for each such implementation method (2) finding the methods in the dynamic trace corresponding to the implementation method, and finally for each dynamic method (3) finding the callers of the corresponding dynamic methods.

These three cognitive burdens impose a drag on a developer’s ability to answer his conceptual query. Faced with these burdens, a developer may choose one of three approaches.

- The developer may reformulate his approach for answering his current conceptual query to use other tools. But this approach may lead to the developer missing information relevant to the question.
- The developer may choose an entirely different line of inquiry. But the time spent to this point may have been wasted.
- The developer may decide to persevere and accept the additional work required to manage these burdens. This extra work may account for reports of developers spending significant amounts of time simply navigating through the source code of a system [57, 68], examining code irrelevant to their tasks [57, 79, 88], or becoming disoriented during software development (which we discuss in Chapter 2) [20].

This dissertation describes an approach for expressing conceptual queries that we believe addresses the three cognitive burdens. The thesis of this dissertation
is that a model can be introduced to represent and compose different sources of information about a program so as to broaden the set of possible conceptual queries that can be answered using software tools. By directly answering conceptual queries, the cognitive burdens are shifted from the developer to the software tools [21].

1.1 Overview of Related Work

At first glance, it is surprising that these problems persist despite the significant efforts that have been devoted to integrating tools and formulating query languages to support software development.

The efforts in tool integration have focused on sharing artefacts between independent tools [e.g., 90], or triggering transformations upon a change to an artefact [e.g., 112]. These solutions are aimed at preventing inconsistencies developing between artefacts from different standalone tools as the artefacts evolve over the lifetime of the software system. The solutions are concerned with propagating changes between tools rather than helping developers answer queries using the different tools.

Modern integrated development environments (ides) provide a different form of tool integration in that they allow other tools to be accessed from within a consistent environment. But the integration is relatively shallow and ad hoc: tools can be invoked and results displayed in the environment, but the results are shown in separate windows or views. Any integration between two tools generally occurs as one tool establishing a bridge to the domain model of the other; this integration is often only unidirectional. For example, the Eclipse Test and Performance Tools Platform (TPTP)\(^2\) provides some support for opening the Java source code corresponding to a profiling element, but there is no way to open the profiling elements from a corresponding Java element. Where there is no integration between the tools, the developer must perform the work to switch to the other tool, find the corresponding elements, and continue his query.

Query languages, and tools based on query languages, provide significant power to express and compose queries. But the few implementations that support cross-

\(^2\)The Eclipse Test and Performance Tools Platform (TPTP) provides an integrated set of tools for profiling and testing applications developed using the Eclipse platform. TPTP is itself developed using the JDT. ([http://www.eclipse.org/tptp/](http://www.eclipse.org/tptp/ verified 2007/12/03))
artefact querying, such as GSEE \(^{28}\) and sextant \(^{81}\), assume that the artefact domains are complementary, with no overlap, and where there is a correspondence between elements from different artefact domains, the correspondence is exact and one-to-one. Even if the query languages could be extended to incorporate other sources of program information, they lack a mechanism to select or prioritize between sources of information with overlap. And so the developer must perform this work themselves.

1.2 Overview of Approach

This dissertation describes an approach for expressing conceptual queries that helps a developer avoid the three cognitive burdens defined above. We have defined a model that supports the composition and integration of different sources of program information to form a single queryable knowledge-base suitable for answering conceptual queries. The use of this model addresses the two previously identified issues of choosing between overlapping sources and establishing correspondences between similar elements.

We have implemented this model in a configurable tool, called Ferret, that expands the ability for directly supporting a developer’s conceptual queries. We have used these facilities to provide a rich set of conceptual queries identified from the literature, blogs, and our own experience. By directly answering conceptual queries, we push the mapping, scoping, and composition issues to the tool, to be dealt with at the time of configuration \(^{21}\).

1.2.1 Spheres: A Model to Compose Sources of Program Information

Underlying our tool is a tool- and language-agnostic model based on the binary relational algebra\(^{3}\) for representing and integrating information from different sources of program information. In this model, each source of information about a program is represented as a sphere\(^{4}\). A sphere captures both the individual elements defined by the source and the relations as expressed by the source’s concrete queries. For instance, the static Java relationships that can be extracted from the source code

---

\(^{1}\)We provide an overview of the binary relational algebra in Appendix A.

\(^{2}\)Spheres are named for the cold-war concept of competing spheres of influence, with the attendant contention from overlapping boundaries.

---
forms a sphere, the revision history recorded in a software repository forms a sphere, and the method calls information embodied in a dynamic execution trace forms a sphere. A conceptual query is expressed as a sequence of operations on binary relations, where the relations are resolved by name against a provided sphere, which may be a composite sphere.

The model provides a set of composition functions for combining spheres. The composition functions define how the different spheres are consulted when resolving a relation. This means that the user can specify the precedence of sources of information during consultation. The model also defines a mechanism for expressing correspondeces between elements from different spheres. Correspondences arise when two spheres use different representations for what a human developer would consider to be the same element. Correspondences are often one-to-one, such as occurs when two static Java cross-reference tools handle the same method. But correspondences may also be one-to-many when a concept in one sphere's domain has no single equivalent in another sphere. For example, the dynamic Java run-time behaviour has no representation of a Java interface, but a developer may reason across the interface by considering its implementing classes. As the correspondences between elements may be imperfect, the model captures and propagates the fidelity of the correspondences through the query. The correspondences of the source element are also checked when resolving a relation, thus an element may have more relations than previously supposed.

Our model provides a number of benefits over existing alternatives:

1. Through the composition functions, the model supports expressing conceptual queries that use different sources of program information.

2. The model supports the unanticipated integration of different sources of program information. The use of correspondences allows elements shown in a tool's user interface (UI) to drive queries from other sources of information. Different tools can be developed independently, which helps contain the complexity of developing integrated packages.

3. The model preserves the full expressiveness of the binary relational algebra while simplifying the logic expressions that would otherwise be required to provide equivalent functionality.
1.2.2 Ferret: An Implementation of the Sphere Model

We have implemented this model in a new software exploration tool, named Ferret. This implementation serves to demonstrate that the sphere model can express sufficiently rich conceptual queries, that the sphere model can be implemented in a reasonably efficient manner, and also supports experimentation with conceptual queries. We have integrated four sources of program information, and have implemented a number of conceptual queries identified from the literature, blogs, and our own development experiences. A number of these conceptual queries use information from multiple spheres. A number of these conceptual queries have not been previously implemented, and their implementations benefit from being able to compose spheres.

Ferret is implemented as a single view in the Eclipse IDE. Upon being invoked for a particular element, Ferret computes and displays the results of all known conceptual queries in a structured tree view. Queries that are not supported for the query source element and queries with no results are not displayed. Figure 1.4 shows an example of Ferret in action, displaying the results of three conceptual queries (indicated with a ‘Q’) about a Java interface named MCallEvent. The first conceptual query lists the classes that implement the interface MCallEvent. The second conceptual query lists the methods that instantiate MCallEvent. The final conceptual query lists the methods declared to return MCallEvent. We have found that selecting a single conceptual query from the available conceptual queries is somewhat daunting; by computing and displaying all conceptual queries, the answers to the different conceptual queries summarizes how the query program element relates to the software system.

The particular example shown in Figure 1.4 used a knowledge-base where a sphere representing a Java program’s dynamic run-time behaviour has been used to replace or supersede a sphere representing the static relations described in the application’s source code. This sphere composition means that when a conceptual query requests the callers of a particular Java method, the callers will be identified using the dynamic run-time sphere rather than the static Java sphere.

\footnote{A developer familiar with Eclipse Java Development Tools (JDT) and Eclipse Test and Performance Tools Platform (TPTP) would notice that the icons used in the first and second queries are from different icon sets.}
Figure 1.4: The typical Ferret view in Eclipse showing the results of three conceptual queries about a Java interface named MCallEvent.

Ferret provides a number of features to support examining the results of the different conceptual queries. The results of each query may be clustered by attributes of the results; the third query in Figure 1.4, for example, has had its results clustered by their declared package. For each conceptual query, Ferret displays the number of results for the query, and the number of possible clusterings of the results. We describe Ferret in more detail in Chapter 4.

1.3 Evaluation

Substantiating our thesis requires addressing four claims: (i) demonstrating that our model broadens the set of conceptual queries answerable by software exploration tools, (ii) showing that the conceptual queries that we have implemented are relevant to developers, (iii) demonstrating that the use of conceptual queries improves a developer’s effectiveness during software exploration activities, and (iv) demonstrating that the sphere model and conceptual queries can be efficiently implemented.

Claim 1 (Expressiveness of the Model): The sphere model broadens the set of possible conceptual queries answerable by tools.

We demonstrate the expressiveness of the model by showing that the model supports the definition of a range of conceptual queries that have been discussed in the literature and requested or motivated from developer blogs. Some of these queries are implemented in other tools; others are unique to our tool. Some of these queries benefit from being able to integrate different sources of program information.
Claim 2 (Relevance): The conceptual queries implemented are relevant to developers.

We demonstrate the relevance of the conceptual queries implemented by reporting on a diary study of four professional software developers using Ferret as part of their day to day work. The diary study examined their usage of the implemented conceptual queries and the perceived utility of composing dynamic and static information in their work.

Claim 3 (Developer Effectiveness): Direct support for conceptual queries helps improve a developer’s effectiveness during software exploration activities.

We assess the impact on a developer’s navigation by examining the results of a controlled experiment of 18 professional software developers planning complex changes to an open-source codebase. This experiment compared the impact from using Ferret or one of two other software exploration tools with Eclipse alone [22]. The tools were compared by their effects on the developers’ subjective mental workload, on their exploration behaviour, and on the correctness of the solutions. We find a seeming contradiction in that we find no apparent effect from Ferret or either of the other two tools, yet the majority of the developers using Ferret report finding Ferret to be useful and ask about the availability of the tool. This interest suggests that Ferret was providing useful and relevant information to the developers, but the effects from improvements to search operations are dominated by individual strategies and the cognitive requirements of programming tasks.

Claim 4 (Efficient Implementation): The sphere model and conceptual queries can be efficiently implemented.

We show that the sphere model can be efficiently implemented by reporting on a set of benchmarks of Ferret in use on a real software system.

1.4 Organization of the Dissertation

In the remainder of this dissertation we present the details of the sphere model, its implementation in the Ferret tool, and demonstrate how the sphere model broadens the support for expressing conceptual queries.
In Chapter 2 we provide an initial characterization of disorientation as experienced by software developers. We identify a number of factors that appear to exacerbate the onset of disorientation, as well as identifying factors that help developers in remaining oriented. This study helped refine our understanding of disorientation and how it related to the three cognitive burdens identified above. This analysis of disorientation during software development forms the first contribution of this dissertation.

In Chapter 3 we present a model capable of representing and integrating different sources of program information. These sources are represented by a notion called a sphere. This model is tool- and language-independent. The use of spheres supports answering a subset of the conceptual queries asked by developers. The description of the model and the elements of the model supporting the composition of different sources form the second contribution of this dissertation.

We present an implementation of this model in a new software exploration tool, named Ferret, in Chapter 4. Ferret is an instantiation of our model for four different spheres for Java applications. We further describe several of its features for helping investigate the results from a conceptual query that should help in preventing disorientation. Ferret provides implementations of a number of conceptual queries, some of which have not been implemented previously in other exploration tools. The Ferret tool forms the third contribution of this thesis.

Based on the model and tool defined in Chapters 3 and 4, we present an evaluation of the thesis of this dissertation in Chapter 5. This evaluation shows that the sphere model is sufficiently expressive to actually implement an interesting set of conceptual queries, and that the conceptual queries that we have implemented are actually of use. We describe the results of an experiment to study whether the improved support for conceptual queries leads to a measurable impact on the effectiveness of software developers at work.

In Chapter 6 we discuss the main issues that arose during the development and investigation of the approach described in this thesis, and identify some issues for future work.

We place our work in perspective to related work in Chapter 7, and finally conclude and summarize the dissertation in Chapter 8.
Chapter 2

Disorientation During Software Exploration

In Chapter 1, we described three cognitive burdens for developers that arise in answering even seemingly-simple conceptual queries with existing tools. We first became aware of these burdens from our investigations into occurrences of disorientation during software development [20]. Disorientation is a mental condition where a person feels that they are lost. People become disoriented when using a computer-based system because of difficulties in obtaining the information required to perform their task [110]. In this chapter, we describe an observational field study into whether and how software developers, a class of computer users not previously studied, experience disorientation. Through this study we refined our understanding of disorientation and how it related to the three burdens detailed in Chapter 1.

We begin by reviewing background work to disorientation as it applies in other computer-based fields in Section 2.1. We then describe our observational field study in Section 2.2. From this study we identified a set of three factors we believe contributes to disorientation in software developers. We conclude the chapter in Section 2.3 by relating these three contributing disorienting factors to the three burdens from the Introduction.
2.1 What is Disorientation?

Disorientation has been studied in other computer-based domains, such as menu systems [25, 62], medical monitoring systems [111], hypertext (where it is often called the lost in hyperspace phenomenon) [e.g., 15, 30, 55], and spreadsheets [104]. Mantei, in the earliest study of disorientation, described disorientation as the self-awareness of being mentally lost (subjective lostness), as compared to—and independent of—being actually lost (objective lostness) [62]. Elm and Woods describe disorientation as when “the user does not have a clear conception of relationships within the system, does not know his present location in the system relative to the display structure, and finds it difficult to decide where to look next within the system” [25, p. 927].

Watts-Perotti and Woods [104], in summarizing much of the research on disorientation, describe that disorientation may manifest in several ways: (i) as feelings of lostness, where the users are unable to determine where they are or what they are examining; (ii) as display thrashing, where users must repeatedly switch between different displays, referring to the content displayed on a computer screen at some particular time, to correlate information of interest; and (iii) as mental workload from interface management.

2.1.1 Effects of Disorientation

Disorientation is not immediate, but sets in after some time [62]. The work on disorientation in hypertext separated the effects of disorientation into three categories [55]:

**Navigational Disorientation Problems**: where users are unable to access items of interest or identify their location, caused by incomplete or incorrect knowledge of the information structure;

**Task Management Problems**: where users fail to either resume or complete previously-suspended tasks, or do not pursue planned digressions;

**Information Management Problems**: where users are unable to synthesize or remember information found.
Users develop work-arounds to prevent the occurrence of disorientation and may even deliberately avoid the use of displays even though they provide additional information \[104\].

### 2.1.2 Causes of Disorientation

Disorientation may be induced by poor structure in both the organization of the information space (the content) and the user interface (the medium) \[62\]. A structure is considered to be more navigable when it is possible to easily find a short path to a target node by making decisions based only on information available at each step \[31\]. Information spaces that are structured as large, irregular networks are much likelier to induce disorientation during navigation activities; the introduction of some semblance of regular structure on the underlying network can dramatically change their navigability \[31\] and aid in preventing disorientation \[62, 70\].

The UI plays an essential role in discovering the structure in an information space, and thus in preserving orientation \[62\]. Computer-based systems are able to present only a very small window onto a very large information space, rather like peering through a keyhole. This small aperture can make it difficult to locate information or to piece together a coherent picture of a situation \[104\]. The UI can help the user by adding structure to an information space and by organizing the information using meaningful categories to support more efficient traversals of the information space \[31\]. When the UI provides inadequate support to the user to navigate within this information space, the interface becomes a set of serial displays, where each display is perceptually independent of both its prior and subsequent displays. The user must instead carry the mental burden of transitioning and reorienting between each display.

Disorientation is not a phenomenon restricted to novice users, but has also been found to affect expert users of an interface \[25, 62, 104\].

### 2.2 Disorientation in Software Development

Although there are passing references to disorientation during software development tasks found in the research literature \[e.g., 5, 39, 96, 97\], and some tools appeal to preservation of user context as an argument to the effectiveness of their tech-
niques [e.g., 40, 49, 91], we have found no studies reporting on how and when disorientation occurs for software developers. To help address this gap, we conducted an exploratory field study. Our qualitative study sought to answer two questions: (Q1) Do expert developers become disoriented? Or is disorientation only a phenomenon suffered by novices or newcomers? (Q2) Are there factors of the development environment that exacerbate or prevent disorientation, or aid in recovery from disorientation?

We observed and interviewed eight developers from the Eclipse Project as they conducted their normal development work using the JDT. We also interviewed two other IBM developers who used the JDT, but who were working on a closed-source system. These expert developers were working on their normal tasks on a large software system.

We expected that developers should become disoriented over the course of the study. Software systems share many characteristics of the information spaces used in the other computer-based domains mentioned above: development tasks generally require correlating information gleaned from different sources, such as different views or different tools; source code contains links or relationships that refer to one or several different locations elsewhere in the source code; and developers use specialized tools to help navigate through the relationships embedded in the source code. Software systems also have three important differences that we believed would further exacerbate the onset of disorientation. The first difference relates to the ability to find or create an organizing structure over the information space. The other computer-based domains either tend to allow creators to change the underlying organization of the information structure, or the information structures of these other domains typically have an inherent spatialization. For example, hypertext authors can choose a linear document structure or a seemingly-random structure; spreadsheets have an inherent two-dimensional spatial structure. In contrast, software developers rarely have the opportunity to change the structure of their programming languages and due to the sheer size and diversity of the relations realized in the software system, there is no inherent spatialization [11]. The second difference lies in the rates of change in software systems: software systems are continuously changed and gradually lose their planned coherency [4]. Thus developers must continually explore or refresh their knowledge of parts of the system.
During our study, we witnessed two episodes of disorientation, and two developers reported experiencing disorientation (which had not been apparent from our observation). Furthermore, eight of ten developers interviewed reported having become disoriented on previous occasions. However, we were surprised not to have witnessed more episodes of disorientation. In hindsight, a paucity of disorienting events is not surprising as developers would have almost certainly abandoned a tool that induced unbearable amounts of disorientation, or have at least developed work-arounds.

We examined patterns of complaints about the Eclipse UI by using visual momentum (Section 2.2.1) to identify factors that contribute to the disorientation.

2.2.1 Study Format

The field study was conducted at IBM Canada’s Ottawa Software Laboratory in the two weeks prior to the M8 milestone release of Eclipse 3.0. Of the ten participants, eight were active developers of the Eclipse Platform (referred to as P1–P8), and the other two were former Eclipse Platform developers (referred to as E1,E2) who were now developing a closed-source system. All participants normally use Eclipse for Java development. Although the participants worked on the Eclipse Platform, none were from the teams responsible for developing the actual JDT. We chose this population to eliminate the possibility of disorientation occurring because of novice effects (Q1). We chose to examine Eclipse because it is a popular state-of-the-practice IDE and because the Eclipse Project had previously undertaken an effort to address suspected factors relating to complaints of disorientation [17].

We observed P1–P8 separately for two hours each as they pursued their normal development tasks, collecting video tapes, screen captures, and detailed notes. We were unable to observe E1,E2 at work due to confidentiality concerns. We also conducted semi-structured interviews: a brief interview was conducted after one hour to ask about the participant’s progress and current approach, and a longer interview took place at the end of the session to obtain more detail. This study design allowed us to stress realism, an important consideration as we assumed disorientation was more likely to occur as developers work on larger systems on non-trivial tasks.
Questioning Convention

During the interviews participants were asked about their program navigation approach, the tools they used, the difficulties they encountered, and specifics of their chosen change tasks. Questions asked included: What information did you access? How did you find it? How did you know it was relevant? What tools did you use? Were any your own custom tools? Did they provide too much information, too little? Did you run into any problems finding the necessary information? What kinds of information do you wish could have been available? Questions were also asked about the specifics of the change task. Only towards the conclusion of the final interview were the participants asked outright if they had experienced any episodes of lostness during the session, and for specific details. Did you become lost during the session or in previous sessions? What did that entail? All interviews were taped and subsequently transcribed word for word, including features such as pauses and inflection.

Method of Qualitative Analysis

Our analysis in brief involved analyzing the interview transcripts and field notes to identify difficulties described in using the Eclipse user interface, and correlating them with the screen captures. The interview transcripts were first perused to identify interesting comments. They were then read through systematically, and events and supporting quotes extracted and grouped into common themes. The screen captures were analyzed to gather information on how the participants used features of their development tools or other applications, including counting the number of transitions between Eclipse windows and other applications, such as e-mail readers or web browsers, and the number of times where there was a total replacement of content—where information in a window became completely obscured. We then used visual momentum to assess the situations around the occurrences of disorientation.

Visual Momentum The navigation support afforded by a system can be qualitatively assessed using visual momentum [109]. Visual momentum is a heuristic measure of a user’s ability to extract relevant information across views and displays, and
can serve as a proxy measure for evaluating a system’s ability to convey the underlying network of an information structure. This concept was inspired by techniques used in cinematography to guide viewers where to look next while preserving their sense of orientation. Interfaces with high visual momentum aid in co-ordinating the information displayed: users are able focus completely on their task and are not concerned with managing the interface. Interfaces with low visual momentum require the user to carry the mental burden of transitioning and reorienting between each display.

There are several techniques that increase the visual momentum for an interface [104, 109]:

- **longshots**, high-level overviews that are generally tailored to a task;
- **functional overlays**, display surrounding context of the display currently being examined;
- **navigational landmarks**, identify prominent elements, helping users reorient between displays;
- **bookmarks**, record positions of interest;
- **spatial dedication**, support the establishment of a spatial frame of reference between the information displayed and its meaning.

Watts-Perotti and Woods [104] used visual momentum to demonstrate the impact of these techniques on alleviating disorientation in spreadsheets.

### 2.2.2 Overview of Study Results

The developers observed chose what they believed would be small to medium tasks that could be completed in a day: none were observed to have modified more than 25 files or classes during the study period. Table 2.1 presents a rough summary of their work tasks along with a self-assessed estimation of the task complexity in comparison to their normal tasks; an asterisk (*) indicates that the task proved to be more complex than expected due to difficulties that arose during the task. The first
Table 2.1: Summary of tasks observed with reported complexity as compared with other usual tasks; an asterisk (*) indicates tasks that proved to be more complex than expected.

<table>
<thead>
<tr>
<th>Task(s)</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1 Ensuring backwards compatibility of reworked subsystem; fixing test-suites</td>
<td>Average*</td>
</tr>
<tr>
<td>P2 Communicating information between separated layers</td>
<td>Less</td>
</tr>
<tr>
<td>P3 Implementing last-minute minor features, bug fixing, constructing test suites</td>
<td>Less</td>
</tr>
<tr>
<td>P4 Porting task; modifying component to use lazy computation</td>
<td>Less</td>
</tr>
<tr>
<td>P5 Finessing Eclipse start-up appearance</td>
<td>Less</td>
</tr>
<tr>
<td>P6 Attempting to understand the interactions between a set of event listeners as part of larger task</td>
<td>Hard</td>
</tr>
<tr>
<td>P7 Porting task</td>
<td>Average</td>
</tr>
<tr>
<td>P8 Script verification; bug identification and work-around</td>
<td>Less*</td>
</tr>
<tr>
<td>E1 Business process product development</td>
<td>–</td>
</tr>
<tr>
<td>E2 Business process product development</td>
<td>–</td>
</tr>
</tbody>
</table>

three rows of Table 2.2 further summarizes the tasks by whether they were working on code they had written, the number of files modified during the study, and whether they had been interrupted by others during the study.

During the study, four of the eight developers either reported (P2,P8) or were observed (P4,P6) becoming disoriented. After a prolonged period, P6 brought up a dialog to open a class (Open Type) and then momentarily blanked, unable to remember the name. We suspect he was mentally retracing his past steps in an attempt to recall the class name from the situation. P4 structured his development work using two different but similar-looking windows, one for making code changes and the other for modifying test suites. When returning to the code, having performed some testing, he was often visibly confused as to which window he should be using. P4 also said he must remember to later optimize some quickly hacked code, but did not write this down and was not subsequently observed to have performed the task.

P2 and P8 reported becoming lost during their development task. These occa-
Table 2.2: General summary of observations and responses.

<table>
<thead>
<tr>
<th></th>
<th>P2</th>
<th>P8</th>
<th>P6</th>
<th>P4</th>
<th>P1</th>
<th>P3</th>
<th>P5</th>
<th>P7</th>
<th>E1</th>
<th>E2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modifying own code?</td>
<td>75%</td>
<td>50%</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>shared</td>
<td>shared</td>
</tr>
<tr>
<td># Files modified</td>
<td>5</td>
<td>8</td>
<td>25</td>
<td>6</td>
<td>1</td>
<td>13</td>
<td>2</td>
<td>2</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>Interrupted?</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>Disoriented in study?</td>
<td>rep</td>
<td>rep</td>
<td>obs</td>
<td>obs</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Disoriented previously?</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Feels supported by Eclipse?</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Customizes Eclipse</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Runs Eclipse full-screen</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>–</td>
</tr>
<tr>
<td>Thrashing</td>
<td>high</td>
<td>high</td>
<td>high</td>
<td>med</td>
<td>low</td>
<td>med</td>
<td>med</td>
<td>med</td>
<td>–</td>
<td>–</td>
</tr>
</tbody>
</table>
sions were not readily differentiable from their other actions. P2, who was working on a problem that involved several well-separated components, reported omitting one of these components accidentally more than once, and forgetting which operations were processed by which components:

P2: When I was starting, the first time, I actually skipped a level...I do this frequently because we inserted a new level [...] months ago, but I still do it. I flip up to the run-time level and I should be at the [other] level. And then dealing with the command line, I always forget which things get processed at which level.

P8 reported getting lost in the file system and resorted to using very slow and considered movements to prevent becoming lost in some other directory. The follow-up interviews revealed that eight of the ten developers had experienced disorientation on previous occasions.

E2: Well sure, I can remember times sitting in front of the screen thinking, what am I looking at?

P3: When I'm [...] in browsing mode, I'll totally forget how I got to certain places.

Several noted that they rarely became lost with code with which they were familiar (P2,P3,E1):

E1: [When starting on this project] lot of the code was new to me so I had to browse a lot to see 'where was this class,' 'who references it,' 'what does it really do here.' Sometimes you go into [a] method [from] another method, and I have no idea where you were coming from any more, you forgot, you lost track. But that usually happens, from what I remember, with code that I wasn’t very familiar with.

The fourth and fifth row of Table 2.2 summarize which participants experienced disorientation. A value of rep in the row labelled 'Disoriented in study?' indicates that disorientation was self-reported during the interview; a value of obs indicates that evidence of disorientation was observed by the researcher. The following row, labelled 'Disoriented previously?', refers to a participant self-reporting having experienced disorientation in their past.

The remainder of Table 2.2 summarizes the participants’ use of Eclipse. We asked each participant whether he felt supported by Eclipse, and whether he customized Eclipse’s appearance beyond setting predefined preferences. Only four of the participants customize Eclipse’s appearance in terms of tool layouts (e.g., an Eclipse view or plug-in), with the remainder using Eclipse’s default configuration.
Most of the participants felt that the tools generally provided just the right amount of information (P1, P2, P5, P7, P8) – or even too much (E1, who used a minimal setup so as to avoid distractions). Most participants ran Eclipse as a single window maximized to occupy the full screen. The final row of Table 2.2 describes the amount of thrashing observed; this is described in more detail in Section 2.2.4.

Participants also spent time on other related tasks during the study period, such as responding to queries from users and other developers via e-mail, instant messaging, visits by other developers, and triaging and responding to problem reports. These other tasks were often interleaved during down-time, such as occurred during builds, waiting for Eclipse to start, or when fetching and committing files to or from the version control system. Although not formally verified, many of the e-mails received seemed to pertain to e-mails generated in response to changes to problem reports (P1, P3).

2.2.3 Characterizing Disorientation

In examining these incidents, we believe that the disorientation exhibited by developers can be characterized as two related sub-problems: navigational and task disorientation.

Navigational disorientation problems. We characterize these problems as when the developer:

- has lost their sense of location and direction (what am I looking at?);
- is unable to access an item of interest (why can't I find...);
- has lost their past history, unable to remember how they came to their current or past locations (what was I doing?).

Task disorientation problems. We characterize these problems as when the developer:

- cannot remember the task intended once at a location (what was I going to do?);
has pursued or was distracted by another, perhaps related, problem (an *embedded digression* [30]), but failed to return to the original task.

In forming this characterization of disorientation in software development, we considered both navigation problems as well as the developer's intent (or task). Our resulting characterization is similar to that resulting from the efforts in examining disorientation in hypertext (Section 2.1.1). However, we specifically exclude the information management problem category. In the hypertext categorization, this category generally recognizes a problem styled as “the art gallery problem” [30], where a browser encounters difficulty summarizing what has been browsed. In eliminating this third category, we follow the reasoning of Mayes et al. [64], whom emphasize a difference between disorientation in a spatial network (the structure of the information space) and in a conceptual space (the underlying structure of the information space). Some conceptual disorientation should be expected during knowledge-discovery tasks, where one's current understanding must be continually updated with new information garnered about the information space. Such learning forms a large part of the software exploration process, as developers explore the source code to update their understanding of the complex causal relations in the system code.

2.2.4 Factors Contributing to Disorientation

Eclipse provides many features that support high visual momentum, such as overlays of related information (Section 2.2.5). However, through our analysis, we identified three factors contributing to the Eclipse UI becoming a set of serial displays, which may lead to developer disorientation.

**Factor 1: Absence of connecting navigation context**

The first factor identified pertains to a lack of connecting context when switching between files. This problem occurred most often during program exploration, as developers follow program relationships to understand or assess parts of the source code.

Eclipse supports many sophisticated program navigation traversals such as finding all callers of a method, all references to a field, or the declaration of a class. Many
such functions are tied to hot-keys, meaning that they can be invoked with little effort, enabling rapid descents through a call chain. All developers that we observed made use of these program navigation operations.

These rapid descents may lead to examining multiple files, causing many interface changes as the different views recontextualize themselves to the changing files.

**P6:** Yes, I do find [the code paths traversed] difficult to remember. One of the things, Eclipse makes it so easy to go down the call path, that it's hard to keep track of it. Whereas if I'm jumping from method to method, you lose context of what class you're in, because all there is a title at the top that's flipping. […] It's not clear as to what path you're actually following. […] There's a broader view to it that you don't get just by jumping methods.

**E1:** Sometimes you go into method into another method, and I have no idea where you were coming from any more, you forgot, you lost track.

**P8:** I think if the navigation was not easy, I would more carefully remember these details.

There is rarely any visible connection as to how the developer came to the current file, and the developer must instead remember the connections or rebuild them, which can be difficult. Thus each file switch is perceptually independent from the previous: this is what is meant by low visual momentum. Eclipse's editor tabs are inadequate for re-orienting as they are not maintained in most recently used order.

**P8:** But when I'm lost, what I tend to do is go Alt-Back, back a row. “I'm lost, give me my context back! Where [am I]?” And after three or four things, if I'm not coming back on what I was expecting, I close everything.

**E2:** I'm just not interested in [using the editor tabs] because it's always a mess. […] I don't know what's going on in there in my mind, so I just don't use them.

These issues explain the behaviour of P5 and P6 observed during a long period of source exploration. Both would periodically stop to close unimportant files, ensuring their editor tabs contained only those files relevant to their task.

**Factor 2: Thrashing to view necessary context**

The second factor identified pertains to an inability to simultaneously view all of the information necessary for a task.
Participants complained about not being able to see more of the surrounding context of the source code being viewed, and P2,P3,E1 customized their Eclipse window layouts to increase available screen space for source code. But given the sizes of these large systems, developers are effectively limited examining the system as a whole through a limited keyhole view \cite{104}. As a result, developers must frequently switch between displays, maintaining relevant information in their working memory. P2,P4,P8 were observed to repeatedly scroll and jump both between and within files to correlate content from several windows or files. This frequent switching indicates low visual momentum.

**P2:** My biggest challenge is that I only have one view of the world, and it's because [Eclipse shows] one editor. And I can't see the different pieces of code, and it drives me crazy.

**P6:** I can't see enough information in files that I'm working on.

Although Eclipse provides some support for examining inter-file relationships without replacing the currently-displayed editor (functional overlays), such as showing the Javadoc for a called method in a tool tip, this was not always sufficient to meet their needs.

We use the term *thrashing* \cite{44} to refer to frequent and recurring switching. We assigned a subjective rating of the thrashing exhibited by each participant (Table 2.2), with *low* indicating little to no thrashing, *med* for some thrashing, and *high* for wild thrashing.\footnote{This subjective measure was assessed after counting the number of transitions between Eclipse editors and other applications, as well as the number of times where there was a total replacement of content. Our measure may overstate the thrashing as we did not incorporate a measure of time nor did we take into account the context to eliminate false positives. For example, some participants would process e-mail while waiting for a regression test pass to complete; although this would technically result in total or near-total replacement, it was not a genuine switch as the old information was not necessary to be carried forward. The switch to e-mail was still counted however.} Note that developers who exhibited high degrees of thrashing also reported or were observed being disoriented. The switching generally involved switching between files in Eclipse, or between Eclipse and either their problem reporting system or their e-mail application to record information. The thrashing generally occurred when they appeared to forget information to be copied across, and had to return to the original source.

One potential way to avoid thrashing is to use more than one window. P4 was
the only developer observed to use more than one Eclipse window, but his two windows were nearly completely overlapping and were used instead to manage different aspects of his work, rather than provide complementary information. This developer still exhibited a medium degree of thrashing as a result.

**Factor 3: Pursuit of digressions**

The third factor we identified involved the absence of support for managing developers' switches in work. These switches occur at a lower-level than the interruptions as noted by González and Mark [37].

We found developers often managing several tasks at any particular moment. Some tasks are suspended to pursue subtasks, called an *embedded digression* [30]. These digressions are usually related to the task: for example, attending a problem report may require some exploration to view the implications of a change. Some digressions may become more substantial or spawn further digressions: we observed P1 and P8 fixing or diagnosing bugs, and what were expected to be five minute jobs took more than an hour and a half.

Developers were observed to pursue embedded digressions without recording the task they had suspended. Because Eclipse had no knowledge of this digression, the developers had to remember to resume their suspended task, which is known to be unreliable [45]. If the environment had high visual momentum, the displays associated with a common task would be perceptually tied and displays related to other tasks would appear perceptually independent, such that a developer will be able to discern whether a particular display pertains to their current subtask or original task.

### 2.2.5 Strategies for Remaining Oriented

Our detailed examination also identified a number of strategies or work-arounds taken by developers to minimize disorientation, of which we discuss two here.

**Strategy 1: Using Environmental Supports to Remain On Task**

We observed developers using many of the non-obscuring information presentation techniques provided by Eclipse. These techniques, used for displaying information
Figure 2.1: Two examples of environmental supports provided by Eclipse: (a) a hover-style error descriptions presents relevant information in-place; (b) an editor outline bar, to the right of the scrollbar, locates relevant markers, such as errors or warnings, in relation to the file.

about inter- and intra-file relationships, remove the need to move away from the current position. From our observation and comments in interviews, these features were used and appreciated by developers. Three examples of such features are:

**Tool Tips:** All developers were observed to use Eclipse's tool tips in some manner, which are used to display helpful information about the source code, such as error text or auto-completion suggestions (e.g., Figure 2.1(a)). For functions that modify code, an additional tool tip window is used to show a preview of the change. These tool tips serve as functional overlays [104], providing supporting information without obscuring or replacing the code.

**Editor Outline Bar:** This bar provides a compressed overview of the file contents (e.g., Figure 2.1(b)), showing the locations of relevant markers such as errors, warnings, T000s, and breakpoints. With the appropriate option enabled, it will also show cross-references to a selected field within the file.

**Problems View:** This view presents a list of the outstanding errors, and is an example of Eclipse's dynamically-updated views. Most standard Eclipse views respond to changes as they occur, meaning that side-effects of changes are immediately available, supporting the principle of immediacy [98]. For example, saving a file triggers the incremental compilation, thus updating the problems view and outline bars with any syntax errors found.
**Strategy 2: Avoiding Unfamiliar Code**

Developers reported actively avoiding any need to understand new code. P5 was observed to look for examples rather than trying to understand the internals of the package itself. E1 and E2, both of whom used several external packages, commented that they both did the same. Developers avoid browsing code written by others to avoid having to learn new code, saving both time and effort.

This is reflected in the organization of the Eclipse development. The Eclipse developers (P1-P8) were part of groups where code ownership was a strong concept; a developer typically modifies only their code and not that of others. Developers are thus very familiar with their code, but less familiar with code of other groups or even their team-mates. This generally was not a problem as they would simply ask the responsible developer for an internal component (P1,P3,P4,P8).

By restricting themselves to their own code, or code they have become familiar with, developers are less likely to suffer disorientation (reported by P2,P3,E1):

**P3:** I’m intimate with the classes, [...] I wrote these classes, I’ve been all over them and I know what’s where and who’s where, so I know how to navigate to get to them.

This behaviour is consistent with Mantei’s finding that familiarity with the network aided in preventing disorientation [62].

**2.2.6 Study Validity**

A number of factors may contribute to disorientation, other than those involving the IDE, such as the experience of the developers, the familiarity with the IDE, the size and complexity of source code, the types of programming languages, and the rate of interruption. Although we could have tried to hypothesize factors and attempt to examine disorientation in a controlled laboratory setting, there is no theory on which to build these hypotheses, and furthermore no measures to use for assessing disorientation. We believe our study setup represents the worst case for these factors: if disorientation was purely related to a lack of familiarity with a codebase or lack of expertise, then we would expect disorientation to be minimal or non-existent for well-experienced developers developing familiar code bases using state-of-the-practice tools with which they were extremely comfortable.

Might the working practices of the developers contribute to possible disorienc-
2.3 Relating Disorientation and Exploration of Software

Although disorientation was not a frequent occurrence in the study, half of the developers observed became disoriented during the study and the majority of developers reported having become disoriented on a previous occasion.

In reflecting on the comments made by developers in our study and examining how they traversed through the source code of their software systems, we believe that developers often became disoriented when exploring to understand some part of their software system. In light of Sillito et al. [84]’s observation of a mismatch between the questions asked by developers (what we call conceptual queries) and the concrete queries provided by the tools they use, the factors we have identified reflect a deeper problem in making queries. In our study we witnessed developers performing additional processing of the results from a query (P3,P4), correlating the results from different queries and different tools (P2,P4,P6,P8), and frequently switching between editors and windows to view required information (P2,P4,P6). We believe this additional processing work was necessary as the information required to answer their conceptual queries was not immediately at hand, but instead scattered through different tools. The developers engaged in processing and correlating results from concrete queries provided by their tools as the concrete queries did not match the conceptual queries asked by developers. These are the mapping, scoping, and correlating burdens identified in Chapter 1. The developers frequently switched between editors and correlated queries from different tools because differences in the underlying models rendered it difficult to integrate the tools.

How can we begin to bridge this gap between the conceptual queries actually sought by developers and the concrete queries provided by software exploration tools? In the remainder of this dissertation we describe a model to support the integration of the results of different software exploration tools. We use this model to express a set of empirically identified conceptual queries. By addressing conceptual
queries actually posed by developers, we address the cognitive burdens of mapping, scoping, and composition that we identified in Chapter 1.
Chapter 3

The Sphere Model

In this chapter we present a simple model for representing and integrating different sources of program information. This model forms the underlying basis for our tool, Ferret, which we describe in Chapter 4. Each source of program information is represented as a sphere. This model is defined in terms of the binary relational algebra. Any source of program information can be described as a sphere providing that the source can be described in terms of relations between elements. Spheres can be composed in a principled manner using a set of composition functions. The expression of a conceptual query then corresponds to making concrete queries against a sphere, which may be a composite sphere.

Throughout the chapter we present some example bindings for the Java programming language.

3.1 The Binary Relational Algebra

We express the sphere model using the binary relational algebra. A binary relation is an association between two elements belonging to the same or different sets. The binary relational algebra provides a set of operators for manipulating a set of relations. We chose the binary relational algebra for two reasons. First, the relational model has been used in other software exploration tools [14, 47, 67], and the binary relational algebra is sufficient of representing the domain models of the sources of information that we have investigated. Second, the relational algebra is sufficiently
powerful to express interesting queries, and yet is simple enough to allow easy verification of our implementation.

We provide an overview of the binary relational algebra in Appendix A.

3.2 Spheres

A sphere is a logical collection of information about a software system. For instance, the static Java relationships represented in the source code forms a sphere, the revision history recorded in a software repository forms a sphere, and the calls information embodied in a particular dynamic execution trace forms a sphere.

More formally, a sphere $S$ is a tuple

$$S = (E, L, R)$$

where

- $E$ is a set of elements represented in the source of program information;
- $L$ is a set of relation labels describing the types of relationships represented that may exist between the elements;
- $R \subseteq L \times E \times E$ is a set of labelled relations. There may be multiple relations between two elements, but each is treated as a distinct relation with different labels.

It is important to note that the set of relation labels, $L$, must be logically consistent within a sphere. That is to say identically-labelled relations between elements of the same type should have the same meaning.

We project a labelled relation $\ell$ through a sphere from an element $e$ by selecting the relations with a corresponding label:

$$\pi_\ell(e) = \{e' \mid \ell \in L \land (\ell, e, e') \in R \}.$$
3.3 Representing a Software Analysis Tool as a Sphere

The concrete queries provided by a software analysis tool can be modelled by a sphere, providing the information available from the tool can be expressed in terms of relations between distinct elements.

Consider, for example, mapping the static structure maintained by the JDT about Java applications. In this mapping, the set of elements $V$ are formed from the individual methods, fields, classes, interfaces, and packages declared by the Java applications. The relation labels correspond to the types of Java semantic relations expressed in the source code such as implements and extends, declares, and calls. Finally, the relations correspond to the tuples that fulfill the semantics expressed by the labels. For example, in defining a sphere for the static relations as represented in the JDT, the relation $\text{CALLS}$ corresponds to the specification

$$\text{CALLS} = \{(x, y) \in V \times V \mid \text{method } x \text{ has a call to method } y\}.$$ 

Issuing a concrete query $Q$ from an element $e$ then corresponds to projecting a relation labelled $Q$ for a particular element, $\pi_Q(e)$.

Other sources of program information can be mapped to spheres as demonstrated in Table 3.1. For example, advanced static Java tools provide representations of classes, interfaces, methods, fields, and packages and support querying for the implementers of an interface, the subclass of a type, or the callers of a method. These particular sources were chosen as they represent often-used sources of information but whose best-of-breed implementations have clear gaps in the degree of integration between them. We are not aware of other approaches featuring improved integration.

There are some tools that are best represented by multiple spheres. Such tools generally require some user-provided context to function. For example, the results from a dynamic Java tool is highly dependent on the particular dynamic runtime traces that it has been configured to use. Although a set of traces may share common elements (e.g., they are from runs of the same program) and have the same relation labels, each trace will represent a different set of relations. Each trace should be represented by a separate dynamic runtime sphere. It is true that almost all tools depend on user-provided context to some degree, but there are many tools where
Table 3.1: Examples of some possible elements and relations in a Java-based software system

<table>
<thead>
<tr>
<th>Type</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Elements</td>
<td>static: classes, interfaces, methods, fields, packages</td>
</tr>
<tr>
<td></td>
<td>dynamic: classes, methods</td>
</tr>
<tr>
<td></td>
<td>evolution: programmers, files, elements, revisions, transactions</td>
</tr>
<tr>
<td></td>
<td>Eclipse plug-ins: extension points, extensions, plug-ins, identifiers, Java types</td>
</tr>
<tr>
<td>Relations</td>
<td>static: implements, extends, declares, instantiates, overrides, returns,</td>
</tr>
<tr>
<td></td>
<td>has-argument, throws, catches, tests-instanceof, casts, declared-type, gets, sets</td>
</tr>
<tr>
<td></td>
<td>dynamic: calls, instantiates, was-invoked, was-instantiated</td>
</tr>
<tr>
<td></td>
<td>evolution: modified-by, modified-in</td>
</tr>
<tr>
<td></td>
<td>Eclipse plug-ins: specifies, extends, depends</td>
</tr>
</tbody>
</table>

the user-provided context is unlikely to change over the lifetime of the tool, and are best represented as a single sphere.

3.4 Composing Spheres

One of the more powerful features of this model is its support for composing several spheres, such as using the results of one sphere to replace the information from another. As a simple example, consider a situation where a developer has been examining the source code for an interface and asks “which of the implementations of this interface were actually instantiated in this last run?” Composing the static information about the software system with the dynamic information from a run-time trace allows a tool to answer such a conceptual query. When the composition is over multiple dynamic spheres, this same query can operate over several scenarios of use of the program.

3.4.1 Composing Sphere Relations

There are several ways to compose spheres. We express the composition of sphere $S_1$ by $S_2$ using function $f$ as:

$$S_2 \circ f S_1 = (E_1 \cup E_2, L_1 \cup L_2, f(R_1, R_2))$$  \hspace{1cm} (3.1)
We treat the composition of the elements and labels as simple unions of sets as the two spheres may have common elements (i.e., the two spheres share a common representation, such as the many tools that use the JDT’s Java model), or the two spheres share common labels (i.e., the two spheres provide common operations, such as the static and dynamic spheres both describe CALLS relations between methods). This definition pushes the effect of composition to the function on the relations.

We have found use cases for three particular composition functions. Composition functions \( f \) are not required to be commutative. We provide simple illustrative examples of each function in terms of the static and dynamic spheres.

**Union**: Union includes all relations from both composing spheres.

\[
f(R_1, R_2) = R_1 \cup R_2
\]

We have used the union to combine dynamic profiles resulting from different runs of a program. Union is commutative.

**Replacement**: Replacement causes those relations of \( R_1 \) with a label from \( R_2 \) to be removed and replaced with the relations from \( R_2 \).

\[
X = \{(l, x, y) \in R_1 | l \notin L_2\}
\]

\[
f(R_1, R_2) = X \cup R_2
\]

We have used replacement to compose a static sphere with a dynamic sphere, thus replacing the CALLS and INSTANTIATES relations with what actually happened during the execution. Replacement is not commutative.

**Transformation**: Transformation performs a join of the relations of \( R_1 \) by a subset of \( R_2 \) for a particular label \( l_r \) in \( R_2 \).

\[
f(R_1, R_2, l_r) = \{(l, x, z) | (l, x, y) \in R_1 \land (l_r, y, z) \in R_2\}
\]

We have used transformation to apply a blanket restriction to all relations from \( R_1 \), so as to restrict the relations of a static Java sphere (\( R_1 \)) to those methods actually invoked in a dynamic trace (represented in \( R_2 \)). This was
accomplished by defining \( l_r = \text{WasInvoked} \) to be an identity relation where

\[(\text{WasInvoked}, x, x) \in R_2 \text{ if } x \text{ is either not a method, or } x \text{ is a method invoked in the trace.} \]

There are other possible composition functions, such as intersection, subtraction, and symmetric difference, but we have not found a use case to consider their inclusion. We discuss these further in Section 6.1.2.

### 3.4.2 Composing Sphere Elements

Our definition of sphere composition (Equation 3.1) defines elements of the sphere resulting from the composition as being the union of the sets of elements from the composing spheres. However there are occasions where a human developer would consider an element from one sphere to correspond to some otherwise seemingly different elements from the other spheres for the purpose of making a query. Consider the following situations:

1. A class \( C \) in the static source code should correspond to the occurrence of the class \( C' \) in the dynamic trace, and vice versa. Similarly a method \( m \) defined on \( C \) in the static source should correspond to the method \( m' \) on \( C' \) in the dynamic trace.

2. An interface \( I \) in the static source has no direct correspondence to any element in a dynamic trace resulting from execution of that program, but it may be considered to have a correspondence to all classes \( C' \) in the dynamic trace that correspond to any class \( C \) in the source that implements \( I \). Similarly a method \( m \) declared by interface \( I \) has no direct correspondence to any element in a dynamic trace, but it may be considered to have a correspondence to all methods \( m' \) of class \( C' \) where \( C'.m' \) is in the dynamic trace and corresponds to some method \( C.m \) in the source, and where \( C \) is a class implementing \( I \).

3. The body of a method as defined in a particular file revision as committed to a repository may have a correspondence to the representation for that method in the static source.
In each of these examples, the human developer is mentally converting from one particular representation to another, to treat one object from one representation as one or more equivalent objects in another representations.

We model this process of establishing correspondences using an adaptation of Wing and Ockerbloom’s notion of a respectful type converter [107]. They define:

A converter $C$ taking $A \rightarrow B$ respects type $T$ when both $A$ and $B$ are able to be viewed as instances of $T$.

$T$ represents a set of common properties or behaviours that must be preserved across the conversion. We relax and extend this definition in two ways. First, in their definition, $T$ is considered to be a common ancestor of $A$ and $B$. We remove this requirement: in our current sphere definition, the correspondences from $A$ to $B$ are often not reflected in the concrete representations. Second, we allow an instance from $A$ to correspond to one or more instances of $B$: an element from one sphere should be treated as a set of items from a different sphere if a programmer would consider them to be the same.

These three examples also demonstrate that correspondences may differ in at least two ways:

the valency of the correspondence: There may be no single best match for an element: an element in one sphere may correspond to multiple elements in another sphere. This is illustrated with the conversion of a Java interface to its implementing classes in the dynamic run-time.

the fidelity of the correspondence: Some correspondences are more tenuous than others. Correspondences differ by the degree of information preserved in the conversion. We capture this degree of information preservation or information loss in a conversion using an ordinal measure we call the fidelity. We discuss the use of fidelity in the next section.

Correspondence is not required to be symmetric, as illustrated by the following example. A Java interface may correspond to its implementing classes in a dynamic execution trace, however each of those implementing classes will correspond to their matching class in the static source, and not to the original interface.
Modelling the Effect of Correspondences

Correspondence is modelled as a function on the relations defined by a sphere. Imagine relations \( R_i \subseteq L_i \times E_i \times E_i \) as a relational matrix \( M_i = E_i \times E_i \) with each matrix entry holding the applicable set of labels. We construct the corresponding matrix for the composite sphere, \( M = E \times E \) where \( E = E_1 \cup E_2 \), as follows:

\[
M = \begin{pmatrix} M_1 & M_{1 \rightarrow 2} \\ M_{2 \rightarrow 1} & M_2 \end{pmatrix}
\]

We construct \( M_{1 \rightarrow 2} = E_1 \times E_2 \) by considering the three possible ways in how a converter \( C : E_1 \rightarrow E_2 \) maps an input to outputs. Each row of the matrix \( M_{1 \rightarrow 2} \) corresponds to row merge or duplication operations on the relational matrix \( M_2 \).

1 \( \rightarrow \) 1: element \( e \in E_1 \) corresponds to a unique element \( e' \in E_2 \): the row in \( M_{1 \rightarrow 2} \) for \( e \) is formed from the row in \( M_2 \) for \( e' \).

1 \( \rightarrow \) n: element \( e \in E_1 \) corresponds to several elements \( e'_i \in E_2 \): the row in \( M_{1 \rightarrow 2} \) for \( e \) is formed by the union of the the rows in \( M_2 \) for \( e'_i \).

1 \( \rightarrow \) 0: element \( e \in E_1 \) corresponds to no elements in \( E_2 \): the row in \( M_{1 \rightarrow 2} \) for \( e \) is empty.

The matrix \( M_{2 \rightarrow 1} \) is constructed following the same process.

3.4.3 Tracking Fidelity

We described fidelity as an ordinal measure describing the degree of information preservation or information loss from a conversion (Section 3.4.2). But although a converted element may not have a perfect correspondence to its source, it can still be used as a source for inquiry.

We currently describe fidelity using three levels: EXACT, EQUIVALENT, and APPROXIMATE.

**EXACT** indicates a conversion where there is a perfect one-to-one and onto correspondence. For example, a method defined on a class for a static Java tool will correspond exactly to the same method from a dynamic runtime tool, assuming the dynamic run-time was taken from the same source base.
**EQUIVALENT** indicates a very near match: although there is not a perfect conversion, the elements are felt to be equivalent from a programmer’s perspective. For example, a Java interface will have an equivalent correspondence to its implementing classes in the dynamic run-time, providing the trace was obtained from the same source code.

**APPROXIMATE** indicates an imperfect match, where there is a similarity, but some differences. For example, a method from a particular version of a file as committed to a repository will have only an approximate correspondence to the representation of the method in the source code being edited.

To help illustrate the differences between these levels, consider an example where a programmer is using Ferret over a union of the static Java and evolution spheres (Table 3.1). We have a Java class \( t \) that is instantiated in methods \( m_1, m_2 \) and \( m_3 \). We examine the source for an earlier edition of \( t \), an evolution element \( s \), which we consider to have an **APPROXIMATE** correspondence to \( t \). Now consider projecting the relation **INSTANTIATORS from** \( s \). Although the evolution sphere has no such relation, the conversion rules from Section 3.4.2 cause the projection to be the results of projecting from \( t \), resulting in the methods \( m_1, m_2 \) and \( m_3 \). But because they were determined from \( t \), which has an **APPROXIMATE** correspondence to \( s \), then the resulting methods must have fidelity of **APPROXIMATE** at best. To have a different fidelity would mean that the results of a relation could be more certain than the input.

As it may be useful for a relation to involve some speculation, we cause relations to have an associated fidelity too. The general principle is that fidelity of the results of projecting a relation should be the less of the fidelity of the input to the relation and the fidelity of the relation itself. As a conceptual query may require projecting a sequence of relations, then the fidelity should be monotonically decreasing over the course of the query.

Thus the fidelity provides an assessment of the soundness of a query’s results. Our use of fidelity serves a similar purpose to Balzer’s pollution markers [2], where computation results that fail to meet specified constraints are marked as being polluted. The markers are propagated through any subsequent computations that use a polluted result, indicating that the computation is polluted as well. We should note
that the fidelity only annotates the results of a query, and does not affect the query computation.

3.5 Expressing Conceptual Queries

Using this model, conceptual queries are expressed as sequences of projections and compositions on the labelled relations defined by a sphere. For example, finding the instantiators of an interface requires composing the results of two concrete queries, first to find the implementers of the interface and then to find the instantiators of those interfaces. A conceptual query may also identify a subset of the results of a projected relation, such as identifying a subset of elements with some particular property. Our adaptation-related conceptual queries for Eclipse Plug-in Development Environment (PDE) (see Appendix B) must filter extensions that do not match the criteria required of the query.

3.6 Summary

The sphere model provides a principled means for composing different perspectives, such as represented by tools. This is done in two ways: (i) by supporting correspondences between elements from different perspectives that recognizes there is not always a single one-to-one correspondence between elements; and (ii) by providing means to integrate and rationalize the relational information provided by different perspectives.

The sphere model is reducible to any query language based on the relational algebra. The key difference in the Ferret model and other query languages lies in the creation of the underlying knowledge-base upon which queries are executed. The sphere resulting from compositions is itself a sphere with a set of elements and a set of labelled relations between those elements.

The sphere model is also language independent. Our use of the binary relational algebra does not depend on details of specific programming languages or tools. As such we expect that the model should be capable of representing tools for other languages providing those tools can be modelled as a set of elements and relations on those elements.
Chapter 4

The Ferret Tool

We have built a proof-of-concept tool, named Ferret, that implements the sphere model. By building Ferret, we can show that the sphere model can express sufficiently rich conceptual queries, demonstrate that the conceptual queries can be implemented efficiently, and enable experimentation with software developers to determine which conceptual queries are useful.

Ferret currently implements 36 conceptual queries across four spheres. These conceptual queries are used to summarize how a particular program element of interest is related to the rest of the software system. This summary is displayed in a single view to the Eclipse IDE.

We begin in Section 4.1 by describing what it is like to use Ferret. We then describe the implementation of the model and of conceptual queries in Section 4.2. We describe the steps involved in extending Ferret to new tools in Section 4.3.

4.1 Using Ferret

Ferret is integrated into the Eclipse IDE as a single view, as shown in Figure 4.1. We first describe how to request a query from Ferret (Section 4.1.1) and describe Ferret’s supports for examining the results of those queries (Section 4.1.2). We then describe how to configure Ferret prior to its use (Section 4.1.3).
4.1.1 Requesting Queries

Ferret's basic user interface is quite simple: when a developer wishes to know more about some program element, he invokes Ferret, using one of several mechanisms described later in this chapter. When invoked, Ferret computes all 36 configured conceptual queries for the particular element (Table 4.2, p. 58), and the results of the various conceptual queries are displayed as a structured tree view. Figure 4.1, for example, shows the results of having invoked Ferret on an interface MCallEvent. The answers to several conceptual queries are displayed, such as the which classes provide implementations of this interface, which methods instantiate the interface, and which methods are declared to return instances of this type.

Ferret can be invoked either explicitly or implicitly. Ferret is invoked explicitly either through a keyboard shortcut or from a program element’s context menu. Ferret also treats the programmer's selection of program elements in the IDE (except for selections within the Ferret view) as an implicit invocation of Ferret’s queries; this style of implicit invocation can be disabled. The support for implicit invocation supports an active querying interaction style for providing information about potential elements of interest as indicated by being selected by the developer. Some conceptual queries can operate with multiple inputs and correspond to the questions of the form “what common [feature] is supported by these [elements]?”; these queries are indicated in Table 4.2 by an asterisk. The conceptual queries are processed in a background thread, ensuring the UI remains responsive, and the results of the queries are filled in as each query is completed. A new invocation will interrupt any other invocation is progress. Although all configured conceptual queries are performed on an
Table 4.1: Assigned categories for Ferret’s conceptual queries.

<table>
<thead>
<tr>
<th>Category</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>inter-class</td>
<td>Queries about how elements are used by other elements within the system.</td>
</tr>
<tr>
<td>intra-class</td>
<td>Queries about the internal functioning, or behaviour, of an element.</td>
</tr>
<tr>
<td>inheritance</td>
<td>Queries about an element’s relation to others in a type hierarchy.</td>
</tr>
<tr>
<td>declarations</td>
<td>Queries about the structure or form of the element.</td>
</tr>
<tr>
<td>evolution</td>
<td>Queries about the evolution history of the element.</td>
</tr>
</tbody>
</table>

invocation, some are likely incompatible with the provided input; this is discussed further in Section 4.2.2.

The conceptual queries are grouped by an assigned category (Table 4.1). These categories were assigned based on the type of information provided by the query. Each conceptual query is represented as a branch in the displayed tree; the children of a given node represent the results of the conceptual query. The conceptual queries shown in Figure 4.1 are all collapsed, as compared to the conceptual query in Figure 4.3 which is expanded and showing the results of a particular query.

For each conceptual query, Ferret displays the query category, the number of results for the query, and the number of possible clusterings of the results. When displaying the results of the query, Ferret also informs the programmer of the fidelity of the results. Ferret by default only displays queries that successfully returned a result: queries that are not applicable for the particular element from which Ferret is triggered, queries that use concrete queries that cannot be resolved, and queries with no results, are not displayed.

4.1.2 Investigating Query Results

Ferret provides a number of features to help developers investigate the elements resulting from an invocation. Double-clicking on an element opens the corresponding artefact containing that element for viewing, such as the file containing a method.
Figure 4.2: The context menu for a JDT query result in the Ferret view. The items above the separating line are Ferret items; the items below the separating line are artefact-specific items.

Right-clicking on an element pops up a context menu, as shown in Figure 4.2. From this menu, the developer can open the element or its parent (or containing element), issue a new Ferret invocation from the element, in addition to other operations specific to the element; these element-specific operations are provided by the other tools in Eclipse. A programmer may manually remove any result from the view should he deem the result to be uninteresting.

The view’s toolbar also contains a number of features to keep the developer oriented and to help manage the results list (Figure 4.1). At the far left is a description of the current query element(s). The remainder of the toolbar provides a set of buttons. From left to right, the pinning button (), temporarily disables Ferret’s implicit invocation, thus preventing the current contents displayed in the view from being changed. The plus button (), expands all queries by one level, and the minus button (), collapses all queries. The house button (), opens the artefact containing the source input(s) of the current query, to help the developer in returning from a digression. The house button’s drop-down menu allows the developer to select from the source input(s) from previous Ferret queries. Upon selecting an input, Ferret reopens the artefact containing the source input(s) and re-invokes the conceptual queries for the source input(s). The final button, showing a hierarchy (), opens the current query element’s parent or containing element.

As a conceptual query may have many results, Ferret supports grouping the re-
results using secondary relationships between the results. Ferret currently provides means for clustering according to the attributes of the results. Figure 4.3, for example, shows a number of possible clusterings for a set of Java methods, such as by declaring package or by the types of the arguments. Other attributes are supported, such as using Eclipse's naming convention to distinguish public application programming interface (API) elements from internal elements. Clustered groups may themselves be further clustered using other attributes, so that a developer can drill down into the results. A particular conceptual query may provide additional clusterings based on properties calculated during the query. For example, the query “what methods instantiate this type?”, that includes the results of queries on implementors of an interface, adds an additional clustering for the type instantiated. The clusters are not necessarily disjoint; for example, when clustering using the types of the arguments of a method, a method with multiple arguments will appear in a cluster for each type of which it has an argument.

As a developer may wish to pursue a minor digression to find out additional information about a result, Ferret supports cascading a new round of conceptual queries for a query result. These new conceptual queries are opened in-place as children of the query result, thus preserving the context of the original query. This ability is similar to that offered in JQuery [49].

Finally, Eclipse maintains a navigation history for each workbench window,
recording the user’s transitions between editors. This feature is similar to a web-browser’s history of document navigation, with analogous backward and forward movement operations. Investigating a set of query results typically causes each result to be added to the Eclipse navigation history as a sequence of navigations. To better support short digressions, Ferret has an option to instead restore the state of Eclipse’s web-style backward/forward navigation history upon opening a different query result: thus moving backwards along the navigation history will commence relative to the source of the query, rather than trail through the results viewed.

4.1.3 Configuring Ferret

The conceptual queries are computed against a developer-configured composite sphere. The composition is configured by the developer through the Ferret Sphere Configuration Wizard, as shown in Figure 4.4. This wizard supports arranging a composition from a set of composition functions and spheres through a point and click interface. The composition functions offered correspond to those functions defined in Section 3.4.1. Configuring a composite sphere is only necessary on the first invocation of Ferret; the configuration remains in effect until Ferret is closed or the user chooses to reconfigure the composite sphere.

Ferret provides four different spheres corresponding to four sources of program information: static Java, dynamic Java, software evolution, and Eclipse plugin information (see Table 3.1). Figure 4.4 includes a fifth sphere labelled ‘Static Java restricted to actually-executed methods (JDT/TPTP)’ which is itself a composite sphere that is described in Section 4.2.5. The spheres may prompt the user for additional configuration information if necessary; for instance, the selection of the dynamic sphere prompts for the selection of the appropriate trace data to use.

4.2 Implementation of the Model

The Ferret tool is a faithful implementation of the model described in Chapter 3 and instantiated as a proof-of-concept in the Eclipse development environment. At a high level, a conceptual query is issued in the context of one or more elements of a program, such as a method declaration, and the results of the query are computed using a set of relations that are resolved by name through a sphere.
Figure 4.4: The Ferret sphere configuration wizard, used to compose one or more spheres.

4.2.1 Spheres

Spheres are responsible for binding relation implementations to relation names. Ferret currently implements four spheres corresponding to the four sources: static, dynamic, software evolution, and PDE (see Table 3.1). The relation implementations are implemented using APIs provided by the sources. The static Java sphere is implemented using the public Core and Search APIs provided by the JDT. The dynamic Java run-time sphere is implemented using the profile querying facilities of the TPTP. The software evolution sphere is implemented using facilities from Kenyon [7], a source code repository analysis framework supporting systems such as CVS [6], amongst others. The plug-in development sphere is implemented using the internal representation model from the PDE.

The relation names form a flat namespace to allow spheres to provide different implementations for a particular relation name. The relation name captures the semantics of the relation. For example, the dynamic Java sphere provides some re-
lations with common semantics to those provided by the static Java sphere and are thus configured with the same label.

### 4.2.2 Conceptual Queries

Ferret currently implements 36 conceptual queries, which are listed in Table 4.2. The queries we have implemented were identified from the literature, blogs, or from our own experiences developing Ferret; the motivations for queries is tracked in Table 4.2. Some of these queries have analogs to existing queries but are augmented in their ability to use other sources of program information. Other queries, such as the evolution queries for relating source changes to code, do not have direct analogs in any existing tool (marked with a dagger in Table 4.2). In determining whether a conceptual query is unique to Ferret, we specifically exclude a tool should that tool require additional mapping or scoping of its results to answer a conceptual query. Some conceptual queries are included to serve as a demonstration of cross-artefact queries. For example, the \( \text{PDE} \)-related queries link declarative information specified in Eclipse plug-in manifests to static source elements.

A conceptual query is specified in terms of relational operators over relation names. The relations named in the definition of a conceptual query are resolved over a composite sphere that serves as a knowledge-base for the program. For example, the query “what instantiates this type?” can be expressed as the formula:

\[
\text{IMPLEMENTORS} \circ \text{INSTANTIATORS}
\]

The relation IMPLEMENTORS implicitly takes the given input, likely to be a Java type, and returns all types that implement the input; the relation INSTANTIATORS returns all instantiators of a given input. These relations are resolved using the developer-configured composite sphere. The definitions used for the Ferret's conceptual queries are provided in Appendix B.

If a relation cannot be resolved, then the conceptual query silently fails. In this respect, conceptual queries implement a form of dynamic typing: if all the relations are resolved, then an answer can be computed for the query; if some relation cannot be resolved, then the query was not applicable for its inputs. This approach to implementing the model isolates conceptual queries from the sources of program information used to compute the results of the query.
4.2.3 Composing Spheres and Resolving Relations

Conceptual queries are executed in the context of a sphere, which is used to resolve the relations named in the conceptual query’s definition. The sphere may be a composite sphere specified using the composition functions defined in Section 3.4.1. The composition functions determine the order in which spheres are consulted for possible candidate relation implementations. The functions union, and transformation cause all spheres to be resolved and to perform some function on the combined results. The function replacement selects an individual sphere.

Relations are always resolved in the context of some input from which the relation is to be projected. This input comes from a selection from some information displayed by a tool that is represented by one of the spheres. A relation implementation is considered resolved only after verifying that it supports the input. For example, the static Java sphere’s implementation of IMPLEMENTORS requires its input to be a JDT type instance (an ITyp e). This support generally entails that the inputs can be converted to an instance of some particular type, and that possible correspondences of the inputs meet a minimum required fidelity. If the input cannot be converted, then the relation implementation is rejected and resolution continues.

For example, both the static Java sphere and the dynamic Java sphere provide implementations of a CALLS relation. Should these spheres be composed using the union composition function, then both spheres will be consulted for a CALLS relation. Should they be composed using replacement, each sphere is consulted in turn until one successfully resolves the relation. The former case is useful for capturing all possible calls, including calls that are implicit in the code, such as calls via the Java reflection mechanism. The latter case is useful if only the calls actually made during execution are of interest. We provide an example of resolving a relation in Section 4.2.5.

4.2.4 Supporting Correspondences

Key to the relation resolving process is the identification of correspondences for inputs. For example, the implementation of the IMPLEMENTORS relation in the static Java sphere using the JDT requires its input to be a JDT representation of a Java type, an ITyp e. The resolving process will attempt to find any ITyp es that correspond to its
Correspondences are essential when a sphere is used to represent a tool that does not have a proper UI to expose the tool elements. For example, Kenyon is a framework without a UI; through the use of correspondences, JDT elements are able to trigger Kenyon-related conceptual queries.

Correspondences are implemented in Ferret using a mediating broker architecture [107]. Each broker specifies the source and destination types supported, the maximum possible fidelity of the correspondence, and the maximum possible valency of its correspondence. To reduce the number of brokers necessary, the brokers are assembled into a graph. A correspondence may result in a path with several intermediate correspondences. As there may be multiple candidate paths for converting a source type to some desired target type, we select the best path by (i) considering only paths where the intermediate correspondence steps are univalent (i.e., where there is exactly one corresponding instance) so as to avoid any possible ballooning in the results, and (ii) weighting each path by its total fidelity. The fidelity of the resulting correspondence is the minimum fidelity of the correspondence on the path. Where there are multiple paths that are equally good, we simply choose one of the paths. A correspondence once made is cached for the source instance.

The currently defined correspondence graph is shown in Figure 4.5, where the types are grouped by related purposes. Each edge defines a correspondence, and is implemented by some broker. This graph is discussed further in Section 4.3.4.

4.2.5 Example of Resolving a Relation

To illustrate the implementation of the model, we trace through the resolution of the 

\texttt{returnType} relation when provided a particular \texttt{JDT} interface \texttt{(type \texttt{IType})}. This relation matches a type to the methods declared as returning an object of that type. We use the composite sphere corresponding to the following definition; we often use this definition when trying to understand some particular functionality of a software system.

```
replacement[
  dynamic---java---sphere(tptp---trace---file),
  transformation[
```
Figure 4.5: The correspondence broker network in Ferret. Lines with arrows indicate possible correspondences; red enclosures indicate groupings of related elements.

static −java−sphere,
dynamic−java−sphere(tptp−trace−file),
"WasInvoked"]

This particular sphere composition uses the replacement function at the top level. Thus function causes RETURNS_TYPE to be resolved against its sub-spheres in succession until one is successful. Since a dynamic Java sphere does not provide an RETURNS_TYPE, the replacement causes resolution to proceed to the next sphere.

The next sphere is a composite sphere obtained using the transformation function. The transformation function first resolves RETURNS_TYPE against the static Java sphere. This resolution is successful as the sphere provides an implementation for RETURNS_TYPE, and the implementation expects a JDT type instance (type IType) as input. The transformation function then composes the results from projecting RETURNS_TYPE to another relation, WASINVOKED, which is projected against the dynamic Java sphere.

The dynamic Java sphere provides an implementation of WASINVOKED. WASINVOKED is a filter that removes methods that were never called. Its function is described as follows:
• if its input is a \texttt{TPTP} method, or can be converted to a \texttt{TPTP} method:
  
  – if the method was actually recorded as being invoked in the provided
dynamic execution trace, then return the input,
  
  – else the input was not sent, so return nothing;

• else the input is not a method, so return the input.

The implementation of \texttt{WasInvoked} must attempt to convert its input to native
\texttt{TPTP} methods (of type \texttt{TRCMethod}). The correspondence network, shown in Figure 4.5, supports converting \texttt{JDT} methods (type \texttt{IMethod}) to the corresponding native \texttt{TPTP} method (\texttt{TRCMethod}). Methods defined on interfaces, which have no executable components, are converted to the \texttt{TPTP} methods corresponding to the class methods that implement the interface methods.\footnote{Thus each of the methods are checked for \texttt{WasInvoked}, which effectively filters the methods by those actually sent. These filtered methods are the result of the transformed \texttt{RETURNSTYPE} relation and, since the \texttt{RETURNSTYPE} was successfully resolved, the replacement returns the type-returning methods that were actually sent as the overall results of the projection of \texttt{RETURNSTYPE}.}

Thus with this composite sphere, projecting \texttt{RETURNSTYPE} from a particular
interface \(t\) results in the methods declared to return type \(t\) that were actually invoked
as recorded in the given dynamic trace file. Performing this query without Ferret
would require correlating a list of methods from a \texttt{JDT} view using linear searches
through a \texttt{TPTP} view.

\subsection{Extending Ferret}

The Ferret tool is designed to be extensible. Conceptual queries, spheres, sphere
composition functions, type correspondences, clusterings, and attribute classifiers
are all configured through a set of Eclipse extension points \cite{extpoints}. Extension points
provide a mechanism for decoupling the code using some functionality from the
implementations.

\footnote{In this dissertation, we use ‘class methods’ to refer to the methods defined by a class (i.e., that have
an implementation) and ‘interface methods’ to refer to the methods defined by an interface.}
The relation amongst the classes implementing Ferret’s functionality is shown in Figure 4.6. All classes are defined in the Java package ca.ubc.cs.ferret.model.

4.3.1 Adding a Sphere

Adding a new sphere requires (i) identifying the types of elements described by the sphere, (ii) identifying the relations to be exposed by the sphere, and (iii) identifying correspondences to elements of other spheres. For the identified elements, we may need to add correspondence brokers to establish any correspondences between elements of the new sphere and those of other spheres. The identified relations must be reconciled against the relation names provided by other spheres.

Spheres implement the Java type ISphere. Spheres have two responsibilities: (i) to resolve relations given a relation name and some input, and (ii) to hold any state required by the sphere (e.g., the dynamic runtime trace for TPTP). Spheres have an associated helper class, a SphereHelper, that provides helper functions for handling UI issues with respect to elements of the sphere, such as pretty-printing, interpreting textual specifications of elements (e.g., a type name from a text file), and to open an appropriate editor for an element. The sphere helpers are also responsible for providing the spheres for the configuration wizard (Section 4.1.3). The sphere helpers are specified through an the extension point ca.ubc.cs.ferret.sphereHelpers.
4.3.2 Adding a Conceptual Query

Conceptual queries are implemented by Java classes fulfilling the IConceptualQuery interface and are specified through the ca.ubc.cs.ferret.cqs extension point. Conceptual queries are implemented by requesting relations by name through a configured sphere; the resolution process is described in Section 4.2.3.

Conceptual queries return their results using a frame-like data structure \([10]\) called a solution. Our implementations have generally only had a single element per solution frame. Some pde-related queries, such as the adaptable to/from queries shown in Figure 4.7, use the frame to include additional details that form part of an answer to the query. These details are presented as children of the primary solution element in the results tree, but prefixed with connecting text (“factory” and “specified by” in the example).

4.3.3 Adding a Sphere Composition Function

Sphere composition functions (Section 3.4.1) are implemented by Java classes fulfilling the ISphereCompositor interface, and are specified through the ca.ubc.cs.ferret.scfs extension point. These compositors implement a mechanism to affect how relations are resolved across its child spheres. Although the description of composition functions in the sphere model are couched in immediate terms (Section 3.4.1), their implementation performs composition on demand. As such, sphere composition functions implement ISphere and behave like spheres.
4.3.4 Adding a Possible Correspondence

The actual establishment of a correspondence is performed by a correspondence broker. A correspondence broker, called a type converter internally, is implemented by a Java class fulfilling the ca.ubc.cs.ferret.types.ITypeConverter interface, and is specified through the ca.ubc.cs.ferret.typeConverters extension point. A type converter declaration must specify the source type, the destination type, the best fidelity possible, and the broker class. The broker is responsible for taking an instance of the source type and finding its corresponding instance (or instances) of the destination type. If no corresponding instance(s) can be found, then it signals so and the correspondence framework will continue to seek another correspondence.

In performing a correspondence, we walk the class hierarchy chain of the source type in order to find the most specific correspondence to the desired type. For each class in the hierarchy chain, we first look for a correspondence to the desired type, and then consider the interfaces the class implements. If no correspondence is found we proceed to the class’ superclass. The desired type is considered absolute: we do not consider its hierarchy to avoid issues of selecting from the best of several possible correspondences.

The set of currently configured correspondences is shown in Figure 4.5. The types in this graph are grouped into enclosures to indicate types with a related purpose. In our implementation of Ferret, we have created two classes of correspondences. The first class of correspondence are those that handle correspondences of elements between different spheres. The JDT ⇔ TPTP and Kenyon ⇔ JDT correspondences serve as examples of this kind.

The second class of correspondences exist to convert types that are conceptually part of one of the provided spheres into a canonical form. For example, the java.reflect and JDT/Debug enclosures represent live Java instances from a runtime, such as in a stack frame of the debugger. These instances have a likely correspondence to types defined in the static Java sphere. The correspondences within the PDE enclosure exist for a similar reason: the PDE implementation maintains several different representations of the same object within different UI elements. These canonical correspondences allow Ferret to be invoked from different views in the UI.
4.4 Summary

In this chapter we have described our proof-of-concept tool, Ferret. Ferret implements a wide range of conceptual queries and supports four different spheres. Together these demonstrate that the sphere model can express a rich set of conceptual queries. We have described the UI features provided by Ferret for performing queries and described how Ferret can be extended to integrate new tools.
Table 4.2: Conceptual queries currently defined for Ferret. ‘✔’ indicates spheres providing relations used in the conceptual query. ‘*’ indicates queries supporting plural inputs; ‘†’ indicates queries that are not available in current tools and ‘‡’ indicates queries that have partial support in current tools.

<table>
<thead>
<tr>
<th>Category / Conceptual Queries</th>
<th>Contributing Spheres</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Static</td>
</tr>
<tr>
<td><strong>inter-class</strong></td>
<td></td>
</tr>
<tr>
<td>1 †What methods return instances of this type? [46]</td>
<td>✔</td>
</tr>
<tr>
<td>2 *What methods instantiate this type? [84]</td>
<td>✔</td>
</tr>
<tr>
<td>3  *[What methods take an argument of this type? [84]</td>
<td>✔</td>
</tr>
<tr>
<td>4  What fields are declared as being of this type?</td>
<td>✔</td>
</tr>
<tr>
<td>5  *What calls this method? [84]</td>
<td>✔</td>
</tr>
<tr>
<td>6  *Where is the value of this field retrieved? [84]</td>
<td>✔</td>
</tr>
<tr>
<td>7  *Where is the value of this field set? [84]</td>
<td>✔</td>
</tr>
<tr>
<td>8  *What tests instanceof against this type? [49]</td>
<td>✔</td>
</tr>
<tr>
<td>9  *Where are casts to this type? [49]</td>
<td>✔</td>
</tr>
<tr>
<td>10  *What throws exceptions of this type? [84]</td>
<td>✔</td>
</tr>
<tr>
<td>11  *What catches exceptions of this type?</td>
<td>✔</td>
</tr>
<tr>
<td>12  *What references this type by name? [84]</td>
<td>✔</td>
</tr>
<tr>
<td>13  *Which of the classes in this package were actually used?</td>
<td>✔</td>
</tr>
<tr>
<td>14  *Which methods defined by this class were actually used?</td>
<td>✔</td>
</tr>
<tr>
<td><strong>intra-class</strong></td>
<td></td>
</tr>
<tr>
<td>15 †∗What fields does this type or method access? [84]</td>
<td>✔</td>
</tr>
<tr>
<td>16  †∗What methods does this type or method call? [84]</td>
<td>✔</td>
</tr>
<tr>
<td>17  †∗What types does this {type, method, extension} reference?</td>
<td>✔</td>
</tr>
<tr>
<td><strong>inheritance</strong></td>
<td></td>
</tr>
<tr>
<td>18  *What interfaces does this type implement? [84]</td>
<td>✔</td>
</tr>
<tr>
<td>19   What are this class’ subclasses? [84]</td>
<td>✔</td>
</tr>
<tr>
<td>20  *What classes implement this interface? [84]</td>
<td>✔</td>
</tr>
<tr>
<td>21  *What interfaces extend this interface? [84]</td>
<td>✔</td>
</tr>
<tr>
<td>22 †What are this type’s siblings? [84]</td>
<td>✔</td>
</tr>
<tr>
<td><strong>declaration</strong></td>
<td></td>
</tr>
<tr>
<td>23  What are all the fields declared by this type? [84]</td>
<td>✔</td>
</tr>
<tr>
<td>24  What class methods implement this interface method? [84]</td>
<td>✔</td>
</tr>
</tbody>
</table>

(table continues)
### Table 4.2: (continued)

<table>
<thead>
<tr>
<th>Category / Conceptual Queries</th>
<th>Contributing Spheres</th>
</tr>
</thead>
<tbody>
<tr>
<td>25   *What interface methods specify this class method? [84]</td>
<td>✔         ✔</td>
</tr>
<tr>
<td>26   What class methods does this method override?</td>
<td>✔         ✔</td>
</tr>
<tr>
<td>27   What class methods override this method?</td>
<td>✔         ✔</td>
</tr>
<tr>
<td>28   ‡What extension points or extensions reference this {type, file, folder}?</td>
<td>✔         ✔</td>
</tr>
<tr>
<td>29   †What types is this type adaptable to?</td>
<td>✔         ✔</td>
</tr>
<tr>
<td>30   †What types could this type have been adapted from?</td>
<td>✔         ✔</td>
</tr>
<tr>
<td>31   What extensions are there of this extension point?</td>
<td>✔         ✔</td>
</tr>
<tr>
<td>32   What plug-ins depend on this plug-in?</td>
<td>✔         ✔</td>
</tr>
<tr>
<td>evolution</td>
<td></td>
</tr>
<tr>
<td>33   ‡What transactions changed this element? [74]</td>
<td>✔</td>
</tr>
<tr>
<td>34   ‡Who has changed this element, and when? [74, 99]</td>
<td>✔</td>
</tr>
<tr>
<td>35   ‡What elements were changed in this transaction? [74]</td>
<td>✔</td>
</tr>
<tr>
<td>36   †What files were changed in this transaction? [92]</td>
<td>✔</td>
</tr>
</tbody>
</table>

*Transaction is used to signify a changeset.
Chapter 5

Evaluation

This dissertation has introduced a model to represent and compose different sources of information about a program. The thesis of this dissertation is that this model broadens the set of possible conceptual queries that can be answered using software tools. Substantiating our thesis requires addressing four claims: (i) demonstrating that our model broadens the set of conceptual queries answerable by software exploration tools, (ii) demonstrating that the conceptual queries that we have implemented are relevant to developers, (iii) demonstrating that providing direct support for conceptual queries has a positive impact on the effectiveness of software developers at work, and (iv) demonstrating that the sphere model and conceptual queries can be efficiently implemented. We support the first claim through the implementation of a wide range of conceptual queries in Ferret. We provide support for the second claim from a diary study where four industrial developers used Ferret over the course of several days to perform their queries. We attempted to test the third and final claim through a formal experiment to compare the impact on developer effectiveness from three software exploration tools when compared to using JDT alone. We were unable to test this claim as the results of this study were dominated by task effects and by the strategies employed by the individual developers. The results of the study generally cast doubt on the possibility of quantitative comparisons of the effectiveness of software exploration tools. We support the fourth claim by describing the results of several benchmarks of Ferret on real projects to show that is possible to implement queries efficiently using the sphere model.
5.1 Claim: Expressiveness of the model

Claim 1: The sphere model broadens the set of conceptual queries answerable by tools.

We argue the expressiveness of the sphere model by first showing that the model is sufficiently expressive to support a wide range of conceptual queries.

Our proof-of-concept tool implements 36 conceptual queries and integrates four different spheres. These queries are listed in Table 4.2 (p. 58) which relates each conceptual query to its motivating source from the literature, blogs, or discussions. A number of these queries benefit from being able to integrate different spheres.

A number of these queries are unique to Ferret (indicated by a ‘†’ in Table 4.2). Other queries make use of information that has not previously been made available in an accessible form. For example, Ferret’s evolution-related queries are able to relate source changes by programming language-specific structural information.

5.2 Claim: Relevance

Claim 2: Conceptual queries are relevant to developers.

Having implemented 36 queries requested from a variety of sources, we now turn to the question of whether these conceptual queries are of use to developers during their day-to-day work. More specifically, we sought to discover:

- Which conceptual queries implemented in Ferret are used by developers?
- Is the composition of static and dynamic program information, which have some overlap in their concrete queries, useful?
- Are there features of Ferret that the developers find particularly useful?

To answer these questions we undertook an event-based diary study [9]. We recruited four professional software developers from a large company to use Ferret during their day-to-day programming activities. These activities were on familiar codebases. The developers were equipped with a version of Ferret with the JDT, PDE,
and TPTP spheres; we were unable to use the evolution-related queries as our repository interface library software was not compatible with the repository software in use at the company. We provided a 20-minute tutorial on configuring and using Ferret. The developers were encouraged to use the following sphere configuration when combining static and dynamic program information (similar to the sphere configuration in Section 4.2.5):

```
replacement[
  pde—sphere,
  dynamic—java—sphere(tptp—trace—file),
transformation[
  static —java—sphere,
  dynamic—java—sphere(tptp—trace—file),
  "WasInvoked"]
```

or when not using dynamic program information, the following configuration:

```
replacement[
  pde—sphere,
  static —java—sphere]
```

Both Ferret and the Eclipse IDE were instrumented with the Mylar (now Mylyn) Monitor [68] which recorded the queries used as well as details of how the developers used other features in the IDE. As we had found in previous studies that developers often revert to using the tools with which they are familiar, we included some monitoring software to remind the developers of Ferret should they undertake long periods of searching or exploration without using Ferret. The developers were asked to maintain a journal, reflecting on their usage, as well as recording any events when Ferret failed to do something or surprised them in some fashion. We interviewed the developers at the end of each day, asking about their usage of Ferret and discussing the events recorded in their journal. We retrieved the instrumentation logs and journals at the conclusion of the study.

The diary study was originally scheduled to last two days. We anticipated that developers would spend the first day learning to use the tool and incorporate it into their exploration strategies, and would be able to use the tool productively on the second day. We did not initially schedule more time for the diary study because we
did not believe we would be able to recruit developers for use of an experimental tool in their daily work for a longer period of time.

5.2.1 Results

Three of the developers used an integration of two spheres (the static Java and PDE spheres), and one developer used the integration of three different spheres (the static Java, PDE, and dynamic Java spheres). The developers provided limited journal entries, generally recording questions or small usability or performance issues. The monitor logs provided a record of Eclipse and Ferret use, and the developers were able to provide feedback during the end-of-day interview. The developers reported finding Ferret useful, and volunteered to continue using Ferret past the two days originally allocated to the study.

The developers used almost all of the conceptual queries provided by Ferret that were available to them. Table 5.2 reports the average use of each query. Not surprisingly, the most heavily used queries correspond to common queries in JDT. For instance “what calls this method?” accounted for 28% of the usage. For 3 of the 4 developers, this query would have returned the same results as the equivalent JDT query because they were not able to make use of the TPTP integration.

The data in Table 5.2 also shows that the developers used some queries more than once that are not supported by JDT. For example, the intra-class queries for “what (fields, methods, types) does this element access?” were used surprisingly often: we have not found these queries to be useful in our own development work. Presumably the developers found these queries to be more simple and time efficient than the alternative of reading through the source code for the element.

P4 was the only developer to use the TPTP integration to incorporate dynamic runtime information.\(^1\) P4 created a trace to examine some anomalous run-time behaviour exhibited by a particular set of unit tests. He reported that having the ability to only see what was actually called to be useful by eliminating spurious calls made from other tests.

All the developers disabled Ferret’s implicit invocations, reporting that they did not find this capability to be useful, and preferring to invoke Ferret manually. It is

\(^1\) P1 and P2 were not able to use the TPTP integration due to problems with their installation which prevented its use; P3 reported not being in a situation where he felt he needed it.
not clear whether their dislike of the implicit invocations was due to a dislike of the reactiveness or because of the possible performance implications: the developers were acutely conscious of possible drags on performance, which we discuss further in Section 6.2.2.

Table 5.1 presents the usage statistics over the course of the period of the study. The Eclipse usage time was assessed by accumulating the time between events in the monitoring logs, less any gaps of four minutes or more. This table lists the number of explicit invocations of Ferret (“Invoked”); the number of cascading invocations where Ferret was triggered in-place from a previous result (“Cascaded”); and the number of query expansions where a developer examined the results of one of the queries (“Expanded”). Also included are the number of times where a developer caused the results of a query to be clustered by some attribute (“Clusterings”) and the number of Eclipse-based searches made.

The fact that expansions were used more than invocations and cascades indicates that the developers generally looked at the results of several queries per invocation. P3 made particularly heavy use of the clustering, primarily to cluster by package type. He noted that some of his queries had a large number of results, and he was only interested in those from a particular project which were all in a single package.

P1 and P3 both used a small number of Eclipse-based searches during their sessions. The general lack of use of Eclipse-based searches indicates that Ferret seemed to have provided a suitable mix of conceptual queries. P1 explained that he knew precisely what he was looking for, and although Ferret could have returned the information, he did not want to wait for Ferret to find the results. P3’s queries were for occurrences of text strings, which does not match Ferret’s focus on conceptual queries.

Finally, P2 and P3 reported in follow-up e-mails to be continuing to use the tool beyond the end of the study.
Table 5.1: Ferret usage over the course of the diary study; conceptual queries 33–36, which involve the evolution sphere, were not included.

<table>
<thead>
<tr>
<th>Id</th>
<th># Days Used</th>
<th>Eclipse Usage (hh:mm)</th>
<th>Ferret Queries</th>
<th># Eclipse Searches</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Invoked</td>
<td>Cascades</td>
</tr>
<tr>
<td>P1</td>
<td>4</td>
<td>20:24</td>
<td>47</td>
<td>43</td>
</tr>
<tr>
<td>P2</td>
<td>3</td>
<td>14:20</td>
<td>20</td>
<td>10</td>
</tr>
<tr>
<td>P3</td>
<td>3</td>
<td>14:53</td>
<td>31</td>
<td>20</td>
</tr>
<tr>
<td>P4</td>
<td>2</td>
<td>9:04</td>
<td>10</td>
<td>9</td>
</tr>
</tbody>
</table>

Table 5.2: Usage of Ferret's conceptual queries from Study #1 (described in Section 5.2) using the static Java, dynamic Java, and PDE spheres. The percentage for each participant reflects the percentage of their individual use. The ‘Total’ columns reflects the average usage of the query across the four developers.

<table>
<thead>
<tr>
<th>Category / Conceptual Queries</th>
<th>P1</th>
<th>P2</th>
<th>P3</th>
<th>P4</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>#</td>
<td>%</td>
<td>#</td>
<td>%</td>
<td>#</td>
</tr>
<tr>
<td>inter-class</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 What methods return instances of this type?</td>
<td>--</td>
<td>--</td>
<td>1</td>
<td>2%</td>
<td>--</td>
</tr>
<tr>
<td>2 What methods instantiate this type?</td>
<td>7</td>
<td>7%</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>3 What methods take an argument of this type?</td>
<td>1</td>
<td>1%</td>
<td>--</td>
<td>--</td>
<td>1</td>
</tr>
<tr>
<td>4 What fields are declared as being of this type?</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>7</td>
<td>11%</td>
</tr>
<tr>
<td>5 What calls this method?</td>
<td>34</td>
<td>32%</td>
<td>18</td>
<td>36%</td>
<td>16</td>
</tr>
<tr>
<td>6 Where is the value of this field retrieved?</td>
<td>7</td>
<td>7%</td>
<td>2</td>
<td>4%</td>
<td>14</td>
</tr>
<tr>
<td>7 Where is the value of this field set?</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>5</td>
<td>8%</td>
</tr>
<tr>
<td>8 What tests instanceof against this type?</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>2</td>
<td>3%</td>
</tr>
<tr>
<td>9 Where are casts to this type?</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>1</td>
<td>2%</td>
</tr>
<tr>
<td>10 What throws exceptions of this type?</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>11 What catches exceptions of this type?</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>12 What references this type by name?</td>
<td>6</td>
<td>6%</td>
<td>3</td>
<td>6%</td>
<td>2</td>
</tr>
<tr>
<td>13 Which of the classes in this package were actually used?</td>
<td>--</td>
<td>--</td>
<td>1</td>
<td>2%</td>
<td>--</td>
</tr>
<tr>
<td>14 Which methods defined by this class were actually used?</td>
<td>--</td>
<td>--</td>
<td>2</td>
<td>4%</td>
<td>--</td>
</tr>
</tbody>
</table>

(table continues)
<table>
<thead>
<tr>
<th>Category / Conceptual Queries</th>
<th>P1</th>
<th>P2</th>
<th>P3</th>
<th>P4</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>intra-class</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15 What fields does this type or method access?</td>
<td>5 5%</td>
<td>6 12%</td>
<td>2 3%</td>
<td>3 6%</td>
<td>16 5.8%</td>
</tr>
<tr>
<td>16 What methods does this type or method call?</td>
<td>4 4%</td>
<td>7 14%</td>
<td>2 3%</td>
<td>10 6%</td>
<td>23 8.3%</td>
</tr>
<tr>
<td>17 ‡ What types does this {type, method, or extension} reference?</td>
<td>4 4%</td>
<td>3 6%</td>
<td>2 3%</td>
<td>9 17%</td>
<td>18 6.5%</td>
</tr>
<tr>
<td><strong>inheritance</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18 What interfaces does this type implement?</td>
<td>– –</td>
<td>– –</td>
<td>– –</td>
<td>– –</td>
<td>– –</td>
</tr>
<tr>
<td>19 What are this class’ subclasses?</td>
<td>4 4%</td>
<td>– –</td>
<td>2 3%</td>
<td>– –</td>
<td>6 2.2%</td>
</tr>
<tr>
<td>20 What classes implement this interface?</td>
<td>5 5%</td>
<td>– –</td>
<td>3 5%</td>
<td>– –</td>
<td>8 2.9%</td>
</tr>
<tr>
<td>21 What interfaces extend this interface?</td>
<td>5 5%</td>
<td>– –</td>
<td>– –</td>
<td>– –</td>
<td>5 1.8%</td>
</tr>
<tr>
<td>22 What are this type’s siblings?</td>
<td>1 1%</td>
<td>1 2%</td>
<td>– –</td>
<td>– –</td>
<td>2 0.7%</td>
</tr>
<tr>
<td><strong>declaration</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>23 What are all the fields declared by this type?</td>
<td>4 4%</td>
<td>3 6%</td>
<td>– –</td>
<td>– –</td>
<td>9 3.2%</td>
</tr>
<tr>
<td>24 What class methods implement this interface method?</td>
<td>2 2%</td>
<td>– –</td>
<td>2 3%</td>
<td>2 4%</td>
<td>6 2.2%</td>
</tr>
<tr>
<td>25 What interface methods specify this class method?</td>
<td>– –</td>
<td>– –</td>
<td>1 2%</td>
<td>1 2%</td>
<td>2 0.7%</td>
</tr>
<tr>
<td>26 What class methods does this method override?</td>
<td>1 1%</td>
<td>1 2%</td>
<td>1 2%</td>
<td>2 4%</td>
<td>5 1.8%</td>
</tr>
<tr>
<td>27 What class methods override this method?</td>
<td>2 2%</td>
<td>– –</td>
<td>3 5%</td>
<td>2 4%</td>
<td>7 2.5%</td>
</tr>
<tr>
<td>28 What extension points or extensions reference this type?</td>
<td>4 4%</td>
<td>2 4%</td>
<td>– –</td>
<td>– –</td>
<td>6 2.2%</td>
</tr>
<tr>
<td>29 What types is this type adaptable to?</td>
<td>3 3%</td>
<td>– –</td>
<td>– –</td>
<td>– –</td>
<td>3 1.1%</td>
</tr>
<tr>
<td>30 What types could this type have been adapted from?</td>
<td>2 2%</td>
<td>– –</td>
<td>– –</td>
<td>– –</td>
<td>2 0.7%</td>
</tr>
<tr>
<td>31 What extensions are there of this extension point?</td>
<td>4 4%</td>
<td>– –</td>
<td>– –</td>
<td>– –</td>
<td>4 1.4%</td>
</tr>
<tr>
<td>32 What plug-ins depend on this plug-in?</td>
<td>2 2%</td>
<td>– –</td>
<td>– –</td>
<td>– –</td>
<td>2 0.7%</td>
</tr>
</tbody>
</table>
5.3 **Claim: Improves Developer Effectiveness**

**Claim 3 (Developer Effectiveness):** Direct support for conceptual queries helps improve a developer’s effectiveness during software exploration activities.

In this section, we attempt to discover whether the use of conceptual queries has an impact on the effectiveness of software developers in code comprehension tasks.

To answer this question, we undertook a controlled experiment to compare the impact from Ferret and two other specialized software exploration tools on developer effectiveness as compared to using the **JDT** [22]. This experiment specifically sought to answer the following three questions:

- Do specialized exploration tools alleviate the demands made upon a developer’s *mental resources* during a task?
- Do specialized exploration tools help a developer focus on the program elements relevant to the task and avoid irrelevant elements? (*exploration behaviour*)
- Do specialized exploration tools help a developer garner a better understanding of how the task should be carried out on the code? (*correctness*)

To help ensure a realistic setting for our experiment, we recruited 18 professional developers and had the developers use the tools to plan complex changes to an open-source codebase. The tools were compared by their effects on the developers’ subjective mental workload, on their exploration behaviour, and on the correctness of the solutions. We expected that the developers using Ferret should experience a positive result for each.

### 5.3.1 The Experiment

Our experiment sought to assess the effectiveness of three tools introduced to ease the exploration of source code. Supporting materials are included in [Appendix C](#).

**Exploration Tools**

We chose to compare features of Ferret with those of two other specialized exploration tools: JQuery [49], and a pre-release of Suade [78, 105]. Each of these tools
Figure 5.1: JQuery, a query-based source browser where a query’s results are displayed as a tree, and the results provide the context for further queries.  

We chose these tools as they represent a range of different approaches for obtaining and presenting static information extracted from program text, and all have a goal of improving program exploration. Ferret was configured with only the static Java sphere.  

JQuery is a query-based source code browser [49] that shows query results in context (Figure 5.1). A JQuery user can issue a variety of pre-written queries, a superset of the standard queries from the Eclipse IDE. Results of a query are displayed as a tree with expandable and collapsible nodes. Individual elements in the tree provide the context for further queries, whose results are shown in-place. This hierarchical display provides an explicit map of the navigation paths as the tree expansions capture the history of a developer’s exploration.  

Suade [105] is an implementation of the topology-analysis algorithm described by Robillard [78] (Figure 5.2). When requested by the user, Suade recommends additional program elements for investigation using an estimate of their structural relatedness to a previously-identified set of program elements. The recommended program elements are displayed in a separate Eclipse view, ranked according to their estimated relatedness. Elements deemed interesting by the developer can be added.

---

Versions: Eclipse R3.1.2 (www.eclipse.org), Ferret 0.2.14.2 (www.cs.ubc.ca/~bsd/research/ferret/), JQuery 3.1.5 (jquery.cs.ubc.ca), and Suade pre-release 0.1.2 (www.cs.mcgill.ca/~swevo/suade/); verified 2007/03/30
Suade uses structural relatedness to a previously-identified set of program elements to recommend additional program elements for investigation.

to the input set and used for requesting additional suggestions.

Table 5.3 compares the overlap in the conceptual queries supported by Ferret and the queries directly supported by JDT, jQuery and Suade; we exclude those of Ferret’s conceptual queries that require spheres other than the static Java sphere. In making this comparison, we do not include situations where a tool may have some support for answering a conceptual query but with additional mapping or scoping. Suade, which recommends elements as identified by an algorithm, has no direct overlap with any particular query, but its purpose rather overlaps with all queries.

Hypotheses and Measures

We transformed our three study questions into the following three hypotheses.

**Hypothesis 1:** Developers using an exploration tool should report a lower mental workload to complete a task as compared to JDT alone.

We expected that an exploration tool should alleviate the demands made upon a developer’s mental resources while undertaking a task. It may be that an exploration tool has little effect on the output produced by the developer, but improves the developer’s ability to produce the output. We assess this support using the Task Load Index (TLX) [42], a well-validated and easy-to-administer questionnaire for assessing the subjective mental workload experienced by a subject performing a task.\(^3\)

The TLX combines a subject’s report of six different factors (the mental, physical,
Table 5.3: Overlap between Ferret’s static Java conceptual queries and the queries directly supported by JDT, JQuery, and Suade. ‘†’ indicates queries that are not available in current tools and ‘‡’ indicates queries that have partial support in current tools.

<table>
<thead>
<tr>
<th>Category / Conceptual Queries</th>
<th>Eclipse</th>
<th>JQuery</th>
<th>Suade</th>
</tr>
</thead>
<tbody>
<tr>
<td>inter-class</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>What methods return instances of this type?</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>2</td>
<td>What methods instantiate this type?</td>
<td></td>
<td>✔</td>
</tr>
<tr>
<td>3</td>
<td>What methods take an argument of this type?</td>
<td></td>
<td>✔</td>
</tr>
<tr>
<td>4</td>
<td>†What fields are declared as being of this type?</td>
<td></td>
<td>✔</td>
</tr>
<tr>
<td>5</td>
<td>What calls this method?</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>6</td>
<td>Where is the value of this field retrieved?</td>
<td></td>
<td>✔</td>
</tr>
<tr>
<td>7</td>
<td>Where is the value of this field set?</td>
<td></td>
<td>✔</td>
</tr>
<tr>
<td>8</td>
<td>What tests instanceof against this type?</td>
<td></td>
<td>✔</td>
</tr>
<tr>
<td>9</td>
<td>Where are casts to this type?</td>
<td></td>
<td>✔</td>
</tr>
<tr>
<td>10</td>
<td>What throws exceptions of this type?</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>What catches exceptions of this type?</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>What references this type by name?</td>
<td>✔</td>
<td></td>
</tr>
<tr>
<td>intra-class</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>‡What fields does this type or method access?</td>
<td></td>
<td>✔</td>
</tr>
<tr>
<td>16</td>
<td>‡What methods does this type or method call?</td>
<td></td>
<td>✔</td>
</tr>
<tr>
<td>17</td>
<td>‡What types does this {type, method} reference?</td>
<td></td>
<td></td>
</tr>
<tr>
<td>inheritance</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>What interfaces does this type implement?</td>
<td></td>
<td>✔</td>
</tr>
<tr>
<td>19</td>
<td>What are this class’ subclasses?</td>
<td></td>
<td>✔</td>
</tr>
<tr>
<td>20</td>
<td>What classes implement this interface?</td>
<td></td>
<td>✔</td>
</tr>
<tr>
<td>21</td>
<td>What interfaces extend this interface?</td>
<td></td>
<td>✔</td>
</tr>
<tr>
<td>22</td>
<td>†What are this type’s siblings?</td>
<td></td>
<td></td>
</tr>
<tr>
<td>declaration</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>‡What are all fields declared by this type?</td>
<td></td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>What class methods implement this interface method?</td>
<td></td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>What interface methods specify this class method?</td>
<td></td>
<td>✔</td>
</tr>
<tr>
<td>26</td>
<td>What class methods does this method override?</td>
<td></td>
<td>✔</td>
</tr>
<tr>
<td>27</td>
<td>What class methods override this method?</td>
<td></td>
<td>✔</td>
</tr>
</tbody>
</table>

*aSupported by JQuery for methods only.

*bJDT shows an indicator, providing the method does not override another method.
and temporal demands; the effort required; the subject’s view of his performance; and the frustration experienced) to provide a subjective assessment of workload. To compare scores between subjects, the scores are normalized by taking the difference from a baseline task performed without the treatment of interest.

**Hypothesis 2: Developers using an exploration tool should not need to explore as much code as compared to JDT alone.**

We expected that an exploration tool should help the developers focus on the program elements relevant to the task and avoid examining irrelevant elements. We assess this hypothesis in two ways. We first compare the numbers of unique program elements viewed by the developer in each of the tasks (Viewed). Recognizing that some program elements are more relevant to the task solution than others, we also examine the developer’s navigation patterns, comparing the distances strayed from the solution elements during the exploration.

**Hypothesis 3: Developers using an exploration tool should identify more of the salient elements for the task as compared to JDT alone.**

We expected that developers using an exploration tool should garner a better understanding of how the task should be carried out on the code through the improved support for exploring relations in the source code. We assess this hypothesis by comparing the quality of their solutions, determined by the percentage of program elements correctly reported from model solutions (Quality).

**Design**

Our within-subjects experiment compared a developer’s code exploration behaviour when using one of the exploration tools with the same developer’s behaviour when using the standard exploration facilities in JDT. A within-subjects design satisfied the requirements of the TLX to have a baseline for comparison, and allowed compensating for strong individual differences seen in developers [e.g., 79, 80].

Each subject was asked to investigate and document a solution for two change tasks. The order of the tasks was randomized between subjects. Each subject used the normal Eclipse facilities for the first task, which served as a baseline for comparison as required for the TLX. For the second task, each subject was randomly assigned to use one of the three exploration tools. The two tasks (referred to as AS
Table 5.4: Allocation of developers per treatment group.

<table>
<thead>
<tr>
<th>Tool Combination</th>
<th>AS–SR</th>
<th></th>
<th>SR–AS</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Eclipse / JQuery</td>
<td>$G_1$</td>
<td>$G_1$</td>
<td>$G_2$</td>
</tr>
<tr>
<td>Eclipse / Ferret</td>
<td>$G_3$</td>
<td>$G_3$</td>
<td>$G_4$</td>
</tr>
<tr>
<td>Eclipse / Suade</td>
<td>$G_5$</td>
<td>$G_5$</td>
<td>$G_6$</td>
</tr>
</tbody>
</table>

and sr) are described shortly. By stressing the documentation to guide another developer in making the necessary changes, we aimed to emphasize the searching to understand the behaviour of the application.

This experiment used a two-way, repeated measures, factorial design. The two-way refers to the two independent variables (or between-subjects variables): Tool, the exploration tool used for the second task; and Task Order, the order in which AS and sr were tackled. A factorial design tests all combinations of the independent variables, and allows studying the effect of each factor as well as interactions between the factors. We thus have six treatment conditions. Our design has repeated measures as each subject was involved in two trials, referred to as the variable Trial. As we were comparing three exploration tools, with two possible orderings of the tasks (AS–sr or sr–AS), and two trials, this is a $3 \times 2 \times 2$ design. This design may be diagrammed as in Table 5.4, where $T_i$ represents the trial and $G_i$ represents the treatment conditions.

Both the choice of exploration tool and task order were randomized within blocks [54], where a block is a sequence of experiment runs exercising all combinations of variables. This randomization resulted in six treatment conditions; we chose to have three subjects in each treatment condition, resulting in 18 subjects.
Procedure

Each subject was asked to work on two change tasks to an open-source editor, jEdit 4.1-pre6. This version of jEdit comprises approximately 65 KLOC\(^6\) and 679 classes. The subjects had 40 minutes to perform each task. As the exploration tools use only static program information, the subjects were instructed to not use the debugger to ensure a level playing field. The experiment was performed on a 2 GHz duo-core laptop equipped with 1 GB of RAM and a 1400 × 1050 screen.

Each subject was instructed to create a plan for performing the tasks, identifying the program elements that either need to be changed or that need to be understood. The plan was to be appropriate for a senior student to rapidly turn into working code. The plan was to be captured using a specialized Relevant Elements view (TABLE 5.3.1).

After completing the first task, each subject followed a 10 minute tutorial on the exploration tool to be used in the second task. Each tutorial demonstrated the particular tool’s use on an unrelated codebase, JHotDraw 6.0b1.\(^6\) The tutorials were assembled from introductory materials provided by the developers of each tool. To help a subject retain information from the tutorial, we had them use the assigned tool to answer a relatively simple question about JHotDraw; if unsuccessful, the experimenter provided the solution and demonstrated how the tool might be used to obtain the answer. The experimenter also answered any questions and provided tips on tool strategies.

Each subject was asked to use their assigned tool to its fullest extent possible. If the subject was not successful at finding the information, he was allowed to revert to the standard facilities in Eclipse. Because finding a starting point is difficult, we provided an initial seed, a single pertinent element to help start each task.

The experiment was piloted on three students. Minor changes were made in light of comments from the first two students, and the third student, who was the most experienced, was able to successfully complete the experiment. Each of these students felt the tasks were of equivalent difficulty.

\(^4\)www.jedit.org, verified 2007/03/30.
\(^5\)Thousands of lines of code, only counting non-blank lines of code.
\(^6\)www.jhotdraw.org, verified 2007/03/30.
Subjects

We recruited 18 professional developers; each met our criterion of Eclipse proficiency having either used Eclipse for at least 6 months, or having successfully developed an Eclipse plug-in. These developers came from six different companies or organizations, one of which involved developers from two very different business divisions. All developers were male, had a minimum of three years experience writing software in an industrial setting, and had worked on software projects with at least five other developers. All but one had a minimum of two years experience programming with Java; one developer had only one year of Java experience.

JEdit was unknown to all but two developers, neither of whom had seen the code for at least three years. In our follow-up interviews, neither claimed any advantage from that previous exposure.

Tasks

The experiment involved identifying the changes required for two tasks. One of the tasks, \( \text{as} \), involved adding a capability to the autosaving functionality, and was chosen to be identical to that used in previous work \([79]\). The other task, \( \text{sr} \), involved reloading settings when the settings files were edited within jEdit. We identified this task by examining the jEdit source code and identifying a small piece of functionality that was easily removed. The \( \text{sr} \) task seemed comparable to \( \text{as} \): both interacted with only small amounts of the overall system, and both used already-existing functionality and required little new code.

Model solutions to the tasks were created by identifying the methods, fields, and types in the existing source that were either necessary to understand or that required modification. The model solutions did not include new methods or new types that might have been necessary. Types were only included when the solution required them to be subclassed, or when their entire workings were required to be understood in detail. The model solution for the \( \text{as} \) task was identified from the solution from Robillard et al.’s original study. As the \( \text{sr} \) task was created by removing existing code, we simply examined the code that was removed.
Data Collection

We had two primary sources for data collection. The Eclipse workspace was instrumented with the Mylar (now Mylyn) Monitor [68] to collect information on Eclipse and tool usage. Developers were also asked to record their development plan using a custom tool called the Relevant Elements view.

The Mylar Monitor provides a record of the exploration and interaction undertaken by a subject during the experiment, called the monitor history. Events registered include the selections of UI elements, switches between editors, and command executions. Additional instrumentation recorded the program source elements currently visible or made visible through scrolling.

The Relevant Elements view was built for this experiment for the developers to capture and describe the program elements that are relevant to the task. This view supports one-way interaction only, like paper, meaning that it is easy to add elements to the view, but cannot be used to re-open those elements, so as to prevent its being used as an exploration replacement by the developers.

The solutions created by the developers required some clean-up before being processed. Although developers were encouraged to reference all elements necessary for understanding or implementing the change, most did not. We processed the developer solutions to add elements that were referred to in their comments. For example, one developer instructed in his solution:

This is done by iterating over $\text{JEdit.getBuffers()}$ and calling $\text{buffer.get AutosaveFile().delete()}$.

From this description, we added $\text{JEdit.getBuffers()}$ and $\text{Buffer.getAutosaveFile()}$.

Not all situations were so clear cut. We were conservative, adding elements only when referred to in the solution. For example, in the following excerpt:

(The setting can be retrieved from JEdit properties)

This developer had referenced the fetching of properties elsewhere, and so we added $\text{JEdit.getProperty()}$. 

75
Data Analysis

We analyzed the data both visually and statistically to test our three hypotheses.

Visual Assessments  One form of visual analysis entailed plotting and comparing various measurements grouped by the different task orderings and tools.

A second form involved the analysis of plots, called distance profiles, of the developers’ exploration of the programs by comparing the distance they strayed from the solution elements during the exploration. To produce these plots for a particular task, we first transform the jEdit source code into a directed graph. We map types, methods, and field in the source to nodes in the graph. We add edges for the following relations between program elements: declares field, method, and inner type; extends and implements type; returns type; has argument of type; references field; calls method; creates object instance; throws exception; and catches exception. These edges correspond to relations that can be found in a single step from direct examination of the source code or by using a simple query. We then assign a distance to each node calculated as the node’s minimum distance $d$ (ignoring edge directionality) to any of the nodes identified in the model solution ([Table 5.3.1]) for that task. A node with $d = 0$ is a solution element. A node with $d = 1$ is a program element that either uses or is used by a solution element. Nodes with $d > 1$ have no direct tie to a solution element.

A plot for a particular developer using a particular tool is then created by processing the interaction history collected by the Mylar Monitor. We determine the distance for each new element that appeared on the developer’s screen and assigned the distance of the screen to be the minimum distance of any elements currently on-screen.

Statistical Assessments  Statistical analyses were used to assess support for our findings through use of Analysis of Variances (ANOVAS) [48]. ANOVAS are a set of statistical techniques to identify sources of variability between groups. A one-way ANOVA assumes a single independent variable; a two-way ANOVA assumes two independent variables. ANOVAS allow dealing not only with the individual effects of the independent variables, but also any interactions that may arise between them. The result
of an ANOVA is a set of statistical assessments of the different individual effects and effect combinations. We chose a 10% significance level in assessing the statistical tests\(^7\) as it is generally considered suitable for exploratory analysis.

We were unable to test the samples for normality; such tests generally require seven or more samples, and our design with 18 developers resulted in only three samples per group. However ANOVA is generally robust to non-normality providing the variances in the groups are the same [48], which was confirmed.\(^8\)

5.3.2 Results

We present the results of the experiment by examining each of the three hypotheses. We note that all developers did use their assigned exploration tool throughout their task, although two of the Ferret users, one of the JQuery users, and three of the Suade users also reverted to the Eclipse search facilities on occasion.

Mental Support

**Hypothesis 1:** Developers using an exploration tool should report a lower mental workload to complete a task as compared to Eclipse alone.

Figure 5.3 shows the differences in TLX scores reported by the developers, grouped by the different exploration tools and the task orderings, as well as grouped by only the task orderings.\(^9\) Each point corresponds to an individual developer and corresponds to \(t_2 - t_1\) where \(t_i\) is the TLX score from task \(i\), and where the second task was undertaken with the exploration tool. A TLX difference greater than zero indicates that the developer felt that the second task (always undertaken with the exploration tool) was harder than the first, and a difference less than zero that the second task was easier. The data supports the first hypothesis if the TLX differences are generally less than zero, indicating that the tasks in which the exploration tool were used were easier.

\(^7\)All statistical tests performed using R 2.3.1 ([www.r-project.org](http://www.r-project.org)) on NetBSD 4.99.1 on a 2 GHz Intel Core Duo.

\(^8\)Homogeneity of variances assessed using Bartlett’s test.

\(^9\)We present many of the results graphically using boxplots showing the 0, 25, 50, 75, and 100 percentiles; percentiles are particularly appropriate for summarizing data with possible skew. Results from individual developers are represented by single points, which may be jittered to reveal overlaps.
Figure 5.3: Differences in TLX scores, separated by tool and task order, and by task-order only.

Table 5.5: ANOVA table for TLX.

<table>
<thead>
<tr>
<th>Source</th>
<th>df</th>
<th>SS</th>
<th>MS</th>
<th>F</th>
<th>p</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task Order</td>
<td>1</td>
<td>2701.8</td>
<td>2701.8</td>
<td>7.0646</td>
<td>0.02087*</td>
</tr>
<tr>
<td>Tool</td>
<td>2</td>
<td>4.5</td>
<td>2.3</td>
<td>0.0059</td>
<td>0.99412</td>
</tr>
<tr>
<td>Task Order × Tool</td>
<td>2</td>
<td>342.6</td>
<td>171.3</td>
<td>0.4479</td>
<td>0.64922</td>
</tr>
<tr>
<td>Residuals</td>
<td>12</td>
<td>4589.4</td>
<td>382.4</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

From Figure 5.3, we see that the TLX differences lie both above and below zero regardless of the exploration tool used. Grouping the differences by task order, we see that the differences are reflected around zero, indicating that the AS and SR tasks were perceived as having different degrees of difficulty. Table 5.5 presents the results of a two-way ANOVA comparing the TLX differences by task ordering and exploration tool; the ANOVA reports the only statistically significant difference being by task order \((F(1, 12) = 7.06, p = 0.02)\). From Figure 5.3, the AS task appears to impose less workload than the SR task, regardless of tool. A pairwise comparison of the TLX differences grouped only by the two different task orderings, ignoring
the exploration tools, reports a mean difference of $-24.5$ with a 90% confidence interval of $[-40.9, -8.1]$.$^{10}$

As a result of the overwhelming effect of the task on the result, the TLX difference cannot be attributed only to the tool, and we have no evidence to support our hypothesis. The exploration tools may actually alleviate subjective mental workload, but the effect associated with the tasks outweighs the effect associated with the tools.

In our follow-up interviews, we asked whether and how the developers found their respective tool useful. We also asked what feedback they would provide to the developers of the tools. Four of six developers reported Ferret as useful, four of five reported JQuery as useful, and two of five reported Suade as useful. The two non-respondents for JQuery and Suade were ambivalent about the tool used. Two of the Suade dislikers attributed their dislike to difficulty in evaluating the reasons behind Suade’s recommendations. Several developers from each tool group asked about the availability of their respective tool. As the developers used only a single tool, we cannot compare their impressions between tools.

**Context Explored**

**Hypothesis 2: Developers using an exploration tool should not need to explore as much code as compared to Eclipse alone.**

We expected that developers using the tools should be more focused on the program elements relevant to the task.

---

$^{10}$Computed using Tukey’s Honestly Significant Difference procedure.
tool was used. These numbers capture neither repeat visits to previously-viewed program elements, nor the amount of time spent examining the program elements.

From Figure 5.4, we see that most of the points lie below zero, indicating that developers seemed to look at more code for their first task when Eclipse was used. Table 5.6 reports the results of a two-way repeated-measures ANOVA comparing the Viewed scores by task ordering and exploration tool, reports the only statistically significant effect being an effect by trial \( F(1, 12) = 3.56, p = 0.08 \). There is no statistical support for a difference between tools. A paired \( t \)-test\(^\text{11} \) reports a mean difference of \(-26.7\) elements between the first and second trials with a 90% confidence interval of \([-53.5, -0.05]\). This difference between trials can be interpreted in two ways: (i) the exploration tools had an equal and relatively uniform effect, or (ii) there were carry-over effects between trials.

**Detailed Examination of Navigation Paths:** We also undertook a more detailed qualitative analysis of the developers’ actual program exploration by comparing the distance they strayed from the solution elements (described in Table 5.3.1). For

\(^{11}\) A paired \( t \)-test is used as the values are within subjects.
Table 5.6: ANOVA table for unique program elements viewed.

<table>
<thead>
<tr>
<th>Source</th>
<th>df</th>
<th>SS</th>
<th>MS</th>
<th>F</th>
<th>p</th>
</tr>
</thead>
<tbody>
<tr>
<td>Between subjects</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tool</td>
<td>2</td>
<td>24454</td>
<td>12227</td>
<td>1.6897</td>
<td>0.2257</td>
</tr>
<tr>
<td>Task Order</td>
<td>1</td>
<td>9669</td>
<td>9669</td>
<td>1.3362</td>
<td>0.2702</td>
</tr>
<tr>
<td>Tool × Task.Order</td>
<td>2</td>
<td>5486</td>
<td>2743</td>
<td>0.3791</td>
<td>0.6924</td>
</tr>
<tr>
<td>Residuals</td>
<td>12</td>
<td>86836</td>
<td>7236</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Within subjects</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Trial</td>
<td>1</td>
<td>6453.4</td>
<td>6453.4</td>
<td>3.5575</td>
<td>0.0837*</td>
</tr>
<tr>
<td>Tool × Trial</td>
<td>2</td>
<td>4282.1</td>
<td>2141.0</td>
<td>1.1803</td>
<td>0.3405</td>
</tr>
<tr>
<td>Task Order × Trial</td>
<td>1</td>
<td>3560.1</td>
<td>3560.1</td>
<td>1.9625</td>
<td>0.1866</td>
</tr>
<tr>
<td>Tool × Task Order × Trial</td>
<td>2</td>
<td>6501.1</td>
<td>3250.5</td>
<td>1.7919</td>
<td>0.2085</td>
</tr>
<tr>
<td>Residuals</td>
<td>12</td>
<td>21768.3</td>
<td>1814.0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

both AS or SR tasks, the maximum distance assigned to any program element was six steps, and all developers remained within a distance of three steps (i.e., \( d \leq 3 \)).

Figure 5.5 shows examples of distance profiles for several users from their two tasks.\[^{12}\] A distance profile indicates only that the code displayed by Eclipse had the minimum distance listed; the developer may have spent some of that time looking at another artefact, such as the task descriptions. These graphs have three additional annotations, an ‘×’ marks where a solution element was first seen, a ‘o’ where a solution element was first correctly identified, and a ‘+’ when a different file was viewed.

In comparing the distance profiles, we see three slight trends, all involving spikes. Spikes are meaningful in that they represent occasions where the developer has started to investigate beyond elements directly related to the solution (i.e., where \( d > 1 \)). Of interest are the duration and frequency of the spikes: how long was it before the developer realized that the line of inquiry was not actually related to the task?

The first trend we observe is a difference by task: generally the distance profiles

\[^{12}\]The full set of distance profiles are included in Section C.3.
Figure 5.5: Distance profiles for users 6, 7, 8, and 10: (a) is from the first task, always undertaken with Eclipse; (b) is from the second task, undertaken with one of the exploration tools. The greyed profiles (7(a), 8(a), 6(b), and 10(b)) are from the AS task; the white profiles (7(b), 8(b), 6(a), and 10(a)) are from the SR task.
from sr appear to be spikier than those from as, with more frequent movements between the different distance levels. Such spikes may arise from examining an element and quickly realizing its irrelevance to the task. Although the sr task involved fewer actual solution elements, it did require examining a class hierarchy, which may account for some of the rapid descents. Further support for this observation is provided by comparing the average amounts of time spent per task in examining items with $d \leq 1$ versus $d > 1$. Using only the times from the first trial, which was always undertaken with Eclipse, the developers spent more time exploring items with $d \leq 1$ for as (92%, s.d. ±9%) as compared to sr (73%, s.d. ±12%).

The distance profiles also reflect individual differences. Even though the developers generally started from the same locations (the seeds), the profiles immediately diverge as developers chose different areas or items to explore. We also see wide variances in the amounts of time spent at the different distance levels $d$.

The final trend is that JQuery appeared to produce less-pronounced profiles: spikes appear to be less frequent, but longer-lasting. This may arise as the query tree structure promotes more breadth-first-search examination of query results; with the other tools, a new query generally causes the query context to be lost, perhaps promoting more depth-first-search examination.

Figure 5.6 compares the percentage of the time spent investigating program elements with $d \leq 1$. Developers using Eclipse seem to spend most of their time viewing such elements for the as task, but less so for sr. The tools may have helped somewhat in sr, given that the mean of time spent in $d \leq 1$ appears higher for the exploration tools for sr. This effect is slight at best, and a two-way ANOVA provides no statistical support for this speculation.

From examining the ‘×’ in the distance profiles, developers came across new relevant elements the entire duration. We see no evidence of an early broad strategy, followed by a period of refinement.

Generally we did not see any consistent behaviour across the users. If there had been a strong tool effect than we would have expected to see more consistent patterns.
Figure 5.6: Comparisons of the percent of time spent looking at code at distance $d \leq 1$ for AS and SR.

Solution Items Identified

Hypothesis 3: Developers using an exploration tool should identify more of the salient elements for the task as compared to Eclipse alone.

We expected that developers using the exploration tools should have garnered a better understanding of how the task should be carried out on the code. Their identified solution sets should be missing fewer of the key elements required for a correct solution.

Figure 5.7 compares the percentage of solution program elements available to the developers, distinguishing between selected, visible, and identified program elements. Selected elements refers to the explicit selection of user interface elements corresponding to program elements: the developers asked to view these elements. Visible elements refers to program elements whose definitions were either partially or wholly on-screen. Finally, identified elements refers to the program elements correctly identified by developers as being required for the solution. The numbers shown are expressed as the percentage of the solution program elements identified, as normalized by the number of solution program elements for each task.

We report these three separate measures as no single measure captures the in-
Figure 5.7: Between-subjects percentage of solution program elements respectively selected, visible, and identified by the developers.

formation required. The program elements selected value underestimates the program elements investigated as it does not account for program elements made subsequently visible through scrolling. The visible program elements value overestimates the program elements investigated as it assumes the developer has examined all of the program elements available on-screen. In examining the program elements visible, we see that none of the developers using Eclipse and only a very few of those developers using the exploration tools saw all of the solution program elements required. None of these developers correctly identified all solution program elements. We use the percentage of solution program elements correctly identified as the quality of the solution.

Figure 5.8 compares the within-subject differences in the solution quality, as assessed by the percentage of solution program elements correctly identified as compared to our ideal solutions, differentiating by task and tool. Each point corresponds to an individual developer and corresponds to $q_2 - q_1$ where $q_i$ is the percentage of solution program elements correctly identified from task $i$, and where the second task was undertaken with the exploration tool. A number greater than zero would imply that the developer produces a higher quality solution during the sec-
Figure 5.8: Within-subject differences in the percentage of solution elements correctly identified, separated by tool and task.

...ond task when an exploration tool was used. The only visible effect seems to be by task ordering and there does not appear to be any effect by exploration tool. A two-way repeated-measures ANOVA comparing the Quality scores by task ordering and exploration tool (Table 5.7) confirms this observation, showing a statistically significant interaction effect between the task ordering and the trial ($F(1, 12) = 21.4, p = 0.0006$).

Table 5.7 also shows a statistically significant effect by tool between subjects, however this is of no practical significance. This would mean that the measures differed by the tool group, regardless of the trial—but the first trials were never undertaken with the exploration tools. No practical significance can be assigned to this result.

From our follow-up interviews, it seemed that many of the developers appeared to have made more headway than would appear from their solutions. It is possible that they were willing to speculate in the interview, but would not put their speculations to ‘paper’, so to speak.
Table 5.7: ANOVA table for percentage of solution program elements correctly identified.

<table>
<thead>
<tr>
<th>Source</th>
<th>df</th>
<th>SS</th>
<th>MS</th>
<th>F</th>
<th>p</th>
</tr>
</thead>
<tbody>
<tr>
<td>Between subjects</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tool</td>
<td>2</td>
<td>0.208885</td>
<td>0.104442</td>
<td>4.8045</td>
<td>0.02933*</td>
</tr>
<tr>
<td>Task Order</td>
<td>1</td>
<td>0.017243</td>
<td>0.017243</td>
<td>0.7932</td>
<td>0.39064</td>
</tr>
<tr>
<td>Tool × Task Order</td>
<td>2</td>
<td>0.019892</td>
<td>0.009946</td>
<td>0.4575</td>
<td>0.64345</td>
</tr>
<tr>
<td>Residuals</td>
<td>12</td>
<td>0.260863</td>
<td>0.0217396</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Within subjects</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Trial</td>
<td>1</td>
<td>0.00041</td>
<td>0.00041</td>
<td>0.0245</td>
<td>0.8783307</td>
</tr>
<tr>
<td>Tool × Trial</td>
<td>2</td>
<td>0.02259</td>
<td>0.01129</td>
<td>0.6767</td>
<td>0.5266703</td>
</tr>
<tr>
<td>Task Order × Trial</td>
<td>1</td>
<td>0.35758</td>
<td>0.35758</td>
<td>21.4272</td>
<td>0.0005811*</td>
</tr>
<tr>
<td>Tool × Task Order × Trial</td>
<td>2</td>
<td>0.03919</td>
<td>0.01959</td>
<td>1.1741</td>
<td>0.3422107</td>
</tr>
<tr>
<td>Residuals</td>
<td>12</td>
<td>0.20026</td>
<td>0.01669</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

5.3.3 Discussion

The Absence of Practical Effects from Tools

We found no evidence of any practical effect from the software exploration tools. There are several possible explanations.

There were effects from the tools but our measures were inaccurate or do not capture the phenomena of interest. Our analysis showed no effect from the tools used, despite positive comments made by the subjects about the tools tested and requests from those subjects for the tools. It may be that we require more sensitive measures than the simple, unweighted scores used, or require a sample with more than 18 developers. Our qualitative analysis relied largely on the distance profiles that take an information-centric view of how the task was approached by developers. As there may be other possible solutions to the tasks, the measures of distance from ideal solutions should be treated with some caution; however virtually all the solution elements would be required in some form by any solution. The deep cognitive na-
ture of programming work may not be reflected in these behavioural measurements, both quantitative and qualitative, that we used.

Our experience does shed doubt on the usefulness of measures proposed and used in other experiments, such as the response times for answers [75]. We believe a more comprehensive set of measures needs to be developed than has been proposed to date. We suggest such a set include the use of the TLX. Although we were unable to establish the necessary baseline conditions to use the TLX to determine the perceived impact of the tools because of the unanticipated differences in perceived difficulty of the tasks, we were able to use it to gauge the developers’ perception of task difficulty.

Any effects from the exploration tools are being confounded by differences between the tasks, or confounded by carry-over (or learning) effects. The two tasks used in our experiment were sufficiently complex that no developer successfully identified all the relevant elements, and their perceived difficulty confounded the results from TLX (Section 5.3.2). It is possible that the other two measures of effectiveness are also affected by task difficulty, thus confounding an effect by the tool. Selecting a different initial seed, a different task, a different domain, or even using a different coding style, might produce a very different result.

To assess the likelihood of learning effects, we computed the distance profile for one task using the model solution of the other task. We found no statistical significance in correlation between the percentage of time in $d = 0$ and the solution qualities. This result suggests there was little carry-over effect between the tasks.

The exploration tools have no practical effect for these particular tasks. We see two possible explanations. First, it is possible that the tools are providing useful and relevant information to the developers, but that the developers are not taking in the information. Developers may need further explanations for the information, or may need to discover the information on their own to make use of it.

Second, since none of the developers correctly identified all relevant elements for either task, it may be that the static program information is insufficient to identify all of the relevant information for a program change task. It may be that Eclipse

---

13 Pearson’s $r = 0.36$ at the 10% level ($n = 18$).
already does a good job, and that these new tools are simply repackaging readily-available information in new forms—finessing existing solutions, rather than providing new means to solve the problems.

**Impacts on Validity**

In addition to the possible construct validity threats mentioned above, there are several other factors that pose threats to our experiment.

The first factor arises from our choice to use real tasks as realistic tasks rarely have a single possible approach. This situation arose in our study where two of our subjects proposed very different solutions to the Sr task as each supposed additional requirement beyond what was stated.

Another factor comes from the imposition of a time limit. We chose a 40 minute limit as it seemed long enough for the developers to make at least some headway in understanding the task, and yet minimized total necessary time so as to avoid turning off potential subjects. In each task, six of the 18 subjects finished their task before the time allotted.

With the exception of disallowing the use of the debugger, we did not control what tools the developers could use. The dynamic information from debugging could have been significant and helped diagnose some of the trickier interactions.

Finally, we also noticed one exploration strategy that was used almost exclusively by developers from a single organization. These developers used Eclipse's Call Hierarchy view which provides a means of drilling down through method calls and whose results are shown in-place in a tree. This view possibly confers similar benefits to JQuery. Several of these developers commented that they had been shown this view by another developer in their organization. This kind of cultural communication of helpful strategies may pose additional threats to validity.

**Implications for Future Evaluations**

It is possible that the developers were hampered by unfamiliar UIs. This might be remedied by additional training or possibly by having more adaptive tools, such that the tool will suggest itself for use. Longitudinal evaluation approaches might be more appropriate in such circumstances, providing opportunities for subjects to
learn and integrate the tool into their strategies.

Previous studies report finding effects for particular exploration tools when using elementary search tasks [22], whereas our study failed to find any noticeable effects for complex tasks of a longer duration. This suggests that the effects from improvements to elementary search operations are dominated by the cognitive requirements of programming tasks.

Might we have had different results with a different set of tasks? It is possible that the tools studied were unsuitable for the particular tasks used. However, it is not clear why this might be: jEdit is implemented using plain Java, and neither task involved complicated architectures or other paradigms. Future work should address understanding more precisely the characteristics of the tasks we used that made developers perceive them differently.

5.4 Claim: Efficient Implementation

Claim 4 (Efficient Implementation): The sphere model and conceptual queries can be efficiently implemented.

A model is of little use if it is impossible to implement queries efficiently. In this section, we demonstrate that our approach performs sufficiently for use in real systems by presenting the results of several benchmarks of Ferret in use. In Section 6.2.2, we relate subjective impressions on Ferret performance from several software developers.

Table 5.8 presents average timings taken to perform a Ferret invocation for a selection of types and methods from ArgoUML. The table presents timings for two different sphere configurations. The first configuration (labelled ‘C1’) used only the static Java sphere and corresponds most closely to invoking a number of queries using a normal static Java tool. This configuration allows comparing the time taken with a static Java tool and, more importantly, calibrating against what a human might accept from a tool. The second configuration (labelled ‘C2’) used the sphere configuration described in Section 4.2.5 (p. 51), which combined static Java

---

14Timings were performed for ArgoUML 0.13.6 (1605 Java classes), running on a Lenovo ThinkPad T60 2GHz Core Duo with 1GB of RAM using the Sun JDK 1.5.0_11-b03, Ferret 0.3.0.200708261920, Eclipse SDK R3.3.0, and with 36 projects open.
information and dynamic program information from a dynamic execution trace. This dynamic execution trace had filtered all classes from packages matching java*, com.sun*, com.ibm*, sun*, and org* except for those classes matching org.argouml*. The resulting trace recorded 417 291 events over 2 604 methods defined in 760 classes. Rows 1–3 measure the performance of queries for a Java class or interface, and rows 4–6 measure the performance of queries for Java methods.

In comparing the timings between the two configurations, the addition of the dynamic Java sphere seems to have a mixed effect with some query times increasing slightly and others decreasing or remaining the same. The increases in query times (rows 3, 4 and 6) correspond to queries using Java interfaces or methods defined on a Java interface: the increase likely results from the conversion between the JDT and TPTP representations required for the WAS-INVOKED relation (as described in Section 4.2.5). The decreases in query times (rows 2 and 4) likely benefit from the faster query times from the TPTP queries, which are done from an in-memory representation.

In comparing within the two configurations there appears to be quite a significant difference in Ferret’s query performance for different Java types. Rows 1–3 show a dramatic difference in the performance of queries for three different types. We attribute these differences to the number of users of the Java type on which the queries focused: for example, the classes PropPanelEvent and PropPanelCallEvent have very few localized uses whereas the interface TargetListener is widely implemented. As Ferret’s static Java sphere currently uses the JDT search facilities to implement the concrete queries, which defers most of the processing required for a search to search time, elements that are widely used will require more time to compute the relations. In comparing rows 4–6, there is a similar but more subdued difference in Ferret’s query performance for methods.

Although these times are not instantaneous, they are generally faster than the times required by a human to visit and combine the results of queries between multiple tools. These times can also be improved by using better-suited query back-ends. Other search back-ends might optimize search time, which would be better for Ferret since it issues multiple queries at once.
Table 5.8: Benchmark timings for use of Ferret for two different sphere configurations (seconds, average of five runs); configuration C1 represents the static Java sphere, and C2 for the sphere configuration as described in Section 4.2.5.

<table>
<thead>
<tr>
<th>Row</th>
<th>Element</th>
<th>C1</th>
<th>C2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>PropPanelCallEvent (class)</td>
<td>0.3</td>
<td>0.3</td>
</tr>
<tr>
<td>2</td>
<td>PropPanelEvent (abstract class)</td>
<td>0.6</td>
<td>0.5</td>
</tr>
<tr>
<td>3</td>
<td>TargetListener (interface: 232 implementers)</td>
<td>14.3</td>
<td>15.5</td>
</tr>
<tr>
<td>4</td>
<td>PropPanelEvent.paramScroll (field)</td>
<td>0.3</td>
<td>0.4</td>
</tr>
<tr>
<td>5</td>
<td>TabProps.findPanelFor(Class) (method)</td>
<td>0.5</td>
<td>0.2</td>
</tr>
<tr>
<td>6</td>
<td>TargetListener.targetSet(Object) (method)</td>
<td>1.1</td>
<td>1.9</td>
</tr>
</tbody>
</table>

5.5 Summary

In this chapter we assessed four claims to substantiate our thesis.

In Section 5.1 we demonstrated the expressiveness of our sphere model by showing the range of conceptual queries supported by our model.

In Section 5.2 we used the results of a diary study to argue that the conceptual queries implemented have practical relevance to developers during their day to day work.

In Section 5.3 we describe our attempt to find demonstrable evidence that our approach, as embodied in the Ferret tool, leads to an improvement in developer effectiveness. We instead found a seeming contradiction in that we found no apparent effect from Ferret or either of the other two tools, yet the majority of the developers using Ferret report finding Ferret to be useful and ask about the availability of the tool. This interest suggests that Ferret was providing useful and relevant information to the developers, but the effects from improvements to search operations are dominated by individual strategies and the cognitive requirements of programming tasks.

Finally, in Section 5.4 we presented a set of benchmarks of Ferret in use to show that our approach performs sufficiently for use in real systems.
Chapter 6

Discussion

In this chapter, we describe the main issues and trade-offs that arose during the development and evaluation of the sphere model and Ferret. We also note avenues for future research involving the sphere model and Ferret.

6.1 The Sphere Model

6.1.1 Is This Model Necessary?

One might question if the problems of representation and integration identified in this dissertation are inherent in integrating information from different sources of program information or simply oversights or unanticipated requirements in the engineering of the original tools. The only tools we can identify whose integration does not benefit from our model are those where the following two conditions hold: (i) domain elements that can be related have a one-to-one correspondence, and (ii) there is no overlap in the concrete queries provided by the tools. For situations where these conditions do not hold, handling the issues of element correspondence and sphere selection as provided by the sphere model would require either deciding to accept limited integration (e.g., supporting only one-to-one correspondences and selecting the concrete queries from one tool to be dominant) or implementing functionality equivalent to that provided by our model.

For example, Table 4.2 (p. 58) suggests that there is some overlap in the static
and dynamic Java spheres. The sphere model allows determining how such overlap should be handled through the use of composition functions. Even had there been no overlap in any of the concrete queries, the static and dynamic Java spheres have one-to-many correspondences.

To summarize, our sphere model makes integrating a new source of information much more simple. By using our sphere model for integration, different tools can continue to be developed independently, and tools can also be used in unanticipated ways. By implementing simple concrete queries and building conceptual queries from these concrete queries, testing becomes more simple, since there is a much more simple interface, and validating the conceptual queries is potentially easier as they are described in high-level terms. And the model supports integrating tools whether the integration requires handling overlap in concrete queries or establishing correspondences between elements.

### 6.1.2 Considering Other Composition Functions

Our sphere model as described in Chapter 3 provides three composition functions, union, replacement and transformation. As noted earlier, other composition functions to represent intersection, subtraction, and symmetric difference, may also be desirable; to date, we have not found a use-case to justify their need. Although these new composition functions are theoretically possible, there are significant complications in their definitions and implementations.

To help illustrate some of the difficulties, consider the following definition for intersection:

**Intersection:** Intersection preserves only the relations in common between the composing spheres.

\[ f(R_1, R_2) = R_1 \cap R_2 \]

At first glance, intersection might seem useful for restricting the results of a static sphere with what was actually called from a set of dynamic profiles, similar to what we achieve using the sphere composition described in Section 4.2.5. However intersection as defined above will strip out all relations that are not common to \( R_1 \) and \( R_2 \). Such behaviour is of limited use when composing two different types of spheres as it greatly constrains the types of conceptual query that can be supported.
Intersection likely only makes sense when $R_1$ and $R_2$ are from the same sphere or when they are sure to have the same relation labels.

There are also complications in implementing composition functions such as intersection. Implementing intersection is straightforward when the elements of the relations are from the same sphere. The implementation is less straightforward when dealing with relations formed of elements from different spheres when there are correspondences between those elements. To help illustrate the difficulty, consider intersecting the following relations $A$ and $B$:

\[
A = \{ \text{(calls, m, n)} \} \quad \text{from the static JDT}
\]

\[
B = \{ \text{(calls, m', n'_1), (calls, m', n'_3)} \} \quad \text{from the dynamic TPTP}
\]

where (i) $m$ and $n$ are JDT methods, (ii) $m', n'_1, n'_2$, and $n'_3$ are TPTP methods, (iii) $m$ corresponds to $m'$, and (iv) $n$ corresponds to $n'_1, n'_2$, and $n'_3$. There are then three possibilities for taking the intersection of these two sets of relations:

1. $\{\text{(calls, m', n'_1), (calls, m', n'_3)}\}$ since $n'_1$ and $n'_3$ are part of the correspondences of $n$. However this can only be correct if each of $n'_1, n'_2$, and $n'_3$ independently correspond to $n$.

2. The empty set, $\emptyset$, since $\{\text{(calls, m', n'_2)}\} \notin A \cup B$ (i.e., all the correspondences of $n$ must be found in $B$).

3. $\{\text{(calls, m, n)}\}$, since this relation is at least partially met (i.e., at least one of the correspondences must be found in $B$).

Possibilities 2 and 3 differ by whether they require all or any of the possible correspondences to be found. These complications arise as intersection, subtraction, and symmetric difference require comparing the equivalence of individual relations between two sets of relations. These comparisons are not found in union, replacement, or transformation. It is to be hoped that, should a solid use-case be found, the use-case will clarify which of the possibilities described above should be chosen.

### 6.1.3 Placing Conceptual Queries in Context

By enabling the integration of different tools, the sphere model broadens the set of conceptual queries that can be answered by tools, although the model is not suffi-
Figure 6.1: A Venn diagram placing the conceptual queries answerable using the sphere model in context to different classes of conceptual queries.

The figure presents a Venn diagram illustrating how the conceptual queries answerable using spheres relate to the conceptual queries asked by developers. In forming this diagram, we distinguish between the class of conceptual queries supported through the sphere model, labelled as $S$, from three other classes of conceptual queries. The first class is formed from the questions identified from observations of developers at work by Sillito et al. [84] and others (discussed in Section 7.1), which are labelled as $J$. A second class of conceptual queries are those that can be answered directly using current software tools, which are labelled as $T$. These classes of conceptual queries are all subsets of the set of possible conceptual queries that may be asked by a developer, which are labelled as $Q$. This diagram conveys that the sphere model has broadened the set of conceptual queries that can be answered by tools, but there remain a sizeable set of questions that cannot yet be answered by any tool.

6.2 The Ferret Implementation

6.2.1 The Granularity of Sphere Elements and Correspondences

The elements of a sphere normally fall out naturally from the source of program information being represented. But for there to be a real benefit from use of our model, the various spheres must expose elements of particular units of granularity such that there is some possibility to find correspondences between elements from the spheres. For example, the static Java sphere exposes methods, fields, types,
initializers, and packages as its elements. Had the Kenyon sphere only exposed elements at the granularity of files then there would have been little chance of triggering interesting queries. Kenyon is able to move beyond simply representing files to also represent meaningful structural content within those files such as Java types and methods, and thus enabling questions about the evolution of individual methods rather than just files.

6.2.2 Barriers to Deploying Ferret

We have noticed two barriers to deploying Ferret in real situations. The first barrier relates to performance implications, which is the more significant of the two as the performance characteristics of a tool will dictate how users use the tool. The second barrier relates to the development costs to integrate new spheres.

Scaling Up: Performance Implications

Much of the feedback from the developers from our diary study (described in Section 5.2) involved querying performance. The version of Ferret tested included an experimental feature to perform queries in the background. The background queries were intended to prime Ferret's cache for potential future queries, but they caused copious numbers of objects to be produced as a side-effect, triggering garbage collection. The developers were particularly distrustful of any background operations within Eclipse when those activities led to the creation of garbage objects as a garbage collection for a Java virtual machine with several hundreds of megabytes of memory could lead to the virtual machine being blocked and unresponsive for several seconds at an inopportune time. Disabling Ferret's background querying prevented the creation of garbage, thus improving the responsiveness of Eclipse, but also disabling a potential performance improvement for Ferret.

Such performance implications are important to identify. Although delays in response times will not necessarily cause users to abandon a tool, such delays will change how users incorporate the tool in their work strategies [94]. Our subjects reported that, having requested a query from Ferret, they were willing to wait several seconds for the results. But they may have used Ferret more often had Ferret been faster.
In the case of the diary study, the garbage was created by the JD T search facilities. The JD T search facilities defer the majority of search processing to search time, so as to minimize the work required at save time. The search processing may require parsing a number of source files and result in the creation of garbage. This search processing is not as much of an issue when performing individual searches, as is their expected use. Ferret, however, issues a number of searches for its different queries which magnifies the effect of the deferred processing required for searches.

One performance improvement for Ferret then lies in finding or creating better-suited query back-ends that perform more search processing at save time.

The Development Cost of a New Sphere

Implementing a new sphere for Ferret has an up-front development cost. We anticipate that this cost would be borne by a group responsible for an organization’s development tools and thus the cost would be amortized across the developers who will benefit from the resulting sphere.

The true development cost of implementing a new sphere is difficult to assess. In our experience from implementing the four spheres described in this dissertation, the most significant barrier to development has been in understanding how to use the models exposed by these tools. For example, the difficulty in implementing the Kenyon sphere came from understanding the expectations of how to use the Kenyon framework to both store and access the information extracted from the source code. The difficulty in implementing the PDE sphere came from understanding the rather convoluted PDE source code in order to expose the correct information. Generally the time spent understanding these tools would have been necessary to integrate them with any other tool.

6.2.3 The Ferret User Interface

Much of the feedback we received from the diary study (Section 5.2) pertained to UI issues. These comments were generally aimed at concrete issues such as font-size changes or minor nits about the display of certain pieces of information. There are three elements about the Ferret UI that we think worthy of further investigation.

On selection of an element, Ferret can automatically compute and display the
results of all conceptual queries issued from that element without the active involvement of the developer. Our expectation was that by removing a barrier in performing searches, developers would be able to focus on the information found rather than the retrieval of that information. The developers in our diary study preferred to manually invoke Ferret. We are not sure if this was a result of the performance issues discussed in Section 6.2.2 or because they genuinely did not find the automatic computation to be useful.

To minimize the possibility of Hawthorne effects during our experiments, we deliberately kept Ferret’s UI very simple by displaying the queries and query results using a tree-style UI widget. Such display structures are widely used throughout Eclipse projects and thus are likely familiar to developers. However, tree UI widgets preclude displaying other important information. Other presentation structures, such as mSpaces [65] or faceted metadata browsers [43], or graphical methods, may present better forms of organizing the information to better inform the user of additional information such as the availability of clusters.

In Section 4.1.1 we described how Ferret’s conceptual queries are grouped into five different categories (inter-class, intra-class, inheritance, declarations, and evolution). These categories were assigned based on the type of information provided by the query. However the categories differ from those proposed by others [52, 61, 84]. Identifying categories that are meaningful to developers during their exploration is likely to be of help.

6.3 Evaluation

6.3.1 Why Does Ferret Help?

Our controlled experiment (Section 5.3) found no evidence of any practical effect from any of the software exploration tools. How do we reconcile this lack of a demonstrable effect with developers in both the diary study and controlled experiment reporting finding Ferret to be useful? Why might we expect Ferret to work? We posit four possibly-overlapping explanations.

First, Ferret provided the developers access to conceptual queries that they either could not or did not know how to make without Ferret. We believe Ferret’s
plethora of specialized conceptual queries reduces the effort required of developers to find the information they are actually seeking. \(P2\) from our diary study (Section 5.2) expressed this point.

Second, Ferret may provide additional awareness of how program elements are used. Ferret displays the results of all configured conceptual queries. As a result, a developer may chance across a question they may not have thought of asking, or known they could ask, and enabling serendipitous (unexpected) discovery \([5]\).

Third, Ferret’s correspondences effectively implement a form of human reasoning called suppositional reasoning \([41]\). Suppositional reasoning is the making of tentative assumptions and then reasoning based on those assumptions. Ferret carries correspondences between different tool-representations of the same objects and supports reasoning. Recording and propagating fidelity across queries helps alert the user to the possibility of incorrect suppositions.

Finally, Ferret has a number of features that increase visual momentum (Section 2.2.1) during software exploration. Ferret serves as functional overlay as it provides additional information about how the elements queried are used within the system. Ferret promotes navigational landmarks by supporting follow-on queries from an element to be displayed in-place in the tree, and has a number of features to support exploratory digressions. Such features should help in alleviating the onset of disorientation.

### 6.3.2 Thoughts on Experimental Designs for Future Evaluations

There have been recent calls for use of standard benchmarks to support comparisons across tools \([85]\). From the results of our experiment in Section 5.3, we would urge caution in adopting this approach for evaluation of software exploration tools without first understanding more precisely the characteristics of the tasks that made programmers perceive them differently. Our experimental results showed a strong confounding effect from our choice of tasks, despite choosing tasks that we thought were similar in scale and that had been successfully piloted. It may be that a different set of tasks would have produced an entirely different result.

Can we characterize what made these tasks more difficult? The results from our experiment suggest that programmers found the SR task to be more difficult than the
as task. We undertook a preliminary qualitative examination of the solutions for these two tasks and identified two differences: the types of changes required, and the background information about jEdit subsystems required to understand and accomplish the task. To correctly solve the AS task required understanding the conditions controlling the autosave functionality, identifying the locations that check these conditions, and then modifying how and when those conditions are set. However, the SR task required using an internal message bus framework for broadcasting editor changes, and required first understanding a hierarchy of messages, followed by identifying the particular messages to monitor.

It may be possible to use such characterizations to formulate a metric to predicting the likely difficulty of a task—a cognitive metric of task difficulty. Given that experiments will use a known task with a known solution, one approach to calculate such a measure might be to consider the program as a graph and compute a graph-based statistic from the solution subgraph.

These difficulties in evaluation are not unique to software engineering. The information retrieval community focusing on exploratory search have encountered similar difficulties in evaluating their approaches [63, 106]. One of the difficulties mentioned is the absence of accepted metrics for evaluating the effectiveness of tools supporting exploratory search [106]. Until such metrics have been found and agreed upon, perhaps we will have to satisfy ourselves with more qualitative or heuristic methods, such as cognitive walkthroughs [89] or cognitive theory-driven protocol analyses [e.g., 102].

6.4 Future Work
Our approach opens several areas for further investigation.

6.4.1 Temporal Information
Our implementation does not handle temporal queries. In implementing our JDT support we included some simple correspondences to map the JDT/Debug representations of types, methods, and fields to their JDT equivalents (as shown in Figure 4.5). The JDT/Debug could be seen as a separate sphere in its own right, supporting questions such as “why was this method not called?” or “why was this value changed?”
### Table 6.1: Examples of other possible spheres

<table>
<thead>
<tr>
<th>Type</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Elements</td>
<td><em>debugging</em>: stack frames, variables, values</td>
</tr>
<tr>
<td></td>
<td><em>bug reports</em>: bug reports, submitters, programmers</td>
</tr>
<tr>
<td></td>
<td><em>UI</em>: widgets, events, methods</td>
</tr>
<tr>
<td>Relations</td>
<td><em>debugging</em>: contains, has-value,</td>
</tr>
<tr>
<td></td>
<td><em>bug reports</em>: fixes, fixed-by, submitted-by, responded-to</td>
</tr>
<tr>
<td></td>
<td><em>UI</em>: implements, defines, handles, contains</td>
</tr>
</tbody>
</table>

Adding such support would require having a representation of values at time \( t \) and to identify causal chains. Ko and Myers [56]'s Whyline show that this is possible.

#### 6.4.2 Using Past Queries to Predict Future Queries

The event logs resulting from our 40 minute sessions in our experimental study generally logged 1405 events on average. Could we use the history of the queries examined to help in determining future queries? The history could be used to prioritize the order of queries, and defer queries that are unlikely to be used. Or there could be patterns in how the queries are examined reflecting the state of mind of the developer (e.g., proceeding from *intra-class* queries to *inter-class* queries).

#### 6.4.3 Other Spheres

There are a number of other spheres that could be implemented, such as those listed in Table 6.1; these are only tentative. We have personally experienced frustration in attempting to identify the body of code responsible for a particular UI widget implementation. Some of these spheres, such as the UI sphere, would be more complicated as they would apply to a program being debugged.
Chapter 7

Related Work

In this chapter, we consider previous work related to the sphere model and to the Ferret tool for software exploration. As the sphere model seeks to help developers in exploring and understanding their programs, we begin in Section 7.1 by reviewing the work in how software developers understand a program. A particular focus in supporting understanding has been on query languages and static cross-referencing tools; we compare our model and tool to these efforts in Section 7.2. Some have extended static cross-referencing efforts to other artefact types; we consider these efforts in Section 7.3. In Section 7.4, we consider efforts to improve program understanding through integration of limited domains, namely dynamic and static information about a program. Previous efforts to more generally deal with software tool integration have attempted to support integrating artefacts or producing new artefacts in response to changes; we examine these in Section 7.5. Finally the database community have long been concerned with integrating heterogenous databases, and we briefly review their efforts in Section 7.7.

7.1 How Developers Understand Programs

Our justification for the sphere model and the Ferret tool is to better support developers in their work. In this section we review work examining how software developers comprehend programs.

A body of work has developed based on the analysis of software developers and their programming comprehension activities. Some of this work has resulted in the
development of several cognitive theories to describe the program comprehension process. These theories can be organized into three categories. Bottom-up theories [e.g., 72] propose that understanding begins in forming low-level abstractions when reading the source code that are then grouped and processed into higher-level abstractions. Top-down theories [e.g., 32] describe the comprehension process as mapping previously-acquired knowledge about the program domain onto structures found in the code. The final category of cognitive theories treat the comprehension process as a mixture of bottom-up and top-down models [e.g., 100]. These theories attempt to describe how developers build and acquire their mental representations of programs but do not provide any guidance as to how developers may approach their exploration of a software system.

Follow-on efforts have attempted to translate the theories of program comprehension into practical design recommendations for tools. von Mayrhauser and Vans [101] analyze a selection of the cognitive theories to identify lists of typical tasks and their associated information needs. Storey et al. [91] examine the cognitive theories to identify fourteen design elements to be considered when building a tool. The design guidance provided by such papers are useful for identifying gaps in the tool support provided to developers.

Another class of work comprise detailed examinations of developers as they engage in their programming activities. For example, Ko et al. [57] examined how developers used JDT to edit their source code to inform the design of new editing tools. Murphy et al. [68] analyzed developers’ tool usage patterns of the tools provided in JDT. Robillard et al. [79] performed a detailed investigation into the characteristics that differentiated between successful and unsuccessful developers in attempting a change task. Although these studies are often limited in the range of situations and users considered, they provide rich detail of how developers use their tools.

A final class of work centres around the analysis of the questions asked by software developers during their programming activities. Letovsky [61] analyzed the questions asked by six developers, identifying five types of high-level conjectures made by developers about low-level program elements. Erdos and Sneed [27] reflected on their personal experiences to identify seven low-level technical questions pertaining to the definition or use of various program elements. Johnson and Erdem [52] examined and categorized the questions asked by developers on a USERNET
newsgroup, classifying them as either goal-oriented (seeking to achieve some result), problem-oriented (symptom-based diagnosis), or system-oriented (seeking to understand the system). Finally Sillito et al. [84] identified a large set of questions asked by developers as they performed a change task. In comparing these questions to available tools, they report a significant mismatch in the queries supported by tools as compared to the questions asked by developers [83, 84]. We use the results of these studies (i) to justify our approach on focusing on developer’s conceptual queries, and (ii) as a source of the conceptual queries.

7.2 Software Exploration Tools

There have been a wide range of query languages and static cross-referencing tools created to help developers in software exploration.

7.2.1 Query Languages

There have been a number of languages proposed for querying software. Most have used a standard database language such as SQL or Datalog (e.g., CodeQuest [26]), a Prolog or Prolog-like implementation (e.g., JQuery [49]), or a custom language (e.g., SCA [71]). Self-hosted, closed-loop development environments, such as Smalltalk [36] or Lisp [95], where the application elements are reified and manipulatable in the language environment, provide the ability to write very focused queries on the program structure; these environments still have implicit correspondences between runtime representations and the reified application elements which must be explicitly addressed when they occur in queries. None of these approaches provide support for the expression of correspondences between elements or for the automatic propagation of queries across corresponding elements. These languages also do not consider how to reconcile the choice of different implementations of the same query. The sphere model addresses both of these issues. But we have not yet determined the limits of the expressibility of conceptual queries using the sphere model.

7.2.2 Cross-Reference Tools

Cross-referencing tools support developers in understanding the relations between program elements in the source code. Such tools generally work by extracting and
storing certain code-level relationships to a database, and then providing an interface to developers to issue queries against such a database. Masterscope [95], part of the Interlisp programming environment, supported interactive querying for callers of a function or accessors of a variable. Later tools, such as the C Information Abstraction System (CIA) [14], showed how cross-referencing queries could be used to automate software engineering tasks. The Rational Environment for developing Ada programs [29] provided advanced capabilities for traversing the relationships between elements and for reviewing revision history.

Queries can be made from the results of other queries forming a path. For example, JQuery [49] maintains all the query paths as a single tree view so that developers can work back along a branch to understand how they came to a location. Such an ability, it is posited, should help prevent the onset of disorientation [49].

Suade [78, 105] uses a topology-analysis algorithm to recommend additional program elements for investigation using an estimate of their structural relatedness to a previously-identified set of program elements. Elements deemed interesting by the developer can be added to the input set and used for requesting additional suggestions.

Our work differs from the work in cross-referencing tools in two respects. The first difference lies in the scope of the information available to be queried. With Ferret’s use of the sphere model, static program information is but one sphere and other sources of program information can be incorporated to provide more focused results to conceptual queries. The second difference lies in the tool approaches in that Ferret seeks to answer conceptual queries found to be of interest to developers. Cross-reference tools generally provide a set of possible concrete queries to be used by a developer, but this requires the developer to map and scope his query, and thus the developer suffers from the three querying burdens identified in Chapter 4.

7.3 Cross-Artefact Search Tools

We have found three tools that support support integrating other sources of information with the information obtained from static analysis of the source code. GSEE [28] claims to have a flexible data model that can be used to examine data from other domains, such as Java Beans. Sextant [81] supports integration of the
relations embedded in Enterprise Java Beans deployment descriptors. TkSee \cite{TkSee} supports a set of cross-artefact searches to link lines in source code to task-specific information and to examine runtime data. Each of these approaches assume that there is a direct correspondence between elements shared between artefacts from different domains. Each of these tools also lacks a mechanism to select or prioritize between sources of information with overlap; as a result, there is no means to reconcile the choice of different implementations of the same query. As we have shown in this dissertation, questions involving the dynamic runtime behaviour of programs require this feature. The sphere model addresses both of these issues.

### 7.4 Integrating Dynamic and Static Views

In addition to the cross-artefact search tools described above, there have been several tools proposed to help integrate static and dynamic program information in order to provide visualizations of the behaviour of program execution.

Richner and Ducasse \cite{RichnerDucasse} describe an approach to integrate static and dynamic information using logic programming. They use logic queries to drive program visualizations. As their implementation was centred around Smalltalk, which has no reification of interfaces, they did not encounter the valency issues in conversions. Where there was overlap in the static and dynamic information, such as has been identified in this dissertation, the overlap was reconciled by using the dynamic information and ignoring the static information.

Systä \cite{Systa} integrates static with dynamic information to produce Unified Modelling Language (UML)-style sequence diagrams and state diagrams. Upon the selection of some subset of the classes or methods, Shimba instruments the program as required to produce either a sequence diagram representing interactions between selected objects, or a state diagram representing the control flow behaviour of the selected object or method. Shimba’s focus is on the production of particular views rather than on a model to integrate different kinds of program information.

The sphere model provides the ability to integrate of static and dynamic information but is driven from supporting queries rather that visualizations.
7.5 Integrating Artefacts from Tools

A number of approaches have focussed on integrating software development tools, such as unifying different user interfaces and data requirements. Several different levels of integration have been recognized, from presentational aspects, to sharing of data formats, to co-operation and notification between tools [103].

Garlan’s views [32] is perhaps closest to our work. Tools within a development environment, for instance, an editor and a compiler, often require different perspectives on similar data to perform a desired action. To avoid duplication of data in such an environment, Garlan proposed the use of views that together formed a logical database over which all tools in the environment could operate. One can ask whether Garlan’s approach could be used to implement conceptual queries by considering each query as a tool. We believe this approach is not feasible because Garlan’s model does not easily support the transformation and combination of the result sets of queries over the data being represented. Our intent is not to enable tools to work together to achieve their intended goal, but rather to gather from a developer’s perspective the results that such tools produce.

A large and varied group of work aims to support integrating software development tools in order to support (i) the automated production of new software artefacts in response to changes to other artefacts [33, 76, 112], (ii) reconciling inconsistencies between artefacts from different stages of the software development lifecycle [3, 13], or (iii) facilitating the sharing of the external artefacts produced by different tools [53, 90]). We take a different approach by focusing on the use of integration of different sources of program information to provide better support to developers during exploration of the source code.

7.6 Integration via Meta-Models

OASIS [50, 51] seeks to integrate different tools by representing each tool as a service. The approach requires a human integrator first identifying the domain elements, called concepts, and the services provided by each tool to be integrated. The concepts and services are then analyzed by the human integrator to construct a common domain ontology. Every tool requires a conceptual service adapter (CSA) that is responsible for mapping the tool’s elements and services to and from an in-
termediate format as represented by the domain ontology. When the user requests a particular service for some particular element, the element’s CSA converts the element to the intermediate format, and then the requested service’s CSA converts from the intermediate format to the service’s native format and then performs the service. OASIS’s focus on services and use of conversions is similar to our sphere model. However OASIS does not handle the issues of possible multivalent correspondences. An additional limitation is that their approach requires all decisions to be made up-front: introducing a new tool could change the domain ontology and require changes to all custom adapters. Ferret only performs conversions as required, and new tools can be added without widespread changes.

Other groups have focused on defining meta-models to express correspondences between software artefacts so as to support reasoning [1, 34, 66]. However these approaches support only coarse-grained artefacts and assume a one-to-one mapping between the artefacts. Other finer-grained meta-models, such as the Dagstuhl Middle Metamodel [60], could be used to integrate information such as with our sphere model; we are not aware of any work that has attempted to do so.

7.7 Information Integration

Although not strictly related to software exploration, the database community have undertaken a number of large and varied research efforts seeking to semantically integrate different sources of information. These efforts have adopted various names such as record linkage [108], name matching [8], data integration [58], information integration [35], semantic interoperability [69], and semantic integration [24]. Doan and Halevy [24] explain that integrating a set of data sources generally involves: schema matching, the merging of the different schemas describing the different data sources into a single global schema; data matching or tuple matching, deciding whether data elements from the different sources refer to the same real-world concept; and data integration, where data from multiple sources are transformed to a single schema.

The issue most similar to those addressed in Ferret has to do with record- or tuple-matching between two databases. Zhao [113] characterizes this problem as follows: given two collections of items $A$ and $B$, for each item $a \in A$ find a $b \in$
such that $a$ and $b$ are the same item. A variety of approaches have been suggested [24]. However there is a fundamental assumption that tuples have a one-to-one match, and the matches are bidirectional.

The Semantic Web effort, whose goal is to support autonomous agents in reasoning from machine-understandable web resources, must also deal with the schema-matching issues. The effort has standardized on Web Ontology Language (owl) [87], the web ontology language. owl supports equating classes, properties, and individual instances using the sameAs operation [87, §4]. This is a bidirectional equivalence however, and owl does not support degrees of similarity. Hence it does not have the same flexibility of correspondences.
Chapter 8

Conclusions

A software developer explores a software system by asking and answering questions using various software exploration tools [84, 88]. We refer to the questions asked by developers as conceptual queries, and the questions answered by tools as concrete queries. To answer his questions, a developer may need to consult various sources providing information about the program, such as the static relationships expressed directly in the source code, the run-time behaviour of a program recorded in a dynamic trace, or evolution history as recorded in a source management system. The developer uses a variety of tools to help answer his questions. Despite the support afforded by tools, developers often struggle to find the necessary information to answer their questions [79, 84, 88] and, as we have described in Chapter 2, may even become disoriented, where they feel mentally lost and are uncertain of what they were trying to accomplish.

We have identified three factors of the state-of-the-practice Eclipse IDEs that may lead to or exacerbate disorientation: (i) an absence of connecting navigation context when switching files to follow program relations, (ii) an inability to simultaneously view all of the information necessary for a task, and (iii) an absence of support for managing a developer’s digressions. But even if all three factors were to be solved, developers would still struggle to find the necessary information to answer their questions. In studying the questions asked by developers when understanding a software system, Sillito et al. [84] observed that there is a significant mismatch in the concrete queries provided by tools as compared to the conceptual
queries asked by developers. From this observation and from the findings from our own efforts in investigating disorientation, we have identified three different burdens encountered by developers in trying to answer their conceptual queries:

1. The developer is responsible for mapping a conceptual query onto the available concrete queries provided by a tool and in scoping the possibly broad results to just those of interest.

2. The developer may need to compose the results from multiple concrete queries to answer a conceptual query, and may need to use the results from one query to drive another query.

3. The developer may need to integrate and reason across several sources that provide information about a program.

These burdens lead to the developers having to spend additional effort to answer their conceptual queries.

This dissertation has advanced a thesis that conceptual queries can be better supported through a model to represent and compose different sources of information about a program. The basis of this model is the sphere, which serves as a simple abstraction to provide a unified view of a source of information about a program. Many of the tools used by a developer can be represented as spheres. Spheres can be composed in a principled fashion to form a single queryable knowledge-base suitable for answering conceptual queries, such that information from a sphere may replace or supplement information from a different sphere. For example, using our sphere model, a developer can use dynamic runtime information from an execution trace to replace information from the static source code to see what actually occurred.

We have implemented this model in a configurable tool, called Ferret. We have used the facilities provided by the model to implement 36 conceptual queries identified from the literature, blogs, and our own experience, and to support the integration of four different sources of program information. Establishing correspondences between similar elements from different spheres allows a query to bridge across different spheres in addition to allowing a tool's UI to drive queries from other sources of information.
We made four claims in this dissertation:

1. **Expressiveness of the Model**: The sphere model broadens the set of possible conceptual queries answerable by tools.

2. **Relevance**: The conceptual queries implemented are relevant to developers.

3. **Developer Effectiveness**: Direct support for conceptual queries helps improve a developer’s effectiveness during software exploration activities.

4. **Efficient Implementation**: The sphere model and conceptual queries can be efficiently implemented.

Using Ferret we have evaluated the expressiveness of the model by implementing 36 different conceptual queries using four spheres. We have shown that the conceptual queries can be efficiently implemented.

To evaluate this thesis’ claims of relevance and effectiveness we have undertaken two empirical studies, a diary study and a controlled experiment. In the diary study, four professional software developers used Ferret during their development work over several days. The developers used almost all of the conceptual queries that were available to them. Perhaps more important, the developers reported finding Ferret useful in their day to day software development, volunteered to extend the study, and some reported continuing to use the tool beyond the end of the study.

To evaluate this thesis’ claim of effectiveness, we have undertaken a controlled experiment. In this controlled experiment we compared the impact on the effectiveness of 18 professional software developers as they attempted to document the changes required to implement two change tasks. We compared Ferret in addition to two other specialized software exploration and to Eclipse by comparing the relief on the mental workload, the change in the amount of source code viewed, and the correctness of their solutions. Although the developers using Ferret generally reported finding Ferret to be useful, we found that none of the specialized exploration tools seemed to have had any practical effect on developer effectiveness. Rather we found that any effects were dominated by the strategies employed by the individual developers and by the task. This suggests that the effects reported from performing small elementary tasks are dominated by the cognitive requirements of realistic
programming tasks. Future evaluations require understanding more precisely the characteristics of tasks that make developers perceive them differently.

8.1 Contributions

This work has made the following contributions to the field of software engineering.

First, we have provided an initial examination of the phenomenon of disorientation in software development, confirming that disorientation affects expert developers, and introduced the use of visual momentum to aid in identifying three factors that contribute to inducing, or help prevent, disorientation.

Second, we have provided a general model for representing and integrating different sources of program information. Each source of program information is represented as a *sphere*. Spheres can be composed in a principled manner using a set of composition functions. The expression of a conceptual query then corresponds to making concrete queries against a sphere, which may be a composite sphere.

Third, we have provided a usable tool supporting the sphere model. The model supports expressing the queries. Our tool also implements a set of conceptual queries that have not been implemented in previous tools. This tool can serve as a platform for experimenting with spheres.

Fourth, we have introduced the use of the Task Load Index (TLX) [42] to gauge how the participants view task difficulty. We have introduced the use of distance profiles to evaluate the degree to which a programmer remains on-task during their program exploration.

Finally, we have conducted two empirical studies to evaluate the practical impact of three software exploration tools on the performance of a software developer. We have used the TLX and distance profiles to argue that any effects of the exploration tools are likely dominated by the strategies employed by a programmer and by the task. Our results suggest that evaluations using small elementary tasks may not generalize to more realistic settings.
Bibliography


118
 Wiley ➢ pages 27

 doi:10.1145/1062455.1062491 ➢ pages 58

 pages 52

 Duxbury, 4th ed. ➢ pages 76, 77


[50] D Jin, JR Cordy (2005). Ontology-based software analysis and
 reengineering tool integration: The oasis service-sharing methodology. In
 pages 108

 94–99. doi:10.1109/ICPC.2006.30 ➢ pages 108

 doi:10.1023/A:1008655629091 ➢ pages 99, 104

 pages 108

 3rd ed. ➢ pages 72

 pages 14

\doi{10.1109/TSE.2006.116} ⇒ pages 5, 104

\doi{10.1145/543613.543644} ⇒ pages 109


\doi{10.1016/j.entcs.2004.01.008} ⇒ pages 109

\doi{10.1016/0164-1212(87)90032-X} ⇒ pages 99, 104


\doi{10.1145/1121949.1121979} ⇒ pages 101


\doi{10.1145/1012807.1012852} ⇒ pages 99

\doi{10.1016/0167-9236(96)00007-3} ⇒ pages 109

\doi{10.1145/60000.55832} ⇒ pages 32


doi:10.1145/234373.234396 ▷ pages 71

doi:10.1109/TSE.2006.94 ▷ pages 7, 106


123


Appendix A

The Binary Relational Algebra

This appendix provides an overview of the binary relational algebra. We use Schmidt and Ströhlein [82] as our reference source.

We first define a relation as:

**Definition A.1** Let $V$ be a set. A homogenous relation $R$ on $V$ is a subset of the cartesian product $V \times V$. Elements $x, y \in V$ are said to be in $R$ if $(x, y) \in R$.

Relations may be defined either by exhaustively listing all pairs, or through a comprehensive specification. Relations are generally labelled with a name. Relations are symmetric.

A relation has a domain and a range.

**Definition A.2 (Domain)** Let $R \subseteq V \times V$. The domain of $R$ is the set

$$\text{dom}(R) = \{x \in V \mid (x, y) \in R\}.$$

**Definition A.3 (Range)** Let $R \subseteq V \times V$. The range of $R$ is the set

$$\text{ran}(R) = \{y \in V \mid (x, y) \in R\}.$$

A relation may be transposed.

**Definition A.4 (Transpose)** Let $R \in V \times V$. The transpose of $R$ is defined as

$$R^T = \{(y, x) \mid (x, y) \in R\}.$$
The binary relational algebra provides a suite of operators that take relations as their operands and return relations as their result. The algebra includes the traditional set operators suitably defined for relations such as union ($\cup$), intersection ($\cap$), and difference ($-$). The algebra also provides other operators, three of which are composition, domain restriction, and range restriction.

**Definition A.5 (Composition)** Let $R, S \subseteq V \times V$ be relations. Then the composition $R \circ S \subseteq V \times V$ is defined as:

$$R \circ S = \{(x, z) \in V \times V \mid \exists y \in R : (x, y) \in R \land (y, z) \in S\}.$$

**Definition A.6 (Domain restriction)** The domain restriction operator $S \triangleleft R$ selects the pairs of relation $R$ whose first element is contained in set $S$:

$$S \triangleleft R = \{(x, y) \in R \mid x \in S\}.$$

**Definition A.7 (Range restriction)** The range restriction operator $R \triangleright S$ selects the pairs of relation $R$ whose second element is contained in set $S$:

$$R \triangleright S = \{(x, y) \in R \mid y \in S\}.$$

Predicates may be expressed as relations where the elements affirmed by the predicate form the domain and range of the relation.
Appendix B

Conceptual Query Definitions

This appendix provides a listing of the definitions for the 36 conceptual queries implemented in Ferret. Ferret conceptual queries follow these definitions by requesting a named relation from an instance of ISphere. Relations have been underlined where there are implementations of the relations provided by different spheres.

As the relation names form a flat namespace, we have used prefixing to avoid potential clashes. Prefixes were omitted in the body of the dissertation to avoid complicating the examples. The prefixes used in our implementation include:

‘Oo-’: for concrete queries relating to object-oriented matters, such as those provided by the static and dynamic Java spheres. This prefix also allows the conceptual queries to be used for other object-oriented languages.

‘Evo-’: for evolution-related concrete queries, such as those provided by the Kenyon sphere.


‘Dyn-’: for specifically dynamic runtime-related concrete queries that do not belong in the ‘Oo-’ prefix.

Some of these prefixes differ from the sphere names. This is necessary as some spheres provide relation implementations with common semantics. The relation names capture the semantics of the relations.
Some of the conceptual queries correspond to queries available in existing tools and have been created to provide all information fundamentally related to software exploration in one place, to try to alleviate the need for developers to switch tools for commonly requested information. As the individual relations are generally useful to developers, they are usually complemented with a corresponding conceptual query.

If a relation cannot be resolved for its input, then the relation is resolved to the null relation. This resolution means that the types of the input may change the meaning of the definitions.

In the following definitions we use the in-place relation notation of $x.R.y$ to indicate a relation taking $x$ to $y$. Where obvious, the $x$ and $y$ are omitted. The conceptual queries take an input and return a result set (indicated respectively by input and result). The definitions of some queries make use of intermediate results which are shown in lower-case italics.

### B.1 Relation Definitions

Table B.1 provides the supported input and result types for the implemented relations. Some of these relations are predicates, such as OoIsFIELD, where the elements affirmed by the predicate form the domain and range of the relation. Tracing through the typing information requires some knowledge of the inheritance relations in the JDT domain model (Figure B.1), as well as the type correspondences as defined in Figure 4.5 (p. 52).

#### Table B.1: Implemented relation definitions for Ferret, listing the supported input and result types; relations have been underlined where there are implementations of the relations provided by different spheres.

<table>
<thead>
<tr>
<th>Input Type</th>
<th>Relation</th>
<th>Results Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>static java sphere</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IType</td>
<td>OoAllSubtypes$^3$</td>
<td>IType</td>
</tr>
<tr>
<td>IType</td>
<td>OoAllSupertypes$^3$</td>
<td>IType</td>
</tr>
<tr>
<td>IType</td>
<td>OsSiblings</td>
<td>IType</td>
</tr>
</tbody>
</table>

$^3$OoAllSubtypes and OoAllSupertypes use ‘All’ to denote that they correspond to all the sub/super-types and not just the immediate subtypes or supertypes.
Table B.1: (continued)

<table>
<thead>
<tr>
<th>Input Type</th>
<th>Relation</th>
<th>Results Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>IType</td>
<td>OoSUPERTYPES</td>
<td>IType</td>
</tr>
<tr>
<td>IType</td>
<td>OoIMPLEMENTORS</td>
<td>IType</td>
</tr>
<tr>
<td>IType</td>
<td>OoINSTANCEOF</td>
<td>IMember</td>
</tr>
<tr>
<td>IType</td>
<td>OoINSTANTIATORS</td>
<td>IMember</td>
</tr>
<tr>
<td>IType</td>
<td>OoCASTSToTYPE</td>
<td>IMember</td>
</tr>
<tr>
<td>IType</td>
<td>OoCATCHESException</td>
<td>IMember</td>
</tr>
<tr>
<td>IType</td>
<td>OoTHROWSException</td>
<td>IMember</td>
</tr>
<tr>
<td>IMethod</td>
<td>OoREFERENCESType</td>
<td>IMember</td>
</tr>
<tr>
<td>IMethod</td>
<td>OoREFERENCESMETHOD</td>
<td>IMember</td>
</tr>
<tr>
<td>IType</td>
<td>OoMETHODOReturnsType</td>
<td>IMethod</td>
</tr>
<tr>
<td>IType</td>
<td>OoMETHOdWithArgOfType</td>
<td>IMethod</td>
</tr>
<tr>
<td>IField</td>
<td>OoFIELDGETTERS</td>
<td>IMember</td>
</tr>
<tr>
<td>IField</td>
<td>OoFIELDSetters</td>
<td>IMember</td>
</tr>
<tr>
<td>IMember</td>
<td>OoTYPESREFERENCES</td>
<td>IType</td>
</tr>
<tr>
<td>IMember</td>
<td>OoMETHODSCalled</td>
<td>IMethod</td>
</tr>
<tr>
<td>IMember</td>
<td>OoFIELDSUsed</td>
<td>IField</td>
</tr>
<tr>
<td>IPackageFragment</td>
<td>OoPROVIDEDTYPES</td>
<td>IType</td>
</tr>
<tr>
<td>IType</td>
<td>OoDECLAREDFIELDS</td>
<td>IField</td>
</tr>
<tr>
<td>IType</td>
<td>OoDECLAREDMETHODS</td>
<td>IMethod</td>
</tr>
<tr>
<td>IMember</td>
<td>OoDECLARINGTYPE</td>
<td>IType</td>
</tr>
<tr>
<td>IMethod</td>
<td>OoCLASS</td>
<td>IType</td>
</tr>
<tr>
<td>IMember</td>
<td>OoISINTERFACE</td>
<td>IType</td>
</tr>
<tr>
<td>IMember</td>
<td>OoISFIELD</td>
<td>IField</td>
</tr>
<tr>
<td>IMethod</td>
<td>OoSIGNATURE</td>
<td>String</td>
</tr>
</tbody>
</table>

Dynamic java sphere

<table>
<thead>
<tr>
<th>TRClass</th>
<th>DynWASINSTANTIATED</th>
<th>TRClass</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRMethod</td>
<td>DynWASINVOKED</td>
<td>TRMethod</td>
</tr>
<tr>
<td>TRClass</td>
<td>OoINSTANTIATORS</td>
<td>TRMethod</td>
</tr>
<tr>
<td>TRMethod</td>
<td>OoREFERENCESMETHOD</td>
<td>TRMethod</td>
</tr>
<tr>
<td>TRMethod</td>
<td>OoMETHODSCalled</td>
<td>TRMethod</td>
</tr>
<tr>
<td>TRMethod</td>
<td>OoSIGNATURE</td>
<td>String</td>
</tr>
</tbody>
</table>

Kenyon sphere

<table>
<thead>
<tr>
<th>IKHandleSource</th>
<th>EvoMODIFICATIONS</th>
<th>KTransaction</th>
</tr>
</thead>
<tbody>
<tr>
<td>KTransaction</td>
<td>EvoAUTHOR</td>
<td>String</td>
</tr>
<tr>
<td>KTransaction</td>
<td>EvoELEMENTSCHANGED</td>
<td>Node</td>
</tr>
<tr>
<td>KTransaction</td>
<td>EvoFILESCHANGED</td>
<td>Node</td>
</tr>
</tbody>
</table>

Plug-ins sphere

<table>
<thead>
<tr>
<th>IType</th>
<th>PdeADAPTABLEFROM</th>
<th>IPluginElement</th>
</tr>
</thead>
<tbody>
<tr>
<td>IType</td>
<td>PdeADAPTABLETO</td>
<td>IPluginElement</td>
</tr>
</tbody>
</table>

(table continues)
Figure B.1: The JDT domain model.

Table B.1: (continued)

<table>
<thead>
<tr>
<th>Input Type</th>
<th>Relation</th>
<th>Results Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>PdeIdentifier</td>
<td>PdeReferencesIdentifier</td>
<td>IPluginElement</td>
</tr>
<tr>
<td>IPluginExtensionPoint</td>
<td>PdeDeclaredExtensions</td>
<td>IPluginExtension</td>
</tr>
<tr>
<td>IPluginModelBase</td>
<td>PdeDependents</td>
<td>IPluginModelBase</td>
</tr>
<tr>
<td>IPluginExtension</td>
<td>OoTypesReferenced</td>
<td>IType</td>
</tr>
</tbody>
</table>

B.2 Inter-Class Queries

1. What methods return instances of this type?

\[
\text{result} \gets \text{ran}(\text{input} \circ \text{OoMethodReturnsType})
\]

2. What methods instantiate this type?

\[
\text{instantiators} \gets \text{OoImplementors} \circ \text{OoInstantiators} \\
\text{result} \gets \text{ran}(\text{input} \circ \text{instantiators})
\]

3. What methods take an argument of this type?

\[
\text{result} \gets \text{ran}(\text{input} \circ \text{OoMethodWithArgOfType})
\]
4. What are all the fields declared by this type?

\[ \text{fields} \leftarrow \text{OoReferencesType} \circ \text{OoIsField} \]

\[ \text{result} \leftarrow \text{ran}(\text{input} \circ \text{fields}) \]

5. What calls this method?

\[ \text{result} \leftarrow \text{ran}(\text{input} \circ \text{OoReferencesMethod}) \]

6. Where is the value of this field retrieved?

\[ \text{result} \leftarrow \text{ran}(\text{input} \circ \text{OoFieldGetters}) \]

7. Where is the value of this field set?

\[ \text{result} \leftarrow \text{ran}(\text{input} \circ \text{OoFieldSetters}) \]

8. What tests instanceof against this type?

\[ \text{result} \leftarrow \text{ran}(\text{input} \circ \text{OoInstanceOf}) \]

9. Where are casts to this type?

\[ \text{result} \leftarrow \text{ran}(\text{input} \circ \text{OoCastsToType}) \]

10. What throws exceptions of this type?

\[ \text{result} \leftarrow \text{ran}(\text{input} \circ \text{OoThrowsException}) \]

11. What catches exceptions of this type?

\[ \text{result} \leftarrow \text{ran}(\text{input} \circ \text{OoCatchesException}) \]
12. What references this type by name?

\[ \text{result} \leftarrow \text{ran}(\text{input} \odot \text{OoREFERENCESType}) \]

13. Which of the classes in this package were actually used?

\[ \text{instantiated} \leftarrow \text{OoProvidedTypes} \odot \text{DynWasInstantiated} \]
\[ \text{result} \leftarrow \text{ran}(\text{input} \odot \text{instantiated}) \]

This query assumes the use of type correspondences as defined in Figure 4.5 (p. 52) to map an instance of \text{IType} to instance(s) of TRCClass.

14. Which methods defined by this class were actually used?

\[ \text{used} \leftarrow \text{OoDeclaredMethods} \odot \text{DynWasInvoked} \]
\[ \text{result} \leftarrow \text{ran}(\text{input} \odot \text{used}) \]

This query assumes the use of type correspondences as defined in Figure 4.5 (p. 52) to map an instance of \text{IMethod} to instance(s) of TRCMethod.

### B.3 Intra-Class Queries

15. What fields does this type or method access?

\[ \text{result} \leftarrow \text{ran}(\text{input} \odot \text{OoFieldsUsed}) \]

16. What methods does this type or method call?

\[ \text{result} \leftarrow \text{ran}(\text{input} \odot \text{OoMethodsCalled}) \]

17. What types does this \{ type, method, extension \} reference?

\[ \text{result} \leftarrow \text{ran}(\text{input} \odot \text{OoTypesReferenced}) \]
B.4 Inheritance Queries

18. What interfaces does this type implement?

\[
\text{interfaces} \leftarrow \text{OoAllSupertypes} \circ \text{OoIsInterface} \\
\text{result} \leftarrow \text{ran}(\text{input} \circ \text{interfaces})
\]

19. What are this class’ subclasses?

\[
\text{subclasses} \leftarrow \text{OoIsClass} \circ \text{OoAllSubtypes} \circ \text{OoIsClass} \\
\text{result} \leftarrow \text{ran}(\text{input} \circ \text{subclasses})
\]

20. What classes implement this interface?

\[
\text{implementors} \leftarrow \text{OoIsInterface} \circ \text{OoAllSubtypes} \circ \text{OoIsClass} \\
\text{result} \leftarrow \text{ran}(\text{input} \circ \text{implementors})
\]

21. What interfaces extend this interface?

\[
\text{extensions} \leftarrow \text{OoIsInterface} \circ \text{OoAllSubtypes} \circ \text{OoIsInterface} \\
\text{result} \leftarrow \text{ran}(\text{input} \circ \text{extensions})
\]

22. What are this type’s siblings?

\[
\text{siblings} \leftarrow \text{OoSupertypes} \circ \text{OoAllSubtypes} \\
\text{result} \leftarrow \text{ran}(\text{input} \circ \text{siblings})
\]

B.5 Declaration Queries

Queries 24–27 use an intermediate value \( n \) to store an interim result for further filtering.
23. What fields are declared by this type?

\[ result \leftarrow \text{ran}(\text{input} \circ \text{OoDeclaredFields}) \]

24. What class methods implement this interface method?

\[
\begin{align*}
\text{methods} & \leftarrow \text{OoDeclaringType} \circ \text{OoIsInterface} \circ \text{OoAllSubtypes} \\
& \circ \text{OoIsClass} \circ \text{OoDeclaredMethods} \\
\text{result} & \leftarrow \{ y | (x, y) \in \text{methods} \land \\
& \text{ran}(y \circ \text{OoSINGATURE}) = \text{ran}(\text{input} \circ \text{OoSINGATURE}) \}
\end{align*}
\]

25. What interface methods specify this class method?

\[
\begin{align*}
\text{methods} & \leftarrow \text{OoDeclaringType} \circ \text{OoIsClass} \circ \text{OoAllSupertypes} \\
& \circ \text{OoIsInterface} \circ \text{OoDeclaredMethods} \\
\text{result} & \leftarrow \{ y | (x, y) \in \text{methods} \land \\
& \text{ran}(y \circ \text{OoSINGATURE}) = \text{ran}(\text{input} \circ \text{OoSINGATURE}) \}
\end{align*}
\]

26. What class methods does this method override?

\[
\begin{align*}
\text{methods} & \leftarrow \text{OoDeclaringType} \circ \text{OoIsClass} \circ \text{OoAllSupertypes} \\
& \circ \text{OoIsClass} \circ \text{OoDeclaredMethods} \\
\text{result} & \leftarrow \{ y | (x, y) \in \text{methods} \land \\
& \text{ran}(y \circ \text{OoSINGATURE}) = \text{ran}(\text{input} \circ \text{OoSINGATURE}) \}
\end{align*}
\]

27. What class methods override this method?

\[
\begin{align*}
\text{methods} & \leftarrow \text{OoDeclaringType} \circ \text{OoIsClass} \circ \text{OoAllSubtypes} \\
& \circ \text{OoIsClass} \circ \text{OoDeclaredMethods} \\
\text{result} & \leftarrow \{ y | (x, y) \in \text{methods} \land \\
& \text{ran}(y \circ \text{OoSINGATURE}) = \text{ran}(\text{input} \circ \text{OoSINGATURE}) \}
\end{align*}
\]
28. What extension points or extensions reference this \{ type, file, folder \}?

\[
\text{result} \leftarrow \text{ran}(\text{input} \circ \text{OorReferencesIdentifier})
\]

29. What types is this type adaptable to?

\[
\text{elements} \leftarrow \text{PdeAdaptableTo} \sqcup (\text{OoAllSupertypes} \circ \text{PdeAdaptableTo})
\]

\[
\text{result} \leftarrow \text{ran}(\text{input} \circ \text{elements})
\]

This query extracts the type and declaring plugin from the resulting \|\text{-Plug}\|-in\|\text{-Ele\-ments}.

30. What types could this type have been adapted from?

\[
\text{result} \leftarrow \text{ran}(\text{input} \circ \text{PdeAdaptableFrom})
\]

This query extracts the type and declaring plugin from the resulting \|\text{-Plug}\|-in\|\text{-Ele\-ments}.

31. What extensions are there of this extension point?

\[
\text{result} \leftarrow \text{ran}(\text{input} \circ \text{PdeDeclaredExtensions})
\]

32. What plug-ins depend on this plug-in?

\[
\text{result} \leftarrow \text{ran}(\text{input} \circ \text{PdeDependents})
\]

33. What transactions changed this element?

\[
\text{result} \leftarrow \text{ran}(\text{input} \circ \text{EvoModifications})
\]

B.6 Evolution Queries
34. Who has changed this element, and when?

\[
\text{modifications} \leftarrow \text{ran}(input \triangleleft \text{EvoModifications})
\]
\[
\text{result} \leftarrow \text{modifications} \triangleleft \text{EvoAuthor}
\]

This query results in a relation of transactions to their author.

35. What elements were changed in this transaction?

\[
\text{result} \leftarrow \text{ran}(input \triangleleft \text{EvoElementsCochanged})
\]

36. What files were changed in this transaction?

\[
\text{result} \leftarrow \text{ran}(input \triangleleft \text{EvoFilesCochanged})
\]
Appendix C

Laboratory Experiment: Supporting Materials

This chapter includes the supporting materials from the laboratory experiment described in Section 5.3. Section C.1 features the procedures manual, the task descriptions, and the tool tutorials. Section C.2 features the Tlx questionnaire. Section C.3 includes the distance profiles. These materials and the data from the study are available at [http://www.cs.ubc.ca/~bsd/research/icpc2007/].

C.1 Experiment Instructions and Procedures

We include here the experiment procedures manual (p. 139), the task descriptions (as p. 142 and sr p. 145), and tutorials for Ferret (p. 148), JQuery (p. 151), and Suade (p. 154). Note that at the time of the experiment, Ferret was called Ariadne and Suade was called Suggestions.

The second task, which was to be performed with one of the specialized software exploration tools, also included the following instructions in change request:

You are to perform the software exploration using the provided software exploration tool. We ask that you attempt to use this tool for as much of your exploration as possible: we are interested in what information it cannot provide, and the circumstances. If you cannot find the answer to your query, then by all means use the normal Eclipse facilities.
**Procedures Manual**

**Workspace Set-up**

4 workspaces, each with a copy of modified jEdit-4.1pre6 and (unmodified) JHotDraw-6.0b1, the latest versions of the experiment plugins, the Mylar Monitor is started, and the IBM 1.4.2 JVM installed.

- **plain**: Eclipse R3.1.2 with only monitoring plugins.
  1. The Relevant Elements view should be open.
  2. jEdit debug profile has been run.
- **jquery**: Eclipse R3.1.2 with jQuery and monitoring plugins.
  1. Two working sets should be created, one for jEdit and the other with JHotDraw.
  2. The jQuery view should be opened in the same location as the Package Explorer view, with two tabs, one for JHotDraw, and the other for jEdit.
  3. The Package Explorer should be closed.
  4. The Relevant Elements view should be open.
  5. jEdit debug profile has been run.
- **ariadne**: Eclipse R3.1.2 with Ariadne, Ariadne JDT, and monitoring plugins.
  1. The Ariadne view should be open, and the static JDT option preselected.
  2. The Relevant Elements view should be open.
  3. jEdit debug profile has been run.
- **suggestor**: Eclipse R3.1.2 with Concern Mapper, Suggestions, and monitoring plugins.
  1. The Concern Mapper and Suggestions views should be open.
  2. The Relevant Elements view should be open.
  3. jEdit debug profile has been run.

Obtained printouts of:

- The Overview of the Experiment
- The Relevant Elements documentation
- The Interview Questions, 2 copies of the Consent Form, and the NASA TLX.
- The jQuery, Ariadne, Suggestor tutorials
- The task descriptions: A-T1, A-T2, B-T1, B-T2.

**Prior to Participant’s Arrival**

1. Assign id for incoming participant, and note date and time.
2. Assign and strike configuration from configuration list.
3. Copy and configure workspace directories for respective tasks by using the c:\expt\src\configN script
4. Unzip appropriate workspaces and start first task workspace.
5. Restore window focus from follows-mouse.
6. Quit Gaim and Workrave
7. Layout 2 copies of consent form and appropriate reference card.
8. Start the first Eclipse workspace (c:\expt\task-plain\run.bat). Verify that the monitor is started and is not obfuscating references.
Administering the Experiment

Introduction (5 minutes)

1. Welcome subjects and walk through clauses of consent form: emphasizing anonymity of traces to be made public. Have them sign two copies, one for their references.
2. Have them read the Overview of Experiment page. Summarize:
   1. They will be planning two program modification tasks to jEdit. Ask if they have ever viewed the source code for jEdit.
   2. Emphasize that they are not to actually perform the changes, but rather should identify the particular classes and methods to be used, and describe any new classes required. These instructions should be appropriate for giving to a first-year coop student to rapidly turn to working code.
3. They are to record the plan using the Relevant Elements tool. Have them do the Relevant Elements tutorial. Remind them that for existing classes/methods/fields, using Control-F9; for new elements, use the new element button.
4. The investigator should situate themselves out of the participant’s line-of-sight.

Task 1 (50 minutes + 4 minutes)

1. [10 minutes] Ask to read the first task description, to be performed using Eclipse:
   - Please read the task description and ask any questions.
   - We have allocated 10 minutes.
   - We cannot answer any content questions during the experiment, and only with yes/no answers.
   - Please let me know when you are ready to start the task.
2. [40 minutes] Proceed with the actual task:
   - Verify that the Mylar Monitor is running.
   - They have 40 minutes.
   - Ask if they would like updates on time.
   - They may rearrange windows as they like.
   - **Note start and completion times.**
   - Take field notes.
3. [4 minutes] Ask to explain what they considered.
   - Start audio recording
   - State participant id, date, and time
   - Ask to explain what they considered.
   - Stop audio recording
   - [4 minutes] Administer NASA TLX questionnaire. Note that it is one page.
     - During this time, terminate the first workspace and start the second workspace. Pre-cache information, if possible:
       - For JQuery: ensure two tabs are open on the JHotDraw and jEdit working sets. Warm up the jQuery caches for both JHotDraw and jEdit by right-clicking on respective tabs in the JQuery view’s open tree browser. Ensure rules are present (triangle-menu -> reload). (3 mins approx)
       - For Ariadne: ensure Static Java operation toolbox selected.

**Ask if they would like to take a break.**
Task 2 (60 minutes)

1. [10 minutes] Provide copy of appropriate tool tutorial.
   - Ask to read the tutorial.
   - Show JHotDraw. Explain that JHotDraw is a simple figure editor, somewhat like Adobe Illustrator or Corel Draw, providing tools to draw, edit, and composite figures. Ask them to explain the purpose of org.jhotdraw.standard.NullTool within the JHotDraw system.

2. [10 minutes] Ask to read the first task description, to be performed using the appropriate tool:
   - Please read the task description and ask any questions.
   - We have allocated 10 minutes.
   - We cannot answer any content questions during the experiment, and only with yes/no answers.
   - Please let me know when you are ready to start the task.

3. [40 minutes] Proceed with the actual task:
   - Verify monitor is enabled.
   - Remind them that they have 40 minutes.
   - They may rearrange windows as they like.
   - Please attempt to use the tool as much as possible: we are interested in what information it cannot provide and the circumstances. If you cannot find the answer to your query, then by all means use the normal Eclipse facilities.
   - Provide updates every 10 minutes on time remaining.
   - **Note start and completion times.**
   - Take field notes.


Interview (10 minutes)

1. Turn on audio recording device.
2. State participant id and date and time.
3. Conduct interview
4. Turn off audio recording device.

Conclusion

1. Create directory in results directory with participant identifier.
2. Copy the appropriate mylar-history.xml and rel-elmtd-*.xml files to results directory using the c:\expt\src\store id.
3. If using Suggestor, save concern graph.
4. Clean the directories using c:\expt\src\clean

Updated: 2006-07-11
Task: Autosave (A-T1)

This task involves planning a change to jEdit’s autosave feature. You will need to first familiarize yourself with jEdit and one of its feature: autosave.

Overview of JEdit, Buffers, and Autosave

1. Launch jEdit to check it out. From the Eclipse Run configurations menu, choose jEdit-4-1-pre6. jEdit will appear momentarily. Some warning messages will be displayed in the console: this is normal. Do not close jEdit.
2. In JEdit, an opened text file is called aBuffer. The following is an extract from the JEdit Manual (section 2.1):

   Several files can be opened and edited at once. Each open file is referred to as a buffer. The combo box above the text area selects the buffer to edit. Different emblems are displayed next to buffer names in the list, depending on the buffer’s state; a red disk is shown for buffers with unsaved changes, a lock is shown for read-only buffers, and a spark is shown for new buffers which don’t yet exist on disk.

3. JEdit has an autosave feature. The following is an extract from the JEdit Manual (section 3.3.1):

   "The autosave feature protects your work from computer crashes and such. Every 30 seconds, all buffers with unsaved changes are written out to their respective file names, enclosed in hash ("#") characters. For example, program.c will be autosaved to #program.c#.”

   JEdit will also generate backup files, which are terminated with a tilde (~) character. These have nothing to do with your task in this study. You can completely ignore them.

   Saving a buffer using one of the commands in the previous section automatically deleted the autosave file, so they will only ever be visible in the unlikely event of a JEdit (or operating system) crash.

   If an autosave file is found while a buffer is being loaded, JEdit will offer to recover the autosaved data.

   The autosave feature can be configured in the Loading and Saving pane of the Utilities>Global Options dialog box;

4. In JEdit, set the autosave frequency to 5 seconds.
5. Open the file C:\temp\Test.txt
6. Add a character to the file and do not save the file.
7. Look in C:\temp. You should see the autosave file.
8. Save the test buffer in JEdit. The autosave file should disappear.
9. Add a character to the test buffer and do not save it. Wait 5 seconds.
10. Kill JEdit using the terminate button on the Eclipse console (the button with the red square).
11. Launch jEdit again. JEdit will attempt to recover the autosave file. Click yes. **ATTENTION:** A bug in the code of JEdit will cause the program to hang if you do not click yes or no in the recovery dialog before the time in the autosave frequency. To avoid this, just click yes or no before the 5 seconds (or whatever) of the autosave frequency are over. If the program hangs, you can kill it using the terminate button on the console. You do not have to worry about this bug for the study. Your modification will pass the test cases even in the presence of this bug.

From a user perspective, that's all there is to the autosave feature. You can close JEdit now.

**Change Request**

You are to create a a plan for performing the following task. This involves identifying the relevant program elements that need to be changed and how they should be changed, and those program elements that are necessary to understand for the change. **NOTE:** You are not actually required perform the changes. Rather you should identify the particular classes and methods to be used describe any new classes required. These instructions should be appropriate for giving to a first-year coop student to rapidly turn to working code.

**Change Task:** Modify the application so that the users can explicitly disable the autosave feature. The modified version should meet the following requirements:

1. jEdit shall have a checkbox labeled "Enable Autosave" above the autosave frequency field in the Loading and Saving pane of the global options. This checkbox shall control whether the autosave feature is enabled or not.
2. The state of the autosave feature should persist between different executions of the tool.
3. When the autosave feature is disabled, all autosave backup files for existing buffers shall be immediately deleted from disk.
4. When the autosave feature is enabled, all dirty buffers should be saved within the specified autosave frequency.
5. When the autosave feature is disabled, the tool should never attempt to recover from an autosave backup, if for some reason an autosave backup is present. In this case the autosave backup should be left as is.

**Expert Knowledge**

You are given the following expert knowledge about the source code:

A checkbox should be added to org.jgit.spjedit.options.LoadSaveOptionPane to enable/disable the autosave. The autosave timer is in org.jgit.spjedit.Autosave.

**Test Cases**

To help understand the possible interactions, consider the following tests:

1. **With autosave enabled, modify a buffer.** The buffer should be autosaved within the specified time.
2. **Save the buffer.** The autosave file should be deleted.
3. **Modify the buffer.** The buffer should be autosaved within the specified time. **Disable the**
autosave feature. The autosave file should be deleted.
4. Enable the autosave feature. The buffer should be autosaved within the specified time.
5. Disable the autosave feature. The autosave file should be deleted. Modify the buffer. No autosave file should appear.
7. Disable autosave. Close JEdit. Copy and rename the test file so it looks like an autosave file. Launch JEdit. No recovery should be attempted. Autosave should be disabled.

Task

1. You must make no change to the source code. You are not allowed to perform temporary changes, or try out different alternatives.
2. Do not use the debugger.

Please notify the investigator when you are ready to commence.
Task: Reload Settings on Change (A-T2)

This task involves planning a change to jEdit’s edit modes and macros features. You will need to first familiarize yourself with jEdit and the edit modes and macros.

Overview of jEdit, Edit Modes, and Macros

1. Launch jEdit to check it out. From the Eclipse Run configurations menu, choose jEdit-4.1-pre6. JEdit will appear momentarily. Some warning messages will be displayed in the console: this is normal. Do not close JEdit.

2. In jEdit, an opened text file is called a Buffer. The following is an extract from the JEdit Manual (section 2.1):

   Several files can be opened and edited at once. Each open file is referred to as a buffer.

   JEdit supports context-specific behaviour and actions through the use of edit modes and macros. From the jEdit user manual:

   An edit mode specifies syntax highlighting rules, auto indent behavior, and various other customizations for editing a certain file type.

   Macros in jEdit are short scripts written in a scripting language called BeanShell. They provide an easy way to automate repetitive keyboard and menu procedures, as well as access to the objects and methods created by jEdit. Macros also provide a powerful facility for customizing jEdit and automating complex text processing and programming tasks.

3. The appropriate edit mode is determined when a file is opened, and may be explicitly set as well. Choose File -> Open and open the file modes/xml.xml. Notice how different elements of the XML file are highlighted in different colours and fonts. All edit modes are described using XML: this particular file describes the edit mode for XML files.

4. Choose File -> Open and look at the contents of the jEdit-4.1-pre6 directory. Note the presence the directory named modes: this directory contains file definitions for the edit modes. The jEdit user manual notes:

   jEdit looks for edit modes in two locations; the modes subdirectory of the jEdit settings directory, and the modes subdirectory of the jEdit install directory. The location of the settings directory is system-specific.

   You should also notice a file named catalog. The jEdit user manual notes:

   Each mode directory contains a catalog file. All edit modes contained in that directory must be listed in the catalog, otherwise they will not be available to jEdit.

5. Now choose the Macros menu. Explore the various submenus and note their arrangement.
From the jEdit user manual:

6. Choose File -> Open and look at the contents of the jEdit-4.1pre6 directory. Using the file chooser dialog, look at the directory structure of the macros directory; note its correspondance with the contents of the Macros menu. From the jEdit user manual:

Macros are defined in files written in a language called BeanShell. [...]

Macros are stored in two places: the macros subdirectory of the jEdit home directory, and the macros subdirectory of the user-specific settings directory.

There is no central description file for macros analogous to the edit modes' catalog file.

When jEdit first loads, it scans the designated macro directories and assembles a listing of individual macros in the Macros menu. When scanning the names, jEdit will delete underscore characters and the .bsh extension for menu labels, so that List.Useful.Information.bsh, for example, will be displayed in the Macros menu as List Useful Information.

Change Request

You are to create a plan for performing the following task. This involves identifying the relevant program elements that need to be changed and how they should be changed, and those program elements that are necessary to understand for the change. NOTE: You are not actually required perform the changes. Rather you should identify the particular classes and methods to be used describe any new classes required. These instructions should be appropriate for giving to a first-year coop student to rapidly turn to working code.

You are to perform the software exploration using the provided software exploration tool. We ask that you attempt to use this tool for as much of your exploration as possible: we are interested in what information it cannot provide, and the circumstances. If you cannot find the answer to your query, then by all means use the normal Eclipse facilities.

Change Task: jEdit may be used for editing the macro and edit mode definitions. Your task is to add support
to dynamically reload any edit modes or update its macro listings should they change. The modified version should meet the following requirements:

1. Upon modifying an edit mode, the edit mode definition should be reloaded and applied to all current buffers.
2. Upon modifying a BeanShell macro, the Macros menu contents should be refreshed and updated.

**Expert Knowledge**

You are given the following expert knowledge about the source code:

jEdit's internals provides support for an event broadcast system, including file system changes, in org.gjt.sp.jedit.EditBus.

**Task**

1. You must make no change to the source code. You are not allowed to perform temporary changes, or try out different alternatives.
2. Do not use the debugger.

Please notify the investigator when you are ready to commence.
Ariadne Tutorial

For this part of the study, you will again explore the jEdit source code to describe the changes necessary to successfully resolve a change task. For this exploration you will use a software exploration tool called Ariadne. Feel free to ask the investigator any questions during this tutorial.

Overview of Ariadne

Ariadne is a software exploration tool that automatically issues queries on a developer’s behalf. As each program element is examined, Ariadne displays the results from a set of conceptual queries about that element. These conceptual queries include queries such as:

- for a type:
  - What methods return objects of this type?
  - What methods instantiate objects of this type?
  - What methods reference objects of this type as an argument?
  - What classes implement this interface?
  - What interfaces does this type implement?
- for a method:
  - What methods or initializers reference this method?
  - What interfaces define this method? (specification)
  - What classes implement this interface method?
  - What are the other alternative implementations of this method in the this hierarchy?
- for a field:
  - Where is the contents of a field obtains? (getters)
  - Where is the contents of a field set? (setters)

Basic Use

Ariadne is automatic, meaning that it responds to your interactions with the Eclipse environment such as selecting an item from the Outline view. The queries with results are listed in the Dossier view using an Eclipse tree view.

The queries are indicated with the following icon: Q. Queries in progress are indicated with a small hourglass: Q.

The Dossier

The Dossier is made up of five columns:

- Description:
  - A description of the element.
- Category:
  - Queries are grouped by category (discussed below).
- #E:
  - The total number of elements for either the query or the current group.
- Clusterings:
  - The currently selected clustering, or the number of possible clusterings (discussed below).
The number of groups in the currently selected clustering (discussed below).

**Query Categories**

Queries are sorted by their category. There are currently four query categories:

- **declarations:** Queries about relations involving the declarations of elements.
- **inter-class:** Queries about relations between different classes.
- **intra-class:** Queries about relations within a class.
- **hierarchical:** Queries about relations involving type inheritance.

**Clusterings**

The results of queries may be clustered by their different attributes. For example: the references to a method, which are generally themselves methods, may be clustered into different groups identified by attributes such as their access protection (public, private, package, or default), their containing type, their package, or boolean attributes such as static vs. non-static. Some clusterings may have results belong to multiple groups, such as might arise for a clustering by argument types: the method `Arrays.asList(Collection, Object[])` would added to groups for `Collection` and `Object`.

**The Dossier Toolbar**

The Dossier view features four command buttons.

- Collapse all branches of the tree.
- Do not cause the dossier to change with new selections. Queries may still be made explicitly using the keyboard shortcut F9.
- Re-open original query element.
- Select from previously queried-for elements.

**Hands On**

Try Ariadne out!

- Bring up JHotDraw’s org.jhotdraw.framework.Figure interface in the editor using Open Type (Control-Shift-T).

- If the Dossier is not already present, push the F9 key. The Dossier view should pop up and be populated with a set of queries. When the queries are finished, which should take a few seconds, you should see a view like this:
Try expanding some of the tree elements, such as for the query superinterfaces, showing the super interfaces specified by Figure. Right-click on the first result for this query, Cloneable: the normal Java operations are available from this elements.

The query implementors has many results (44), however Ariadne was able to find 3 different means of grouping these results. To access these groupings, right click on the element, and choose the Cluster results by... menu.

Choose to cluster by the enclosing package: to cluster the results by their package.

Now click on the re-open original query element button. This causes the original queried-upon element Figure to be re-opened.

Try selecting the Figure.decompose() method from the Outline view, and notice the Dossier's contents have changed with the query results about the new element.

You may wish to try some of the other toolbar buttons.

When you are ready to proceed, please notify the investigator.

last updated: 2006/06/27
JQuery Tutorial

JQuery is a flexible, query-based source code browser. Having created a working set on the elements of interest, a JQuery user can issue a variety of pre-written queries whose results are displayed as a tree. Individual elements in the tree can then be queried individually in the same fashion.

Getting Start

This section will get you acquainted with the JQuery user interface and the more common features of JQuery by going through a simple example. The code base used for this example and other examples is JHotDraw, a Java GUI framework for structured graphics.

Select the JHotDraw tab, if it is not already selected.

We are now ready to run our first query. To do this, we:

1. Right click inside the tree subtab which brings up a context-sensitive menu. The contents of the menu depend on what type of element we click on.

2. Select **Available Top Level Queries -> Package Browser (Java files)** to run a query that will show us all of the packages in the working set and all of the classes and interfaces in those packages.

3. The query result shows up in the tree subtab.
We can now navigate the results of the query by expanding tree nodes until we get to something that interests us.

To conclude this example, we will perform two sub-queries on the `org.jhotdraw.framework.Figure` interface.

1. Navigate into the `org.jhotdraw.framework` package from the main Package Browser query.

2. Right click on the `Figure` interface to bring up a context-sensitive menu.

3. Select **Members -> Methods**

4. The results of the sub-query show up as a child node of `Figure`. We can navigate these results the same as the top-level query results. If we wanted to, we could keep running subqueries until we found what we were looking for (e.g. all of the calls to the `addToContainer` method).
5. JQUERY supports more queries than just on the structural information in Java code. Right click again on the Figure interface and select the Usage -> Argument. The results of this sub-query show those methods with an argument of type Figure.

You may wish to try some of the other queries and sub-queries.

When you are ready to proceed, please notify the investigator.

Last Updated: 2006/07/11
Suggestions Tutorial

Overview of Suggestions

The Suggestion plugin is a software exploration tool that finds code elements that are relevant to the task at hand. By analysing the contents of a concern (a set of program elements pertaining to a particular software behaviour) created with the ConcernMapper plugin (see below), the Suggestion plugin finds elements that are structurally related to those in the specific concern and ranks them according to their relevance.

The relations analyzed to generate suggestions are:

- for each method in the specified concern: which methods call it, which methods it calls and which fields it accesses;
- for each field in the concern: which methods access it.

Basic Use

Creating a Concern Model

Using the Suggestion plugin requires first constructing a concern model using the ConcernMapper view. The contents of the ConcernMapper view represent a concern model, which is made up of multiple concerns. The following view shows a single concern, which has no elements assigned:

New concerns are added using the new-concern button; you will be prompted for a name for the concern. Elements are added to the concern by dragging and dropping fields or methods. It is generally helpful to give a concern a useful name, usually describing the behaviour being investigated. Concerns can be renamed by right-clicking on the concern.

Generating Suggestions

Having built part of a concern using the ConcernMapper, the Suggestion plugin may be used to generate additional suggestions. This done by dragging a concern and dropping it on the Suggestions view. Alternatively, clicking the Suggest button will generate suggestions for the concern that is currently selected in the ConcernMapper view.

Once the suggestions are generated, it is possible to sort them by name, degree, or related element by
selecting the appropriate column in the Suggestions view. Elements whose degree is below a specified level can be filtered out using the Filter button. Elements that are deemed relevant can then be moved into the appropriate concern in the concern model by obtaining the context menu on a suggestion and selecting Move to concern model.

The Suggestions View

The Suggestions view displays the suggested elements. The view has three columns:

- **Element**: The name of the element that is suggested.
- **Reason**: The reasons for which this element is suggested.
- **Degree**: The degree to which the suggested element is deemed relevant by the tool.

The Suggestions Toolbar

The Suggestions view has two buttons.

- **i**: Generate suggestions for selected concern.
- **x**: Filter out elements whose degree is below a certain threshold.

The Suggestions Preferences

To access the preferences for the Suggestion plugin, select Preferences from the Window menu on the eclipse menu bar. In the left pane expand the ConcernMapper node and select Suggestions.

The Suggestions preference page allows to set the filter on or off and to set its threshold degree.

Try it!

1. Create a new concern called decomposition.
2. Find org.jhotdraw.standard.AbstractFigure and drag its decompose() method to the decomposition concern.
3. Cause the Suggestion plugin to make suggestions. Examine the list of resulting elements and drag any applicable ones onto the concern.
4. Repeat!
C.2 The Task Load Index (TLX) Questionnaire

The Task Load Index (TLX) [42] is a well-validated and easy-to-administer questionnaire for assessing the subjective mental workload experienced by a subject performing a task. The TLX workload questionnaire (shown on p. 157) combines a subject's report of six different factors (the mental, physical, and temporal demands; the effort required; the subject's view of his performance; and the frustration experienced) to provide a subjective assessment of workload. To enable comparison between subjects, a score is normalized by taking its difference from the score for a baseline task performed without the treatment of interest. The scores are weighted using the results of a pair-wise comparison performed by the subject to produce a ranking of factors by difficulty (shown on p. 158).
Workload Questionnaire (Task 1)

We’d like to ask you about how difficult you perceived the task you just completed to be. Please answer the questions below by marking an X in the appropriate box in the provided rating scales.

Mental Demand: How much mental and perceptual activity was required (e.g., thinking, deciding, calculating, remembering, looking, searching, etc.)? Was the task easy or demanding, simple or complex, exacting or forgiving?

<table>
<thead>
<tr>
<th>Low</th>
<th>High</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Physical Demand: How much physical activity was required (e.g., pushing, pulling, turning, controlling, activating, etc.)? Was the task easy or demanding, slow or brisk, slack or strenuous, restful or laborious?

<table>
<thead>
<tr>
<th>Low</th>
<th>High</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Temporal Demand: How much time pressure did you feel due to the rate or pace at which the tasks or task elements occurred? Was the pace slow and leisurely or rapid and frantic?

<table>
<thead>
<tr>
<th>Low</th>
<th>High</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Effort: How hard did you have to work (mentally and physically) to accomplish your level of performance?

<table>
<thead>
<tr>
<th>Low</th>
<th>High</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Performance: How successful do you think you were in accomplishing the goals of the task set by the experimenter (or yourself)? How satisfied were you with your performance in accomplishing these goals?

<table>
<thead>
<tr>
<th>Good</th>
<th>Poor</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Frustration: How insecure, discouraged, irritated, stressed and annoyed versus secure, gratified, content, relaxed and complacent did you feel during the task?

<table>
<thead>
<tr>
<th>Low</th>
<th>High</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Participant ID:
Importance of Different Workload Categories

Please select the member of each pair that had the more significant effect on the overall workload for all the tasks performed in this study. Descriptions of the categories are found at the bottom of the page.

<table>
<thead>
<tr>
<th>Physical Demand</th>
<th>or</th>
<th>Mental Demand</th>
</tr>
</thead>
<tbody>
<tr>
<td>Temporal Demand</td>
<td>or</td>
<td>Mental Demand</td>
</tr>
<tr>
<td>Performance</td>
<td>or</td>
<td>Mental Demand</td>
</tr>
<tr>
<td>Frustration</td>
<td>or</td>
<td>Mental Demand</td>
</tr>
<tr>
<td>Effort</td>
<td>or</td>
<td>Mental Demand</td>
</tr>
<tr>
<td>Temporal Demand</td>
<td>or</td>
<td>Physical Demand</td>
</tr>
<tr>
<td>Performance</td>
<td>or</td>
<td>Physical Demand</td>
</tr>
<tr>
<td>Frustration</td>
<td>or</td>
<td>Physical Demand</td>
</tr>
<tr>
<td>Effort</td>
<td>or</td>
<td>Physical Demand</td>
</tr>
<tr>
<td>Temporal Demand</td>
<td>or</td>
<td>Performance</td>
</tr>
<tr>
<td>Temporal Demand</td>
<td>or</td>
<td>Frustration</td>
</tr>
<tr>
<td>Temporal Demand</td>
<td>or</td>
<td>Effort</td>
</tr>
<tr>
<td>Performance</td>
<td>or</td>
<td>Frustration</td>
</tr>
<tr>
<td>Performance</td>
<td>or</td>
<td>Effort</td>
</tr>
<tr>
<td>Effort</td>
<td>or</td>
<td>Frustration</td>
</tr>
</tbody>
</table>

Category definitions

- **Mental Demand**: How much mental and perceptual activity was required (such as thinking, deciding, calculating, remembering, looking, or searching)? Was the task easy or demanding, simple or complex, exacting or forgiving?
- **Physical Demand**: How much physical activity was required (such as pushing, pulling, turning, controlling, or activating)? Was the task easy or demanding, slow or brisk, slack or strenuous, restful or laborious?
- **Temporal Demand**: How much time pressure did you feel due to the rate or pace at which the tasks or task elements occurred? Was the pace slow and leisurely or rapid and frantic?
- **Effort**: How hard did you have to work (mentally and physically) to accomplish your level of performance?
- **Performance**: How successful do you think you were in accomplishing the goals of the task? How satisfied were you with your performance in accomplishing these goals?
- **Frustration Level**: How insecure, discouraged, irritated, stressed and annoyed versus secure, gratified, content, relaxed, and compliant did you feel during the task?

Participant ID:
C.3 Distance Profiles

This section presents the full set of distance profiles from the experiment described in Section 5.3. The profiles are sorted by the second specialized exploration tool used and then by task order.

C.3.1 Jquery

04 (T1 SR Eclipse) 17 (T1 SR Eclipse) 07 (T1 AS Eclipse) 13 (T1 AS Eclipse) 01 (T1 AS Eclipse)

C.3.2 Ferret

04 (T2 SR JQuery) 17 (T2 AS JQuery) 11 (T2 AS Ferret)
Appendix D

UBC Research Ethics Board
Certificates of Approval


B06–0201: Comparing impact of software development tools on developer exploration patterns (p. 163).

H06–80201: Deeper assessment of usefulness of Ferret (Amendment of B06–0201) (p. 164).
Certificate of Approval

Principal Investigator: Murphy, G.C.
Department: Computer Science
Number: B04-0130

Institution(s) where research will be carried out:

Co-Investigators:
Alvis, Brian, Computer Science

Sponsoring Agencies:
Natural Science Engineering Research Council

Title:
Characterization of Disorientation Experienced by Software Developers

Approval Date: Feb. 10, 2004
Term (in years): 1
Documents included in this approval:

Certification:

The protocol describing the above-named project has been reviewed by the Committee and the experimental procedures were found to be acceptable on ethical grounds for research involving human subjects.

[Signature elided]

Approval of the Behavioural Research Ethics Board by one of the following:
Dr. James Frankish, Chair,
Dr. Cay Holbrook, Associate Chair,
Dr. Susan Rowley, Associate Chair

This Certificate of Approval is valid for the above term provided there is no change in the experimental procedures.
Certificate of Approval

PRINCIPAL INVESTIGATOR
Murphy, G.C.

DEPARTMENT
Computer Science

INSTITUTION(S) WHERE RESEARCH WILL BE CARRIED OUT
UBC Campus

CO-INVESTIGATORS
De Alwis, Brian Simon, Computer Science

SPONSORING AGENCIES
Natural Science Engineering Research Council

TITLE:
Comparing Impact of Software Development Tools on Developer Exploration Patterns

APPROVAL DATE
MAY 22, 2006

TERM (YEARS):
1

DOCUMENTS INCLUDED IN THIS APPROVAL:
Feb. 23, 2006, Questionnaires / Feb. 24, 2006, Advertisement / Contact letter / Consent form

CERTIFICATION
The application for ethical review of the above-named project has been reviewed and the procedures were found to be acceptable on ethical grounds for research involving human subjects.

[Signature elided]

Approved on behalf of the Behavioural Research Ethics Board by one of the following:
Dr. Peter Suedfeld, Chair,
Dr. Susan Rowley, Associate Chair
Dr. Jim Rupert, Associate Chair
Dr. Arminee Kazanjian, Associate Chair

This Certificate of Approval is valid for the above term provided there is no change in the experimental procedures.
CERTIFICATE OF APPROVAL - MINIMAL RISK AMENDMENT

PRINCIPAL INVESTIGATOR: Sail C. Murphy
DEPARTMENT: UBC Science/Computer Science
UBC BREC NUMBER: H06-80201

INSTITUTION(S) WHERE RESEARCH WILL BE CARRIED OUT:

<table>
<thead>
<tr>
<th>Institution</th>
<th>Site</th>
</tr>
</thead>
<tbody>
<tr>
<td>UBC</td>
<td>Point Grey site</td>
</tr>
</tbody>
</table>

Other locations where the research will be conducted:
This study will be administered on-site at up to three computer businesses in the lower mainland, Victoria, and Ottawa. Data gathering will happen on the participants’ computers, and will subsequently be transferred to a server at UBC.

CO-INVESTIGATOR(S):
Brian de Alwis

SPONSORING AGENCIES:
Natural Sciences and Engineering Research Council of Canada (NSERC)

PROJECT TITLE:
Comparing impact of Software Development Tools on Developer Exploration Patterns

Expiry Date - Approval of an amendment does not change the expiry date on the current UBC BREC approval of this study. An application for renewal is required on or before: February 15, 2008

AMENDMENT(S):

<table>
<thead>
<tr>
<th>Document Name</th>
<th>Version</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>Consent Forms:</td>
<td>N/A</td>
<td>June 19, 2007</td>
</tr>
<tr>
<td>Main study consent</td>
<td>N/A</td>
<td>June 19, 2007</td>
</tr>
<tr>
<td>Advertisement</td>
<td>N/A</td>
<td>June 19, 2007</td>
</tr>
<tr>
<td>Questionnaire, Questionnaire Cover Letter, Tests:</td>
<td>N/A</td>
<td>June 19, 2007</td>
</tr>
<tr>
<td>Interview script</td>
<td>N/A</td>
<td>June 19, 2007</td>
</tr>
<tr>
<td>Letter of Initial Contact:</td>
<td>N/A</td>
<td>June 19, 2007</td>
</tr>
<tr>
<td>letter of initial contact</td>
<td>N/A</td>
<td>June 19, 2007</td>
</tr>
<tr>
<td>letter of invitation to companies</td>
<td>N/A</td>
<td>July 16, 2007</td>
</tr>
</tbody>
</table>

The amendment(s) and the document(s) listed above have been reviewed and the procedures were found to be acceptable on ethical grounds for research involving human subjects.

Approval is issued on behalf of the Behavioural Research Ethics Board and signed electronically by one of the following:

Dr. Peter Suedfeld, Chair
Dr. Jim Rupert, Associate Chair
Dr. Arminee Kazanjian, Associate Chair
Dr. M. Judith Lynam, Associate Chair
Dr. Laune Ford, Associate Chair